

R Programming for Research

Colorado State University, ERHS 535

Brooke Anderson, Rachel Severson, and Nicholas Good

2023-08-17

Contents

Online course book, ERHS 535	9
Course information	1
0.1 Course overview	1
0.2 Course syllabus	1
0.3 Time and place	1
0.4 Detailed schedule	2
0.5 Final group project	3
0.6 Course set-up	5
0.7 Coursebook	5
0.8 In-course Exercise Chapter 0	6
I Part I: Preliminaries	9
1 R Preliminaries	11
1.1 Objectives	11
1.2 R and R Studio	12
1.3 Communicating with R	16
1.4 Functions	19
1.5 Objects and assignment	22
1.6 More on communicating with R	28
1.7 R scripts	29
1.8 The “package” system	33

1.9 R's most basic object types	41
1.10 In-course Exercise Chapter 1	54
II Part II: Basics	75
2 Entering and cleaning data #1	77
2.1 Objectives	77
2.2 Overview	78
2.3 Reading data into R	78
2.4 Directories and pathnames	83
2.5 Data cleaning	92
2.6 Piping	100
2.7 In-course Exercise Chapter 2	102
3 Exploring data #1	115
3.1 Objectives	115
3.2 Simple statistics functions	116
3.3 Factor vectors	120
3.4 Data from a package	120
3.5 Dates in R	122
3.6 Logical vectors	126
3.7 Plots to explore data	129
3.8 In-course Exercise Chapter 3	144
4 Reporting data results #1	171
4.1 Guidelines for good plots	171
4.2 High data density	172
4.3 Meaningful labels	174
4.4 References	176
4.5 Highlighting	179
4.6 Order	182
4.7 Small multiples	183

CONTENTS	5
4.8 Advanced customization	186
4.9 To find out more	192
4.10 In-course exercise–Chapter 4	192
5 Reproducible research #1	217
5.1 What is reproducible research?	217
5.2 Markdown	218
5.3 Literate programming in R	220
5.4 Style guidelines	224
5.5 More with knitr	227
5.6 In-course exercise Chapter 5	231
III Part III: Intermediate	235
6 Entering and cleaning data #2	237
6.1 Tidy data	237
6.2 Joining datasets	244
6.3 Longer data	246
6.4 Working with factors	250
6.5 String operations and regular expressions	252
6.6 Tidy select	260
6.7 In-course exercise Chapter 6	261
7 Exploring data #2	283
7.1 Simple statistical tests in R	283
7.2 Matrices	287
7.3 Lists	289
7.4 Apply a test multiple times	294
7.5 Regression models	299
7.6 Handling model objects	307
7.7 Functions	309
7.8 In-course exercise for Chapter 7	316

8 Reporting results #2	359
8.1 Course lectures	359
8.2 Example data	359
8.3 <code>ggplot2</code> extras and extensions	360
8.4 Simple features	368
8.5 In-course exercise Chapter 8	387
9 Reproducible research #2	397
9.1 Templates	397
9.2 R Projects	399
9.3 git	400
9.4 In-course exercise Chapter 9	408
IV Part IV: Advanced	411
10 Entering and cleaning data #3	413
10.1 Pulling online data	413
10.2 Example R API wrapper packages	417
10.3 <code>tigris</code> package	418
10.4 <code>countyweather</code>	421
10.5 Cleaning very messy data	423
10.6 In-course exercise Chapter 10	430
11 Exploring data #3	437
11.1 Simulations	437
11.2 Other computationally-intensive approaches	450
12 Reporting data results #3	453
12.1 Shiny web apps	453
12.2 <code>htmlWidgets</code>	469

13 Reproducible research #3	481
13.1 Overview of R packages	481
13.2 Basic example package: <i>weathermetrics</i>	487
13.3 Elements of an R package	490
13.4 Creating an R package	498
A Appendix A: Vocabulary	505
A.1 Quiz 1—R Preliminaries	505
A.2 Quiz 2—Entering / cleaning data #1	506
A.3 Quiz 3	507
A.4 Quiz 4	508
A.5 Quiz 5	508
A.6 Quiz 6	509
A.7 Quiz 7	509
A.8 Quiz 8	509
A.9 Quiz 9	510
B Appendix B: Homework	511
B.1 Homework #1	511
B.2 Homework #2	514
B.3 Homework #3	516
B.4 Homework #4	518
B.5 Homework #5	521
B.6 Homework #6	522

Online course book, ERHS 535

This is the online book for Colorado State University's *R Programming for Research* courses (ERHS 535, ERHS 581A3, and ERHS 581A4).

This book includes course information, course notes, links to download pdfs of lecture slides, in-course exercises, homework assignments, and vocabulary lists for quizzes for this course.

““Give someone a program, you frustrate them for a day; teach them how to program, you frustrate them for a lifetime.”—David Leinweber”

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Course information

Download a pdf of the lecture slides covering this topic.

0.1 Course overview

This book is for Colorado State University's **R Programming for Research** courses (ERHS 535, ERHS 581A3, and ERHS 581A4). The courses offer in-depth instruction on data collection, data management, programming, and visualization, using data examples relevant to data-intensive research.

0.2 Course syllabus

The syllabus for ERHS 535 is available for download by [clicking here](#). The syllabus for ERHS 581A3 is available for download by [clicking here](#).

0.3 Time and place

Students for ERHS 535, ERHS 581A3, and ERHS 581A4 will meet together. Students in ERHS 535 will meet for the entire semester, completing a three-credit course. Students in ERHS 581A3 will meet for the first five weeks of the semester, completing a one-credit course. Students in ERHS 581A4 will meet from the sixth week to the final week of the semester, completing a two-credit course.

The course meets in a mixed format, with about two hours of asynchronous video lectures per week and about two hours of in-person meeting time. Course lectures are provided through YouTube videos. These are embedded in the online course book, and the book also provides links to playlists for each chapter of all the videos for that chapter.

Exceptions to the regular live meeting times are:

- There will be no meeting on Labor Day.
- There are no course meetings the week of Thanksgiving.

0.4 Detailed schedule

Here is a more detailed view of the schedule for this course for Fall 2023:

Week	Class date	Videos	In-class exercises	Grades
1	Aug. 21	None	Ch 0: 0.7.1–0.7.3	
1	Aug. 23	Ch 1, Videos 1–9	Ch 1: 1.10.3–1.10.10	
2	Aug. 28	Ch 1, Video 10; Ch 2, Videos 1–4	Ch 2: 2.7.1–2.7.3	
2	Aug. 30	Ch 2, Videos 5–10	Ch 2: 2.7.4–2.7.6	Q1
3	Sept. 6	Ch 3, Videos 1–5	Ch 3: 3.8.1–3.8.3	Q1
4	Sept. 11	Ch 3, Videos 6–10	Ch 3: 3.8.4–3.8.5	HW
4	Sept. 13	Ch 4, Videos 1–4	Ch 3: 3.8.6; Ch 4: 4.10.1	Q1
5	Sept. 18	Ch 4, Videos 5–7	Ch 4: 4.10.2–4.10.3	
5	Sept. 20	Ch 5, Videos 1–4	Ch 5: 5.6.1–5.6.4	Q1
6	Sept. 25	Ch 5, Videos 5–6, Ch 6, Videos 1–3	Ch 6: 6.7.1–6.7.2	
6	Sept. 27	Ch 6, Videos 4–8	Ch 6: 6.7.2–6.7.3	Q1
7	Oct. 2	Ch 6, Videos 9–12	Ch 6: 6.7.4–6.7.5	
7	Oct. 4	No videos	Work on Q4 from HW 3	Q1
8	Oct. 9	Ch 7, Videos 1–3	Ch 7: 7.8.2–7.8.3	
8	Oct. 11	Ch 7, Videos 4–5	Ch 7: 7.8.3	Q1
9	Oct. 16	Ch 7, Videos 6 (Optional: Ch 7, Videos 7–8)	Ch 7: 7.8.4–7.8.5	
9	Oct. 18	Ch 7, Videos 9–11	Ch 7: 7.8.6	Q1
10	Oct. 23	Ch 7, Videos 12–14	Ch 7: 7.8.7	
10	Oct. 25	Ch 8, Video 1	Ch 8: 8.5.1–8.5.2	Q1
11	Oct. 30	Ch 8, Videos 2–3	Ch 8: 8.5.3	
11	Nov. 1	No videos	Ch 8: 8.5.4	HW
12	Nov. 6	Ch 9, Videos 1–4	Guest exercise	
12	Nov. 8	Ch 9, Videos 5–7	Guest exercise	
13	Nov. 13	Ch 9, Videos 8–9	Ch 9: 9.4.1–9.4.3	
13	Nov. 15	Ch 12, Videos 1–3	Group work on final project	HW
14	Nov. 27	Ch 12, Videos 4–5	Group work on final project	
14	Nov. 29	Ch 10, Videos 1–3	Group work on final project	
15	Dec. 4	Ch 10, Videos 4–5	Group work on final project	
15	Dec. 6	No videos	Group work on final project	HW

Students in ERHS 581A3 will be in weeks 1–5 of this schedule. Students in ERHS 581A4 will be in weeks 6–16 of this schedule.

0.5 Final group project

You will do the final group project in groups of 3–4. The final product will be a flexdashboard (html created from RMarkdown file) that presents interesting facets of a dataset to viewers and allows them to interact in at least some of the panes. There is more information about flexdashboards available at <https://rmarkdown.rstudio.com/flexdashboard/examples.html>.

Your group’s dashboard should include a text block somewhere on the dashboard that explains the data (including the data source) as well as what the dashboard can be used to explore and how to navigate it (for example, see the left-most pane in this dashboard: https://walkerke.shinyapps.io/neighborhood_diversity/). Besides this text block, your dashboard should have at least three panes with output (figures or tables). At least one should be interactive (using htmlwidgets). You may include everything in a single page (like <https://beta.rstudioconnect.com/jjallaire/htmlwidgets-waste-sites/htmlwidgets-waste-sites.html>) or you can spread results and text across several tabs of a dashboard (like https://walkerke.shinyapps.io/neighborhood_diversity/#explore-metros and <https://beta.rstudioconnect.com/jjallaire/htmlwidgets-showcase-storyboard/htmlwidgets-showcase-storyboard.html>).

There is extensive documentation about building flexdashboards here: <https://rmarkdown.rstudio.com/flexdashboard/using.html>. To divide group work, you may find it helpful to have some group members work on designing and setting up the framework for the dashboard, and then other members focus on writing the code for the figures and tables that will go in specific panes of the dashboard. Everything can then be put together by moving the code for those figures and tables into the flexdashboard RMarkdown file.

You will have in-class group work time to work on this during the in-person sessions. This project will also require some work with your group outside of class. I will provide feedback and help during the in-class group work time.

The final group project will be graded with A through F, with the following point values (out of 30 possible):

- **30 points** for an A
- **25 points** for a B
- **20 points** for a C
- **15 points** for a D
- **10 points** for an F

If you turn nothing in, you will get **0 points**.

0.5.0.1 Final presentation

- In total, the group's presentation should last 15 minutes. There will then be 5 minutes for questions.
- Split the presentation up into two parts: (1) an overview of your flexdashboard (about 7 minutes) and (2) a tutorial-style discussion of how you used R to do the project (about 8 minutes).
- The overview of your flexdashboard should:
 - Explain the data sources for your dashboard
 - Explain what a user should be able to explore or understand by using your dashboard
 - Walk viewers through each pane in the flexdashboard and how any interactive elements can be used
- The tutorial part should include the following sections:
 - **Interesting packages / techniques:** Spend a bit more time on any parts that you found particularly interesting or exciting. Were there packages you used that were helpful that we haven't talked about in class? Did you find out how to do anything that you think other students could use in the future? Did you end up writing a lot of functions to use? Did you have an interesting way of sharing code and data among your group members?
 - **Lessons learned:** If you were to do this project again from scratch, what would you do differently? Were there any big wrong turns along the way? Did you find out how to do something late in the project that would have saved you time if you'd started using it earlier?

0.5.0.2 Flexdashboard

The grade for the flexdashboard will be based on the following criteria:

- It should work.
- It should include text that is clearly written, without grammatical errors or typos. Any graphics are easily to interpret and follow some of the principles of good graphics covered in class.
- It includes at least three rendered outputs (e.g., plots, tables). At least one should be interactive.
- It is self-contained—in other words, a user shouldn't need to hear you explain the dashboard to understand it. It should include enough information on the app for a user to figure out how to use the app and interpret the output.

0.6 Course set-up

Please download and install the latest version of R and RStudio (Desktop version, Open Source edition) installed. Both are free for anyone to download.

Students in ERHS 535 and ERHS 581A4 will also need to download and install a version of LaTeX (MikTeX for Windows and MacTeX for Macs). They will also need to download and install git software and create a GitHub account.

Here are useful links for this set-up:

- R: <https://cran.r-project.org>
- RStudio: <https://www.rstudio.com/products/rstudio/#Desktop>
- Install MikTeX: <https://miktex.org/> (only ERHS 535 / 581A4 with Windows)
- Install MacTeX: <http://www.tug.org/mactex/> (only ERHS 535 / 581A4 with Macs)
- Install git: <https://git-scm.com/downloads> (only ERHS 535 / 581A4)
- Sign-up for a GitHub account: <https://github.com> (only ERHS 535 / 581A4)

0.7 Coursebook

This coursebook will serve as the only required textbook for this course. I am still in the process of editing and adding to this book, so content may change somewhat over the semester (particularly for later weeks, which is currently in a rawer draft than the beginning of the book). We typically cover about a chapter of the book each week of the course.

This coursebook includes:

- Embedded video lectures for the course
- Links to the slides presented in video lectures for each topic
- In-course exercises, typically including links to the data used in the exercise
- An appendix with homework assignments
- A list of vocabulary and concepts that should be mastered for each quiz

If you find any typos or bugs, or if you have any suggestions for how the book can be improved, feel free to post it on the book's GitHub Issues page.

This book was developed using Yihui Xie's wonderful bookdown framework. The book is built using code that combines R code, data, and text to create a book for which R code and examples can be re-executed every time the book is re-built, which helps identify bugs and broken code examples quickly. The online book is hosted using GitHub's free GitHub Pages. All material for this book is available and can be explored at the book's GitHub repository.

0.7.1 Other helpful books (not required)

The best book to supplement the coursebook and lectures for this course is R for Data Science, by Garrett Grolemund and Hadley Wickham. The entire book is freely available online through the same format at the coursebook. You can also purchase a paper version of the book (published by O'Reilly) through Amazon, Barnes & Noble, etc., for around \$40. This book is an excellent and up-to-date reference by some of the best R programmers in the world.

There are a number of other useful books available on general R programming, including:

- R for Dummies
- R Cookbook
- R Graphics Cookbook
- Roger Peng's Leanpub books
- Various books on bookdown.org

The R programming language is used extensively within certain fields, including statistics and bioinformatics. If you are using R for a specific type of analysis, you will be able to find many books with advice on using R for both general and specific statistical analysis, including many available in print or online through the CSU library.

0.8 In-course Exercise Chapter 0

Today, we'll practice doing an in-course exercise. The purpose of this in-course exercise is to help you explore the online book and videos for this class to make sure that you are comfortable with navigating those resources. You will work on this for about 20 minutes.

The last prompt requires you to have RStudio installed. If nobody in the group has RStudio installed, you can read through this part without doing the activities.

0.8.1 Navigate the online course book

Find the following in the book:

- The schedule for this course, including the dates that homeworks are due and the dates that quizzes will be given
- Information on the grading policies for quizzes for this course (hint: find the link to download the syllabus)

- When do you have your first quiz? What are the vocabulary terms you will be responsible for in the first quiz?
- For homework:
 - How many homeworks are due in total over the full semester (for those taking ERHS 535)?
 - When are the first and second homework due?
 - Find the assignment information for the first homework in the course book.
 - How will you turn in your first homework?
- Find information on the requirements for “Course set-up” from the chapter on course information. Click on the links for downloading R and RStudio. Make sure everyone in the group sees where they will need to go to download R and RStudio. You will need to do this before the class meeting on Wednesday.

0.8.2 Find the video lectures

- Look through the book to find links to the YouTube video lectures for Chapter 1 (“R Preliminaries”). You should be able to find:
 - Embedded videos throughout the text
 - Links to download the pdfs for each video below each embedded video
 - A link to a YouTube playlist with all the video lectures for the chapter at the beginning of the chapter text.
- Download the pdf copy of the slides for the first video lecture for Chapter 1.
- Watch the very beginning of the first video lecture for Chapter 1.
- You will need to watch the first nine video lectures for Chapter 1 before we meet on Wednesday.
 - What are the titles of these nine videos?
 - What is the title of the only video for Chapter 1 that you are **not** required to watch before Wednesday’s meeting?

0.8.3 Try coding a bit

- Open RStudio
- Open an R script by going to “File” -> “New File” -> “R Script”
- Type the following code in the R script:

```
sample(c("Johnny", "Waylon", "Willie", "Kris"))
```

- Highlight the code you just typed and then click the “Run” bottom on the top right of the script file. Look in the “Console” pane of RStudio. What output do you see?
- In the script, replace each name with the name of a member of your group. Re-run the code by highlighting it again and the pressing the “Run” button. What output do you see in the console?
- Without changing the code, highlight and run it three more times. Is the output the same each time?
- Click your cursor in the R script pane and then save this R script by going to “File” -> “Save”. Save the file somewhere that’s easy to find, like on your Desktop (you can delete it after today). Save it with the name “example.R”.

Part I

Part I: Preliminaries

Chapter 1

R Preliminaries

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

1.1 Objectives

After this chapter, you should:

- Know what free and open source software is and some of its advantages over proprietary software
- Understand the difference between R and RStudio
- Be able to download both R and RStudio to your own computer
- Understand that R has a basic core of code that you initially download, and that this “base R” can be expanded by installing a variety of packages
- Be able to install a package from CRAN to your computer
- Be able to load a package that you have installed to use its functions within an R session
- Be able to access help documentation (vignettes, helpfiles) for a package and its functions
- Be able to submit R expressions at the console prompt to communicate with R
- Understand the structure for calling a function and specifying options for that function
- Know what an R object is and how to assign an R object a name to reference it in later code
- Be able to create vector objects of numeric and character classes
- Be able to explore and extract elements from vector objects
- Be able to create dataframe objects

- Be able to explore and extract elements from dataframe objects
- Be able to describe the difference between running R code from the console versus writing and running R code in an R script

1.2 R and R Studio

Download a pdf of the lecture slides for this video.

1.2.1 What is R?

R is an open-source programming language that evolved from the S language. The S language was developed at Bell Labs in the 1970s, which is the same place (and about the same time) that the C programming language was developed.

R itself was developed in the 1990s–2000s at the University of Auckland. It is open-source software, freely and openly distributed under the GNU General Public License (GPL). The base version of R that you download when you install R on your computer includes the critical code for running R, but you can also install and run “packages” that people all over the world have developed to extend R.

With new developments, R is becoming more and more useful for a variety of programming tasks. However, where it really shines is in working with data and doing statistical analysis. R is currently popular in a number of fields, including:

- Statistics
- Machine learning
- Data analysis

R is an **interpreted language**. That means that you can communicate with it interactively, from a command line. Other common interpreted languages include Python and Perl.

R has some of the same strengths (quick and easy to code, interfaces well with other languages, easy to work interactively) and weaknesses (slower than compiled languages) as Python. For data-related tasks, R and Python are fairly neck-and-neck (with Julia an up-and-coming option). However, R is still the first choice of statisticians in most fields, so I would argue that R has a advantage if you want to have access to cutting-edge statistical methods.

“The best thing about R is that it was developed by statisticians.
The worst thing about R is that... it was developed by statisticians.”
-Bo Cowgill, Google, at the Bay Area R Users Group

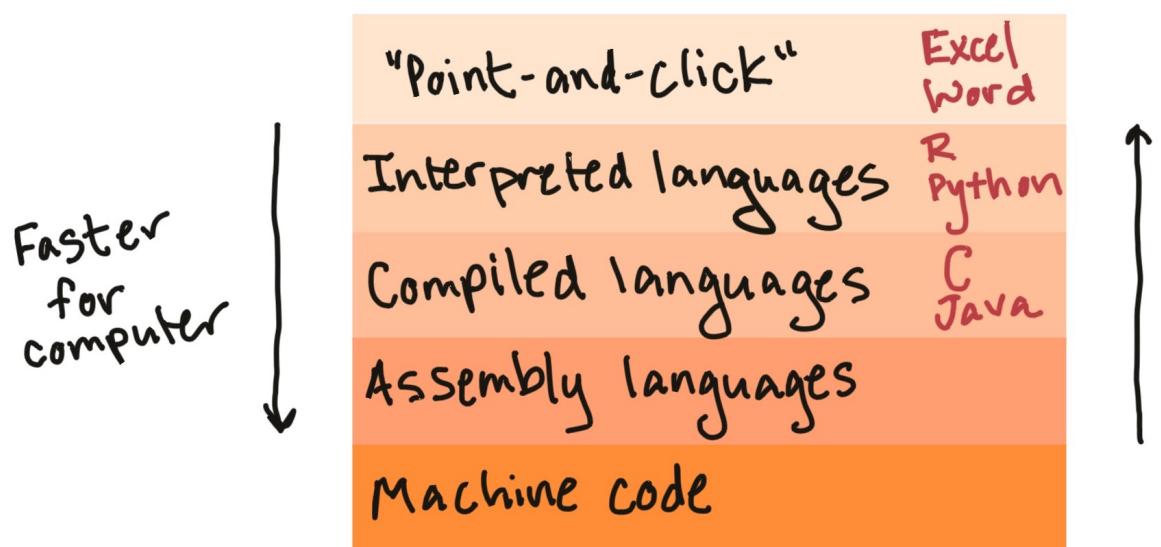


Figure 1.1: Broad types of software programs. R is an interpreted language. 'Point-and-click' programs, like Excel and Word, are often easiest for a new user to get started with, but are slower for the computer and are restricted in the functionality they offer. By contrast, compiled languages (like C and Java), assembly languages, and machine code are faster for the computer and allow you to create a wider range of things, but can take longer to code and take longer for a new user to learn to work with.

1.2.2 Free and open-source software

“Life is too short to run proprietary software.” – Bdale Garbee

R is **free and open-source software**. Many other popular statistical programming languages, conversely, are proprietary (for example, SAS and SPSS). It’s useful to know what it means for software to be “open-source”, both conceptually and in terms of how you will be able to use and add to R in your own work.

R is free, and it’s tempting to think of open-source software just as “free software”. Things, however, are a little more subtle than that. It helps to consider some different meanings of the word “free”. “Free” can mean:

- *Gratis*: Free as in beer
- *Libre*: Free as in speech

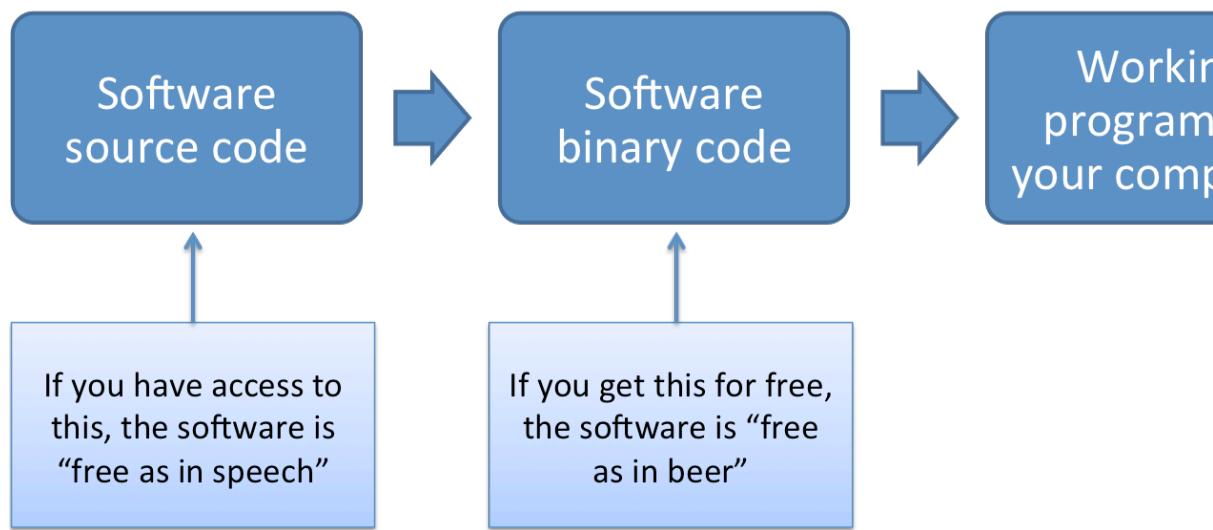


Figure 1.2: An overview of how software can be each type of free (beer and speech). For software programs developed using a compiled programming language, the final product that you open on your computer is run by machine-readable binary code. A developer can give you this code for free (as in beer) without sharing any of the original source code with you. This means you can’t dig in to figure out how the software works and how you can extend it. By contrast, open-source software (free as in speech) is software for which you have access to the human-readable code that was used as input in creating the software binaries. With open-source code, you can figure out exactly how the program is coded.

Open-source software is the *libre* type of free (Figure 1.2). This means that, with software that is open-source, you can:

- Access all of the code that makes up the software
- Change the code as you'd like for your own applications
- Build on the code with your own extensions
- Share the software and its code, as well as your extensions, with others

Often, open-source software is also free, making it “free and open-source software”, or “FOSS”.

Popular open source licenses for R and R packages include the GPL and MIT licenses.

“Making Linux GPL’d was definitely the best thing I ever did.” –
Linus Torvalds

In practice, this means that, once you are familiar with the software, you can dig deeply into the code to figure out exactly how it’s performing certain tasks. This can be useful for finding bugs and eliminating bugs, and also can help researchers figure out if there are any limitations in how the code works for their specific research.

It also means that you can build your own software on top of existing R software and its extensions. I explain a bit more about R packages a bit later, but this open-source nature of R (and other languages, including Python) has created a large community of people worldwide who develop and share extensions to R. As a result, you can pull in packages that let you do all kinds of things in R, like visualizing Tweets, cleaning up accelerometer data, analyzing complex surveys, fitting machine learning models, and a wealth of other cool things.

“Despite its name, open-source software is less vulnerable to hacking than the secret, black box systems like those being used in polling places now. That’s because anyone can see how open-source systems operate. Bugs can be spotted and remedied, deterring those who would attempt attacks. This makes them much more secure than closed-source models like Microsoft’s, which only Microsoft employees can get into to fix.” – Woolsey and Fox. *To Protect Voting, Use Open-Source Software*. New York Times. August 3, 2017.

You can download the latest version of R from CRAN. Be sure to select the distribution for your type of computer system. R is updated occasionally; you should plan to re-install R at least once a year, to make sure you’re working with one of the newer versions. Check your current R version (one way is by running `sessionInfo()` at the R console) to make sure you’re not using an outdated version of R. Defaults should be fine for everything.

“The R engine ... is pretty well uniformly excellent code but you have to take my word for that. Actually, you don’t. The whole engine is open source so, if you wish, you can check every line of it. If people were out to push dodgy software, this is not the way they’d go about it.” - Bill Venables, R-help (January 2004)

“Talk is cheap. Show me the code.” - Linus Torvalds

Download a pdf of the lecture slides for this video.

1.2.3 What is RStudio?

To get the R software, you’ll download R from the R Project for Statistical Computing. This is enough for you to use R on your own computer. However, I would suggest one additional, free piece of software to improve your experience while working with R, RStudio.

RStudio is an integrated development environment (IDE) for R. This basically means that it provides you an interface for running R and coding in R, with a lot of nice extras that will make your life easier.

You download RStudio separately from R—you’ll want to download and install R itself first, and then you can download RStudio. You want the Desktop version with the free license. Defaults should be fine for everything.

RStudio (the company) is a leader in the R community. Currently, the company:

- Develops and freely provides the RStudio IDE
- Provides excellent resources for learning and using R (e.g., cheatsheets, free online books)
- Is producing some of the most-used R packages
- Employs some of the top people in R development
- Is a key member of The R Consortium (others include Microsoft, IBM, and Google)

R has been advancing by leaps in bounds in terms of what it can do and the elegance with which it does it, in large part because of the enormous contributions of people involved with RStudio.

Download a pdf of the lecture slides for this video.

1.3 Communicating with R

Because R is an interpreted language, you can communicate with it interactively. You do this using the following general steps:

1. Open an **R session**
2. At the **prompt** in the **console**, enter an **R expression**
3. Read R's "response" (the **output**)
4. Repeat 2 and 3
5. Close the R session

1.3.1 R sessions, the console, and the command prompt

An **R session** is an instance of you using R. To open an R session, double-click on the icon for “RStudio” on your computer. When RStudio opens, you will be in a “fresh” R session, unless you restore a saved session (which I strongly recommend against). This means that, once you open RStudio, you will need to “set up” your session, including loading any packages you need (which we’ll talk about later) and reading in any data (which we’ll also talk about).

In RStudio, there screen is divided into several “panes”. We’ll start with the pane called “Console”. The **console** lets you “talk” to R. This is where you can “talk” to R by typing an **expression** at the **prompt** (the caret symbol, “>”). You press the “Return” key to send this message to R.

Once you press “Return”, R will respond in one of three ways:

1. R does whatever you asked it to do with the expression and prints the output (if any) of doing that, as well as a new prompt so you can ask it something new
2. R doesn’t think you’ve finished asking you something, and instead of giving you a new prompt (“>”) it gives you a “+”. This means that R is still listening, waiting for you to finish asking it something.
3. R tries to do what you asked it to, but it can’t. It gives you an **error message**, as well as a new prompt so you can try again or ask it something new.

1.3.2 R expressions, function calls, and objects

To “talk” with R, you need to know how to give it a complete **expression**. Most expressions you’ll want to give R will be some combination of two elements:

1. **Function calls**
2. **Object assignments**

We’ll go through both these pieces and also look at how you can combine them together for some expressions.

According to John Chambers, one of the creators of R’s precursor S:

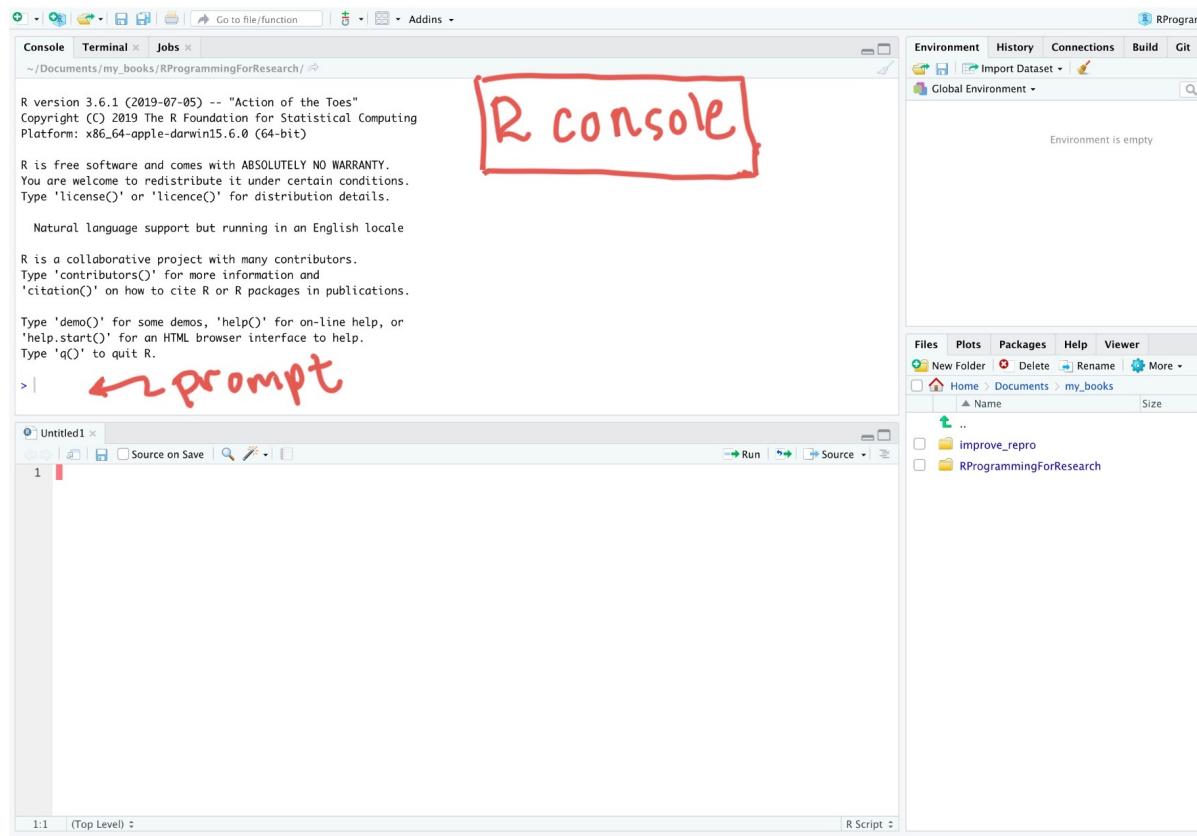


Figure 1.3: Finding the 'Console' pane and the command prompt in RStudio.

1. Everything that exists in R is an **object**
2. Everything that happens in R is a **call to a function**

Download a pdf of the lecture slides for this video.

1.4 Functions

In general, function calls in R take the following structure:

```
## Generic code (this won't run)
function_name(formal_argument_1 = named_argument_1,
              formal_argument_2 = named_argument_2,
              [etc.])
```



Sometimes, we'll show “generic” code in a code block, that doesn’t actually work if you put it in R, but instead shows the generic structure of an R call. We’ll try to always include a comment with any generic code, so you’ll know not to try to run it in R.

A function call forms a complete R expression, and the output will be the result of running `print` or `show` on the object that is output by the function call. Here is an example of this structure:

```
print(x = "Hello world")
```

```
## [1] "Hello world"
```

Figure 1.4 shows an example of the typical elements of a function call. In this example, we’re **calling** a function with the **name** `print`. It has one **argument**, with a **formal argument** of `x`, which in this call we’ve provided the **named argument** “Hello world”.

The **arguments** are how you customize the call to an R function. For example, you can use change the named argument value to print different messages with the `print` function:

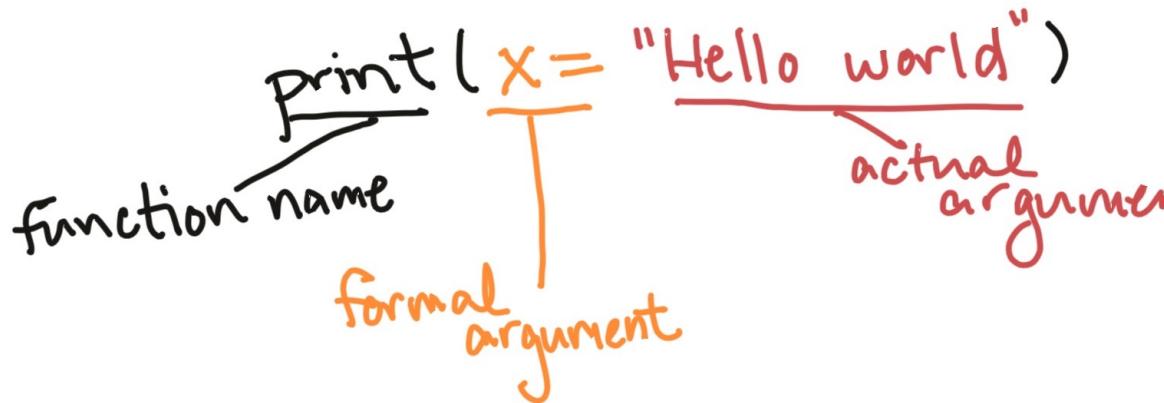


Figure 1.4: Main parts of a function call. This example is calling a function with the name 'print'. The function call has one argument, with a formal argument of 'x', which in this call is provided the named argument 'Hello world'.

```
print(x = "Hello world")
```

```
## [1] "Hello world"
```

```
print(x = "Hi Fort Collins")
```

```
## [1] "Hi Fort Collins"
```

Some functions do not require any arguments. For example, the `getRversion` function will print out the version of R you are using.

```
getRversion()
```

```
## [1] '4.2.3'
```

Some functions will accept multiple arguments. For example, the `print` function allows you to specify whether the output should include quotation marks, using the `quote` formal argument:

```
print(x = "Hello world", quote = TRUE)
```

```
## [1] "Hello world"
```

```
print(x = "Hello world", quote = FALSE)
```

```
## [1] Hello world
```

Arguments can be **required** or **optional**.

For a required argument, if you don't provide a value for the argument when you call the function, R will respond with an error. For example, `x` is a **required argument** for the `print` function, so if you try to call the function without it, you'll get an error:

```
print()
```

```
Error in print.default() : argument "x" is
missing, with no default
```

For an **optional argument** on the other hand, R knows a **default value** for that argument, so if you don't give it a value for that argument, it will just use the default value for that argument.

For example, for the `print` function, the `quote` argument has the default value `TRUE`. So if you don't specify a value for that argument, R will assume it should use `quote = TRUE`. That's why the following two calls give the same result:

```
print(x = "Hello world", quote = TRUE)
```

```
## [1] "Hello world"
```

```
print(x = "Hello world")
```

```
## [1] "Hello world"
```

Often, you'll want to find out more about a function, including:

- Examples of how to use the function
- Which arguments you can include for the function
- Which arguments are required versus optional
- What the default values are for optional arguments.

You can find out all this information in the function's **helpfile**, which you can access using the function `?`. For example, the `mean` function will let you calculate the mean (average) of a group of numbers. To find out more about this function, at the console type:

```
?mean
```

This will open a helpfile in the "Help" pane in RStudio. Figure 1.5 shows some of the key elements of an example helpfile, the helpfile for the `mean` function. In particular, the "Usage" section helps you figure out which arguments are **required** and which are **optional** in the Usage section of the helpfile.

There's one class of functions that looks a bit different from others. These are the infix **operator** functions. Instead using parentheses after the function name, they usually go *between* two arguments. One common example is the `+` operator:

```
2 + 3
```

```
## [1] 5
```

There are operators for several mathematical functions: `+`, `-`, `*`, `/`. There are also other operators, including **logical operators** and **assignment operators**, which we'll cover later.

Download a pdf of the lecture slides for this video.

1.5 Objects and assignment

In R, a variety of different types and structures of data can be saved in what's called **objects**. For right now, you can just think of an R object as a discrete container of data in R.

Function calls will produce an object. If you just call a function, as we've been doing, then R will respond by printing out that object. However, we'll

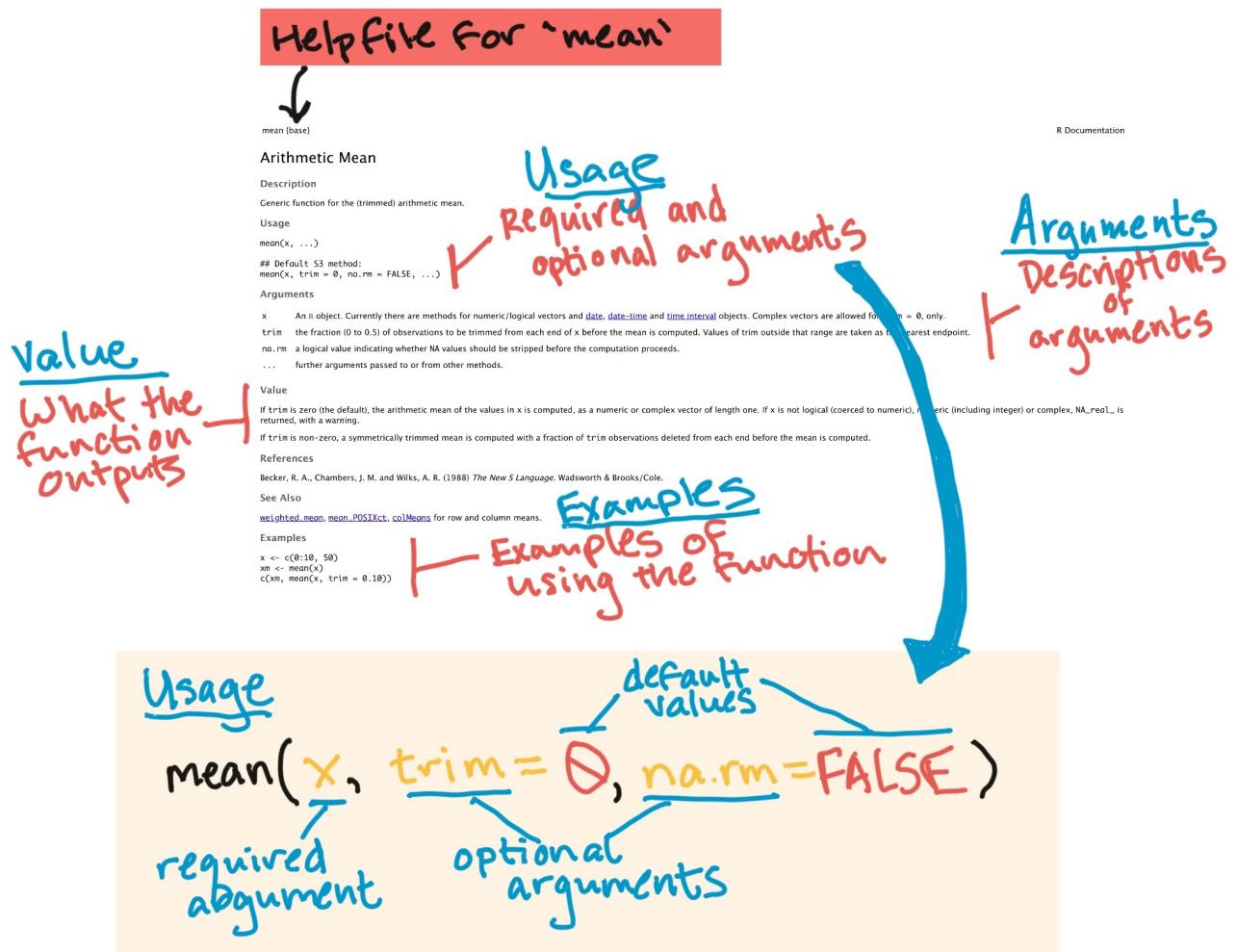


Figure 1.5: Navigating a helpfile. This example shows some key parts of the helpfile for the 'mean' function.

often want to use that object some more. For example, we might want to use it as an argument later in our “conversation” with R, when we call another function later. If you want to re-use the results of a function call later, you can **assign** that **object** to an **object name**. This kind of expression is called an **assignment expression**.

Once you do this, you can use that *object name* to refer to the object. This means that you don’t need to re-create the object each time you need it—instead you can create it once and then just reference it by name each time you need it after that. For example, you can read in data from an external file as a dataframe object and assign it an object name. Then, when you need that data later, you won’t need to read it in again from the external file.

The **gets arrow**, `<-`, is R’s assignment operator. It takes whatever you’ve created on the right hand side of the `<-` and saves it as an object with the name you put on the left hand side of the `<-`:

```
## Note: Generic code-- this will not work
[object name] <- [object]
```

For example, if I just type `"Hello world"`, R will print it back to me, but won’t save it anywhere for me to use later:

```
"Hello world"
```

```
## [1] "Hello world"
```

However, if I assign it to an object, I can “refer” to that object in a later expression. For example, the code below assigns the **object** `"Hello world"` the **object name** `message`. Later, I can just refer to this object using the name `message`, for example in a function call to the `print` function:

```
message <- "Hello world"
print(x = message)
```

```
## [1] "Hello world"
```

When you enter an **assignment expression** like this at the R console, if everything goes right, then R will “respond” by giving you a new prompt, without any kind of message.

However, there are three ways you can check to make sure that the object was assigned to the object name:

1. Enter the object's name at the prompt and press return. The default if you do this is for R to "respond" by calling the `print` function with that object as the `x` argument.
2. Call the `ls` function (which doesn't require any arguments). This will list all the object names that have been assigned in the current R session.
3. Look in the "Environment" pane in RStudio. This also lists all the object names that have been assigned in the current R session.

Here's are examples of these strategies:

1. Enter the object's name at the prompt and press return:

```
message
```

```
## [1] "Hello world"
```

2. Call the `ls` function:

```
ls()
```

```
## [1] "a"      "message"
```

3. Look in the "Environment" pane in RStudio (see Figure 1.6).

You can make assignments in R using either the gets arrow (`<-`) or `=`. When you read other people's code, you'll see both. R gurus advise using `<-` rather than `=` when coding in R, and as you move to doing more complex things, some subtle problems might crop up if you use `=`. I have heard from someone in the know that you can tell the age of a programmer by whether he or she uses the gets arrow or `=`, with `=` more common among the young and hip. For this course, however, I am asking you to code according to Hadley Wickham's R style guide, which specifies using the gets arrow for assignment.

While you will be coding with the gets arrow exclusively in this course, it will be helpful for you to know that the two assignment arrows do pretty much the same thing:

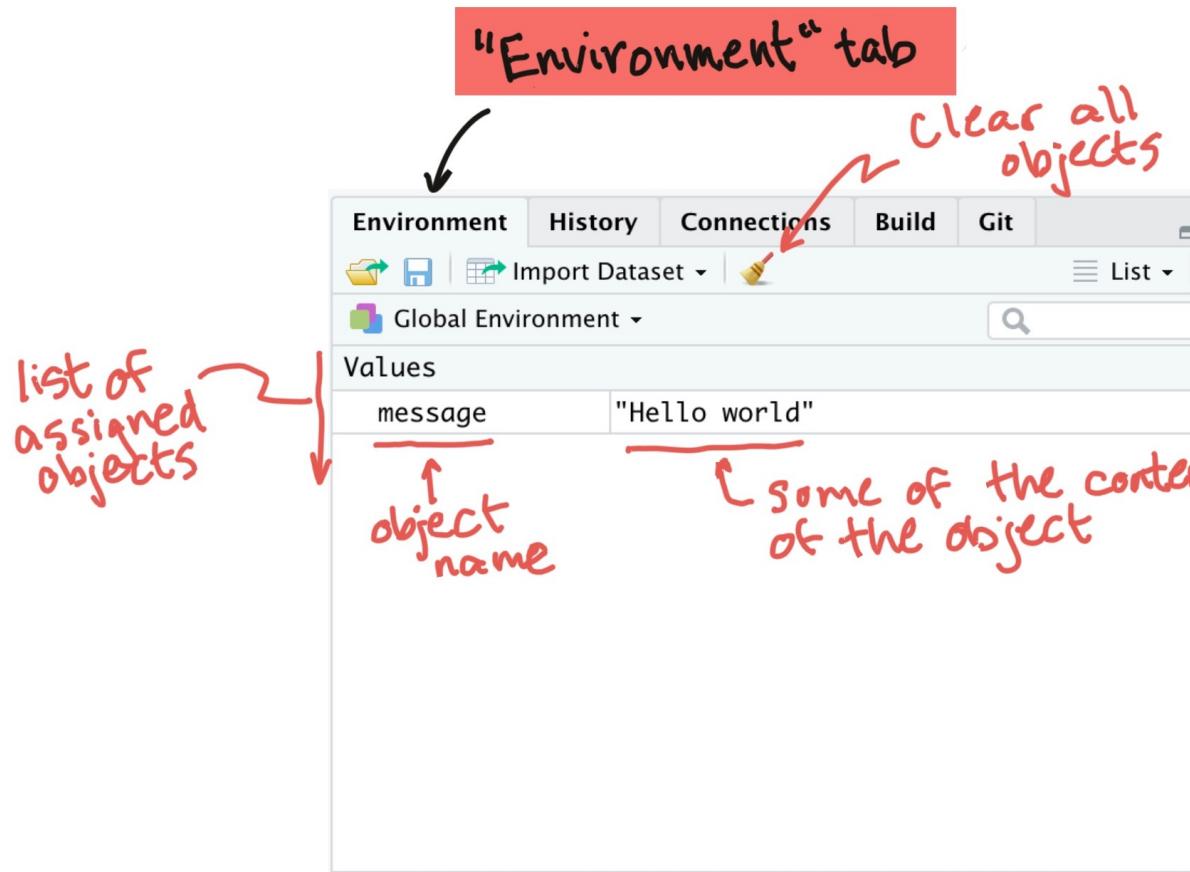


Figure 1.6: 'Environment' pane in RStudio. This shows the names and first few values of all objects that have been assigned to object names in the global environment.

```
one_to_ten <- 1:10
one_to_ten
```

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

```
one_to_ten = 1:10
one_to_ten
```

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

While the gets arrow takes two key strokes instead of one (like the equals sign), you can somewhat get around this limitation by using RStudio's keyboard shortcut for the gets arrow. This shortcut is Alt + - on Windows and Option + - on Macs. To see a full list of RStudio keyboard shortcuts, go to the "Help" tab in RStudio and select "Keyboard Shortcuts".

There are some absolute **rules** for the names you can use for an object name:

- Use only letters, numbers, and underscores
- Don't start with anything but a letter

If you try to assign an object to a name that doesn't follow the "hard" rules, you'll get an error. For example, all of these expressions will give you an error:

```
1message <- "Hello world"
_message <- "Hello world"
message! <- "Hello world"
```

In addition to these fixed rules, there are also some guidelines for naming objects that you should adopt now, since they will make your life easier as you advance to writing more complex code in R. The following three guidelines for naming objects are from Hadley Wickham's R style guide:

- Use lower case for variable names (`message`, not `Message`)
- Use an underscore as a separator (`message_one`, not `messageOne`)
- Avoid using names that are already defined in R (e.g., don't name an object `mean`, because a `mean` function exists)

“Don’t call your matrix ‘matrix’. Would you call your dog ‘dog’?
 Anyway, it might clash with the function ‘matrix’.” - Barry Rowlingson, R-help (October 2004)

Another good practice is to name objects after nouns (e.g., `message`) and later, when you start writing functions, name those after verbs (e.g., `print_message`). You’ll want your object names to be short enough that they don’t take forever to type as you’re coding, but not so short that you can’t remember what they stand for.



Sometimes, you’ll want to create an object that you won’t want to keep for very long. For example, you might want to create a small object to test some code, but you plan to not need the object again once you’ve done that. You may want to come up with some short, generic object names that you use for these kinds of objects, so that you’ll know that you can delete them without problems when you want to clean up your R session. There are all kinds of traditions for these placeholder variable names in computer science. `foo` and `bar` are two popular choices, as are, evidently, `xyzzy`, `spam`, `ham`, and `norf`. There are different placeholder names in different languages: for example, `toto`, `truc`, and `azerty` (French); and `pippo`, `pluto`, `paperino` (Disney character names; Italian). See the Wikipedia page on metasyntactic variables to find out more.

Download a pdf of the lecture slides for this video.

1.6 More on communicating with R

What if you want to “compose” a call from more than one function call? One way to do it is to assign the output from the first function call to a name and then use that name for the next call. For example:

```
message <- paste("Hello", "world")
print(x = message)
```

```
## [1] "Hello world"
```

If you give two objects the same name, the most recent definition will be used (i.e., objects can be overwritten by assigning new content to the same object name). For example:

```
a <- 1:10
b <- LETTERS [1:3]

a
```

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

```
b
```

```
## [1] "A" "B" "C"
```

```
a <- b
a
```

```
## [1] "A" "B" "C"
```

To create an R expression you can “nest” one function call inside another function call. For example:

```
print(x = paste("Hello", "world"))
```

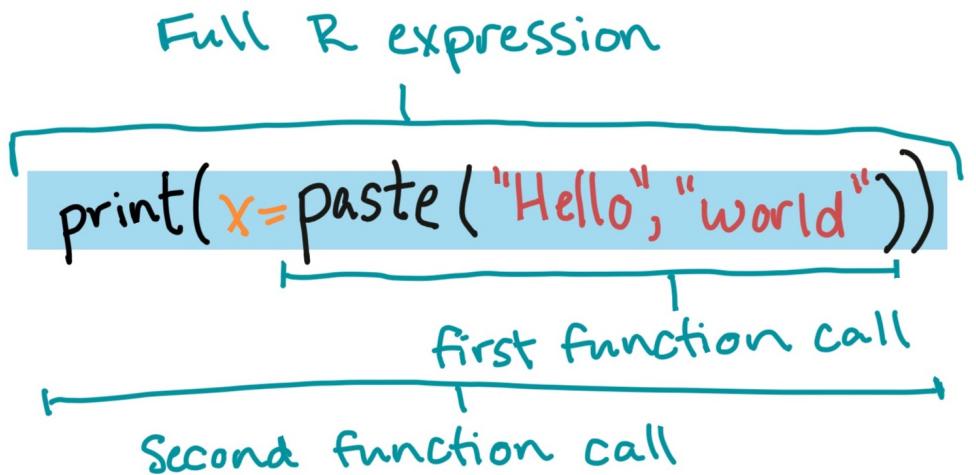
```
## [1] "Hello world"
```

Just like with math, the order that the functions are evaluated moves from the inner set of parentheses to the outer one (Figure 1.7). There’s one more way we’ll look at later called “piping”.

1.7 R scripts

This is a good point in learning R for you to start putting your code in R scripts, rather than entering commands at the console.

An R script is a plain text file where you can save a series of R commands. You can save the script and open it up later to see (or re-do) what you did earlier, just like you could with something like a Word document when you’re writing a paper. To open a new R script in RStudio, go to the menu bar and select “File”



① $\text{paste}(\text{"Hello", "world"}) \rightarrow \text{"Hello world"}$

② $\text{print}(x = \text{"Hello world"})$

Figure 1.7: 'Environment' pane in RStudio. This shows the names and first few values of all objects that have been assigned to object names in the global environment.

The screenshot shows the RStudio interface with an R script editor window. The window title is "InCourseExercises_Week2.Rmd". The code editor contains the following R code:

```
1 ## Some example code to show a script file
2
3 one_to_ten <- 1:10
4
5 course_dates <- data.frame(session = c(1, 2, 3),
6                             topic = c("Basic R",
7                                   "Getting and Cleaning Data 1",
8                                   "Exploring Data 1"))
9
10 a <- 1:4 ; b <- rnorm(10)
11 |
```

Annotations with arrows point to specific elements:

- A red arrow points to the "Save" button in the toolbar, labeled "Save" button.
- A red arrow points to the "Run" button in the toolbar, labeled "Run" button.
- A callout box labeled "Commented" has an arrow pointing to the multi-line comment starting at line 5.
- A callout box labeled "One command across several lines" has an arrow pointing to the semicolon-separated assignment statement at line 10.
- A callout box labeled "Two commands on one line" has an arrow pointing to the line separator at line 11.

Figure 1.8: Example of an R script in RStudio.

-> “New File” -> “R Script”. Alternatively, you can use the keyboard shortcut Command-Shift-N. Figure 1.8 gives an example of an R script file opened in RStudio and points out some interesting elements.

To save a script you’re working on, you can click on the “Save” button (which looks like a floppy disk) at the top of your R script window in RStudio or use the keyboard shortcut Command-S. You should save R scripts using a “.R” file extension.

Within the R script, you’ll usually want to type your code so there’s one command per line. If your command runs long, you can write a single call over multiple lines. It’s unusual to put more than one command on a single line of a script file, but you can if you separate the commands with semicolons (;). These rules all correspond to how you can enter commands at the console.

Running R code from a script file is very easy in RStudio. You can use either the “Run” button or Command-Return, and any code that is selected (i.e., that you’ve highlighted with your cursor) will run at the console. You can use this functionality to run a single line of code, multiple lines of code, or even just part of a specific line of code. If no code is highlighted, then R will instead run all the code on the line with the cursor and then move the cursor down to the next line in the script.

You can also run all of the code in a script. To do this, use the “Source” button at the top of the script window. You can also run the entire script either from the console or from within another script by using the `source()` function, with the filename of the script you want to run as the argument. For example, to run all of the code in a file named “MyFile.R” that is saved in your current working directory, run:

```
source("MyFile.R")
```

You can add comments into an R script to let others know (and remind yourself) what you’re doing and why. To do this, use R’s comment character, `#`. Any line on a script line that starts with `#` will not be read by R. You can also take advantage of commenting to comment out certain parts of code that you don’t want to run at the moment.

While it’s generally best to write your R code in a script and run it from there rather than entering it interactively at the R console, there are some exceptions. A main example is when you’re initially checking out a dataset, to make sure you’ve read it in correctly. It often makes more sense to run commands for this task, like `str()`, `head()`, `tail()`, and `summary()`, at the console. These are all examples of commands where you’re trying to look at something about your data **right now**, rather than code that builds toward your analysis, or helps you read in or clean up your data.

1.7.1 Commenting code

Sometimes, you’ll want to include notes in your code. You can do this in all programming languages by using a *comment character* to start the line with your comment. In R, the comment character is the hash symbol, `#`. R will skip any line that starts with `#` in a script. For example, if you run the following code:

```
# Don't print this.  
"But print this"
```

```
## [1] "But print this"
```

R will only print the second, uncommented line.

You can also use a comment in the middle of a line, to add a note on what you’re doing in that line of the code. R will skip any part of the code from the hash symbol on. For example:

```
"Print this" ## But not this, it's a comment.
```

```
## [1] "Print this"
```

There’s typically no reason to use code comments when running commands at the R console. However, it’s very important to get in the practice of including meaningful comments in R scripts. This helps you remember what you did when you revisit your code later.

“You know you’re brilliant, but maybe you’d like to understand what you did 2 weeks from now.” – Linus Torvalds

Download a pdf of the lecture slides for this video.

1.8 The “package” system

1.8.1 R packages

“Any doubts about R’s big-league status should be put to rest, now that we have a Sudoku Puzzle Solver. Take that, SAS!” - David Brahm (announcing the sudoku package), R-packages (January 2006)

Your original download of R is only a starting point. You can expand functionality of R with what are called *packages*, or extensions with new code and functionality that add to the basic “base R” environment. To me, this is a bit like the toy train set that my son was obsessed with for a while. You first buy a very basic set that looks something like Figure 1.9.



Figure 1.9: The toy version of base R.

To take full advantage of R, you’ll want to add on packages. In the case of the train set, at this point, a doting grandparent adds on extensively through birthday presents, so you end up with something that looks like Figure 1.10.

Each package is basically a bundle of extra R functions. They may also include help documentation, datasets, and some other objects, but typically the heart of an R package is the new functions it provides.

You can get these “add-on” packages in a number of ways. The main source for installing packages for R remains the Comprehensive R Archive Network, or CRAN. However, GitHub is growing in popularity, especially for packages that are still in development. You can also create and share packages among your collaborators or co-workers, without ever posting them publicly. In the



Figure 1.10: The toy version of what your R set-up will look like once you find cool packages to use for your research.

“Advanced” section of this course, you will learn some about writing your own R package.

1.8.2 Installing from CRAN

Dirk Eddelbuettel @eddelbuettel · 27 Jan 2017

Big congratulations to @gbwandleron whose new package ' just became package 10,000 on CRAN !!

CRAN Package Updates @CRANberriesFeed

9999 packages on CRAN right now, so imagine dozens of in suspense waiting for the package to make it 10k ...

2 35 93

Figure 1.11: Celebrating CRAN’s 10,000th package.

The most popular place from which to get packages is currently CRAN, which has over 10,000 R packages available (Figure 1.11). You can install packages from CRAN using R code, with the `install.packages` function. For example, telephone keypads include letters for each number (Figure 1.12), which allow companies to have “named” phone numbers that are easier for people to remember, like 1-800-GO-FEDEX and 1-800-FLOWERS.

The `phonenumbers` package is a cool little package that will convert between numbers and letters based on the telephone keypad. Since this package is on CRAN, you can install the package to your computer using the `install.packages` function:

```
install.packages(pkgs = "phonenumbers")
```

This downloads the package from CRAN and saves it in a special location on your computer where R can load it when you’re ready to use it. Once you’ve installed a package to your computer this way, you don’t need to re-run this



Figure 1.12: Telephone keypad with letters corresponding to each number.

`install.packages` for the package ever again (unless the package maintainer posts an updated version).

Just like R itself, packages often evolve and are updated by their maintainers. You should update your packages as new versions come out. Typically, you have to reinstall packages when you update your version of R, so this is a good chance to get the most up-to-date version of the packages you use.

1.8.3 Loading an installed package

Once you have installed a package, it will be saved to your computer. However, you won’t be able to access its functions within an R session until you *load* it in that R session. Loading a package essentially makes all of the package’s functions available to you.

You can load a package in an R session using the `library` function, with the package name inside the parentheses.

```
library(package = "phonenumbers")
```

Figure 1.13 provides a conceptual picture of the different steps of installing and loading a package.

Once a package is loaded, you can use all its exported (i.e., public) functions by calling them directly. For example, the `phonenumbers` has a function called

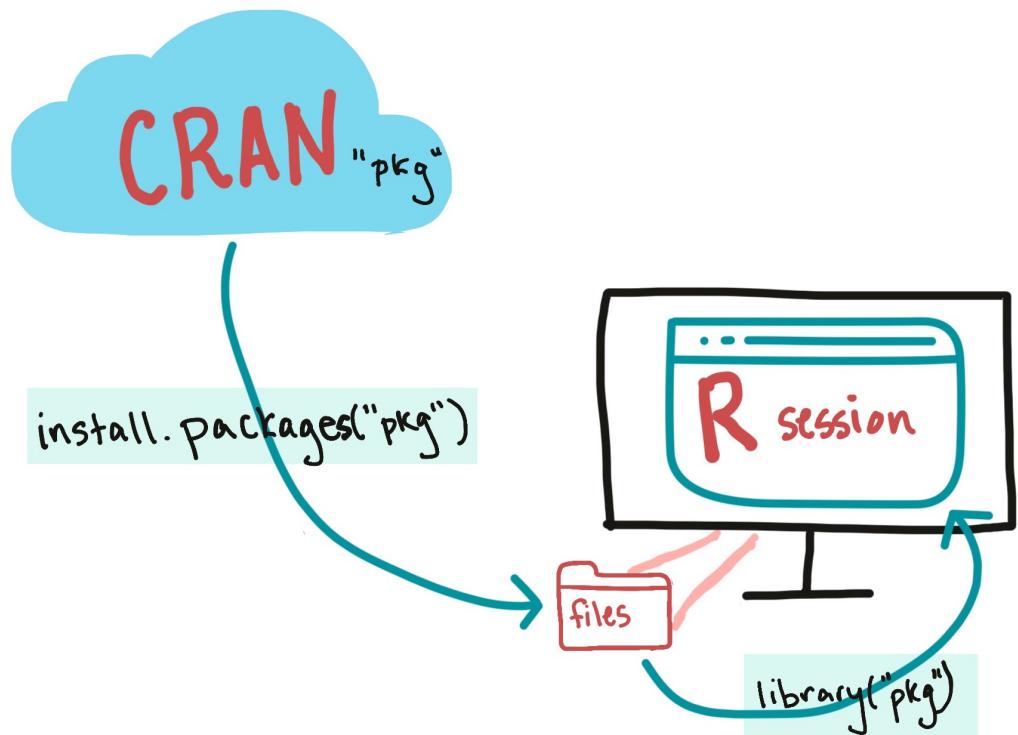


Figure 1.13: Install a package (with 'install.packages') to get it onto your computer. Load it (with 'library') to get it into your R session.

`letterToNumber` that converts a character string to a number. If you have not loaded the `phonenumbers` package in your current R session and try to use this function, you will get an error. However, once you’ve loaded `phonenumbers` using the `library` function, you can use this function in your R session:

```
fedex_number <- "GoFedEx"
letterToNumber(value = fedex_number)
```

```
## [1] "4633339"
```



R vectors can have several different *classes*. One common class is the character class, which is the class of the character string we’re using here (“GoFedEx”). You’ll always put character strings in quotation marks. Another key class is numeric (`numbers`). Later in the course, we’ll introduce other classes that vectors can have, including factors and dates. For the simplest vector classes, these classes are determined by the type of data that the vector stores.

When you open RStudio, unless you reload the history of a previous R session (which I typically strongly **do not** recommend), you will start your work in a “fresh” R session. This means that, once you open RStudio, you will need to run the code to load any packages, define any objects, and read in any data that you will need for analysis in that session.

If you are using a package in academic research, you should cite it, especially if it implements an algorithm or method that is not standard. You can use the `citation` function to get the information you need about how to cite a package:

```
citation(package = "phonenumbers")
```

```
##
## To cite package 'phonenumbers' in publications use:
##
##   Myles S (2021). _phonenumbers: Convert Letters to Numbers and Back as
##   on a Telephone Keypad_. R package version 0.2.3,
##   <https://CRAN.R-project.org/package=phonenumbers>.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
```

```
##   title = {phononenumber: Convert Letters to Numbers and Back as on a Telephone Key}
##   author = {Steve Myles},
##   year = {2021},
##   note = {R package version 0.2.3},
##   url = {https://CRAN.R-project.org/package=phononenumber},
## }
```



We've talked here about loading packages using the `library` function to access their functions. However, this is not the only way to access the package's functions. The syntax `[package name]::[function name]` (e.g., `phononenumber::letterToNumber(fedex)`) will allow you to use a function from a package you have installed on your computer, even if its package has not been loaded in the current R session. Typically, this syntax is not used much in data analysis scripts, in part because it makes the code much longer. However, you will occasionally see it used to distinguish between two functions from different packages that have the same name, as this format makes the desired function unambiguous. One example where this syntax often is needed is when both `plyr` and `dplyr` packages are loaded in an R session, since these share functions with the same name.

Packages typically include some documentation to help users. These include:

- **Package vignettes:** Longer, tutorial-style documents that walk the user through the basics of how to use the package and often give some helpful example cases of the package in use.
- **Function helpfiles:** Files for each external function (i.e., the package maintainer wants it to be used by others) within the package, following an established structure. These include information about what inputs are required and optional for the function, what output will be created, and what options can be selected by the user. In many cases, these also include examples of using the function.

To determine which vignettes are available for a package, you can use the `vignette` function, with the package's name specified for the `package` option:

```
vignette(package = "phononenumber")
```

From the output of this, you can call any of the package's vignettes directly. For example, the previous call tells you that this package only has one vignette, and that vignette has the same name as the package ("phononenumber"). Once

you know the name of the vignette you would like to open, you can also use `vignette` to open it:

```
vignette(topic = "phonenumbers")
```

To access the helpfile for any function within a package you've loaded, you can use `? followed by the function's name:`

```
?letterToNumber
```

Download a pdf of the lecture slides for this video.

1.9 R's most basic object types

An R object stores some type of data that you want to use later in your R code, without fully recreating it. The content of R objects can vary from very simple (the "GoFedEx" string in the example code above) to very complex objects with lots of elements (for example, a machine learning model).

Objects can be structured in different ways, in terms of how they "hold" data. These difference structures are called **object classes**. One class of objects can be a subtype of a more general object class.

There are a variety of different object types in R, shaped to fit different types of objects ranging from the simple to complex. In this section, we'll start by describing two object types that you will use most often in basic data analysis, **vectors** (1-dimensional objects) and **dataframes** (2-dimensional objects).

For these two object classes (vectors and dataframes), we'll look at:

1. How that class is structured
2. How to make a new object with that class
3. How to extract values from objects with that class

In later classes, we'll spend a lot of time learning how to do other things with objects from these two classes, plus learn some other classes.

1.9.1 Vectors

To get an initial grasp of the *vector* object type in R, think of it as a 1-dimensional object, or a string of values. Figure 1.14 provides an example



Figure 1.14: An example of the structure of an R object with the vector class. This object class contains data as a string of values, all with the same data type.

of the structure for a very simple vector, one that holds the names of the three main characters in the *Harry Potter* book series.

All values in a vector must be of the same data type (i.e., all numbers, all characters, all dates). If you try to create a vector with elements from different types (like "FedEx", which is a character, and 3, a number), R will coerce all of the elements to the most generic type of any of the elements (i.e., "FedEx" and "3" will both become characters, since "3" can be changed to a character, but "FedEx" can't be changed to a number). Figure 1.15 gives some examples of different classes of vectors.

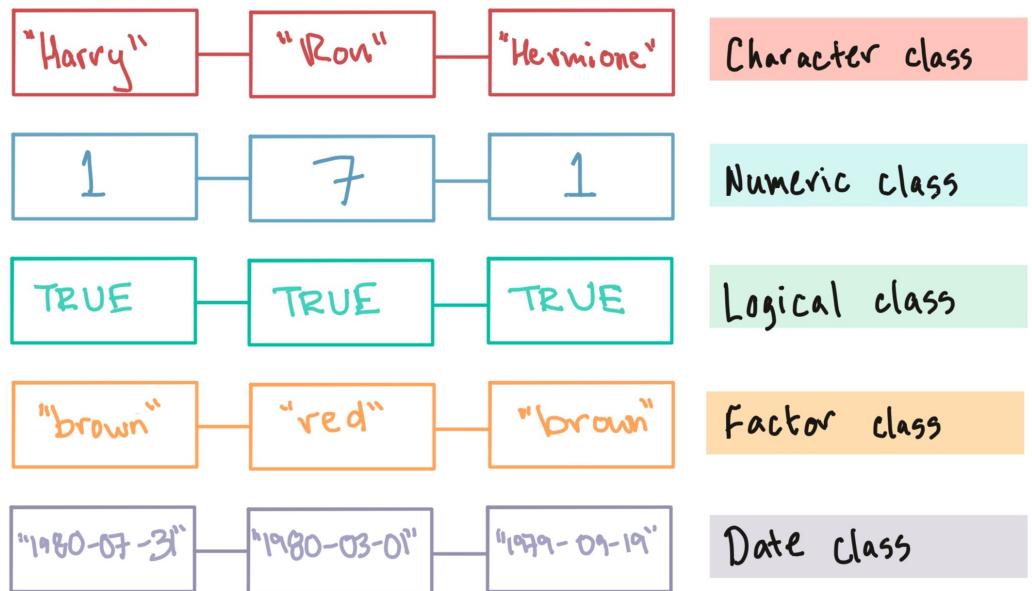


Figure 1.15: Examples of vectors of different classes. All the values in a vector must be of the same type (e.g., all numbers, all characters). There are different classes of vectors depending on the type of data they store.

To create a vector from different elements, you'll use the concatenation function, `c` to join them together, with commas between the elements. For example, to create the vector shown in Figure 1.14, you can run:

```
c("Harry", "Ron", "Hermione")
## [1] "Harry"     "Ron"       "Hermione"
```

If you want to use that object later, you can assign it an object name in the expression:

```
main_characters <- c("Harry", "Ron", "Hermione")
print(x = main_characters)
## [1] "Harry"     "Ron"       "Hermione"
```

This **assignment expression**, for assigning a vector an object name, follows the structure we covered earlier for function calls and assignment expressions (Figure 1.16).

If you create a numeric vector, you should not put the values in quotation marks:

```
n_kids <- c(1, 7, 1)
```

If you mix classes when you create the vector, R will coerce all the elements to most generic of the elements' classes:

```
mixed_classes <- c(1, 3, "five")
mixed_classes
## [1] "1"      "3"      "five"
```

Notice that the two integers, 1 and 3, are now in quotation marks, once they are put in a vector with a value with the character data type. You can use the `class` function to determine the class of an object:

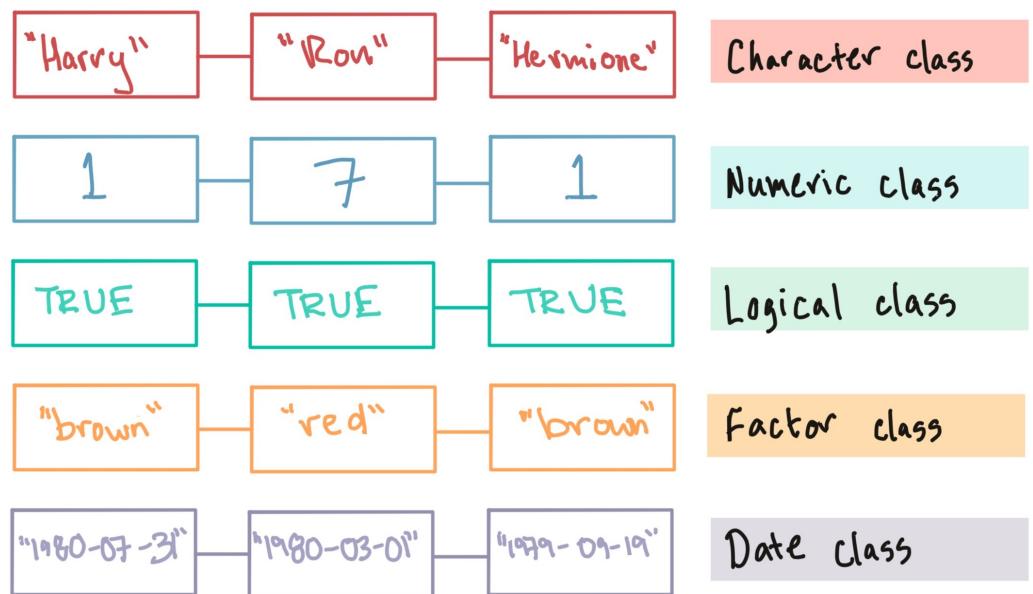


Figure 1.16: Elements of the assignment expression for creating a vector and assigning it an object name.

```
class(x = mixed_classes)
```

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

A vector's *length* is the number of elements in the vector. You can use the `length` function to determine a vector's length:

```
length(x = mixed_classes)
```

```
## [1] 3
```

Once you create an object, you will often want to reference the whole object in future code. However, there will be some times when you'll want to reference just certain elements of the object (for example, the first three values). You can pull out certain values from a vector by using indexing with square brackets (`[...]`) to identify the locations of the element you want to extract. For example, to extract the second element of the `main_characters` vector, you can run:

```
main_characters[2] # Get the second value
```

```
## [1] "Ron"
```

You can use this same method to extract more than one value. You just need to create a numeric vector with the position of each element you want to extract and pass that in the square brackets. For example, to extract the first and third elements of the `main_characters` vect, you can run:

```
main_characters[c(1, 3)] # Get first and third values
```

```
## [1] "Harry"    "Hermione"
```

The `:` operator can be very helpful with extracting values from a vector. This operator creates a sequence of values from the value before the `:` to the value after `:`, going by units of 1. For example, if you want to create a list of the numbers between 1 and 10, you can run:

1:10

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

If you want to extract the first two values from the `main_characters` vector, you can use the `:` operator:

```
main_characters[1:2] # Get the first two values
```

```
## [1] "Harry" "Ron"
```

You can also use logic to pull out some values of a vector. For example, you might only want to pull out even values from the `fibonacci` vector. We'll cover using logical expressions to index vectors later in the book.



One thing that people often find confusing when they start using R is knowing when to use and not use quotation marks. The general rule is that you use quotation marks when you want to refer to a character string literally, but no quotation marks when you want to refer to the value in a previously-defined object. For example, if you saved the string "Anderson" as the object `my_name` (`my_name <- "Anderson"`), then in later code, if you type `my_name` (no quotation marks), you'll get "Anderson", while if you type out "`my_name`" (with quotation marks), you'll get "my_name" (what you typed, literally).

One thing that makes this rule confusing is that there are a few cases in R where you really should (by this rule) use quotation marks, but the function is coded to let you be lazy and get away without them. One example is the `library` function. In the code earlier in this section to load the "phonenumbers" package, you want to literally load the package "phonenumbers", rather than load whatever character string is saved in the object named `phonenumbers`. However, `library` is one of the functions where you can be lazy and skip the quotation marks, and it will still load "phonenumbers" for you. Therefore, if you want, this function also works if you call `library(package = phonenumbers)` (without the quotation marks) instead of how we actually called it (`library(package = phonenumbers)`).

Download a pdf of the lecture slides for this video.

1.9.2 Dataframes

A dataframe is a 2-dimensional object, and is made of one or more vectors of the same length stuck together side-by-side. It is the closest R has to an Excel spreadsheet-type structure. Figure 1.17 gives a conceptual example of a dataframe created from several of the vector examples in Figure ??.

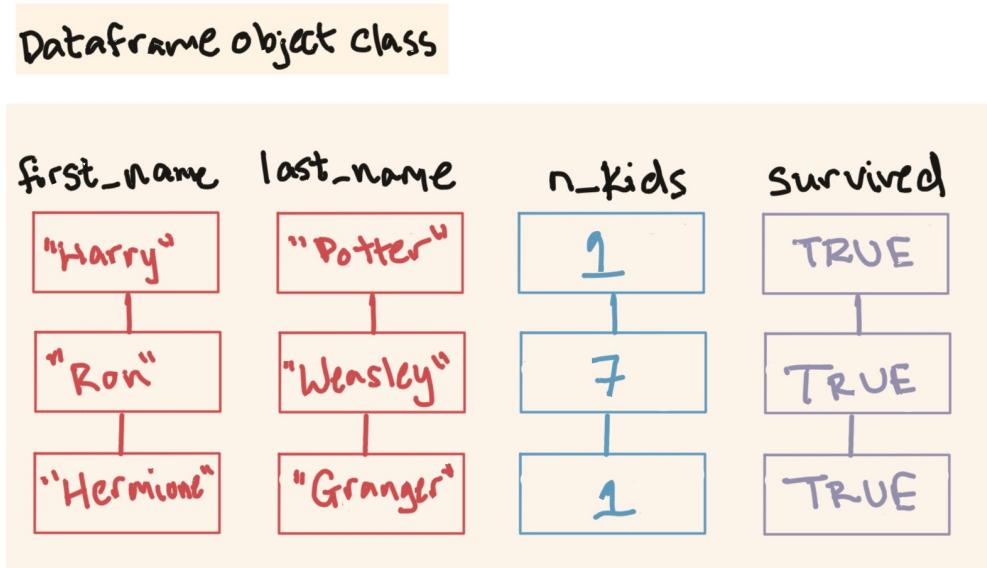


Figure 1.17: An example dataframe, created from several vectors of the same length and with observations aligned across vector positions (for example, the first value in each vector provides a value for Harry, the second for Ron).

Here's how the dataframe in Figure 1.17 will look in R:

```
## # A tibble: 3 x 4
##   first_name last_name n_kids survived
##   <chr>      <chr>     <dbl> <lgl>
## 1 Harry      Potter      1 TRUE
## 2 Ron        Weasley     7 TRUE
## 3 Hermione   Granger     1 TRUE
```

This dataframe is arranged in rows and columns, with column names for each column (Figure 1.18). Note that each row of this dataframe gives a different observation (in this case, our unit of observation is a Harry Potter character). Each column gives a different type of information (first name, last name, birth year, and whether they're still alive) for each of the observations (Beatles).

Notice that the number of elements in each of the columns must be the same in this dataframe, but that the different columns can have different classes of data (e.g., character vectors for `first_name` and `last_name`, logical value of TRUE or FALSE for `alive`).

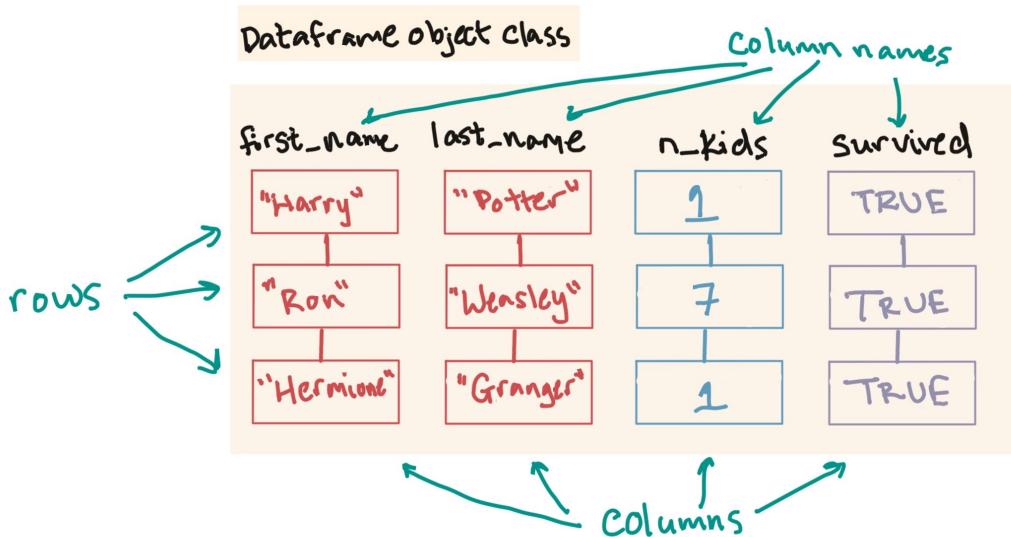


Figure 1.18: The elements of a dataframe: columns, rows, and column names.

We'll be working with a specific class of dataframe called a **tibble**. You can create tibble dataframes using the `tibble` function from the `tibble` package. However, most often you will create a dataframe by reading in data from a file, using something like `read_csv` from the `readr` package.



There are base R functions for both of these tasks (`data.frame` and `read.csv`, respectively), eliminating the need to load additional packages with a `library` call. However, the series of packages that make up what's called the "tidyverse" have brought a huge improvement in the ease and speed of working with data in R. We will be teaching these tools in this course, and that's why we're going directly to `tibble` and `read_csv` from the start, rather than base R equivalents. Later in the course, we'll talk more about this "tidyverse" and what makes it so great.

To create a dataframe, you can use the `tibble` function from the `tibble` package. The general format for using `tibble` is:

```
## Note: Generic code
[name of object] <- tibble([1st column name] = [1st column content],
                           [2nd column name] = [2nd column content])
```

with an equals sign between the column name and column content for each column, and commas between each of the columns.

Here is an example of the code used to create the *Harry Potter* tibble dataframe shown above:

```
library(package = "tibble")
hp_data <- tibble(first_name = c("Harry", "Ron", "Hermione"),
                  last_name = c("Potter", "Weasley", "Granger"),
                  n_kids = c(1, 7, 1),
                  survived = c(TRUE, TRUE, TRUE))
hp_data
```

```
## # A tibble: 3 x 4
##   first_name last_name n_kids survived
##   <chr>      <chr>     <dbl> <lgl>
## 1 Harry      Potter      1 TRUE
## 2 Ron        Weasley     7 TRUE
## 3 Hermione   Granger     1 TRUE
```

You can also create a dataframe by sticking together vectors you already have saved as R objects. For example:

```
hp_data <- tibble(first_name = main_characters,
                  last_name = c("Potter", "Weasley", "Granger"),
                  n_kids = n_kids,
                  survived = c(TRUE, TRUE, TRUE))
hp_data
```

```
## # A tibble: 3 x 4
##   first_name last_name n_kids survived
##   <chr>      <chr>     <dbl> <lgl>
## 1 Harry      Potter      1 TRUE
## 2 Ron        Weasley     7 TRUE
## 3 Hermione   Granger     1 TRUE
```

Note that this call requires that the `main_characters` and `n_kids` vectors are the same length, although they don't have to be (and in this case aren't) the same class of objects (`main_characters` is a character class, `n_kids` is numeric).



You can put more than one function call in a single line of R code, as in this example (the `c` creates a vector, while the `tibble` creates a dataframe, using the vectors created by the calls to `c`). When you use multiple functions within a single R call, R will evaluate starting from the inner-most parentheses out, much like the order of operations in a math equation with parentheses.

So far, we've only shown how to create dataframes from scratch within an R session. Usually, however, you'll create R dataframes instead by reading in data from an outside file using the `read_csv` from the `readr` package and related functions. For example, you might want to analyze data on all the guests that came on the *Daily Show*, circa Jon Stewart. If you have this data in a comma-separated (csv) file on your computer called “`daily_show_guests.csv`” (see the In-Course Exercise for instructions on downloading it), you can read it into your R session with the following code:

```
library(package = "readr")
daily_show <- read_csv(file = "daily_show_guests.csv",
                      skip = 4)
```

In this code, the `read_csv` function is reading in the data from the file “`daily_show_guests.csv`”, while the gets arrow (`<-`) assigns that data to the object `daily_show`, which you can then reference in later code to explore and plot the data.

You can use the functions `dim`, `nrow`, and `ncol` to figure out the dimensions (number of rows and columns) of a dataframe:

```
dim(x = daily_show)
```

```
## [1] 2693     5
```

```
nrow(x = daily_show)
```

[1] 2693

```
ncol(x = daily_show)
```

```
## [1] 5
```

Base R also has some useful functions for quickly exploring dataframes:

- **str**: Show the structure of an R object, including a dataframe.
 - **summary**: Give summaries of each column of a dataframe.

For example, you can explore the data we just pulled in on the *Daily Show* with:

```
str(object = daily_show)
```

```

##> #> spc_tbl_ [2,693 x 5] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##> #>   $ YEAR                  : num [1:2693] 1999 1999 1999 1999 1999 ...
##> #>   $ GoogleKnowlege_Occupation: chr [1:2693] "actor" "Comedian" "television actress" "film actre
##> #>   $ Show                  : chr [1:2693] "1/11/99" "1/12/99" "1/13/99" "1/14/99" ...
##> #>   $ Group                 : chr [1:2693] "Acting" "Comedy" "Acting" "Acting" ...
##> #>   $ Raw_Guest_List        : chr [1:2693] "Michael J. Fox" "Sandra Bernhard" "Tracey Ullman"
##> - attr(*, "spec")=
##> .. cols(
##> ..   YEAR = col_double(),
##> ..   GoogleKnowlege_Occupation = col_character(),
##> ..   Show = col_character(),
##> ..   Group = col_character(),
##> ..   Raw_Guest_List = col_character()
##> .. )
##> - attr(*, "problems")=<externalptr>

```

```
summary(object = daily_show)
```

```
##      YEAR GoogleKnowlege_Occupation Show Group
##  Min. :1999 Length:2693          Length:2693 Length:2693
##  1st Qu.:2003 Class  :character   Class  :character Class  :character
##  Median :2007 Mode   :character   Mode   :character Mode   :character
```

```
##  Mean    :2007
##  3rd Qu.:2011
##  Max.   :2015
##  Raw_Guest_List
##  Length:2693
##  Class  :character
##  Mode   :character
##
##
##
```

To extract data from a dataframe, you can use some functions from the `dplyr` package, `select` and `slice`. The `select` function will pull out columns, while the `slice` function will pull out rows. In this chapter, we'll talk about how to extract certain rows or columns of a dataframe by their *position* (i.e., row or column number). Later in the book, we'll talk about other ways to extract values from dataframes.

For example, if you wanted to get the first two rows of the `hp_data` dataframe, you could run:

```
library(package = "dplyr")
slice(.data = hp_data, c(1:2))
```

```
## # A tibble: 2 x 4
##   first_name last_name n_kids survived
##   <chr>      <chr>     <dbl> <lgl>
## 1 Harry      Potter      1 TRUE
## 2 Ron        Weasley     7 TRUE
```

If you wanted to get the first and fourth columns, you could run:

```
select(.data = hp_data, c(1, 4))
```

```
## # A tibble: 3 x 2
##   first_name survived
##   <chr>      <lgl>
## 1 Harry      TRUE
## 2 Ron        TRUE
## 3 Hermione  TRUE
```

You can compose calls from both functions. For example, you could extract the values in the first and fourth columns of the first two rows with:

```
select(.data = slice(.data = hp_data, c(1:2)), c(1, 4))
```

```
## # A tibble: 2 x 2
##   first_name survived
##   <chr>      <lgl>
## 1 Harry      TRUE
## 2 Ron        TRUE
```

You can use square-bracket indexing (`[..., ...]`) for dataframes, too, but now they'll have two dimensions (rows, then columns). Put the rows you want before the comma, the columns after. If you want all of something (e.g., all rows in the dataframe), leave the designated spot blank. Here are two examples of using square-bracket indexing to pull a subset of the `hp_data` dataframe we created above:

```
hp_data[1:2, 2] # First two rows, second column
```

```
## # A tibble: 2 x 1
##   last_name
##   <chr>
## 1 Potter
## 2 Weasley
```

```
hp_data[3, ] # Last row, all columns
```

```
## # A tibble: 1 x 4
##   first_name last_name n_kids survived
##   <chr>      <chr>     <dbl> <lgl>
## 1 Hermione   Granger       1 TRUE
```



If you forget to put the comma in the indexing for a dataframe (e.g., `fibonacci_seq[1:2]`), you will index out the *columns* that fall at that position or positions. To avoid confusion, I suggest that you always use indexing with a comma when working with dataframes.

Download a pdf of the lecture slides for this video.

1.10 In-course Exercise Chapter 1

You will take turns sharing your screens as you work through this exercise. Before you start, open your R session and use the `sample` function, with all of your group members' names, to randomly shuffle your names (revisit the in-course exercise in the "Course Overview" chapter if you need a reminder).

You should do this on only one group's computer. The order that you get from R is the order that you should take turns sharing your screen and leading the effort on coding for your group. When you are not sharing your screen, help out with suggestions, especially for general directions you could take to approach a question. (There are standards for pair programming that we'll discuss next week, and these will provide more advice on how to productively code in a group.)

1.10.1 Trying out the code in slides for first lecture videos

Have one person in your group share their screen and take the lead in typing the code or doing the other work for this part.

To start, you'll try running some simple code in R, using examples from the video lectures for Chapter 1. Take the following steps:

1. Open an R session and find the "Console" pane.
2. Go through the slides for video lectures 4 ("Function calls") and 5 ("Objects and assignments"). Find any examples of R expressions and try them out at the prompt in the console.
3. Once you've run an assignment expression, find the "Environment" pane. Check that the object name that you assigned now appears there.

1.10.2 Writing your code as an R script

While the R console is fine for initially exploring data, you should get in the habit of writing up R code in an R script for most of your data analysis projects in R.

- Open a new R script and save it to your current working directory (i.e., wherever you saved the data you downloaded for this exercise).
- Take some of the code that you wrote for this exercise. Put it in the R script. Do not put more than one function call per line (but it's fine to have longer function calls span a few lines).
- Use the "Run" button to run a single line of this code. Check the console to see what happens when you do.
- Highlight a few lines of the code and use "Run" to run them.

- Try using the keyboard shortcut (Command-Return) to run the line of code your cursor is currently on. Try doing this with a function call that runs across several lines of the R script file— what do you see at the console?
- Try running the whole script using “Source”. Again, look at the console after you “source” the script.
- Close your R session (and save any changes to your R script). Do **not** save your R session history. Re-open R and see if you can re-open your R script and re-run it. Try using `ls()` to list the objects in your R session before and after you re-run your script. Does anything about the result surprise you?

1.10.3 About the dataset

Trade the screen sharing to the next member of your group.

For the rest of today’s class, you’ll be using a dataset of all the guests on *The Daily Show* when Jon Stewart was the host. This data was originally collected by Nate Silver’s website, FiveThirtyEight and is available on FiveThirtyEight’s GitHub page under the Creative Commons Attribution 4.0 International License. I have copied this data into my GitHub repository for this class. The only change made to the original file was to add (commented) attribution information at the start of the file.

First, check out a bit more about this data and its source:

- It’s often helpful to use prior knowledge to help check out or validate your dataset. One thing we might want to know about this data is if it covers the whole time that Jon Stewart hosted *The Daily Show*. Use Google to find out the dates he started and finished as host.
- Briefly browse around FiveThirtyEight’s GitHub data page. What are some other datasets available that you find interesting? For any dataset, you can scroll to the bottom of the page to get to the compiled README.md content, which gives the full titles and links to relevant datasets. You can also click on any dataset to get more information.
- Look at the GitHub page for this *Daily Show* data. How many columns will be in this dataset? What kind of information does the data include? What do the columns show? What do the rows show?



In this exercise, you’re using data posted by FiveThirtyEight on GitHub. We’ll be using a lot of data that’s on GitHub this semester, and GitHub is being used behind-the-scenes for both this book and the course note slides. We’ll talk more about GitHub later, but you might find it interesting to explore a bit now. It’s a place where people can post, work on, and

share code in a number of programming languages— it’s been referred to as “Facebook for Nerds”. You can search GitHub repositories and code specifically by programming language, so it can be a good way to find example R code from which to learn.

1.10.4 Manually creating vectors

Start by manually creating some vectors and data frames with a small subset of this data.

- Use the concatenate function (`c`) to create a vector “from scratch” with the names of the five guests to appear on the show (these could be the first five guests, or you could also randomly pick five guests). Assign this vector the object name `five_guests`. What class (numeric or character) do you think this vector will be? Will you need to use quotation marks for each element you add to the vector?
- Use square bracket indexing to print out the following subsets of this vector (you’ll have one R expression per subset): (1) The first guest in the vector; (2) The third and fifth guests; (3) The second through fourth guests.
- Create a new vector called `first_guest` with just the first guest in the vector, using the square bracket indexing you used in the previous step.
- In the same way, create a vector with the year of each of these five guests’ appearances. Assign this vector to an object named `appearance_year`. What class (numeric or character) do you think this vector will be? Will you need to use quotation marks for each element you add to the vector?
- Use the `class` function to determine the classes (e.g., numeric, character) of each of the vectors you just created.

Example R code:

```
# I picked five random guests from throughout the dataset. The guests you pick will
# likely be different.

# Create a vector with the names of five guests
five_guests <- c("Miss Piggy", "Stanley Tucci", "Kermit the Frog",
                 "Hank Azaria", "Al Gore")

# Use square-bracket indexing to print out some subsets of the data
five_guests[1]

## [1] "Miss Piggy"
```

```
five_guests[c(3, 5)]
```

```
## [1] "Kermit the Frog" "Al Gore"
```

```
five_guests[2:4]
```

```
## [1] "Stanley Tucci"    "Kermit the Frog" "Hank Azaria"
```

```
# Save just the first guest in a separate object
first_guest <- five_guests[1]
first_guest
```

```
## [1] "Miss Piggy"
```

```
# Create a vector with the year of the appearance of each guest
appearance_year <- c(1999, 2000, 2001, 2001, 2002)
```

```
# Figure out the classes of the two vectors you just created
class(x = five_guests)
```

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

```
class(x = appearance_year)
```

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

1.10.5 Installing and using a package

Trade the screen sharing to the next member of your group. Have the person who was sharing their screen save their R script and send it to this person through the Zoom chat. The new person should open this R script and use it to

re-run the last part of the analysis, so that the vectors defined in the last part of the exercise can be used here.

The `stringr` package includes a number of functions that make it easier to work with character strings in R. In particular, it includes functions to change the capitalization of words in character strings. Here, you'll install and load this package and then use it to work with the `five_guests` vector we created in the last section.

- If you have not already installed the `stringr` package, install it from CRAN.
- Load the `stringr` package in your current R session, so you will be able to use its functions.
- Check if the package has a vignette. If so, check out that vignette.
- See if you can use the `str_to_lower` function from the `stringr` package to convert all the names in your `five_guests` vector to lowercase.
- See if you can find a function in the `stringr` package that you can use to convert all the names in your `five_guests` vector to uppercase. (Hint: At the R console, try typing `?stringr::` and then the Tab key.)

Example R code:

```
# If you need to, install the package from CRAN
install.packages(pkgs = "stringr")
```

```
# Load the package into your current R session
library(package = "stringr")
```

```
# Open the package's vignette
vignette(topic = "stringr")
```

```
# Convert the `five_guests` strings to lowercase
str_to_lower(string = five_guests)
```

```
## [1] "miss piggy"      "stanley tucci"    "kermit the frog" "hank azaria"
## [5] "al gore"
```

```
# Convert the `five_guests` strings to uppercase
str_to_upper(string = five_guests)

## [1] "MISS PIGGY"      "STANLEY TUCCI"    "KERMIT THE FROG" "HANK AZARIA"
## [5] "AL GORE"
```

1.10.6 Manually creating a dataframe

- Combine the two vectors you created earlier, `five_guests` and `appearance_year` to create a dataframe named `guest_list`. For the columns, use the same column names used in the original, raw data for the guest names and appearance year. Print out this dataframe at the R console to make sure it looks like you thought it would.
- Use functions from the `dplyr` package to print out the following subsets of this dataframe (you'll have one R call per subset): (1) The appearance year of the first guest; (2) Names of the third through fifth guests; (3) Names of all guests; (4) Both names and appearance years of the first and third guests.
- The `str` function can be used to figure out the structure of a dataframe. Run this command on the `guest_list` dataframe you created. What information does this give you? Use the helpfile for `str` to help you figure this out (which you can access by running `?str`). Do you see anything that surprises you?
- Use the `ls` function to list all the objects you currently have defined in your R session. Compare this list to the “Environment” pane in RStudio.

Example R code:

```
# Create the data frame, then print it out to make sure it looks like you thought
# it would
library(package = "tibble")
guest_list <- tibble(Raw_Guest_List = five_guests,
                     YEAR = appearance_year)
guest_list

## # A tibble: 5 x 2
##   Raw_Guest_List     YEAR
##   <chr>              <dbl>
## 1 Miss Piggy          1999
## 2 Stanley Tucci       2000
## 3 Kermit the Frog     2001
```

```
## 4 Hank Azaria      2001  
## 5 Al Gore         2002
```

```
# Use functions from the dplyr package to extract values from the dataframe  
library(package = "dplyr")  
slice(.data = select(.data = guest_list, 2), 1)
```

```
## # A tibble: 1 x 1  
##   YEAR  
##   <dbl>  
## 1 1999
```

```
slice(.data = select(.data = guest_list, 1), 3:5)
```

```
## # A tibble: 3 x 1  
##   Raw_Guest_List  
##   <chr>  
## 1 Kermit the Frog  
## 2 Hank Azaria  
## 3 Al Gore
```

```
select(.data = guest_list, 1)
```

```
## # A tibble: 5 x 1  
##   Raw_Guest_List  
##   <chr>  
## 1 Miss Piggy  
## 2 Stanley Tucci  
## 3 Kermit the Frog  
## 4 Hank Azaria  
## 5 Al Gore
```

```
slice(.data = guest_list, c(1, 3))
```

```
## # A tibble: 2 x 2
##   Raw_Guest_List    YEAR
##   <chr>              <dbl>
## 1 Miss Piggy        1999
## 2 Kermit the Frog  2001
```

```
# Use `str` to check out the structure of the data frame you created
str(guest_list)
```

```
## tibble [5 x 2] (S3:tbl_df/tbl/data.frame)
## $ Raw_Guest_List: chr [1:5] "Miss Piggy" "Stanley Tucci" "Kermit the Frog" "Hank Azaria" ...
## $ YEAR          : num [1:5] 1999 2000 2001 2001 2002
```

1.10.7 Getting the data onto your computer

Next, we will work with the whole dataset. Download the data from GitHub onto your computer. It is **very important** for you to use this link rather than downloading the data from the FiveThirtyEight GitHub page, because there's a small difference between the two files.

In class, we created an R Project for you to use for this class. Put the *Daily Show* data in that directory.

Take the following steps to get the data onto your computer

- Download the file from GitHub. Right click on `Raw` and then choose “Download linked file”. Put the file into the directory you created for this course.
- Use the `list.files` command to make sure that the “`daily_show_guests.csv`” file is in your current working directory (we’ll talk more about working directories, listing files in your working directory, and R Projects later in the semester).

```
# List the files in your current working directory
list.files()
```

```
[1] "daily_show_guests.csv"
```

1.10.8 Getting the data into R

Now that you have the dataset in your working directory, you can read it into R. This dataset is in a *csv* (comma separated values) format. (We will talk more about different file formats in Week 2.) You can read csv files into R using the function `read_csv` from the `readr` package.

Read the data into your R session

- If you do not already have it, install the `readr` package. Then load this package within your current R session using `library`.
- Use the `read_csv` function from the `readr` package to read the data into R and save it as the object `daily_show` (see tips in the next few bullets).
- Use the help file for the `read_csv` function to figure out how this function works. To pull that up, type `?read_csv` at the R console. Can you figure out why it's critical to use the `skip` option and set it to 4? (We will be talking a lot more about the `read_csv` function in Week 2, so don't worry if you don't completely understand it right now.)
- Note that you need to put the file name in quotation marks.
- What would have happened if you'd used `read_csv` but hadn't saved the result as the object `daily_show`? (For example, you'd run the code `read_csv("daily_show_guests.csv", skip = 4)` rather than `daily_show <- read_csv("daily_show_guests.csv")`.)

Example R code:

```
# Install (if needed) and load the `readr` package
install.packages(pkgs = "readr") # You only need to do this if you
                                # do not already have the `readr`#
                                # package.
library(package = "readr")

# Read in dataframe from the csv file with Daily Show guests
daily_show <- read_csv(file = "daily_show_guests.csv", skip = 4)
```

```
## Rows: 2693 Columns: 5
## -- Column specification -----
## Delimiter: ","
## chr (4): GoogleKnowlege_Occupation, Show, Group, Raw_Guest_List
## dbl (1): YEAR
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# Print out the first few rows
daily_show
```

```
## # A tibble: 2,693 x 5
##   YEAR GoogleKnowlege_Occupation Show   Group Raw_Guest_List
##   <dbl> <chr>                  <chr>  <chr>  <chr>
## 1 1999 actor                 1/11/99 Acting Michael J. Fox
## 2 1999 Comedian              1/12/99 Comedy Sandra Bernhard
## 3 1999 television actress  1/13/99 Acting Tracey Ullman
## 4 1999 film actress         1/14/99 Acting Gillian Anderson
## 5 1999 actor                 1/18/99 Acting David Alan Grier
## 6 1999 actor                 1/19/99 Acting William Baldwin
## 7 1999 Singer-lyricist      1/20/99 Musician Michael Stipe
## 8 1999 model                 1/21/99 Media Carmen Electra
## 9 1999 actor                 1/25/99 Acting Matthew Lillard
## 10 1999 stand-up comedian  1/26/99 Comedy David Cross
## # i 2,683 more rows
```

If you have extra time:

- Say this was a really big dataset. You want to check out just the first 10 rows to make sure that you've got your code right before you take the time to pull in the whole dataset. Use the help file for `read_csv` to figure out how to only read in a few rows.
- Look through the help file for other options available for `read_csv`. Can you think of examples when some of these options would be useful?
- Look again at the version of this raw data on FiveThirtyEight's GitHub page (rather than the course's GitHub repository, where you downloaded the data for the course exercise). How are these two versions of the raw data different? How would you need to change your `read_csv` call if you changed to use the FiveThirtyEight version of the raw data?

Example R code:

```
# Read in only the first 10 rows of the dataset
daily_show_first10 <- read_csv(file = "daily_show_guests.csv",
                                skip = 4, n_max = 10)
```

```
## Rows: 10 Columns: 5
## -- Column specification -----
## Delimiter: ","
```

```
## chr (4): GoogleKnowlege_Occupation, Show, Group, Raw_Guest_List
## dbl (1): YEAR
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# Check the dataframe
daily_show_first10
```

```
## # A tibble: 10 x 5
##   YEAR GoogleKnowlege_Occupation Show   Group   Raw_Guest_List
##   <dbl> <chr>                <chr>  <chr>  <chr>
## 1 1999 actor                 1/11/99 Acting Michael J. Fox
## 2 1999 Comedian              1/12/99 Comedy Sandra Bernhard
## 3 1999 television actress  1/13/99 Acting Tracey Ullman
## 4 1999 film actress         1/14/99 Acting Gillian Anderson
## 5 1999 actor                 1/18/99 Acting David Alan Grier
## 6 1999 actor                 1/19/99 Acting William Baldwin
## 7 1999 Singer-lyricist      1/20/99 Musician Michael Stipe
## 8 1999 model                  1/21/99 Media Carmen Electra
## 9 1999 actor                 1/25/99 Acting Matthew Lillard
## 10 1999 stand-up comedian   1/26/99 Comedy David Cross
```

1.10.9 Checking out the data

Trade who is sharing their screen again. The new coder will need to download the data file fresh and move it into a “data” subdirectory of the R project created at the start of the class meeting. The previous coder should save and share his or her’s R script and send that to the new person by Zoom. The new person should start by running that code and making sure everything’s working well on their computer.

You now have the data available in your current R session as the `daily_show` object. You’ll want to check it out to make sure it read in correctly, and also to get a feel for the data. Throughout, you can use the help pages to figure out more about any of the functions being used (for example, `?dim`).

Take the following steps to check out the dataset

- Use the `dim` function to find out how many rows and columns this dataframe has. Based on what you found out about the data from the GitHub page, does it have the number of columns you expected? Based on what you know about the data (that it includes all the guests who

came on The Daily Show with Jon Stewart), do you think it has about the right number of rows?

- Use functions from the `dplyr` package to look at the first two rows of the dataset. Based on this, what does each row “measure” (**unit of observation**)? What information (**variables**) do you get for each “measurement”?
- The `head` function can be used to explore the first few rows of dataframes (see the helpfile at `?head`). Use the `head` function to look at the first few rows of the dataframe. Does it look like the rows go in order by date? What was the date of Jon Stewart’s first show? Does it look like this dataset covers that first show?
- Use the `tail` function to look at the last few rows of the dataframe. What is the last show date covered by the dataframe? Who was the last guest?

Example R code:

```
# Extract values from the dataframe
library(package = "dplyr") # Load the 'dplyr' package
slice(.data = daily_show, 1:2) # Look at the first two rows of data
```

```
## # A tibble: 2 x 5
##   YEAR GoogleKnowlege_Occupation Show   Group Raw_Guest_List
##   <dbl> <chr>                  <chr>  <chr>  <chr>
## 1 1999 actor                  1/11/99 Acting Michael J. Fox
## 2 1999 Comedian                1/12/99 Comedy Sandra Bernhard
```

```
# Check the dimensions of the data
dim(x = daily_show)
```

```
## [1] 2693     5
```

```
head(x = daily_show)
```

```
## # A tibble: 6 x 5
##   YEAR GoogleKnowlege_Occupation Show   Group Raw_Guest_List
##   <dbl> <chr>                  <chr>  <chr>  <chr>
## 1 1999 actor                  1/11/99 Acting Michael J. Fox
## 2 1999 Comedian                1/12/99 Comedy Sandra Bernhard
## 3 1999 television actress    1/13/99 Acting Tracey Ullman
## 4 1999 film actress           1/14/99 Acting Gillian Anderson
## 5 1999 actor                  1/18/99 Acting David Alan Grier
## 6 1999 actor                  1/19/99 Acting William Baldwin
```

```
tail(x = daily_show)

## # A tibble: 6 x 5
##   YEAR GoogleKnowlege_Occupation Show     Group  Raw_Guest_List
##   <dbl> <chr>                  <chr>    <chr>  <chr>
## 1 2015 actor                  7/28/15 Acting Tom Cruise
## 2 2015 biographer              7/29/15 Media  Doris Kearns Goodwin
## 3 2015 director                7/30/15 Media  J. J. Abrams
## 4 2015 stand-up comedian      8/3/15 Comedy Amy Schumer
## 5 2015 actor                  8/4/15 Acting Denis Leary
## 6 2015 comedian               8/5/15 Comedy Louis C.K.
```

If you have extra time:

- Say you wanted to look at the first ten rows of the dataframe, rather than the first six. How could you use an option with `head` to do this?

Example R code:

```
# Look at the first few rows of the data
head(x = daily_show, n = 10)
```

```
## # A tibble: 10 x 5
##   YEAR GoogleKnowlege_Occupation Show     Group  Raw_Guest_List
##   <dbl> <chr>                  <chr>    <chr>  <chr>
## 1 1999 actor                  1/11/99 Acting Michael J. Fox
## 2 1999 Comedian               1/12/99 Comedy Sandra Bernhard
## 3 1999 television actress    1/13/99 Acting Tracey Ullman
## 4 1999 film actress          1/14/99 Acting Gillian Anderson
## 5 1999 actor                  1/18/99 Acting David Alan Grier
## 6 1999 actor                  1/19/99 Acting William Baldwin
## 7 1999 Singer-lyricist       1/20/99 Musician Michael Stipe
## 8 1999 model                  1/21/99 Media Carmen Electra
## 9 1999 actor                  1/25/99 Acting Matthew Lillard
## 10 1999 stand-up comedian    1/26/99 Comedy David Cross
```

1.10.10 Using the data to answer questions

Nate Silver was a guest on *The Daily Show*. Let's use this data to figure out how many times he was a guest and when he was on the show.

Find out more about Nate Silver on The Daily Show

(Don't worry if you don't make it to this sections! I've put it here for groups that move through the rest quickly.)

- The `filter` function from the `dplyr` package can be combined with logical statements to help you create a specific subset of data. For example, if you only wanted data from guest visits in 1999, you could run `filter(.data = daily_show, YEAR == 1999)`. Check out the helpfile for `filter` and use the function to create a new dataframe that only has the rows of `daily_show` when Nate Silver was a guest (`Raw_Guest_List == "Nate Silver"`). Save this as an object named `nate_silver`.
- Print out the full `nate_silver` dataframe by typing `nate_silver`. (You could just use this to answer both questions, but still try the next steps. They would be important with a bigger dataset.)
- To count the number of times Nate Silver was a guest, you'll need to count the number of rows in the new dataset. You can either use the `dim` function or the `nrow` function to do this. What additional information does the `dim` function give you?
- To get the dates when Nate Silver was a guest, you can print out just the `Show` column of the dataframe. There are a few ways you can do this using the `select` function from the `dplyr` package.

Example R code:

```
library(package = "dplyr")
# Create a subset of the data with just Nate Silver appearances
nate_silver <- filter(.data = daily_show, Raw_Guest_List == "Nate Silver")

# Investigate this subset of the data
nate_silver

## # A tibble: 3 x 5
##   YEAR GoogleKnowlege_Occupation Show      Group Raw_Guest_List
##   <dbl> <chr>                  <chr>    <chr> <chr>
## 1 2012 Statistician            10/17/12 Media   Nate Silver
## 2 2012 Statistician            11/7/12  Media   Nate Silver
## 3 2014 Statistician            3/27/14  Media   Nate Silver

dim(x = nate_silver)

## [1] 3 5
```

```
nrow(x = nate_silver)

## [1] 3

select(.data = nate_silver, 3)

## # A tibble: 3 x 1
##   Show
##   <chr>
## 1 10/17/12
## 2 11/7/12
## 3 3/27/14
```

If you have extra time:

- Was Nate Silver the only statistician to be a guest on the show?
- What were the occupations that were only represented by one guest visit? Since `GoogleKnowlege_Occupation` is a factor, you can use the `table` function to create a new vector with the number of times each value of `GoogleKnowlege_Occupation` shows up. You can put this information into a new vector and then pull out only the values that equal 1 (so, only had one guest). (Note that “Statistician” doesn’t show up—there was only one person who was a guest, but he had three visits.) Pick your favorite “one-off” example and find out who the guest was for that occupation.

Example R code:

```
statisticians <- filter(.data = daily_show,
                        GoogleKnowlege_Occupation == "Statistician")
statisticians

## # A tibble: 3 x 5
##   YEAR GoogleKnowlege_Occupation Show      Group Raw_Guest_List
##   <dbl> <chr>                  <chr>    <chr> <chr>
## 1 2012 Statistician           10/17/12 Media  Nate Silver
## 2 2012 Statistician           11/7/12  Media  Nate Silver
## 3 2014 Statistician           3/27/14  Media  Nate Silver
```

```
num_visits <- table(daily_show$GoogleKnowlege_Occupation)
head(x = num_visits) # Note: This is a vector rather than a dataframe
```

```
## 
##      -          0    academic   Academic accountant   activist
##      1          4          3          3          1         14
```

```
single_visits <- num_visits[num_visits == 1] # This is using a "logical operator" to extract values
names(single_visits)
```

```
## [1] "-"
## [2] "accountant"
## [3] "administrator"
## [4] "advocate"
## [5] "aei president"
## [6] "afghan politician"
## [7] "American football running back"
## [8] "american football wide reciever"
## [9] "assistant secretary of defense"
## [10] "assistant to the president for communications"
## [11] "Associate Justice of the Supreme Court of the United States"
## [12] "astronaut"
## [13] "Astronaut"
## [14] "Attorney at law"
## [15] "author of novels"
## [16] "aviator"
## [17] "Baseball athlete"
## [18] "baseball player"
## [19] "Basketball Coach"
## [20] "bass guitarist"
## [21] "bassist"
## [22] "Beach Volleyball Player"
## [23] "boxer"
## [24] "business person"
## [25] "businesswoman"
## [26] "Businesswoman"
## [27] "Cartoonist"
## [28] "celbrity chef"
## [29] "CHARACTER"
## [30] "chess player"
## [31] "chief technology officer of united states"
```

```
## [32] "Choreographer"
## [33] "civil rights activist"
## [34] "Coach"
## [35] "comic"
## [36] "Comic"
## [37] "communications consultant"
## [38] "Composer"
## [39] "comptroller of the us"
## [40] "coorespondant"
## [41] "Critic"
## [42] "designer"
## [43] "Director of the Consumer Financial Protection Bureau"
## [44] "doctor"
## [45] "drummer"
## [46] "Educator"
## [47] "entrepreneur"
## [48] "Ethologist"
## [49] "executive"
## [50] "Executive"
## [51] "fbi agent"
## [52] "Fiction writer"
## [53] "Film-maker"
## [54] "Film critic"
## [55] "film producer"
## [56] "Financier"
## [57] "first lady"
## [58] "first lady of egypt"
## [59] "First Lady of the United States"
## [60] "First Minister of Scotland"
## [61] "Football coach"
## [62] "football player"
## [63] "foreign policy analyst"
## [64] "foreign policy expert"
## [65] "foreign policy strategist"
## [66] "Former American senator"
## [67] "former british prime minister"
## [68] "former cia director"
## [69] "former director of the national economic counscil"
## [70] "Former Director of the Office of Management and Budget"
## [71] "Former First Lady of the United States"
## [72] "former governor of arizona"
## [73] "former governor of arkansas"
## [74] "former governor of california"
## [75] "Former Governor of Indiana"
## [76] "former governor of louisiana"
## [77] "former governor of massachusetts"
```

```
## [78] "former governor of michigan"
## [79] "former governor of missouri"
## [80] "former governor of montans"
## [81] "former governor of new hampshire"
## [82] "Former Governor of New Jersey"
## [83] "Former Governor of New York"
## [84] "former governor of rhode island"
## [85] "Former Governor of Texas"
## [86] "former governor of washington"
## [87] "former govrnor of masssachusetts"
## [88] "Former Mayor of Cincinnati"
## [89] "Former Mayor of New Orleans"
## [90] "former mayor of san antonio"
## [91] "Former member of the United States Senate"
## [92] "former mjority leader"
## [93] "former national security advisio\\r"
## [94] "former omb director"
## [95] "Former President of Mexico"
## [96] "Former President of the Maldives"
## [97] "former press secretary"
## [98] "former secretary of defense"
## [99] "former senator"
## [100] "former senator from kansas"
## [101] "Former United States Deputy Secretary of State"
## [102] "Former United States National Security Advisor"
## [103] "Former United States Secretary of Education"
## [104] "Former United States Secretary of Energy"
## [105] "Former United States Secretary of the Interior"
## [106] "Former United States Secretary of the Treasury"
## [107] "Former United States Secretary of Transportation"
## [108] "former us representativ"
## [109] "former us secretary of education"
## [110] "former white house counsel"
## [111] "Futurist"
## [112] "game show host"
## [113] "Geneticist"
## [114] "governor of new jersey"
## [115] "guitarist"
## [116] "high-altitude mountaineer"
## [117] "Host"
## [118] "Ice hockey coach"
## [119] "illustrator"
## [120] "Innovator"
## [121] "inspector general of homeland security department"
## [122] "intellectual"
## [123] "internet entrepreneur"
```

```
## [124] "investment banker"
## [125] "israeli official"
## [126] "JOURNALIST"
## [127] "Law professor"
## [128] "legal scholar"
## [129] "magician"
## [130] "mathematician"
## [131] "Mayor of Chicago"
## [132] "mayor of london"
## [133] "Media person"
## [134] "minister of defense"
## [135] "Music Producer"
## [136] "Neurologist"
## [137] "Neuroscientist"
## [138] "non profit director"
## [139] "non profit worker"
## [140] "orca trainer"
## [141] "pastor"
## [142] "peace activist"
## [143] "photojournalist"
## [144] "Photojournalist"
## [145] "physicist"
## [146] "pianist"
## [147] "police officer"
## [148] "political consultant"
## [149] "political expert"
## [150] "Political figure"
## [151] "political psychologist"
## [152] "political satirist"
## [153] "political strategist"
## [154] "Pop group"
## [155] "president of liberia"
## [156] "priest"
## [157] "prince"
## [158] "Product line"
## [159] "professional wrestler"
## [160] "psychic"
## [161] "Psychologist"
## [162] "public official"
## [163] "public speaker"
## [164] "publisher"
## [165] "Pundit"
## [166] "Puppeteer"
## [167] "Puzzle Creator"
## [168] "race car driver"
## [169] "Racing driver"
```

```

## [170] "reality show contestant"
## [171] "RNC chairman"
## [172] "Scholar"
## [173] "secretary of state"
## [174] "security expert"
## [175] "Soccer player"
## [176] "social activist"
## [177] "speechwriter"
## [178] "Sports Columnist"
## [179] "Surgeon"
## [180] "swimmer"
## [181] "syrian politician"
## [182] "television actor"
## [183] "television Director"
## [184] "television writer"
## [185] "televison actor"
## [186] "telvision actor"
## [187] "telvision personality"
## [188] "Tennis player"
## [189] "Track and field athlete"
## [190] "TV Producer"
## [191] "united nations official"
## [192] "United States Secretary of Agriculture"
## [193] "United States Secretary of Defense"
## [194] "United States Secretary of Housing and Urban Development"
## [195] "United States Secretary of the Navy"
## [196] "us assistant attorney"
## [197] "us official"
## [198] "us permanent representative to nato"
## [199] "us secetary of education"
## [200] "us secretary of defense"
## [201] "us secretary of energy"
## [202] "white house official"

```

```
filter(.data = daily_show, GoogleKnowlege_Occupation == "chess player")
```

```

## # A tibble: 1 x 5
##   YEAR GoogleKnowlege_Occupation Show   Group Raw_Guest_List
##   <dbl> <chr>                <chr>  <chr> <chr> <chr>
## 1 2012 chess player           11/8/12 Misc   Katie Dellamaggiore and Pobo Ef~

```

```
filter(.data = daily_show, GoogleKnowlege_Occupation == "mathematician")  
  
## # A tibble: 1 x 5  
##   YEAR GoogleKnowlege_Occupation Show     Group      Raw_Guest_List  
##   <dbl> <chr>                  <chr>    <chr>      <chr>  
## 1 2005 mathematician           9/14/05 Academic Dr. William A. Dembski
```

```
filter(.data = daily_show, GoogleKnowlege_Occupation == "orca trainer")
```

```
## # A tibble: 1 x 5  
##   YEAR GoogleKnowlege_Occupation Show     Group      Raw_Guest_List  
##   <dbl> <chr>                  <chr>    <chr>      <chr>  
## 1 2015 orca trainer            3/26/15 Athletics John Hargrove
```

```
filter(.data = daily_show, GoogleKnowlege_Occupation == "Puzzle Creator")
```

```
## # A tibble: 1 x 5  
##   YEAR GoogleKnowlege_Occupation Show     Group      Raw_Guest_List  
##   <dbl> <chr>                  <chr>    <chr>      <chr>  
## 1 2003 Puzzle Creator          8/20/03 Media Will Shortz
```

```
filter(.data = daily_show, GoogleKnowlege_Occupation == "Scholar")
```

```
## # A tibble: 1 x 5  
##   YEAR GoogleKnowlege_Occupation Show     Group      Raw_Guest_List  
##   <dbl> <chr>                  <chr>    <chr>      <chr>  
## 1 2005 Scholar                6/13/05 Academic Larry Diamond
```

Part II

Part II: Basics

Chapter 2

Entering and cleaning data #1

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

2.1 Objectives

After this chapter, you should (know / understand / be able to):

- Understand what a flat file is and how it differs from data stored in a binary file format
- Be able to distinguish between delimited and fixed width formats for flat files
- Be able to identify the delimiter in a delimited file
- Be able to describe a working directory
- Be able to read in different types of flat files
- Be able to read in a few types of binary files (SAS, Excel)
- Understand the difference between relative and absolute file pathnames
- Describe the basics of your computer's directory structure
- Reference files in different locations in your directory structure using relative and absolute pathnames
- Use the basic `dplyr` functions `rename`, `select`, `mutate`, `slice`, `filter`, and `arrange` to work with data in a dataframe object
- Convert a column to a date format using `lubridate` functions
- Extract information from a date object (e.g., month, year, day of week) using `lubridate` functions

- Define a logical operator and know the R syntax for common logical operators
- Use logical operators in conjunction with `dplyr`'s `filter` function to create subsets of a dataframe based on logical conditions
- Use piping to apply multiple `dplyr` functions in sequence to a dataframe

2.2 Overview

Download a pdf of the lecture slides for this video.

There are four basic steps you will often repeat as you prepare to analyze data in R:

1. Identify where the data is (If it's on your computer, which directory? If it's online, what's the url?)
2. Read data into R (e.g., `read_delim`, `read_csv` from the `readr` package) using the file path you figured out in step 1
3. Check to make sure the data came in correctly (`dim`, `head`, `tail`, `str`)
4. Clean the data up

In this chapter, I'll go basics for each of these steps, as well as dive a bit deeper into some related topics you should learn now to make your life easier as you get started using R for research.

2.3 Reading data into R

Data comes in files of all shapes and sizes. R has the capability to read data in from many of these, even proprietary files for other software (e.g., Excel and SAS files). As a small sample, here are some of the types of data files that R can read and work with:

- Flat files (much more about these in just a minute)
- Files from other statistical packages (SAS, Excel, Stata, SPSS)
- Tables on webpages (e.g., the table on ebola outbreaks near the end of this Wikipedia page)
- Data in a database (e.g., MySQL, Oracle)
- Data in JSON and XML formats
- Really crazy data formats used in other disciplines (e.g., netCDF files from climate research, MRI data stored in Analyze, NIfTI, and DICOM formats)
- Geographic shapefiles
- Data through APIs

Often, it is possible to read in and clean up even incredibly messy data, by using functions like `scan` and `readLines` to read the data in a line at a time, and then using regular expressions (which I'll cover in the "Intermediate" section of the course) to clean up each line as it comes in. In over a decade of coding in R, I think the only time I've come across a data file I couldn't get into R was for proprietary precision agriculture data collected at harvest by a combine.

2.3.1 Reading local flat files

Much of the data that you will want to read in will be in flat files. Basically, these are files that you can open using a text editor; the most common type you'll work with are probably comma-separated files (often with a `.csv` or `.txt` file extension). Most flat files come in two general categories:

1. Fixed width files; and
2. Delimited files:
 - ".csv": Comma-separated values
 - ".tab", ".tsv": Tab-separated values
 - Other possible delimiters: colon, semicolon, pipe ("|")

Fixed width files are files where a column always has the same width, for all the rows in the column. These tend to look very neat and easy-to-read when you open them in a text editor. For example, the first few rows of a fixed-width file might look like this:

Course	Number	Day	Time
Intro to Epi	501	M/W/F	9:00-9:50
Advanced Epi	521	T/Th	1:00-2:15

Fixed width files used to be very popular, and they make it easier to look at data when you open the file in a text editor. However, now it's pretty rare to just use a text editor to open a file and check out the data, and these files can be a bit of a pain to read into R and other programs because you sometimes have to specify exactly how wide each of the columns is. You may come across a fixed width file every now and then, though, particularly when working with older data sets, so it's useful to be able to recognize one and to know how to read it in.

Delimited files use some *delimiter* (for example, a column or a tab) to separate each column value within a row. The first few rows of a delimited file might look like this:

```
Course, Number, Day, Time
"Intro to Epi", 501, "M/W/F", "9:00-9:50"
"Advanced Epi", 521, "T/Th", "1:00-2:15"
```

Delimited files are very easy to read into R. You just need to be able to figure out what character is used as a delimiter (commas in the example above) and specify that to R in the function call to read in the data.

These flat files can have a number of different file extensions. The most generic is **.txt**, but they will also have ones more specific to their format, like **.csv** for a comma-delimited file or **.fwf** for a fixed width file.

Download a pdf of the lecture slides for this video.

R can read in data from both fixed width and delimited flat files. The only catch is that you need to tell R a bit more about the format of the flat file, including whether it is fixed width or delimited. If the file is fixed width, you will usually have to tell R the width of each column. If the file is delimited, you'll need to tell R which delimiter is being used.

If the file is delimited, you can use the **read_delim** family of functions from the **readr** package to read it in. This family of functions includes several specialized functions. All members of the **read_delim** family are doing the same basic thing. The only difference is what defaults each function has for the delimiter (**delim**). Members of the **read_delim** family include:

Function	Delimiter
read_csv	comma
read_csv2	semi-colon
read_table2	whitespace
read_tsv	tab

You can use **read_delim** to read in any delimited file, regardless of the delimiter. However, you will need to specify the delimiter using the **delim** parameter. If you remember the more specialized function call (e.g., **read_csv** for a comma delimited file), therefore, you can save yourself some typing.

For example, to read in the Ebola data, which is comma-delimited, you could either use **read_table** with a **delim** argument specified or use **read_csv**, in which case you don't have to specify **delim**:

```
library(package = "readr")

# The following two calls do the same thing
ebola <- read_delim(file = "data/country_timeseries.csv", delim = ",")
```

```
ebola <- read_csv(file = "data/country_timeseries.csv")

## Rows: 122 Columns: 18
## -- Column specification -----
## Delimiter: ","
## chr (1): Date
## dbl (17): Day, Cases_Guinea, Cases_Liberia, Cases_SierraLeone, Cases_Nigeria...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```



The message that R prints after this call (“Parsed with column specification...”) lets you know what classes were used for each column (this function tries to guess the appropriate class and typically gets it right). You can suppress the message using the `cols_types = cols()` argument.

If `readr` doesn’t correctly guess some of the columns classes you can use the `type_convert()` function to take another go at guessing them after you’ve tweaked the formats of the rogue columns.

This family of functions has a few other helpful options you can specify. For example, if you want to skip the first few lines of a file before you start reading in the data, you can use `skip` to set the number of lines to skip. If you only want to read in a few lines of the data, you can use the `n_max` option. For example, if you have a really long file, and you want to save time by only reading in the first ten lines as you figure out what other options to use in `read_delim` for that file, you could include the option `n_max = 10` in the `read_delim` call. Here is a table of some of the most useful options common to the `read_delim` family of functions:

Option	Description
<code>skip</code>	How many lines of the start of the file should you skip?
<code>col_names</code>	What would you like to use as the column names?
<code>col_types</code>	What would you like to use as the column types?
<code>n_max</code>	How many rows do you want to read in?
<code>na</code>	How are missing values coded?



Remember that you can always find out more about a function by looking at its help file. For example, check out `?read_delim` and `?read_fwf`. You can also use the help files to determine the default values of arguments for each function.

So far, we've only looked at functions from the `readr` package for reading in data files. There is a similar family of functions available in base R, the `read.table` family of functions. The `readr` family of functions is very similar to the base R `read.table` functions, but have some more sensible defaults. Compared to the `read.table` family of functions, the `readr` functions:

- Work better with large datasets: faster, includes progress bar
- Have more sensible defaults (e.g., characters default to characters, not factors)

I recommend that you always use the `readr` functions rather than their base R alternatives, given these advantages. However, you are likely to come across code that someone else has written that uses one of these base R functions, so it's helpful to know what they are. Functions in the `read.table` family include:

- `read.csv`
- `read.delim`
- `read.table`
- `read.fwf`



The `readr` package is a member of the *tidyverse* of packages. The *tidyverse* describes an evolving collection of R packages with a common philosophy, and they are unquestionably changing the way people code in R. Many were developed in part or full by Hadley Wickham and others at RStudio. Many of these packages are less than ten years old, but have been rapidly adapted by the R community. As a result, newer examples of R code will often look very different from the code in older R scripts, including examples in books that are more than a few years old. In this course, I'll focus on "tidyverse" functions when possible, but I do put in details about base R equivalent functions or processes at some points—this will help you interpret older code. You can download all the tidyverse packages using `install.packages("tidyverse")`, `library("tidyverse")` makes all the tidyverse functions available for use.

2.3.2 Reading in other file types

Later in the course, we'll talk about how to open a variety of other file types in R. However, you might find it immediately useful to be able to read in files from other statistical programs.

There are two “tidyverse” packages—`readxl` and `haven`—that help with this. They allow you to read in files from the following formats:

File type	Function	Package
Excel	<code>'read_excel'</code>	<code>'readxl'</code>
SAS	<code>'read_sas'</code>	<code>'haven'</code>
SPSS	<code>'read_spss'</code>	<code>'haven'</code>
Stata	<code>'read_stata'</code>	<code>'haven'</code>

2.4 Directories and pathnames

2.4.1 Directory structure

Download a pdf of the lecture slides for this video.

So far, we've only looked at reading in files that are located in your current working directory. For example, if you're working in an R Project, by default the project will open with that directory as the working directory, so you can read files that are saved in that project's main directory using only the file name as a reference.

However, you'll often want to read in files that are located somewhere else on your computer, or even files that are saved on another computer (for example, data files that are posted online). Doing this is very similar to reading in a file that is in your current working directory; the only difference is that you need to give R some directions so it can find the file.

The most common case will be reading in files in a subdirectory of your current working directory. For example, you may have created a “data” subdirectory in one of your R Projects directories to keep all the project's data files in the same place while keeping the structure of the main directory fairly clean. In this case, you'll need to direct R into that subdirectory when you want to read one of those files.

To understand how to give R these directions, you need to have some understanding of the directory structure of your computer. It seems a bit of a pain and a bit complex to have to think about computer directory structure in the “basics” part of this class, but this structure is not terribly complex once you get the idea of it. There are a couple of very good reasons why it's worth learning now.

First, many of the most frustrating errors you get when you start using R trace back to understanding directories and filepaths. For example, when you try to read a file into R using only the filename, and that file is not in your current working directory, you will get an error like:

```
Error in file(file, "rt") : cannot open the connection  
In addition: Warning message:  
In file(file, "rt") : cannot open file 'Ex.csv': No such file or directory
```

This error is especially frustrating when you’re new to R because it happens at the very beginning of your analysis—you can’t even get your data in. Also, if you don’t understand a bit about working directories and how R looks for the file you’re asking it to find, you’d have no idea where to start to fix this error. Second, once you understand how to use pathnames, especially relative pathnames, to tell R how to find a file that is in a directory other than your working directory, you will be able to organize all of your files for a project in a much cleaner way. For example, you can create a directory for your project, then create one subdirectory to store all of your R scripts, and another to store all of your data, and so on. This can help you keep very complex projects more structured and easier to navigate. We’ll talk about these ideas more in the course sections on Reproducible Research, but it’s good to start learning how directory structures and filepaths work early.

Your computer organizes files through a collection of directories. Chances are, you are fairly used to working with these in your daily life already (although you may call them “folders” rather than “directories”). For example, you’ve probably created new directories to store data files and Word documents for a specific project.

Figure 2.1 gives an example file directory structure for a hypothetical computer. Directories are shown in blue, and files in green.

You can notice a few interesting things from Figure 2.1. First, you might notice the structure includes a few of the directories that you use a lot on your own computer, like **Desktop**, **Documents**, and **Downloads**. Next, the directory at the very top is the computer’s root directory, **/**. For a PC, the root directory might something like **C:**; for Unix and Macs, it’s usually **/**. Finally, if you look closely, you’ll notice that it’s possible to have different files in different locations of the directory structure with the same file name. For example, in the figure, there are files names **heat_mort.csv** in both the **CourseText** directory and in the **example_data** directory. These are two different files with different contents, but they can have the same name as long as they’re in different directories. This fact—that you can have files with the same name in different places—should help you appreciate how useful it is that R requires you to give very clear directions to describe exactly which file you want R to read in, if you aren’t reading in something in your current working directory.

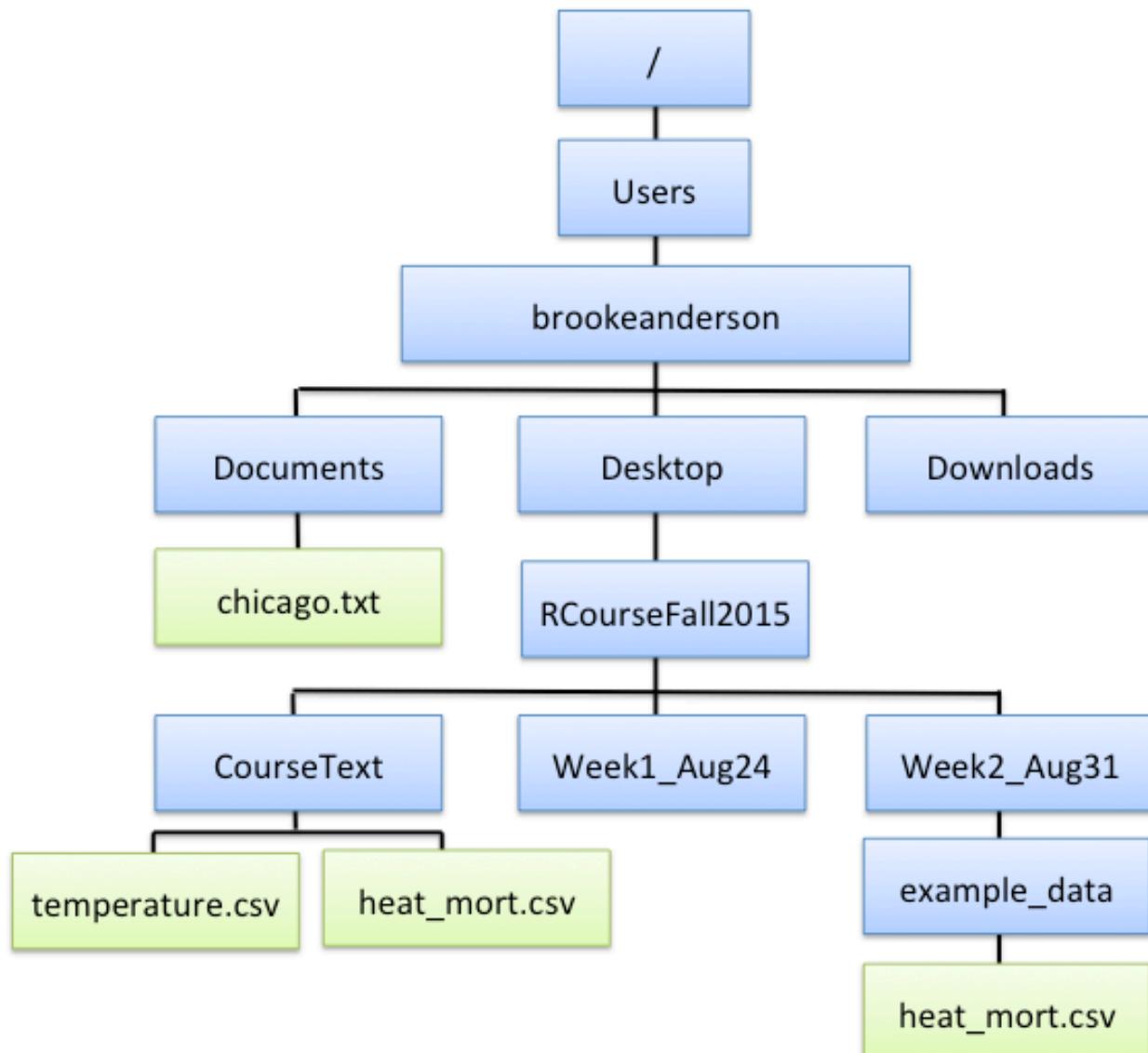


Figure 2.1: An example of file directory structure.

You will have a home directory somewhere near the top of your structure, although it's likely not your root directory. In the hypothetical computer in Figure 2.1, the home directory is `/Users/brookeanderson`. I'll describe just a bit later how you can figure out what your own home directory is on your own computer.

2.4.2 Working directory

When you run R, it's always running from within some working directory, which will be one of the directories somewhere in your computer's directory structure. At any time, you can figure out which directory R is working in by running the command `getwd()` (short for "get working directory"). For example, my R session is currently running in the following directory:

```
getwd()
```

```
## [1] "C:/Users/rseverso/Documents/RProgrammingForResearch"
```

This means that, for my current R session, R is working in the `RProgrammingForResearch` subdirectory of my `brookeanderson` directory (which is my home directory).

There are a few general rules for which working directory R will start in when you open an R session. These are not absolute rules, but they're generally true. If you have R closed, and you open it by double-clicking on an R script, then R will generally open with, as its working directory, the directory in which that script is stored. This is often a very convenient convention, because often any of the data you'll be reading in for that script is somewhere near where the script file is saved in the directory structure. If you open R by double-clicking on the R icon in "Applications" (or something similar on a PC), R will start in its default working directory. You can find out what this is, or change it, in RStudio's "Preferences". Finally, if you open an R Project, R will start in that project's working directory (the directory in which the `.Rproj` file for the project is stored).

2.4.3 File and directory pathnames

Download a pdf of the lecture slides for this video.

Once you get a picture of how your directories and files are organized, you can use pathnames, either absolute or relative, to read in files from different directories than your current working directory. Pathnames are the directions for getting to a directory or file stored on your computer.

When you want to reference a directory or file, you can use one of two types of pathnames:

- *Relative pathname*: How to get to the file or directory from your current working directory
- *Absolute pathname*: How to get to the file or directory from anywhere on the computer

Absolute pathnames are a bit more straightforward conceptually, because they don't depend on your current working directory. However, they're also a lot longer to write, and they're much less convenient if you'll be sharing some of your code with other people who might run it on their own computers. I'll explain this second point a bit more later in this section.

Absolute pathnames give the full directions to a directory or file, starting all the way at the root directory. For example, the `heat_mort.csv` file in the `CourseText` directory has the absolute pathname:

```
"./Users/brookeanderson/Desktop/RCourseFall2015/CourseText/heat_mort.csv"
```

You can use this absolute pathname to read this file in using any of the `readr` functions to read in data. This absolute pathname will *always* work, regardless of your current working directory, because it gives directions from the root—it will always be clear to R exactly what file you're talking about. Here's the code to use to read that file in using the `read.csv` function with the file's absolute pathname:

```
heat_mort <- read_csv(file = "./Users/brookeanderson/Desktop/RCourseFall2015/CourseText/heat_mort.csv")
```

The *relative pathname*, on the other hand, gives R the directions for how to get to a directory or file from the current working directory. If the file or directory you're looking for is pretty close to your current working directory in your directory structure, then a relative pathname can be a much shorter way to tell R how to get to the file than an absolute pathname. However, the relative pathname depends on your current working directory—the relative pathname that works perfectly when you're working in one directory will not work at all once you move into a different working directory.

As an example of a relative pathname, say you're working in the directory `RCourseFall2015` within the file structure shown in Figure 2.1, and you want to read in the `heat_mort.csv` file in the `CourseText` directory. To get from `RCourseFall2015` to that file, you'd need to look in the subdirectory `CourseText`, where you could find `heat_mort.csv`. Therefore, the relative pathname from your working directory would be:

```
"CourseText/heat_mort.csv"
```

You can use this relative pathname to tell R where to find and read in the file:

```
heat_mort <- read_csv("CourseText/heat_mort.csv")
```

While this pathname is much shorter than the absolute pathname, it is important to remember that if you are working in a different working directory, this relative pathname would no longer work.

There are a few abbreviations that can be really useful for pathnames:

Shorthand	Meaning
‘~’	Home directory
‘.’	Current working directory
‘..’	One directory up from current working directory
‘./..’	Two directories up from current working directory

These can help you keep pathnames shorter and also help you move “up-and-over” to get to a file or directory that’s not on the direct path below your current working directory.

For example, my home directory is `/Users/brookeanderson`. You can use the `list.files()` function to list all the files in a directory. If I wanted to list all the files in my `Downloads` directory, which is a direct sub-directory of my home directory, I could use:

```
list.files("~/Downloads")
```

As a second example, say I was working in the working directory `CourseText`, but I wanted to read in the `heat_mort.csv` file that’s in the `example_data` directory, rather than the one in the `CourseText` directory. I can use the `..` abbreviation to tell R to look up one directory from the current working directory, and then down within a subdirectory of that. The relative pathname in this case is:

```
"../Week2_Aug31/example_data/heat_mort.csv"
```

This tells R to look one directory up from the working directory `(..)` (this is also known as the **parent directory** of the current directory), which in this case is to `RCourseFall2015`, and then down within that directory to `Week2_Aug31`, then to `example_data`, and then to look there for the file `heat_mort.csv`.

The relative pathname to read this file while R is working in the `CourseText` directory would be:

```
heat_mort <- read_csv("../Week2_Aug31/example_data/heat_mort.csv")
```

Relative pathnames “break” as soon as you tried them from a different working directory—this fact might make it seem like you would never want to use relative pathnames, and would always want to use absolute ones instead, even if they’re longer. If that were the only consideration (length of the pathname), then perhaps that would be true. However, as you do more and more in R, there will likely be many occasions when you want to use relative pathnames instead. They are particularly useful if you ever want to share a whole directory, with all subdirectories, with a collaborator. In that case, if you’ve used relative pathnames, all the code should work fine for the person you share with, even though they’re running it on their own computer. Conversely, if you’d used absolute pathnames, none of them would work on another computer, because the “top” of the directory structure (i.e., for me, `/Users/brookeanderson/Desktop`) will almost definitely be different for your collaborator’s computer than it is for yours.

If you’re getting errors reading in files, and you think it’s related to the relative pathname you’re using, it’s often helpful to use `list.files()` to make sure the file you’re trying to load is in the directory that the relative pathname you’re using is directing R to.

2.4.4 Diversion: `paste`

This is a good opportunity to explain how to use some functions that can be very helpful when you’re using relative or absolute pathnames: `paste()` and `paste0()`.

As a bit of important background information, it’s important that you understand that you can save a pathname (absolute or relative) as an R object, and then use that R object in calls to later functions like `list.files()` and `read_csv()`. For example, to use the absolute pathname to read the `heat_mort.csv` file in the `CourseText` directory, you could run:

```
my_file <- "/Users/brookeanderson/Desktop/RCourseFall2015/CourseText/heat_mort.csv"
heat_mort <- read_csv(file = my_file)
```

You’ll notice from this code that the pathname to get to a directory or file can sometimes become ungainly and long. To keep your code cleaner, you can address this by using the `paste` or `paste0` functions. These functions come in handy in a lot of other applications, too, but this is a good place to introduce them.

The `paste()` function is very straightforward. It takes, as inputs, a series of different character strings you want to join together, and it pastes them together in a single character string. (As a note, this means that your result vector will only be one element long, for basic uses of `paste()`, while the inputs will be

several different character strings.) You separate all the different things you want to paste together using commas in the function call. For example:

```
paste("Sunday", "Monday", "Tuesday")
```

```
## [1] "Sunday Monday Tuesday"
```

```
length(x = c("Sunday", "Monday", "Tuesday"))
```

```
## [1] 3
```

```
length(x = paste("Sunday", "Monday", "Tuesday"))
```

```
## [1] 1
```

The `paste()` function has an option called `sep =`. This tells R what you want to use to separate the values you're pasting together in the output. The default is for R to use a space, as shown in the example above. To change the separator, you can change this option, and you can put in just about anything you want. For example, if you wanted to paste all the values together without spaces, you could use `sep = ""`:

```
paste("Sunday", "Monday", "Tuesday", sep = "")
```

```
## [1] "SundayMondayTuesday"
```

As a shortcut, instead of using the `sep = ""` option, you could achieve the same thing using the `paste0` function. This function is almost exactly like `paste`, but it defaults to `" "` (i.e., no space) as the separator between values by default:

```
paste0("Sunday", "Monday", "Tuesday")
```

```
## [1] "SundayMondayTuesday"
```

With pathnames, you will usually not want spaces. Therefore, you could think about using `paste0()` to write an object with the pathname you want to ultimately use in commands like `list.files()` and `setwd()`. This will allow you to keep your code cleaner, since you can now divide long pathnames over multiple lines:

```
my_file <- paste0("/Users/brookeanderson/Desktop/",
                   "RCourseFall2015/CourseText/heat_mort.csv")
heat_mort <- read_csv(file = my_file)
```

You will end up using `paste()` and `paste0()` for many other applications, but this is a good example of how you can start using these functions to start to get a feel for them.

2.4.5 Reading online flat files

So far, I've only shown you how to read in data from files that are saved to your computer. R can also read in data directly from the web. If a flat file is posted online, you can read it into R in almost exactly the same way that you would read in a local file. The only difference is that you will use the file's url instead of a local file path for the `file` argument.

With the `read_*` family of functions, you can do this both for flat files from a non-secure webpage (i.e., one that starts with `http`) and for files from a secure webpage (i.e., one that starts with `https`), including GitHub and Dropbox.

For example, to read in data from this GitHub repository of Ebola data, you can run:

```
library("dplyr")
url <- paste0("https://raw.githubusercontent.com/cmrvrivers/",
              "ebola/master/country_timeseries.csv")
ebola <- read_csv(file = url)
slice(.data = (select(.data = ebola, 1:3)), 1:3)
```

```
## # A tibble: 3 x 3
##   Date      Day Cases_Guinea
##   <chr>    <dbl>     <dbl>
## 1 1/5/2015  289      2776
## 2 1/4/2015  288      2775
## 3 1/3/2015  287      2769
```

2.5 Data cleaning

Download a pdf of the lecture slides for this video.

Once you have loaded data into R, you'll likely need to clean it up a little before you're ready to analyze it. Here, I'll go over the first steps of how to do that with functions from `dplyr`, another package in the tidyverse. Here are some of the most common data-cleaning tasks, along with the corresponding `dplyr` function for each:

Task	'dplyr' function
Renaming columns	<code>'rename'</code>
Filtering to certain rows	<code>'filter'</code>
Selecting certain columns	<code>'select'</code>
Adding or changing columns	<code>'mutate'</code>

In this section, I'll describe how to do each of these tasks; in later sections of the course, we'll go much deeper into how to clean messier data.

For the examples in this section, I'll use example data listing guests to the Daily Show. To follow along with these examples, you'll want to load that data, as well as load the `dplyr` package (install it using `install.packages` if you have not already):

```
library("dplyr")
daily_show <- read_csv(file = "data/daily_show_guests.csv", skip = 4)
```

I've used this data in previous examples, but as a reminder, here's what it looks like:

```
head(x = daily_show)
```

```
## # A tibble: 6 x 5
##   YEAR GoogleKnowlege_Occupation Show     Group  Raw_Guest_List
##   <dbl> <chr>                  <chr>    <chr> <chr>
## 1 1999 actor                 1/11/99  Acting Michael J. Fox
## 2 1999 Comedian              1/12/99  Comedy Sandra Bernhard
## 3 1999 television actress  1/13/99  Acting Tracey Ullman
## 4 1999 film actress          1/14/99  Acting Gillian Anderson
## 5 1999 actor                 1/18/99  Acting David Alan Grier
## 6 1999 actor                 1/19/99  Acting William Baldwin
```

2.5.1 Renaming columns

A first step is often re-naming the columns of the dataframe. It can be hard to work with a column name that:

- is long
- includes spaces or other special characters
- includes upper case

You can check out the column names for a dataframe using the `colnames` function, with the dataframe object as the argument. Several of the column names in `daily_show` have some of these issues:

```
colnames(x = daily_show)
```

```
## [1] "YEAR"                      "GoogleKnowlege_Occupation"
## [3] "Show"                        "Group"
## [5] "Raw_Guest_List"
```

To rename these columns, use `rename`. The basic syntax is:

```
## Generic code
rename(.data = dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
```

The first argument is the dataframe for which you'd like to rename columns. Then you list each pair of new versus old column names (in that order) for each of the columns you want to rename. To rename columns in the `daily_show` data using `rename`, for example, you would run:

```
daily_show <- rename(.data = daily_show,
                     year = YEAR,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
head(x = daily_show, 3)
```

```
## # A tibble: 3 x 5
##   year job           date category guest_name
##   <dbl> <chr>        <chr>   <chr>    <chr>
## 1 1999 actor        1/11/99 Acting   Michael J. Fox
## 2 1999 Comedian     1/12/99 Comedy   Sandra Bernhard
## 3 1999 television actress 1/13/99 Acting   Tracey Ullman
```



Many of the functions in tidyverse packages, including those in `dplyr`, provide exceptions to the general rule about when to use quotation marks versus when to leave them off. Unfortunately, this may make it a bit hard to learn when to use quotation marks versus when not to. One way to think about this, which is a bit of an oversimplification but can help as you're learning, is to assume that anytime you're using a `dplyr` function, every column in the dataframe you're working with has been loaded to your R session as its own object.

2.5.2 Selecting columns

Download a pdf of the lecture slides for this video.

Next, you may want to select only some columns of the dataframe. You can use the `select` function from `dplyr` to subset the dataframe to certain columns. The basic structure of this command is:

```
## Generic code
select(.data = dataframe, column_name_1, column_name_2, ...)
```

In this call, you first specify the dataframe to use and then list all of the column names to include in the output dataframe, with commas between each column name. For example, to select all columns in `daily_show` except `year` (since that information is already included in `date`), run:

```
select(.data = daily_show, job, date, category, guest_name)
```

```
## # A tibble: 2,693 x 4
##   job           date category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 actor        1/11/99 Acting   Michael J. Fox
## 2 Comedian     1/12/99 Comedy   Sandra Bernhard
```

```
## 3 television actress 1/13/99 Acting Tracey Ullman
## 4 film actress      1/14/99 Acting Gillian Anderson
## 5 actor              1/18/99 Acting David Alan Grier
## 6 actor              1/19/99 Acting William Baldwin
## 7 Singer-lyricist    1/20/99 Musician Michael Stipe
## 8 model              1/21/99 Media Carmen Electra
## 9 actor              1/25/99 Acting Matthew Lillard
## 10 stand-up comedian 1/26/99 Comedy David Cross
## # i 2,683 more rows
```



Don't forget that, if you want to change column names in the saved object, you must reassign the object to be the output of `rename`. If you run one of these cleaning functions without reassigning the object, R will print out the result, but the object itself won't change. You can take advantage of this, as I've done in this example, to look at the result of applying a function to a dataframe without changing the original dataframe. This can be helpful as you're figuring out how to write your code.

The `select` function also provides some time-saving tools. For example, in the last example, we wanted all the columns except one. Instead of writing out all the columns we want, we can use `-` with the columns we don't want to save time:

```
daily_show <- select(.data = daily_show, -year)
head(x = daily_show, n = 3)
```

```
## # A tibble: 3 x 4
##   job           date   category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 actor         1/11/99 Acting   Michael J. Fox
## 2 Comedian      1/12/99 Comedy   Sandra Bernhard
## 3 television actress 1/13/99 Acting Tracey Ullman
```

2.5.3 Extracting and arranging rows

Download a pdf of the lecture slides for this video.

There are a number of different actions you can take to extract or rearrange rows from a dataset to clean it up for your current analysis, including:

- `slice`

- `sample_n`
- `arrange`
- `filter`

We'll go through what each of these does and how to use them.

2.5.4 Slicing and sampling

The `slice` function from the `dplyr` package can extract certain rows based on their position in the dataframe.

We already looked at this a bit in Chapter 1. In the last chapter, you learned how to use the `slice` function to limit a dataframe to certain rows by row position.

For example, to print the first three rows of the `daily_show` data, you can run:

```
library("dplyr")
slice(.data = daily_show, 1:3)

## # A tibble: 3 x 4
##   job           date   category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 actor         1/11/99 Acting   Michael J. Fox
## 2 Comedian      1/12/99 Comedy   Sandra Bernhard
## 3 television actress 1/13/99 Acting   Tracey Ullman
```

There are some other functions you can use to extract rows from a tibble dataframe, all from the “`dplyr`” package.

For example, if you'd like to extract a random subset of n rows, you can use the `sample_n` function, with the `size` argument set to n .

To extract two random rows from the `daily_show` dataframe, run:

```
sample_n(tbl = daily_show, size = 2)

## # A tibble: 2 x 4
##   job           date   category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 <NA>          3/15/12 <NA>     None
## 2 university professor 8/5/10 Academic Akbar Ahmed
```

2.5.5 Arranging rows

There is also a function, `arrange`, you can use to re-order the rows in a dataframe based on the values in one of its columns. The syntax for this function is:

```
# Generic code
arrange(.data = dataframe, column_to_order_by)
```

If you run this function to use a character vector to order, it will order the rows alphabetically by the values in that column. If you specify a numeric vector, it will order the rows by the numeric value.

For example, we could reorder the `daily_show` data alphabetically by the values in the `category` column with the following call:

```
daily_show <- arrange(.data = daily_show, category)
slice(.data = daily_show, 1:3)
```

```
## # A tibble: 3 x 4
##   job      date    category guest_name
##   <chr>     <chr>    <chr>    <chr>
## 1 professor 10/3/01 Academic Stephen S. Morse
## 2 Professor 12/3/01 Academic Nadine Strossen
## 3 Historian 11/4/03 Academic Michael Beschloss
```

If you want the ordering to be reversed (e.g., from “z” to “a” for character vectors, from higher to lower for numeric, latest to earliest for a Date), you can include the `desc` function.

For example, to reorder the `daily_show` data by job category in descending alphabetical order, you can run:

```
daily_show <- arrange(.data = daily_show,
                      desc(x = category))
slice(.data = daily_show, 1:2)
```

```
## # A tibble: 2 x 4
##   job      date    category guest_name
##   <chr>     <chr>    <chr>    <chr>
## 1 news anchor 3/21/01 media     Jeff Varner
## 2 news anchor 10/15/02 media     Judy Woodruff
```

2.5.6 Filtering to certain rows

Next, you might want to filter the dataset down so that it only includes certain rows. For example, you might want to get a dataset with only the guests from 2015, or only guests who are scientists.

You can use the `filter` function from `dplyr` to filter a dataframe down to a subset of rows. The syntax is:

```
## Generic code
filter(.data = dataframe, logical expression)
```

The `logical expression` in this call gives the condition that a row must meet to be included in the output data frame. For example, if you want to create a data frame that only includes guests who were scientists, you can run:

```
scientists <- filter(.data = daily_show,
                      category == "Science")
head(x = scientists)

## # A tibble: 6 x 4
##   job           date   category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 neurosurgeon 4/28/03 Science Dr Sanjay Gupta
## 2 scientist     1/13/04 Science Catherine Weitz
## 3 physician      6/15/04 Science Hassan Ibrahim
## 4 doctor         9/6/05  Science Dr. Marc Siegel
## 5 astronaut      2/13/06 Science Astronaut Mike Mullane
## 6 Astrophysicist 1/30/07 Science Neil deGrasse Tyson
```

To build a logical expression to use in `filter`, you'll need to know some of R's logical operators. Some of the most commonly used ones are:

Operator	Meaning	Example
<code>==</code>	equals	<code>category == "Acting"</code>
<code>!=</code>	does not equal	<code>category != "Comedy"</code>
<code>%in%</code>	is in	<code>category %in% c("Academic", "Science")</code>
<code>is.na()</code>	is missing	<code>is.na(job)</code>
<code>!is.na()</code>	is not missing	<code>!is.na(job)</code>
<code>&</code>	and	<code>year == 2015 & category == "Academic"</code>
<code> </code>	or	<code>year == 2015 category == "Academic"</code>

We'll use these logical operators and expressions a lot more as the course continues, so they're worth learning by heart.



Two common errors with logical operators are: (1) Using `=` instead of `==` to check if two values are equal; and (2) Using `== NA` instead of `is.na` to check for missing observations.

2.5.7 Add or change columns

Download a pdf of the lecture slides for this video.

You can change a column or add a new column using the `mutate` function from the `dplyr` package. That function has the syntax:

```
# Generic code
mutate(.data = datafram,
       changed_column = function(changed_column),
       new_column = function(other arguments))
```

For example, the `job` column in `daily_show` sometimes uses upper case and sometimes does not (this call uses the `unique` function to list only unique values in this column):

```
head(x = unique(x = daily_show$job), n = 10)
```

```
## [1] "news anchor"      "neurosurgeon"     "scientist"        "physician"
## [5] "doctor"           "astronaut"        "Astrophysicist"   "Surgeon"
## [9] "Neuroscientist"   "primatologist"
```

To make all the observations in the `job` column lowercase, use the `str_to_lower` function from the `stringr` package within a `mutate` function:

```
library(package = "stringr")
mutate(.data = daily_show,
       job = str_to_lower(string = job))
```

```
## # A tibble: 2,693 x 4
##   job          date    category guest_name
```

```

##   <chr>      <chr>      <chr>      <chr>
## 1 news anchor 3/21/01 media    Jeff Varner
## 2 news anchor 10/15/02 media    Judy Woodruff
## 3 news anchor 11/6/02 media    Candy Crowley
## 4 news anchor 4/9/02  media   Judy Woodruff
## 5 news anchor 3/24/11 media    Bret Baier
## 6 neurosurgeon 4/28/03 Science Dr Sanjay Gupta
## 7 scientist    1/13/04 Science Catherine Weitz
## 8 physician     6/15/04 Science Hassan Ibrahim
## 9 doctor        9/6/05  Science Dr. Marc Siegel
## 10 astronaut    2/13/06 Science Astronaut Mike Mullane
## # i 2,683 more rows

```

2.6 Piping

Download a pdf of the lecture slides for this video.

So far, I've shown how to use these `dplyr` functions one at a time to clean up the data, reassigning the dataframe object at each step. However, there's a trick called "piping" that will let you clean up your code a bit when you're writing a script to clean data.

If you look at the format of these `dplyr` functions, you'll notice that they all take a dataframe as their first argument:

```

# Generic code
rename(.data = dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
select(.data = dataframe,
       column_name_1, column_name_2)
filter(.data = dataframe,
       logical expression)
mutate(.data = dataframe,
       changed_column = function(changed_column),
       new_column = function(other arguments))

```

Without piping, you have to reassign the dataframe object at each step of this cleaning if you want the changes saved in the object:

```

daily_show <-read_csv(file = "data/daily_show_guests.csv",
                      skip = 4)

```

```

daily_show <- rename(.data = daily_show,
                      job = GoogleKnowlege_Occupation,
                      date = Show,
                      category = Group,
                      guest_name = Raw_Guest_List)
daily_show <- select(.data = daily_show,
                      -YEAR)
daily_show <- mutate(.data = daily_show,
                      job = str_to_lower(job))
daily_show <- filter(.data = daily_show,
                      category == "Science")

```

Piping lets you clean this code up a bit. It can be used with any function that inputs a dataframe as its first argument. It *pipes* the dataframe created right before the pipe (%>%) into the function right after the pipe. With piping, therefore, the same data cleaning looks like:

```

daily_show <-read_csv(file = "data/daily_show_guests.csv",
                      skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
        date = Show,
        category = Group,
        guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")

```

Notice that, when piping, the first argument (the name of the dataframe) is excluded from all function calls that follow a pipe. This is because piping sends the dataframe from the last step into each of these functions as the dataframe argument.

Download a pdf of the lecture slides for this video.

2.6.1 Base R equivalents to dplyr functions

Just so you know, all of these `dplyr` functions have alternatives, either functions or processes, in base R:

‘dplyr’	Base R equivalent
‘rename’	Reassign ‘colnames’
‘select’	Square bracket indexing
‘filter’	‘subset’
‘mutate’	Use ‘\$’ to change / create columns

You will see these alternatives used in older code examples.

2.7 In-course Exercise Chapter 2

Use the `sample` function to determine the order for your group to rotate in sharing screens.

2.7.1 Downloading and checking out the example data

First, you'll download a directory from GitHub that has the data for this week's exercise and explore the files in that directory.

Have the first person in your group share their screen. Then take the following steps (everyone in the group should do this, but have one person share their screen and talk through what they're doing as you do this):

- Create an R Project directory to use as you practice coding for this class. You may have already done this as you watched this week's slides. If not, take these steps: (1) Open RStudio; (2) In RStudio, go to “File” -> “New Project” -> “New Directory” -> “New Project”. This will prompt you to pick a name for your R Project directory and also select where to save it. You can use a name like “learn_r” for your Project and save it somewhere easy to find, like you Desktop or Documents folder.
- Download the whole directory for this week from Github (https://github.com/geanders/week_2_data). To do that, go the the GitHub page with data for this week's exercise and, in the top right, click on the green “Code” button near the top right. When you click on this green button, one of the choices that you see should be “Download ZIP”. Click on this. This will download a compressed file with the full directory of data, probably to your computer's “Downloads” folder.
- Next, “unzip” the zipped downloaded file from GitHub. To do this, try double-clicking the file, or right click on the file and see if there's a “decompress” or “unzip” option. If you still don't have much luck, try googling how to unzip a zipped file on your computer's operating system. You should end up with six files and one directory (“measles_data”) which itself has sixteen files inside).
- Next move all these files/directories **except “week_2_data.RProj”** into the R Project directory you created in the first step. Don't do this in R, just use whatever technique you usually use on your computer to move files between directories (clicking and dragging the files from one folder to another, for example).
- Open RStudio to the Project you created in the first step, if you haven't already.

- Look through the structure of these files. What files are included? Which files are **flat files**? For any flat files, look inside the files—you can open them with RStudio or another text editor. If you are in your Project in RStudio, look in the “Files” pane, click on the flat file’s name, and select “View File” to get R to open the file with a text editor so you can look at the structure of the file. Which flat files are **delimited** (one category of flat files), and what are their delimiters?

2.7.2 Reading in different types of files

Now you’ll try reading in data from a variety of types of file formats. You will be working with the files you downloaded in the last part of the exercise.

Switch the person in your group who is sharing their screen. Then try the following tasks (there is example R code below if you get stuck):

- Create a new R script where you can put all the code you write for this exercise, as you develop that code over the next few parts of the exercise. Create a subdirectory in your course directory called “R” and save this script there using a .R extension (e.g., “week_2.R”).
- What type of flat file do you think the “ld_genetics.txt” file is? See if you can read it in using a `readr` function (e.g., `read_delim`, `read_fwf`, `read_tsv`, `read_csv`—pick depending on the type of file you think this is) and save it as the R object `ld_genetics`. Use the `summary` function to check out basic statistics on the data.
- Check out the file “measles_data/02-09-2015.txt”. What type of flat file do you think it is? Since it’s in a subdirectory, to read it into R, you’ll need to tell R how to get to it from the project directory, using something called a **relative pathname**. Read this file into R as an object named `ca_measles`, using the relative pathname (“measles_data/02-09-2015.txt”) in place of the file name in the `read_tsv` function call. Use the `col_names` option to name the columns “city” and “count”. What would the default column names be if you didn’t use this option (try this out by running `read_csv` without the `col_names` option)?
- Read in the Excel file “icd-10.xls” and assign it to the object name `idc10`. Use the `readxl` package to do that (examples are at the bottom of the linked page).
- Read in the SAS file `icu.sas7bdat`. To do this, use the `haven` package. Read the file into the R object `icu`.

Example R code:

```
# Load the `readr` package
library(package = "readr")
```

```
# Use `read_tsv` to read this file.  
ld_genetics <- read_tsv(file = "ld_genetics.txt")
```

```
## Rows: 3503 Columns: 9
## -- Column specification -----
## Delimiter: "\t"
## dbl (9): pos, nA, nC, nG, nT, GCsk, TAsk, cGCsk, cTAsk
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
summary(object = ld_genetics)
```

```

##          pos           nA           nC           nG
##  Min.   : 500   Min.   :185   Min.   :120.0   Min.   : 85.0
##  1st Qu.:876000 1st Qu.:288   1st Qu.:173.0   1st Qu.:172.0
##  Median :1751500 Median :308   Median :190.0   Median :189.0
##  Mean    :1751500 Mean   :309   Mean   :191.9   Mean   :191.8
##  3rd Qu.:2627000 3rd Qu.:329   3rd Qu.:209.0   3rd Qu.:208.0
##  Max.   :3502500 Max.   :463   Max.   :321.0   Max.   :326.0
##          nT           GCsk          TAsk          cGCsk
##  Min.   :188.0   Min.   :-189.0000   Min.   :-254.000   Min.   : -453
##  1st Qu.:286.0   1st Qu.: -30.0000   1st Qu.: -36.000   1st Qu.:10796
##  Median :306.0   Median :  0.0000   Median : -2.000   Median :23543
##  Mean    :307.2   Mean   : -0.1293   Mean   : -1.736   Mean   :22889
##  3rd Qu.:328.0   3rd Qu.:  29.0000   3rd Qu.:  32.500   3rd Qu.:34940
##  Max.   :444.0   Max.   : 134.0000   Max.   : 205.000   Max.   :46085
##          cTAsk
##  Min.   : -6247
##  1st Qu.: 1817
##  Median : 7656
##  Mean   : 7855
##  3rd Qu.:15036
##  Max.   :19049

```

```
## Rows: 13 Columns: 2
## -- Column specification -----
## Delimiter: "\t"
## chr (1): city
## dbl (1): count
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
head(x = ca_measles)
```

```
## # A tibble: 6 x 2
##   city           count
##   <chr>          <dbl>
## 1 ALAMEDA         6
## 2 LOS ANGELES     20
## 3 City of Long Beach  2
## 4 City of Pasadena  4
## 5 MARIN           2
## 6 ORANGE          34
```

You'll need the `readxl` package to read in the Excel file. Load that.

```
library(package = "readxl")
```

Use the `read_excel` function to read in the file.

```
icd10 <- read_excel(path = "icd-10.xls")
```

```
head(x = icd10)
```

```
## # A tibble: 6 x 2
##   Code    `ICD Title`
##   <chr>   <chr>
## 1 A00-B99 I. Certain infectious and parasitic diseases
## 2 A00-A09 Intestinal infectious diseases
## 3 A00    Cholera
## 4 A00.0  Cholera due to Vibrio cholerae O1, biovar cholerae
## 5 A00.1  Cholera due to Vibrio cholerae O1, biovar eltor
## 6 A00.9  Cholera, unspecified
```

```
# You'll need the `haven` function to read in the SAS file. Load that.
library(package = "haven")
```

```
# Use the `read_sas` function to read in this file.
icu <- read_sas(data_file = "icu.sas7bdat")
```

```
library(package = "dplyr")
slice(.data = select(.data = icu, 1:5), 1:5)
```

```
## # A tibble: 5 x 5
##       ID    STA   AGE GENDER RACE
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     4     1    87     1     1
## 2     8     0    27     1     1
## 3    12     0    59     0     1
## 4    14     0    77     0     1
## 5    27     1    76     1     1
```

2.7.3 Directory structure

Next, you’ll explore your current working directory and the files in that working directory.

Switch the person in your group who is sharing their screen. Then try the following tasks (there is example R code below if you get stuck):

- Make sure you have opened R and have moved into the R Project you created at the beginning of this exercise.
- In this Project directory, create a new subdirectory called “data” (if you don’t already have that subdirectory—you may have created it if you followed along with the code in the video lectures).
- In the last two steps, you downloaded and explored some data files. These should currently be in the main level of your R Project directory. Move these into the “data” subdirectory you just created. For this, use whatever tools you would normally use on your computer to move files from one directory to another—you don’t have to do this part in R. Keep the “measles” data in its own subdirectory (so, the “data” subdirectory of your project will have its own “measles” subdirectory, which will have those files).

- Check that you are, in fact, in the working directory you think you're in.
Run:

```
getwd()
```

Does it agree with the R project name in the top right hand corner of your R Studio window?

- Now, use the `list.files` function to print out which files or subdirectories you have in your current working directory (the R Project directory):

```
list.files()
```

What results do you get when you run this call? Is this what you were expecting?

- While staying in the same working directory, use `list.files()` to print the names of the available files in the “data” subdirectory using the `path` argument.
- Now see if you can use `list.files()` to list all the files in the “measles_data” subdirectory.
- Read into R the ebola data in `country_timeseries.csv` using the appropriate `readr` function. This will require you to use a relative filename to direct R to how to find that file in the “data” subdirectory of your current working directory. Assign the data you read as an R object named `ebola`.
- How many rows and columns does the `ebola` dataframe you just created have? What are the names of the columns?

Example R code:

Start by running `getwd()`. What you get will depend on your computer set-up. Make sure that you understand the output you get from this call on your computer.

Next, while staying in the same working directory, use `list.files()` to print the names of the available files in the “data” subdirectory by using the `path` argument. How about in the “R” subdirectory (if you created one in your project)?

```
list.files(path = "data")
list.files(path = "R")
```

Note that you can use `list.files` to list the files in your “data” subdirectory using either:

- + A relative pathname
- + An absolute pathname

```
list.files(path = "data") # This is using a relative pathname
list.files(path = "/Users/brookeanderson/Documents/r_course_2018/data") # Absolute path
# (Yours will be different and will depend on how your computer file
# structure is set up.)
```

Now use a relative pathname along with `list.files()` to list all the files in the “measles_data” subdirectory.

```
list.files(path = "data/measles_data")
```

Read in the Ebola data in `country_timeseries.csv` from your current working directory using the appropriate `readr` function and a relative pathname. This will require you to use a relative filename to direct R to how to find that file in the “data” subdirectory of your current working directory.

```
library("readr")
ebola <- read_csv(file = "data/country_timeseries.csv")
```

How many rows and columns does it have? What are the names of the columns?

```
dim(x = ebola)    # Get the dimensions of the data (`nrow` and `ncol` would also work)
colnames(x = ebola) # Get the column names (you can also just print the object: `ebola`)
```

If you have extra time:

- Find out some more about this Ebola dataset by checking out Caitlin Rivers’ Ebola data GitHub repository. Who is Caitlin Rivers? How did she put this dataset together?
- Search for R code related to Ebola research on GitHub. Go to the GitHub home page and use the search bar to search for “ebola”. On the results page, scroll down and use the “Language” sidebar on the left to choose repositories with R code. Did you find any interesting projects?

- When you `list.files()` for the “data” subdirectory, almost everything listed has a file extension, like `.csv`, `.xls`, `.sas7bdat`. One thing does not. Which one? Why does this listing not have a file extension?

2.7.4 Cleaning up data #1

Switch the person in your group who is sharing their screen. Then try out the following tasks:

- Copy the following code into an R script. Figure out what each line does, and add comments to each line of code describing what the code is doing. Use the helpfiles for functions as needed to figure out functions we haven’t covered yet.

```
# Copy this code to an R script and add comments describing what each line is doing
# Install any packages that the code loads but that you don't have.
library(package = "haven")
library(package = "forcats")
library(package = "stringr")

icu <- read_sas(data_file = "data/icu.sas7bdat")

icu <- select(.data = icu, ID, AGE, GENDER)

icu <- rename(.data = icu,
              id = ID,
              age = AGE,
              gender = GENDER)

icu <- mutate(.data = icu,
              gender = as_factor(x = gender),
              gender = fct_recode(.f = gender,
                                  Male = "0",
                                  Female = "1"),
              id = str_c(id))

icu
```

- Following previous parts of the in-course exercise, you have an R object called `ebola` (if you need to, use some code from earlier in this in-course exercise to read in the data and create that object). Create an object called `ebola_liberia` that only has the columns with the date and the

number of cases and deaths in Liberia. How many columns does this new dataframe have? How many observations?

- Change the column names to `date`, `cases`, and `deaths`.
- Add a column called `ratio` that has the ratio of deaths to cases for each observation (i.e., death counts divided by case counts).

Example R code:

```
# Load the dplyr package
library(package = "dplyr")

## Create a subset with just the Liberia columns and Date
ebola_liberia <- select(.data = ebola,
                         Date, Cases_Liberia, Deaths_Liberia)
head(x = ebola_liberia)

## # A tibble: 6 x 3
##   Date      Cases_Liberia Deaths_Liberia
##   <chr>        <dbl>        <dbl>
## 1 1/5/2015      NA          NA
## 2 1/4/2015      NA          NA
## 3 1/3/2015     8166        3496
## 4 1/2/2015     8157        3496
## 5 12/31/2014    8115        3471
## 6 12/28/2014    8018        3423

## How many columns and rows does the whole dataset have (could also use `dim`)?
ncol(x = ebola_liberia)

## [1] 3

nrow(x = ebola_liberia)

## [1] 122
```

```
## Rename the columns
ebola_liberia <- rename(.data = ebola_liberia,
                        date = Date,
                        cases = Cases_Liberia,
                        deaths = Deaths_Liberia)

head(ebola_liberia)
```

```
## # A tibble: 6 x 3
##   date      cases deaths
##   <chr>     <dbl>  <dbl>
## 1 1/5/2015     NA     NA
## 2 1/4/2015     NA     NA
## 3 1/3/2015    8166   3496
## 4 1/2/2015    8157   3496
## 5 12/31/2014   8115   3471
## 6 12/28/2014   8018   3423
```

```
## Add a `ratio` column
ebola_liberia <- mutate(.data = ebola_liberia,
                        ratio = deaths / cases)

head(x = ebola_liberia)
```

```
## # A tibble: 6 x 4
##   date      cases deaths  ratio
##   <chr>     <dbl>  <dbl>  <dbl>
## 1 1/5/2015     NA     NA NA
## 2 1/4/2015     NA     NA NA
## 3 1/3/2015    8166   3496  0.428
## 4 1/2/2015    8157   3496  0.429
## 5 12/31/2014   8115   3471  0.428
## 6 12/28/2014   8018   3423  0.427
```

2.7.5 Cleaning up data #2

Switch the person in your group who is sharing their screen. Then try out the following tasks:

- Filter out all rows from the `ebola_liberia` dataframe that are missing death counts for Liberia. How many rows are in the dataframe now?

- Create a new object called `first_five` that has only the five observations with the highest death counts in Liberia. What date in this dataset had the most deaths?

Example R code:

```
## Filter out the rows that are missing death counts for Liberia
ebola_liberia <- filter(.data = ebola_liberia,
                         !is.na(deaths))
head(x = ebola_liberia)

## # A tibble: 6 x 4
##   date      cases deaths ratio
##   <chr>     <dbl>  <dbl> <dbl>
## 1 1/3/2015    8166   3496 0.428
## 2 1/2/2015    8157   3496 0.429
## 3 12/31/2014  8115   3471 0.428
## 4 12/28/2014  8018   3423 0.427
## 5 12/24/2014  7977   3413 0.428
## 6 12/20/2014  7862   3384 0.430

nrow(x = ebola_liberia)

## [1] 81

## Create an object with just the top five observations in terms of death counts
first_five <- arrange(.data = ebola_liberia,
                      desc(deaths)) # First, rearrange the rows by deaths
first_five <- slice(.data = first_five,
                     1:5) # Limit the dataframe to the first five rows
first_five # Two days tied for the highest deaths counts (Jan. 2 and 3, 2015).

## # A tibble: 5 x 4
##   date      cases deaths ratio
##   <chr>     <dbl>  <dbl> <dbl>
## 1 1/3/2015    8166   3496 0.428
## 2 1/2/2015    8157   3496 0.429
## 3 12/31/2014  8115   3471 0.428
## 4 12/28/2014  8018   3423 0.427
## 5 12/24/2014  7977   3413 0.428
```

2.7.6 Piping

Switch the person in your group who is sharing their screen. Then try out the following tasks:

- Copy the following “piped” code into an R script. Figure out what each line does, and add comments to each line of code describing what the code is doing.

```
# Copy this code to an R script and add comments describing what each line is doing
library(package = "haven")
icu <- read_sas(data_file = "data/icu.sas7bdat") %>%
  select(ID, AGE, GENDER) %>%
  rename(id = ID,
         age = AGE,
         gender = GENDER) %>%
  mutate(gender = as_factor(x = gender),
         gender = fct_recode(.f = gender,
                             Male = "0",
                             Female = "1"),
         id = str_c(id)) %>%
  arrange(age) %>%
  slice(1:10)

icu
```

- In previous sections of the in-course exercise, you have created code to read in and clean the Ebola dataset to create `ebola_liberia`. This included the following cleaning steps: (1) selecting certain columns, (2) renaming those columns, (3) adding a `ratio` column, and (4) removing observations for which the count of deaths in Liberia is missing. Re-write this code to create and clean `ebola_liberia` as “piped” code. Start from reading in the raw data.

Example R code:

```
ebola_liberia <- read_csv(file = "data/country_timeseries.csv") %>%
  select(Date, Cases_Liberia, Deaths_Liberia) %>%
  rename(date = Date,
         cases = Cases_Liberia,
         deaths = Deaths_Liberia) %>%
```

```
mutate(ratio = deaths / cases) %>%
filter(!is.na(x = cases))

head(x = ebola_liberia)
```

```
## # A tibble: 6 x 4
##   date      cases  deaths ratio
##   <chr>     <dbl>   <dbl> <dbl>
## 1 1/3/2015    8166   3496 0.428
## 2 1/2/2015    8157   3496 0.429
## 3 12/31/2014  8115   3471 0.428
## 4 12/28/2014  8018   3423 0.427
## 5 12/24/2014  7977   3413 0.428
## 6 12/20/2014  7862   3384 0.430
```

Chapter 3

Exploring data #1

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

3.1 Objectives

After this chapter, you should (know / understand / be able to):

- Be able to load and use datasets from R packages
- Be able to describe and use logical vectors
- Understand how logical vectors check logical statements against other R vector(s) and store TRUE / FALSE values as 0 / 1 at a deeper level
- Be able to use the `dplyr` function `mutate` to create a logical vector as a new column in a dataframe and the `dplyr` function `filter` with that new column to filter a dataframe to a subset of rows
- Be able to use the bang operator (!) to reverse a logical vector
- Know what the “tidyverse” is and name some of its packages
- Be able to use some simple statistical functions (e.g., `min`, `max`, `mean`, `median`, `cor`, `summary`), including how to handle missing values when using these
- Be able to use the `dplyr` function `summarize` to summarize data, with and without grouping using `group_by`, including with special functions `n`, `n_distinct`, `first`, and `last`
- Understand the three basic elements of `ggplot` plots: data, aesthetics, and geoms
- Be able to create a `ggplot` object, set its data using `data = ...` and its aesthetics using `mapping = aes(...)`, and add on layers (including `geoms`) with `+`

- Be able to create some basic plots (e.g., scatterplots, boxplots, histograms) using `ggplot2` functions
- Understand the difference between setting an aesthetic by mapping it to a column of the dataframe versus setting it to a constant value
- Understand the difference between “statistical” geoms (e.g., histograms, boxplots) and geoms that add one geom element per dataframe observation (row)

Download a pdf of the lecture slides for this video.

3.2 Simple statistics functions

3.2.1 Summary statistics

Download a pdf of the lecture slides for this video.

To explore your data, you’ll need to be able to calculate some simple statistics for vectors, including calculating the mean and range of continuous variables and counting the number of values in each category of a factor or logical vector.

Here are some simple statistics functions you will likely use often:

Function	Description
<code>range()</code>	Range (minimum and maximum) of vector
<code>min()</code> , <code>max()</code>	Minimum or maximum of vector
<code>mean()</code> , <code>median()</code>	Mean or median of vector
<code>sd()</code>	Standard deviation of vector
<code>table()</code>	Number of observations per level for a factor vector
<code>cor()</code>	Determine correlation(s) between two or more vectors
<code>summary()</code>	Summary statistics, depends on class

All of these functions take, as the main argument, the vector or vectors for which you want the statistic. If there are missing values in the vector, you’ll typically need to add an argument to say what to do with the missing values. The parameter name for this varies by function, but for many of these functions it’s `na.rm = TRUE` or `use="complete.obs"`.

```
## -- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
## vforcats    1.0.0     vpurrr      1.0.1
## vggplot2    3.4.2     vtidy       1.3.0
## vlubridate  1.9.2
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
```

```
## x dplyr::lag()    masks stats::lag()  
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
mean(nepali$wt, na.rm = TRUE)
```

```
## [1] 11.18848
```

```
range(nepali$ht, na.rm = TRUE)
```

```
## [1] 52.4 110.7
```

```
sd(nepali$ht, na.rm = TRUE)
```

```
## [1] 11.99779
```

```
table(nepali$sex)
```

```
##  
##   1   2  
## 535 465
```

Most of these functions take a single vector as the input. The `cor` function, however, calculates the correlation between vectors and so takes two or more vectors. If you give it multiple values, it will give the correlation matrix for all the vectors.

```
cor(nepali$wt, nepali$ht, use = "complete.obs")
```

```
## [1] 0.9549871
```

```
cor((nepali %>% select(wt, ht, age)), use = "complete.obs")
```

```
##           wt          ht         age
## wt  1.0000000 0.9549871 0.8775507
## ht  0.9549871 1.0000000 0.9195052
## age 0.8775507 0.9195052 1.0000000
```

R supports object-oriented programming. Your first taste of this shows up with the `summary` function. For the `summary` function, R does not run the same code every time. Instead, R first checks what type of object was input to `summary`, and then it runs a function (*method*) specific to that type of object. For example, if you input a continuous vector, like the `ht` column in `nepali`, to `summary`, the function will return the mean, median, range, and 25th and 75th percentile values:

```
summary(nepali$wt)
```

```
##   Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
##   3.80    9.00   11.10   11.19   13.20   19.20    123
```

However, if you submit a factor vector, like the `sex` column in `nepali`, the `summary` function will return a count of how many elements of the vector are in each factor level (as a note, you could do the same thing with the `table` function):

```
summary(nepali$sex)
```

```
##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
##   1.000 1.000 1.000   1.465 2.000 2.000
```

The `summary` function can also input other data structures, including dataframes, lists, and special object types, like regression model objects. In each case, it performs different actions specific to the object type. Later in this section, we'll cover regression models, and see what the `summary` function returns when it is used with regression model objects.

3.2.2 `summarize` function

You will often want to use these functions in conjunction with the `summarize` function in `dplyr`. For example, to create a new dataframe with the mean weight of children in the `nepali` dataset, you can use `mean` inside a `summarize` function:

```
library(dplyr)
nepali %>%
  summarize(mean_wt = mean(wt, na.rm = TRUE))

##   mean_wt
## 1 11.18848
```

There are also some special functions that are particularly useful with `summarize` and other `dplyr` functions. For example, the `n` function will calculate the number of observations and the `first` function will return the first value of a column:

```
nepali %>%
  summarize(n_children = n(),
            first_id = first(id))

##   n_children first_id
## 1          1000    120011
```

See the “summary function” section of the the RStudio Data Wrangling cheat-sheet for more examples of these special functions.

Often, you will be more interested in summaries within certain groupings of your data, rather than overall summaries. For example, you may be interested in mean height and weight by sex, rather than across all children, for the `nepali` data. It is very easy to calculate these grouped summaries using `dplyr`—you just need to group data using the `group_by` function (also a `dplyr` function) before you run the `summarize` function:

```
nepali %>%
  group_by(sex) %>%
  summarize(mean_wt = mean(wt, na.rm = TRUE),
            n_children = n(),
            first_id = first(id))
```

```
## # A tibble: 2 x 4
##   sex  mean_wt n_children first_id
##   <int>    <dbl>      <int>     <int>
## 1     1     11.4       535    120011
## 2     2     10.9       465    120012
```



Don't forget that you need to save the output to a new object if you want to use it later. The above code, which creates a dataframe with summaries for Nepali children by sex, will only be printed out to your console if run as-is. If you'd like to save this output as an object to use later (for example, for a plot or table), you need to assign it to an R object.

3.3 Factor vectors

Download a pdf of the lecture slides for this video.

3.4 Data from a package

Download a pdf of the lecture slides for this video.

So far we've covered two ways to get data into R:

1. From flat files (either on your computer or online)
2. From binary file formats like SAS and Excel.

Many R packages come with their own data, which is very easy to load and use. For example, the `faraway` package, which complements Julian Faraway's book *Linear Models with R*, has a dataset called `worldcup` that I'll use for some examples and that you'll use for part of this week's in-course exercise. To load this dataset, first load the package with the data (`faraway`) and then use the `data()` function with the dataset name ("worldcup") as the argument to the `data` function:

```
library(faraway)
data("worldcup")
```

Unlike most data objects you'll work with, datasets that are part of an R package will often have their own help files. You can access this help file for a dataset using the `?` operator with the dataset's name:

```
?worldcup
```

This helpful will usually include information about the size of the dataset, as well as definitions for each of the columns.

To get a list of all of the datasets that are available in the packages you currently have loaded, run `data()` without an option inside the parentheses:

```
data()
```



If you run the `library` function without any arguments—`library()`—it works in a similar way. R will open a list of all the R packages that you have installed on your computer and can open with a `library` call.

For this chapter, we'll be working with a modified version of the `nepali` dataset from the `faraway` package. This gives data from a study of the health of a group of Nepalese children. Each observation is a single measurement for a child; there can be multiple observations per child. We'll use a modified version of this dataframe that limits it to the columns with the child's id, sex, weight, height, and age, and limited to each child's first measurement. To create this modified dataset, run the following code:

```
library(dplyr)
library(faraway)
data(nepali)
nepali <- nepali %>%
  # Limit to certain columns
  select(id, sex, wt, ht, age) %>%
  # Convert id and sex to factors
  mutate(id = factor(id),
        sex = factor(sex, levels = c(1, 2),
                     labels = c("Male", "Female"))) %>%
  # Limit to first obs. per child
  distinct(id, .keep_all = TRUE)
```

The first few rows of the data should now look like:

```
nepali %>%
  slice(1:4)
```

```
##      id    sex   wt   ht age
## 1 120011  Male 12.8 91.2 41
## 6 120012 Female 14.9 103.9 57
## 11 120021 Female  7.7 70.1  8
## 16 120022 Female 12.1 86.4 35
```

3.5 Dates in R

Download a pdf of the lecture slides for this video.

As part of the data cleaning process, you may want to change the class of some of the columns in the dataframe. For example, you may want to change a column from a character to a date.

Here are some of the most common vector classes in R:

Class	Example
character	“Chemistry”, “Physics”, “Mathematics”
numeric	10, 20, 30, 40
factor	Male [underlying number: 1], Female [2]
Date	“2010-01-01” [underlying number: 14,610]
logical	TRUE, FALSE

To find out the class of a vector (including a column in a dataframe – remember each column can be thought of as a vector), you can use `class()`:

```
class(daily_show$date)
```

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

It is especially common to need to convert dates during the data cleaning process, since date columns will usually be read into R as characters or factors—you can do some interesting things with vectors that are in a Date class that you cannot do with a vector in a character class.

To convert a vector to the `Date` class, if you’d like to only use base R, you can use the `as.Date` function. I’ll walk through how to use `as.Date`, since it’s often

used in older R code. However, I recommend in your own code that you instead use the `lubridate` package, which I'll talk about later in this section.

To convert a vector to the `Date` class, you can use functions in the `lubridate` package. This package has a series of functions based on the order that date elements are given in the incoming character with date information. For example, in "12/31/99", the date elements are given in the order of month (`m`), day (`d`), year (`y`), so this character string could be converted to the date class with the function `mdy`. As another example, the `ymd` function from `lubridate` can be used to parse a column into a `Date` class, regardless of the original format of the date, as long as the date elements are in the order: year, month, day. For example:

```
library("lubridate")
ymd("2008-10-13")
```

```
## [1] "2008-10-13"
```

```
ymd("'08 Oct 13")
```

```
## [1] "2008-10-13"
```

```
ymd("'08 Oct 13")
```

```
## [1] "2008-10-13"
```

To convert the `date` column in the `daily_show` data into a `Date` class, then, you can run:

```
library(package = "lubridate")

class(x = daily_show$date) # Check the class of the 'date' column before mutating it

## [1] "character"
```

```
daily_show <- mutate(.data = daily_show,
                     date = mdy(date))
head(x = daily_show, n = 3)
```

```
## # A tibble: 3 x 4
##   job           date     category guest_name
##   <chr>        <date>    <chr>    <chr>
## 1 neurosurgeon 2003-04-28 Science  Dr Sanjay Gupta
## 2 scientist     2004-01-13 Science Catherine Weitz
## 3 physician     2004-06-15 Science Hassan Ibrahim
```

```
class(x = daily_show$date) # Check the class of the 'date' column after mutating it
```

```
## [1] "Date"
```

Once you have an object in the `Date` class, you can do things like plot by date, calculate the range of dates, and calculate the total number of days the dataset covers:

```
range(daily_show$date)
diff(x = range(daily_show$date))
```

We could have used these to transform the date in `daily_show`, using the following pipe chain:

```
daily_show <- read_csv(file = "data/daily_show_guests.csv",
                      skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(date = mdy(date)) %>%
  filter(category == "Science")
head(x = daily_show, n = 2)
```

```
## # A tibble: 2 x 4
##   job         date     category guest_name
##   <chr>       <date>    <chr>    <chr>
## 1 neurosurgeon 2003-04-28 Science  Dr Sanjay Gupta
## 2 scientist    2004-01-13 Science Catherine Weitz
```

The `lubridate` package also includes functions to pull out certain elements of a date, including:

- `wday`
- `mday`
- `yday`
- `month`
- `quarter`
- `year`

For example, we could use `wday` to create a new column with the weekday of each show:

```
mutate(.data = daily_show,
       show_day = wday(x = date, label = TRUE)) %>%
  select(date, show_day, guest_name) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 3
##   date     show_day guest_name
##   <date>    <ord>    <chr>
## 1 2003-04-28 Mon     Dr Sanjay Gupta
## 2 2004-01-13 Tue     Catherine Weitz
## 3 2004-06-15 Tue     Hassan Ibrahim
## 4 2005-09-06 Tue     Dr. Marc Siegel
## 5 2006-02-13 Mon     Astronaut Mike Mullane
```



R functions tend to use the timezone of **YOUR** computer's operating system by default, or UTC, or GMT. You need to be careful when working with dates and times to either specify the time zone or convince yourself the default behavior works for your application.

3.6 Logical vectors

Download a pdf of the lecture slides for this video.

Last week, you learned a lot about logical statements and how to use them with the `filter` function from the `dplyr` package. You can also use logical vectors, created with these logical statements, for a lot of other things. For example, you can use them directly in the square bracket indexing (`[..., ...]`) to pull out just the rows of a dataframe that meet a certain condition. For using logical statements in either context, it is helpful to understand a bit more about logical vectors.

When you run a logical statement on a vector, you create a logical vector the same length as the original vector:

```
length(nepali$sex)
```

```
## [1] 200
```

```
length(nepali$sex == "Male")
```

```
## [1] 200
```

The logical vector (`nepali$sex == "Male"` in this example) will have the value `TRUE` at any position where the original vector (`nepali$sex` in this example) met the logical condition you tested, and `FALSE` anywhere else:

```
head(nepali$sex)
```

```
## [1] Male   Female Female Female Male   Male
## Levels: Male Female
```

```
head(nepali$sex == "Male")
```

```
## [1] TRUE FALSE FALSE FALSE  TRUE  TRUE
```

You can “flip” this logical vector (i.e., change every TRUE to FALSE and vice-versa) using the *bang operator*, `!`:

```
is_male <- nepali$sex == "Male" # Save this logical vector as the object named `is_male`  
head(is_male)
```

```
## [1] TRUE FALSE FALSE FALSE TRUE TRUE
```

```
head(!is_male)
```

```
## [1] FALSE TRUE TRUE TRUE FALSE FALSE
```

The bang operator turns out to be very useful. You will often find cases where it’s difficult to write a logical vector to get what you want, but fairly easy to write the inverse (find everything you don’t want). One example is filtering down to non-missing values—the `is.na` function will return TRUE for any value that is NA, so you can use `!is.na()` to identify any non-missing values.

You can do a few cool things with a logical vector. For example, you can use it inside a `filter` function to pull out just the rows of a dataframe where `is_male` is TRUE:

```
nepali %>%  
  filter(is_male) %>%  
  head()
```

```
##      id sex wt ht age  
## 1 120011 Male 12.8 91.2 41  
## 21 120023 Male 14.2 99.4 49  
## 26 120031 Male 13.9 96.4 46  
## 31 120051 Male  8.3 69.5  8  
## 41 120053 Male 15.8 96.0 54  
## 51 120062 Male 12.1 89.9 57
```

Or, with `!`, just the rows where `is_male` is FALSE:

```
nepali %>%
  filter(!is_male) %>%
  head()
```

```
##      id    sex   wt   ht age
## 6 120012 Female 14.9 103.9 57
## 11 120021 Female  7.7  70.1  8
## 16 120022 Female 12.1  86.4 35
## 36 120052 Female 11.8  83.6 32
## 46 120061 Female  8.7  78.5 26
## 71 120082 Female 11.2  79.8 36
```

You can also use `sum()` and `table()` with a logical vector to find out how many of the values in the vector are TRUE AND FALSE. You can use `sum` because R saves logical vectors at a basic level as 0 for FALSE and 1 for TRUE. Therefore, if you add up all the values in a logical vector, you're adding up the number of observations with the value TRUE.

In the example, you can use these functions to find out how many males and females are in the dataset:

```
sum(is_male)
```

```
## [1] 107
```

```
sum(!is_male)
```

```
## [1] 93
```

```
table(is_male)
```

```
## is_male
## FALSE  TRUE
##     93    107
```

Note that you could also achieve the same thing with `dplyr` functions. For example, you could use `mutate` with a logical statement to create an `is_male` column in the `nepali` datafram, then group by the new `is_male` column and count the number of observations in each group using `count`:

```
library(dplyr)
nepali %>%
  mutate(is_male = sex == "Male") %>%
  group_by(is_male) %>%
  count()
```

```
## # A tibble: 2 x 2
## # Groups:   is_male [2]
##   is_male     n
##   <lgl>   <int>
## 1 FALSE      93
## 2 TRUE       107
```

We will cover using `summarize`, including with data that has been grouped with `group_by`, later in this chapter.

Download a pdf of the lecture slides for this video.

3.7 Plots to explore data

Download a pdf of the lecture slides for this video.

Exploratory data analysis is a key step in data analysis and plotting your data in different ways is an important part of this process. In this section, I will focus on the basics of `ggplot2` plotting, to get you started creating some plots to explore your data. This section will focus on making **useful**, rather than **attractive** graphs, since at this stage we are focusing on exploring data for yourself rather than presenting results to others. Next week, I will explain more about how you can customize `ggplot` objects, to help you make plots to communicate with others.

All of the plots we'll make today will use the `ggplot2` package (another member of the tidyverse!). If you don't already have that installed, you'll need to install it. You then need to load the package in your current session of R:

```
# install.packages("ggplot2") ## Uncomment and run if you don't have `ggplot2` installed
library(ggplot2)
```

The process of creating a plot using `ggplot2` follows conventions that are a bit different than most of the code you've seen so far in R (although it is somewhat similar to the idea of piping I introduced in the last chapter). The basic steps behind creating a plot with `ggplot2` are:

1. Create an object of the `ggplot` class, typically specifying the `data` and some or all of the `aesthetics`;
2. Add on `geoms` and other elements to create and customize the plot, using `+`.

You can add on one or many geoms and other elements to create plots that range from very simple to very customized. This week, we'll focus on simple geoms and added elements, and then explore more detailed customization next week.



If R gets to the end of a line and there is not some indication that the call is not over (e.g., `%>%` for piping or `+` for `ggplot2` plots), R interprets that as a message to run the call without reading in further code. A common error when writing `ggplot2` code is to put the `+` to add a geom or element at the beginning of a line rather than the end of a previous line— in this case, R will try to execute the call too soon. To avoid errors, be sure to end lines with `+`, don't start lines with it.

3.7.1 Initializing a ggplot object

The first step in creating a plot using `ggplot2` is to create a `ggplot` object. This object will not, by itself, create a plot with anything in it. Instead, it typically specifies the data frame you want to use and which aesthetics will be mapped to certain columns of that data frame (aesthetics are explained more in the next subsection).

Use the following conventions to initialize a `ggplot` object:

```
## Generic code
object <- ggplot(dataframe, aes(x = column_1, y = column_2))
```

The data frame is the first parameter in a `ggplot` call and, if you like, you can use the parameter definition with that call (e.g., `data = dataframe`). Aesthetics are defined within an `aes` function call that typically is used within the `ggplot` call.



While the `ggplot` call is the place where you will most often see an `aes` call, `aes` can also be used within the calls to add specific geoms. This can be particularly useful if you want to map aesthetics differently for different geoms in your plot. We'll see some examples of this use of `aes` more in later sections, when we talk about customizing plots. The `data` parameter can also be used in geom calls, to use a different data frame from the one defined when creating the original `ggplot` object, although this tends to be less common.

3.7.2 Plot aesthetics

Aesthetics are properties of the plot that can show certain elements of the data. For example, in Figure 3.1, color shows (is mapped to) gender, x-position shows height, and y-position shows weight in a sample data set of measurements of children in Nepal.

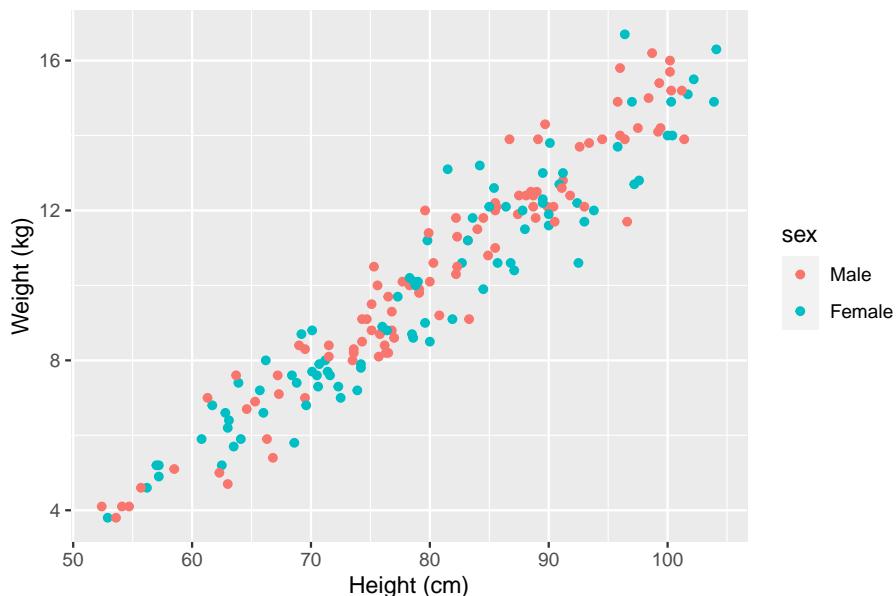


Figure 3.1: Example of how different properties of a plot can show different elements to the data. Here, color indicates gender, position along the x-axis shows height, and position along the y-axis shows weight. This example is a subset of data from the ‘nepali’ dataset in the ‘faraway’ package.



Any of these aesthetics could also be given a constant value, instead of being mapped to an element of the data. For example, all the points could be red, instead of showing gender.

Which aesthetics are required for a plot depend on which geoms (more on those in a second) you're adding to the plot. You can find out the aesthetics you can use for a geom in the “Aesthetics” section of the geom’s help file (e.g., `?geom_point`). Required aesthetics are in bold in this section of the help file and optional ones are not. Common plot aesthetics you might want to specify include:

Code	Description
<code>'x'</code>	Position on x-axis
<code>'y'</code>	Position on y-axis
<code>'shape'</code>	Shape
<code>'color'</code>	Color of border of elements
<code>'fill'</code>	Color of inside of elements
<code>'size'</code>	Size
<code>'alpha'</code>	Transparency (1: opaque; 0: transparent)
<code>'linetype'</code>	Type of line (e.g., solid, dashed)

3.7.3 Adding geoms

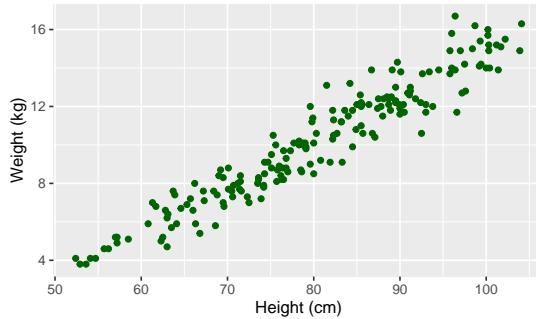
Next, you’ll want to add one or more `geoms` to create the plot. You can add these with `+` after the `ggplot` statement to initialize the `ggplot` object. Some of the most common geoms are:

Plot type	ggplot2 function
Histogram (1 numeric variable)	<code>'geom_histogram'</code>
Scatterplot (2 numeric variables)	<code>'geom_point'</code>
Boxplot (1 numeric variable, possibly 1 factor variable)	<code>'geom_boxplot'</code>
Line graph (2 numeric variables)	<code>'geom_line'</code>

3.7.4 Constant aesthetics

Download a pdf of the lecture slides for this video.

Instead of mapping an aesthetic to an element of your data, you can use a constant value for it. For example, you may want to make all the points green, rather than having color map to gender:



In this case, you'll define that aesthetic when you add the geom, outside of an `aes` statement. In R, you can specify the shape of points with a number. Figure 3.2 shows the shapes that correspond to the numbers 1 to 25 in the `shape` aesthetic. This figure also provides an example of the difference between color (black for all these example points) and fill (red for these examples). You can see that some point shapes include a fill (21 for example), while some are either empty (1) or solid (19).

```
## Warning: `data_frame()` was deprecated in tibble 1.1.0.
## i Please use `tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

If you want to set color to be a constant value, you can do that in R using character strings for different colors. Figure 3.3 gives an example of some of the different blues available in R. To find links to listings of different R colors, google “R colors” and search by “Images”.

3.7.5 Useful plot additions

There are also a number of elements that you can add onto a `ggplot` object using `+`. A few that are used very frequently are:

Element	Description
‘ <code>ggtitle</code> ’	Plot title
‘ <code>xlab</code> ’, ‘ <code>ylab</code> ’	x- and y-axis labels
‘ <code>xlim</code> ’, ‘ <code>ylim</code> ’	Limits of x- and y-axis

3.7.6 Example dataset

For the example plots, I'll use a dataset in the `faraway` package called `nepali`. This gives data from a study of the health of a group of Nepalese children.

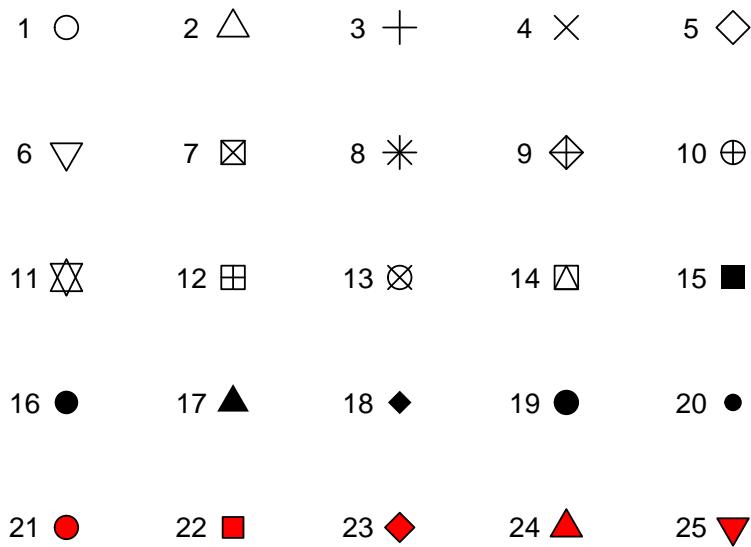


Figure 3.2: Examples of the shapes corresponding to different numeric choices for the ‘shape’ aesthetic. For all examples, ‘color’ is set to black and ‘fill’ to red.

- blue
- blue4
- darkorchid
- deepskyblue2
- steelblue1
- dodgerblue3

Figure 3.3: Example of available shades of blue in R.

```
library(faraway)
data(nepali)
```

I'll be using functions from `dplyr` and `ggplot2`, so those need to be loaded:

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

Each observation is a single measurement for a child; there can be multiple observations per child. I used the following code to select only the columns for child id, sex, weight, height, and age. I also used `distinct` to limit the dataset to only include one measurement for each child, the child's first measurement in the dataset.

```
nepali <- nepali %>%
  select(id, sex, wt, ht, age) %>%
  mutate(id = factor(id),
        sex = factor(sex, levels = c(1, 2),
                     labels = c("Male", "Female"))) %>%
  distinct(id, .keep_all = TRUE)
```

After this cleaning, the data looks like this:

```
head(nepali)
```

```
##      id   sex   wt   ht age
## 1 120011 Male 12.8 91.2 41
## 6 120012 Female 14.9 103.9 57
## 11 120021 Female  7.7 70.1  8
## 16 120022 Female 12.1  86.4 35
## 21 120023 Male 14.2 99.4 49
## 26 120031 Male 13.9 96.4 46
```

3.7.7 Histograms

Download a pdf of the lecture slides for this video.

Histograms show the distribution of a single variable. Therefore, `geom_histogram()` requires only one main aesthetic, `x`, the (numeric) vector for which you want

to create a histogram. For example, to create a histogram of children's heights for the Nepali dataset (Figure 3.4), run:

```
ggplot(nepali, aes(x = ht)) +
  geom_histogram()
```

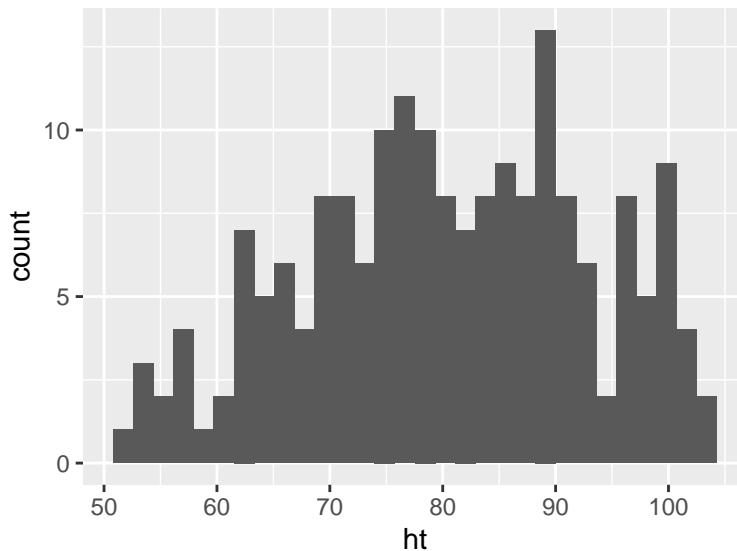


Figure 3.4: Basic example of plotting a histogram with ‘ggplot2’. This histogram shows the distribution of heights for the first recorded measurements of each child in the ‘nepali’ dataset.



If you run the code with no arguments for `binwidth` or `bins` in `geom_histogram`, you will get a message saying “`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.” This message is just saying that a default number of bins was used to create the histogram. You can use arguments to change the number of bins used, but often this default is fine. You may also get a message that observations with missing values were removed.

You can add some elements to the histogram now to customize it a bit. For example (Figure @ref()), you can add a figure title (`ggtitle`) and clearer labels for the x-axis (`xlab`). You can also change the range of values shown by the x-axis (`xlim`).

```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("Height of children") +
  xlab("Height (cm)") + xlim(c(0, 120))
```

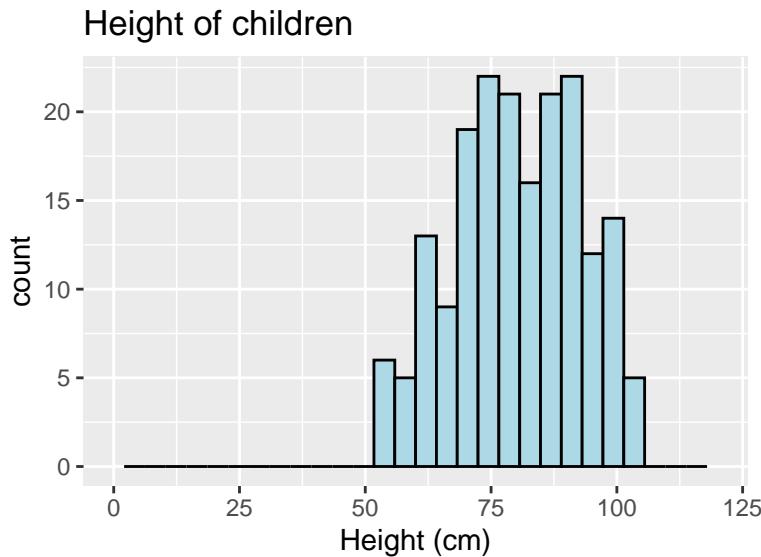


Figure 3.5: Example of adding ggplot elements to customize a histogram.

The geom `geom_histogram` also has special argument for setting the number of width of the bins used in the histogram. Figure ?? shows an example of how you can use the `bins` argument to change the number of bins that are used to make the histogram of height for the `nepali` dataset.

```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black",
                 bins = 40)
```

Similarly, the `binwidth` argument can be used to set the width of bins. Figure 3.7 shows an example of using this function to create a histogram of the Nepali children's heights with binwidths of 10 centimeters (note that this argument is set in the same units as the `x` variable).

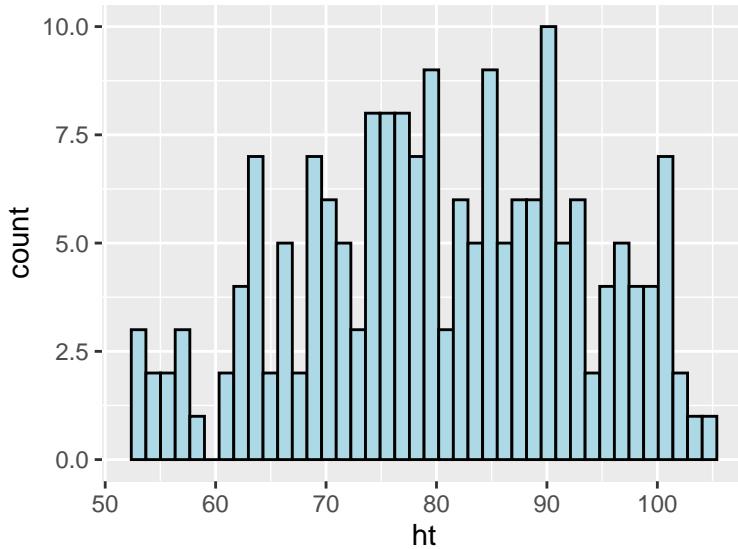


Figure 3.6: Example of using the ‘bins’ argument to change the number of bins used in a histogram.

```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black",
    binwidth = 10)
```

3.7.8 Scatterplots

A scatterplot shows how one variable changes as another changes. You can use the `geom_point` geom to create a scatterplot. For example, to create a scatterplot of height versus age for the Nepali data (Figure 3.8), you can run the following code:

```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point()
```

Again, you can use some of the options and additions to change the plot appearance. For example, to add a title, change the x- and y-axis labels, and change the color and size of the points on the scatterplot (Figure 3.9), you can run:

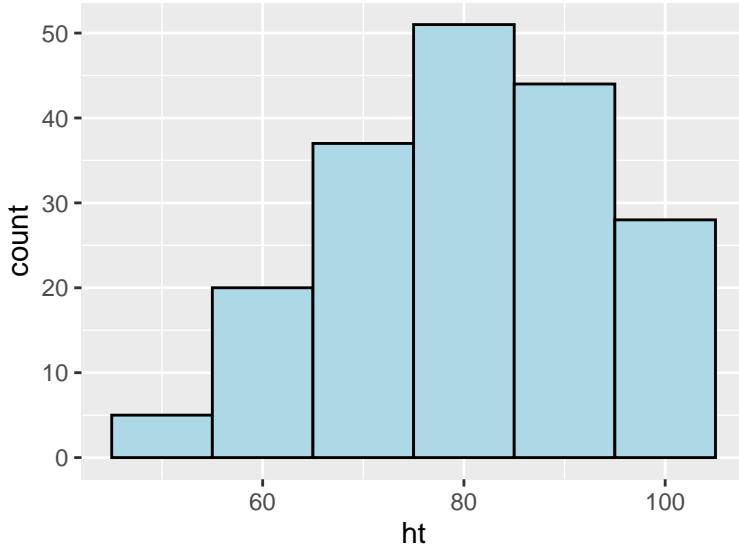


Figure 3.7: Example of using the ‘binwidth’ argument to set the width of each bin used in a histogram.

```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point(color = "blue", size = 0.5) +
  ggtitle("Weight versus Height") +
  xlab("Height (cm)") + ylab("Weight (kg)")
```

You can also try mapping another variable in the dataset to the `color` aesthetic. For example, to use color to show the sex of each child in the scatterplot (Figure 3.10), you can run:

```
ggplot(nepali, aes(x = ht, y = wt, color = sex)) +
  geom_point(size = 0.5) +
  ggtitle("Weight versus Height") +
  xlab("Height (cm)") + ylab("Weight (kg)")
```

3.7.9 Boxplots

Boxplots can be used to show the distribution of a continuous variable. To create a boxplot, you can use the `geom_boxplot` geom. To plot a boxplot for

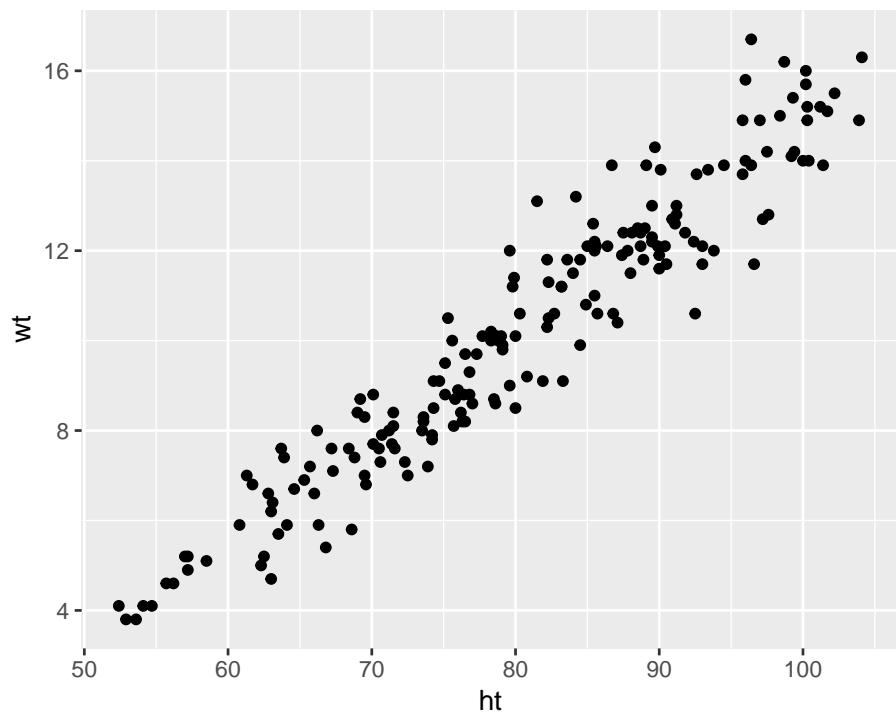


Figure 3.8: Example of creating a scatterplot. This scatterplot shows the relationship between children's heights and weights within the nepali dataset.

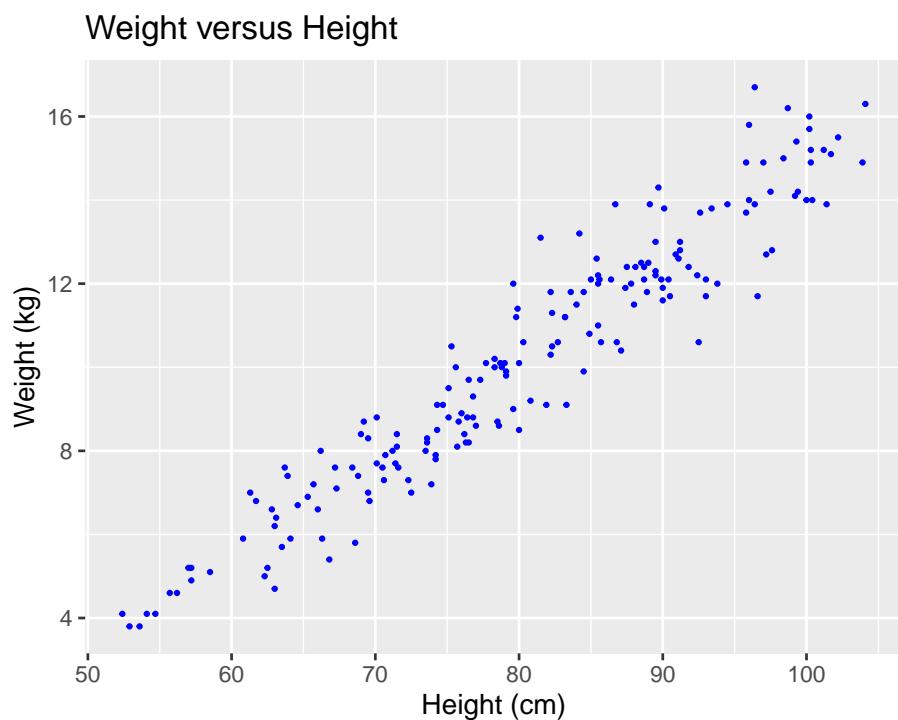


Figure 3.9: Example of adding ggplot elements to customize a scatterplot.

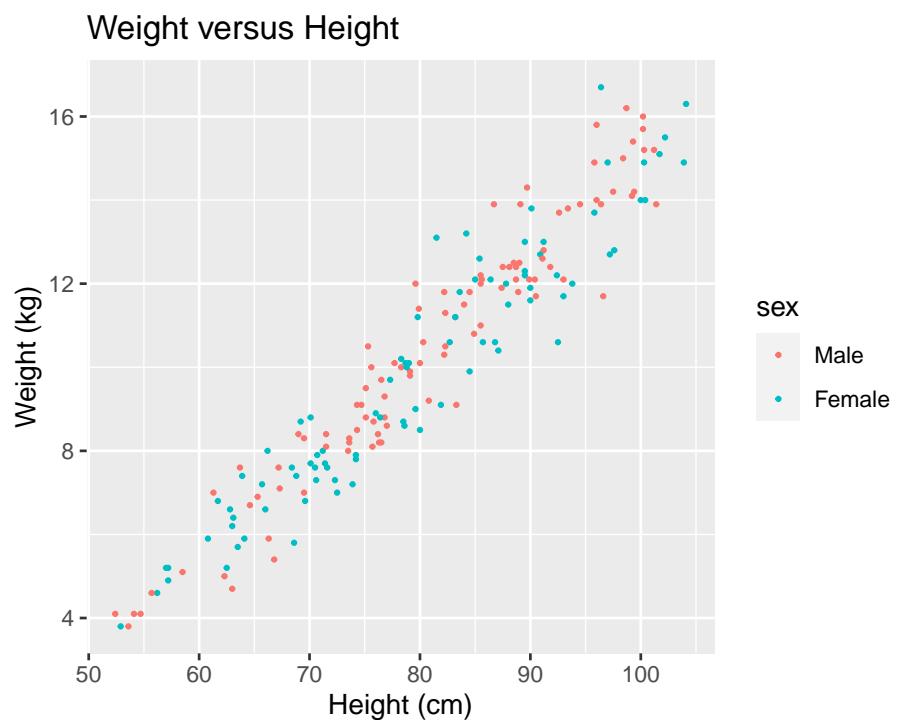


Figure 3.10: Example of mapping color to an element of the data in a scatterplot.

a single, continuous variable, you can map that variable to `y` in the `aes` call, and map `x` to the constant 1. For example, to create a boxplot of the heights of children in the Nepali dataset (Figure 3.11), you can run:

```
ggplot(nepali, aes(x = 1, y = ht)) +  
  geom_boxplot() +  
  xlab("") + ylab("Height (cm)")
```

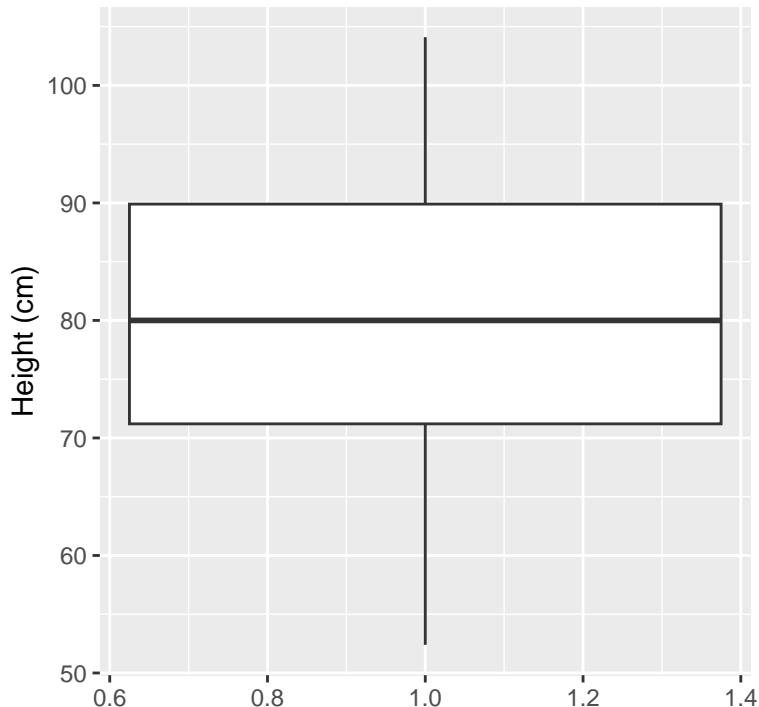


Figure 3.11: Example of creating a boxplot. The example shows the distribution of height data for children in the nepali dataset.

You can also create separate boxplots, one for each level of a factor (Figure 3.12). In this case, you'll need to include two aesthetics (`x` and `y`) when you initialize the `ggplot` object. The `y` variable is the variable for which the distribution will be shown, and the `x` variable should be a discrete (categorical or TRUE/FALSE) variable, and will be used to group the variable. This `x` variable should also be specified as the grouping variable, using `group` within the aesthetic call.

```
ggplot(nepali, aes(x = sex, y = ht, group = sex)) +
  geom_boxplot() +
  xlab("Sex") + ylab("Height (cm)")
```

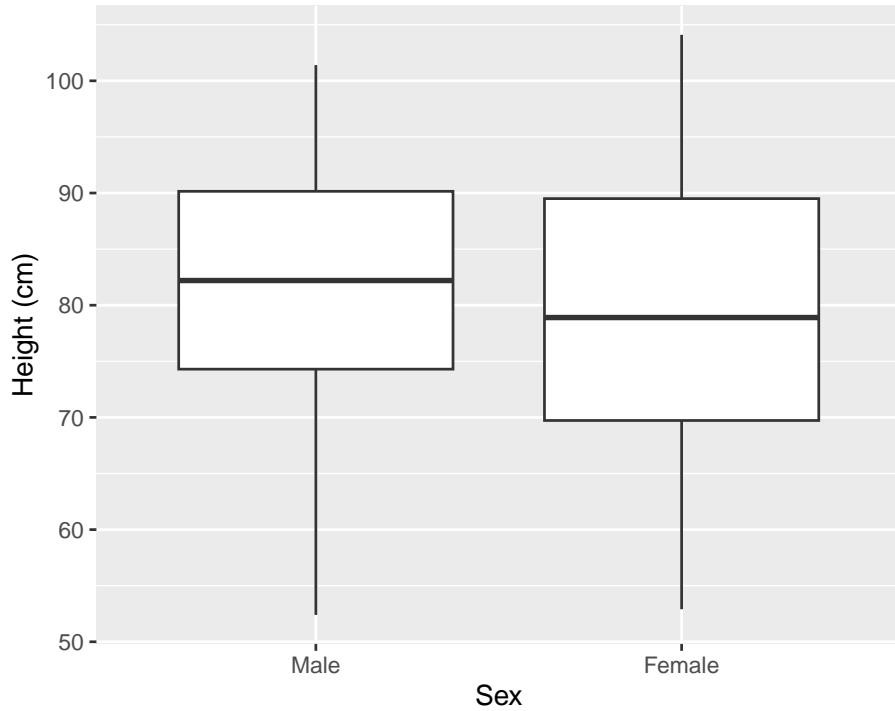


Figure 3.12: Example of creating separate boxplots, divided by a categorical grouping variable in the data.

3.8 In-course Exercise Chapter 3

3.8.1 Loading data from an R package

Pick one person to start sharing their screen.

The data we'll be using today is from a dataset called `worldcup` in the package `faraway`. Load that data so you can use it on your computer (note: you will need to load and install the `faraway` package to do this). Use the help file for the data to find out more about the dataset. Use some basic functions, like `head`, `tail`, `slice`, `colnames`, `str`, and `summary` to check out the data a bit

(if some of these you haven't seen before, remember you can always check their helpfiles!). See if you can figure out:

- What variables are included in this dataset? (Check the column names.)
- What class is each column currently? In particular, which are numbers and which are factors?

3.8.1.1 Example R code:

Load the `faraway` package using `library()` and then load the data using `data()`:

```
## Uncomment the next line if you need to install the package
# install.packages("faraway")
library(faraway)
data("worldcup")
```

Check out the help file for the `worldcup` dataset to find out more about the data. (Note: Only datasets that are parts of packages will have help files.)

```
?worldcup
```

Check out the data a bit:

```
str(worldcup)
```

```
## 'data.frame': 595 obs. of 7 variables:
## $ Team      : Factor w/ 32 levels "Algeria","Argentina",...
## $ Position: Factor w/ 4 levels "Defender","Forward",...
## $ Time     : int 16 351 180 270 46 72 138 33 21 103 ...
## $ Shots    : int 0 0 0 1 2 0 0 0 5 0 ...
## $ Passes   : int 6 101 91 111 16 15 51 9 22 38 ...
## $ Tackles  : int 0 14 6 5 0 0 2 0 0 1 ...
## $ Saves    : int 0 0 0 0 0 0 0 0 0 0 ...
```

```
head(worldcup)
```

```
##          Team   Position Time Shots Passes Tackles Saves
## Abdoun      Algeria Midfielder 16     0     6      0     0
## Abe         Japan    Midfielder 351    0    101     14     0
## Abidal     France     Defender 180    0     91      6     0
## Abou Diaby  France Midfielder 270    1    111      5     0
## Aboubakar Cameroon    Forward  46     2     16      0     0
## Abreu       Uruguay    Forward  72     0     15      0     0
```

```
tail(worldcup)
```

```
##          Team   Position Time Shots Passes Tackles Saves
## van Bommel Netherlands Midfielder 540    2    307     31     0
## van Bronckhorst Netherlands   Defender 540    1    271     10     0
## van Persie  Netherlands    Forward 479    14    108      1     0
## von Bergen Switzerland   Defender 234    0     79      3     0
## Alvaro Pereira Uruguay Midfielder 409    6    140     17     0
## Ozil        Germany Midfielder 497    7    266      3     0
```

```
colnames(worldcup)
```

```
## [1] "Team"      "Position"   "Time"      "Shots"     "Passes"    "Tackles"   "Saves"
```

```
summary(worldcup)
```

	Team	Position	Time	Shots
## Slovakia : 21	Defender :188	Min. : 1.0	Min. : 0.000	
## Uruguay : 21	Forward :143	1st Qu.: 88.0	1st Qu.: 0.000	
## Argentina: 20	Goalkeeper: 36	Median :191.0	Median : 1.000	
## Cameroon : 20	Midfielder:228	Mean :208.9	Mean : 2.304	
## Chile : 20		3rd Qu.:270.0	3rd Qu.: 3.000	
## Paraguay : 20		Max. :570.0	Max. :27.000	
## (Other) :473				
	Passes	Tackles	Saves	
## Min. : 0.00	Min. : 0.000	Min. : 0.0000		
## 1st Qu.: 29.00	1st Qu.: 1.000	1st Qu.: 0.0000		
## Median : 61.00	Median : 3.000	Median : 0.0000		
## Mean : 84.52	Mean : 4.192	Mean : 0.6672		
## 3rd Qu.:115.50	3rd Qu.: 6.000	3rd Qu.: 0.0000		
## Max. :563.00	Max. :34.000	Max. :20.0000		
##				

3.8.2 Exploring the data using simple statistics and `summarize`

Rotate to someone else to share their screen.

Then, try checking out the data using some basic commands for simple statistics, like `mean()`, `range()`, `max()`, and `min()`, as well as the `summarize` and `group_by` functions from the `dplyr` package. Try to answer the following questions:

- What is the mean number of saves that players made?
- What is the mean number of saves just among the goalkeepers?
- Did players from any position other than goalkeeper make a save?
- How many players were there in each position?
- How many forwards were there on each team? Which team had the most shots in total among all its forwards?
- Which team(s) had the defender with the most tackles?

If you have extra time, continuing using the “Data Wrangling” cheatsheet to come up with some other ideas for how you can explore this data, and write up and test code to do that.

3.8.2.1 Example R code:

To calculate the mean number of saves among all the players, use the `mean` function, either by itself or within a `summarize` call:

```
mean(worldcup$Saves)
```

```
## [1] 0.6672269
```

```
worldcup %>%
  summarize(mean_saves = mean(Saves))
```

```
##   mean_saves
## 1  0.6672269
```

There are a few ways to figure out the mean number of saves just among the goalkeepers. One way is to filter the dataset to only goalies and then use `summarize` to calculate the mean number of saves in this filtered subset of the data:

```
worldcup %>%
  filter(Position == "Goalkeeper") %>%
  summarize(mean_saves = mean(Saves))

##   mean_saves
## 1 11.02778
```

The next question is if players from any position other than goalkeeper made a save. One way to figure this out is to group the data by position and then summarize the maximum number of saves. Based on this, it looks like there were not saves from players in any position except goalie:

```
worldcup %>%
  group_by(Position) %>%
  summarize(max_saves = max(Saves))

## # A tibble: 4 x 2
##   Position   max_saves
##   <fct>       <int>
## 1 Defender      0
## 2 Forward       0
## 3 Goalkeeper    20
## 4 Midfielder    0
```

To figure out how many players were there in each position, you can group the data by position and then use the `count` function from `dplyr` to count the number of observations in each group:

```
worldcup %>%
  group_by(Position) %>%
  count()

## # A tibble: 4 x 2
## # Groups:   Position [4]
##   Position     n
##   <fct>   <int>
## 1 Defender    188
## 2 Forward     143
## 3 Goalkeeper   36
## 4 Midfielder  228
```

For the next set of questions, you can filter the data to only Forwards, then group by team to use `summarize` to count up the number of Forwards on each team. You can also use the same `summarize` call to figure out the total number of shots by all Forwards on each team. To figure out which team had the most shots in total among all its forwards, you can use the `arrange` function to reorder the data from the team with the most total shots to the least. It turns out that Uruguay had the most shots by forwards on its team, with a total of 46 shots.

```
worldcup %>%
  filter(Position == "Forward") %>%
  group_by(Team) %>%
  summarize(n_forwards = n(),
            total_forward_shots = sum(Shots)) %>%
  arrange(desc(total_forward_shots))
```

```
## # A tibble: 32 x 3
##   Team      n_forwards total_forward_shots
##   <fct>     <int>           <int>
## 1 Uruguay      5              46
## 2 Argentina    6              45
## 3 Germany       6              41
## 4 Netherlands   5              34
## 5 Spain         3              33
## 6 Ghana          5              32
## 7 Portugal      4              28
## 8 Paraguay      5              25
## 9 Brazil         4              23
## 10 USA          5              21
## # i 22 more rows
```

To figure out which team(s) had the defender with the most tackles, you can filter to only defenders and then use the `top_n` function to identify the players with the top number of tackles. It turns out these players were on the England, Germany, and Chile teams.

```
worldcup %>%
  filter(Position == "Defender") %>%
  top_n(n = 1, wt = Tackles)
```

	Team	Position	Time	Shots	Passes	Tackles	Saves	
##	Glen Johnson	England	Defender	357	3	173	17	0

```
## Lahm           Germany Defender  540      0     360      17      0
## Vidal          Chile  Defender  306      6     178      17      0
```

3.8.3 Exploring the data using logical statements

Rotate to someone else to share their screen.

Then, try checking out the data using logical statements and some of the `dplyr` code we covered in the last chapter (`filter` and `arrange`, for example), to help you answer the following questions:

- What is the range of time that players spent in the game?
- Which player or players played the most time in this World Cup?
- How many players are goalies in this dataset?
- Create a new R object named `brazil_players` that is limited to the players in this dataset that are (1) on the Brazil team and (2) not goalies.

If you have additional time, look over the “Data Manipulation” cheatsheet available in RStudio’s Help section. Make a list of questions you would like to figure out from this example data, and start to plan out how you might be able to answer those questions using functions from `dplyr`. Write the related code and see if it works.

3.8.3.1 Example R code:

To figure out the range of time, you could use `arrange` twice, once with `desc` and once without, to figure out the maximum and minimum values

```
# Minimum time
arrange(worldcup, Time) %>%
  select(Time) %>%
  slice(1)
```

```
##           Time
## Barron      1
```

```
# Maximum time
arrange(worldcup, desc(Time)) %>%
  select(Time) %>%
  slice(1)
```

```
##           Time
## Arevalo Rios  570
```

Later, we will learn about the `n()` function, which you can use within piped code to represent the total number of rows in the dataframe. If you'd like to get the full range of the `Time` column in one pipeline of code, you can use `n()` as a reference within `slice`, to pull both the first and last rows of the dataframe:

```
arrange(worldcup, Time) %>%
  select(Time) %>%
  slice(c(1, n()))
```

```
##           Time
## Barron      1
## Muslera    570
```

Finally, you could also use `min()` and `max()` functions to get the minimum and maximum values of the `Time` column in the `worldcup` dataframe (remember that you can use the `dataframe$column_name` notation to pull a column from a dataframe). Similarly, you there is a function called `range()` you could use to find out the range of time these players played in the World Cup.

```
range(worldcup$Time)
```

```
## [1]  1 570
```

To figure out which player or players played for the most time, there are a few approaches you can take. Here I'm showing two: (1) using `filter` from the `dplyr` package to filter down to rows where where the `Time` for that row equals the maximum play time that you determined from an earlier task (570 minutes); and (2) using the `top_n` function from `dplyr` to pick out the rows with the maximum value (`n = 1`) of the `Time` column (see the help file for `top_n` if you are unfamiliar with this function; we have not covered it in class yet).

```
worldcup %>%
  filter(Time == 570)
```

	Team	Position	Time	Shots	Passes	Tackles	Saves
## Arevalo Rios	Uruguay	Midfielder	570	5	195	21	0
## Maxi Pereira	Uruguay	Midfielder	570	5	182	15	0
## Muslera	Uruguay	Goalkeeper	570	0	75	0	16

```
worldcup %>%
  top_n(n = 1, wt = Time)
```

	Team	Position	Time	Shots	Passes	Tackles	Saves	
## 1	Arevalo Rios	Uruguay	Midfielder	570	5	195	21	0
## 2	Maxi Pereira	Uruguay	Midfielder	570	5	182	15	0
## 3	Muslera	Uruguay	Goalkeeper	570	0	75	0	16

Note: You may have noticed that you lost the players names when you did this using the `dplyr` pipechain. That's because `dplyr` functions convert the data to a dataframe format that does not include rownames. If you want to keep players' names, you can use a function from the `tibble` package called `rownames_to_column` to move those names from the rownames of the data into a column in the dataframe. Use the `var` parameter of this function to specify what you want the new column to be named. For example:

```
library(tibble)
worldcup %>%
  rownames_to_column(var = "Name") %>%
  filter(Time == 570)
```

	Name	Team	Position	Time	Shots	Passes	Tackles	Saves
## 1	Arevalo Rios	Uruguay	Midfielder	570	5	195	21	0
## 2	Maxi Pereira	Uruguay	Midfielder	570	5	182	15	0
## 3	Muslera	Uruguay	Goalkeeper	570	0	75	0	16

There are a few ways to figure out how many players are goalies in this dataset. One way is to use `sum()` on a logical vector of whether the player's position is "Goalkeeper":

```
is_goalie <- worldcup$Position == "Goalkeeper"
sum(is_goalie)

## [1] 36
```

Another way is to use `filter` from `dplyr`, along with a logical statement, to filter the data to only players with the position of "Goalkeeper", and then pipe that filtered subset into the `nrow` function to count the number of rows in the filtered dataframe:

```
worldcup %>%
  filter(Position == "Goalkeeper") %>%
  nrow()
```

```
## [1] 36
```

Next, create a new R object named `brazil_players` that is limited to the players in this dataset that are (1) on the Brazil team and (2) not goalies. You can use a logical statement to filter to rows that meet both these conditions by joining two logical statements in the `filter` function with an `&`:

```
brazil_players <- worldcup %>%
  filter(Team == "Brazil" & Position != "Goalkeeper")
head(brazil_players)
```

	Team	Position	Time	Shots	Passes	Tackles	Saves
## Baptista	Brazil	Midfielder	82	0	42	1	0
## Daniel Alves	Brazil	Defender	310	11	215	6	0
## Elano	Brazil	Midfielder	140	5	57	6	0
## Fabiano	Brazil	Forward	418	9	89	4	0
## Gilberto	Brazil	Defender	33	0	6	4	0
## Gilberto Silva	Brazil	Midfielder	450	3	299	11	0

3.8.4 Exploring the data using basic plots #1

Use some basic plots to check out this data. Try the following:

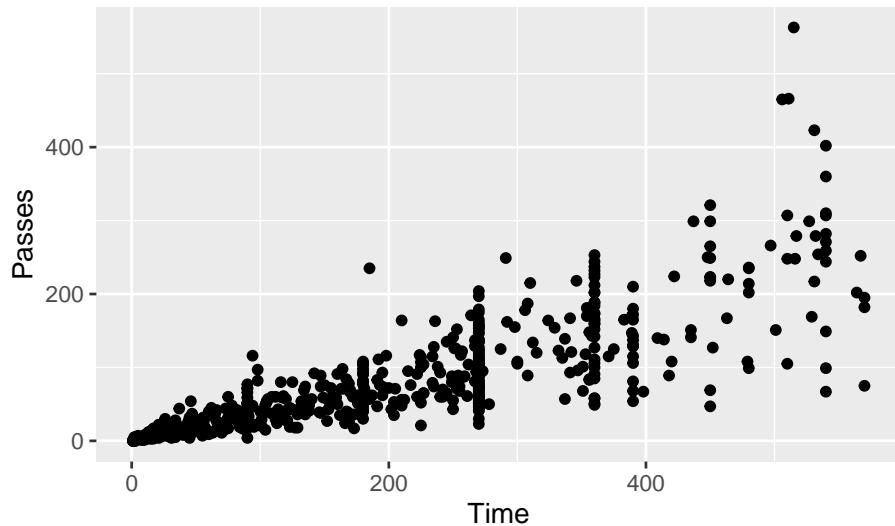
- Create a scatterplot of the `worldcup` data, where each point is a player, the x-axis shows the amount of time the player played in the World Cup, and the y-axis shows the number of passes the player had. Try writing the code both with and without “piping in” the data you want to plot into the `ggplot` function.
- Create the same scatterplot, but have each point in the scatterplot show that player’s position using some aesthetic besides the x or y position (e.g., color, point shape). Add “rug plots” to the margins.
- Create a scatterplot of number of shots (x-axis) versus number of tackles (y-axis) for **just** players on one of the four teams that made the semi-finals (Spain, Netherlands, Germany, Uruguay). Use color to show player’s position and shape to show player’s team. (Hint: you will want to use some `dplyr` code to clean the data before plotting to do this.)

- Create a scatterplot of player time versus passes. Use color to show whether the player was on one of the top 4 teams or not. (Hint: Again, you'll want to use some `dplyr` code before plotting to do this.) For an extra challenge, also try adding each player's name on top of each point. (Hint: check out the `rownames_to_column` function from the `tibble` package to help with this.)
- Did you notice any interesting features of the data when you did any of the graphs in this section?

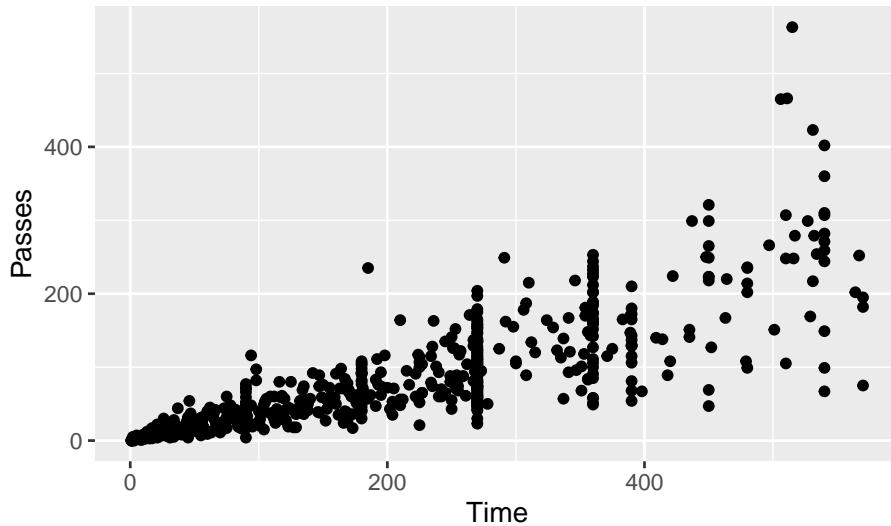
3.8.4.1 Example R code:

Create a scatterplot of Time versus Passes.

```
# Without piping
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes))
```

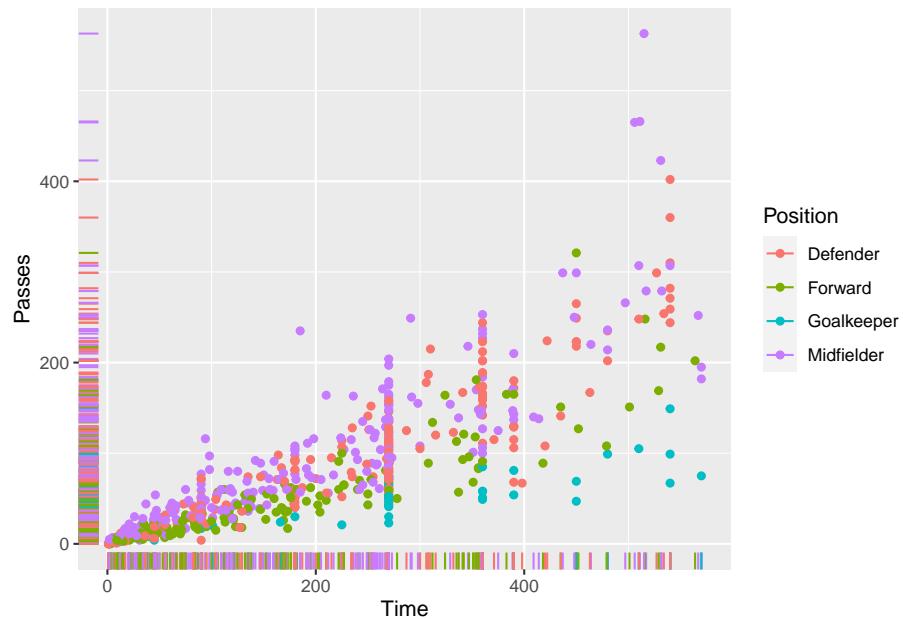


```
# With piping
worldcup %>%
  ggplot() +
  geom_point(mapping = aes(x = Time, y = Passes))
```



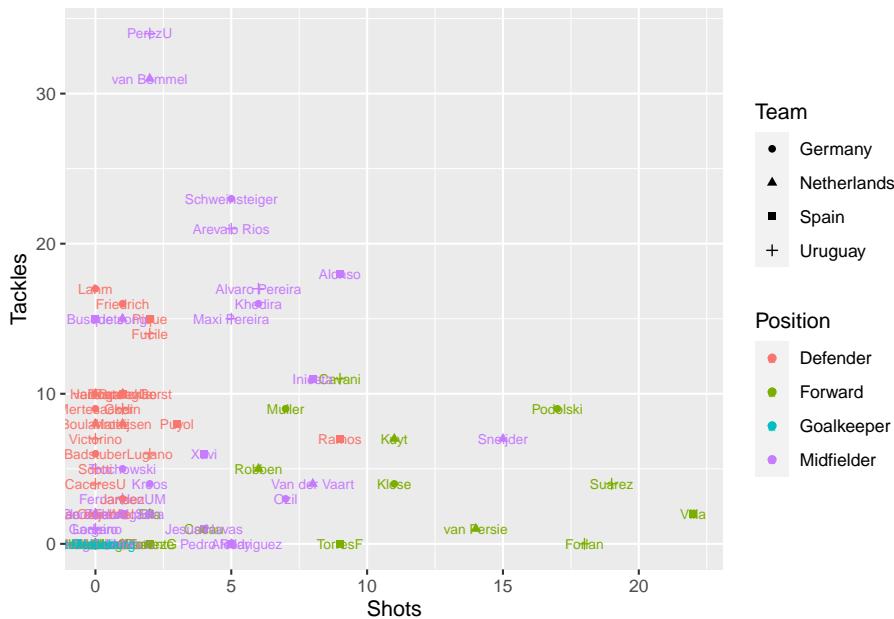
Create the same scatterplot, but have each point in the scatterplot show that player's position.

```
ggplot(worldcup,  
       mapping = aes(x = Time, y = Passes, color = Position)) +  
  geom_point() +  
  geom_rug()
```



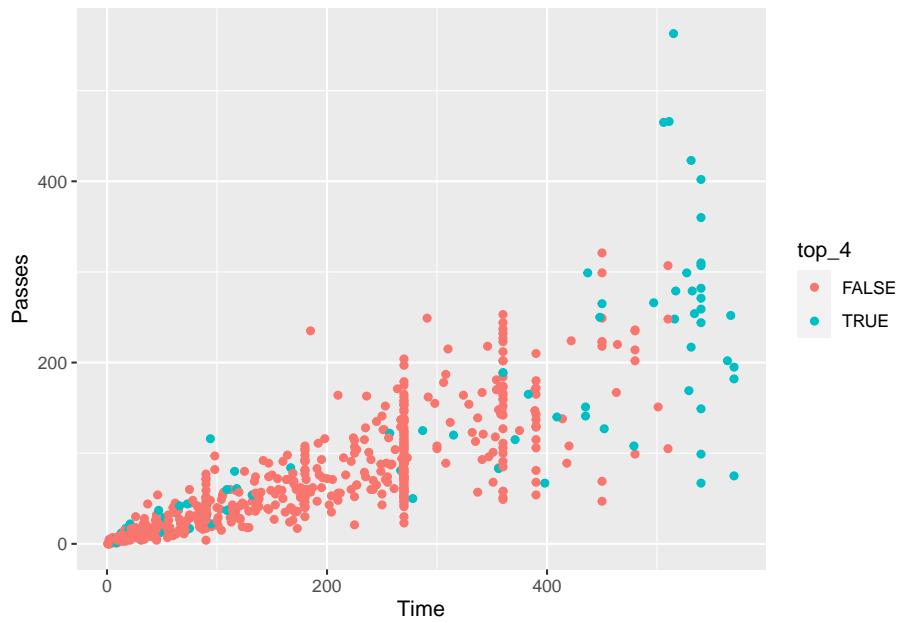
Create a scatterplot of number of shots (x-axis) versus number of tackles (y-axis) for **just** players on one of the four teams that made the semi-finals (Spain, Netherlands, Germany, Uruguay). Use color to show player's position and shape to show player's team. For an extra challenge, also try adding each player's name on top of each point.

```
worldcup %>%
  rownames_to_column(var = "Name") %>%
  filter(Team %in% c("Spain", "Netherlands", "Germany", "Uruguay")) %>%
  ggplot() +
  geom_point(aes(x = Shots, y = Tackles, color = Position, shape = Team)) +
  geom_text(mapping = aes(x = Shots, y = Tackles,
                         color = Position, label = Name),
            size = 2.5)
```



Create a scatterplot of player time versus passes. Use color to show whether the player was on one of the top 4 teams or not.

```
worldcup %>%
  mutate(top_4 = Team %in% c("Spain", "Netherlands", "Germany", "Uruguay")) %>%
  ggplot() +
  geom_point(aes(x = Time, y = Passes, color = top_4))
```



3.8.5 Exploring the data using basic plots #2

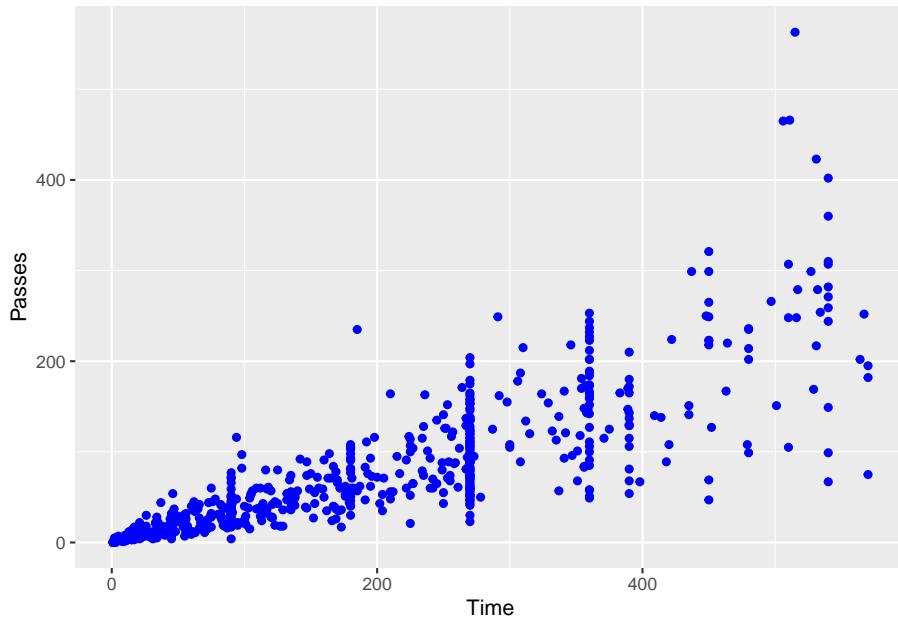
Go back to the code you used in the previous section to create a scatterplot of the `worldcup` data, where each point is a player, the x-axis shows the amount of time the player played in the World Cup, and the y-axis shows the number of passes the player had. Try the following modifications:

- Make all the points blue.
- Google “R colors” to find a list of color names in R. Pick your favorite and make all the points in the scatterplot that color.
- Change the size of the points to make them smaller (hint: check out the `size` aesthetic).
- Make it so the color of the points shows the player’s position and all the points are slightly transparent.
- Change the title of the x-axis to “Time (minutes)” and the y-axis to “Number of passes”.
- Add the title “World Cup statistics” and the subtitle “2010 World Cup”.

3.8.5.1 Example R code:

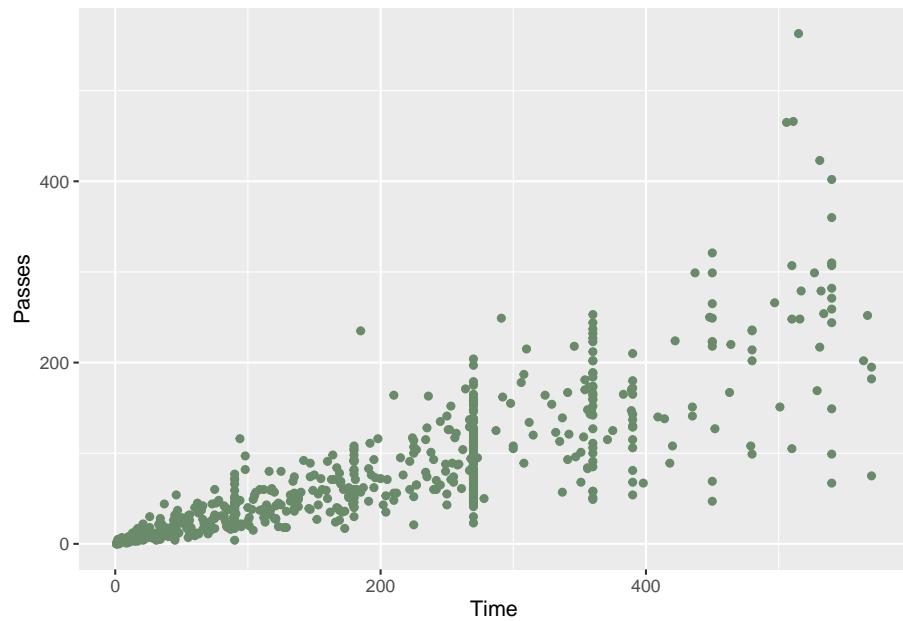
Make all the points blue.

```
ggplot(worldcup) +  
  geom_point(mapping = aes(x = Time, y = Passes),  
             color = "blue")
```



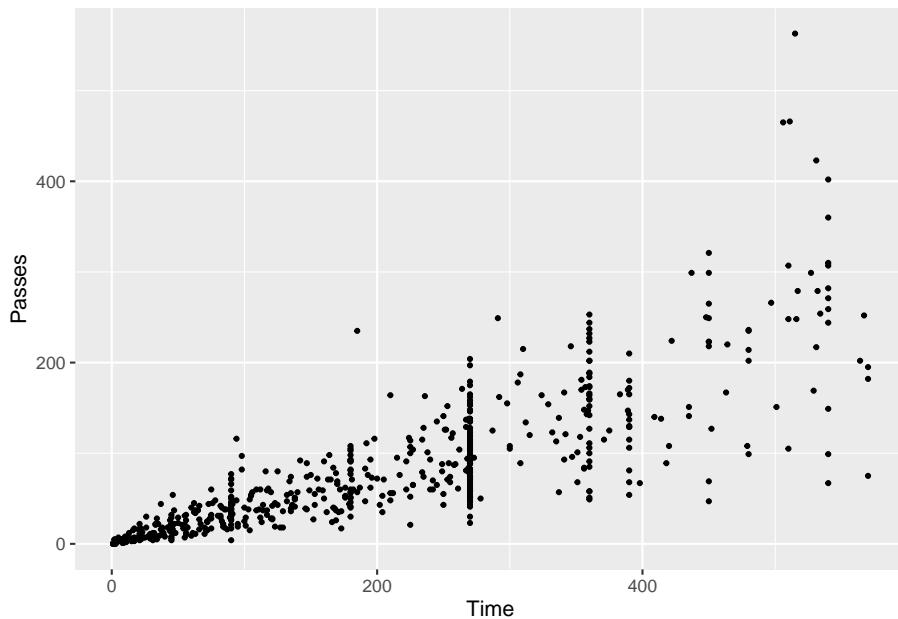
Google “R colors” to find a list of color names in R. Pick your favorite and make all the points in the scatterplot that color.

```
# Make the points "darkseagreen4"  
ggplot(worldcup) +  
  geom_point(mapping = aes(x = Time, y = Passes),  
             color = "darkseagreen4")
```



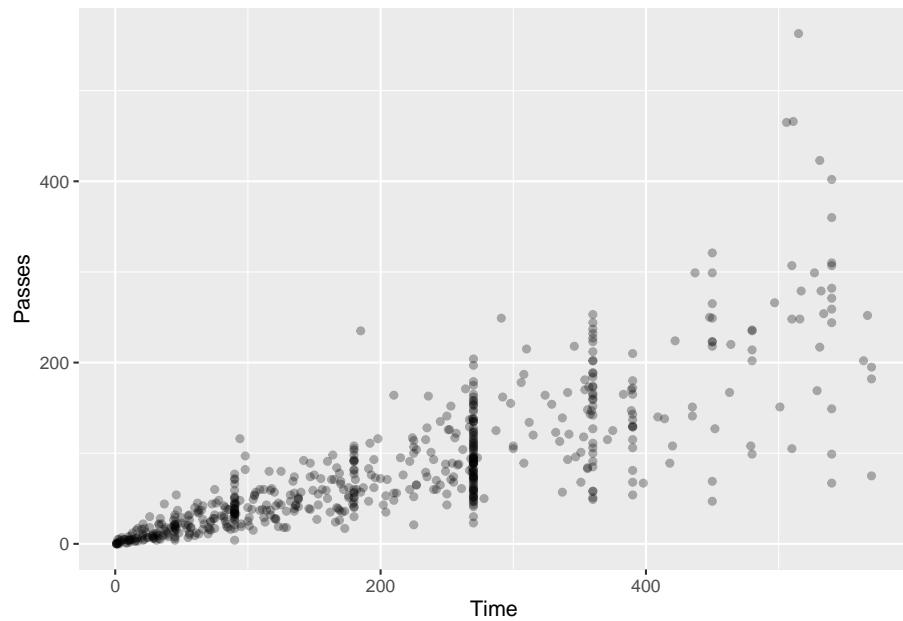
Change the size of the points to make them smaller (hint: check out the `size` aesthetic).

```
ggplot(worldcup) +  
  geom_point(mapping = aes(x = Time, y = Passes),  
             size = 0.8)
```



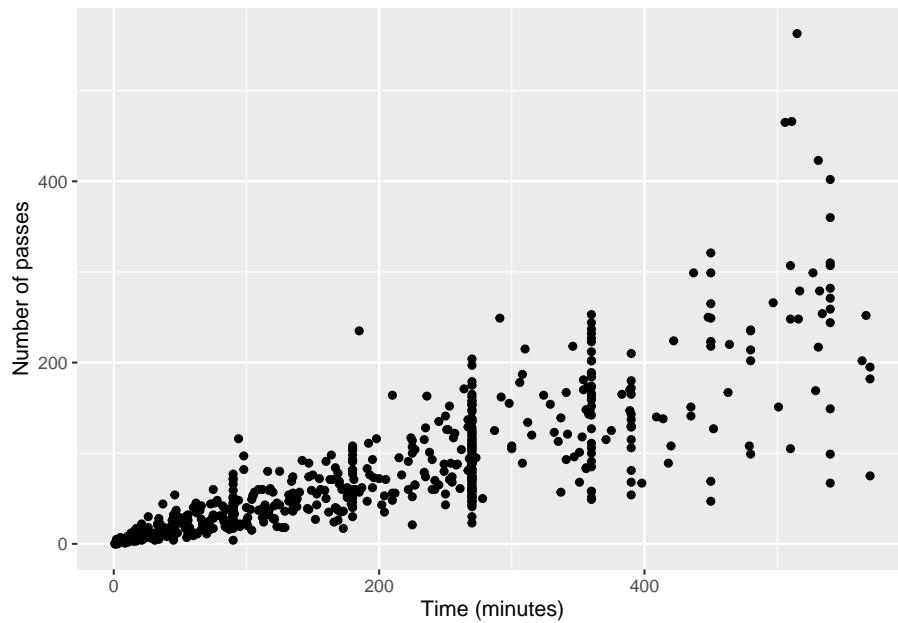
Make it so the color of the points shows the player's position and all the points are slightly transparent.

```
ggplot(worldcup) +  
  geom_point(mapping = aes(x = Time, y = Passes),  
            alpha = 0.3)
```



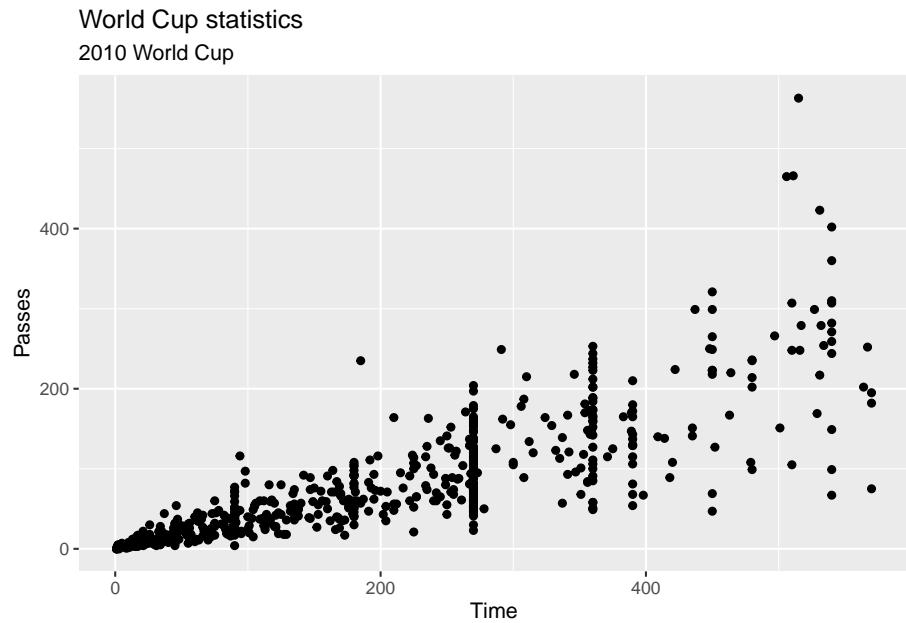
Change the title of the x-axis to “Time (minutes)” and the y-axis to “Number of passes”.

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes)) +
  labs(x = "Time (minutes)", y = "Number of passes")
```



Add the title “World Cup statistics” and the subtitle “2010 World Cup”.

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes)) +
  gtitle("World Cup statistics",
         subtitle = "2010 World Cup")
```



3.8.6 Exploring the data using basic plots #3

Try out creating some plots using the “statistical” geoms to check out this data.
Try the following:

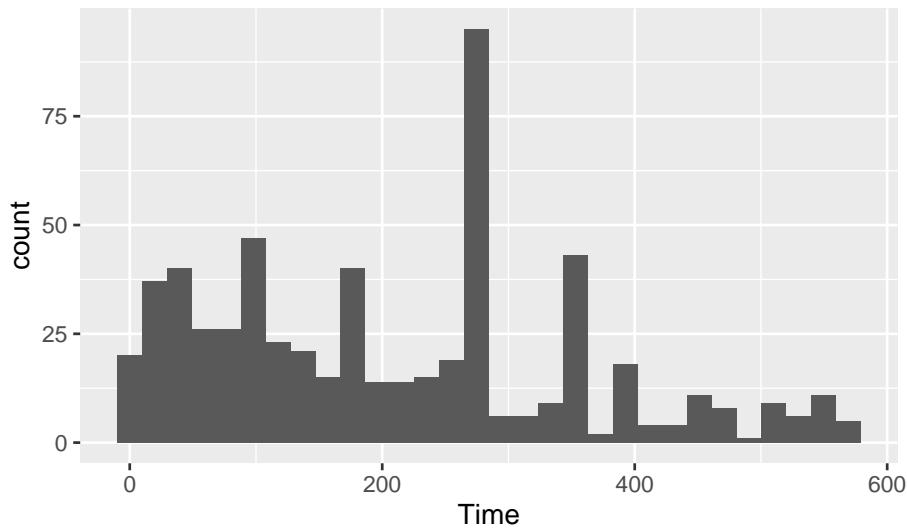
- Plot histograms of all the numeric variables (`Time`, `Shot`, `Passes`, `Tackles`, `Saves`) in the dataset.
- Try customizing the number of bins used for one of the histograms plotted in the previous step.
- Try using constant values for some of the aesthetics (e.g., customize the color and the fill) of the histogram created in the previous step.
- Create a boxplot of `Shots` by position.
- Create a `top_teams` subset with just the four teams that made the semi-finals in the 2010 World Cup (Spain, the Netherlands, Germany, and Uruguay). Plot boxplots of `Shots` and `Saves` by team for just these teams.
- Create a histogram using data only from the four top teams for the amount of time each player played. Use the color aesthetic of the histogram to show team.

3.8.6.1 Example R code

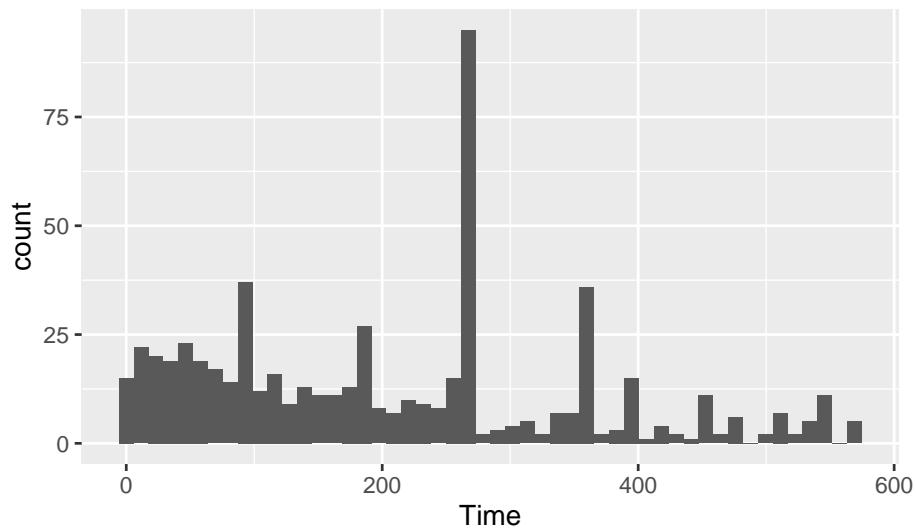
Use histograms to explore the distribution of different variables. If you want to change the number of bins in the histogram, try playing around with the `bins`

and `binwidth` arguments. You can use the `bins` argument to say how many bins you want (e.g., `bins = 50`). You can use the `binwidth` argument to say how wide you want the bins to be (e.g., `binwidth = 10` if you wanted bins to be 10 units wide, in the units of the variable mapped to the `x` aesthetic). Try using `fill` and `color` to change the appearance of the plot. Google “R colors” and search the images to find links to listings of different R colors.

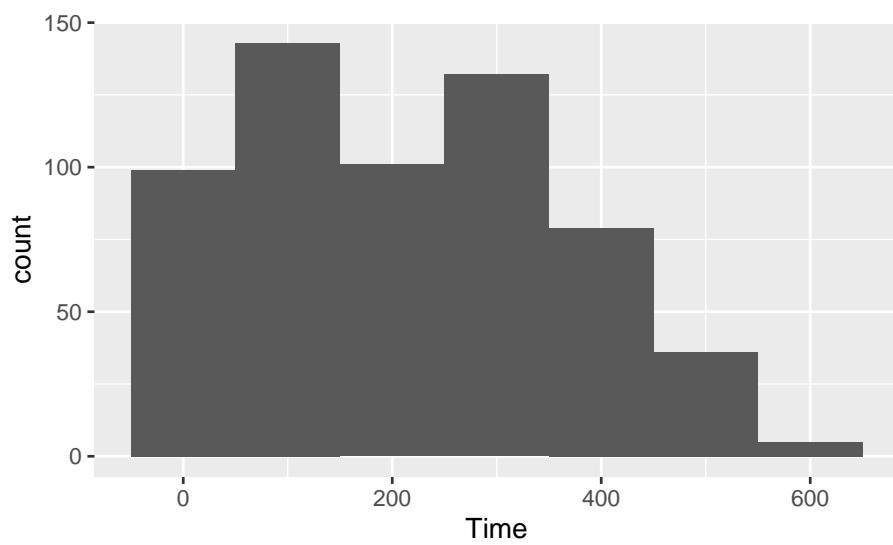
```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram()
```



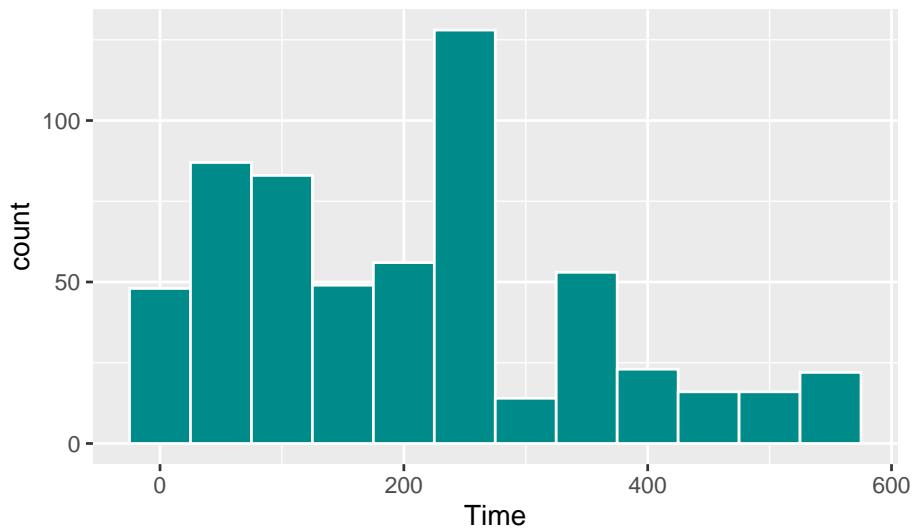
```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram(bins = 50)
```



```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram(binwidth = 100)
```

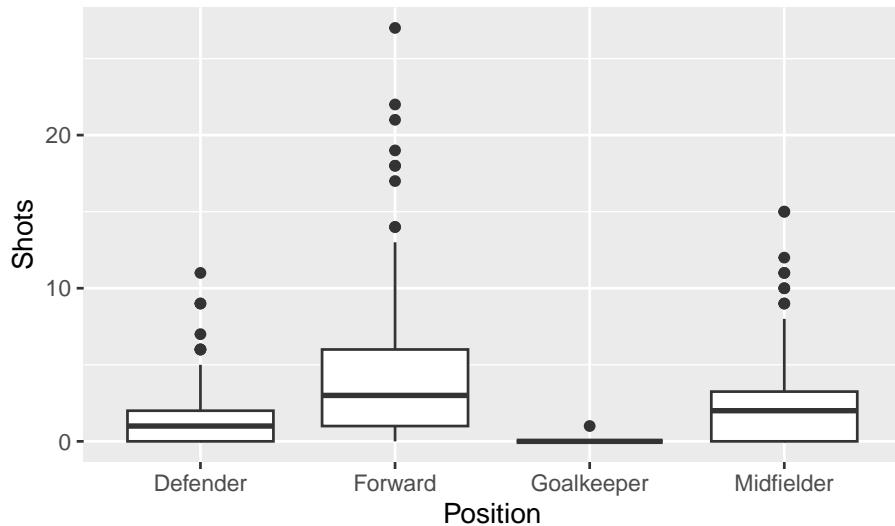


```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram(binwidth = 50, color = "white", fill = "cyan4")
```



To create a boxplot of Shots by Position, you can use `geom_boxplot`:

```
ggplot(worldcup, aes(x = Position, y = Shots)) +  
  geom_boxplot()
```

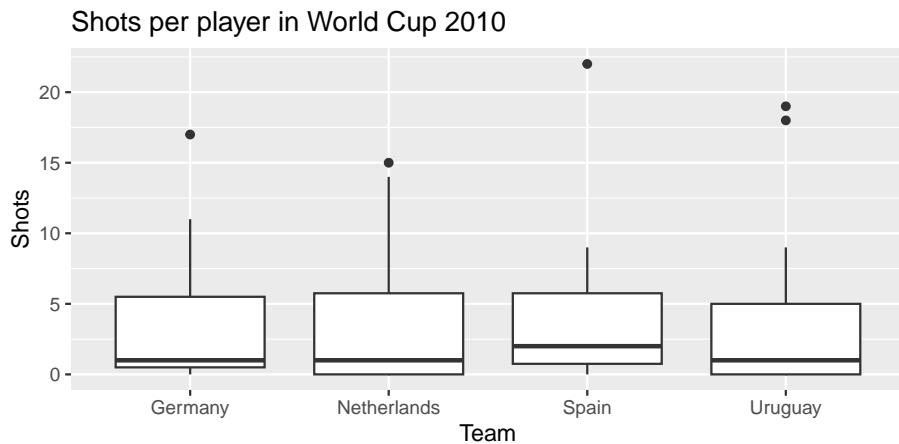


The top four teams in this World Cup were Spain, the Netherlands, Germany, and Uruguay. Create a subset with just the data for these four teams:

```
top_teams <- worldcup %>%
  filter(Team %in% c("Spain", "Netherlands", "Germany", "Uruguay"))
```

Now, you can plot the boxplots, mapping Team to the x aesthetic and Shots to the y aesthetic:

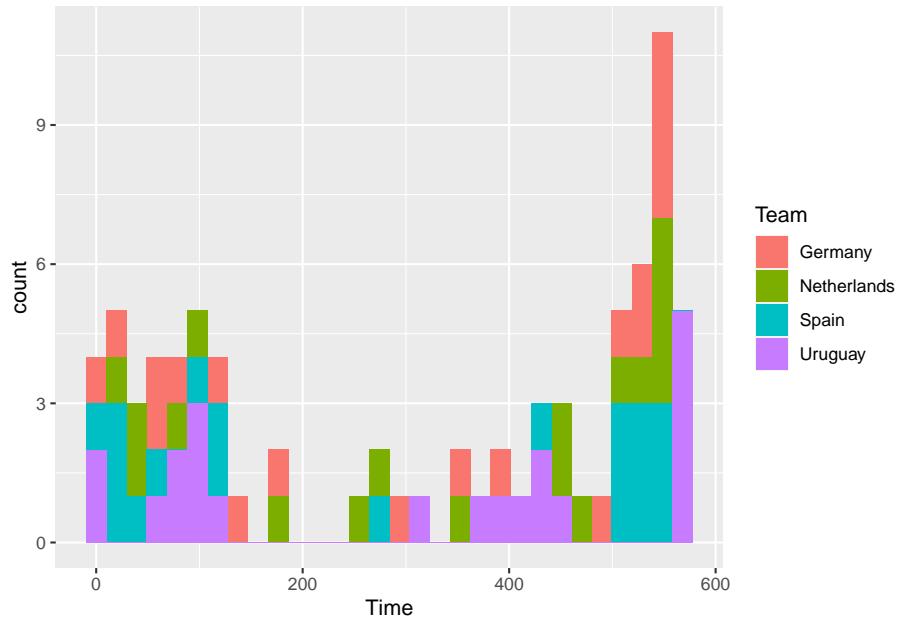
```
ggplot(top_teams, aes(x = Team, y = Shots)) +
  geom_boxplot() +
  ggtitle("Shots per player in World Cup 2010")
```



Create a histogram using data only from the four top teams for the amount of time each player played. Use the color aesthetic of the histogram to show team.

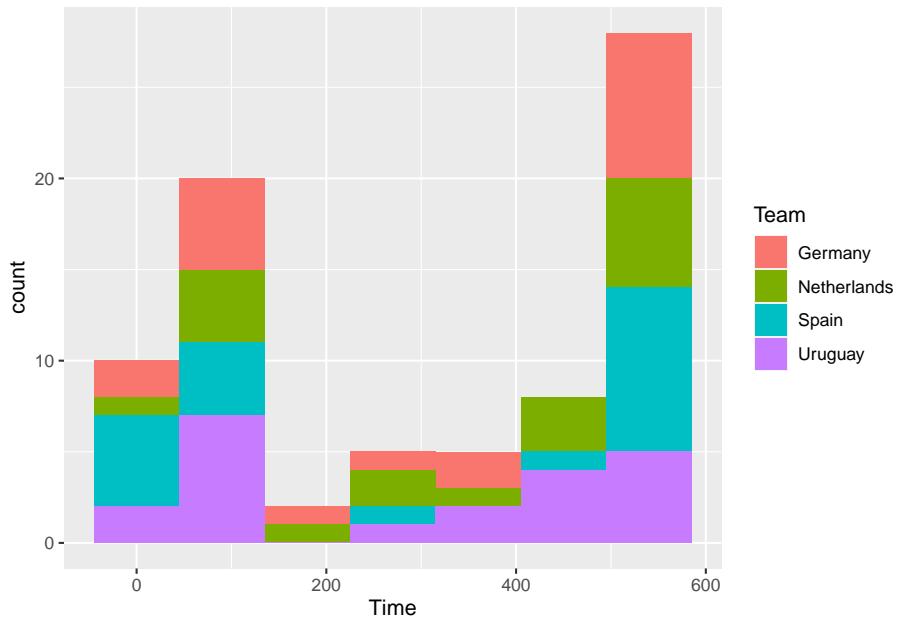
```
ggplot(data = top_teams) +
  geom_histogram(aes(x = Time, fill = Team))

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Note that you can also explore other values for `geom_histogram` arguments. For example, you could change the binwidths to be 90 minutes (since games are 90 minutes).

```
ggplot(data = top_teams) +  
  geom_histogram(aes(x = Time, fill = Team), binwidth = 90)
```



Chapter 4

Reporting data results #1

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

4.1 Guidelines for good plots

Download a pdf of the lecture slides for this video.

There are a number of very thoughtful books and articles about creating graphics that effectively communicate information. Some of the authors I highly recommend (and from whose work I've pulled the guidelines for good graphics we'll talk about this week) are:

- Edward Tufte
- Howard Wainer
- Stephen Few
- Nathan Yau

You should plan, in particular, to read *The Visual Display of Quantitative Information* by Edward Tufte before you graduate.

This week, we'll focus on six guidelines for good graphics, based on the writings of these and other specialists in data display. The guidelines are:

1. Aim for high data density.
2. Use clear, meaningful labels.
3. Provide useful references.
4. Highlight interesting aspects of the data.

5. Make order meaningful.
6. When possible, use small multiples.

For the examples, I'll use `dplyr` for data cleaning and, for plotting, the packages `ggplot2`, `gridExtra`, and `ggthemes`.

```
library(tidyverse) ## Loads `dplyr` and `ggplot2`
library(gridExtra)
library(ggthemes)
```

You can load the data for today's examples with the following code:

```
library(faraway)
data(nepali)
data(worldcup)

library(dlnm)
data(chicagoNMMAPS)
chic <- chicagoNMMAPS
chic_july <- chic %>%
  filter(month == 7 & year == 1995)
```

4.2 High data density

Guideline 1: **Aim for high data density.**

Download a pdf of the lecture slides for this video.

You should try to increase, as much as possible, the **data to ink ratio** in your graphs. This is the ratio of “ink” providing information to all ink used in the figure. One way to think about this is that the only graphs you make that use up a lot of your printer’s ink should be packed with information.

The two graphs in Figure 4.1 show the same information, but use very different amounts of ink. Each shows the number of players in each of four positions in the `worldcup` dataset. Notice how, in the plot on the right, a single dot for each category shows the same information that a whole filled bar is showing on the left. Further, the plot on the right has removed the gridded background, removing even more “ink”.

Figure 4.2 gives another example of two plots that show the same information but with very different data densities. This figure uses the `chicagoNMMAPS`

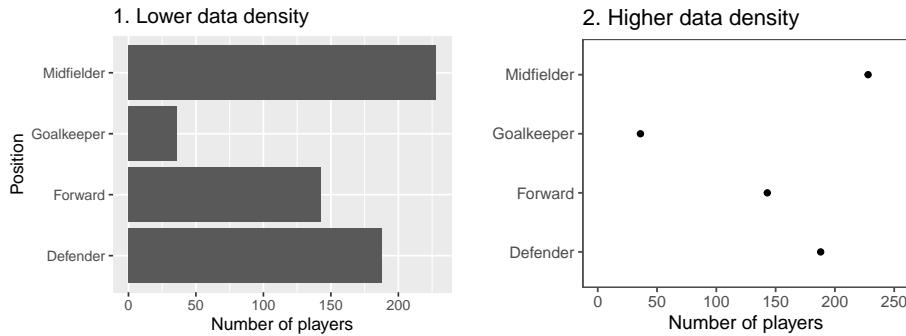


Figure 4.1: Example of plots with lower (left) and higher (right) data-to-ink ratios. Each plot shows the number of players in each position in the `worldcup` dataset from the `faraway` package.

data from the `dlnm` package, which includes daily mortality, weather, and air pollution data for Chicago, IL. Both plots show daily mortality counts during July 1995, when a very severe heat wave hit Chicago. Notice how many of the elements in the plot on the left, including the shading under the mortality time series and the colored background and grid lines, are unnecessary for interpreting the message from the data.

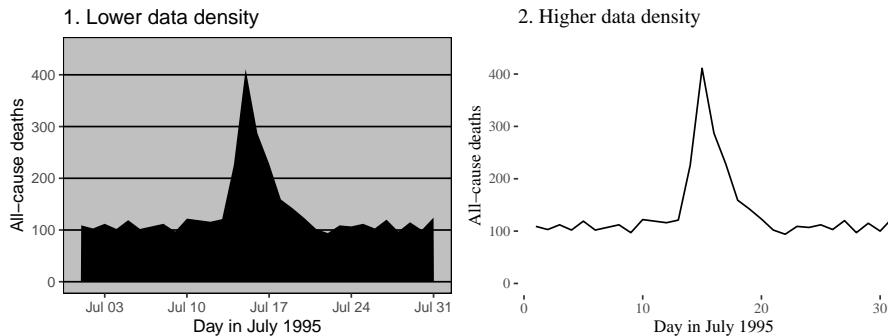


Figure 4.2: Example of plots with lower (left) and higher (right) data-to-ink ratios. Each plot shows daily mortality in Chicago, IL, in July 1995 using the `chicagoNMMAPS` data from the `dlnm` package.

By increasing the data-to-ink ratio in a plot, you can help viewers see the message of the data more quickly. A cluttered plot is harder to interpret. Further, you leave room to add some of the other elements I'll talk about, including highlighting interesting data and adding useful references. Notice how the plots on the left in Figures 4.1 and 4.2 are already cluttered and leave little room for adding extra elements, while the plots on the right of those figures have much more room for additions.

One quick way to increase data density in `ggplot2` is to change the *theme* for the plot. The theme specifies a number of the “background” elements to a plot, including elements like the plot grid, background color, and the font used for labeling. Some themes come with `ggplot2`, including:

- `theme_bw`
- `theme_minimal`
- `theme_void`

You can find more themes in packages that extend `ggplot2`. The `ggthemes` package, in particular, has some excellent additional themes.

Figures 4.3 shows some examples of the effects of using different themes. All show the same information— a plot of daily deaths in Chicago in July 1995. The top left graph shows the graph with the default theme. The other plots show the effects of adding different themes, including the black-and-white theme that comes with `ggplot2` (top right) and various themes from the `ggthemes` package. You can even use themes to add some questionable choices for different elements, like the Excel theme (bottom left).

4.3 Meaningful labels

Guideline 2: **Use clear, meaningful labels.**

Download a pdf of the lecture slides for this video.

Graphs often default to use abbreviations for axis labels and other labeling. For example, the default is for `ggplot2` plots to use column names for the x- and y-axes of a scatterplot. While this is convenient for exploratory plots, it’s often not adequate for plots for presentations and papers. You’ll want to use short and easy-to-type column names in your dataframe to make coding easier, but you should use longer and more meaningful labeling in plots and tables that others need to interpret.

Furthermore, text labels can sometimes be aligned in a way that makes them hard to read. For example, when plotting a categorical variable along the x-axis, it can be difficult to fit labels for each category that are long enough to be meaningful.

Figure 4.4 gives an example of the same information shown with labels that are harder to interpret (left) versus with clear, meaningful labels (right). Notice how the graph on the left is using abbreviations for the categorical variable (“DF” for “Defense”), abbreviations for axis labels (“Pos” for “Position” and “Pls” for “Number of players”), and has the player position labels in a vertical alignment. On the right graph, I have made the graph easier to quickly read

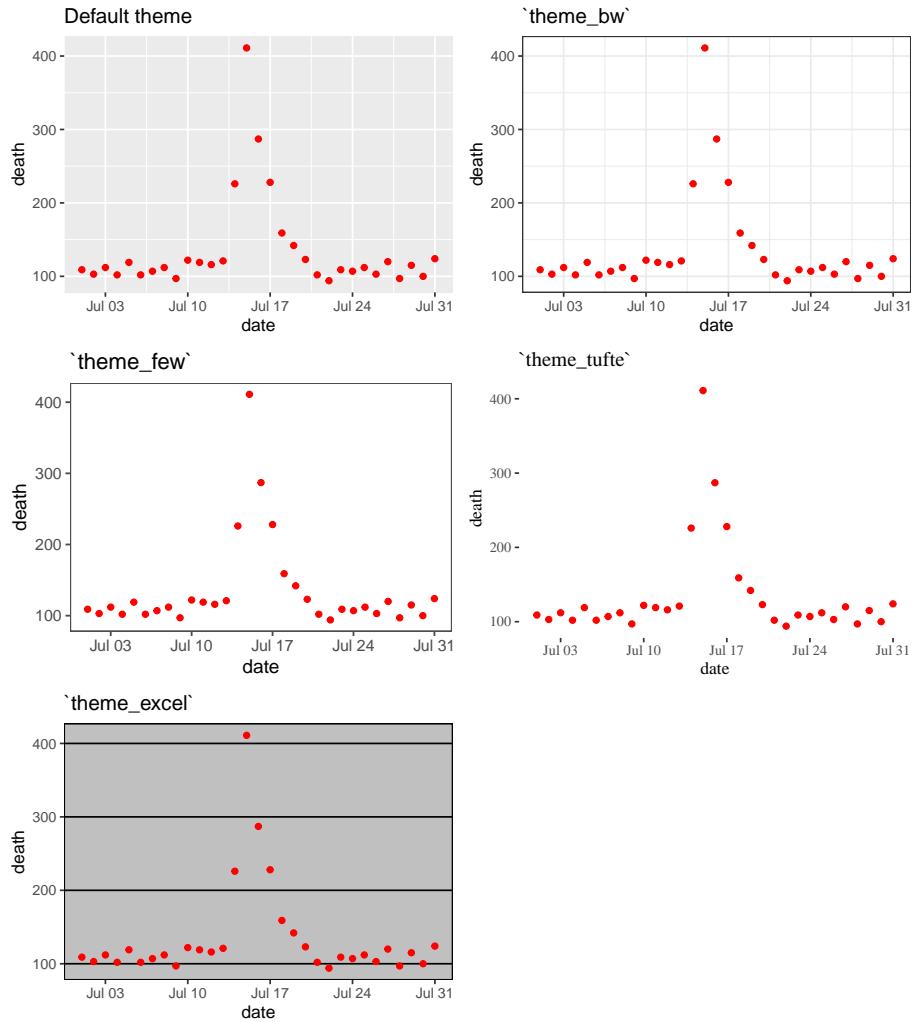


Figure 4.3: Daily mortality in Chicago, IL, in July 1995. This figure gives an example of the plot using different themes.

and interpret by spelling out all labels and switching the x- and y-axes, so that there's room to fully spell out each position while still keeping the alignment horizontal, so the reader doesn't have to turn the page (or their head) to read the values.

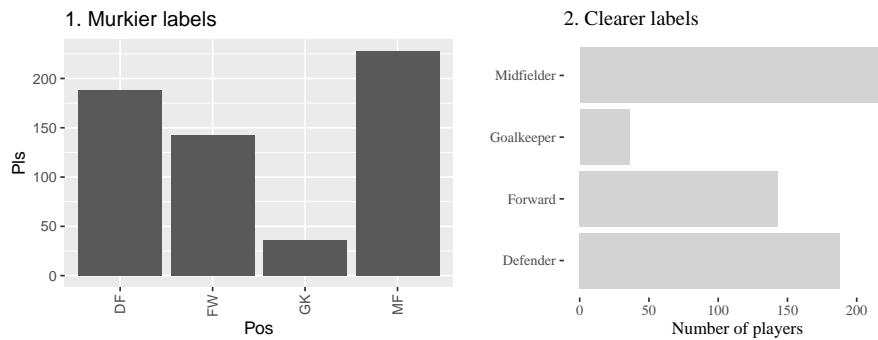


Figure 4.4: The number of players in each position in the worldcup data from the faraway package. Both graphs show the same information, but the left graph has murkier labels, while the right graph has labels that are easier to read and interpret.

There are a few strategies you can use to make labels clearer when plotting with `ggplot2`:

- Add `xlab` and `ylab` elements to the plot, rather than relying on the column names in the original data. You can also relabel x- and y-axes with `scale` elements (e.g., `scale_x_continuous`), and the `scale` functions give you more power to also make other changes to the x- and y-axes (e.g., changing break points for the axis ticks). However, if you only need to change axis labels, `xlab` and `ylab` are often quicker.
- Include units of measurement in axis titles when relevant. If units are dollars or percent, check out the `scales` package, which allows you to add labels directly to axis elements by including arguments like `labels = percent` in `scale` elements. See the helpfile for `scale_x_continuous` for some examples.
- If the x-variable requires longer labels, as is often the case with categorical data (for example, player positions Figure 4.4), consider flipping the coordinates, rather than abbreviating or rotating the labels. You can use `coord_flip` to do this.

4.4 References

Guideline 3: **Provide useful references.**

Data is easier to interpret when you add references. For example, if you show what is typical, it helps viewers interpret how unusual outliers are.

Figure 4.5 shows daily mortality during July 1995 in Chicago, IL. The graph on the right has added shading showing the range of daily death counts in July in Chicago for neighboring years (1990–1994 and 1996–2000). This added reference helps clarify for viewers how unusual the number of deaths during the July 1995 heat wave was.

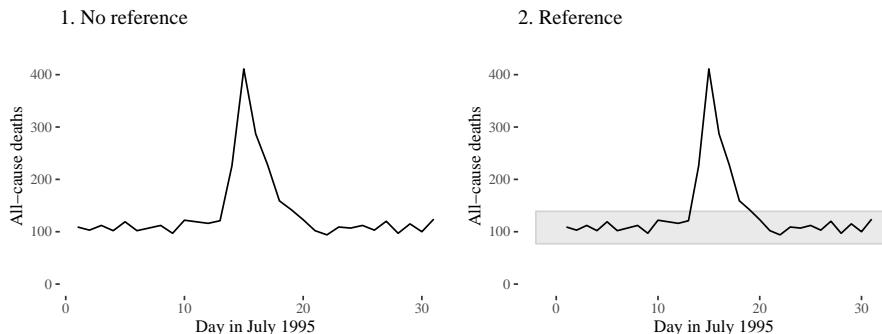


Figure 4.5: Daily mortality during July 1995 in Chicago, IL. In the graph on the right, I have added a shaded region showing the range of daily mortality counts for neighboring years, to show how unusual this event was.

Another useful way to add references is to add a linear or smooth fit to the data, to help clarify trends in the data. Figure 4.6 shows the relationship between passes and shots for Forwards in the `worldcup` dataset. The plot on the right has added a smooth function of the relationship between these two variables.

For scatterplots created with `ggplot2`, you can use the function `geom_smooth` to add a smooth or linear reference line. Here is the code that produces Figure 4.6:

```
ggplot(filter(worldcup, Position == "Forward"),
       geom_point(size = 1.5) +
       theme_few() +
       geom_smooth()
```

The most useful `geom_smooth` parameters to know are:

- `method`: The default is to add a loess curve if the data includes less than 1000 points and a generalized additive model for 1000 points or more. However, you can change to show the fitted line from a linear model using `method = "lm"` or from a generalized linear model using `method = "glm"`.

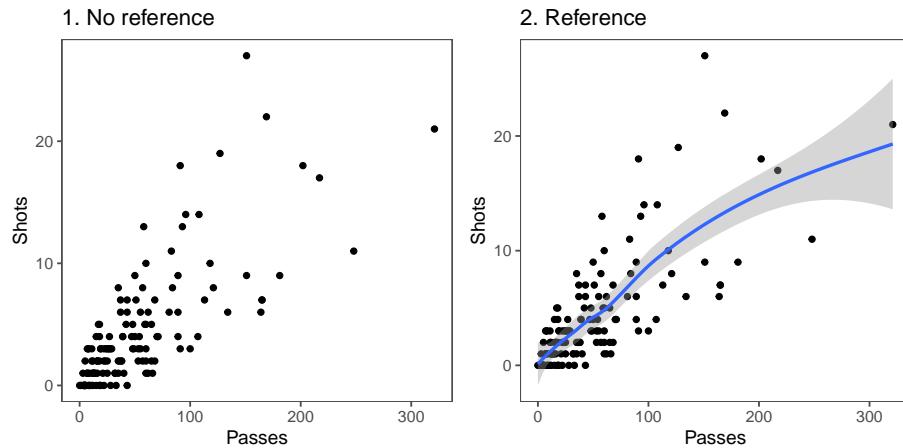


Figure 4.6: Relationship between passes and shots taken among Forwards in the `worldcup` dataset from the `faraway` package. The plot on the right has a smooth function added to help show the relationship between these two variables.

- `span`: How wiggly or smooth the smooth line should be (smaller value: more wiggly; larger value: more smooth)
- `se`: TRUE or FALSE, indicating whether to include shading for 95% confidence intervals.
- `level`: Confidence level for confidence interval (e.g., 0.90 for 90% confidence intervals)

Lines and polygons can also be useful for adding references, as in Figure 4.5. Useful geoms for such shapes include:

- `geom_hline`, `geom_vline`: Add a horizontal or vertical line
- `geom_abline`: Add a line with an intercept and slope
- `geom_polygon`: Add a filled polygon
- `geom_path`: Add an unfilled polygon

You want these references to support the main data shown in the plot, but not overwhelm it. When adding these references:

- Add reference elements first, so they will be plotted under the data, instead of on top of it.
- Use `alpha` to add transparency to these elements.
- Use colors that are unobtrusive (e.g., grays).
- For lines, consider using non-solid line types (e.g., `linetype = 3`).

4.5 Highlighting

Guideline 4: **Highlight interesting aspects.**

Download a pdf of the lecture slides for this video.

Consider adding elements to highlight noteworthy elements of the data. For example, in the graph on the right of Figure 4.7, the days of the heat wave (based on temperature measurements) have been highlighted over the mortality time series by using a thick red line.

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

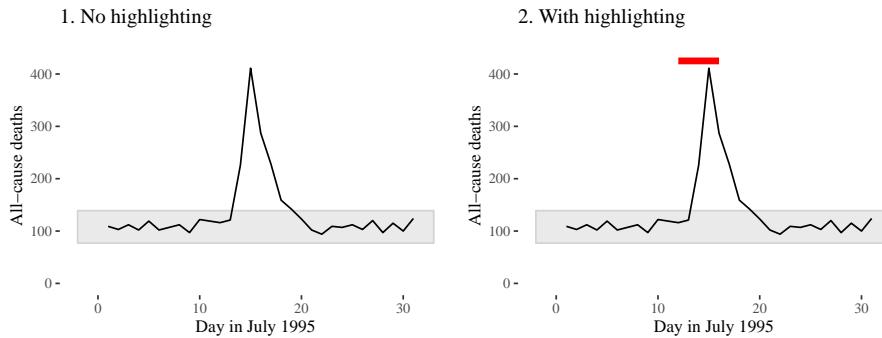
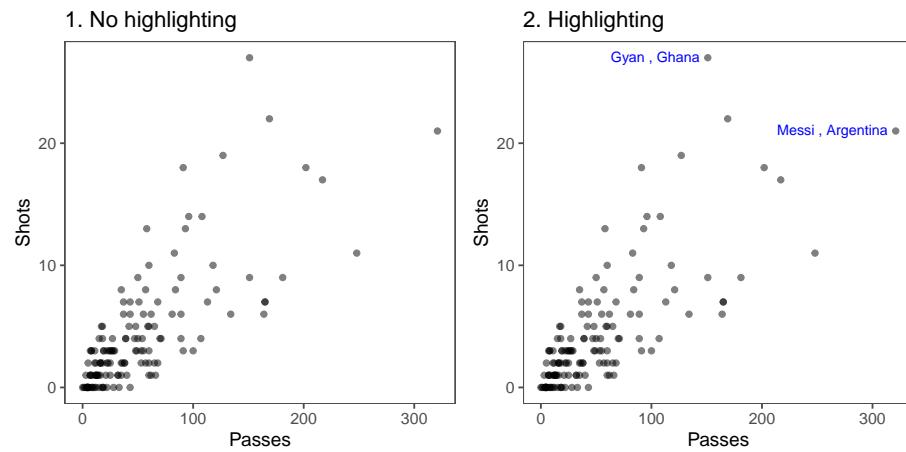


Figure 4.7: Mortality in Chicago, July 1995. In the plot on the right, a thick red line has been added to show the dates of a heat wave.

In the below graphs, the names of the players with the most shots and passes have been added to highlight these unusual points.

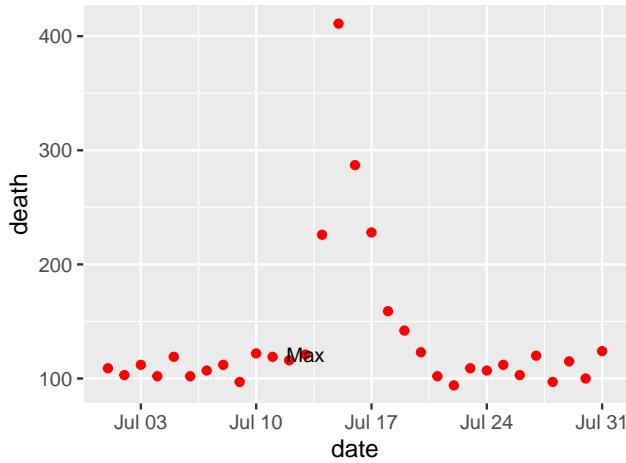


One helpful way to annotate is with text, using `geom_text()`. For this, you'll first need to create a dataframe with the hottest day in the data:

```
hottest_day <- chic_july %>%
  filter(temp == max(temp))
hottest_day[ , 1:6]
```

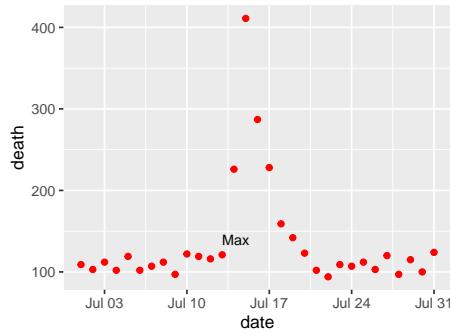
```
##           date time year month doy      dow
## 1 1995-07-13 3116 1995     7 194 Thursday
```

```
chic_plot + geom_text(data = hottest_day,
                      label = "Max",
                      size = 3)
```



With `geom_text`, you'll often want to use position adjustment (the `position` parameter) to move the text so it won't be right on top of the data points:

```
chic_plot + geom_text(data = hottest_day,
                      label = "Max",
                      size = 3, hjust = 0, vjust = -1)
```

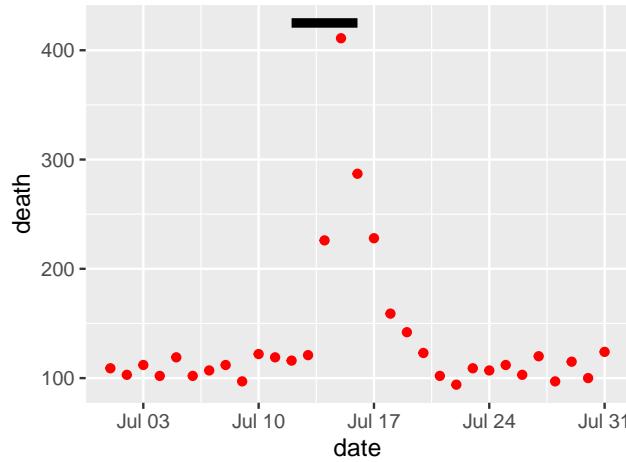


You can also use lines to highlight. For this, it is often useful to create a new dataframe with data for the reference. To add a line for the Chicago heat wave, I've added a dataframe called `hw` with the relevant date range. I'm setting the y-value to be high enough (425) to ensure the line will be placed above the mortality data.

```
hw <- data.frame(date = c(as.Date("1995-07-12"),
                           as.Date("1995-07-16")),
                  death = c(425, 425))

b <- chic_plot +
  geom_line(data = hw,
            aes(x = date, y = death),
            size = 2)
```

b

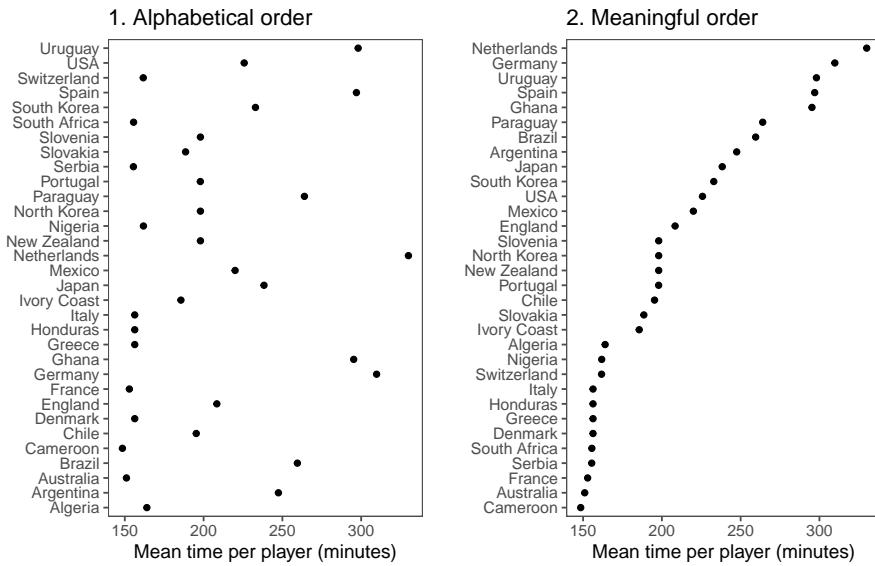


4.6 Order

Guideline 5: **Make order meaningful.**

Download a pdf of the lecture slides for this video.

You can make the ranking of data clearer from a graph by using order to show rank. Often, factor or categorical variables are ordered by something that is not interesting, like alphabetical order.



You can re-order factor variables in a graph by resetting the factor using the `factor` function and changing the order that levels are included in the `levels` parameter.

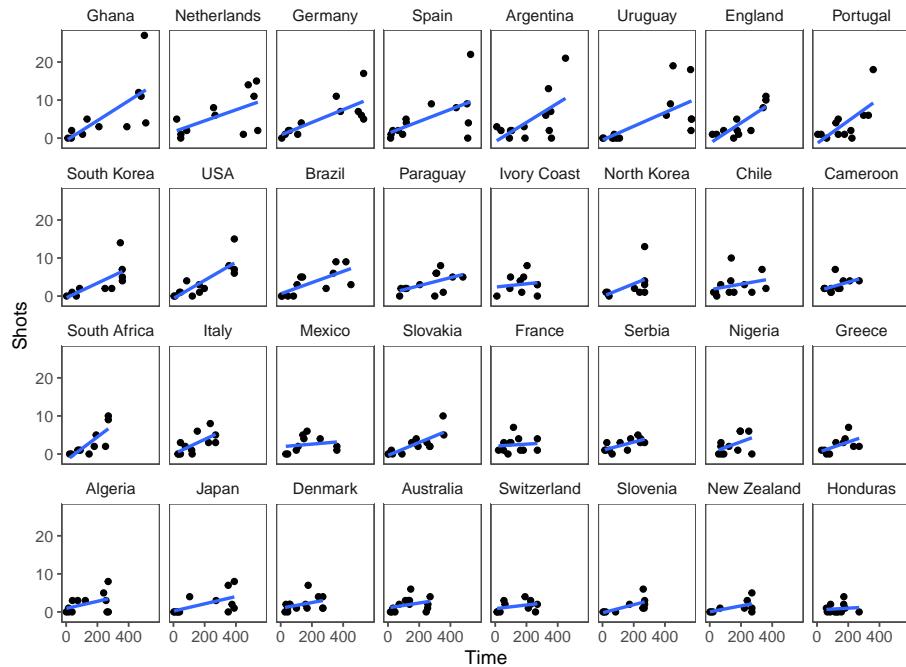
4.7 Small multiples

Guideline 6: When possible, use small multiples.

Download a pdf of the lecture slides for this video.

Small multiples are graphs that use many small plots showing the same thing for different facets of the data. For example, instead of using color in a single plot to show data for males and females, you could use two small plots, one each for males and females.

Typically, in small multiples, all plots use the same x- and y-axes. This makes it easier to compare across plots, and it also allows you to save room by limiting axis annotation.



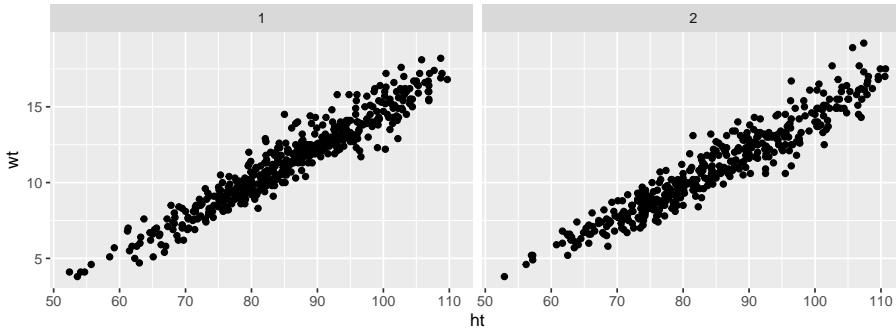
You can use the **facet** functions to create small multiples. This separates the graph into several small graphs, one for each level of a factor.

The **facet** functions are:

- `facet_grid()`
- `facet_wrap()`

For example, to create small multiples by sex for the Nepali dataset, when plotting height versus weight, you can call:

```
gplot(nepali, aes(ht, wt)) +
  geom_point() +
  facet_grid(. ~ sex)
```



The `facet_grid` function can facet by one or two variables. One will be shown by rows, and one by columns:

```
## Generic code
facet_grid([factor for rows] ~ [factor for columns])
```

The `facet_wrap()` function can only facet by one variable, but it can “wrap” the small graphs for that variable, so the don’t all have to be in one row or column:

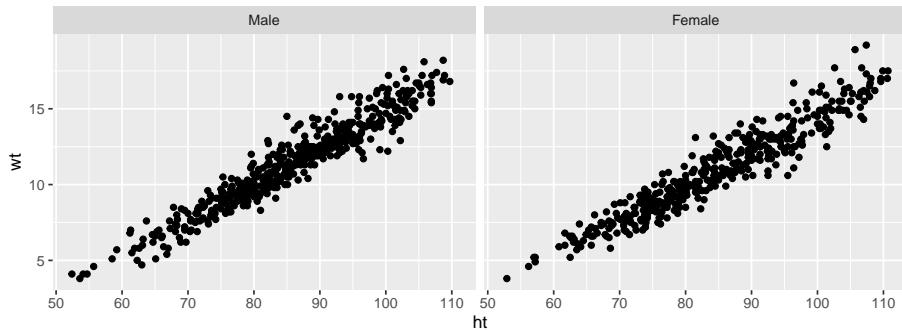
```
## Generic code
facet_wrap(~ [factor for faceting], ncol = [number of columns])
```

Often, when you do facetting, you’ll want to re-name your factors levels or re-order them. For this, you’ll need to use the `factor()` function on the original vector. For example, to rename the `sex` factor levels from “1” and “2” to “Male” and “Female”, you can run:

```
nepali <- nepali %>%
  mutate(sex = factor(sex, levels = c(1, 2),
                      labels = c("Male", "Female")))
```

Notice that the labels for the two graphs have now changed:

```
ggplot(nepali, aes(ht, wt)) +
  geom_point() +
  facet_grid(. ~ sex)
```

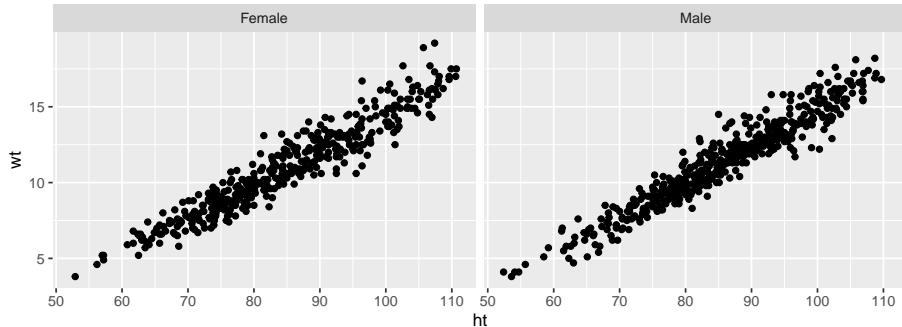


To re-order the factor, and show the plot for “Female” first, you can use `factor` to change the order of the levels:

```
nepali <- nepali %>%
  mutate(sex = factor(sex, levels = c("Female", "Male")))
```

Now notice that the order of the plots has changed:

```
ggplot(nepali, aes(ht, wt)) +
  geom_point() +
  facet_grid(. ~ sex)
```



4.8 Advanced customization

4.8.1 Scales

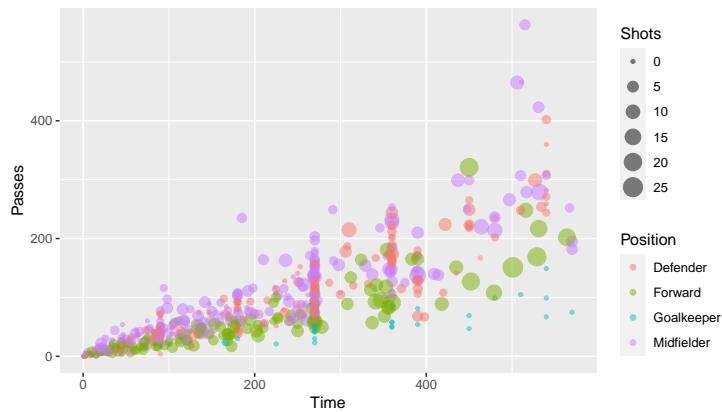
There are a number of different functions for adjusting scales. These follow the following convention:

```
## Generic code
scale_[aesthetic]_[vector type]
```

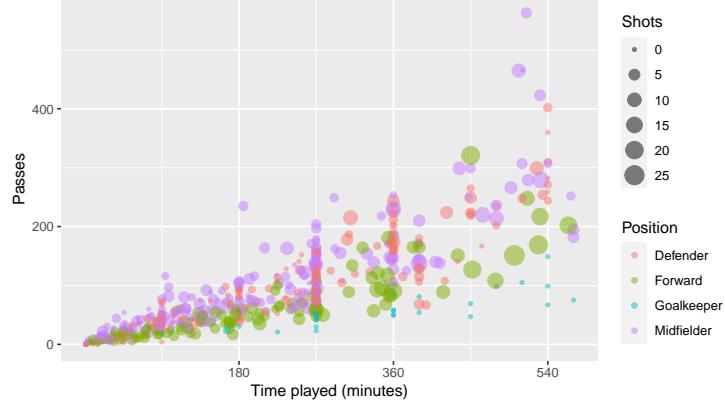
For example, to adjust the x-axis scale for a continuous variable, you'd use `scale_x_continuous`. You can use a `scale` function for an axis to change things like the axis label (which you could also change with `xlab` or `ylab`) as well as position and labeling of breaks.

For example, here is the default for plotting time versus passes for the `worldcup` dataset, with the number of shots taken shown by size and position shown by color:

```
ggplot(worldcup, aes(x = Time, y = Passes,
                      color = Position, size = Shots)) +
  geom_point(alpha = 0.5)
```



```
ggplot(worldcup, aes(x = Time, y = Passes,
                      color = Position, size = Shots)) +
  geom_point(alpha = 0.5) +
  scale_x_continuous(name = "Time played (minutes)",
                     breaks = 90 * c(2, 4, 6),
                     minor_breaks = 90 * c(1, 3, 5))
```

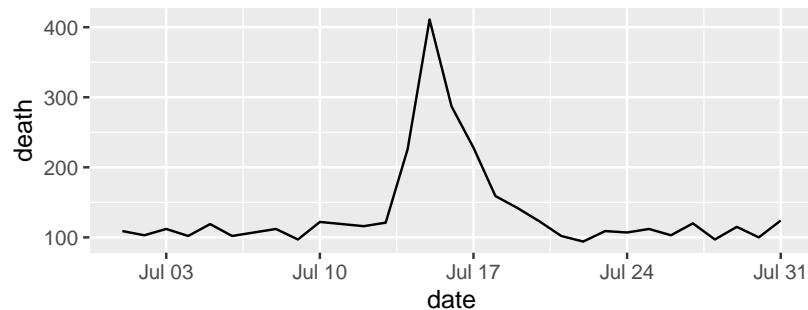


Parameters you might find useful in `scale` functions include:

Parameter	Description
name	Label or legend name
breaks	Vector of break points
minor_breaks	Vector of minor break points
labels	Labels to use for each break
limits	Limits to the range of the axis

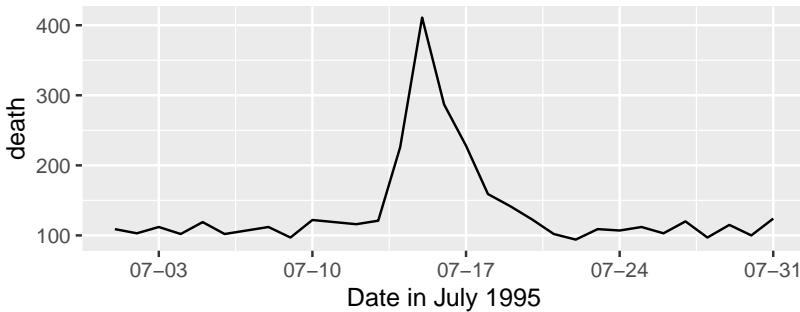
For dates, you can use `scale` functions like `scale_x_date` and `scale_x_datetime`. For example, here's a plot of deaths in Chicago in July 1995 using default values for the x-axis:

```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line()
```



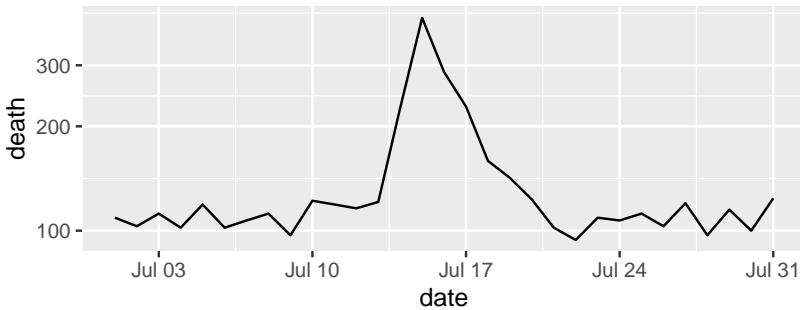
And here's an example of changing the formating and name of the x-axis:

```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line() +
  scale_x_date(name = "Date in July 1995",
               date_labels = "%m-%d")
```



You can also use the `scale` functions to transform an axis. For example, to show the Chicago plot with “deaths” on a log scale, you can run:

```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line() +
  scale_y_log10()
```



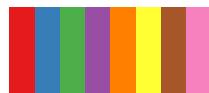
For colors and fills, the conventions for the names of the `scale` functions can vary. For example, to adjust the color scale when you’re mapping a discrete variable (i.e., categorical, like gender or animal breed) to color, you’d use `scale_color_hue`. To adjust the color scale for a continuous variable, like age, you’ll use `scale_color_gradient`.

For any color scales, consider starting with `brewer` first (e.g., `scale_color_brewer`, `scale_color_distiller`). Scale functions from `brewer` allow you to

set colors using different palettes. You can explore these palettes at <http://colorbrewer2.org/>.

The Brewer palettes fall into three categories: sequential, divergent, and qualitative. You should use sequential or divergent for continuous data and qualitative for categorical data. Use `display.brewer.pal` to show the palette for a given number of colors.

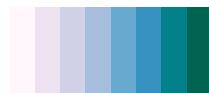
```
library(RColorBrewer)
display.brewer.pal(name = "Set1", n = 8)
display.brewer.pal(name = "PRGn", n = 8)
display.brewer.pal(name = "PuBuGn", n = 8)
```



Set1 (qualitative)



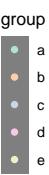
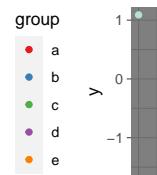
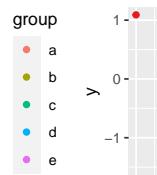
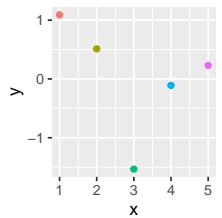
PRGn (divergent)



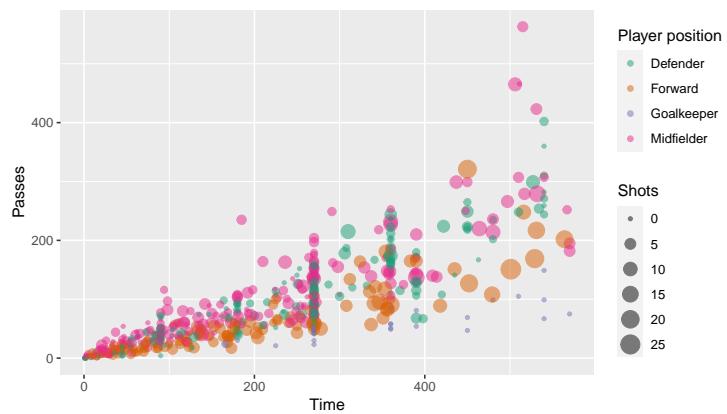
PuBuGn (sequential)

Use the `palette` argument within a `scales` function to customize the palette:

```
a <- ggplot(data.frame(x = 1:5, y = rnorm(5),
                       group = letters[1:5]),
             aes(x = x, y = y, color = group)) +
  geom_point()
b <- a + scale_color_brewer(palette = "Set1")
c <- a + scale_color_brewer(palette = "Pastel2") +
  theme_dark()
grid.arrange(a, b, c, ncol = 3)
```

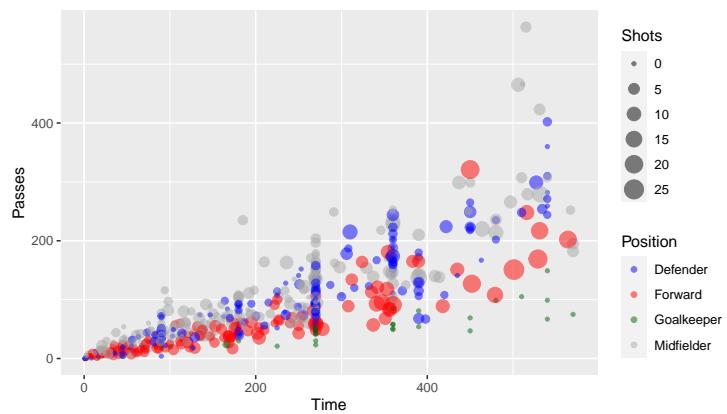


```
ggplot(worldcup, aes(x = Time, y = Passes,
                      color = Position, size = Shots)) +
  geom_point(alpha = 0.5) +
  scale_color_brewer(palette = "Dark2",
                     name = "Player position")
```



You can also set colors manually:

```
ggplot(worldcup, aes(x = Time, y = Passes,
                      color = Position, size = Shots)) +
  geom_point(alpha = 0.5) +
  scale_color_manual(values = c("blue", "red",
                               "darkgreen", "darkgray"))
```



4.9 To find out more

Some excellent further references for plotting are:

- R Graphics Cookbook (book and website)
- Google images

For more technical details about plotting in R:

- ggplot2: Elegant Graphics for Data Analysis, Hadley Wickham
- R Graphics, Paul Murrell

4.10 In-course exercise–Chapter 4

4.10.1 Designing a plot

For today’s exercise, you’ll be building a plot using the `worldcup` data from the `faraway` package. First, load in that data. The name of each player is in the rownames of this data. Use the `tibble::rownames_to_column()` function to move those rownames into a new column named `Player`. Also install and load the `ggplot2` and `ggthemes` packages.

Next, say you want to look at the relationship between the number of minutes that a player played in the 2010 World Cup (`Time`) and the number of shots the player took on goal (`Shots`). On a sheet of paper, and talking with your partner, decide how the two of you would design a plot to explore and present this relationship. How would you incorporate some of the principles of creating good graphs?

4.10.1.1 Example R code

For this section, the only code needed is code to load the required packages, load the data, and move the rownames to a column named `Player`.

```
library(faraway)
data(worldcup)
head(worldcup, 2)
```

	Team	Position	Time	Shots	Passes	Tackles	Saves
## Abdoun	Algeria	Midfielder	16	0	6	0	0
## Abe	Japan	Midfielder	351	0	101	14	0

This dataset has the players' names as rownames, rather than in a column. Once we start using `dplyr` functions, we'll lose these rownames. Therefore, start by converting the rownames to a column called `Player`:

```
library(dplyr)
worldcup <- worldcup %>%
  tibble::rownames_to_column(var = "Player")
head(worldcup, 2)
```

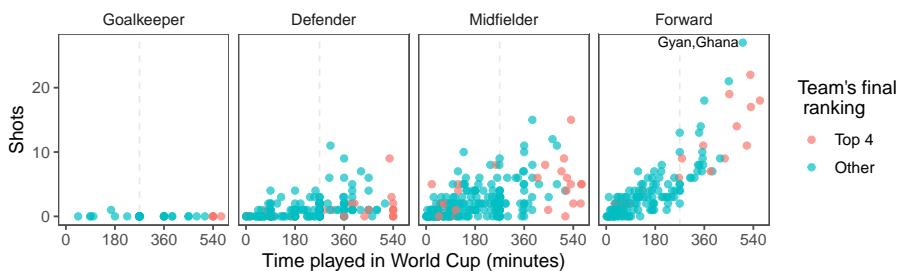
	Player	Team	Position	Time	Shots	Passes	Tackles	Saves
## 1	Abdoun	Algeria	Midfielder	16	0	6	0	0
## 2	Abe	Japan	Midfielder	351	0	101	14	0

Install and load the `ggplot2` package:

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

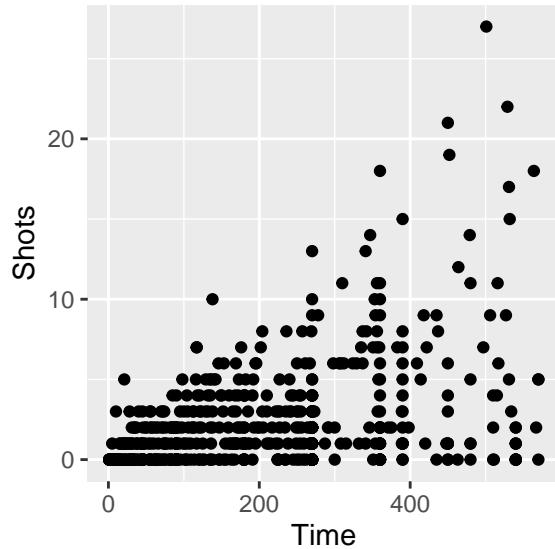
4.10.2 Implementing plot guidelines #1

In this section, we'll work on creating a plot like this:

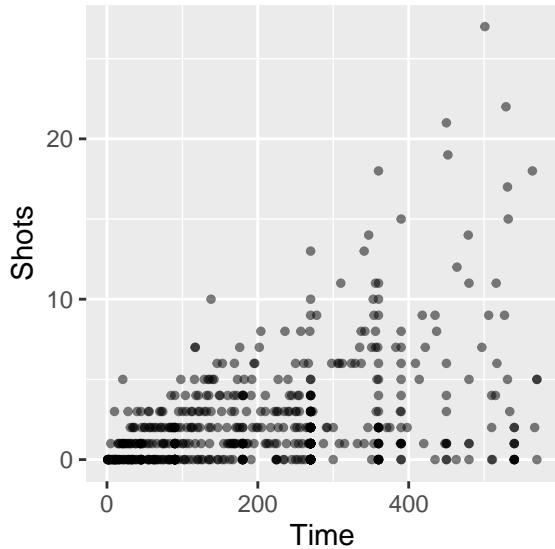


Do the following tasks:

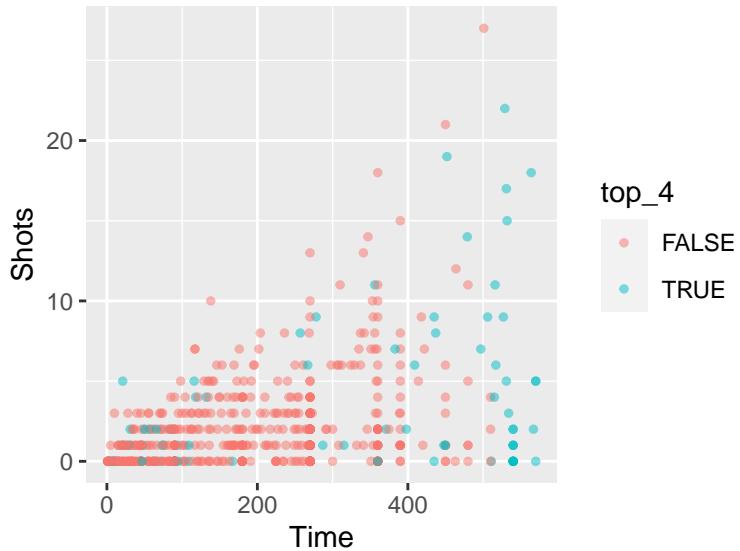
- Create a simple scatterplot of Time versus Shots for the World Cup data. It should look like this:



- Next, before any more coding, talk with your group members about how the ultimate graph we're working on (the one printed at the beginning of this exercise section) is different from the simple one you created with `ggplot` for the last bullet point. Also discuss what you can figure out from this new graph that was less clear from a simpler scatterplot of Time versus Shots for this data.
- Often, in graphs with a lot of points, it's hard to see some of the points, because they overlap other points. Three strategies to address this are: (a) make the points smaller; and (b) make the points somewhat transparent. Try doing these first two with the scatterplot you're creating. At this point, the plot should look something like this:

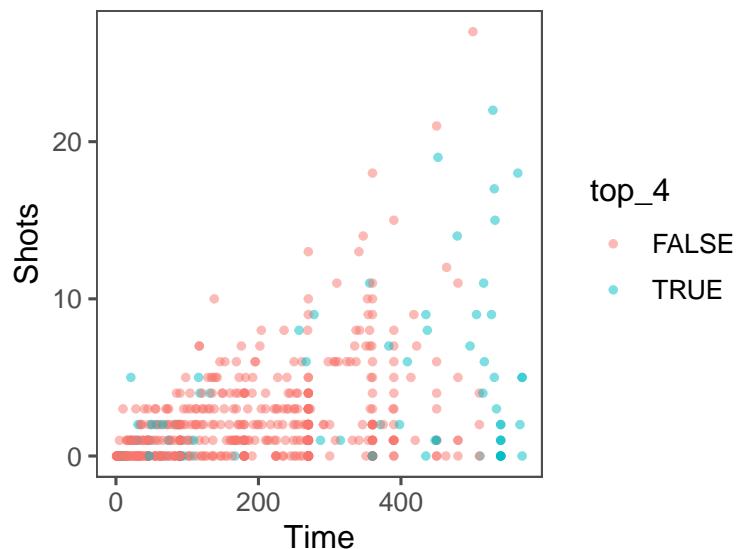


- Create a new column in the `worldcup` data called `top_four` that specifies whether or not the `Team` for that observation was one of the top four teams in the tournament (Netherlands, Uruguay, Spain, and Germany). Make the colors of the points correspond to whether the team was a top-four team. At this point, the plot should look something like this:

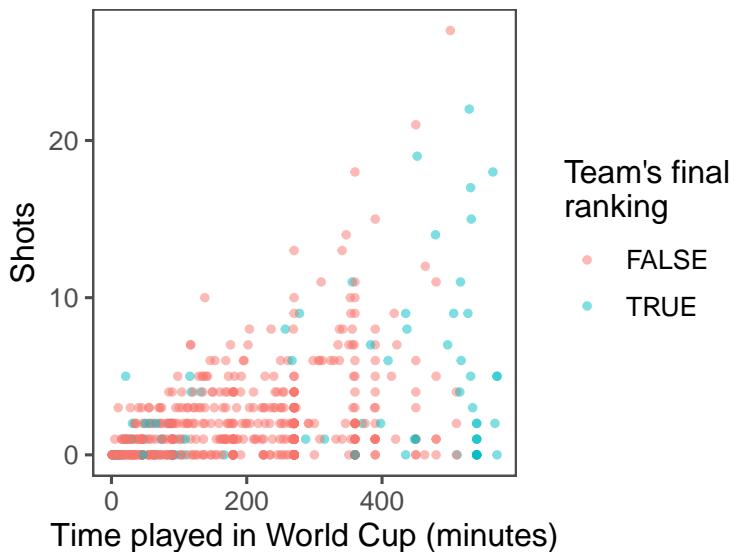


- Increase data density: Try changing the theme, to come up with a graph

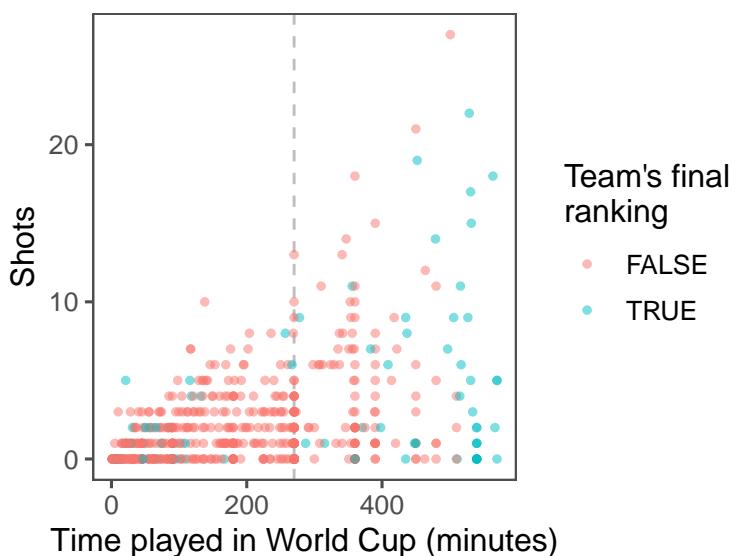
with a bit less non-data ink. From the `ggthemes` package, try some of the following themes: `theme_few()`, `theme_tufte()`, `theme_stata()`, `theme_fivethirtyeight()`, `theme_economist_white()`, and `theme_wsj()`. Pick a theme that helps increase the graph's data density. At this point, the plot should look something like this:



- Use meaningful labels: Use the `labs()` function to make a clearer title for the x-axis. (You may have already written this code in the last section of this exercise.) In addition to setting the x-axis title with the `labs` function, you can also set the title for the color scale (use `color =` within the `labs` function). You may want to make a line break in the color title—you can use the linebreak character (`\n`) inside the character string with the title to do that. At this point, the plot should look something like this:



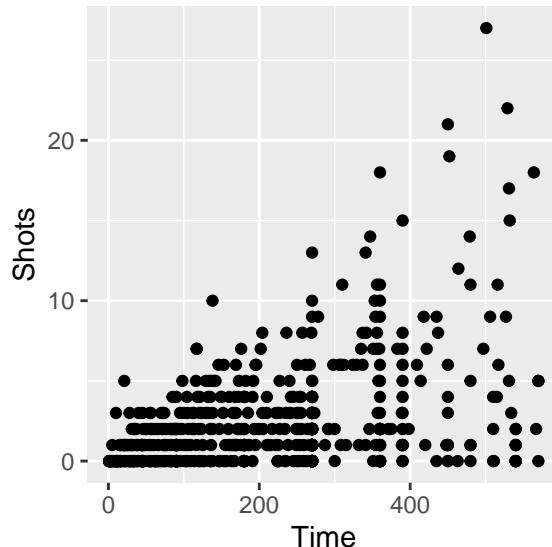
- Provide useful references: The standard time for a soccer game is 90 minutes. In the World Cup, all teams play at least three games, and then the top teams continue and play more games. Add a reference line at 270 minutes (i.e., the amount of standard time played for the three games that all teams play). At this point, the plot should look something like this:



4.10.2.1 Example R code

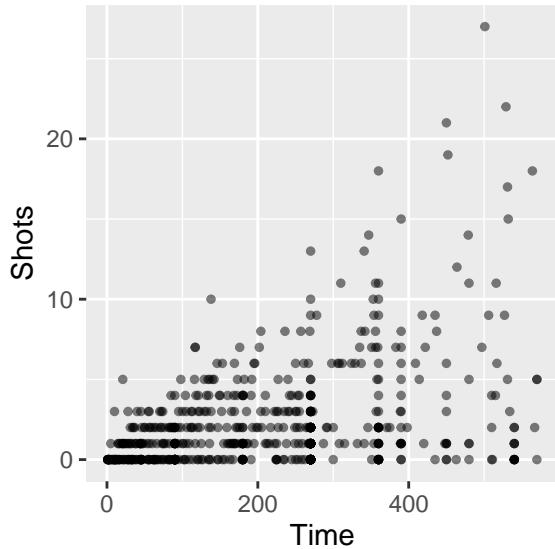
As a reminder, here's the code to do a simple scatterplot of Shots by Time for the `worldcup` data:

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots))
```



Next, try to make it clearer to see the points by making them smaller and somewhat transparent. This can be done with the `size` and `alpha` aesthetics for `geom_point`. For the `size` aesthetic, a value smaller than about 2 = smaller than default, larger than about 2 = larger than default. For the `alpha` aesthetic, closer to 0 = more transparent, closer to 1 = more opaque. As a reminder, in this case you are changing all of the points in the same way, so you will be setting those aesthetics to constant values. That means that you should specify the values **outside** of an `aes` call. This code could make these changes:

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots),
             size = 1, alpha = 0.5)
```



To create a new column called `top_four`, first create vector that lists those top four teams, then create a logical vector in the data frame for whether the team for that observation is in one of the top four teams:

```
worldcup <- worldcup %>%
  mutate(top_4 = Team %in% c("Spain", "Germany",
                            "Uruguay", "Netherlands"))
head(worldcup)
```

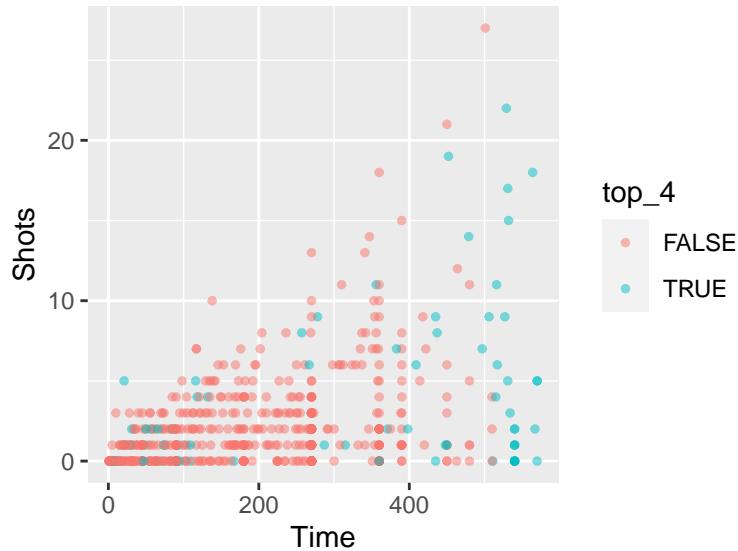
##	Team	Position	Time	Shots	Passes	Tackles	Saves	Player	top_4
## Abdoun	Algeria	Midfielder	16	0	6	0	0	Abdoun	FALSE
## Abe	Japan	Midfielder	351	0	101	14	0	Abe	FALSE
## Abidal	France	Defender	180	0	91	6	0	Abidal	FALSE
## Abou Diaby	France	Midfielder	270	1	111	5	0	Abou Diaby	FALSE
## Aboubakar	Cameroon	Forward	46	2	16	0	0	Aboubakar	FALSE
## Abreu	Uruguay	Forward	72	0	15	0	0	Abreu	TRUE

```
summary(worldcup$top_4)
```

##	Mode	FALSE	TRUE
## logical	517	78	

To color points by this variable, use `color` = in the `aes()` part of the `ggplot()` call:

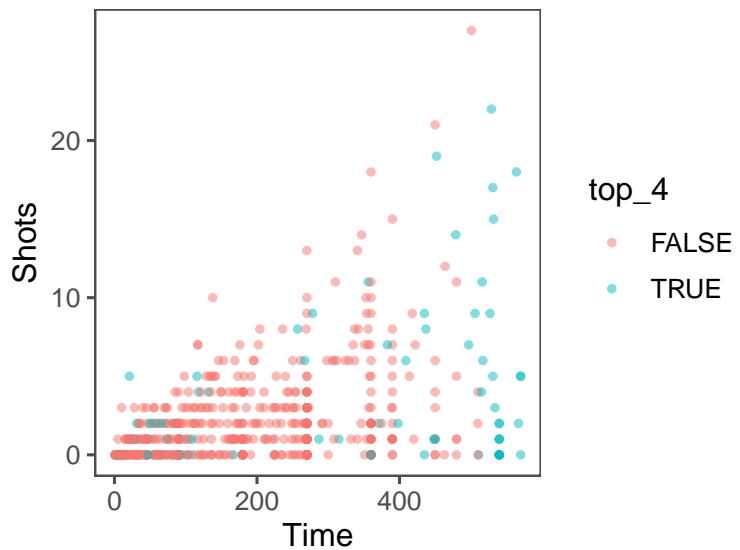
```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5)
```



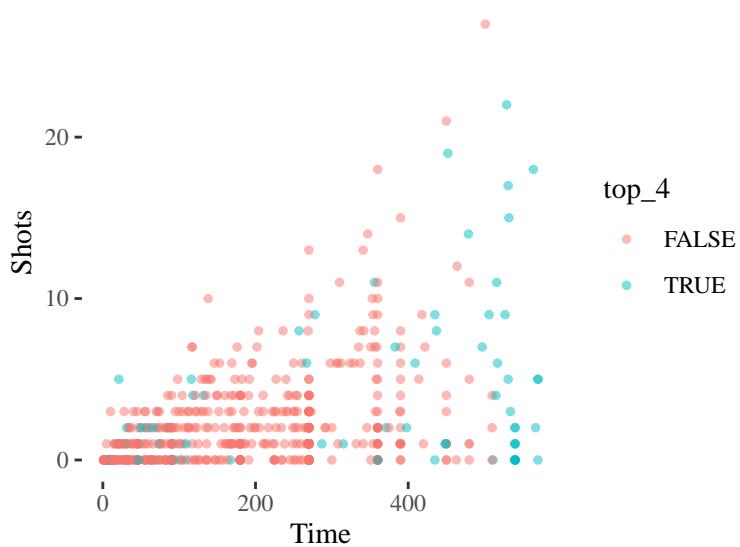
To increase the data density, try out different themes for the plot. First, I'll save everything we've done so far as the object `shot_plot`, then I'll try adding different themes:

```
shot_plot <- ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5)

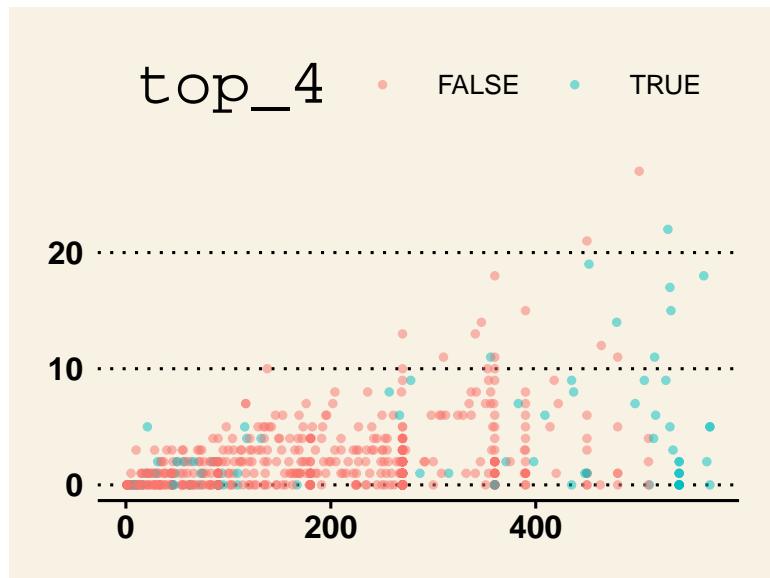
shot_plot + theme_few()
```



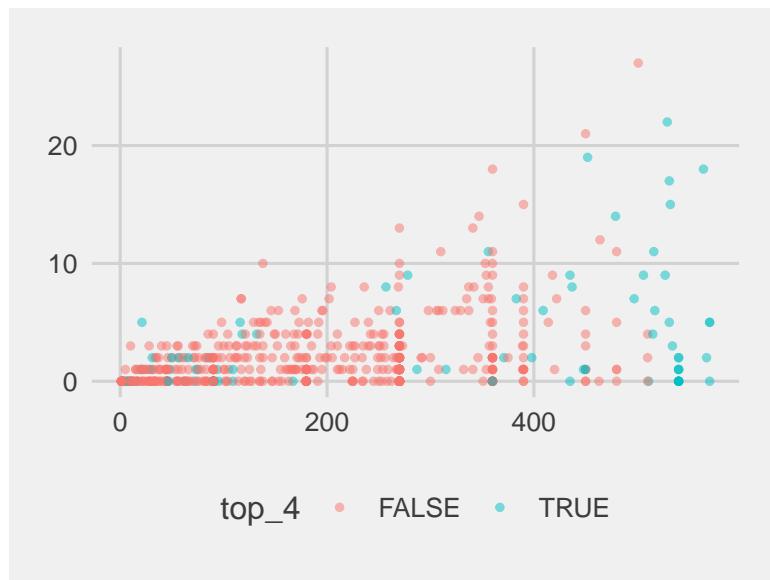
```
shot_plot + theme_tufte()
```



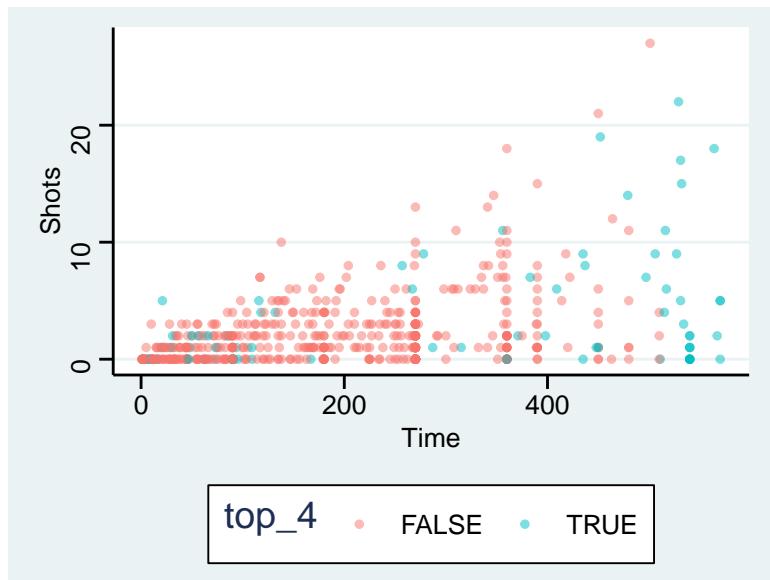
```
shot_plot + theme_wsj()
```



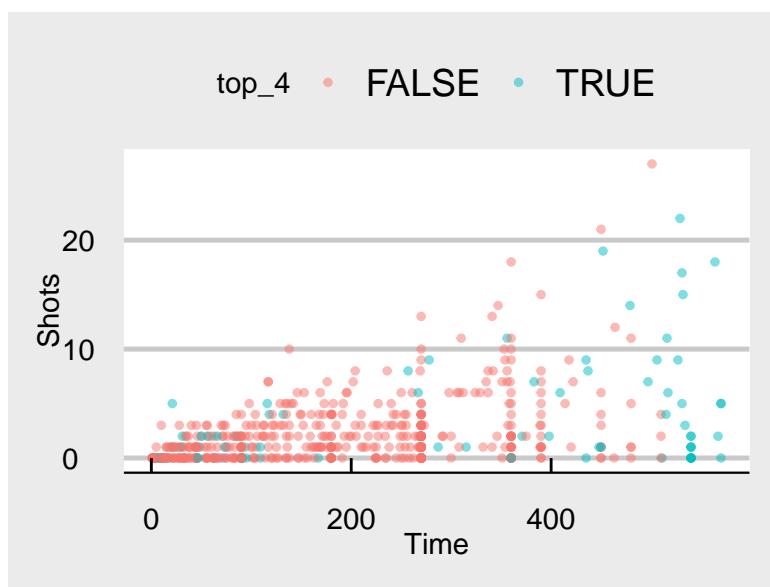
```
shot_plot + theme_fivethirtyeight()
```



```
shot_plot + theme_stata()
```

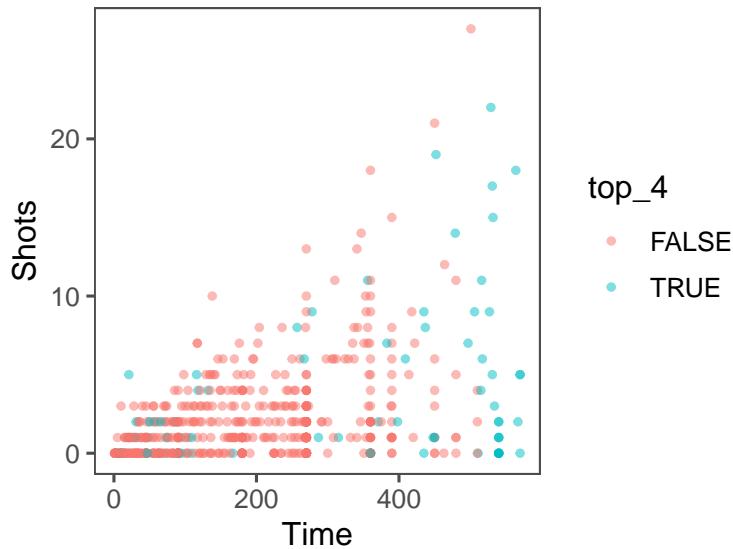


```
shot_plot + theme_economist_white()
```



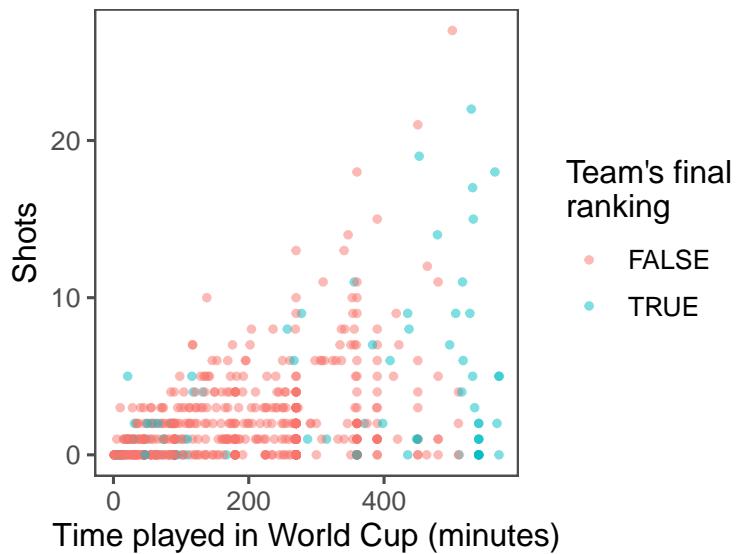
The data density is increased with the `theme_few()` theme, so I'll use that:

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few()
```



To change the titles for some of the scales (the x-axis and color scale), you can use the `labs()` function. Note that you can use `\n` to add a line break inside one of these titles (I've done that for the title for the color scale):

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few() +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking")
```

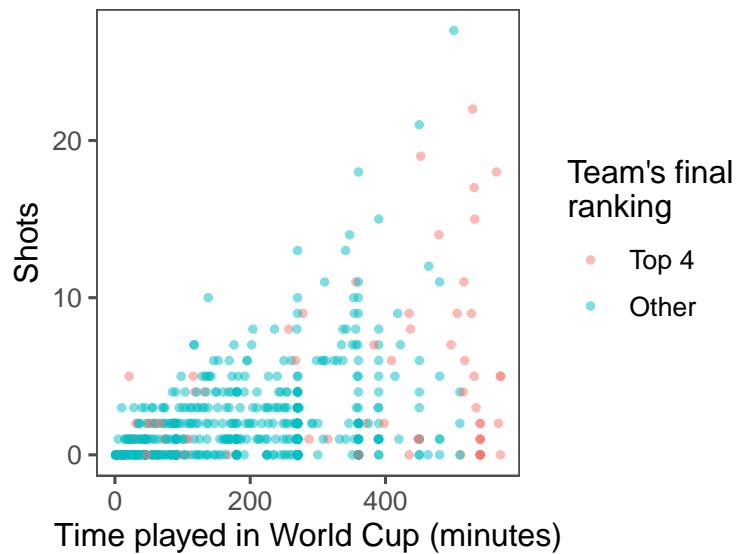


As an extra note, if you want to create nicer labels for the legend for color, convert the `top_four` column into the factor class, with the labels you want to use in the figure legend:

```
worldcup <- worldcup %>%
  mutate(top_4 = factor(top_4, levels = c(TRUE, FALSE),
                        labels = c("Top 4", "Other")))
summary(worldcup$top_4)
```

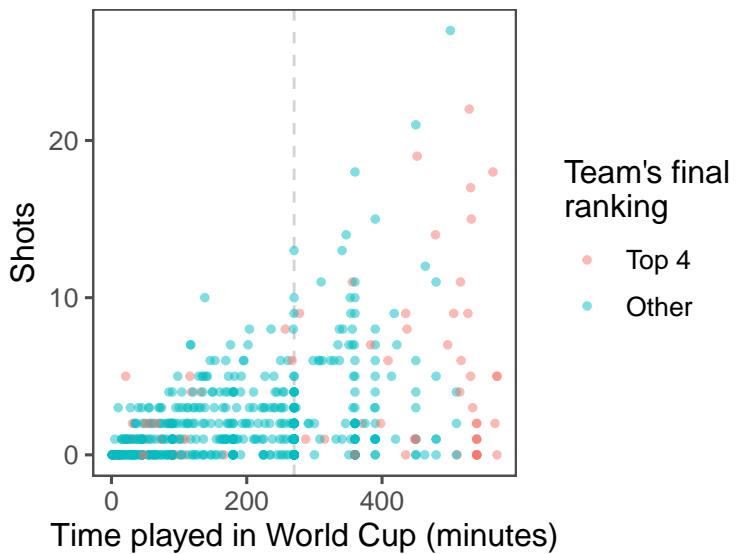
```
## Top 4 Other
##      78    517
```

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few() +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\\nranking")
```



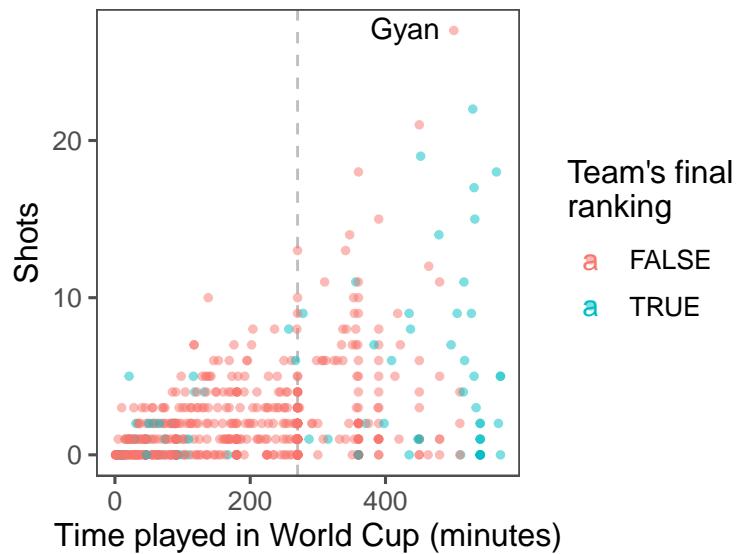
To add a reference line at 270 minutes of time, use the `geom_vline()` function. You'll want to make it a light color (like light gray) and dashed or dotted (`linetype` of 2 or 3), so it won't be too prominent on the graph:

```
ggplot(data = worldcup) +
  geom_vline(xintercept = 270, color = "lightgray", linetype = 2) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few() +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\\nranking")
```

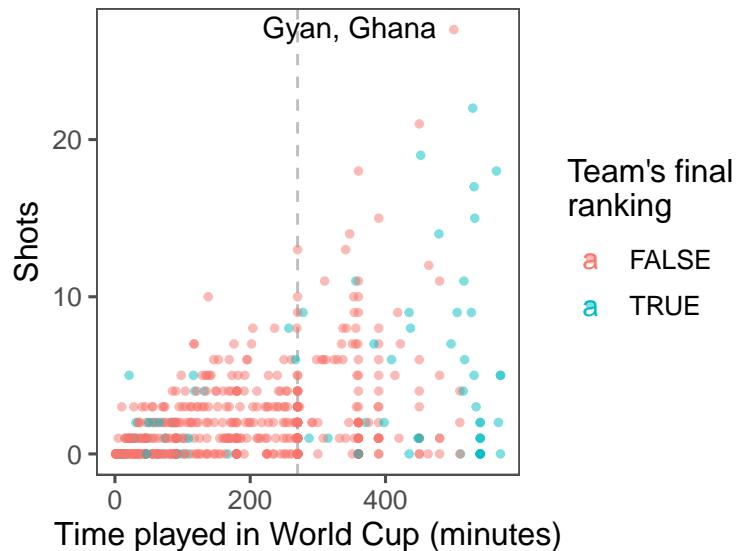


4.10.3 Implementing plot guidelines #2

- Highlighting interesting data: Who had the most shots in the 2010 World Cup? Was he on a top-four team? Use `geom_text()` to label his point on the graph with his name (try out some different values of `hjust` and `vjust` in this function call to get the label in a place you like). At this point, the plot should look something like this:

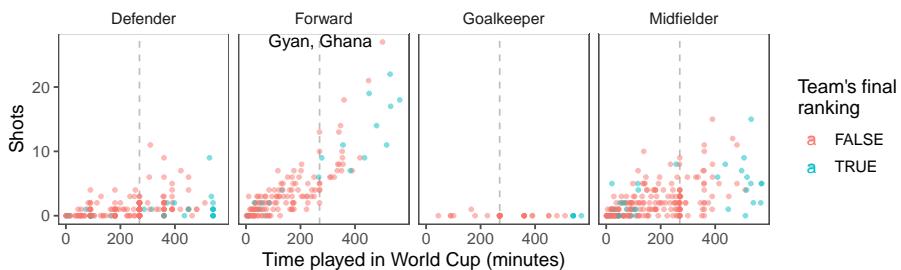


- For labeling the player with the top number of shots, instead of only using the player’s name, use the following format: “[Player’s name], [Player’s team]”. (Hint: You may want to use `mutate` to add a new column, where you used `paste0` to paste together the player’s name, “, ”, and the team name.) At this point, the plot should look something like this:

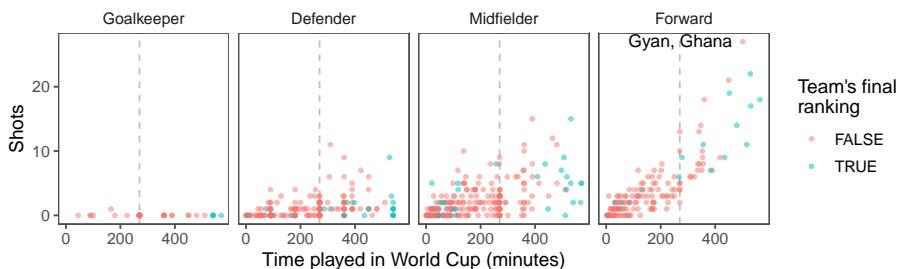


- Create small multiples. The relationship between time played and shots

taken is probably different by the players' positions. Use faceting to create different graphs for each position. At this point, the plot should look something like this:



- Make order meaningful: What order are the faceted graphs currently in? Offensive players have more chances to take shots than defensive players, so that might be a useful ordering for the facets. Re-order the `Position` factor column to go from nearest your own goal to nearest the opponents goal, and then re-plot the graph from the previous step.



4.10.3.1 Example R code

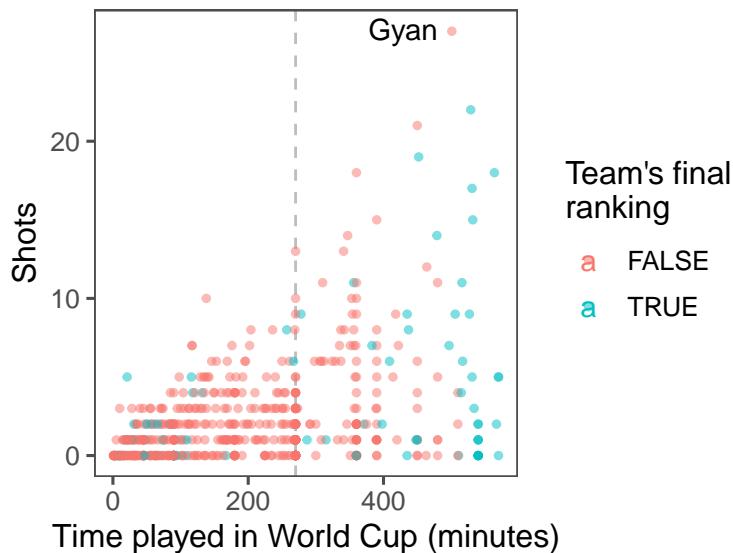
To add a text label with just the player with the most shots, you'll want to create a new dataframe with just the top player. You can use the `top_n` function to do that (the `wt` option is specifying that we want the top player in terms of values in the `Shots` column):

```
top_player <- worldcup %>%
  top_n(n = 1, wt = Shots)
```

Now you can use `geom_text()` to label this player's point on the graph with his name. You may need to mess around with some of the options in `geom_text()`,

like `size`, `hjust`, and `vjust` (`hjust` and `vjust` say where, in relation to the point location, to put the label), to get something you're happy with.

```
worldcup %>%
  mutate(top_4 = Team %in% c("Netherlands", "Uruguay", "Spain", "Germany")) %>%
  ggplot(aes(x = Time, y = Shots, color = top_4)) +
  geom_vline(xintercept = 90 * 3, color = "gray", linetype = 2) +
  geom_point(alpha = 0.5, size = 1) +
  geom_text(data = top_player, aes(label = Player, color = NULL),
            hjust = 1.2, vjust = 0.4) +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\\nranking") +
  theme_few()
```

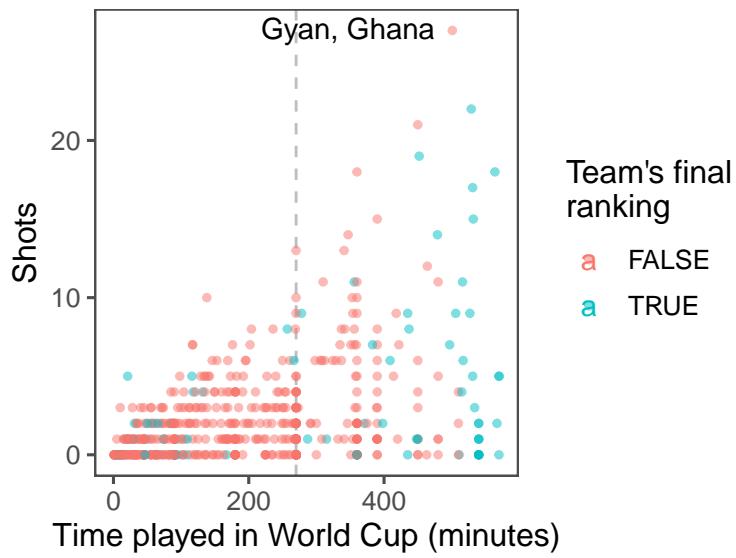


If you want to put both the player's name and his team, you can add a `mutate()` function when you create the new dataframe with just the top player, and then use this for the label:

```
top_player <- worldcup %>%
  top_n(n = 1, wt = Shots) %>%
  mutate(label = paste0(Player, ", ", Team))

worldcup %>%
  mutate(top_4 = Team %in% c("Netherlands", "Uruguay", "Spain", "Germany")) %>%
```

```
ggplot(aes(x = Time, y = Shots, color = top_4)) +
  geom_vline(xintercept = 90 * 3, color = "gray", linetype = 2) +
  geom_point(alpha = 0.5, size = 1) +
  geom_text(data = top_player,
            aes(label = label, color = NULL),
            hjust = 1.1, vjust = 0.4) +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking") +
  theme_few()
```



To create small multiples, use the `facet_wrap()` command (you'll probably want to use `ncol` to specify to use four columns):

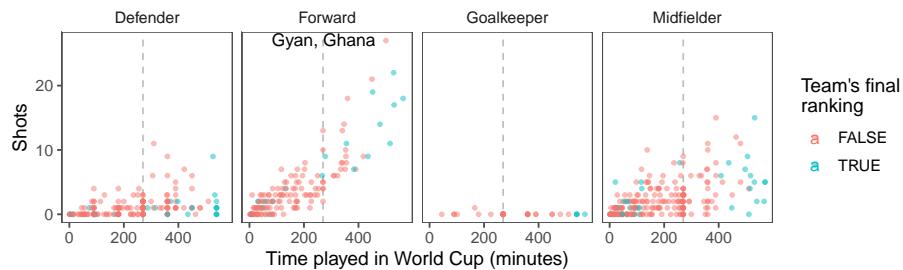
```
top_player <- worldcup %>%
  top_n(n = 1, wt = Shots) %>%
  mutate(label = paste0(Player, ", ", Team))

worldcup %>%
  mutate(top_4 = Team %in% c("Netherlands", "Uruguay", "Spain", "Germany")) %>%
  ggplot(aes(x = Time, y = Shots, color = top_4)) +
  geom_vline(xintercept = 90 * 3, color = "gray", linetype = 2) +
  geom_point(alpha = 0.5, size = 1) +
  geom_text(data = top_player,
            aes(label = label, color = NULL),
```

```

      hjust = 1.1, vjust = 0.4) +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking") +
  theme_few() +
  facet_wrap(~ Position, ncol = 4)

```

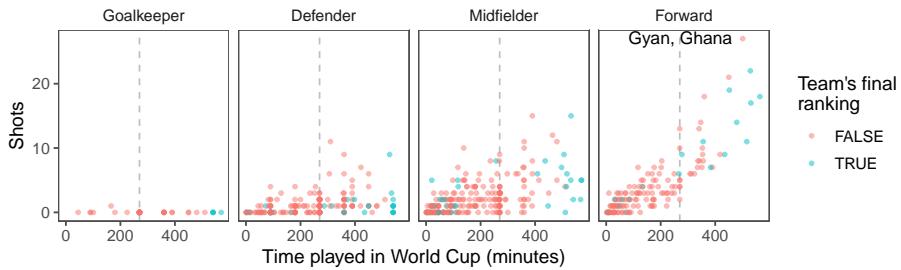


To re-order the `Position` column of the dataframe, add a `mutate` statement before you pipe into the plotting code. Use the `levels` option of the `factor()` function— whatever order you put the factors in for this argument will be the new order in which R saves the levels of this factor.

```

worldcup %>%
  mutate(top_4 = Team %in% c("Netherlands", "Uruguay", "Spain", "Germany"),
         Position = factor(Position, levels = c("Goalkeeper", "Defender",
                                                 "Midfielder", "Forward"))) %>%
  ggplot() +
  geom_vline(xintercept = 90 * 3, color = "gray", linetype = 2) +
  geom_point(aes(x = Time, y = Shots, color = top_4),
             alpha = 0.5, size = 1) +
  geom_text(data = top_player,
            aes(x = Time, y = Shots, label = label),
            hjust = 1.1, vjust = 0.4) +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking") +
  theme_few() +
  facet_wrap(~ Position, ncol = 4)

```



Note from this code example that you can use the `levels` function to find out the levels and their order for a factor-class vector.

```
worldcup <- worldcup %>%
  mutate(Position = factor(Position,
                           levels = c("Goalkeeper", "Defender",
                                      "Midfielder", "Forward")))
levels(worldcup$Position)
```

```
## [1] "Goalkeeper" "Defender"    "Midfielder"   "Forward"
```

4.10.4 Data visualization cheatsheet

RStudio comes with some excellent cheatsheets, which provide quick references to functions and code you might find useful for different tasks. For this part of the group exercise, you'll explore their cheatsheet for data visualization, both to learn some new `ggplot2` code and to become familiar with how to use this cheatsheet as you do your own analysis.

- Open the data visualization cheatsheet. You can do this from RStudio by going to “Help” -> “Cheatsheets” -> “Data Visualization with ggplot2”.
- Notice that different sections give examples with some datasets that come with either base R or ggplot2. For example, under the “Graphical Primitives” section, there is code defining the object `a` as a `ggplot` object using the “seals” dataset: `a <- ggplot(seals, aes(x = long, y = lat))`.
- Go through the cheatsheet and list all of the example datasets that are used in this cheatsheet. Open their helpfiles to learn more about the data.
- Create the example datasets `a` through `l` and `s` through `t` using the code given on the cheatsheet.

- Pick at least one example to try out from each of the following sections: “Graphical Primitives”, “One Variable”, at least three subsections of “Two Variables”, “Three Variables”, “Scales”, “Faceting”, and “Position Adjustments”. As you try these, try to figure out any aesthetics that you aren’t familiar with (e.g., `ymin`, `ymax`). Also, use helpfiles for the geoms to look up parameters you aren’t familiar with (e.g., `stat` for `geom_area`). If you can’t figure out how to interpret a plot, check the helpfile for the associated geom. **Note:** For the `n` geom used in “scales”, it should be defined as `n <- d + geom_bar(aes(fill = f1))`.

4.10.4.1 Example R code

The code for opening the helpfiles for the example datasets is:

```
?seals
?economics
?mpg
?diamonds
?USArrests
```

Note that, for `USArrests`, only some of the columns are pulled out (e.g., `murder = USArrests$murder`) to use in the `data` example dataframe. Further, the “Visualizing error” examples use a dataframe created specifically for these examples, called `df`.



Some of the base R and ggplot2 example datasets have become fairly well-known. Some that you’ll see very often in examples are the `iris`, `mpg`, and `diamonds` datasets.

All of the code to create the datasets `a` through `l` and `s` through `t` is given somewhere on the cheatsheet. Here it is in full:

```
a <- ggplot(seals, aes(x = long, y = lat))
b <- ggplot(economics, aes(date, unemploy))
c <- ggplot(mpg, aes(hwy))
d <- ggplot(mpg, aes(f1))
e <- ggplot(mpg, aes(cty, hwy))
f <- ggplot(mpg, aes(class, hwy))
g <- ggplot(diamonds, aes(cut, color))
h <- ggplot(diamonds, aes(carat, price))
```

```
i <- ggplot(economics, aes(date, unemploy))
df <- data.frame(grp = c("A", "B"), fit = 4.5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))
data <- data.frame(murder = USArrests$Murder,
                    state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2))
l <- ggplot(seals, aes(long, lat))
s <- ggplot(mpg, aes(f1, fill = drv))
t <- ggplot(mpg, aes(cty, hwy)) + geom_point()
```

Notice that, in some places, the aesthetics are defined using the full aesthetic name-value pair (e.g., `aes(x = long, y = lat)`), while in other places the code relies on position for defining which column of a dataframe maps to which aesthetic (e.g., `aes(cty, hwy)` or `aes(f1)`). Either is fine, although relying on position can result in errors if you are not very familiar with the order in which parameters are defined for a function.

This code will vary based on the examples you try, but here is some code for one set of examples:

```
b + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900))
c + geom_dotplot()
f + geom_violin(scale = "area")
h + geom_hex()
j + geom_pointrange()
k + geom_map(aes(map_id = state), map = map) +
  expand_limits(x = map$long, y = map$lat)
l + geom_contour(aes(z = z))
n <- d + geom_bar(aes(fill = f1))
n + scale_fill_brewer(palette = "Blues")
o <- c + geom_dotplot(aes(fill = ...x...))
o + scale_fill_gradient(low = "red", high = "yellow")
t + facet_grid(year ~ f1)
s + geom_bar(position = "fill")
```


Chapter 5

Reproducible research #1

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

5.1 What is reproducible research?

Download a pdf of the lecture slides for this video.

A data analysis is **reproducible** if all the information (data, files, etc.) required is available for someone else to re-do your entire analysis. This includes:

- Data available
- All code for cleaning raw data
- All code and software (specific versions, packages) for analysis

Some advantages of making your research reproducible are:

- You can (easily) figure out what you did six months from now.
- You can (easily) make adjustments to code or data, even early in the process, and re-run all analysis.
- When you're ready to publish, you can (easily) do a last double-check of your full analysis, from cleaning the raw data through generating figures and tables for the paper.
- You can pass along or share a project with others.
- You can give useful code examples to people who want to extend your research.

Here is a famous research example of the dangers of writing code that is hard to double-check or confirm:

- The Economist
- The New York Times
- Simply Statistics

Some of the steps required to making research reproducible are:

- All your raw data should be saved in the project directory. You should have clear documentation on the source of all this data.
- Scripts should be included with all the code used to clean this data into the data set(s) used for final analyses and to create any figures and tables.
- You should include details on the versions of any software used in analysis (for R, this includes the version of R as well as versions of all packages used).
- If possible, there should be no “by hand” steps used in the analysis; instead, all steps should be done using code saved in scripts. For example, you should use a script to clean data, rather than cleaning it by hand in Excel. If any “non-scriptable” steps are unavoidable, you should very clearly document those steps.

There are several software tools that can help you improve the reproducibility of your research:

- **knitr**: Create files that include both your code and text. These can be rendered to create final reports and papers. They keep code within the final file for the report.
- **knitr complements**: Create fancier tables and figures within RMarkdown documents. Packages include `tikzDevice`, `animate`, `xtables`, and `pander`.
- **packrat**: Save versions of each package used for the analysis, then load those package versions when code is run again in the future.

In this section, I will focus on using `knitr` and RMarkdown files.

5.2 Markdown

Download a pdf of the lecture slides for this video.

R Markdown files are mostly written using Markdown. To write R Markdown files, you need to understand what markup languages like Markdown are and how they work.

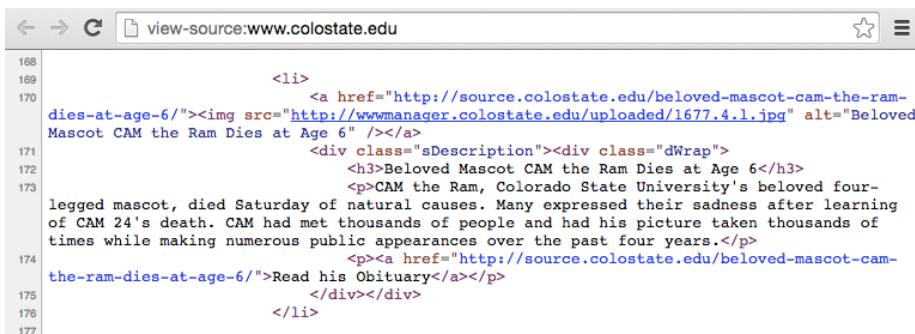
In Word and other word processing programs you have used, you can add formatting using buttons and keyboard shortcuts (e.g., “Ctrl-B” for bold). The file saves the words you type. It also saves the formatting, but you see the final output, rather than the formatting markup, when you edit the file (WYSIWYG – what you see is what you get).

In markup languages, on the other hand, you markup the document directly to show what formatting the final version should have (e.g., you type ****bold**** in the file to end up with a document with **bold**).

Examples of markup languages include:

- HTML (HyperText Markup Language)
- LaTex
- Markdown (a “lightweight” markup language)

For example, Figure 5.1 shows some marked-up HTML code from CSU’s website, while Figure 5.2 shows how that file looks when it’s rendered by a web browser.



The screenshot shows a browser window with the URL "view-source:www.colostate.edu" in the address bar. The page content is a snippet of HTML code. The code includes line numbers on the left (168, 169, 170, 171, 172, 173, 174, 175, 176, 177) and various HTML tags like , , , <div class="sDescription"><div class="dWrap">, <h3>Beloved Mascot CAM the Ram Dies at Age 6</h3>, <p>CAM the Ram, Colorado State University's beloved four-legged mascot, died Saturday of natural causes. Many expressed their sadness after learning of CAM 24's death. CAM had met thousands of people and had his picture taken thousands of times while making numerous public appearances over the past four years.</p>, and <p>Read his Obituary</p>. The code uses CSS classes like "sDescription" and "dWrap".

Figure 5.1: Example of the source of an HTML file.



Figure 5.2: Example of a rendered HTML file.

To write a file in Markdown, you'll need to learn the conventions for creating formatting. This table shows what you would need to write in a flat file for some common formatting choices:

Code	Rendering	Explanation
<code>**text**</code>	<code>**text**</code>	boldface
<code>*text*</code>	<code>*text*</code>	italicized
<code>[text](www.google.com)</code>	<code>[text](www.google.com)</code>	hyperlink
<code># text</code>		first-level header
<code>## text</code>		second-level header

Some other simple things you can do in Markdown include:

- Lists (ordered or bulleted)
- Equations
- Tables
- Figures from file
- Block quotes
- Superscripts

For more Markdown conventions, see RStudio's R Markdown Reference Guide (link also available through "Help" in RStudio).

5.3 Literate programming in R

Download a pdf of the lecture slides for this video.

Literate programming, an idea developed by Donald Knuth, mixes code that can be executed with regular text. The files you create can then be rendered, to run any embedded code. The final output will have results from your code and the regular text.

The `knitr` package can be used for literate programming in R. In essence, `knitr` allows you to write an R Markdown file that can be rendered into a pdf, Word, or HTML document.

Here are the basics of opening and rendering an R Markdown file in RStudio:

- To open a new R Markdown file, go to "File" -> "New File" -> "RMarkdown..." -> for now, chose a "Document" in "HTML" format.
- This will open a new R Markdown file in RStudio. The file extension for R Markdown files is ".Rmd".
- The new file comes with some example code and text. You can run the file as-is to try out the example. You will ultimately delete this example code and text and replace it with your own.

- Once you “knit” the R Markdown file, R will render an HTML file with the output. This is automatically saved in the same directory where you saved your .Rmd file.
- Write everything besides R code using Markdown syntax.

To include R code in an RMarkdown document, you need to separate off the code chunk using the following syntax:

```
```{r}
my_vec <- 1:10
```
```

This syntax tells R how to find the start and end of pieces of R code when the file is rendered. R will walk through, find each piece of R code, run it and create output (printed output or figures, for example), and then pass the file along to another program to complete rendering (e.g., Tex for pdf files).

You can specify a name for each chunk, if you’d like, by including it after “r” when you begin your chunk. For example, to give the name `load_nepali` to a code chunk that loads the `nepali` dataset, specify that name in the start of the code chunk:

```
```{r load_nepali}
library(faraway)
data(nepali)
```
```

Here are a couple of tips for naming code chunks:

- Chunk names must be unique across a document.
- Any chunks you don’t name are given numbers by `knitr`.

You do not have to name each chunk. However, there are some advantages:

- It will be easier to find any errors.
- You can use the chunk labels in referencing for figure labels.
- You can reference chunks later by name.

You can add options when you start a chunk. Many of these options can be set as TRUE / FALSE and include:

| Option | Action |
|------------|---|
| ‘echo’ | Print out the R code? |
| ‘eval’ | Run the R code? |
| ‘messages’ | Print out messages? |
| ‘warnings’ | Print out warnings? |
| ‘include’ | If FALSE, run code, but don’t print code or results |

Other chunk options take values other than TRUE / FALSE. Some you might want to include are:

| Option | Action |
|-------------------|---|
| results | How to print results (e.g., <code>hide</code> runs the code, but doesn’t print the results) |
| fig.width | Width to print your figure, in inches (e.g., <code>fig.width = 4</code>) |
| fig.height | Height to print your figure |

Add these options in the opening brackets and separate multiple ones with commas:

```
```{r messages = FALSE, echo = FALSE}
nepali[1, 1:3]
```
```

I will cover other chunk options later, once you’ve gotten the chance to try writing R Markdown files.

You can set “global” options at the beginning of the document. This will create new defaults for all of the chunks in the document. For example, if you want `echo`, `warning`, and `message` to be FALSE by default in all code chunks, you can run:

```
```{r global_options}
knitr:::opts_chunk$set(echo = FALSE, message = FALSE,
warning = FALSE)
```
```

If you set both global and local chunk options that you set specifically for a chunk will take precedence over global options. For example, running a document with:

```
```{r global_options}
knitr:::opts_chunk$set(echo = FALSE, message = FALSE,
warning = FALSE)
```
```

```
```{r check_nepali, echo = TRUE}
head(nepali, 1)
```
```

would print the code for the `check_nepali` chunk, because the option specified for that specific chunk (`echo = TRUE`) would override the global option (`echo = FALSE`).

You can also include R output directly in your text (“inline”) using backticks:

“There are `r nrow(nepali)` observations in the `nepali` data set. The average age is `r mean(nepali\$age, na.rm = TRUE)` months.”

Once the file is rendered, this gives:

“There are 1000 observations in the `nepali` data set. The average age is 37.662 months.”

Download a pdf of the lecture slides for this video.

Here are two tips that will help you diagnose some problems rendering R Markdown files:

- Be sure to save your R Markdown file before you run it.
- All the code in the file will run “from scratch”— as if you just opened a new R session.
- The code will run using, as a working directory, the directory where you saved the R Markdown file.

You’ll want to try out pieces of your code as you write an R Markdown document. There are a few ways you can do that:

- You can run code in chunks just like you can run code from a script (Ctrl-Return or the “Run” button).
- You can run all the code in a chunk (or all the code in all chunks) using the different options under the “Run” button in RStudio.
- All the “Run” options have keyboard shortcuts, so you can use those.

You can render R Markdown documents to other formats:

- Word

- Pdf (requires that you've installed "Tex" on your computer.)
- Slides (ioslides)

Click the button to the right of "Knit" to see different options for rendering on your computer.

You can freely post your RMarkdown documents at RPubs. If you want to post to RPubs, you need to create an account. Once you do, you can click the "Publish" button on the window that pops up with your rendered file. RPubs can also be a great place to look for interesting example code, although it sometimes can be pretty overwhelmed with MOOC homework.

If you'd like to find out more, here are two good how-to books on reproducible research in R (the CSU library has both in hard copy):

- *Reproducible Research with R and RStudio*, Christopher Gandrud
- *Dynamic Documents with R and knitr*, Yihui Xie

5.4 Style guidelines

Download a pdf of the lecture slides for this video.

R style guidelines provide rules for how to format code in an R script. Some people develop their own style as they learn to code. However, it is easy to get in the habit of following style guidelines, and they offer some important advantages:

- Clean code is easier to read and interpret later.
- It's easier to catch and fix mistakes when code is clear.
- Others can more easily follow and adapt your code if it's clean.
- Some style guidelines will help prevent possible problems (e.g., avoiding . in function names).

For this course, we will use R style guidelines from two sources:

- Google's R style guidelines
- Hadley Wickham's R style guidelines

These two sets of style guidelines are very similar.

Hear are a few guidelines we've already covered in class:

- Use <-, not =, for assignment.
- Guidelines for naming objects:

- All lowercase letters or numbers
- Use underscore (_) to separate words, not camelCase or a dot (.)
(this differs for Google and Wickham style guides)
- Have some consistent names to use for “throw-away” objects (e.g.,
`df`, `ex`, `a`, `b`)
- Make names meaningful
 - Descriptive names for R scripts (“random_group_assignment.R”)
 - Nouns for objects (`todays_groups` for an object with group assignments)
 - Verbs for functions (`make_groups` for the function to assign groups)

5.4.1 Line length

Google: **Keep lines to 80 characters or less**

To set your script pane to be limited to 80 characters, go to “RStudio” -> “Preferences” -> “Code” -> “Display”, and set “Margin Column” to 80.

```
# Do
my_df <- data.frame(n = 1:3,
                      letter = c("a", "b", "c"),
                      cap_letter = c("A", "B", "C"))

# Don't
my_df <- data.frame(n = 1:3, letter = c("a", "b", "c"), cap_letter = c("A", "B", "C"))
```

This guideline helps ensure that your code is formatted in a way that you can see all of the code without scrolling horizontally (left and right).

5.4.2 Spacing

- Binary operators (e.g., `<-`, `+`, `-`) should have a space on either side
- A comma should have a space after it, but not before.
- Colons should not have a space on either side.
- Put spaces before and after `=` when assigning parameter arguments

```
# Do
shots_per_min <- worldcup$Shots / worldcup$Time
#Don't
```

```

shots_per_min<-worldcup$Shots/worldcup$Time

#Do
ave_time <- mean(worldcup[1:10 , "Time"])
#Don't
ave_time<-mean(worldcup[1 : 10 , "Time"])

```

5.4.3 Semicolons

Although you can use a semicolon to put two lines of code on the same line, you should avoid it.

```

# Do
a <- 1:10
b <- 3

# Don't
a <- 1:10; b <- 3

```

5.4.4 Commenting

- For a comment on its own line, use `#`. Follow with a space, then the comment.
- You can put a short comment at the end of a line of R code. In this case, put two spaces after the end of the code, one `#`, and one more space before the comment.
- If it helps make it easier to read your code, separate sections using a comment character followed by many hyphens (e.g., `#-----`). Anything after the comment character is “muted”.

```

# Read in health data -----
# Clean exposure data -----

```

5.4.5 Indentation

Google:

- Within function calls, line up new lines with first letter after opening parenthesis for parameters to function calls:

Example:

```
# Relabel sex variable
nepali$sex <- factor(nepali$sex,
  levels = c(1, 2),
  labels = c("Male", "Female"))
```

5.4.6 Code grouping

- Group related pieces of code together.
- Separate blocks of code by empty spaces.

```
# Load data
library(faraway)
data(nepali)

# Relabel sex variable
nepali$sex <- factor(nepali$sex,
  levels = c(1, 2),
  labels = c("Male", "Female"))
```

Note that this grouping often happens naturally when using tidyverse functions, since they encourage piping (%>% and +).

5.4.7 Broader guidelines

- Omit needless code.
- Don't repeat yourself.

We'll learn more about satisfying these guidelines when we talk about writing your own functions in the next part of the class.

5.5 More with knitr

Download a pdf of the lecture slides for this video.

5.5.1 Equations in knitr

You can write equations in RMarkdown documents by setting them apart with dollar signs (\$). For an equation on a line by itself (**display equation**), you two \$s before and after the equation, on separate lines, then use LaTex syntax for writing the equations.

To help with this, you may want to use this LaTex math cheat sheet.. You may also find an online LaTex equation editor like [Codecogs.com](http://www.codecogs.com) helpful.

Note: Equations denoted this way will always compile for pdf documents, but won't always come through on Markdown files (for example, GitHub won't compile math equations).

For example, writing this in your R Markdown file:

```
$$
E(Y_{\{t\}}) \sim \beta_0 + \beta_1 X_{\{1\}}
$$
```

will result in this rendered equation:

$$E(Y_t) \sim \beta_0 + \beta_1 X_1$$

To put math within a sentence (**inline equation**), just use one \$ on either side of the math. For example, writing this in a R Markdown file:

```
"We are trying to model $E(Y_{\{t\}})$."
```

The rendered document will show up as:

“We are trying to model $E(Y_t)$.”

5.5.2 Figures from file

You can include not only figures that you create with R, but also figures that you have saved on your computer.

The best way to do that is with the `include_graphics` function in `knitr`:

```
library(knitr)
include_graphics("figures/CSU_ram.png")
```



This example would include a figure with the filename “MyFigure.png” that is saved in the “figures” sub-directory of the parent directory of the directory where your .Rmd is saved. Don’t forget that you will need to give an absolute pathway or the relative pathway **from the directory where the .Rmd file is saved**.

5.5.3 Saving graphics files

You can save figures that you create in R. Typically, you won’t need to save figures for an R Markdown file, since you can include figure code directly. However, you will sometimes want to save a figure from a script. You have two options:

- Use the “Export” choice in RStudio
- Write code to export the figure in your R script

To make your research more reproducible, use the second choice.

To use code export a figure you created in R, take three steps:

1. Open a graphics device (e.g., `pdf("MyFile.pdf")`).
2. Write the code to print your plot.
3. Close the graphics device using `dev.off()`.

For example, the following code would save a scatterplot of time versus passes as a pdf named “MyFigure” in the “figures” subdirectory of the current working directory:

```
pdf("figures/MyFigure.pdf", width = 8, height = 6)
ggplot(worldcup, aes(x = Time, y = Passes)) +
  geom_point(aes(color = Position)) +
  theme_bw()
dev.off()
```

If you create multiple plots before you close the device, they'll all save to different pages of the same pdf file.

You can open a number of different graphics devices. Here are some of the functions you can use to open graphics devices:

- `pdf`
- `png`
- `bmp`
- `jpeg`
- `tiff`
- `svg`

You will use a device-specific function to open a graphics device (e.g., `pdf`). However, you will always close these devices with `dev.off`.

Most of the functions to open graphics devices include parameters like `height` and `width`. These can be used to specify the size of the output figure. The units for these depend on the device (e.g., inches for `pdf`, pixels by default for `png`). Use the helpfile for the function to determine these details.

5.5.4 Tables in R Markdown

If you want to create a nice, formatted table from an R data frame, you can do that using `kable` from the `knitr` package.

```
my_df <- data.frame(letters = c("a", "b", "c"),
                     numbers = 1:3)
kable(my_df)
```

| letters | numbers |
|---------|---------|
| a | 1 |
| b | 2 |
| c | 3 |

There are a few options for the `kable` function:

| arg | expl |
|-----------------------|---|
| <code>colnames</code> | Column names (default: column name in the data frame) |
| <code>align</code> | A vector giving the alignment for each column ('l', 'c', 'r') |
| <code>caption</code> | Table caption |

Table 5.3: My new table

| First 3 letters | First 3 numbers |
|-----------------|-----------------|
| a | 1.50 |
| b | -0.61 |
| c | 0.46 |

| arg | expl |
|---------------------|--|
| <code>digits</code> | Number of digits to round to. If you want to round columns different amounts, use a vector with one element for each column. |

```
my.df <- data.frame(letters = c("a", "b", "c"),
                     numbers = rnorm(3))
kable(my.df, digits = 2, align = c("r", "c"),
      caption = "My new table",
      col.names = c("First 3 letters",
                   "First 3 numbers"))
```

From Yihui:

“Want more features? No, that is all I have. You should turn to other packages for help. I’m not going to reinvent their wheels.”

If you want to do fancier tables, you may want to explore the `xtable` and `pander` packages. As a note, these might both be more effective when compiling to pdf, rather than html.

5.6 In-course exercise Chapter 5

For all of today’s tasks, you’ll use the code from last week’s in-course exercise to do the exercises. This week we are not focusing on writing new code, but rather on how to take R code and put it in an R Markdown file, so we can create reports from files that include the original code.

5.6.1 Creating a Markdown document

First, you’ll create a Markdown document, without any R code in it yet.

In RStudio, go to “File” -> “New File” -> “R Markdown”. From the window that brings up, choose “Document” on the left-hand column and “HTML” as the output format. A new file will open in the script pane of your RStudio session. Save this file (you may pick the name and directory). The file extension should be “.Rmd”.

First, before you try to write your own Markdown, try rendering the example that the script includes by default. (This code is always included, as a template, when you first open a new RMarkdown file using the RStudio “New file” interface we used in this example.) Try rendering this default R Markdown example by clicking the “Knit” button at the top of the script file.

For some of you, you may not yet have everything you need on your computer to be able to get this to work. If so, let me know. RStudio usually includes all the necessary tools when you install it, but there may be some exceptions.

If you could get the document to knit, do the following tasks:

- Look through the HTML document that was created. Compare it to the R Markdown script that created it, and see if you can understand, at least broadly, what’s going on.
- Look in the directory where you saved the R Markdown file. You should now also see a new, .html file in that folder. Try opening it with a web browser like Safari.
- Go back to the R Markdown file. Delete everything after the initial header information (everything after the 6th line). In the header information, make sure the title, author, and date are things you’re happy with. If not, change them.
- Using Markdown syntax, write up a description of the data (`worldcup`) we used last week to create the fancier figure. Try to include the following elements:
 - Bold and italic text
 - Hyperlinks
 - A list, either ordered or bulleted
 - Headers

5.6.2 Adding in R code

Now incorporate the R code from previous weeks’ exercises into your document. Once you get the document to render with some basic pieces of code in it, try the following:

- Try some different chunk options. For example, try setting `echo = FALSE` in some of your code chunks. Similarly, try using the options `results = "hide"` and `include = FALSE`.

- You should have at least one code chunk that generates figures. Try experimenting with the `fig.width` and `fig.height` options for the chunk to change the size of the figure.
- Try using the global commands. See if you can switch the `echo` default value for this document from TRUE (the usual default) to FALSE.

5.6.3 Working with R Markdown documents

Finally, try the following tasks to get some experience working with R Markdown files in RStudio:

- Go to one of your code chunks. Locate the small gray arrow just to the left of the line where you initiate the code chunk. Click on it and see what happens. Then click on it again.
- Put your cursor inside one of your code chunks. Try using the “Run” button (or Ctrl-Return) to run code in that chunk at your R console. Did it work?
- Pick a code chunk in your document. Put your cursor somewhere in the code in that chunk. Click on the “Run” button and choose “Run All Chunks Above”. What did that do? If it did not work, what do you think might be going on? (Hint: Check `getwd()` and think about which directory you’ve used to save your R Markdown file.)
- Pick another chunk of code. Put the cursor somewhere in the code for that chunk. Click on the “Run” button and choose “Run Current Chunk”. Then try “Run Next Chunk”. Try to figure out all the options the “Run” button gives you and when each might be useful.
- Click on the small gray arrow to the right of the “Knit HTML” button. If the option is offered, select “Knit Word” and try it. What does this do?

5.6.4 R style guidelines

Go through all the R code in your R Markdown file. Are there are places where your code is not following style conventions for R? Clean up your code to correct any of these issues.

Part III

Part III: Intermediate

Chapter 6

Entering and cleaning data #2

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

6.1 Tidy data

Download a pdf of the lecture slides for this video.

All of the material in this section comes directly from Hadley Wickham's paper on tidy data. You will need to read this paper to prepare for the quiz on this section.

Getting your data into a “tidy” format makes it easier to model and plot. By taking the time to tidy your data at the start of an analysis, you will save yourself time, and make it easier to plan out later steps.

Characteristics of tidy data are:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Here are five common problems that Hadley Wickham has identified that keep data from being tidy:

1. Column headers are values, not variable names.

2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables.

Here are examples (again, from Hadley Wickham's paper on tidy data, which is required reading for this week of the course) of each of these problems.

1. Column headers are values, not variable names.

| religion | <\$10k | \$10-20k | \$20-30k | \$30-40k | \$40-50k | \$50-75k |
|-------------------------|--------|----------|----------|----------|----------|----------|
| Agnostic | 27 | 34 | 60 | 81 | 76 | 137 |
| Atheist | 12 | 27 | 37 | 52 | 35 | 70 |
| Buddhist | 27 | 21 | 30 | 34 | 33 | 58 |
| Catholic | 418 | 617 | 732 | 670 | 638 | 1116 |
| Don't know/refused | 15 | 14 | 15 | 11 | 10 | 35 |
| Evangelical Prot | 575 | 869 | 1064 | 982 | 881 | 1486 |
| Hindu | 1 | 9 | 7 | 9 | 11 | 34 |
| Historically Black Prot | 228 | 244 | 236 | 238 | 197 | 223 |
| Jehovah's Witness | 20 | 27 | 24 | 24 | 21 | 30 |
| Jewish | 19 | 19 | 25 | 25 | 30 | 95 |

Solution:

| religion | income | freq |
|----------|--------------------|------|
| Agnostic | <\$10k | 27 |
| Agnostic | \$10-20k | 34 |
| Agnostic | \$20-30k | 60 |
| Agnostic | \$30-40k | 81 |
| Agnostic | \$40-50k | 76 |
| Agnostic | \$50-75k | 137 |
| Agnostic | \$75-100k | 122 |
| Agnostic | \$100-150k | 109 |
| Agnostic | >150k | 84 |
| Agnostic | Don't know/refused | 96 |

2. Multiple variables are stored in one column.

| country | year | column | cases |
|---------|------|--------|-------|
| AD | 2000 | m014 | 0 |
| AD | 2000 | m1524 | 0 |
| AD | 2000 | m2534 | 1 |
| AD | 2000 | m3544 | 0 |
| AD | 2000 | m4554 | 0 |
| AD | 2000 | m5564 | 0 |
| AD | 2000 | m65 | 0 |
| AE | 2000 | m014 | 2 |
| AE | 2000 | m1524 | 4 |
| AE | 2000 | m2534 | 4 |
| AE | 2000 | m3544 | 6 |
| AE | 2000 | m4554 | 5 |
| AE | 2000 | m5564 | 12 |
| AE | 2000 | m65 | 10 |
| AE | 2000 | f014 | 3 |

Solution:

| country | year | sex | age | cases |
|---------|------|-----|-------|-------|
| AD | 2000 | m | 0-14 | 0 |
| AD | 2000 | m | 15-24 | 0 |
| AD | 2000 | m | 25-34 | 1 |
| AD | 2000 | m | 35-44 | 0 |
| AD | 2000 | m | 45-54 | 0 |
| AD | 2000 | m | 55-64 | 0 |
| AD | 2000 | m | 65+ | 0 |
| AE | 2000 | m | 0-14 | 2 |
| AE | 2000 | m | 15-24 | 4 |
| AE | 2000 | m | 25-34 | 4 |
| AE | 2000 | m | 35-44 | 6 |
| AE | 2000 | m | 45-54 | 5 |
| AE | 2000 | m | 55-64 | 12 |
| AE | 2000 | m | 65+ | 10 |
| AE | 2000 | f | 0-14 | 3 |

3. Variables are stored in both rows and columns.

| id | year | month | element | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 |
|---------|------|-------|---------|----|------|------|----|------|----|----|----|
| MX17004 | 2010 | 1 | tmax | — | — | — | — | — | — | — | — |
| MX17004 | 2010 | 1 | tmin | — | — | — | — | — | — | — | — |
| MX17004 | 2010 | 2 | tmax | — | 27.3 | 24.1 | — | — | — | — | — |
| MX17004 | 2010 | 2 | tmin | — | 14.4 | 14.4 | — | — | — | — | — |
| MX17004 | 2010 | 3 | tmax | — | — | — | — | 32.1 | — | — | — |
| MX17004 | 2010 | 3 | tmin | — | — | — | — | 14.2 | — | — | — |
| MX17004 | 2010 | 4 | tmax | — | — | — | — | — | — | — | — |
| MX17004 | 2010 | 4 | tmin | — | — | — | — | — | — | — | — |
| MX17004 | 2010 | 5 | tmax | — | — | — | — | — | — | — | — |
| MX17004 | 2010 | 5 | tmin | — | — | — | — | — | — | — | — |

Solution:

| id | date | element | value | id | date | tmax | tmin |
|---------|------------|---------|-------|---------|------------|------|------|
| MX17004 | 2010-01-30 | tmax | 27.8 | MX17004 | 2010-01-30 | 27.8 | 14.5 |
| MX17004 | 2010-01-30 | tmin | 14.5 | MX17004 | 2010-02-02 | 27.3 | 14.4 |
| MX17004 | 2010-02-02 | tmax | 27.3 | MX17004 | 2010-02-03 | 24.1 | 14.4 |
| MX17004 | 2010-02-02 | tmin | 14.4 | MX17004 | 2010-02-11 | 29.7 | 13.4 |
| MX17004 | 2010-02-03 | tmax | 24.1 | MX17004 | 2010-02-23 | 29.9 | 10.7 |
| MX17004 | 2010-02-03 | tmin | 14.4 | MX17004 | 2010-03-05 | 32.1 | 14.2 |
| MX17004 | 2010-02-11 | tmax | 29.7 | MX17004 | 2010-03-10 | 34.5 | 16.8 |
| MX17004 | 2010-02-11 | tmin | 13.4 | MX17004 | 2010-03-16 | 31.1 | 17.6 |
| MX17004 | 2010-02-23 | tmax | 29.9 | MX17004 | 2010-04-27 | 36.3 | 16.7 |
| MX17004 | 2010-02-23 | tmin | 10.7 | MX17004 | 2010-05-27 | 33.2 | 18.2 |

4. Multiple types of observational units are stored in the same table.

| year | artist | time | track | date | week | rank |
|------|--------------|------|-------------------------|------------|------|------|
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-02-26 | 1 | 87 |
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-03-04 | 2 | 82 |
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-03-11 | 3 | 72 |
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-03-18 | 4 | 77 |
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-03-25 | 5 | 87 |
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-04-01 | 6 | 94 |
| 2000 | 2 Pac | 4:22 | Baby Don't Cry | 2000-04-08 | 7 | 99 |
| 2000 | 2Ge+her | 3:15 | The Hardest Part Of ... | 2000-09-02 | 1 | 91 |
| 2000 | 2Ge+her | 3:15 | The Hardest Part Of ... | 2000-09-09 | 2 | 87 |
| 2000 | 2Ge+her | 3:15 | The Hardest Part Of ... | 2000-09-16 | 3 | 92 |
| 2000 | 3 Doors Down | 3:53 | Kryptonite | 2000-04-08 | 1 | 81 |
| 2000 | 3 Doors Down | 3:53 | Kryptonite | 2000-04-15 | 2 | 70 |
| 2000 | 3 Doors Down | 3:53 | Kryptonite | 2000-04-22 | 3 | 68 |
| 2000 | 3 Doors Down | 3:53 | Kryptonite | 2000-04-29 | 4 | 67 |
| 2000 | 3 Doors Down | 3:53 | Kryptonite | 2000-05-06 | 5 | 66 |

Solution:

| id | artist | track | time | id | date | rank |
|----|---------------------|-------------------------|------|----|------------|------|
| 1 | 2 Pac | Baby Don't Cry | 4:22 | 1 | 2000-02-26 | 87 |
| 2 | 2Ge+her | The Hardest Part Of ... | 3:15 | 1 | 2000-03-04 | 82 |
| 3 | 3 Doors Down | Kryptonite | 3:53 | 1 | 2000-03-11 | 72 |
| 4 | 3 Doors Down | Loser | 4:24 | 1 | 2000-03-18 | 77 |
| 5 | 504 Boyz | Wobble Wobble | 3:35 | 1 | 2000-03-25 | 87 |
| 6 | 98^0 | Give Me Just One Nig... | 3:24 | 1 | 2000-04-01 | 94 |
| 7 | A*Teens | Dancing Queen | 3:44 | 1 | 2000-04-08 | 99 |
| 8 | Aaliyah | I Don't Wanna | 4:15 | 2 | 2000-09-02 | 91 |
| 9 | Aaliyah | Try Again | 4:03 | 2 | 2000-09-09 | 87 |
| 10 | Adams, Yolanda | Open My Heart | 5:30 | 2 | 2000-09-16 | 92 |
| 11 | Adkins, Trace | More | 3:05 | 3 | 2000-04-08 | 81 |
| 12 | Aguilera, Christina | Come On Over Baby | 3:38 | 3 | 2000-04-15 | 70 |
| 13 | Aguilera, Christina | I Turn To You | 4:00 | 3 | 2000-04-22 | 68 |
| 14 | Aguilera, Christina | What A Girl Wants | 3:18 | 3 | 2000-04-29 | 67 |
| 15 | Alice Deejay | Better Off Alone | 6:50 | 3 | 2000-05-06 | 66 |

5. A single observational unit is stored in multiple tables.

Example: exposure and outcome data stored in different files:

- File 1: Daily mortality counts
- File 2: Daily air pollution measurements

6.2 Joining datasets

Download a pdf of the lecture slides for this video.

So far, you have only worked with a single data source at a time. When you work on your own projects, however, you typically will need to merge together two or more datasets to create the a data frame to answer your research question. For example, for air pollution epidemiology, you will often have to join several datasets:

- Health outcome data (e.g., number of deaths per day)
- Air pollution concentrations
- Weather measurements (since weather can be a confounder)
- Demographic data

The `dplyr` package has a family of different functions to join two dataframes together, the `*_join` family of functions. All combine two dataframes, which I'll call `x` and `y` here.

The functions include:

- `inner_join(x, y)`: Keep only rows where there are observations in both `x` and `y`.
- `left_join(x, y)`: Keep all rows from `x`, whether they have a match in `y` or not.
- `right_join(x, y)`: Keep all rows from `y`, whether they have a match in `x` or not.
- `full_join(x, y)`: Keep all rows from both `x` and `y`, whether they have a match in the other dataset or not.

In the examples, I'll use two datasets, `x` and `y`. Both datasets include the column `course`. The other column in `x` is `grade`, while the other column in `y` is `day`. Observations exist for courses `x` and `y` in both datasets, but for `w` and `z` in only one dataset.

```
x <- data.frame(course = c("x", "y", "z"),
                 grade = c(90, 82, 78))
y <- data.frame(course = c("w", "x", "y"),
                 day = c("Tues", "Mon / Fri", "Tue"))
```

Here is what these two example datasets look like:

```
x
```

```
## course grade
## 1     x    90
## 2     y    82
## 3     z    78
```

```
y
```

```
## course      day
## 1     w      Tues
## 2     x Mon / Fri
## 3     y      Tue
```

With `inner_join`, you'll only get the observations that show up in both datasets. That means you'll lose data on `z` (only in the first dataset) and `w` (only in the second dataset).

```
inner_join(x, y)
```

```
## Joining with `by = join_by(course)`

## course grade      day
## 1     x    90 Mon / Fri
## 2     y    82      Tue
```

With `left_join`, you'll keep everything in `x` (the “left” dataset), but not keep things in `y` that don't match something in `x`. That means that, here, you'll lose `w`:

```
left_join(x, y)
```

```
## Joining with `by = join_by(course)`

## course grade      day
## 1     x    90 Mon / Fri
## 2     y    82      Tue
## 3     z    78      <NA>
```

`right_join` is the opposite:

```
right_join(x, y)

## Joining with `by = join_by(course)`

##   course grade      day
## 1     x    90 Mon / Fri
## 2     y    82   Tue
## 3     w    NA Tues
```

`full_join` keeps everything from both datasets:

```
full_join(x, y)

## Joining with `by = join_by(course)`

##   course grade      day
## 1     x    90 Mon / Fri
## 2     y    82   Tue
## 3     z    78 <NA>
## 4     w    NA Tues
```

6.3 Longer data

Download a pdf of the lecture slides for this video.

There are two functions from the `tidyverse` package (another member of the tidyverse) that you can use to change between wider and longr data: `pivot_longer` and `pivot_wider`. Here is a description of these two functions:

- `pivot_longer`: Takes several columns and pivots them down into two columns. One of the new columns contains the former column names and the other contains the former cell values.
- `pivot_wider`: Takes two columns and pivots them up into multiple columns. Column names for the new columns will come from one column and the cell values from the other.

The following examples are show the effects of making a dataset longer or wider.

Here is some simulated wide data:

```
wide_stocks[1:3, ]
```

```
## # A tibble: 3 x 4
##   time      X      Y      Z
##   <date>    <dbl>  <dbl>  <dbl>
## 1 2009-01-01 0.677 -0.0274 4.69
## 2 2009-01-02 0.886  0.891 -0.111
## 3 2009-01-03 0.400  -3.79  -4.05
```

In the `wide_stocks` dataset, there are separate columns for three different stocks (X, Y, and Z). Each cell gives the value for a certain stock on a certain day. This data isn't "tidy", because the identify of the stock (X, Y, or Z) is a variable, and you'll probably want to include it as a variable in modeling.

```
wide_stocks[1:3, ]
```

```
## # A tibble: 3 x 4
##   time      X      Y      Z
##   <date>    <dbl>  <dbl>  <dbl>
## 1 2009-01-01 0.677 -0.0274 4.69
## 2 2009-01-02 0.886  0.891 -0.111
## 3 2009-01-03 0.400  -3.79  -4.05
```

If you want to convert the dataframe to have all stock values in a single column, you can use `pivot_longer` to convert wide data to long data:

```
long_stocks <- pivot_longer(data = wide_stocks,
                             cols = -time,
                             names_to = "stock",
                             values_to = "price")
long_stocks[1:5, ]
```

```
## # A tibble: 5 x 3
##   time      stock  price
##   <date>    <chr>   <dbl>
## 1 2009-01-01 X       0.677
```

```
## 2 2009-01-01 Y      -0.0274
## 3 2009-01-01 Z      4.69
## 4 2009-01-02 X      0.886
## 5 2009-01-02 Y      0.891
```

In this “longer” dataframe, there is now one column that gives the identify of the stock (`stock`) and another column that gives the price of that stock that day (`price`):

```
long_stocks[1:5, ]
```

```
## # A tibble: 5 x 3
##   time     stock   price
##   <date>   <chr>    <dbl>
## 1 2009-01-01 X      0.677
## 2 2009-01-01 Y     -0.0274
## 3 2009-01-01 Z      4.69
## 4 2009-01-02 X      0.886
## 5 2009-01-02 Y      0.891
```

The format for a `pivots_longer` call is:

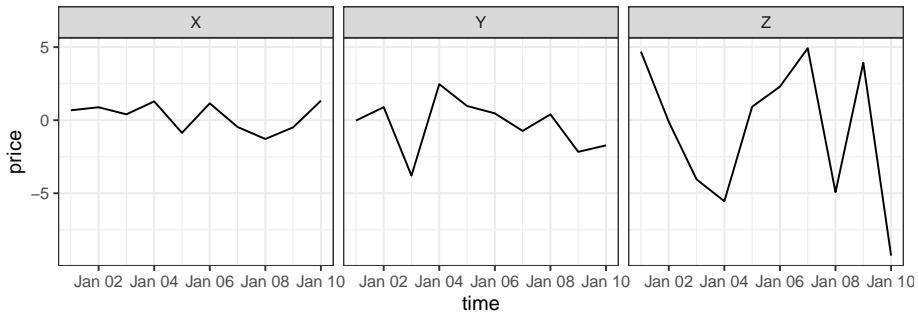
```
## Generic code
new_df <- pivot_longer(old_df,
                       cols = [name(s) of the columns you want to make longer],
                       names_to = [name of new column to store the old column names],
                       values_to = [name of new column to store the old values])
```

Three important notes:

- Everything is pivoted into one of two columns – one column with the old column names, and one column with the old cell values
- With the `names_to` and `values_to` arguments, you are just providing column names for the two columns that everything’s pivoted into.
- If there is a column you don’t want to include in the pivot (`date` in the example), use `-` to exclude it in the `cols` argument.

Notice how easy it is, now that the data is longer, to use `stock` for aesthetics of faceting in a `ggplot2` call:

```
ggplot(long_stocks, aes(x = time, y = price)) +
  geom_line() +
  facet_grid(. ~ stock) +
  theme_bw()
```



If you have data in a “longer” format and would like to make it “wider”, you can use `pivot_wider` to do that:

```
stocks <- pivot_wider(long_stocks,
                      names_from = "stock",
                      values_from = price)
stocks[1:5, ]
```

```
## # A tibble: 5 x 4
##   time       X       Y       Z
##   <date>   <dbl>   <dbl>   <dbl>
## 1 2009-01-01 0.677 -0.0274  4.69
## 2 2009-01-02 0.886  0.891  -0.111
## 3 2009-01-03 0.400 -3.79   -4.05
## 4 2009-01-04 1.29   2.47   -5.55
## 5 2009-01-05 -0.865 0.973   0.922
```

Notice that this reverses the action of `pivot_longer`.

The “wider” your data the less likely it is to be tidy, so won’t use `pivot_wider` frequently when you are preparing data for analysis. However, `pivot_wider` can be very helpful in creating tables for final reports and presentations.

For example, if you wanted to create a table with means and standard deviations for each of the three stocks, you could use `pivot_wider` to rearrange the final summary to create an attractive table.

```
stock_summary <- long_stocks %>%
  group_by(stock) %>%
  summarize(N = n(), mean = mean(price), sd = sd(price))
stock_summary
```

```
## # A tibble: 3 x 4
##   stock     N   mean    sd
##   <chr> <int> <dbl> <dbl>
## 1 X         10  0.263  0.963
## 2 Y         10 -0.325  1.82 
## 3 Z         10 -0.714  4.94
```

```
stock_summary %>%
  mutate("Mean (Std.dev.)" = paste0(round(mean, 2), " (",
    round(sd, 2), ")")) %>%
  dplyr::select(-mean, -sd) %>%
  mutate(N = as.character(N)) %>% # might be able to deal with this in pivot_longer ca
  pivot_longer(cols = -stock, names_to = "Statistic", values_to = "Value") %>%
  pivot_wider(names_from = "stock", values_from = "Value") %>%
  knitr::kable()
```

| Statistic | X | Y | Z |
|-----------------|-------------|--------------|--------------|
| N | 10 | 10 | 10 |
| Mean (Std.dev.) | 0.26 (0.96) | -0.32 (1.82) | -0.71 (4.94) |

Download a pdf of the lecture slides for this video.

Download a pdf of the lecture slides for this video.

6.4 Working with factors

Download a pdf of the lecture slides for this video.

Hadley Wickham has developed a package called **forcats** that helps you work with categorical variables (factors). I'll show some examples of its functions using the **worldcup** dataset:

```
library(forcats)
library(faraway)
data(worldcup)
```

The `fct_recode` function can be used to change the labels of a function (along the lines of using `factor` with `levels` and `labels` to reset factor labels).

One big advantage is that `fct_recode` lets you change labels for some, but not all, levels. For example, here are the team names:

```
library(stringr)
worldcup %>%
  filter(str_detect(Team, "^\u00c9US")) %>%
  slice(1:3) %>% select(Team, Position, Time)

##           Team   Position Time
## Beasley     USA Midfielder   10
## Bocanegra   USA   Defender  390
## Bornstein   USA   Defender  200
```

If you just want to change “USA” to “United States,” you can run:

```
worldcup <- worldcup %>%
  mutate(Team = fct_recode(Team, `United States` = "USA"))
worldcup %>%
  filter(str_detect(Team, "^\u00c9Un")) %>%
  slice(1:3) %>% select(Team, Position, Time)
```

```
##           Team   Position Time
## Beasley United States Midfielder   10
## Bocanegra United States   Defender  390
## Bornstein United States   Defender  200
```

You can use the `fct_lump` function to lump uncommon factors into an “Other” category. For example, to lump the two least common positions together, you can run (`n` specifies how many categories to keep outside of “Other”):

```
worldcup %>%
  dplyr::mutate(Position = forcats::fct_lump(Position, n = 2)) %>%
  dplyr::count(Position)

##    Position   n
## 1   Defender 188
## 2 Midfielder 228
## 3      Other 179
```

You can use the `fct_infreq` function to reorder the levels of a factor from most common to least common:

```
levels(worldcup$Position)

## [1] "Defender"    "Forward"     "Goalkeeper"   "Midfielder"

worldcup <- worldcup %>%
  mutate(Position = fct_infreq(Position))
levels(worldcup$Position)

## [1] "Midfielder"  "Defender"    "Forward"     "Goalkeeper"
```

If you want to reorder one factor by another variable (ascending order), you can use `fct_reorder` (e.g., homework 3). For example, to relevel `Position` by the average shots on goals for each position, you can run:

```
levels(worldcup$Position)

## [1] "Midfielder"  "Defender"    "Forward"     "Goalkeeper"

worldcup <- worldcup %>%
  group_by(Position) %>%
  mutate(ave_shots = mean(Shots)) %>%
  ungroup() %>%
  mutate(Position = fct_reorder(Position, ave_shots))
levels(worldcup$Position)

## [1] "Goalkeeper" "Defender"    "Midfielder"  "Forward"
```

6.5 String operations and regular expressions

Download a pdf of the lecture slides for this video.

Download a pdf of the lecture slides for this video.

For these examples, we'll use some data on passengers of the Titanic. You can load this data using:

```
# install.packages("titanic")
library(titanic)
data("titanic_train")
```

We will be using the `stringr` package:

```
library(stringr)
```

This data includes a column called “Name” with passenger names. This column is somewhat messy and includes several elements that we might want to separate (last name, first name, title). Here are the first few values of “Name”:

```
titanic_train %>% select(Name) %>% slice(1:3)
```

| ## | Name |
|------|---|
| ## 1 | Braund, Mr. Owen Harris |
| ## 2 | Cumings, Mrs. John Bradley (Florence Briggs Thayer) |
| ## 3 | Heikkinen, Miss. Laina |

We’ve already done some things to manipulate strings. For example, if we wanted to separate “Name” into last name and first name (including title), we could actually do that with the `separate` function:

```
titanic_train %>%
  select(Name) %>%
  slice(1:3) %>%
  separate(Name, c("last_name", "first_name"), sep = ", ")
```

| ## | last_name | first_name |
|------|--|-----------------|
| ## 1 | Braund | Mr. Owen Harris |
| ## 2 | Cumings Mrs. John Bradley (Florence Briggs Thayer) | Miss. Laina |
| ## 3 | Heikkinen | |

Notice that `separate` is looking for a regular pattern (”,”) and then doing something based on the location of that pattern in each string (splitting the string).

There are a variety of functions in R that can perform manipulations based on finding regular patterns in character strings.

The `str_detect` function will look through each element of a character vector for a designated pattern. If the pattern is there, it will return TRUE, and otherwise FALSE. The convention is:

```
## Generic code
str_detect(string = [vector you want to check],
           pattern = [pattern you want to check for])
```

For example, to create a logical vector specifying which of the Titanic passenger names include “Mrs.”, you can call:

```
mrs <- str_detect(titanic_train$Name, "Mrs.")
head(mrs)
```

```
## [1] FALSE TRUE FALSE TRUE FALSE FALSE
```

The result is a logical vector, so `str_detect` can be used in `filter` to subset data to only rows where the passenger’s name includes “Mrs.”:

```
titanic_train %>%
  filter(str_detect(Name, "Mrs."))
  select(Name) %>%
  slice(1:3)
```

```
##                                     Name
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2          Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

There is an older, base R function called `grepl` that does something very similar (although note that the order of the arguments is reversed).

```
titanic_train %>%
  filter(grepl("Mrs.", Name)) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                                     Name
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2          Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

The `str_extract` function can be used to extract a string (if it exists) from each value in a character vector. It follows similar conventions to `str_detect`:

```
## Generic code
str_extract(string = [vector you want to check],
            pattern = [pattern you want to check for])
```

For example, you might want to extract “Mrs.” if it exists in a passenger’s name:

```
titanic_train %>%
  mutate(mrs = str_extract(Name, "Mrs."))
  select(Name, mrs)
  slice(1:3)
```

```
##                                     Name   mrs
## 1                     Braund, Mr. Owen Harris <NA>
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) Mrs.
## 3                     Heikkinen, Miss. Laina <NA>
```

Notice that now we’re creating a new column (`mrs`) that either has “Mrs.” (if there’s a match) or is missing (`NA`) if there’s not a match.

For this first example, we were looking for an exact string (“Mrs”). However, you can use patterns that match a particular pattern, but not an exact string. For example, we could expand the regular expression to find “Mr.” or “Mrs.”:

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr\\\\.|Mrs\\\\."))
  select(Name, title)
  slice(1:3)
```

```
##                                     Name   title
## 1                     Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs.
## 3                     Heikkinen, Miss. Laina <NA>
```

Note that this pattern uses a special operator (`|`) to find one pattern **or** another. Double backslashes (`\\\`) **escape** the special character “.”.

[Download a pdf of the lecture slides for this video.](#)

[Download a pdf of the lecture slides for this video.](#)

As a note, in regular expressions, all of the following characters are special characters that need to be escaped with backslashes if you want to use them literally:

```
. * + ^ ? $ \ | ( ) [ ] { }
```

Notice that “Mr.” and “Mrs.” both start with “Mr”, end with “.”, and may or may not have an “s” in between.

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr(s)*\\.|")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
##                                     Name title
## 1 Braund, Mr. Owen Harris     Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs.
## 3 Heikkinen, Miss. Laina    <NA>
```

This pattern uses `(s)*` to match zero or more “s”s at this spot in the pattern.

In the previous code, we found “Mr.” and “Mrs.”, but missed “Miss.”. We could tweak the pattern again to try to capture that, as well. For all three, we have the pattern that it starts with “M”, has some lowercase letters, and then ends with “.”.

```
titanic_train %>%
  mutate(title = str_extract(Name, "M[a-z]+\\.|")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
##                                     Name title
## 1 Braund, Mr. Owen Harris     Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs.
## 3 Heikkinen, Miss. Laina    Miss.
```

The last pattern used `[a-z]+` to match one or more lowercase letters. The `[a-z]` is a **character class**.

You can also match digits (`[0-9]`), uppercase letters (`[A-Z]`), just some letters (`[aeiou]`), etc.

You can negate a character class by starting it with `^`. For example, `[^0-9]` will match anything that **isn’t** a digit.

Sometimes, you want to match a pattern, but then only subset a part of it. For example, each passenger seems to have a title (“Mr.”, “Mrs.”, etc.) that comes after “,” and before “.”. We can use this pattern to find the title, but then we get some extra stuff with the match:

```
titanic_train %>%
  mutate(title = str_extract(Name, ",\\s[A-Za-z]*\\.\\s")) %>%
  select(title) %>%
  slice(1:3)

##      title
## 1    Mr.
## 2   Mrs.
## 3 Miss.
```

As a note, in this pattern, `\s` is used to match a space.

We are getting things like “, Mr. ”, when we really want “Mr”. We can use the `str_match` function to do this. We group what we want to extract from the pattern in parentheses, and then the function returns a matrix. The first column is the full pattern match, and each following column gives just what matches within the groups.

```
head(str_match(titanic_train$Name,
               pattern = ",\\s([A-Za-z]*)\\.\\s"))

##      [,1]      [,2]
## [1,] ", Mr. "    "Mr"
## [2,] ", Mrs. "   "Mrs"
## [3,] ", Miss. "  "Miss"
## [4,] ", Mrs. "   "Mrs"
## [5,] ", Mr. "    "Mr"
## [6,] ", Mr. "    "Mr"
```

To get just the title, then, we can run:

```
titanic_train %>%
  mutate(title =
         str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[, 2]) %>%
  select(Name, title) %>%
  slice(1:3)
```

```

##                                     Name title
## 1 Braund, Mr. Owen Harris      Mr
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs
## 3 Heikkinen, Miss. Laina    Miss

```

The [, 2] pulls out just the second column from the matrix returned by `str_match`.

Here are some of the most common titles:

```

titanic_train %>%
  mutate(title =
    str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[, 2]) %>%
  group_by(title) %>% summarize(n = n()) %>%
  arrange(desc(n)) %>% slice(1:5)

## # A tibble: 5 x 2
##   title     n
##   <chr>  <int>
## 1 Mr        517
## 2 Miss      182
## 3 Mrs       125
## 4 Master     40
## 5 Dr         7

```

Here are a few other examples of regular expressions in action with this dataset.

Get just names that start with (“`^`”) the letter “A”:

```

titanic_train %>%
  filter(str_detect(Name, "^\u00c5")) %>%
  select(Name) %>%
  slice(1:3)

```

```

##                                     Name
## 1 Allen, Mr. William Henry
## 2 Andersson, Mr. Anders Johan
## 3 Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)

```

Get names with “II” or “III” (`{2,}` says to match at least two times):

```
titanic_train %>%
  filter(str_detect(Name, "I{2,}")) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                                     Name
## 1 Carter, Master. William Thornton II
## 2 Roebling, Mr. Washington Augustus II
```

Get names with “Andersen” or “Anderson” (alternatives in square brackets):

```
titanic_train %>%
  filter(str_detect(Name, "Anders[eo]n")) %>%
  select(Name)
```

```
##                                     Name
## 1 Andersen-Jensen, Miss. Carla Christine Nielsine
## 2                               Anderson, Mr. Harry
## 3                           Walker, Mr. William Anderson
## 4                         Olsvigen, Mr. Thor Anderson
## 5      Soholt, Mr. Peter Andreas Lauritz Andersen
```

Get names that start with (“^” outside of brackets) the letters “A” and “B”:

```
titanic_train %>%
  filter(str_detect(Name, "^ [AB]")) %>%
  select(Name) %>%
  slice(1:3)
```

```
##                                     Name
## 1 Braund, Mr. Owen Harris
## 2 Allen, Mr. William Henry
## 3 Bonnell, Miss. Elizabeth
```

Get names that end with (“\$”) the letter “b” (either lowercase or uppercase):

```
titanic_train %>%
  filter(str_detect(Name, "[bB]$")) %>%
  select(Name)
```

```

##                               Name
## 1   Emir, Mr. Farred Chehab
## 2 Goldschmidt, Mr. George B
## 3          Cook, Mr. Jacob
## 4        Pasic, Mr. Jakob

```

Some useful regular expression operators include:

| Operator | Meaning |
|----------|------------------------------------|
| . | Any character |
| * | Match 0 or more times (greedy) |
| *? | Match 0 or more times (non-greedy) |
| + | Match 1 or more times (greedy) |
| +? | Match 1 or more times (non-greedy) |
| ^ | Starts with (in brackets, negates) |
| \$ | Ends with |
| [...] | Character classes |

For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Chapter 14 of R For Data Science
- <http://gskinner.com/RegExr>: Interactive tool for helping you build regular expression pattern strings

6.6 Tidy select

Download a pdf of the lecture slides for this video.

There are `tidyverse` functions to make selecting variables more straightforwards. You can call these functions as arguments of the `select` function to streamline variable selection. Examples include: `starts_with()`, `ends_with()`, and `contains()`.

Here we use `starts_with("t")` to select all variables that begin with t.

```

titanic_train %>%
  select(starts_with("t")) %>%
  slice(1:3)

```

```

##                               Ticket
## 1           A/5 21171
## 2           PC 17599
## 3 STON/O2. 3101282

```

The are also tidyverse functions that allow us to easily operate on a selection of variables. These functions are called **scoped variants**. You can identify these functions by these `_all`, `_at`, and `_if` suffixes.

Here we use `select_if` to select all the numeric variables in a dataframe and convert their names to lower case (a handy function to tidy the variable names).

```
titanic_train %>%
  select_if(is.numeric, tolower) %>%
  slice(1:3)

##   passengerid survived pcclass age sibsp parch     fare
## 1           1        0      3  22     1     0 7.2500
## 2           2        1      1  38     1     0 71.2833
## 3           3        1      3  26     0     0 7.9250
```

The `select_if` function takes the following form.

```
## Generic code
new_df <- select_if(old_df,
                     .predicate [selects the variable to keep],
                     .funs = [the function to apply to the selected columns])
```

Here we use `select_at` to select all the variables that contain `ss` in their name and then convert their names to lower case (a handy function to tidy the variable names).

```
titanic_train %>%
  select_at(vars(contains("ss")), tolower) %>%
  slice(1:3)

##   passengerid pcclass
## 1           1      3
## 2           2      1
## 3           3      3
```

6.7 In-course exercise Chapter 6

For today's exercise, we'll be using the following three datasets (click on the file name to access the correct file for today's class for each dataset):

| File name | Description |
|-------------------------------------|---|
| <code>country_timeseries.csv</code> | Ebola cases by country for the 2014 outbreak |
| <code>mexico_exposure.csv</code> | Daily death counts and environmental measurements and |
| <code>mexico_deaths.csv</code> | for Mexico City, Mexico, for 2008 |
| <code>measles_data/</code> | Number of cases of measles in CA since end of Dec. 2014 |

Note that you likely have already downloaded all the files in the `measles_data` folder, since we used them in an earlier in-course exercise. If so, there is no need to re-download those files.

Here are the sources for this data:

- `country_timeseries.csv` : Caitlin Rivers' Ebola repository (Caitlin originally collected this data from the WHO and WHO Situation reports)
- `mexico_exposure.csv` and `mexico_deaths.csv` : one of Hadley Wickham's GitHub repos (Hadley got the data originally from the Secretaría de Salud of Mexico's website, although it appears the link is now broken. I separated the data into two dataframes so students could practice merging.)
- `measles_data/`: one of scarpino's GitHub repos (Data originally from pdfs from the California Department of Public Health)



If you want to use these data further, you should go back and pull them from their original sources. They are here only for use in R code examples for this course.

Here are some of the packages you will need for this exercise:

```
library(dplyr)
library(gridExtra)
library(ggthemes)
```

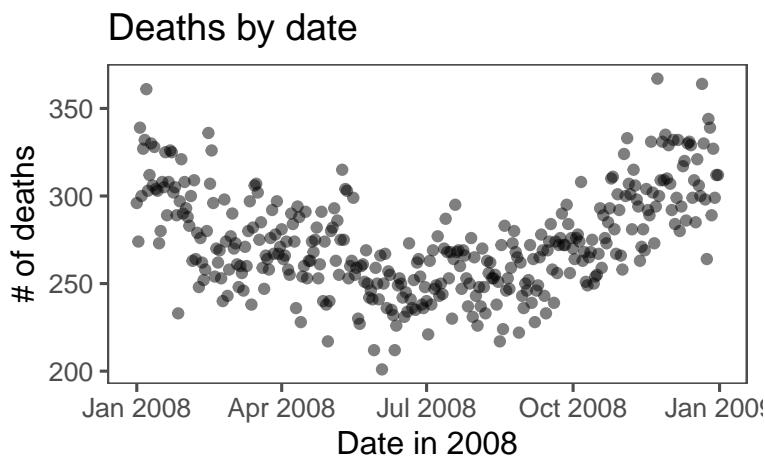
6.7.1 Designing tidy data

1. Check out the `country_timeseries.csv` file on Ebola for this week's example data. Talk with your partner and decide what changes you would need to make to this dataset to turn it into a "tidy" dataset, in particular which of the five common "untidy" problems the data currently has and why.

2. Do the same for the data on daily mortality and daily weather in Mexico.
3. Do the same for the set of files with measles data.

6.7.2 Easier data wrangling

- Use `read_csv` to read the Mexico data (exposure and mortality) directly from GitHub into your R session. Call the dataframes `mex_deaths` and `mex_exp`.
- Are there any values of the `day` column in `mex_deaths` that is not present in the `day` column of `mex_exp`? How about vice-versa? (Hint: There are a few ways you could check this. One is to try filtering down to just rows in one dataframe where the `day` values are not present in the `day` values from the other dataframe. The `%in%` logical vector may be useful.)
- Merge the two datasets together to create the dataframe `mexico`. Exclude all columns except the outcome (deaths), day, and mean temperature.
- Convert the day to a Date class.
- If you did not already, try combining all the steps in the previous task into one “chained” pipeline of code using the pipe operator, `%>%`.
- Use this new dataframe to plot deaths by date in Mexico using `ggplot2`. The final plot should look like this:



6.7.2.1 Example R code

Use `read_csv` to read the `mexico` data (exposure and mortality) directly from GitHub into your R session. Call the dataframes `mex_deaths` and `mex_exp`:

```
deaths_url <- paste0("https://github.com/geanders/RProgrammingForResearch/",
                      "raw/master/data/mexico_deaths.csv")
mex_deaths <- read_csv(deaths_url)
head(mex_deaths)
```

```
## # A tibble: 6 x 2
##   day     deaths
##   <chr>   <dbl>
## 1 1/1/08    296
## 2 1/2/08    274
## 3 1/3/08    339
## 4 1/4/08    300
## 5 1/5/08    327
## 6 1/6/08    332
```

```
exposure_url <- paste0("https://github.com/geanders/RProgrammingForResearch/",
                       "raw/master/data/mexico_exposure.csv")
mex_exp <- read_csv(exposure_url)
head(mex_exp)
```

```
## # A tibble: 6 x 14
##   day   temp_min temp_max temp_mean humidity   wind      NO     NO2     NOX     O3
##   <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 1/1/~     7.8    17.8    11.8    53.5  2.66  0.00925 0.0187  0.0278 NA
## 2 1/2/~     2.6     9.8    6.64    61.7  3.35  0.00542 0.0187  0.0241  0.0201
## 3 1/3/~     1.1    15.6    7.04    59.9  1.89  0.0160  0.0381  0.0540  0.0184
## 4 1/4/~     3.1    20.6    10.9    57.5  1.20  0.0408  0.0584  0.0993  0.0215
## 5 1/5/~     6       21.3    13.4    45.7  0.988  0.0469  0.0602  0.107   0.0239
## 6 1/6/~     7.2    22.1    14.3    40.8  0.854  0.0286  0.051   0.0795  0.0249
## # i 4 more variables: CO <dbl>, S02 <dbl>, PM10 <dbl>, PM25 <dbl>
```

Check if there are any values of the day column in `mex_deaths` that are not present in the `day` column of `mex_exp` and vice-versa.

```
mex_deaths %>%
  filter(!(day %in% mex_exp$day))
```

```
## # A tibble: 0 x 2
## # i 2 variables: day <chr>, deaths <dbl>
```

```
mex_exp %>%
  filter(!(day %in% mex_deaths$day))

## # A tibble: 0 x 14
## # i 14 variables: day <chr>, temp_min <dbl>, temp_max <dbl>, temp_mean <dbl>,
## #   humidity <dbl>, wind <dbl>, NO <dbl>, NO2 <dbl>, NOX <dbl>, O3 <dbl>,
## #   CO <dbl>, SO2 <dbl>, PM10 <dbl>, PM25 <dbl>
```

One important note is that, when you're doing this check, you do *not* want to overwrite your original dataframe, so be sure that you do not reassign this output to `mex_deaths` or `mex_exp`.

An even quicker way to do check this is to create a logical vector that checks this and use `sum` to add up the values in the logical vector. If the sum is zero, that tells you that the logical check is never true, so there are no cases where there is a `day` value in one dataframe that is not also in the other dataframe.

```
sum(!(mex_deaths$day %in% mex_exp$day))
```

```
## [1] 0
```

```
sum(!(mex_exp$day %in% mex_deaths$day))
```

```
## [1] 0
```

Merge the two datasets together to create the dataframe `mexico`. Exclude all columns except the outcome (deaths), date, and mean temperature.

```
mexico <- full_join(mex_deaths, mex_exp, by = "day")
mexico <- select(mexico, day, deaths, temp_mean)
```

Convert the date to a date class.

```
library(lubridate) ## For parsing dates
mexico <- mutate(mexico, day = mdy(day))
```

Try combining all the steps in the previous task into one “chained” command:

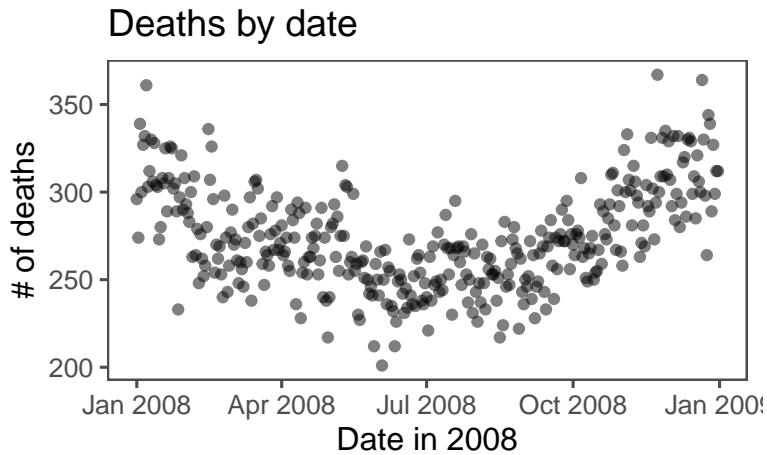
```
mexico <- full_join(mex_deaths, mex_exp, by = "day") %>%
  select(day, deaths, temp_mean) %>%
  mutate(day = mdy(day))
head(mexico)
```

```
## # A tibble: 6 x 3
##   day      deaths temp_mean
##   <date>    <dbl>     <dbl>
## 1 2008-01-01    296     11.8
## 2 2008-01-02    274      6.64
## 3 2008-01-03    339      7.04
## 4 2008-01-04    300     10.9
## 5 2008-01-05    327     13.4
## 6 2008-01-06    332     14.3
```

Note that, in this case, all the values of `day` in `mex_deaths` have one and only one matching value in `mex_exp`, and vice-versa. Because of this, we would have gotten the same `mexico` dataframe if we’d used `inner_join`, `left_join` or `right_join` instead of `full_join`. The differences between these `*_join` functions come into play when you have some values of your matching column that aren’t in both of the dataframes you’re joining.

Use this new dataframe to plot deaths by date using `ggplot`:

```
ggplot(data = mexico) +
  geom_point(mapping = aes(x = day, y = deaths),
             size = 1.5, alpha = 0.5) +
  labs(x = "Date in 2008", y = "# of deaths") +
  ggtitle("Deaths by date") +
  theme_few()
```



6.7.3 More extensive data wrangling

- Read the Ebola data directly from GitHub into your R session. Call the dataframe `ebola`.
- Use `dplyr` functions to create a tidy dataset. First, change it from “wide” data to “long” data. Name the new column with the key `variable` and the new column with the values `count`. The first few lines of the “long” version of the dataset should look like this:

```
## # A tibble: 6 x 4
##   Date      Day variable     count
##   <chr>    <dbl> <chr>      <dbl>
## 1 1/5/2015 289 Cases_Guinea  2776
## 2 1/5/2015 289 Cases_Liberia NA
## 3 1/5/2015 289 Cases_SierraLeone 10030
## 4 1/5/2015 289 Cases_Nigeria  NA
## 5 1/5/2015 289 Cases_Senegal  NA
## 6 1/5/2015 289 Cases_UnitedStates NA
```

- Convert the `Date` column to a Date class.
- Use the `separate` function to separate the `variable` column into two columns, `type` (“Cases” or “Deaths”) and `country` (“Guinea”, “Liberia”, etc.). At this point, the data should look like this:

```
## # A tibble: 6 x 5
##   Date      Day type country     count
##   <date>    <dbl> <chr> <chr>      <dbl>
## 1 2015-01-05 289 Cases Guinea  2776
```

```

## 2 2015-01-05    289 Cases Liberia      NA
## 3 2015-01-05    289 Cases SierraLeone 10030
## 4 2015-01-05    289 Cases Nigeria     NA
## 5 2015-01-05    289 Cases Senegal     NA
## 6 2015-01-05    289 Cases UnitedStates NA

```

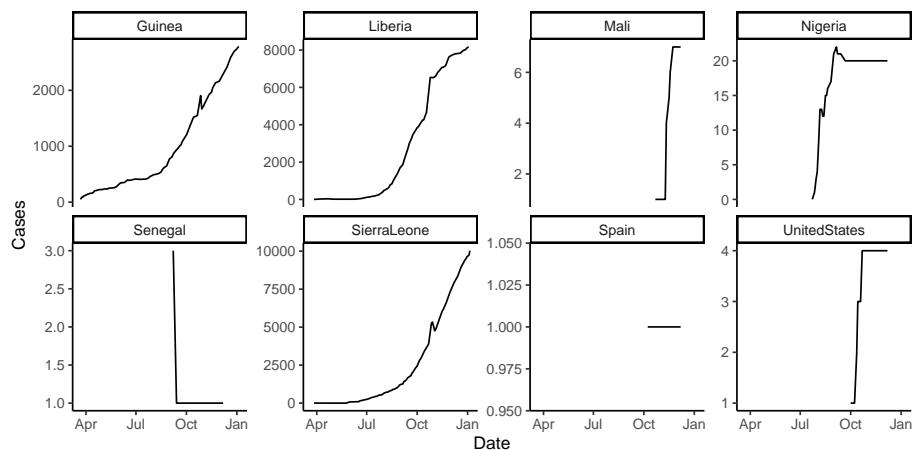
- Use the `pivot_wider` function to convert the data so you have separate columns for the two variables of numbers of `Cases` and `Deaths`. At this point, the dataframe should look like this:

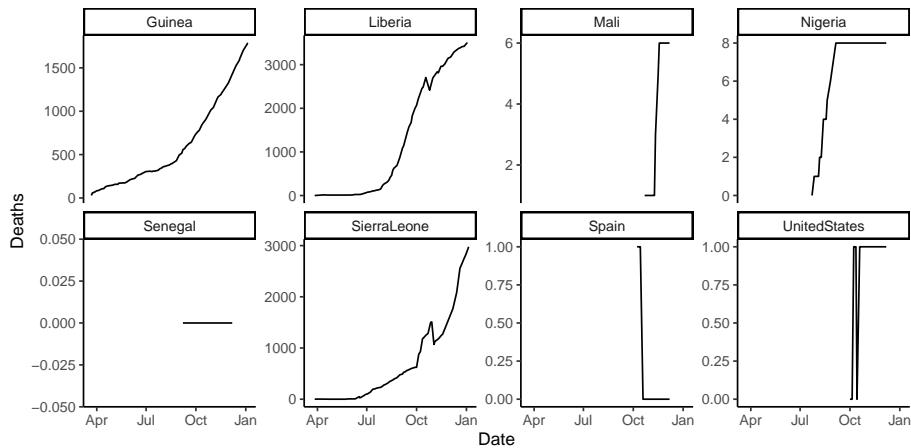
```

## # A tibble: 6 x 5
##   Date       Day country   Cases Deaths
## <date>     <dbl> <chr>     <dbl>  <dbl>
## 1 2015-01-05 289 Guinea    2776   1786
## 2 2015-01-05 289 Liberia   NA     NA
## 3 2015-01-05 289 SierraLeone 10030  2977
## 4 2015-01-05 289 Nigeria   NA     NA
## 5 2015-01-05 289 Senegal   NA     NA
## 6 2015-01-05 289 UnitedStates NA     NA

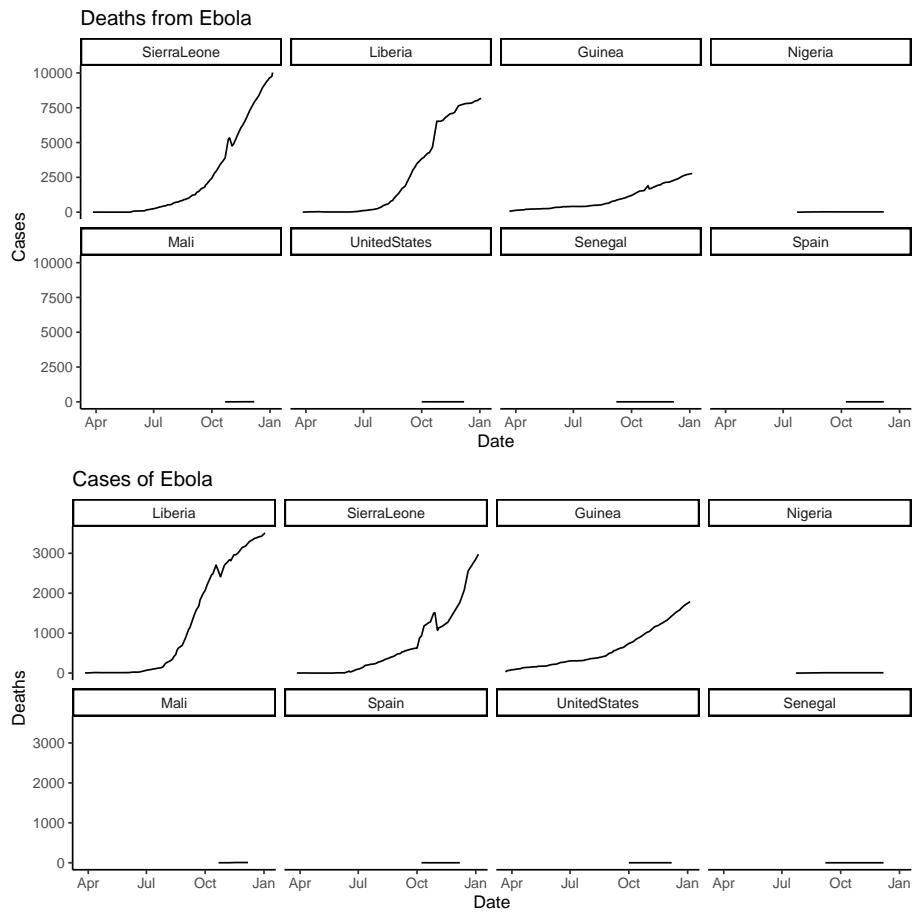
```

- Remove any observations where counts of both cases and deaths are missing for that country on that date.
- Now that your data is tidy, create one plot showing Ebola cases by date, faceted by country, and one showing Ebola deaths by date, also faceted by country. Try using the option `scales = "free_y"` in the `facet_wrap` function and see how that changes these graphs. Discuss with your group the advantages and disadvantages of using this option when creating these small multiple plots. The plots should look something like this (if you're using the `scales = "free_y"` option):





- Based on these plots, what would your next questions be about this data before you used it for an analysis?
- Can you put all of the steps of this cleaning process into just a few “chained” code pipelines using `%>%`?
- If you have extra time (super-challenge!): There is a function called `fct_reorder` in the `forcats` package that can be used to reorder the levels of a factor in a data frame based on another column in the same data frame. This function can be very useful for using a meaningful order when plotting. We’ll cover the `forcats` package in a later class, but today check out the help file for `fct_reorder` and see if you can figure out how to use it to reorder the small multiple plots in order of the maximum number of cases or deaths (for the two plots respectively) in each country. You’ll be able to do this by changing the code in `facet_wrap` from `~country` to `~ fct_reorder(country, ...)`, but with the `...` replaced with certain arguments. If you’re getting stuck, try running the examples in the `fct_reorder` helpfile to get a feel for how this function can be used when plotting. The plots will look something like this:



6.7.3.1 Example R code

Read the data in using `read_csv`.

```
ebola_url <- paste0("https://github.com/geanders/RProgrammingForResearch/",
                      "raw/master/data/country_timeseries.csv")
ebola <- read_csv(ebola_url)

head(ebola)
```

```
## # A tibble: 6 x 18
##   Date      Day Cases_Guinea Cases_Liberia Cases_SierraLeone Cases_Nigeria
##   <chr>     <dbl>    <dbl>        <dbl>        <dbl>        <dbl>
## 1 2014-04-01 0       0           0           0           0
## 2 2014-05-01 1       0           0           0           0
## 3 2014-06-01 2       0           0           0           0
## 4 2014-07-01 3       0           0           0           0
## 5 2014-08-01 4       0           0           0           0
## 6 2014-09-01 5       0           0           0           0
```

```

## 1 1/5/2015      289      2776       NA      10030       NA
## 2 1/4/2015      288      2775       NA      9780       NA
## 3 1/3/2015      287      2769     8166      9722       NA
## 4 1/2/2015      286       NA     8157       NA      9633       NA
## 5 12/31/2014    284      2730     8115      9633       NA
## 6 12/28/2014    281      2706     8018      9446       NA
## # i 12 more variables: Cases_Senegal <dbl>, Cases_UnitedStates <dbl>,
## #   Cases_Spain <dbl>, Cases_Mali <dbl>, Deaths_Guinea <dbl>,
## #   Deaths_Liberia <dbl>, Deaths_SierraLeone <dbl>, Deaths_Nigeria <dbl>,
## #   Deaths_Senegal <dbl>, Deaths_UnitedStates <dbl>, Deaths_Spain <dbl>,
## #   Deaths_Mali <dbl>

```

Change the data to long data using the `pivot_longer` function from `tidyverse`:

```

ebola <- ebola %>%
  pivot_longer(cols = c(-Date, -Day), names_to = "variable", values_to = "count")
head(ebola)

```

```

## # A tibble: 6 x 4
##   Date      Day variable     count
##   <chr>     <dbl> <chr>      <dbl>
## 1 1/5/2015   289 Cases_Guinea    2776
## 2 1/5/2015   289 Cases_Liberia     NA
## 3 1/5/2015   289 Cases_SierraLeone 10030
## 4 1/5/2015   289 Cases_Nigeria     NA
## 5 1/5/2015   289 Cases_Senegal     NA
## 6 1/5/2015   289 Cases_UnitedStates NA

```

Convert Date to a date class:

```

ebola <- ebola %>%
  mutate(Date = mdy(Date))
head(ebola)

```

```

## # A tibble: 6 x 4
##   Date      Day variable     count
##   <date>     <dbl> <chr>      <dbl>
## 1 2015-01-05 289 Cases_Guinea    2776
## 2 2015-01-05 289 Cases_Liberia     NA
## 3 2015-01-05 289 Cases_SierraLeone 10030
## 4 2015-01-05 289 Cases_Nigeria     NA
## 5 2015-01-05 289 Cases_Senegal     NA
## 6 2015-01-05 289 Cases_UnitedStates NA

```

Split variable into type and country:

```
ebola <- ebola %>%
  separate(variable, c("type", "country"), sep = "_")

head(ebola)
```

```
## # A tibble: 6 x 5
##   Date      Day type  country    count
##   <date>    <dbl> <chr> <chr>     <dbl>
## 1 2015-01-05  289 Cases Guinea     2776
## 2 2015-01-05  289 Cases Liberia    NA
## 3 2015-01-05  289 Cases SierraLeone 10030
## 4 2015-01-05  289 Cases Nigeria    NA
## 5 2015-01-05  289 Cases Senegal    NA
## 6 2015-01-05  289 Cases UnitedStates NA
```

Convert the data so you have separate columns for the two variables of numbers of Cases and Deaths:

```
ebola <- pivot_wider(ebola, names_from = type, values_from = count)
head(ebola)
```

```
## # A tibble: 6 x 5
##   Date      Day country    Cases Deaths
##   <date>    <dbl> <chr>     <dbl>  <dbl>
## 1 2015-01-05  289 Guinea     2776   1786
## 2 2015-01-05  289 Liberia    NA     NA
## 3 2015-01-05  289 SierraLeone 10030  2977
## 4 2015-01-05  289 Nigeria    NA     NA
## 5 2015-01-05  289 Senegal    NA     NA
## 6 2015-01-05  289 UnitedStates NA     NA
```

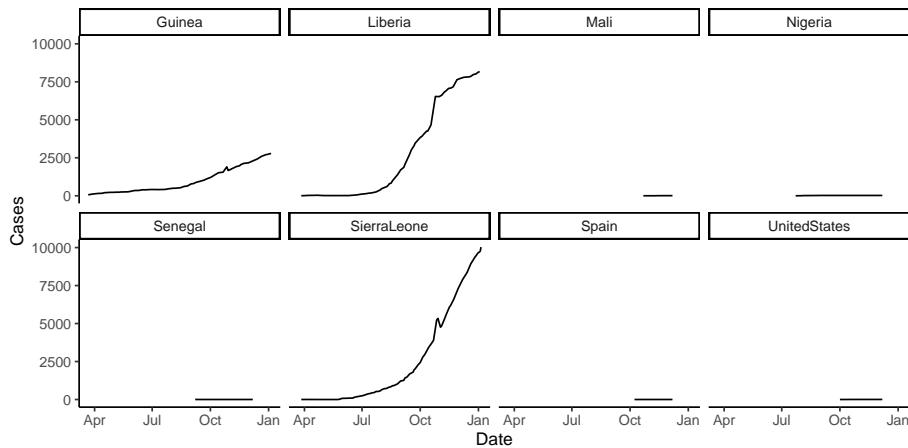
Remove any observations where counts of cases or deaths are missing for that country:

```
ebola <- filter(ebola, !is.na(Cases) & !is.na(Deaths))
head(ebola)
```

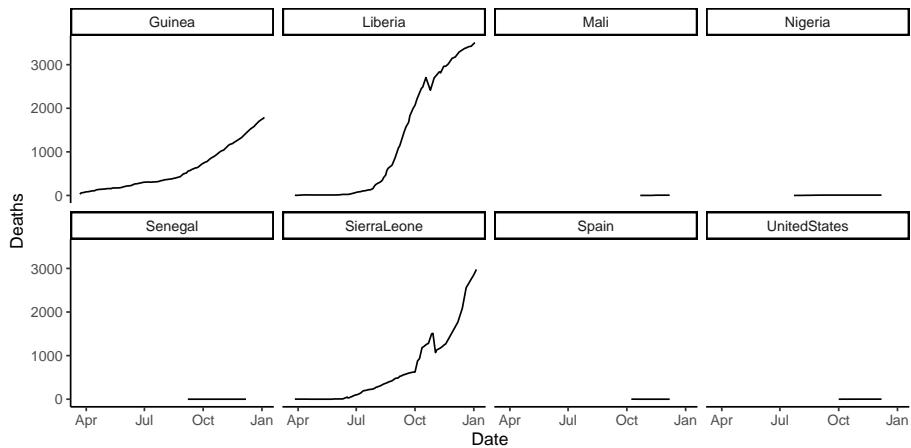
```
## # A tibble: 6 x 5
##   Date      Day country    Cases Deaths
## <date>    <dbl> <chr>     <dbl>  <dbl>
## 1 2015-01-05 289 Guinea     2776  1786
## 2 2015-01-05 289 SierraLeone 10030  2977
## 3 2015-01-04 288 Guinea     2775  1781
## 4 2015-01-04 288 SierraLeone  9780  2943
## 5 2015-01-03 287 Guinea     2769  1767
## 6 2015-01-03 287 Liberia    8166  3496
```

Now that your data is tidy, create one plot showing ebola cases by date, faceted by country, and one showing ebola deaths by date, also faceted by country:

```
ggplot(ebola, aes(x = Date, y = Cases)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4) +
  theme_classic()
```

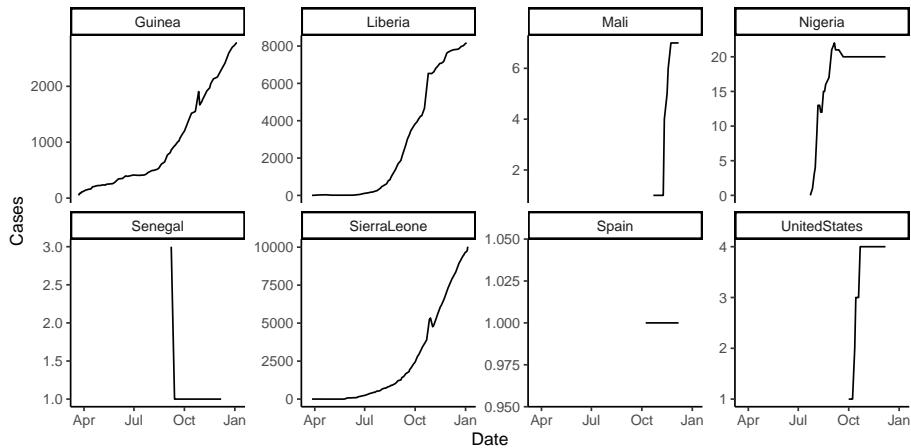


```
ggplot(ebola, aes(x = Date, y = Deaths)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4) +
  theme_classic()
```

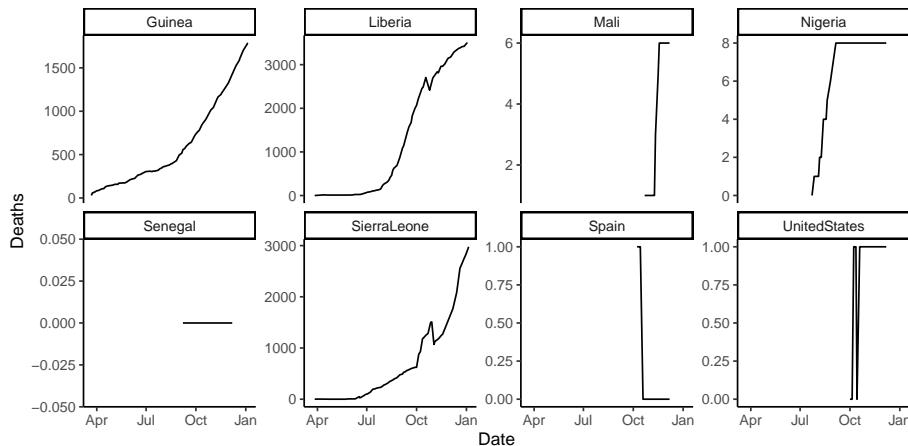


Try using the option `scales = "free_y"` in the `facet_wrap()` function (in the `gridExtra` package) and see how that changes these graphs:

```
ggplot(ebola, aes(x = Date, y = Cases)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4, scales = "free_y") +
  theme_classic()
```



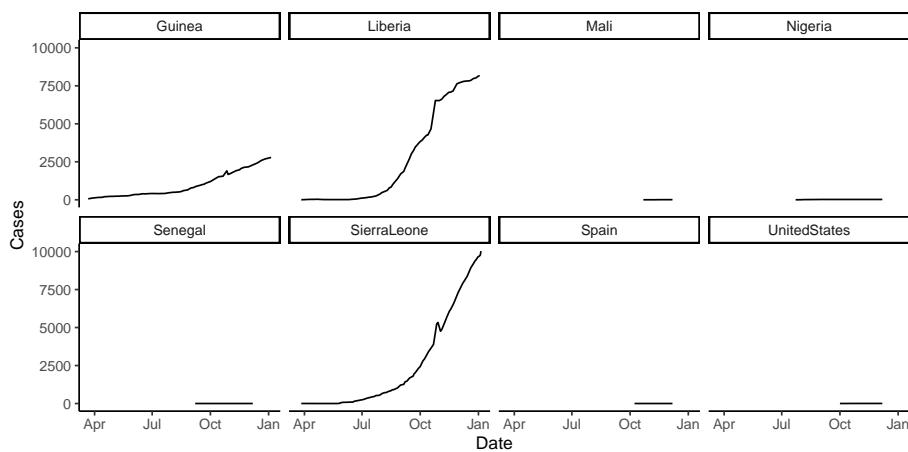
```
ggplot(ebola, aes(x = Date, y = Deaths)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4, scales = "free_y") +
  theme_classic()
```



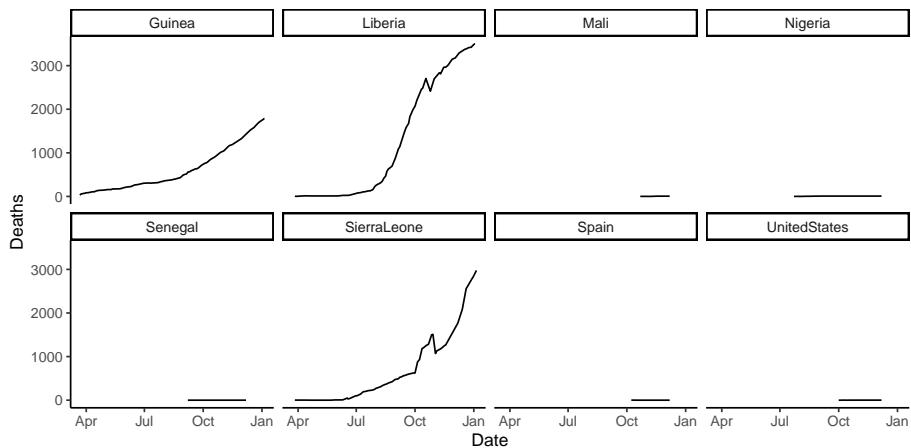
Put all of the steps of this cleaning process into just a few “chaining” calls.

```
ebola <- read_csv(ebola_url) %>%
  pivot_longer(cols = c(-Date, -Day), names_to = "variable", values_to = "count") %>%
  mutate(Date = mdy(Date)) %>%
  separate(variable, c("type", "country"), sep = "_") %>%
  pivot_wider(names_from = type, values_from = count) %>%
  filter(!is.na(Cases) & !is.na(Deaths))

ggplot(ebola, aes(x = Date, y = Cases)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4) +
  theme_classic()
```

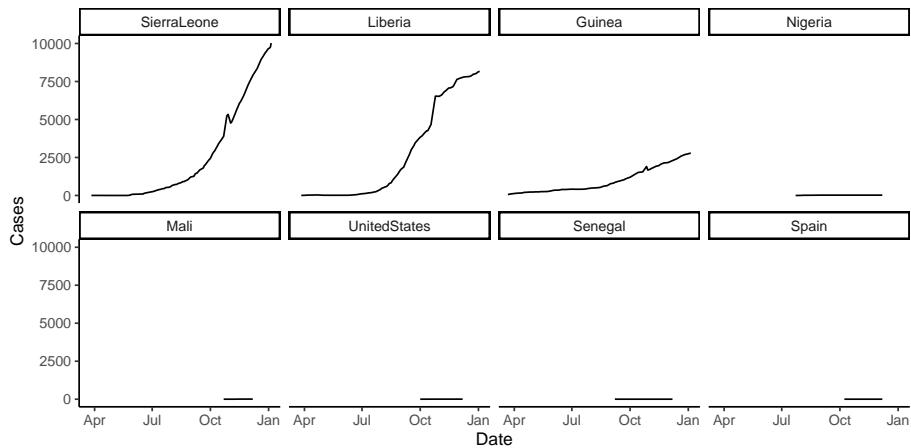


```
ggplot(ebola, aes(x = Date, y = Deaths)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4) +
  theme_classic()
```

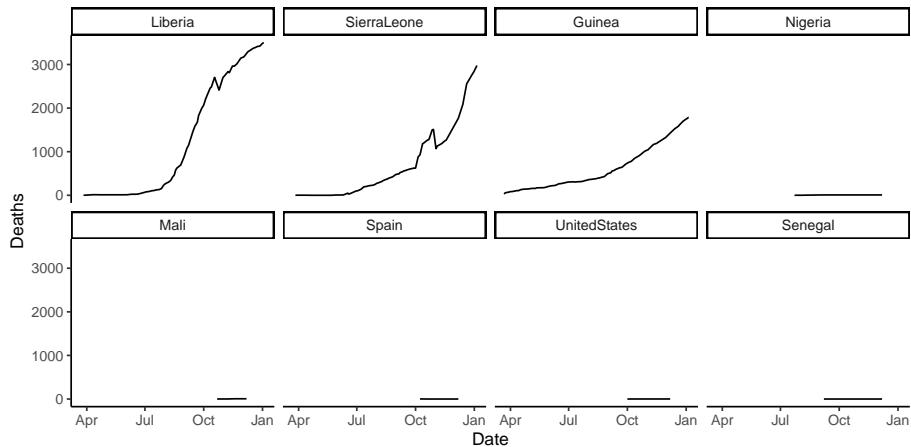


Use the `fct_reorder` function inside the `facet_wrap` function call to reorder the small-multiple graphs.

```
library(forcats)
ggplot(ebola, aes(x = Date, y = Cases)) +
  geom_line() +
  facet_wrap(~ fct_reorder(country, Cases, .fun = max, .desc = TRUE),
             ncol = 4) +
  theme_classic()
```



```
ggplot(ebola, aes(x = Date, y = Deaths)) +
  geom_line() +
  facet_wrap(~ fct_reorder(country, Deaths, .fun = max, .desc = TRUE),
             ncol = 4) +
  theme_classic()
```

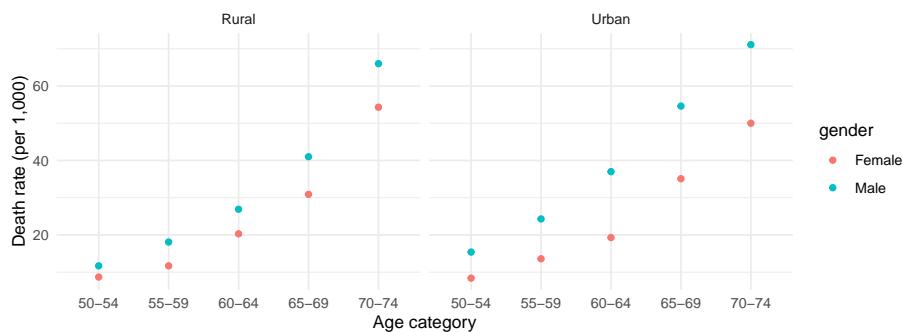


6.7.4 Tidying VADeaths data

R comes with a dataset called **VADeaths** that gives death rates per 1,000 people in Virginia in 1940 by age, sex, and rural / urban.

- Use `data("VADeaths")` to load this data. Make sure you understand what each column and row is showing – use the helpfile (`?VADeaths`) if you need.

- Go through the three characteristics of tidy data and the five common problems in untidy data that we talked about in class. Sketch out (you’re welcome to use the whiteboards) what a tidy version of this data would look like.
- Open a new R script file. Write R code to transform this dataset into a tidy dataset. Try using a pipe chain, with `%>%` and tidyverse functions, to clean the data.
- Use the tidy data to create the following graph:



There is no example R code for this – try to figure out the code yourselves. I go over a solution in the following video. You may find the RStudio Data Wrangling cheatsheet helpful for remembering which tidyverse functions do what.

Download a pdf of the lecture slides for this video.

6.7.5 Baby names

In the Fall 2018 session, we seem to have an unusually high percent of the class with names that start with an “A” or “K”. In this part of the exercise, we’ll see if we can figure out whether the proportion of “A” and “K” names is unusual.

There is a package on CRAN called `babynames` with data on baby names by year in the United States, based on data from the U.S.’s Social Security Administration. We will use this data to compare the proportion of “A” and “K” names in our class with the proportion in these baby names. We’ll also do a few other things to explore this data.

- First, check out patterns in your own name. Is your name included in this dataset? Has your name been used for males and females? How have the patterns in the proportion of babies with your name, for both males and females, changed over time (use a plot to look at this)?
- In the year you were born, what were the 5 most popular baby names for males and females? Try to come up with some attractive ways (figures and tables) to show this.

6.7.5.1 Example R code

Install and load the `babynames` package and its “babynames” dataframe:

```
# install.packages("babynames")
library(babynames)
data("babynames")
```

Remember that you can use `?babynames` to find out more about this dataframe.

Check out patterns in your own name. Is your name included in this dataset? You can use `filter` to create a subset of this data where you’ve filtered down to just the rows with your name. To see if your name ever shows up, you can use `count` on this dataframe—if your name never shows up, then you will have 0 rows in the new dataframe. As long as there’s at least one row, your name shows up somewhere. (If your name is not in here, try your middle or last name, or the name of a fictional character you like, for the rest of these exercises.)

```
my_name <- babynames %>%
  filter(name == "Brooke")
my_name %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    174
```

Has your name been used for males and females? To figure this out, you can group the dataset with rows with your name by the `sex` column and then use `count` to count the number of rows in the dataset for males and females. If your name has only been used for one gender, then only one row will result from running this code (for an example, try my first name, “Georgiana”).

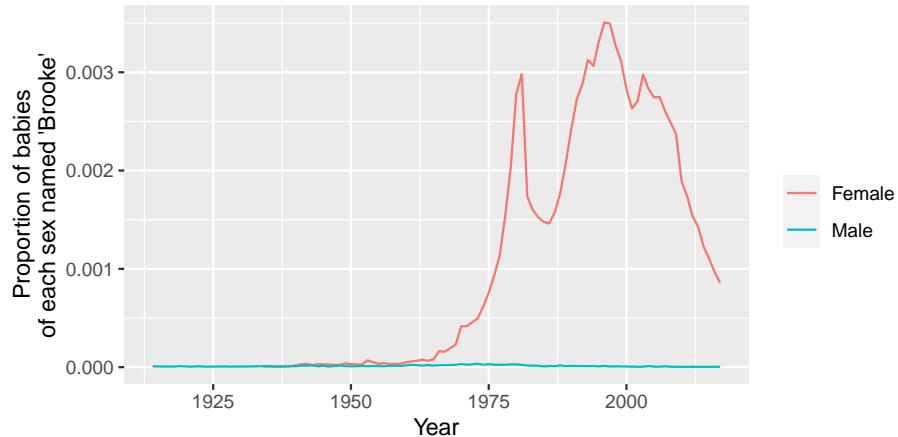
```
my_name %>%
  group_by(sex) %>%
  count()
```

```
## # A tibble: 2 x 2
##   # Groups:   sex [2]
```

```
##   sex      n
##   <chr> <int>
## 1 F       82
## 2 M       92
```

How have the patterns in the proportion of babies with your name, for both males and females, changed over time (use a plot to look at this)? To check this out, I recommend you create a plot of the proportion of babies with your name (`prop`) versus year (`year`). You can use color to show these patterns for males and females separately. I've done some extra things here to (1) relabel the `sex` factor, so the label shows up with clearer names and (2) change the labels for the x-, y-, and color scales.

```
library(forcats)
my_name %>%
  mutate(sex = fct_recode(sex, Male = "M", Female = "F")) %>%
  ggplot(aes(x = year, y = prop, color = sex)) +
  geom_line() +
  labs(x = "Year", y = "Proportion of babies\nof each sex named 'Brooke'", color = "")
```



In the year you were born, what were the 5 most popular baby names for males and females?

```
top_my_year <- babynames %>%
  filter(year == 1981) %>%
  group_by(sex) %>%
```

```
arrange(desc(prop)) %>%
  slice(1:5)

top_my_year
```

```
## # A tibble: 10 x 5
## # Groups:   sex [2]
##   year sex   name      n    prop
##   <dbl> <chr> <chr> <int>  <dbl>
## 1 1981 F   Jennifer 57049 0.0319
## 2 1981 F   Jessica  42532 0.0238
## 3 1981 F   Amanda   34374 0.0192
## 4 1981 F   Sarah    28173 0.0158
## 5 1981 F   Melissa  28000 0.0157
## 6 1981 M   Michael  68765 0.0369
## 7 1981 M   Christopher 50233 0.0270
## 8 1981 M   Matthew  43330 0.0233
## 9 1981 M   Jason    41932 0.0225
## 10 1981 M  David    40659 0.0218
```

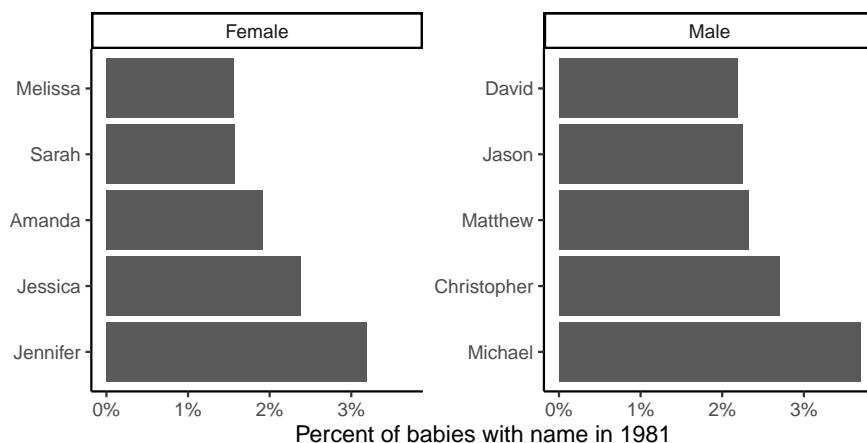
If you'd like to show this in a prettier way, you could show this as a table:

```
library(knitr)
top_my_year %>%
  mutate(rank = 1:n()) %>% # Since the data is grouped by sex, this will rank
                        # separately for females and males
  ungroup() %>% # You have to ungroup before you can run `mutate` on `sex`
  mutate(sex = fct_recode(sex, Male = "M", Female = "F"),
         percent = round(100 * prop, 1),
         percent = paste(percent, "%", sep = "")) %>%
  select(sex, rank, name, percent) %>%
  kable()
```

| sex | rank | name | percent |
|--------|------|-------------|---------|
| Female | 1 | Jennifer | 3.2% |
| Female | 2 | Jessica | 2.4% |
| Female | 3 | Amanda | 1.9% |
| Female | 4 | Sarah | 1.6% |
| Female | 5 | Melissa | 1.6% |
| Male | 1 | Michael | 3.7% |
| Male | 2 | Christopher | 2.7% |
| Male | 3 | Matthew | 2.3% |
| Male | 4 | Jason | 2.3% |
| Male | 5 | David | 2.2% |

You could also show it as a figure:

```
library(scales)
top_my_year %>%
  ungroup() %>%
  mutate(name = fct_reorder(name, prop, .desc = TRUE),
         sex = fct_recode(sex, Male = "M", Female = "F")) %>%
  ggplot(aes(x = name)) +
  geom_bar(aes(weight = prop)) +
  coord_flip() +
  labs(x = "", y = "Percent of babies with name in 1981") +
  theme(legend.position = "top") +
  scale_y_continuous(labels = percent) +
  facet_wrap(~ sex, scales = "free_y") +
  theme_classic()
```



Chapter 7

Exploring data #2

The video lectures for this chapter are embedded at relevant places in the text, with links to download a pdf of the associated slides for each video. You can also access a full playlist for the videos for this chapter.

7.1 Simple statistical tests in R

Download a pdf of the lecture slides for this video.

Download a pdf of the lecture slides for this video.

Let's pull the fatal accident data just for the county that includes Las Vegas, NV. Each US county has a unique identifier (FIPS code), composed of a two-digit state FIPS and a three-digit county FIPS code. The state FIPS for Nevada is 32; the county FIPS for Clark County is 003. Therefore, we can filter down to Clark County data in the FARS data we collected with the following code:

```
library(readr)
library(dplyr)
clark_co_accidents <- read_csv("data/accident.csv") %>%
  filter(STATE == 32 & COUNTY == 3)
```

We can also check the number of accidents:

```
clark_co_accidents %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    201
```

We want to test if the probability, on a Friday or Saturday, of a fatal accident occurring is higher than on other days of the week. Let's clean the data up a bit as a start:

```
library(tidyr)
library(lubridate)
clark_co_accidents <- clark_co_accidents %>%
  select(DAY, MONTH, YEAR) %>%
  unite(date, DAY, MONTH, YEAR, sep = "-") %>%
  mutate(date = dmy(date))
```

Here's what the data looks like now:

```
clark_co_accidents %>%
  slice(1:5)
```

```
## # A tibble: 5 x 1
##   date
##   <date>
## 1 2016-01-10
## 2 2016-02-21
## 3 2016-01-06
## 4 2016-01-13
## 5 2016-02-18
```

Next, let's get the count of accidents by date:

```
clark_co_accidents <- clark_co_accidents %>%
  group_by(date) %>%
  count() %>%
  ungroup()
clark_co_accidents %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##   date      n
##   <date>    <int>
## 1 2016-01-03     1
## 2 2016-01-06     1
## 3 2016-01-09     3
```

We're missing the dates without a fatal crash, so let's add those. First, create a dataframe with all dates in 2016:

```
all_dates <- tibble(date = seq(ymd("2016-01-01"),
                               ymd("2016-12-31"), by = 1))
all_dates %>%
  slice(1:5)
```

```
## # A tibble: 5 x 1
##   date
##   <date>
## 1 2016-01-01
## 2 2016-01-02
## 3 2016-01-03
## 4 2016-01-04
## 5 2016-01-05
```

Then merge this with the original dataset on Las Vegas fatal crashes and make any day missing from the fatal crashes dataset have a "0" for number of fatal accidents (n):

```
clark_co_accidents <- clark_co_accidents %>%
  right_join(all_dates, by = "date") %>%
  # If `n` is missing, set to 0. Otherwise keep value.
  mutate(n = ifelse(is.na(n), 0, n))
clark_co_accidents %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##   date      n
##   <date>    <dbl>
## 1 2016-01-03     1
## 2 2016-01-06     1
## 3 2016-01-09     3
```

Next, let's add some information about day of week and weekend:

```
clark_co_accidents <- clark_co_accidents %>%
  mutate(weekday = wday(date, label = TRUE),
        weekend = weekday %in% c("Fri", "Sat"))
clark_co_accidents %>%
  slice(1:3)
```

```
## # A tibble: 3 x 4
##   date      n weekday weekend
##   <date>    <dbl> <ord>   <lgl>
## 1 2016-01-03     1 Sun     FALSE
## 2 2016-01-06     1 Wed     FALSE
## 3 2016-01-09     3 Sat     TRUE
```

Now let's calculate the probability that a day has at least one fatal crash, separately for weekends and weekdays:

```
clark_co_accidents <- clark_co_accidents %>%
  mutate(any_crash = n > 0)
crash_prob <- clark_co_accidents %>%
  group_by(weekend) %>%
  summarize(n_days = n(),
            crash_days = sum(any_crash)) %>%
  mutate(prob_crash_day = crash_days / n_days)
crash_prob
```

```
## # A tibble: 2 x 4
##   weekend n_days crash_days prob_crash_day
##   <lgl>    <int>     <int>          <dbl>
## 1 FALSE      260       107         0.412
## 2 TRUE       106        43         0.406
```

In R, you can use `prop.test` to test if two proportions are equal. Inputs include the total number of trials in each group (`n` =) and the number of “successes” (`x` =):

```
prop.test(x = crash_prob$crash_days,
          n = crash_prob$n_days)
```

```

## 
## 2-sample test for equality of proportions with continuity correction
##
## data: crash_prob$crash_days out of crash_prob$n_days
## X-squared = 1.5978e-30, df = 1, p-value = 1
## alternative hypothesis: two.sided
## 95 percent confidence interval:
## -0.1109757 0.1227318
## sample estimates:
## prop 1 prop 2
## 0.4115385 0.4056604

```

I won't be teaching in this course how to find the correct statistical test. That's something you'll hopefully learn in a statistics course. There are also a variety of books that can help you with this, including some that you can access free online through CSU's library. One servicable introduction is "Statistical Analysis with R for Dummies".

You can create an object from the output of any statistical test in R. Typically, this will be (at least at some level) in an object class called a "list":

```

vegas_test <- prop.test(x = crash_prob$crash_days,
                         n = crash_prob$n_days)
is.list(vegas_test)

```

```
## [1] TRUE
```

So far, we've mostly worked with two object types in R, **dataframes** and **vectors**. In the next subsection we'll look more at two object classes we haven't looked at much, **matrices** and **lists**. Both have important roles once you start applying more advanced methods to analyze your data.

Download a pdf of the lecture slides for this video.

7.2 Matrices

A matrix is like a data frame, but all the values in all columns must be of the same class (e.g., numeric, character). R uses matrices a lot for its underlying math (e.g., for the linear algebra operations required for fitting regression models). R can do matrix operations quite quickly.

You can create a matrix with the **matrix** function. Input a vector with the values to fill the matrix and **ncol** to set the number of columns:

```
foo <- matrix(1:10, ncol = 5)
foo
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

By default, the matrix will fill up by column. You can fill it by row with the `byrow` function:

```
foo <- matrix(1:10, ncol = 5, byrow = TRUE)
foo
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

In certain situations, you might want to work with a matrix instead of a data frame (for example, in cases where you were concerned about speed – a matrix is more memory efficient than the corresponding data frame). If you want to convert a data frame to a matrix, you can use the `as.matrix` function:

```
foo <- tibble(col_1 = 1:2, col_2 = 3:4,
              col_3 = 5:6, col_4 = 7:8,
              col_5 = 9:10)
(foo <- as.matrix(foo))
```

```
##      col_1 col_2 col_3 col_4 col_5
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

You can index matrices with square brackets, just like data frames:

```
foo[1, 1:2]
```

```
## col_1 col_2
##      1    3
```

You cannot, however, use `dplyr` functions with matrices:

```
foo %>% filter(col_1 == 1)
```

All elements in a matrix must have the same class.

The matrix will default to make all values the most general class of any of the values, in any column. For example, if we replaced one numeric value with the character “a”, everything would turn into a character:

```
foo[1, 1] <- "a"
foo
```

```
##      col_1 col_2 col_3 col_4 col_5
## [1,] "a"   "3"   "5"   "7"   "9"
## [2,] "2"   "4"   "6"   "8"   "10"
```

7.3 Lists

A list has different elements, just like a data frame has different columns. However, the different elements of a list can have different lengths (unlike the columns of a data frame). The different elements can also have different classes.

```
bar <- list(some_letters = letters[1:3],
             some_numbers = 1:5,
             some_logical_values = c(TRUE, FALSE))
bar
```

```
## $some_letters
## [1] "a" "b" "c"
##
## $some_numbers
## [1] 1 2 3 4 5
##
## $some_logical_values
## [1] TRUE FALSE
```

To index an element from a list, use double square brackets. You can use bracket indexing either with numbers (which element in the list?) or with names:

```
bar[[1]]
```

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

```
bar[["some_numbers"]]
```

```
## [1] 1 2 3 4 5
```

You can also index lists with the `$` operator:

```
bar$some_logical_values
```

```
## [1] TRUE FALSE
```

To access a specific value within a list element we can index the element e.g.:

```
bar[[1]][[2]]
```

```
## [1] "b"
```

Lists can be used to contain data with an unusual structure and / or lots of different components. For example, the information from fitting a regression is often stored as a list:

```
my_mod <- glm(rnorm(10) ~ c(1:10))
is.list(my_mod)
```

```
## [1] TRUE
```

The `names` function returns the name of each element in the list:

```
head(names(my_mod), 3)

## [1] "coefficients"  "residuals"      "fitted.values"

my_mod[["coefficients"]]

## (Intercept)      c(1:10)
## 0.23562084 -0.06551959
```

A list can even contain other lists. We can use the `str` function to see the structure of a list:

```
a_list <- list(list("a", "b"), list(1, 2))

str(a_list)

## List of 2
## $ :List of 2
##   ..$ : chr "a"
##   ..$ : chr "b"
## $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
```

Sometimes you'll see unnecessary lists-of-lists, perhaps when importing data into R created. Or a list with multiple elements that you would like to combine. You can remove a level of hierarchy from a list using the `flatten` function from the `purrr` package:

```
library(purrr)
a_list

## [[1]]
## [[1]][[1]]
## [1] "a"
##
## [[1]][[2]]
```

```
## [1] "b"
##
##
## [[2]]
## [[2]][[1]]
## [1] 1
##
## [[2]][[2]]
## [1] 2
```

```
flatten(a_list)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] 2
```

Let's look at the list object from the statistical test we ran for Las Vegas:

```
str(vegas_test)
```

```
## List of 9
## $ statistic : Named num 1.6e-30
##   ..- attr(*, "names")= chr "X-squared"
## $ parameter : Named num 1
##   ..- attr(*, "names")= chr "df"
## $ p.value   : num 1
## $ estimate  : Named num [1:2] 0.412 0.406
##   ..- attr(*, "names")= chr [1:2] "prop 1" "prop 2"
## $ null.value: NULL
## $ conf.int  : num [1:2] -0.111 0.123
##   ..- attr(*, "conf.level")= num 0.95
## $ alternative: chr "two.sided"
## $ method    : chr "2-sample test for equality of proportions with continuity correction"
## $ data.name  : chr "crash_prob$crash_days out of crash_prob$n_days"
##   - attr(*, "class")= chr "htest"
```

Using `str` to print out the list's structure doesn't produce the easiest to digest output. We can use the `jsonedit` function from the `listviewer` package to create a widget in the `viewer` pane to more easily explore our list.

```
library(listviewer)
jsonedit(vegas_test)
```

We can pull out an element using the `$` notation:

```
vegas_test$p.value
```

```
## [1] 1
```

Or using the `[[` notation:

```
vegas_test[[4]]
```

```
##      prop 1      prop 2
## 0.4115385 0.4056604
```

You may have noticed, though, that this output is not a tidy dataframe. Ack! That means we can't use all the tidyverse tricks we've learned so far in the course! Fortunately, David Robinson noticed this problem and came up with a package called `broom` that can "tidy up" a lot of these kinds of objects.

The `broom` package has three main functions:

- `glance`: Return a one-row, tidy dataframe from a model or other R object
- `tidy`: Return a tidy dataframe from a model or other R object
- `augment`: "Augment" the dataframe you input to the statistical function

Here is the output for `tidy` for the `vegas_test` object (`augment` won't work for this type of object, and `glance` gives the same thing as `tidy`):

```
library(broom)
tidy(vegas_test)
```

```
## # A tibble: 1 x 9
##   estimate1 estimate2 statistic p.value parameter conf.low conf.high method
##       <dbl>      <dbl>     <dbl>    <dbl>      <dbl>      <dbl>     <dbl> <chr>
## 1     0.412     0.406  1.60e-30     1.00        1     -0.111     0.123 2-sample t-
## # i 1 more variable: alternative <chr>
```

7.4 Apply a test multiple times

Download a pdf of the lecture slides for this video.

[Download a pdf of the lecture slides for this video.](#)

Download a pdf of the lecture slides for this video.

Often, we don't want to just apply a statistical test to our entire data set.

Let's look at an example from the `microbiome` package.

```
library(microbiome)
data(peerj32)
print(names(peerj32))
```

```
## [1] "lipids"    "microbes"   "meta"       "phyloseq"
```

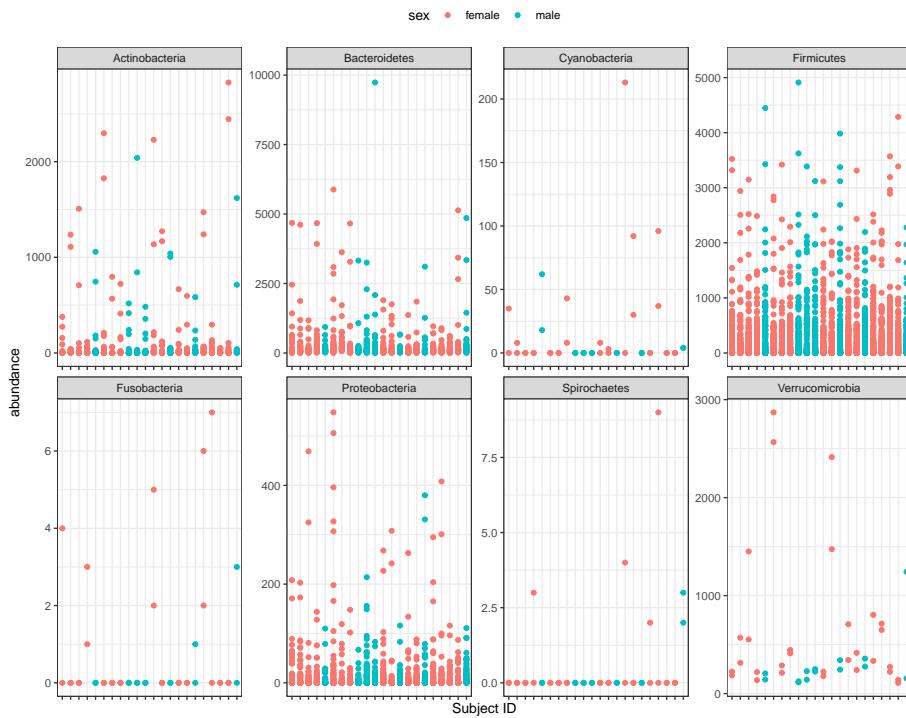
Like we saw before, the list-like `phyloseq` objects require a little tidying before we can use them easily.

```
library(dplyr)
peerj32_tibble <- (psmelt(peerj32$phyloseq)) %>%
  dplyr::select(-Sample) %>%
  rename_all(tolower) %>%
  as_tibble()
```

```
## # A tibble: 3 x 10
##   otu              abundance  time sex   subject sample group phylum family genus
##   <chr>            <dbl> <int> <fct> <fct>   <chr> <fct> <chr> <chr> <chr>
## 1 Bacteroides vu~     9734     1 male  S19     sampl~ LGG  Bacte~ Bacte~ Bact~
## 2 Bacteroides vu~     5883     1 fema~ S14     sampl~ Plac~ Bacte~ Bacte~ Bact~
## 3 Bacteroides vu~     5134     2 fema~ S8      sampl~ LGG  Bacte~ Bacte~ Bact~
```

We can make a plot from the resulting tibble to help us explore the data.

```
library(ggplot2)
ggplot(peerj32_tibble, aes(x = subject, y = abundance, color = sex)) +
  geom_point() +
  theme_bw() +
  facet_wrap(~phylum, ncol = 4, scales = "free") +
  theme(axis.text.x = element_blank(),
        legend.position = "top") +
  xlab("Subject ID")
```



We can group the data by `phylum` and `group` and then use the `nest` function from the `tidyverse` package to create a new dataframe containing the nested data:

```
library(tidyverse)
nested_data <- peerj32_tibble %>%
  group_by(phylum, group) %>%
  nest()
```

```
slice(nested_data, 1:3)
```

```
## # A tibble: 16 x 3
## # Groups:   phylum, group [16]
##   group   phylum      data
##   <fct>   <chr>       <list>
## 1 LGG     Actinobacteria <tibble [128 x 8]>
## 2 Placebo Actinobacteria <tibble [224 x 8]>
## 3 LGG     Bacteroidetes  <tibble [256 x 8]>
## 4 Placebo Bacteroidetes  <tibble [448 x 8]>
## 5 LGG     Cyanobacteria <tibble [16 x 8]>
## 6 Placebo Cyanobacteria <tibble [28 x 8]>
## 7 LGG     Firmicutes    <tibble [1,216 x 8]>
## 8 Placebo Firmicutes    <tibble [2,128 x 8]>
## 9 LGG     Fusobacteria  <tibble [16 x 8]>
## 10 Placebo Fusobacteria <tibble [28 x 8]>
## 11 LGG    Proteobacteria <tibble [416 x 8]>
## 12 Placebo Proteobacteria <tibble [728 x 8]>
## 13 LGG    Spirochaetes  <tibble [16 x 8]>
## 14 Placebo Spirochaetes <tibble [28 x 8]>
## 15 LGG    Verrucomicrobia <tibble [16 x 8]>
## 16 Placebo Verrucomicrobia <tibble [28 x 8]>
```

We can then use the `map` function from the `purrr` package to apply functions to each nested dataframe. Let's start by counting the rows in each nested dataframe and filtering out dataframes with less than 25 rows.

```
library(purrr)

filtered_data <- nested_data %>%
  mutate(n_rows = purrr::map(data, ~ nrow(.x))) %>%
  filter(n_rows > 25)
```

Now let's perform a t-test on each out the nested dataframes. Remember, each nested dataframe is one unique combination of phylum and group.

```
t_tests <- filtered_data %>%
  mutate(t_test = purrr::map(data, ~t.test(abundance ~ sex, data = .x)))
```

```
t_tests
```

```
## # A tibble: 12 x 5
## # Groups: phylum, group [12]
##   group    phylum      data      n_rows     t_test
##   <fct>   <chr>       <list>    <list>    <list>
## 1 LGG     Bacteroidetes <tibble [256 x 8]>  <int [1]> <htest>
## 2 Placebo Bacteroidetes <tibble [448 x 8]>  <int [1]> <htest>
## 3 Placebo Firmicutes   <tibble [2,128 x 8]> <int [1]> <htest>
## 4 LGG     Firmicutes   <tibble [1,216 x 8]> <int [1]> <htest>
## 5 Placebo Verrucomicrobia <tibble [28 x 8]>  <int [1]> <htest>
## 6 LGG     Actinobacteria <tibble [128 x 8]>  <int [1]> <htest>
## 7 Placebo Actinobacteria <tibble [224 x 8]>  <int [1]> <htest>
## 8 Placebo Proteobacteria <tibble [728 x 8]>  <int [1]> <htest>
## 9 LGG     Proteobacteria <tibble [416 x 8]>  <int [1]> <htest>
## 10 Placebo Cyanobacteria <tibble [28 x 8]>  <int [1]> <htest>
## 11 Placebo Fusobacteria <tibble [28 x 8]>  <int [1]> <htest>
## 12 Placebo Spirochaetes <tibble [28 x 8]>  <int [1]> <htest>
```

The resulting dataframe contains a new column, which contains the model objects. Just as we mapped the `t.test` function, we can map the *tidying* functions from the `broom` package to extract the information we want from the nested model objects.

```
summary <- t_tests %>%
  mutate(summary = purrr::map(t_test, broom::glance))
```

```
summary
```

```
## # A tibble: 12 x 6
## # Groups: phylum, group [12]
##   group    phylum      data      n_rows     t_test   summary
##   <fct>   <chr>       <list>    <list>    <list>   <list>
## 1 LGG     Bacteroidetes <tibble [256 x 8]>  <int [1]> <htest> <tibble>
## 2 Placebo Bacteroidetes <tibble [448 x 8]>  <int [1]> <htest> <tibble>
## 3 Placebo Firmicutes   <tibble [2,128 x 8]> <int [1]> <htest> <tibble>
## 4 LGG     Firmicutes   <tibble [1,216 x 8]> <int [1]> <htest> <tibble>
## 5 Placebo Verrucomicrobia <tibble [28 x 8]>  <int [1]> <htest> <tibble>
## 6 LGG     Actinobacteria <tibble [128 x 8]>  <int [1]> <htest> <tibble>
## 7 Placebo Actinobacteria <tibble [224 x 8]>  <int [1]> <htest> <tibble>
```

```
##  8 Placebo Proteobacteria <tibble [728 x 8]> <int [1]> <htest> <tibble>
##  9 LGG    Proteobacteria <tibble [416 x 8]> <int [1]> <htest> <tibble>
## 10 Placebo Cyanobacteria <tibble [28 x 8]> <int [1]> <htest> <tibble>
## 11 Placebo Fusobacteria <tibble [28 x 8]> <int [1]> <htest> <tibble>
## 12 Placebo Spirochaetes <tibble [28 x 8]> <int [1]> <htest> <tibble>
```

The final step is to return a tidy dataframe, we can do this using the `unnest` function from the `tidyverse` package. Note: it is also wise to `ungroup` after you are done operating on your grouped variables - you may run into problems if you forget your dataframe is grouped later on!

```
unnested <- summary %>%
  unnest(summary) %>%
  ungroup() %>%
  dplyr::select(group, phylum, p.value)
```

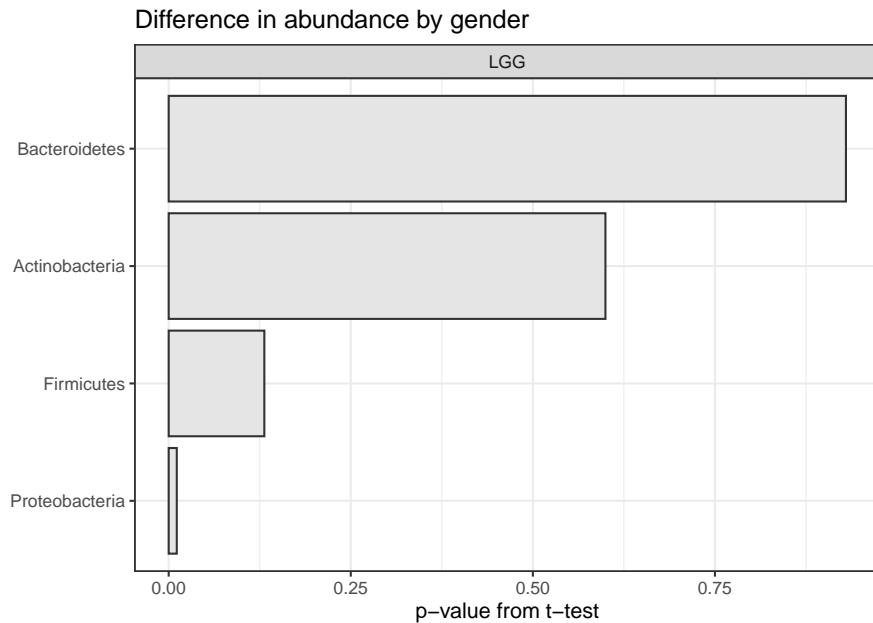
unnested

```
## # A tibble: 12 x 3
##   group     phylum      p.value
##   <fct>    <chr>       <dbl>
## 1 LGG      Bacteroidetes 0.930
## 2 Placebo  Bacteroidetes 0.828
## 3 Placebo  Firmicutes   0.344
## 4 LGG      Firmicutes   0.131
## 5 Placebo  Verrucomicrobia 0.106
## 6 LGG      Actinobacteria 0.600
## 7 Placebo  Actinobacteria 0.997
## 8 Placebo  Proteobacteria 0.856
## 9 LGG      Proteobacteria 0.0111
## 10 Placebo Cyanobacteria 0.193
## 11 Placebo Fusobacteria  0.265
## 12 Placebo Spirochaetes  0.728
```

We can then plot the results. Let's do this for the "LGG" (non-placebo) group, using our principles for good graphics.

```
p_data <- unnested %>%
  filter(group == "LGG") %>%
  mutate(phylum = fct_reorder(phylum, p.value))
```

```
ggplot(p_data, aes(y = p.value, x = phylum)) +
  facet_wrap(~group) +
  geom_bar(stat="identity", fill = "grey90", color = "grey20") +
  coord_flip() +
  theme_bw() +
  ggtitle("Difference in abundance by gender") +
  xlab("") + ylab("p-value from t-test")
```



7.5 Regression models

Download a pdf of the lecture slides for this video.

Download a pdf of the lecture slides for this video.

7.5.1 Formula structure

Regression models can be used to estimate how the expected value of a *dependent variable* changes as *independent variables* change.

In R, regression formulas take this structure:

```
## Generic code
[response variable] ~ [indep. var. 1] + [indep. var. 2] + ...
```

Notice that a tilde, `~`, is used to separate the independent and dependent variables and that a plus sign, `+`, is used to join independent variables. This format mimics the statistical notation:

$$Y_i \sim X_1 + X_2 + X_3$$

You will use this type of structure in R for a lot of different function calls, including those for linear models (fit with the `lm` function) and generalized linear models (fit with the `glm` function).

There are some conventions that can be used in R formulas. Common ones include:

| Convention | Meaning |
|------------------|---|
| <code>I()</code> | evaluate the formula inside <code>I()</code> before fitting (e.g., <code>I(x1 + x2)</code>) |
| <code>:</code> | fit the interaction between <code>x1</code> and <code>x2</code> variables |
| <code>*</code> | fit the main effects and interaction for both variables (e.g., <code>x1*x2</code> equals <code>x1 + x2 + x1:x2</code>) |
| <code>.</code> | include as independent variables all variables other than the response (e.g., <code>y ~ .</code>) |
| <code>1</code> | intercept (e.g., <code>y ~ 1</code> for an intercept-only model) |
| <code>-</code> | do not include a variable in the data frame as an independent variables (e.g., <code>y ~ . - x1</code>); usually used in conjunction with <code>.</code> or <code>1</code> |

7.5.2 Linear models

To fit a linear model, you can use the function `lm()`. This function is part of the `stats` package, which comes installed with base R. In this function, you can use the `data` option to specify the data frame from which to get the vectors.

```
## Warning: There was 1 warning in `mutate()`.
## i In argument: `sex = fct_recode(factor(sex), Male = "1", Female = "2")`.
## Caused by warning:
## ! Unknown levels in `f`: 1, 2
```

```
mod_a <- lm(wt ~ ht, data = nepali)
```

This previous call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where:

- Y_i : weight of child i
- $X_{1,i}$: height of child i

If you run the `lm` function without saving it as an object, R will fit the regression and print out the function call and the estimated model coefficients:

```
lm(wt ~ ht, data = nepali)
```

```
## 
## Call:
## lm(formula = wt ~ ht, data = nepali)
## 
## Coefficients:
## (Intercept)          ht
##           -8.7581      0.2342
```

However, to be able to use the model later for things like predictions and model assessments, you should save the output of the function as an R object:

```
mod_a <- lm(wt ~ ht, data = nepali)
```

This object has a special class, `lm`:

```
class(mod_a)
```

```
## [1] "lm"
```

This class is a special type of list object. If you use `is.list` to check, you can confirm that this object is a list:

```
is.list(mod_a)
```

```
## [1] TRUE
```

There are a number of functions that you can apply to an `lm` object. These include:

| Function | Description |
|---------------------------|---|
| <code>summary</code> | Get a variety of information on the model, including coefficients and p-values for the coefficients |
| <code>coefficients</code> | Pull out just the coefficients for a model |
| <code>fitted</code> | Get the fitted values from the model (for the data used to fit the model) |
| <code>plot</code> | Create plots to help assess model assumptions |
| <code>residuals</code> | Get the model residuals |

For example, you can get the coefficients from the model by running:

```
coefficients(mod_a)
```

```
## (Intercept)          ht
## -8.7581466   0.2341969
```

The estimated coefficient for the intercept is always given under the name “(Intercept)”. Estimated coefficients for independent variables are given based on their column names in the original data (“ht” here, for β_1 , or the estimated increase in expected weight for a one unit increase in height).

You can use the output from a `coefficients` call to plot a regression line based on the model fit on top of points showing the original data (Figure 7.1).

```
mod_coef <- coefficients(mod_a)
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point(size = 0.2) +
  xlab("Height (cm)") + ylab("Weight (kg)") +
  geom_abline(aes(intercept = mod_coef[1],
                  slope = mod_coef[2]), col = "blue")
```

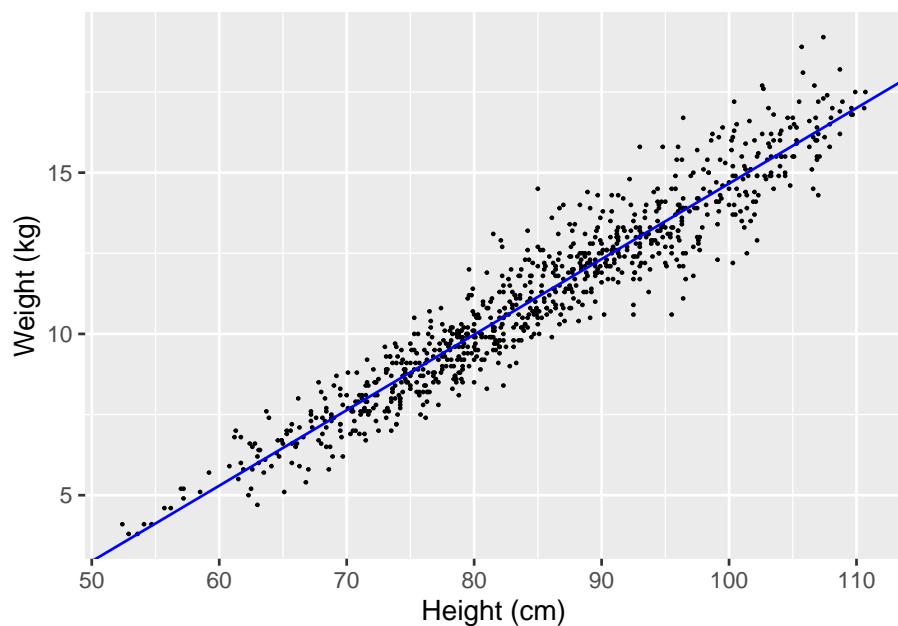


Figure 7.1: Example of using the output from a coefficients call to add a regression line to a scatterplot.



You can also add a linear regression line to a scatterplot by adding the geom `geom_smooth` using the argument `method = "lm"`.

You can use the function `residuals` on an `lm` object to pull out the residuals from the model fit:

```
head(residuals(mod_a))

##          1         2         3         4         6         7
##  0.1993922 -0.4329393 -0.4373953 -0.1355300 -0.6749080 -1.0838199
```

The result of a `residuals` call is a vector with one element for each of the non-missing observations (rows) in the data frame you used to fit the model. Each value gives the different between the model fitted value and the observed value for each of these observations, in the same order the observations show up in the data frame. The residuals are in the same order as the observations in the original data frame.



You can also use the shorter function `coef` as an alternative to `coefficients` and the shorter function `resid` as an alternative to `residuals`.

As noted in the subsection on simple statistics functions, the `summary` function returns different output depending on the type of object that is input to the function. If you input a regression model object to `summary`, the function gives you a lot of information about the model. For example, here is the output returned by running `summary` for the linear regression model object we just created:

```
summary(mod_a)

##
## Call:
## lm(formula = wt ~ ht, data = nepali)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.0077 -0.5479 -0.0293  0.4972  3.3514
```

```

## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -8.758147   0.211529 -41.40 <2e-16 ***
## ht          0.234197   0.002459  95.23 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.8733 on 875 degrees of freedom
##   (123 observations deleted due to missingness)
## Multiple R-squared:  0.912, Adjusted R-squared:  0.9119
## F-statistic:  9068 on 1 and 875 DF,  p-value: < 2.2e-16

```

This output includes a lot of useful elements, including (1) basic summary statistics for the residuals (to meet model assumptions, the median should be around zero and the absolute values fairly similar for the first and third quantiles), (2) coefficient estimates, standard errors, and p-values, and (3) some model summary statistics, including residual standard error, degrees of freedom, number of missing observations, and F-statistic.

The object returned by the `summary()` function when it is applied to an `lm` object is a list, which you can confirm using the `is.list` function:

```
is.list(summary(mod_a))
```

```
## [1] TRUE
```

With any list, you can use the `names` function to get the names of all of the different elements of the object:

```
names(summary(mod_a))
```

```

## [1] "call"         "terms"        "residuals"      "coefficients"
## [5] "aliased"      "sigma"         "df"            "r.squared"
## [9] "adj.r.squared" "fstatistic"    "cov.unscaled"  "na.action"

```

You can use the `$` operator to pull out any element of the list. For example, to pull out the table with information on the estimated model coefficients, you can run:

```
summary(mod_a)$coefficients
```

```
##             Estimate Std. Error t value Pr(>|t|) 
## (Intercept) -8.7581466 0.211529182 -41.40396 2.411051e-208
## ht           0.2341969 0.002459347  95.22726 0.000000e+00
```

The `plot` function, like the `summary` function, will give different output depending on the class of the object that you input. For an `lm` object, you can use the `plot` function to get a number of useful diagnostic plots that will help you check regression assumptions (Figure 7.2):

```
plot(mod_a)
```

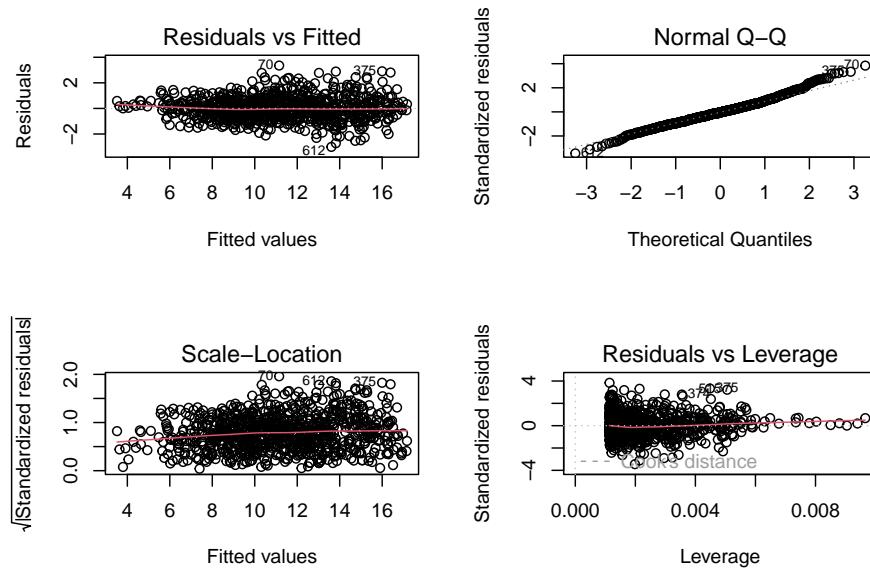


Figure 7.2: Example output from running the `plot` function with an `lm` object as the input.

You can also use binary variables or factors as independent variables in regression models. For example, in the `nepali` dataset, `sex` is a factor variable with the levels “Male” and “Female”. You can fit a linear model of weight regressed on sex for this data with the call:

```
mod_b <- lm(wt ~ sex, data = nepali)
```

This call fits the model:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \epsilon_i$$

where $X_{1,i}$: sex of child i , where 0 = male and 1 = female.

Here are the estimated coefficients from fitting this model:

```
summary(mod_b)$coefficients
```

```
##             Estimate Std. Error   t value Pr(>|t|)    
## (Intercept) 11.4226374  0.1375427 83.047920 0.00000000
## sexFemale   -0.4866184  0.1982810 -2.454185 0.01431429
```

You'll notice that, in addition to an estimated intercept ((`Intercept`)), the other estimated coefficient is `sexFemale` rather than just `sex`, although the column name in the data frame input to `lm` for this variable is `sex`.

This is because, when a factor or binary variable is input as an independent variable in a linear regression model, R will fit an estimated coefficient for all levels of factors *except* the first factor level. By default, this first factor level is used as the baseline level, and so its estimated mean is given by the estimated intercept, while the other model coefficients give the estimated *difference* from this baseline.

For example, the model fit above tells us that the estimated mean weight of males is 11.4, while the estimated mean weight of females is $11.4 + -0.5 = 10.9$.

7.6 Handling model objects

The `broom` package contains tools for converting statistical objects into nice tidy data frames. The tools in the `broom` package make it easy to process statistical results in R using the tools of the `tidyverse`.

7.6.1 broom::tidy

The `tidy()` function returns a data frame with information on the fitted model terms. For example, when applied to one of our linear models we get:

```
library(broom)
kable(tidy(mod_a), digits = 3)
```

| term | estimate | std.error | statistic | p.value |
|-------------|----------|-----------|-----------|---------|
| (Intercept) | -8.758 | 0.212 | -41.404 | 0 |
| ht | 0.234 | 0.002 | 95.227 | 0 |

```
class(tidy(mod_a))
```

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

You can pass arguments to the `tidy()` function. For example, include confidence intervals:

```
kable(tidy(mod_a, conf.int = TRUE), digits = 3)
```

| term | estimate | std.error | statistic | p.value | conf.low | conf.high |
|-------------|----------|-----------|-----------|---------|----------|-----------|
| (Intercept) | -8.758 | 0.212 | -41.404 | 0 | -9.173 | -8.343 |
| ht | 0.234 | 0.002 | 95.227 | 0 | 0.229 | 0.239 |

7.6.2 broom::augment

The `augment()` function adds information about a fitted model to the dataset used to fit the model. For example, when applied to one of our linear models we get information on the fitted values and residuals included in the output:

```
kable(head(broom::augment(mod_a), 3), digits = 3)
```

| .rownames | wt | ht | .fitted | .resid | .hat | .sigma | .cooksdi | .std.resid |
|-----------|------|------|---------|--------|-------|--------|----------|------------|
| 1 | 12.8 | 91.2 | 12.601 | 0.199 | 0.001 | 0.874 | 0 | 0.228 |
| 2 | 12.8 | 93.9 | 13.233 | -0.433 | 0.002 | 0.874 | 0 | -0.496 |
| 3 | 13.1 | 95.2 | 13.537 | -0.437 | 0.002 | 0.874 | 0 | -0.501 |

7.6.3 broom::glance

The `glance()` function returns a one row summary of a fitted model object:
For example:

```
kable(glance(mod_a, conf.int = TRUE), digits = 3)
```

| r.squared | adj.r.squared | sigma | statistic | p.value | df | logLik | AIC | BIC | deviance | df.residuals |
|-----------|---------------|-------|-----------|---------|----|-----------|---------|---------|----------|--------------|
| 0.912 | 0.912 | 0.873 | 9068.23 | 0 | 1 | -1124.615 | 2255.23 | 2269.56 | 667.351 | 8 |

7.6.4 References– statistics in R

One great (and free online for CSU students through our library) book to find out more about using R for basic statistics is:

- Introductory Statistics with R

If you want all the details about fitting linear models and GLMs in R, Julian Faraway's books are fantastic. He has one on linear models and one on extensions including logistic and Poisson models:

- Linear Models with R (also free online through the CSU library)
 - Extending the Linear Model with R
-

7.7 Functions

Download a pdf of the lecture slides for this video.

Download a pdf of the lecture slides for this video.

Download a pdf of the lecture slides for this video.

As you move to larger projects, you will find yourself using the same code a lot.

Examples include:

- Reading in data from a specific type of equipment (air pollution monitor, accelerometer)
- Running a specific type of analysis
- Creating a specific type of plot or map

If you find yourself cutting and pasting a lot, convert the code to a function.

Advantages of writing functions include:

- Coding is more efficient
- Easier to change your code (if you've cut and paste code and you want to change something, you have to change it everywhere - this is an easy way to accidentally create bugs in your code)
- Easier to share code with others

You can name a function anything you want (although try to avoid names of preexisting-existing functions). You then define any inputs (arguments; separate multiple arguments with commas) and put the code to run in braces:

```
## Note: this code will not run
[function name] <- function([any arguments]){
    [code to run]
}
```

Here is an example of a very basic function. This function takes a number as input and adds 1 to that number.

```
add_one <- function(number){
    out <- number + 1
    return(out)
}

add_one(number = 3)
```

```
## [1] 4
```

```
add_one(number = -1)
```

```
## [1] 0
```

- Functions can input any type of R object (for example, vectors, data frames, even other functions and ggplot objects)
- Similarly, functions can output any type of R object
- When defining a function, you can set default values for some of the parameters
- You can explicitly specify the value to return from the function

For example, the following function inputs a data frame (`datafr`) and a one-element vector (`child_id`) and returns only rows in the data frame where it's `id` column matches `child_id`. It includes a default value for `datafr`, but not for `child_id`.

```
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
  return(datafr)
}
```

If an argument is not given for a parameter with a default, the function will run using the default value for that parameter. For example:

```
subset_nepali(child_id = "120011")
```

```
##      id sex   wt   ht mage lit died alive age
## 1 120011 Male 12.8 91.2   35   0    2     5  41
## 2 120011 Male 12.8 93.9   35   0    2     5  45
## 3 120011 Male 13.1 95.2   35   0    2     5  49
## 4 120011 Male 13.8 96.9   35   0    2     5  53
## 5 120011 Male    NA    NA   35   0    2     5  57
```

If an argument is not given for a parameter without a default, the function call will result in an error. For example:

```
subset_nepali(datafr = nepali)
```

```
## Error in `filter()`:
## i In argument: `id == child_id`.
## Caused by error:
## ! argument "child_id" is missing, with no default
```

By default, the function will return the last defined object, although the choice of using `return` can affect printing behavior when you run the function. For example, I could have written the `subset_nepali` function like this:

```
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
}
```

In this case, the output will not automatically print out when you call the function without assigning it to an R object:

```
subset_nepali(child_id = "120011")
```

However, the output can be assigned to an R object in the same way as when the function was defined without `return`:

```
first_childs_data <- subset_nepali(child_id = "120011")
first_childs_data
```

```
##      id sex   wt  ht mage lit died alive age
## 1 120011 Male 12.8 91.2  35   0   2     5  41
## 2 120011 Male 12.8 93.9  35   0   2     5  45
## 3 120011 Male 13.1 95.2  35   0   2     5  49
## 4 120011 Male 13.8 96.9  35   0   2     5  53
## 5 120011 Male   NA   NA  35   0   2     5  57
```



R is very “good” at running functions! It will look for (scope) the variables in your function in various places (environments). So your functions may run even when you don’t expect them to, potentially, with unexpected results!

The `return` function can also be used to return an object other than the last defined object (although this doesn’t tend to be something you need to do very often). If you did not use `return` in the following code, it will output “Test output”:

```
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
  a <- "Test output"
}
(subset_nepali(child_id = "120011"))
```

```
## [1] "Test output"
```

Conversely, you can use `return` to output `datafr`, even though it's not the last object defined:

```
subset_nepali <- function(datafr = nepali, child_id){
  datafr <- datafr %>%
    filter(id == child_id)
  a <- "Test output"
  return(datafr)
}
subset_nepali(child_id = "120011")
```

```
##      id sex   wt ht mage lit died alive age
## 1 120011 Male 12.8 91.2   35   0    2     5  41
## 2 120011 Male 12.8 93.9   35   0    2     5  45
## 3 120011 Male 13.1 95.2   35   0    2     5  49
## 4 120011 Male 13.8 96.9   35   0    2     5  53
## 5 120011 Male    NA    NA   35   0    2     5  57
```

7.7.1 if / else statements

There are other control structures you can use in your R code. Two that you will commonly use within R functions are `if` and `ifelse` statements.

An `if` statement tells R that, `if` a certain condition is true, `do` run some code. For example, if you wanted to print out only odd numbers between 1 and 5, one way to do that is with an `if` statement: (Note: the `%%` operator in R returns the remainder of the first value (i) divided by the second value (2).)

```
for(i in 1:5){
  if(i %% 2 == 1){
    print(i)
  }
}
```

```
## [1] 1
## [1] 3
## [1] 5
```

The `if` statement runs some code if a condition is true, but does nothing if it is false. If you'd like different code to run depending on whether the condition is true or false, you can use an `if / else` or an `if / else if / else` statement.

```
for(i in 1:5){
  if(i %% 2 == 1){
    print(i)
  } else {
    print(paste(i, "is even"))
  }
}
```

```
## [1] 1
## [1] "2 is even"
## [1] 3
## [1] "4 is even"
## [1] 5
```

What would this code do?

```
for(i in 1:100){
  if(i %% 3 == 0 & i %% 5 == 0){
    print("FizzBuzz")
  } else if(i %% 3 == 0){
    print("Fizz")
  } else if(i %% 5 == 0){
    print("Buzz")
  } else {
    print(i)
  }
}
```

If / else statements are extremely useful in functions.

In R, the `if` statement evaluates everything in the parentheses and, if that evaluates to TRUE, runs everything in the braces. This means that you can trigger code in an `if` statement with a single-value logical vector:

```
weekend <- TRUE
if(weekend){
  print("It's the weekend!")
}

## [1] "It's the weekend!"
```

This functionality can be useful with parameters you choose to include when writing your own functions (e.g., `print = TRUE`).

7.7.2 Some other control structures

The control structure you are most likely to use in data analysis with R is the “if / else” statement. However, there are a few other control structures you may occasionally find useful:

- `next`
- `break`
- `while`

You can use the `next` structure to skip to the next round of a loop when a certain condition is met. For example, we could have used this code to print out odd numbers between 1 and 5:

```
i <- 0
while(i < 5){
  i <- 1 + i
  if(i %% 2 == 0){
    next
  }
  print(i)
}
```

```
## [1] 1
## [1] 3
## [1] 5
```

You can use `break` to break out of a loop if a certain condition is met. For example, the final code will break out of the loop once `i` is over 2, so it will only print the numbers 1 through 3:

```
i <- 0
while(i <= 5){
  if(i > 2){
    break
  }
  i <- 1 + i
  print(i)
}
```

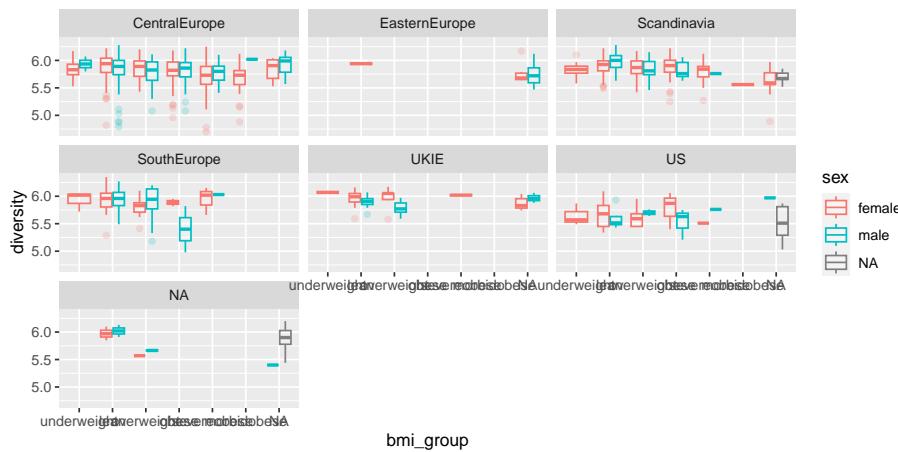
```
## [1] 1
## [1] 2
## [1] 3
```

7.8 In-course exercise for Chapter 7

7.8.1 Exploring taxonomic profiling data

- We'll be using a package on Bioconductor called `microbiome`. You'll need to install that package from Bioconductor. This uses code that's different from the default you use to download a package from CRAN. Go to the Bioconductor page for the `microbiome` package and figure out how to install this package based on instructions on that page.
- The `microbiome` that includes tools for exploring and analysing microbiome profiling data. This package has a website with tutorial information here. We want to explore a dataset on genus-level microbiota profiling ([atlas1006](#)). Navigate to the tutorial webpage to figure out how you can get this example raw data loaded in your R session. Use the `class` and `str` functions to start exploring this data. Is it in a dataframe (tibble)? Is it in a tidy format? How is the data structured?
- On the `microbiome` page, find the documentation describing the `atlas1006` data. Look through this documentation to figure out what information is included in the data. Also, check the helpfile for this dataset and look up the original article describing the data (you can find the article information in the help resources).
- The `atlas1006` data is stored in a special object class called a “phyloseq” object (you should have seen this when you used `class` with the object). You can pull certain parts of this data using special functions called “accessors”. One is `get_variable`. Try running `get_variable` with the `atlas1006` data. What do you think this data is showing?

- Which different nationalities are represented by the study subjects, based on the dataframe you extracted in the last step? How many samples have each nationality? Which different BMI groups are included? Does it look like the study was balanced among these groups?
- Based on the data you extracted, does it look like diversity varies much between males and females? Across BMI groups?
- Discuss what steps you would need to take to create the following plot. To start, don't write any code, just develop a plan. Talk about what the dataset should look like right before you create the plot and what functions you could use to get the data from its current format to that format.
- Try to write the code to create this plot. This will include some code for cleaning the data and some code for plotting. I will add one example answer after class, but I'd like you to try to figure it out yourselves first.



7.8.1.1 Example R code

Install the `microbiome` package from Bioconductor:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")

BiocManager::install("microbiome")
```

Load the `atlas1006` example data in the `microbiome` package and explore it with `str` and `class`:

```
library(microbiome)
data(atlas1006)
```

```
class(atlas1006)
```

```
## [1] "phyloseq"
## attr(,"package")
## [1] "phyloseq"
```

```
str(atlas1006)
```

```
## Formal class 'phyloseq' [package "phyloseq"] with 5 slots
## ..@ otu_table:Formal class 'otu_table' [package "phyloseq"] with 2 slots
## ... .Data      : num [1:130, 1:1151] 0 0 0 21 1 72 0 0 176 10 ...
## ... . . - attr(*, "dimnames")=List of 2
## ... . . . $ : chr [1:130] "Actinomycetaceae" "Aerococcus" "Aeromonas" "Akkermansia"
## ... . . . . $ : chr [1:1151] "Sample-1" "Sample-2" "Sample-3" "Sample-4" ...
## ... . . @ taxa_are_rows: logi TRUE
## ... . . . $ dim     : int [1:2] 130 1151
## ... . . . $ dimnames:List of 2
## ... . . . . $ : chr [1:130] "Actinomycetaceae" "Aerococcus" "Aeromonas" "Akkermansia"
## ... . . . . $ : chr [1:1151] "Sample-1" "Sample-2" "Sample-3" "Sample-4" ...
## ..@ tax_table:Formal class 'taxonomyTable' [package "phyloseq"] with 1 slot
## ... . . @ .Data: chr [1:130, 1:3] "Actinobacteria" "Firmicutes" "Proteobacteria" ...
## ... . . . - attr(*, "dimnames")=List of 2
## ... . . . . $ : chr [1:130] "Actinomycetaceae" "Aerococcus" "Aeromonas" "Akkermansia"
## ... . . . . $ : chr [1:3] "Phylum" "Family" "Genus"
## ... . . . $ dim     : int [1:2] 130 3
## ... . . . $ dimnames:List of 2
## ... . . . . $ : chr [1:130] "Actinomycetaceae" "Aerococcus" "Aeromonas" "Akkermansia"
## ... . . . . $ : chr [1:3] "Phylum" "Family" "Genus"
## ..@ sam_data :'data.frame':   1151 obs. of  10 variables:
## Formal class 'sample_data' [package "phyloseq"] with 4 slots
## ... . . @ .Data      :List of 10
## ... . . . . $ : int [1:1151] 28 24 52 22 25 42 25 27 21 25 ...
## ... . . . . $ : Factor w/ 2 levels "female","male": 2 1 2 1 1 2 1 1 1 1 ...
## ... . . . . $ : Factor w/ 6 levels "CentralEurope",...: 6 6 6 6 6 6 6 6 6 6 ...
## ... . . . . $ : Factor w/ 3 levels "o","p","r": NA NA NA NA NA NA NA NA NA ...
## ... . . . . $ : Factor w/ 40 levels "1","2","3","4",...: 1 1 1 1 1 1 1 1 1 1 ...
## ... . . . . $ : num [1:1151] 5.76 6.06 5.5 5.87 5.89 5.53 5.49 5.38 5.34 5.64 ...
## ... . . . . $ : Factor w/ 6 levels "underweight",...: 5 4 2 1 2 2 1 2 2 2 ...
```

```

## ... . . . . $ : Factor w/ 1006 levels "1","2","3","4",...: 1 2 3 4 5 6 7 8 9 10 ...
## ... . . . . $ : num [1:1151] 0 0 0 0 0 0 0 0 0 ...
## ... . . . . $ : chr [1:1151] "Sample-1" "Sample-2" "Sample-3" "Sample-4" ...
## ... . . . @ names : chr [1:10] "age" "sex" "nationality" "DNA_extraction_method" ...
## ... . . . @ row.names: chr [1:1151] "Sample-1" "Sample-2" "Sample-3" "Sample-4" ...
## ... . . . @ .S3Class : chr "data.frame"
## ..@ phy_tree : NULL
## ..@ refseq : NULL

```

Pull out the data frame that contains information on each study subject in the `atlas1006` data by using the `get_variable` accessor function:

```
get_variable(atlas1006) %>% head()
```

```

##      age   sex nationality DNA_extraction_method project diversity
## Sample-1 28 male        US                  <NA>     1    5.76
## Sample-2 24 female      US                  <NA>     1    6.06
## Sample-3 52 male        US                  <NA>     1    5.50
## Sample-4 22 female      US                  <NA>     1    5.87
## Sample-5 25 female      US                  <NA>     1    5.89
## Sample-6 42 male        US                  <NA>     1    5.53
##          bmi_group subject time   sample
## Sample-1 severeobese      1    0 Sample-1
## Sample-2      obese       2    0 Sample-2
## Sample-3      lean        3    0 Sample-3
## Sample-4 underweight     4    0 Sample-4
## Sample-5      lean        5    0 Sample-5
## Sample-6      lean        6    0 Sample-6

```

Note: Since the first argument to `get_variable` is the phyloseq object (here, the `atlas1006` data object), you can pipe into the function, like this:

```
atlas1006 %>%
  get_variable() %>%
  head()
```

Find out the different nationalities included in the data:

```
atlas1006 %>%
  get_variable() %>%
  as_tibble() %>% # Change output from a data.frame to a tibble
  group_by(nationality) %>%
  count()
```

```
## # A tibble: 7 x 2
## # Groups:   nationality [7]
##   nationality     n
##   <fct>        <int>
## 1 CentralEurope    650
## 2 EasternEurope     15
## 3 Scandinavia      271
## 4 SouthEurope       89
## 5 UKIE              50
## 6 US                 44
## 7 <NA>               32
```

It looks like most subjects were from Central Europe, with the next-largest group from Scandinavia. **Note:** If you wanted to rearrange this summary to give the nationalities in order of the number of subjects in each, you could add on a `forcats` function:

```
atlas1006 %>%
  get_variable() %>%
  as_tibble() %>% # Change output from a data.frame to a tibble
  mutate(nationality = fct_infreq(nationality)) %>%
  group_by(nationality) %>%
  count()
```

```
## # A tibble: 7 x 2
## # Groups:   nationality [7]
##   nationality     n
##   <fct>        <int>
## 1 CentralEurope    650
## 2 Scandinavia      271
## 3 SouthEurope       89
## 4 UKIE              50
## 5 US                 44
## 6 EasternEurope     15
## 7 <NA>               32
```

Find out the different BMI groups included in the data and if the study seemed to be balanced across those groups:

```
atlas1006 %>%
  get_variable() %>%
  as_tibble() %>% # Change output from a data.frame to a tibble
  group_by(bmi_group) %>%
  count()

## # A tibble: 7 x 2
## # Groups:   bmi_group [7]
##   bmi_group     n
##   <fct>       <int>
## 1 underweight    21
## 2 lean          484
## 3 overweight     197
## 4 obese          222
## 5 severeobese    99
## 6 morbidobese    22
## 7 <NA>         106
```

There were six different BMI groups. Over 100 study subjects had this information missing. The samples were not evenly distributed across these BMI groups. Instead, the most common (lean) had almost 500 subjects, while the smaller BMI-group samples were around 20 people.

See if it looks like diversity varies much between males and females:

```
atlas1006 %>%
  get_variable() %>%
  as_tibble() %>%
  group_by(sex) %>%
  summarize(mean_diversity = mean(diversity))

## # A tibble: 3 x 2
##   sex      mean_diversity
##   <fct>            <dbl>
## 1 female           5.84
## 2 male             5.85
## 3 <NA>            5.76
```

See if it looks like diversity varies much across BMI groups:

```
atlas1006 %>%
  get_variable() %>%
  as_tibble() %>%
  group_by(bmi_group) %>%
  summarize(mean_diversity = mean(diversity))
```

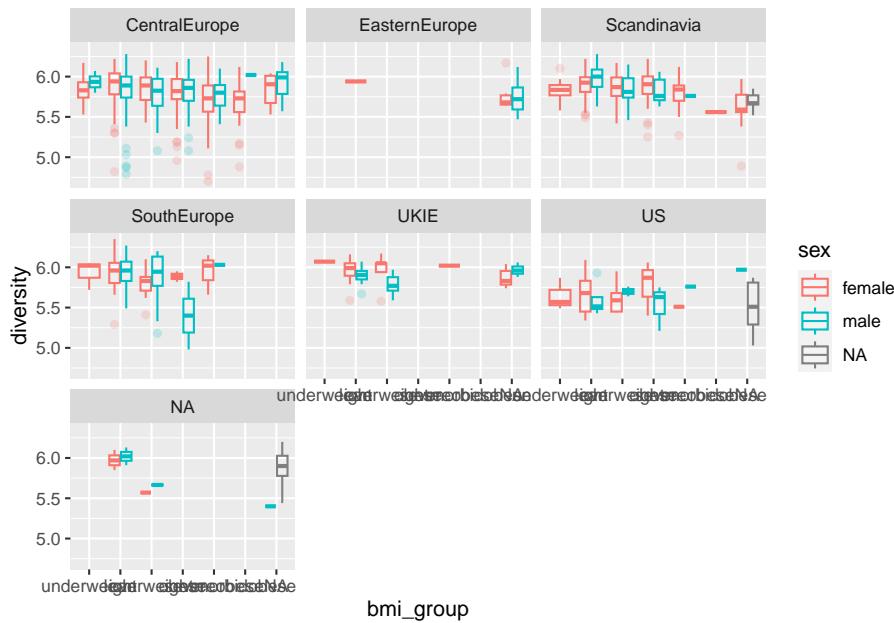
```
## # A tibble: 7 x 2
##   bmi_group  mean_diversity
##   <fct>          <dbl>
## 1 underweight      5.84
## 2 lean              5.89
## 3 overweight        5.83
## 4 obese             5.82
## 5 severeobese       5.74
## 6 morbidobese       5.67
## 7 <NA>              5.81
```

Here is the code for the plot:

```
library(tidyverse)
library(ggthemes)
library(microbiome)

data(atlas1006)

atlas1006 %>%
  get_variable() %>%
  ggplot(aes(x = bmi_group, y = diversity, color = sex)) +
  geom_boxplot(alpha = 0.2) +
  facet_wrap(~ nationality)
```



7.8.2 More with baby names

Let's look at baby names that we started looking at last class, based on the letter they start with.

- For the full dataframe, what proportion of baby names start with each letter? See if you can create a figure to help show this. Create the same plot using the names of people from our class.
- What proportion of names start with “C” or “S” across the full dataset?

7.8.2.1 Example R code

For the full dataframe, what proportion of baby names start with “S”?

To start, create a column with the first letter of each name. You can use functions in the `stringr` package to do this. The easiest might be to use the position of the first letter to pull that information.

```
library(babynames)
library(stringr)
top_letters <- babynames %>%
  mutate(first_letter = str_sub(name, 1, 1))
```

```
top_letters %>%
  select(name, first_letter) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 2
##   name     first_letter
##   <chr>    <chr>
## 1 Mary      M
## 2 Anna      A
## 3 Emma      E
## 4 Elizabeth E
## 5 Minnie   M
```

Now we can group by letter and figure out these proportions. First, while the data is grouped, count the number of names with each letter. Then, ungroup and use mutate to divide this by the total number of names:

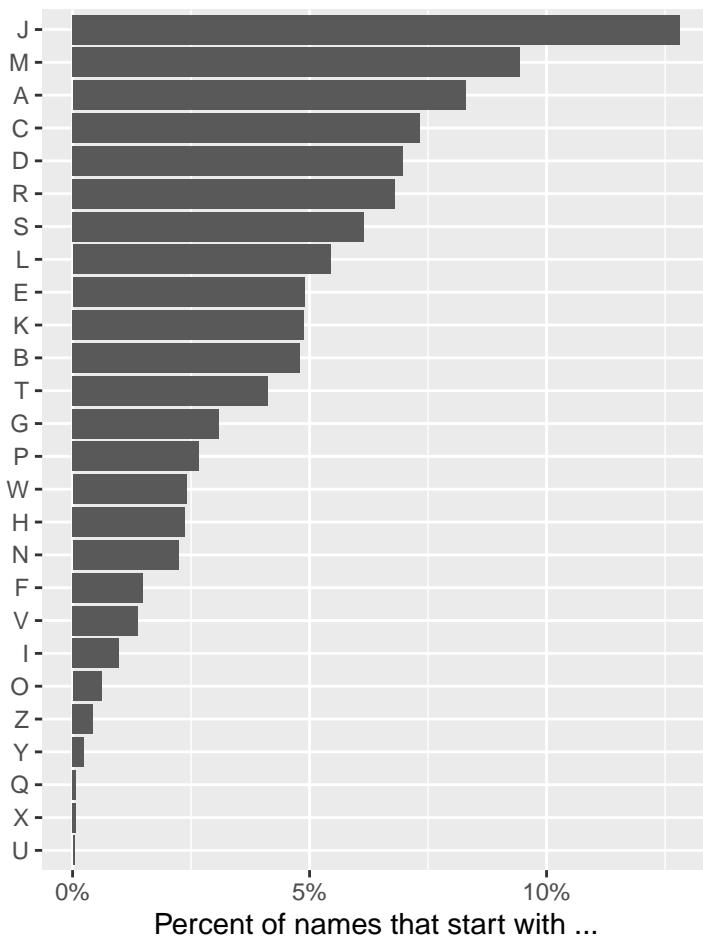
```
top_letters <- top_letters %>%
  group_by(first_letter) %>%
  summarize(n = sum(n)) %>%
  ungroup() %>%
  mutate(prop = n / sum(n)) %>%
  arrange(desc(prop))

top_letters

## # A tibble: 26 x 3
##   first_letter     n   prop
##   <chr>       <int>  <dbl>
## 1 J           44612175 0.128
## 2 M           32864210 0.0944
## 3 A           28855232 0.0829
## 4 C           25533863 0.0733
## 5 D           24240271 0.0696
## 6 R           23702794 0.0681
## 7 S           21373830 0.0614
## 8 L           18942067 0.0544
## 9 E           17033760 0.0489
## 10 K          17006684 0.0489
## # i 16 more rows
```

Here's one way to visualize this:

```
library(scales)
top_letters %>%
  mutate(first_letter = fct_reorder(first_letter, prop)) %>%
  ggplot(aes(x = first_letter)) +
  geom_bar(aes(weight = prop)) +
  coord_flip() +
  scale_y_continuous(labels = percent) +
  labs(x = "", y = "Percent of names that start with ...")
```



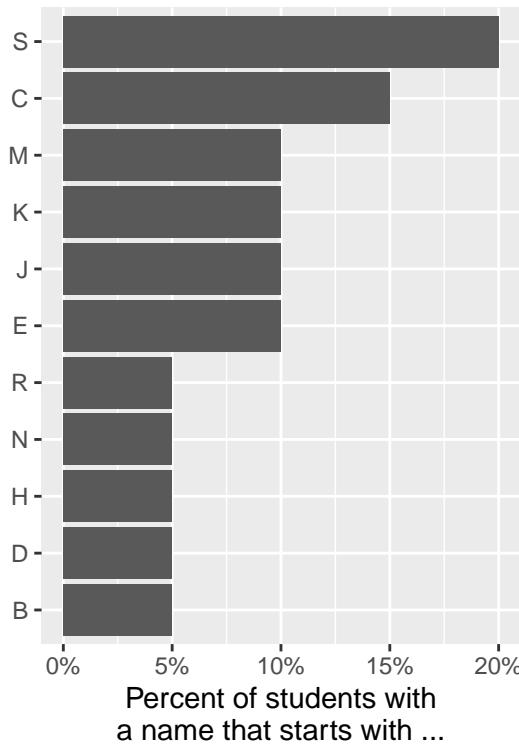
Create the same plot using the names of people in our class. First, create a vector with the names of people in our class:

```
student_list <- data_frame(name = c("Burton", "Caroline", "Chaoyu", "Collin",
                                    "Daniel", "Eric", "Erin", "Heather",
                                    "Jacob", "Jessica", "Khum", "Kyle",
                                    "Matthew", "Molly", "Nikki", "Rachel",
                                    "Sere", "Shayna", "Sherry", "Sunny"))
student_list <- student_list %>%
  mutate(first_letter = str_sub(name, 1, 1))
student_list
```

```
## # A tibble: 20 x 2
##   name    first_letter
##   <chr>   <chr>
## 1 Burton   B
## 2 Caroline C
## 3 Chaoyu   C
## 4 Collin   C
## 5 Daniel   D
## 6 Eric     E
## 7 Erin     E
## 8 Heather  H
## 9 Jacob    J
## 10 Jessica J
## 11 Khum    K
## 12 Kyle    K
## 13 Matthew M
## 14 Molly   M
## 15 Nikki   N
## 16 Rachel  R
## 17 Sere    S
## 18 Shayna  S
## 19 Sherry  S
## 20 Sunny   S
```

```
library(scales)
student_list %>%
  group_by(first_letter) %>%
  count() %>%
  ungroup() %>%
  mutate(prop = n / sum(n)) %>%
  mutate(first_letter = fct_reorder(first_letter, prop)) %>%
  ggplot(aes(x = first_letter)) +
  geom_bar(aes(weight = prop)) +
```

```
coord_flip() +
scale_y_continuous(labels = percent) +
labs(x = "", y = "Percent of students with\na name that starts with ...")
```



What proportion of names start with “C” or “S” across the full dataset? You can create a dataframe that (1) pulls out the first letter of each name (just like we did for the last part of the question) and (2) tests whether that first letter is an “A” or a “K” (using a logical statement):

```
c_or_s <- babynames %>%
  mutate(first_letter = str_sub(name, 1, 1),
        c_or_s = first_letter %in% c("C", "S"))
```

```
c_or_s %>%
  select(name, first_letter, c_or_s) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 3
```

```
##   name      first_letter c_or_s
##   <chr>    <chr>        <lgl>
## 1 Mary      M           FALSE
## 2 Anna      A           FALSE
## 3 Emma      E           FALSE
## 4 Elizabeth E           FALSE
## 5 Minnie    M           FALSE
```

Next, group by this logical column (`c_or_s`) and figure out the number of baby names for each group. Then, to get the proportion of the total, ungroup and mutate to divide by the total number across the data:

```
c_or_s %>%
  group_by(c_or_s) %>%
  count() %>%
  ungroup() %>%
  mutate(prop = n / sum(n))
```

```
## # A tibble: 2 x 3
##   c_or_s     n   prop
##   <lgl>   <int> <dbl>
## 1 FALSE  1655755 0.860
## 2 TRUE   268910  0.140
```

7.8.3 Running a simple statistical test

In the last part of the in-course exercise, we found out that about 14% of babies born in the United States between 1980 and 1995 had names that started with an “C” or “S” (268,910 babies out of 1,924,665).

- What is the proportion of people with names that start with an “C” or “S” in our class?
- Use a simple statistical test to test the hypothesis that the class comes from a binomial distribution with the same distribution as babies born in the US over the time tracked by this data, in terms of chance of having a name that starts with “C” or “S”. (Hint: You will be comparing two proportions. Try googling for a statistical test in R that does that.)
- See if you can figure out a way to make a single “tidy” pipeline for the whole analysis (and output the result as a tidy dataframe). Does the `tidy` function from `broom` give different information from this test than the output we got for the Shapiro-Wilk test?
- You may get the warning “Chi-squared approximation may be incorrect”. See if you can figure out this warning if you get it with the test you used.

7.8.3.1 Example R code

Here is a vector with names in our class:

```
library(stringr)
student_list <- tibble(name = c("Burton", "Caroline", "Chaoyu", "Collin",
                               "Daniel", "Eric", "Erin", "Heather",
                               "Jacob", "Jessica", "Khum", "Kyle",
                               "Matthew", "Molly", "Nikki", "Rachel",
                               "Sere", "Shayna", "Sherry", "Sunny"))
student_list <- student_list %>%
  mutate(first_letter = str_sub(name, 1, 1))
student_list %>%
  slice(1:3)

## # A tibble: 3 x 2
##   name     first_letter
##   <chr>    <chr>
## 1 Burton   B
## 2 Caroline C
## 3 Chaoyu   C
```

Let's get the total number of students, and then the total number with a name that starts with "C" or "S":

```
tot_students <- student_list %>%
  count()
tot_students

## # A tibble: 1 x 1
##       n
##   <int>
## 1     20

c_or_s_students <- student_list %>%
  mutate(c_or_s = first_letter %in% c("C", "S")) %>%
  group_by(c_or_s) %>%
  count()
c_or_s_students
```

```
## # A tibble: 2 x 2
## # Groups:   c_or_s [2]
##   c_or_s     n
##   <lgcl> <int>
## 1 FALSE      13
## 2 TRUE       7
```

The proportion of students with names starting with “A” or “K” are $7 / 20 = 0.35$.

You could run a statistical test comparing these two proportions (check the help file for the function to figure out where to include each piece):

```
prop.test(x = c(268910, 7), n = c(1924665, 20))
```

```
## Warning in prop.test(x = c(268910, 7), n = c(1924665, 20)): Chi-squared
## approximation may be incorrect

##
## 2-sample test for equality of proportions with continuity correction
##
## data: c(268910, 7) out of c(1924665, 20)
## X-squared = 5.7121, df = 1, p-value = 0.01685
## alternative hypothesis: two.sided
## 95 percent confidence interval:
## -0.44432032 0.02375596
## sample estimates:
## prop 1   prop 2
## 0.1397178 0.3500000
```

There are a few different ways you could run this test. For example, you could also test whether the proportion in our class is consistent with the null hypothesis that you were drawn from a binomial distribution with a proportion of 0.14 (in-line with the national values):

```
prop.test(x = 7, n = 20, p = 0.14)
```

```
## Warning in prop.test(x = 7, n = 20, p = 0.14): Chi-squared approximation may be
## incorrect
```

```
##  
## 1-sample proportions test with continuity correction  
##  
## data: 7 out of 20, null probability 0.14  
## X-squared = 5.6852, df = 1, p-value = 0.01711  
## alternative hypothesis: true p is not equal to 0.14  
## 95 percent confidence interval:  
## 0.1630867 0.5905104  
## sample estimates:  
## p  
## 0.35
```

You can also see if we can pipe into `prop.test` by summing up the number of successes (“1”: the person’s name starts with “C” or “S”). Because this is using an unsummarized form of the data, it lets us use some of the tidyverse tools more easily:

```
library(purrr)  
library(broom)  
student_list %>%  
  mutate(c_or_s = first_letter %in% c("C", "S")) %>%  
  pull("c_or_s") %>%  
  sum() %>%  
  prop.test(n = 20, p = 0.14) %>%  
  tidy()  
  
## Warning in prop.test(., n = 20, p = 0.14): Chi-squared approximation may be  
## incorrect  
  
## # A tibble: 1 x 8  
##   estimate statistic p.value parameter conf.low conf.high method      alternative  
##       <dbl>     <dbl>    <dbl>    <int>     <dbl>     <dbl> <chr>      <chr>  
## 1      0.35      5.69   0.0171        1     0.163     0.591 1-sample ~ two.sided
```

Finally, when we run this test, we get the warning that “Chi-squared approximation may be incorrect”. Based on googling ‘r `prop.test` “Chi-squared approximation may be incorrect”’, it sounds like we might be getting this error because we have a pretty low number of people in the class. One recommendation is to use `binom.test`, which will run as an exact binomial test:

```
binom.test(x = 7, n = 20, p = 0.14)
```

```

## Exact binomial test
##
## data: 7 and 20
## number of successes = 7, number of trials = 20, p-value = 0.01534
## alternative hypothesis: true probability of success is not equal to 0.14
## 95 percent confidence interval:
## 0.1539092 0.5921885
## sample estimates:
## probability of success
##                         0.35

```

7.8.4 Using regression models to explore data #1

For this exercise, you will need the following packages. If do not have them already, you will need to install them.

```

library(ggplot2)
library(broom)
library(ggfortify)

```

For this part of the exercise, you'll use a dataset on weather, air pollution, and mortality counts in Chicago, IL. This dataset is called `chicagoNMMAPS` and is part of the `dlnm` package. Change the name of the data frame to `chic` (this object name is shorter and will be easier to work with). Check out the data a bit to see what variables you have, and then perform the following tasks:

- Write out (on paper, not in R) the regression equation for regressing dew-point temperature on temperature.
- Try fitting a linear regression of dew point temperature (`dptp`) regressed on temperature (`temp`). Save this model as the object `mod_1` (i.e., is the dependent variable of dewpoint temperature linearly associated with the independent variable of temperature).
- Based on this regression, does there seem to be a relationship between temperature and dewpoint temperature in Chicago? (Hint: Try using `glance` and `tidy` from the `broom` package on the model object to get more information about the model you fit.) What is the coefficient for temperature (in other words, for every 1 degree increase in temperature, how much do we expect the dewpoint temperature to change?)? What is the p-value for the coefficient for temperature?
- Plot temperature (x-axis) versus dewpoint temperature (y-axis) for Chicago. Add in the regression line from the model you fit by using the results from `augment`.

- Use `autoplot` on the model object to generate some model diagnostic plots (make sure you have the `ggfortify` package loaded and installed).
-

7.8.4.1 Example R code:

The regression equation for the model you want to fit, regressing dewpoint temperature on temperature, is:

$$Y_t \sim \beta_0 + \beta_1 X_t$$

where Y_t is the dewpoint temperature on day t , X_t is the temperature on day t , and β_0 and β_1 are model coefficients.

Install and load the `dlnm` package and then load the `chicagoNMMAPS` data. Change the name of the data frame to `chic`, so it will be shorter to call for the rest of your work.

```
# install.packages("dlnm")
library(dlnm)
data("chicagoNMMAPS")
chic <- chicagoNMMAPS
```

Fit a linear regression of `dptp` on `temp` and save as the object `mod_1`:

```
mod_1 <- lm(dptp ~ temp, data = chic)
mod_1
```

```
##
## Call:
## lm(formula = dptp ~ temp, data = chic)
##
## Coefficients:
## (Intercept)      temp
##       24.025     1.621
```

Use functions from the `broom` package to pull the same information about the model in a “tidy” format. To find out if the evidence for a linear association between temperature and dewpoint temperature, use the `tidy` function to get model coefficients in a tidy format:

```
tidy(mod_1)
```

```
## # A tibble: 2 x 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept) 24.0     0.113     213.      0
## 2 temp        1.62     0.00763    212.      0
```

There does seem to be an association between temperature and dewpoint temperature: a unit increase in temperature is associated with a 1.6 unit increase in dewpoint temperature. The p-value for the temperature coefficient is $<2e-16$. This is far below 0.05, which suggests we would be very unlikely to see such a strong association by chance if the null hypothesis, that the two variables are not associated, were true.

You can also check overall model summaries using the `glance` function:

```
glance(mod_1)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic p.value    df logLik     AIC     BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     0.898         0.898  5.90     45108.      0     1 -16332. 32670. 32690.
## # i 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

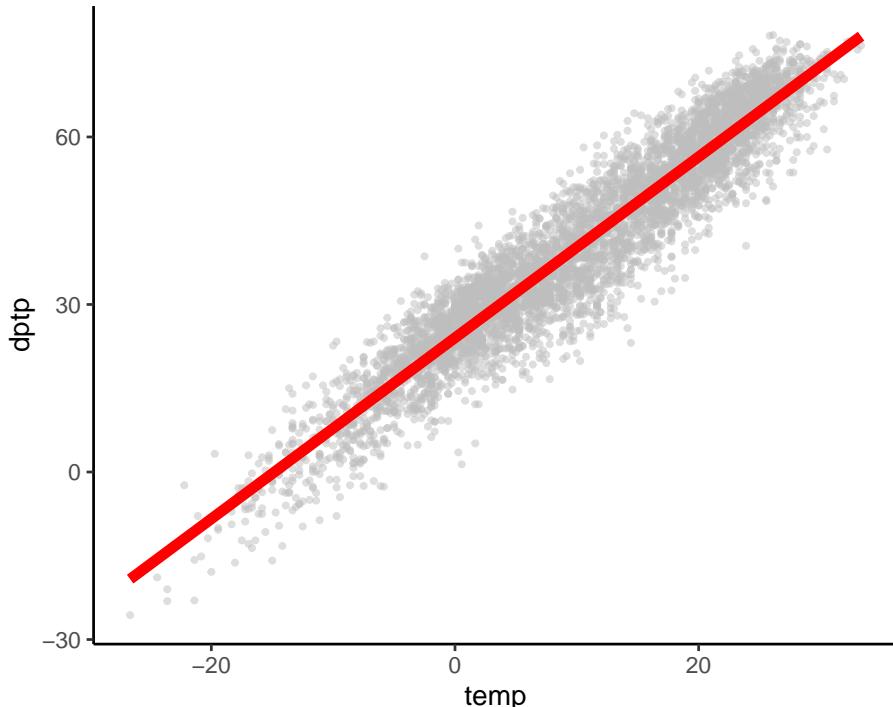
To create plots of the observations and the fit model, use the `augment` function to add model output (e.g., predictions, residuals) to the original data frame of observed temperatures and dew point temperatures:

```
augment(mod_1) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 8
##   dptp   temp .fitted .resid     .hat .sigma   .cooksdi .std.resid
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>     <dbl>
## 1 31.5 -0.278   23.6    7.93 0.000376   5.90 0.000340    1.34
## 2 29.9  0.556   24.9    4.95 0.000348   5.90 0.000123    0.839
## 3 27.4  0.556   24.9    2.45 0.000348   5.90 0.0000300   0.415
```

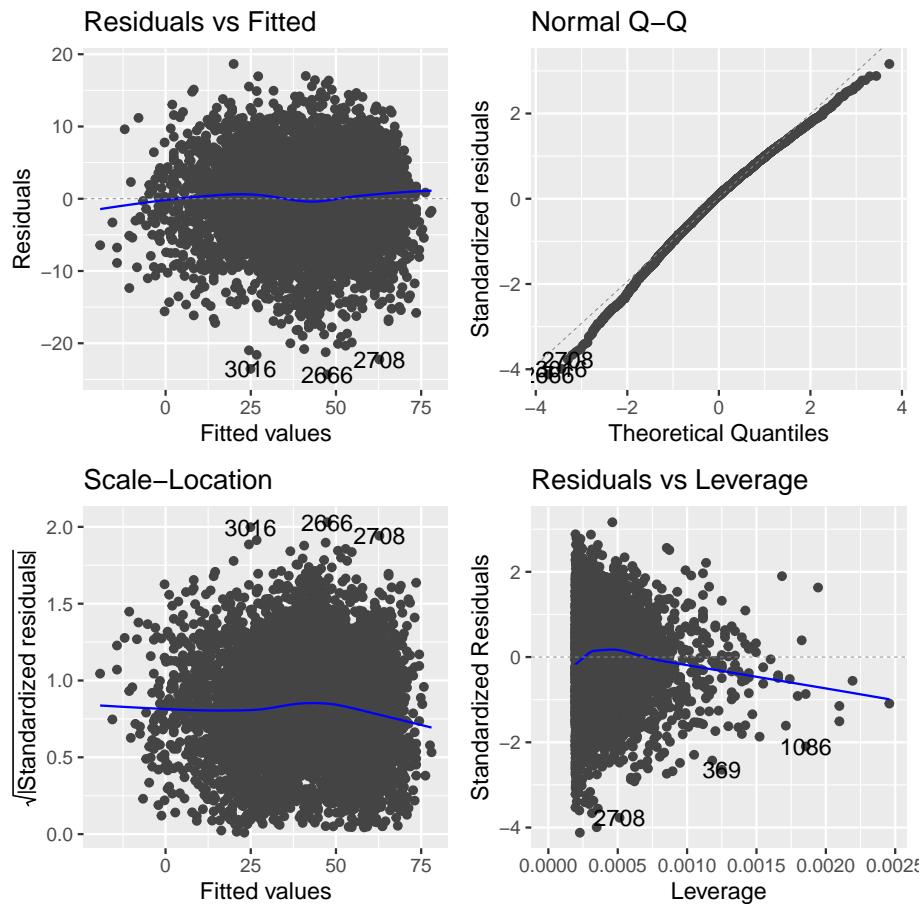
Plot these two variables and add in the fitted line from the model (note: I've used the `color` option to make the color of the points gray). Use the output from `augment` to create a plot of the original data, with the predicted values used to plot a fitted line.

```
augment(mod_1) %>%
  ggplot(aes(x = temp, y = dptp)) +
  geom_point(size = 0.8, alpha = 0.5, col = "gray") +
  geom_line(aes(x = temp, y = .fitted), color = "red", size = 2) +
  theme_classic()
```



Plot some plots to check model assumptions for the model you fit using the `autoplot` function on your model object:

```
autoplot(mod_1)
```



7.8.5 Using regression models to explore data #2

- Try fitting the regression from the last part of the in-course exercise as a GLM, using `glm()` (but still assuming the outcome variable is normally distributed). Are your coefficients different?
- Does PM_{10} vary by day of the week? (Hint: The `dow` variable is a factor that gives day of the week. You can do an ANOVA analysis by fitting a linear model using this variable as the independent variable. Some of the overall model summaries will compare this model to an intercept-only model.) What day of the week is PM_{10} generally highest? (Check the model coefficients to figure this out.) Try to write out (on paper) the regression equation for the model you're fitting.

- Try using `glm()` to run a Poisson regression of respiratory deaths (`resp`) on temperature during summer days. Start by creating a subset with just summer days called `summer`. (Hint: Use the `month` function with the argument `label = TRUE` from `lubridate` to do this—just pull out the subset where the month June, July, or August.) Try to write out the regression equation for the model you’re fitting.
 - The coefficient for the temperature variable in this model is our best estimate (based on this model) of the **log relative risk** for a one degree Celcius increase in temperature. What is the **relative risk** associated with a one degree Celsius increase?
-

7.8.5.1 Example R code:

Try fitting the model from the last part of the in-course exercise using `glm()`. Call it `mod_1a`. Compare the coefficients for the two models. You can use the `tidy` function on an `lm` or `glm` object to pull out just the model coefficients and associated model results. Here, I’ve used a pipeline of code to create a tidy data frame that merges these “tidy” coefficient outputs (from the two models) into a single data frame:

```
mod_1a <- glm(dptp ~ temp, data = chic)

tidy(mod_1) %>%
  select(term, estimate) %>%
  inner_join(mod_1a %>% tidy() %>% select(term, estimate), by = "term") %>%
  rename(estimate_lm_mod = estimate.x,
        estimate_glm_mod = estimate.y)

## # A tibble: 2 x 3
##   term      estimate_lm_mod estimate_glm_mod
##   <chr>          <dbl>            <dbl>
## 1 (Intercept)    24.0            24.0
## 2 temp           1.62            1.62
```

The results from the two models are identical.

As a note, you could have also just run `tidy` on each model object, without merging them together into a single data frame:

```
tidy(mod_1)
```

```
## # A tibble: 2 x 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept) 24.0     0.113     213.      0
## 2 temp        1.62     0.00763    212.      0
```

```
tidy(mod_1a)
```

```
## # A tibble: 2 x 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept) 24.0     0.113     213.      0
## 2 temp        1.62     0.00763    212.      0
```

Fit a model of PM_{10} regressed on day of week, where day of week is a factor.

```
mod_2 <- lm(pm10 ~ dow, data = chic)
tidy(mod_2)
```

```
## # A tibble: 7 x 5
##   term      estimate std.error statistic   p.value
##   <chr>     <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept) 27.5     0.730     37.7 7.47e-273
## 2 dowMonday    6.13     1.03      5.93 3.22e- 9
## 3 dowTuesday   6.80     1.03      6.62 4.05e-11
## 4 dowWednesday 8.48     1.03      8.26 1.85e-16
## 5 dowThursday   8.80     1.02      8.60 1.08e-17
## 6 dowFriday     9.48     1.03      9.24 3.61e-20
## 7 dowSaturday   3.66     1.03      3.56 3.68e- 4
```

Use `glance` to check some of the overall summaries of this model. The `statistic` column here is the F statistic from test comparing this model to an intercept-only model.

```
glance(mod_2)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic  p.value     df logLik     AIC     BIC
##       <dbl>         <dbl>    <dbl>      <dbl>     <dbl>     <dbl>    <dbl>    <dbl>
## 1     0.0259      0.0247  19.1      21.5 4.61e-25     6 -21234. 42484. 42536.
## # i 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

As a note, you may have heard in previous statistics classes that you can use the `anova()` command to compare this model to a model with only an intercept (i.e., one that only fits a global mean and uses that as the expected value for all of the observations). Note that, in this case, the F value from `anova` for this model comparison is the same as the `statistic` you got in the overall summary statistics you get with `glance` in the previous code.

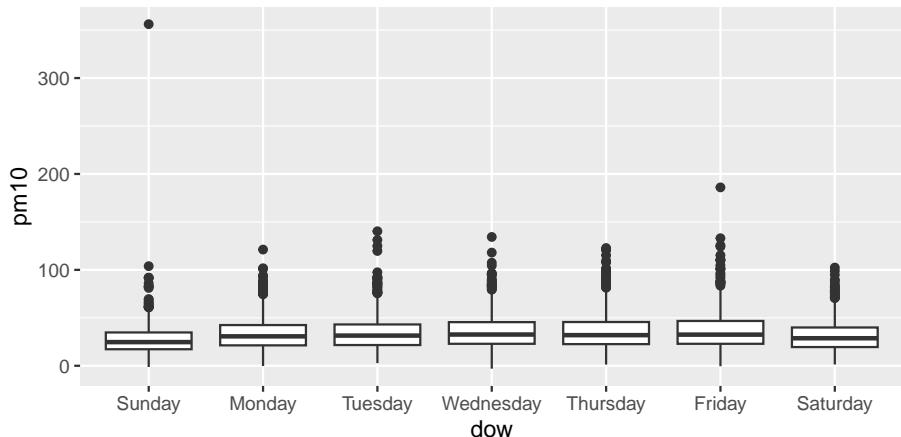
```
anova(mod_2)
```

```
## Analysis of Variance Table
##
## Response: pm10
##             Df  Sum Sq Mean Sq F value    Pr(>F)
## dow          6  46924  7820.6    21.5 < 2.2e-16 ***
## Residuals 4856 1766407   363.8
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The overall p-value from `anova` for with day-of-week coefficients versus the model that just has an intercept is $< 2.2\text{e-}16$. This is well below 0.05, which suggests that day-of-week is associated with PM10 concentration, as a model that includes day-of-week does a much better job of explaining variation in PM10 than a model without it does.

Use a boxplot to visually compare PM10 by day of week.

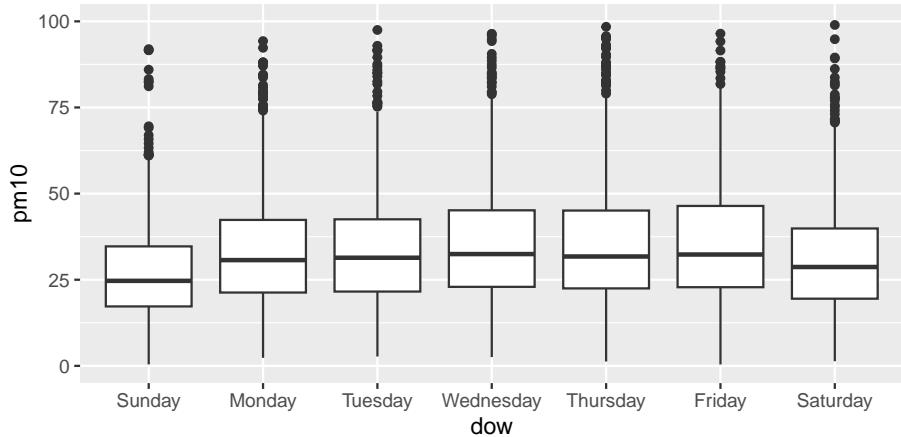
```
ggplot(chic, aes(x = dow, y = pm10)) +
  geom_boxplot()
```



Now try the same plot, but try using the `ylim =` option to change the limits on the y-axis for the graph, so you can get a better idea of the pattern by day of week (some of the extreme values are very high, which makes it hard to compare by eye when the y-axis extends to include them all).

```
ggplot(chic, aes(x = dow, y = pm10)) +
  geom_boxplot() +
  ylim(c(0, 100))
```

```
## Warning: Removed 292 rows containing non-finite values (`stat_boxplot()`).
```



Create a subset called `summer` with just the summer days:

```

library(lubridate)
summer <- chic %>%
  mutate(month = month(date, label = TRUE)) %>%
  filter(month %in% c("Jun", "Jul", "Aug"))
summer %>%
  slice(1:3)

##           date time year month doy      dow death cvd resp      temp dptp   rhum
## 1 1987-06-01 152 1987    Jun 152 Monday  112   60     5 23.61111 68.50 71.875
## 2 1987-06-02 153 1987    Jun 153 Tuesday 111   57     7 22.22222 64.75 95.250
## 3 1987-06-03 154 1987    Jun 154 Wednesday 120   59     9 20.55556 47.25 47.125
##          pm10      o3
## 1 22.95607 34.94623
## 2 31.31339 18.96620
## 3 34.95607 23.59270

```

Use `glm()` to fit a Poisson model of respiratory deaths regressed on temperature. Since you want to fit a Poisson model, use the option `family = poisson(link = "log")`.

```

mod_3 <- glm(resp ~ temp, data = summer,
              family = poisson(link = "log"))
glance(mod_3)

```

```

## # A tibble: 1 x 8
##   null.deviance df.null logLik   AIC   BIC deviance df.residual nobs
##             <dbl>   <int>  <dbl>  <dbl>  <dbl>       <int>     <int>
## 1         1499.     1287 -3211.  6425.  6436.      1494.      1286  1288

```

```

tidy(mod_3)

```

```

## # A tibble: 2 x 5
##   term      estimate std.error statistic  p.value
##   <chr>      <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) 1.91      0.0584    32.7  6.60e-235
## 2 temp        0.00614   0.00258    2.38  1.74e- 2

```

Use the fitted model coefficient to determine the relative risk for a one degree Celsius increase in temperature. First, remember that you can use the `tidy()`

function to read out the model coefficients. The second of these is the value for the temperature coefficient. That means that you can use indexing ([2]) to get just that value. That's the log relative risk; take the exponent to get the relative risk.

```
tidy(mod_3) %>%
  filter(term == "temp") %>%
  mutate(log_rr = exp(estimate))

## # A tibble: 1 x 6
##   term    estimate std.error statistic p.value log_rr
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 temp     0.00614   0.00258     2.38    0.0174    1.01
```

As a note, you can use the `conf.int` parameter in `tidy` to also pull confidence intervals:

```
tidy(mod_3, conf.int = TRUE)

## # A tibble: 2 x 7
##   term      estimate std.error statistic  p.value conf.low conf.high
##   <chr>       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) 1.91      0.0584    32.7  6.60e-235  1.80      2.02
## 2 temp        0.00614   0.00258     2.38  1.74e- 2  0.00108   0.0112
```

You could use this to get the confidence interval for relative risk (check out the `mutate_at` function if you haven't seen it before):

```
tidy(mod_3, conf.int = TRUE) %>%
  select(term, estimate, conf.low, conf.high) %>%
  filter(term == "temp") %>%
  mutate_at(vars(estimate:conf.high), funs(exp(.)))

## Warning: `funs()` was deprecated in dplyr 0.8.0.
## i Please use a list of either functions or lambdas:
##
## # Simple named list: list(mean = mean, median = median)
##
## # Auto named with `tibble::lst()`: tibble::lst(mean, median)
```

```

## 
## # Using lambdas list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

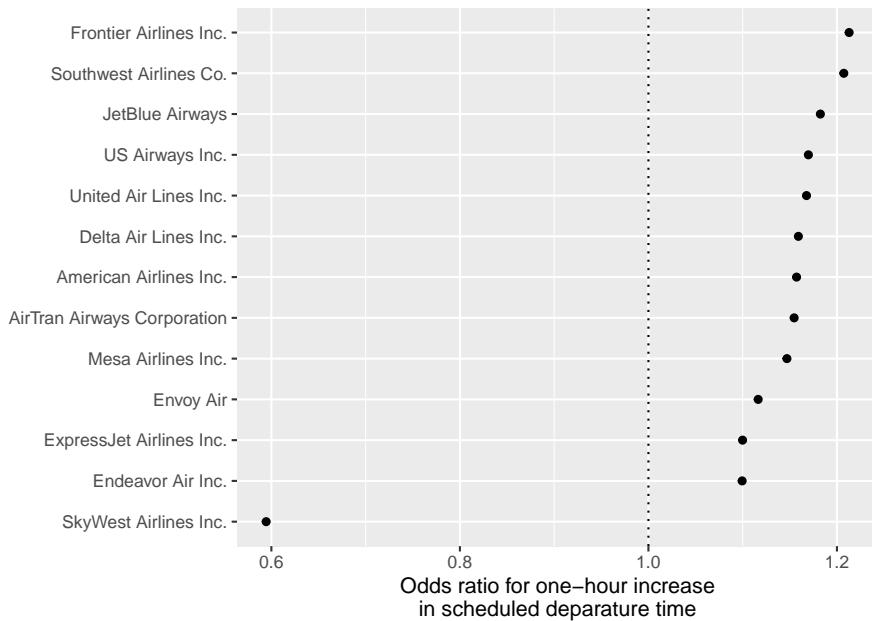
## # A tibble: 1 x 4
##   term    estimate conf.low conf.high
##   <chr>     <dbl>     <dbl>     <dbl>
## 1 temp      1.01     1.00     1.01

```

7.8.6 Trying out nesting and mapping

- We'll be using data that's in an R package called "nycflights13". This data package can be installed from CRAN. Install the package and then load the package and its "flights" dataset. So that this data will be easier to work with, remove all columns except for those for the departure delay (`dep_delay`), the carrier (`carrier`), the hour the flight was supposed to leave (`hour`), and the airport the flight left from (`origin`). Also, limit the dataset to only flights that left from LaGuardia Airport ("LGA").
- We want to figure out if the probability of a flight leaving 15 minutes late or more increases over the day for flights leaving LaGuardia. Filter out all the rows where the departure delay is missing and then create a new column called `late_dep` that is true if the flight left 15 minutes or more late and false otherwise. What proportion of all flights leaving from LaGuardia leave 15 minutes late or later?
- Next, determine what proportion of all flights are delayed base on the hour that the flight was scheduled to depart (`hour`). Create a plot showing how the probability of leaving 15 minutes or more late changes by hour.
- Fit a generalized linear model for the association between the binary variable of whether the flight was 15 minutes or more late (`late_dep`) and the hour the flight was scheduled to leave (`hour`). Use a binomial model (add `family = binomial(link = "logit")` in the `glm` call). The estimate from the model for `hour` will be an estimate of the log odds ratio for a one-hour increase in scheduled departure time. Take the exponent of this estimate with `exp` to get an estimated odds ratio for a one-hour increase in scheduled departure time. Is this estimate larger than 1.0?
- Next, we want to see if this association is similar across airlines. First, create a dataframe called `nested_flights` where the `flights` data is grouped by airline (`carrier`) and then nest the data, so that there is a "data" list-column where each item is a dataframe of flight delay data for a specific carrier.
- Use the `map` function from `purrr` inside a `mutate` statement to apply the `glm` code you used earlier for the whole dataset, but in this case for the data for each airline character. Then use the `map` function inside a `mutate` statement again to "tidy" the data.

- Clean the data up a bit. Remove the columns for `data` and `glm_result` and then `unnest` the dataframe list-column with the tidy version of the model results. Filter to get only the estimates for the “hour” term. Then calculate an odds ratio (`or`) by taking the exponent (check out the `exp` function) of the original estimate.
- The package has a dataframe with the full name of each carrier (`airlines`). Join this data into the data you’ve been working with, so you have the full names of airlines.
- Finally, create the following plot with each airline’s odds ratio for the change in the chance of a delay per one-hour increase in the scheduled hour of departure:



7.8.6.1 Example R code:

Install the “nycflights13” package from CRAN. Load the package and its “flights” dataset.

```
library(ggplot2)
library(nycflights13)
data(flights)
```

So that this data will be easier to work with, remove all columns except for those for the departure delay (`dep_delay`), the carrier (`carrier`), the hour the flight was supposed to leave (`hour`), and the airport the flight left from (`origin`).

```
flights <- flights %>%
  select(dep_delay, carrier, hour, origin) %>%
  filter(origin == "LGA")
```

```
flights
```

```
## # A tibble: 104,662 x 4
##   dep_delay carrier  hour origin
##       <dbl> <chr>    <dbl> <chr>
## 1      4 UA        5 LGA
## 2     -6 DL        6 LGA
## 3     -3 EV        6 LGA
## 4     -2 AA        6 LGA
## 5     -1 AA        6 LGA
## 6      0 B6        6 LGA
## 7      0 MQ        6 LGA
## 8     -8 DL        6 LGA
## 9     -3 MQ        6 LGA
## 10    13 AA        6 LGA
## # i 104,652 more rows
```

Filter out all the rows where the departure delay is missing and then create a new column called `late_dep` that is true if the flight left 15 minutes or more late and false otherwise.

```
flights <- flights %>%
  filter(!is.na(dep_delay)) %>%
  mutate(late_dep = dep_delay > 15)
```

```
flights
```

```
## # A tibble: 101,509 x 5
##   dep_delay carrier  hour origin late_dep
##       <dbl> <chr>    <dbl> <chr>   <lgl>
## 1      4 UA        5 LGA    FALSE
## 2     -6 DL        6 LGA    FALSE
## 3     -3 EV        6 LGA    FALSE
## 4     -2 AA        6 LGA    FALSE
```

```

##   5      -1 AA      6 LGA    FALSE
##   6       0 B6      6 LGA    FALSE
##   7       0 MQ      6 LGA    FALSE
##   8      -8 DL      6 LGA    FALSE
##   9      -3 MQ      6 LGA    FALSE
##  10      13 AA      6 LGA    FALSE
## # i 101,499 more rows

```

What proportion of all flights leaving from LaGuardia leave 15 minutes late or later? To check this, remember that `TRUE` is saved as a “1” and `FALSE` is saved as a “0”. That means that we can take the mean of a logical vector to get the proportion of trials that are true.

```
flights %>% pull("late_dep") %>% mean()
```

```
## [1] 0.1889685
```

Determine what proportion of all flights are delayed base on the hour that the flight was scheduled to depart (`hour`). Create a plot showing how the probability of leaving 15 minutes or more late changes by hour.

```

flights_late <- flights %>%
  group_by(hour) %>%
  summarize(prob_late = mean(late_dep))

```

```
flights_late
```

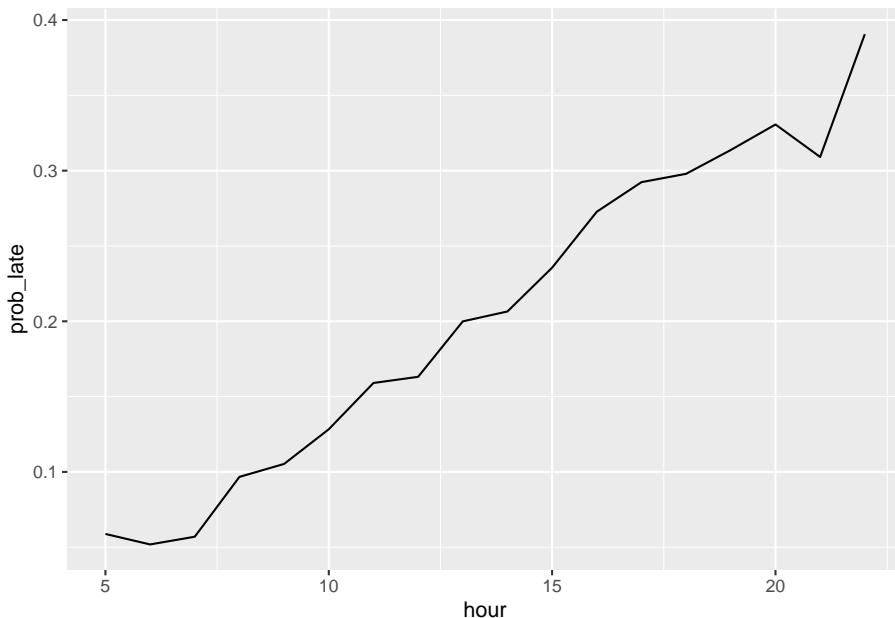
```

## # A tibble: 18 x 2
##       hour prob_late
##   <dbl>     <dbl>
## 1     5     0.0588
## 2     6     0.0519
## 3     7     0.0570
## 4     8     0.0967
## 5     9     0.105
## 6    10     0.128
## 7    11     0.159
## 8    12     0.163
## 9    13     0.200
## 10   14     0.207
## 11   15     0.236

```

```
## 12    16    0.273
## 13    17    0.292
## 14    18    0.298
## 15    19    0.314
## 16    20    0.331
## 17    21    0.309
## 18    22    0.391
```

```
flights_late %>%
  ggplot(aes(x = hour, y = prob_late)) +
  geom_line()
```



Fit a generalized linear model for the association between the binary variable of whether the flight was 15 minutes or more late (`late_dep`) and the hour the flight was scheduled to leave (`hour`). Use a binomial model (add `family = binomial(link = "logit")` in the `glm` call). The estimate from the model for `hour` will be an estimate of the log odds ratio for a one-hour increase in scheduled departure time. Take the exponent of this estimate with `exp` to get an estimated odds ratio for a one-hour increase in scheduled departure time. Is this estimate larger than 1.0?

```
glm(late_dep ~ hour, data = flights, family = binomial(link = "logit"))

##
## Call: glm(formula = late_dep ~ hour, family = binomial(link = "logit"),
##           data = flights)
##
## Coefficients:
## (Intercept)      hour
## -3.3643       0.1399
##
## Degrees of Freedom: 101508 Total (i.e. Null); 101507 Residual
## Null Deviance: 98410
## Residual Deviance: 92810      AIC: 92810
```

```
library(broom)
glm(late_dep ~ hour, data = flights, family = binomial(link = "logit")) %>%
  tidy() %>%
    # Tidy the model results
  filter(term == "hour") %>% # Only look at the estimate for `hour`
  mutate(or = exp(estimate)) # Estimate is log odds ratio. Take exponent for odds ratio

##
## # A tibble: 1 x 6
##   term  estimate std.error statistic p.value    or
##   <chr>    <dbl>     <dbl>     <dbl>    <dbl> <dbl>
## 1 hour     0.140    0.00195    71.7      0  1.15
```

Next, we want to see if this association is similar across airlines. First, create a dataframe called `nested_flights` where the `flights` data is grouped by airline (`carrier`) and then nest the data, so that there is a “data” list-column where each item is a dataframe of flight delay data for a specific carrier:

```
nested_flights <- flights %>%
  group_by(carrier) %>%
  nest()

nested_flights
```

```
##
## # A tibble: 13 x 2
## # Groups:   carrier [13]
##   carrier data
```

```

##   <chr>  <list>
## 1 UA    <tibble [7,837 x 4]>
## 2 DL    <tibble [22,857 x 4]>
## 3 EV    <tibble [8,255 x 4]>
## 4 AA    <tibble [15,063 x 4]>
## 5 B6    <tibble [5,925 x 4]>
## 6 MQ    <tibble [16,189 x 4]>
## 7 WN    <tibble [6,000 x 4]>
## 8 FL    <tibble [3,187 x 4]>
## 9 US    <tibble [12,574 x 4]>
## 10 F9   <tibble [682 x 4]>
## 11 9E   <tibble [2,372 x 4]>
## 12 YV   <tibble [545 x 4]>
## 13 00   <tibble [23 x 4]>

```

To check the contents of the list-column, try:

```

nested_flights$data[[1]] %>% # Get the first element of the "data" column
  head()

## # A tibble: 6 x 4
##   dep_delay hour origin late_dep
##       <dbl> <dbl> <chr>   <lgl>
## 1        4     5 LGA     FALSE
## 2       -4     6 LGA     FALSE
## 3        1     6 LGA     FALSE
## 4        9     7 LGA     FALSE
## 5       -4     7 LGA     FALSE
## 6        2     7 LGA     FALSE

```

Use the `map` function from `purrr` inside a `mutate` statement to apply the `glm` code you used earlier for the whole dataset, but in this case for the data for each airline character:

```

library(purrr)
library(tidyr)

prob_late <- nested_flights %>%
  mutate(glm_result = purrr::map(data, ~ glm(late_dep ~ hour,
                                              data = .x, family = binomial(link = "logit"))))

# Check the results for the first element of the "glm_result" column:
prob_late$glm_result[[1]]

```

```
## 
## Call: glm(formula = late_dep ~ hour, family = binomial(link = "logit"),
##           data = .x)
## 
## Coefficients:
## (Intercept)      hour
## -3.4305       0.1551
## 
## Degrees of Freedom: 7836 Total (i.e. Null); 7835 Residual
## Null Deviance:    7601
## Residual Deviance: 7064 AIC: 7068
```

Then use the `map` function inside a `mutate` statement again to “tidy” the data.

```
prob_late <- prob_late %>%
  mutate(glm_tidy = purrr::map(glm_result, ~ tidy(.x)))
```

Remove the columns for `data` and `glm_result`:

```
prob_late <- prob_late %>%
  select(-data, -glm_result)
```

Then ‘`unnest`’ the dataframe list-column with the tidy version of the model results:

```
prob_late <- prob_late %>%
  unnest(glm_tidy)
```

Filter to get only the estimates for the “hour” term:

```
prob_late <- prob_late %>%
  filter(term == "hour")
```

Then calculate an odds ratio (`or`) by taking the exponent (check the `exp` function) of the original estimate.

```
prob_late <- prob_late %>%
  mutate(or = exp(estimate))
head(prob_late)
```

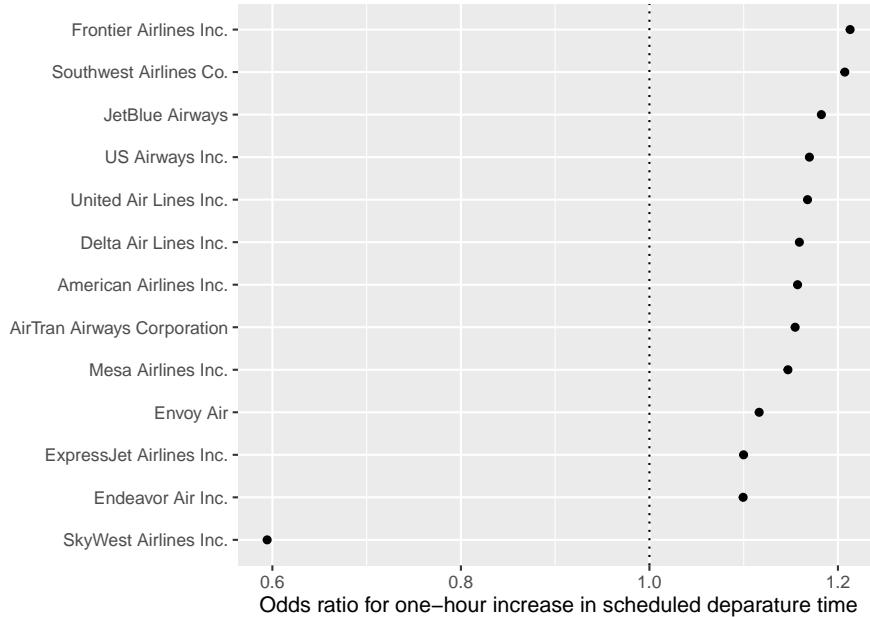
```
## # A tibble: 6 x 7
## # Groups:   carrier [6]
##   carrier term  estimate std.error statistic   p.value     or
##   <chr>    <chr>     <dbl>     <dbl>     <dbl>     <dbl> <dbl>
## 1 UA       hour      0.155    0.00705    22.0  2.88e-107 1.17
## 2 DL       hour      0.148    0.00454    32.6  1.80e-232 1.16
## 3 EV       hour      0.0952   0.00572    16.7  2.79e- 62 1.10
## 4 AA       hour      0.146    0.00580    25.2  1.05e-139 1.16
## 5 B6       hour      0.168    0.00702    23.9  7.13e-126 1.18
## 6 MQ       hour      0.110    0.00468    23.5  2.23e-122 1.12
```

The package has a dataframe with the full name of each carrier (`airlines`). Join this data into the data you've been working with, so you have the full names of airlines:

```
prob_late <- left_join(prob_late, airlines, by = "carrier")
```

Finally, create the following plot with each airline's association between hour in the day and the chance of a delay:

```
library(forcats)
data(airlines)
prob_late %>%
  ungroup() %>%
  mutate(estimate = exp(estimate)) %>%
  mutate(name = fct_reorder(name, estimate)) %>%
  ggplot(aes(x = estimate, y = name)) +
  geom_point() +
  geom_vline(xintercept = 1, linetype = 3) +
  labs(x = "Odds ratio for one-hour increase in scheduled departure time", y = "")
```



7.8.7 Writing functions

- Say that you have a four-letter character string (e.g., “ling”) and that you want to move the last letter to the front of the string to create a new four-letter character string (e.g., “glin”). Write some R code to do this.
- Next, write a function named `move_letter` that does the same thing—takes a four-letter character string (e.g., “ling”) and creates a new four-letter character string where the last letter in the original string has been moved to the front of the string (e.g., “glin”). It should input one parameter (`word`, the original four-letter character string or a vector of four-letter character strings).
- Read the word list at <https://raw.githubusercontent.com/dwyl/english-words/master/words.txt> into an R dataframe called `word_list`. It will only have one column; name this column `word`.
- Write code that can take a vector of character strings (e.g., `c('ling', 'scat', 'soil')`) and return a logical vector that says whether that character string is a word in the `word_list` dataframe you created in the last step (e.g., `c(FALSE, TRUE, TRUE)` for a vector where the first value isn’t a word but the second and third are). You may find the `pull` function useful in writing this code (to pull the `word` column out of the `word_list` dataframe).
- Write a function called `is_word` that inputs (1) a vector of character strings and (2) a dataframe with a column called “word” that lists real words. The function should return a logical vector saying whether each

character string is a real word. The function should have two arguments: `words_to_check`, which is a vector of character strings (e.g., `c('ling', 'scat', 'soil')`), and `real_word_list`, which is a dataframe with a column called `words` of real English words (e.g., the `word_list` dataframe you created from the word list on GitHub). Set the `real_word_list` argument to have the default value of `words`, the dataframe you created earlier in this exercise by reading in the dataframe of English words from GitHub.

- Try using the function with a different word list. As an example, you could read in and use the word list of Google's top 10,000 English words from <https://raw.githubusercontent.com/first20hours/google-10000-english/master/google-10000-english-no-swears.txt>.
 - Try using these functions to solve the word puzzle problem in the last homework.
-

7.8.7.1 Example R code

Say that you have a four-letter character string (e.g., “ling”) and that you want to move the last letter to the front of the string to create a new four-letter character string (e.g., “glin”). You can use functions from the `stringr` package to help with this.

```
library(stringr)

# Start with an example word. You can use the example word from
# the problem statement ('ling').
word <- "ling"

# Break the word into two parts
first_three_letters <- str_sub(word, 1, 3)
last_letter <- str_sub(word, 4, 4)

first_three_letters

## [1] "lin"

last_letter

## [1] "g"
```

```
# Put the parts back together in the right order
```

```
new_word <- str_c(last_letter, first_three_letters) # You could also use `paste` or `p
```

```
new_word
```

```
## [1] "glin"
```

Write a function named `move_letter` that takes a four-letter character string (e.g., “ling”) and creates a new four-letter character string where the last letter in the original string has been moved to the front of the string (e.g., “glin”). It should input one parameter (`word`, the original four-letter character string or a vector of four-letter character strings).

To do this, take the code that you just wrote and put it inside a function named `move_letter`:

```
move_letter <- function(word){
  # Break the word into two parts
  first_three_letters <- str_sub(word, 1, 3)
  last_letter <- str_sub(word, 4, 4)

  # Put the parts back together in the right order
  str_c(last_letter, first_three_letters)
}
```

```
# Check the function
move_letter(word = "ling")
```

```
## [1] "glin"
```

```
# Try with some other four-letter strings
```

```
move_letter(word = "cats")
```

```
## [1] "scat"
```

```
move_letter(word = "oils")
```

```
## [1] "soil"
```

```
# Check using with a vector of four-letter character strings
move_letter(c("cats", "oils"))

## [1] "scat" "soil"
```

Notice that now `word` is being used as an argument in the function. It's as if you assigned the string or vector of strings that you want to convert to the object name `word`, and then you run all the code. The output of the function is the last expression in the function code (`str_c(last_letter, first_three_letters)`). Also, note that you can add comments inside the function with `#`, just like you can with other R code.

Read the word list at <https://raw.githubusercontent.com/dwyl/english-words/master/words.txt> into an R dataframe called `words`. It will only have one column; name this column `word_list`.

```
library(readr)

word_list <- read_csv("https://raw.githubusercontent.com/dwyl/english-words/master/words.txt",
                      col_names = "word")

## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

## Rows: 466550 Columns: 1
## -- Column specification -----
## Delimiter: ","
## chr (1): word
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Write code that can take a vector of character strings (e.g., `c('ling', 'scat', 'soil')`) and return a logical vector that says whether that character string is a word in the `word_list` dataframe you created in the last step (e.g., `c('TRUE', 'FALSE', 'TRUE')` for a vector where the second value isn't a word but the first and third are).

```
library(purrr)

words_to_check <- c("ling", "scat", "soil")
words_to_check %in% pull(word_list, "word")
```

```
## [1] TRUE FALSE TRUE
```

This code works because you can use `pull` to extract a column (as a vector) from a dataframe, and then you can use the `%in%` operator to see if each value in one vector (the `words_to_check` vector in this example) is one of the values in the second vector (the `word` column from the `words` dataframe in this example).

Write a function called `is_word` that inputs (1) a vector of character strings and (2) a dataframe with a column called “word” that lists real words. The function should return a logical vector saying whether each character string is a real word. The function should have two arguments: `words_to_check`, which is a vector of character strings and `real_word_list` which is a dataframe with a column called `words` of real English words. Set the `real_word_list` argument to have the default value of `words`, the dataframe you created earlier in this exercise by reading in the dataframe of English words from GitHub.

```
# Put the code you wrote inside a function
is_word <- function(words_to_check, real_word_list = word_list){
  words_to_check %in% pull(real_word_list, "word")
}

# Check the function
is_word(words_to_check = c("ling", "scat", "soil"))
```

```
## [1] TRUE FALSE TRUE
```

Note that, if you want to use the `word_list` dataframe for your real word list, you don’t have to specify that when you call the `is_word` function, since you set that as your default value.

If you wanted to use a different word list, you can specify a different value for the `real_word_list` argument when you run the function. For example, to use Google’s top 10,000 English word list from GitHub instead, use:

```
google_words <- read_csv("https://raw.githubusercontent.com/first20hours/google-10000-"
                          "en-us/1.0/words.txt",
                          col_names = "word")
```

```
## Rows: 9894 Columns: 1
## -- Column specification -----
## Delimiter: ","
## chr (1): word
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
is_word(words_to_check = c("ling", "scat", "soil"), real_word_list = google_words)
```

```
## [1] FALSE FALSE TRUE
```

In this case, there was a rarer word (“ling”) that counted as a word when the original word list was used, but not when the Google top-10,000 words list was used.

Try using these functions to solve the word puzzle problem in the last homework.

```
word_list %>%
  filter(str_detect(word, "^.{7}$")) %>%
  separate(word, into = c("first_word", "second_word"),
           sep = 4, remove = FALSE) %>%
  mutate(second_word = move_letter(second_word)) %>%
  filter(is_word(first_word) & is_word(second_word)) %>%
  unite("new_word", c("first_word", "second_word"), sep = " ")
```

This is using regular expressions
Note that you can use `separate`
Use the first function here
Use the second function here
You can use `unite` to put them together

```
## # A tibble: 103 x 2
##   word     new_word
##   <chr>    <chr>
## 1 mailings mail sing
## 2 maillots mail slot
## 3 majorate majo erat
## 4 majoring majo grin
## 5 malmiest malm ties
## 6 maltiest malt ties
## 7 mandalas mand sala
## 8 mandated mand date
## 9 mandates mand sate
## 10 mangiest mang ties
## # i 93 more rows
```

You can get the choices down to even fewer options if you match the new words against the Google top-10,000 words list, by using the `real_word_list` option in the `is_word` function you wrote. (However, “maillots” is not a common enough word that you can also start from that shorter word list!)

```
word_list %>%
  filter(str_detect(word, "^\w{7}$")) %>%
  separate(word, into = c("first_word", "second_word"),
           sep = 4, remove = FALSE) %>%
  mutate(second_word = move_letter(second_word)) %>%
  filter(is_word(first_word, real_word_list = google_words) &
           is_word(second_word, real_word_list = google_words)) %>%
  unite("new_word", c("first_word", "second_word"), sep = " ")

## # A tibble: 43 x 2
##   word     new_word
##   <chr>    <chr>
## 1 mailings mail sing
## 2 maillots mail slot
## 3 maintops main stop
## 4 markings mark sing
## 5 maskings mask sing
## 6 massager mass rage
## 7 massages mass sage
## 8 massiest mass ties
## 9 mattings matt sing
## 10 mealiest meal ties
## # i 33 more rows
```

Chapter 8

Reporting results #2

Download a pdf of the lecture slides covering this topic.

8.1 Course lectures

In 2019, we cancelled our in-course meeting for bad weather.

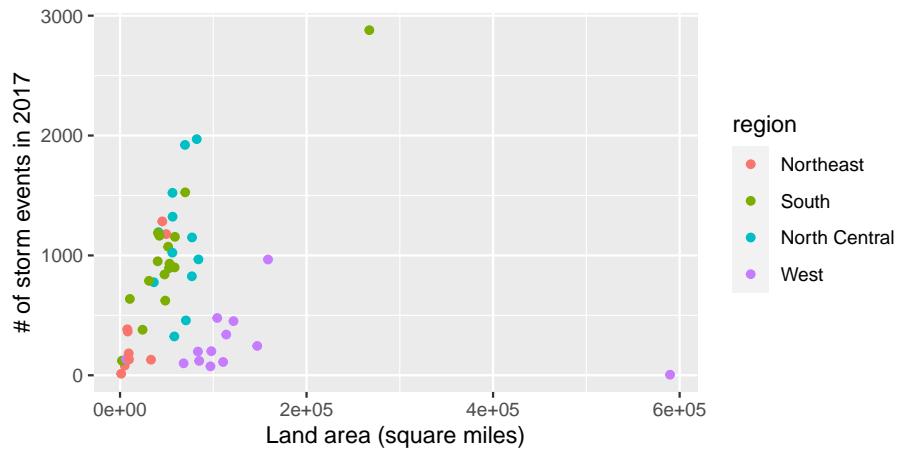
There are videos of the lecture available, covering ggplot2 extensions and mapping in ggplot:

Example commit.

8.2 Example data

This week, we'll be using some example data from NOAA's Storm Events Database. This data lists major weather-related storm events during 2017. For each event, it includes information like the start and end dates, where it happened, associated deaths, injuries, and property damage, and some other characteristics.

See the in-course exercises for this week for more on getting and cleaning this data. As part of the in-course exercise, you'll be making the following plot and saving it as the object `storm_plot`:



We'll be using the data and this plot in the next sections.

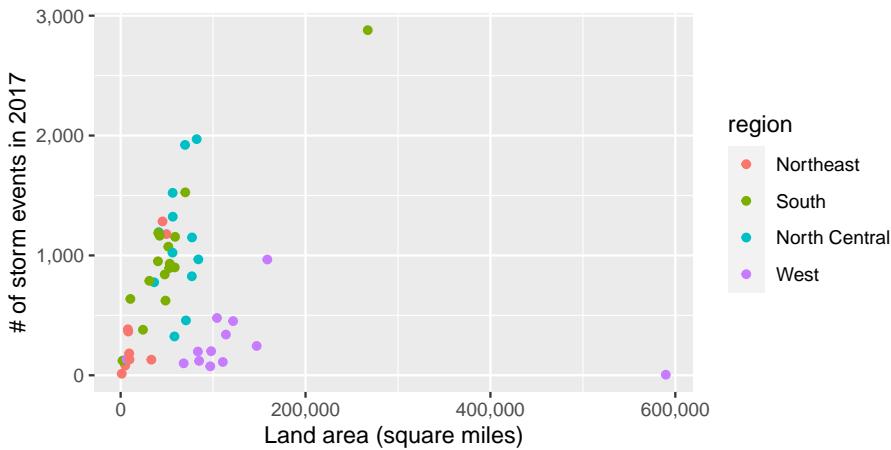
8.3 ggplot2 extras and extensions

8.3.1 scales package

The `scales` package gives you a few more options for labeling with your `ggplot` scales. For example, if you wanted to change the notation for the axes in the plot of state area versus number of storm events, you could use the `scales` package to add commas to the numeric axis values.

For the rest of these slides, I've saved the `ggplot` object with out plot to the object named `storm_plot`, so we don't have to repeat that code every time.

```
library(scales)
storm_plot +
  scale_x_continuous(labels = comma) +
  scale_y_continuous(labels = comma)
```



The `scales` package also includes labeling functions for:

- dollars (`labels = dollar`)
- percent (`labels = percent`)

8.3.2 ggplot2 extensions

The `ggplot2` framework is set up so that others can create packages that “extend” the system, creating functions that can be added on as layers to a `ggplot` object. Some of the types of extensions available include:

- More themes
- Useful additions (things that you may be able to do without the package, but that the package makes easier)
- Tools for plotting different types of data

There is a gallery with links to `ggplot2` extensions at <https://exts.ggplot2.tidyverse.org/gallery/>. This list may not be exhaustive—there may be other extensions on CRAN or on GitHub that the package maintainer did not submit for this gallery.

8.3.3 More ggplot2 themes

You have already played around a lot with using `ggplot` themes to change how your graphs look. Several people have created packages with additional themes:

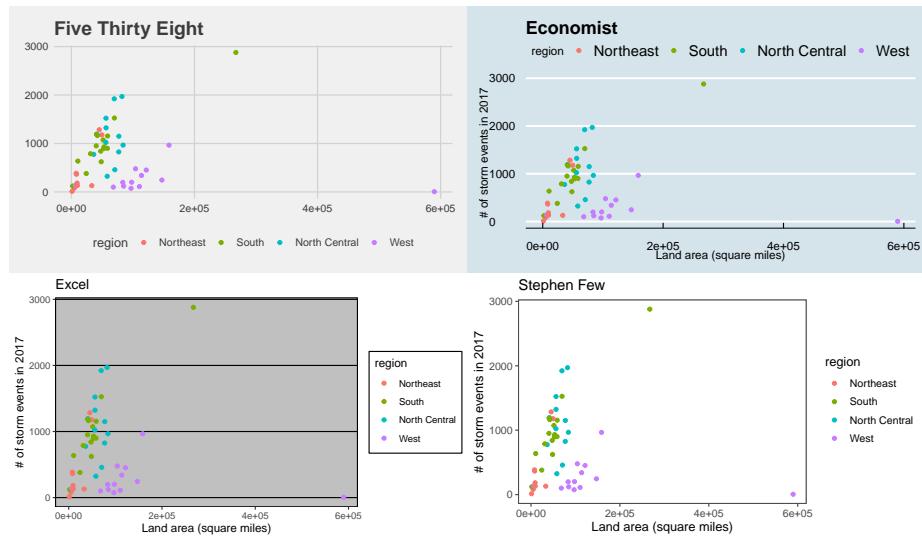
- `ggthemes`
- `ggthemr`

- `ggtech`
- `ggsci`

```
library(ggthemes)
library(gridExtra)

a <- storm_plot +
  theme_fivethirtyeight() +
  ggtitle("Five Thirty Eight")
b <- storm_plot +
  theme_economist() +
  ggtitle("Economist")
c <- storm_plot +
  theme_excel() +
  ggtitle("Excel")
d <- storm_plot +
  theme_few() +
  ggtitle("Stephen Few")

grid.arrange(a, b, c, d, ncol = 2)
```



8.3.4 Other useful `ggplot2` extensions

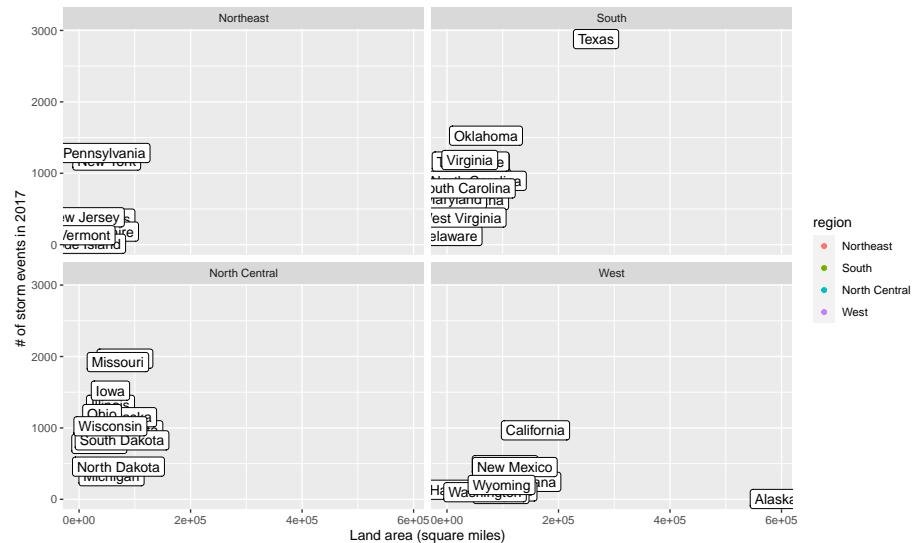
Other `ggplot2` extensions do things you might have been able to figure out how to do without the extension, but the extension makes it much easier to do. These tasks include:

- Highlighting interesting points
- “Repelling” text labels
- Arranging plots

8.3.4.1 Repelling / highlighting with text labels

The first is repelling text labels. When you add labels to points on a plot, they often overlap:

```
storm_plot + facet_wrap(~ region) +
  geom_label(aes(label = state))
```



The `ggrepel` package helps make sure that these labels don't overlap:

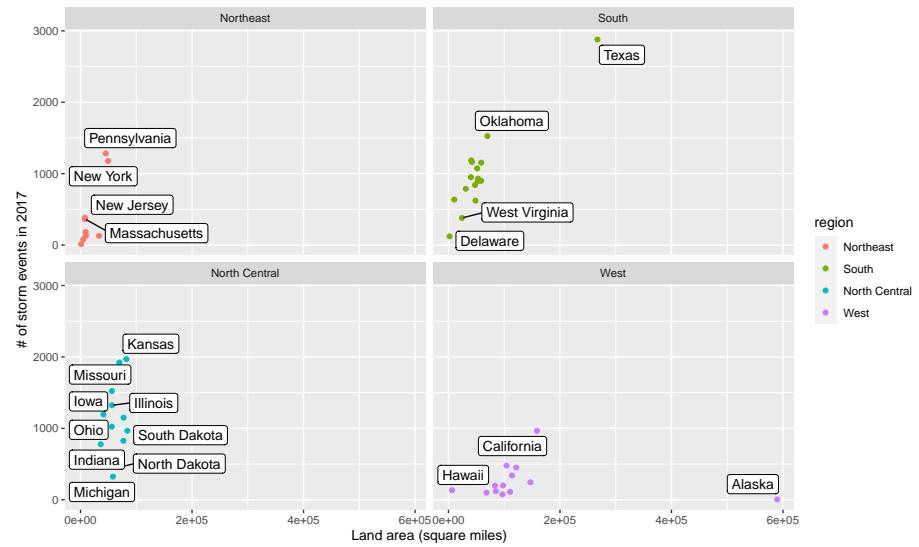
```
library(ggrepel)
storm_plot + facet_wrap(~ region) +
  geom_label_repel(aes(label = state))
```

```
## Warning: ggrepel: 5 unlabeled data points (too many overlaps). Consider
## increasing max.overlaps

## Warning: ggrepel: 3 unlabeled data points (too many overlaps). Consider
## increasing max.overlaps
```

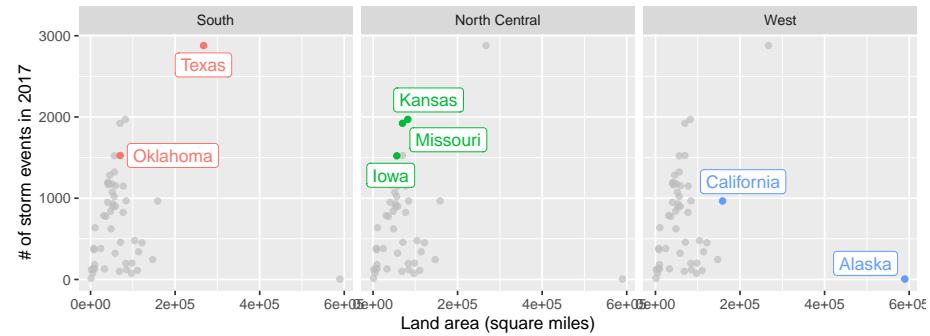
```
## Warning: ggrepel: 12 unlabeled data points (too many overlaps). Consider
## increasing max.overlaps
```

```
## Warning: ggrepel: 10 unlabeled data points (too many overlaps). Consider
## increasing max.overlaps
```



It may be too much to label every point. Instead, you may just want to highlight notable point. You can use the `gghighlight` package to do that.

```
library(gghighlight)
storm_plot + facet_wrap(~ region) +
  gghighlight(area > 150000 | n > 1500, label_key = state)
```

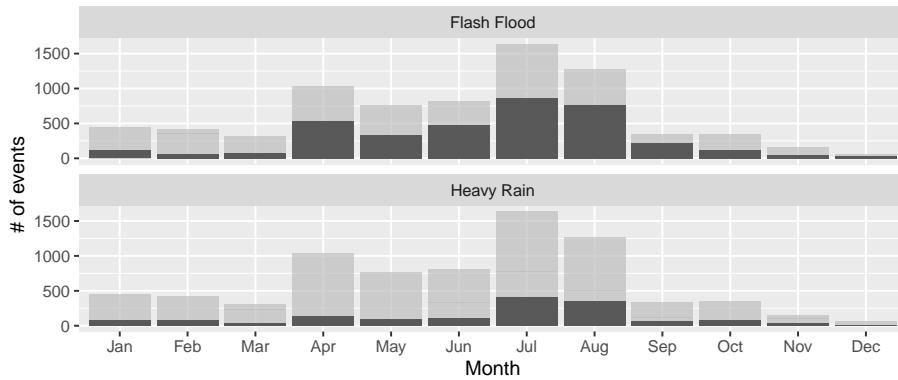


The `gghighlight` package also works for things like histograms. For example, you could create a dataset with the count by day-of-year of certain types of events:

```
storms_by_month <- storms_2017 %>%
  filter(event_type %in% c("Flood", "Flash Flood", "Heavy Rain")) %>%
  mutate(month = month(begin_date_time, label = TRUE)) %>%
  group_by(month, event_type) %>%
  count() %>%
  ungroup()
storms_by_month %>%
  slice(1:4)
```

```
## # A tibble: 4 x 3
##   month event_type     n
##   <ord> <chr>     <int>
## 1 Jan   Flash Flood    113
## 2 Jan   Flood          255
## 3 Jan   Heavy Rain     80
## 4 Feb   Flash Flood    65
```

```
ggplot(storms_by_month, aes(x = month, y = n, group = event_type)) +
  geom_bar(stat = "identity") +
  labs(x = "Month", y = "# of events") +
  gghighlight(max(n) > 400, label_key = event_type) +
  facet_wrap(~ event_type, ncol = 1)
```



8.3.4.2 Arranging plots

You may have multiple related plots you want to have as multiple panels of a single figure. There are a few packages that help with this. One very good one is `patchwork`. You need to install this from GitHub:

```
devtools::install_github("thomasp85/patchwork")
```

Find out more: <https://github.com/thomasp85/patchwork#patchwork>

Say we want to plot seasonal patterns in events in the five counties with the highest number of events in 2017. We can use `dplyr` to figure out these counties:

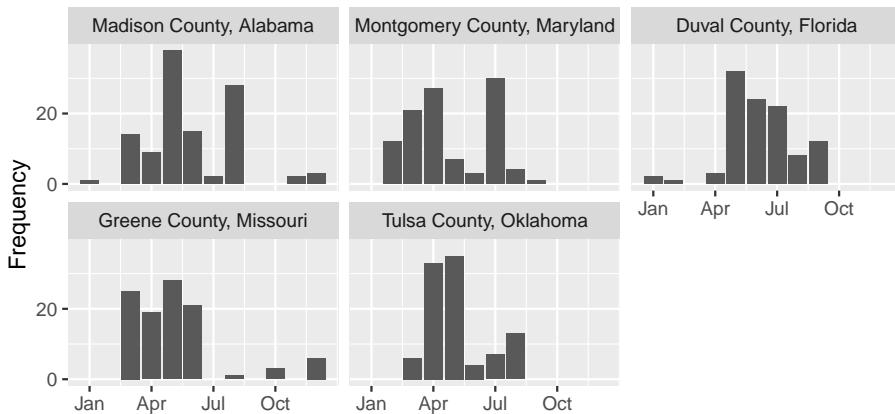
```
top_counties <- storms_2017 %>%
  group_by(fips, state, cz_name) %>%
  count() %>%
  ungroup() %>%
  top_n(5, wt = n)
```

Then create a plot with the time patterns:

```
library(forcats)
top_counties_month <- storms_2017 %>%
  semi_join(top_counties, by = "fips") %>%
  mutate(month = month(begin_date_time),
        county = paste(cz_name, " County, ", state, sep = "")) %>%
  count(county, month) %>%
  ggplot(aes(x = month, y = n)) +
  geom_bar(stat = "identity") +
  facet_wrap(~ fct_reorder(county, n, .fun = sum, .desc = TRUE), nrow = 2) +
  scale_x_continuous(name = "", breaks = c(1, 4, 7, 10),
                     labels = c("Jan", "Apr", "Jul", "Oct")) +
  scale_y_continuous(name = "Frequency", breaks = c(0, 20))
```

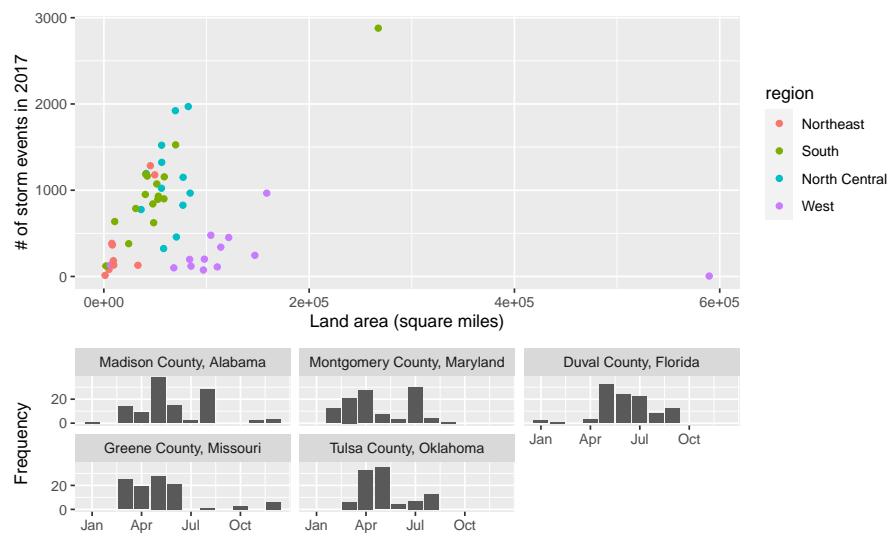
Here's this plot:

```
top_counties_month
```



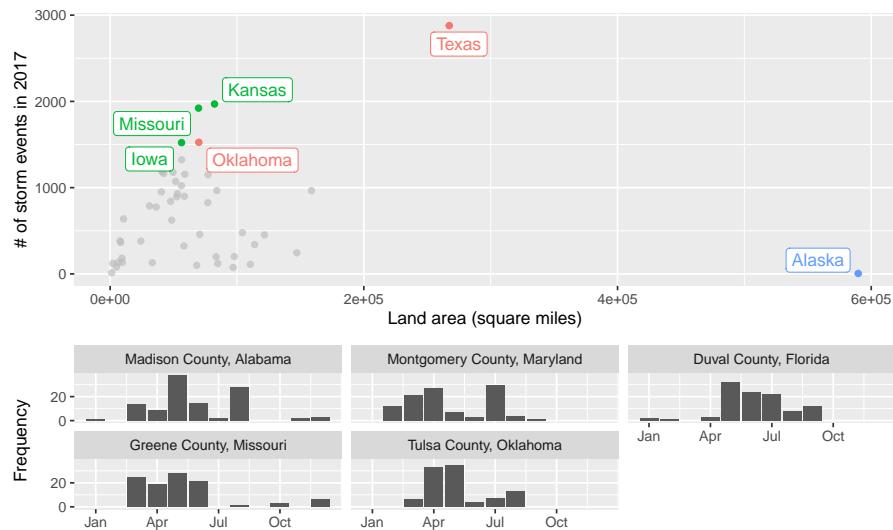
Now that you have two ggplot objects (`storm_plot` and `top_counties_month`), you can use `patchwork` to put them together:

```
library(patchwork)
storm_plot +
  top_counties_month +
  plot_layout(ncol = 1, heights = c(2, 1))
```



A slightly fancier version:

```
(storm_plot + theme(legend.position = "top") +
  gghighlight(n > 1500 | area > 200000,
               label_key = state)) +
  top_counties_month +
  plot_layout(ncol = 1, heights = c(2, 1))
```



Other packages for arranging `ggplot` objects include:

- `gridExtra`
- `cowplot`

8.4 Simple features

8.4.1 Introduction to simple features

`sf` objects: “Simple features”

- R framework that is in active development
- There will likely be changes in the near future
- Plays very well with tidyverse functions, including `dplyr` and `ggplot2` tools

```
library(sf)
```

To show simple features, we'll pull in the Colorado county boundaries from the U.S. Census.

To do this, we'll use the `tigris` package, which accesses the U.S. Census API. It allows you to pull geographic data for U.S. counties, states, tracts, voting districts, roads, rails, and a number of other geographies.

To learn more about the `tigris` package, check out this article: <https://journal.r-project.org/archive/2016/RJ-2016-043/index.html>

With `tigris`, you can read in data for county boundaries using the `counties` function.

We'll use the option `class = "sf"` to read these spatial dataframes in as `sf` objects.

```
library(tigris)
co_counties <- counties(state = "CO", cb = TRUE, class = "sf")
```

```
class(co_counties)
```

```
## [1] "sf"           "data.frame"
```

You can think of an `sf` object as a dataframe, but with one special column called `geometry`.

```
co_counties %>%
  slice(1:3)
```

```
## Simple feature collection with 3 features and 12 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -108.3811 ymin: 36.99902 xmax: -105.0487 ymax: 39.92525
## Geodetic CRS:   NAD83
##   STATEFP COUNTYFP COUNTYNS      AFFGEOID GEOID      NAME      NAMELSAD
## 1       08      037 00198134 0500000US08037 08037    Eagle    Eagle County
## 2       08      059 00198145 0500000US08059 08059 Jefferson Jefferson County
## 3       08      067 00198148 0500000US08067 08067 La Plata La Plata County
```

```
##   STUSPS STATE_NAME LSAD      ALAND     AWATER      geometry
## 1      CO  Colorado    06 4362754228 18970639 MULTIPOLYGON (((-107.1137 3...
## 2      CO  Colorado    06 1979735379 25071495 MULTIPOLYGON (((-105.0558 3...
## 3      CO  Colorado    06 4376255277 25642579 MULTIPOLYGON (((-108.3796 3...
```

The `geometry` column has a special class (`sfc`):

```
class(co_counties$geometry)

## [1] "sfc_MULTIPOLYGON" "sfc"
```

You'll notice there's some extra stuff up at the top, too:

- **Geometry type:** Points, polygons, lines
- **Dimension:** Often two-dimensional, but can go up to four (if you have, for example, time for each measurement and some measure of measurement error / uncertainty)
- **Bounding box (bbox):** The x- and y-range of the data included
- **EPSG:** The EPSG Geodetic Parameter Dataset code for the Coordinate Reference Systems
- **Projection (proj4string):** How the data is currently projected, includes projection (“+proj”) and datum (“+datum”)

You can pull some of this information out of the `geometry` column. For example, you can pull out the coordinates of the bounding box:

```
st_bbox(co_counties$geometry)      # For all counties
```

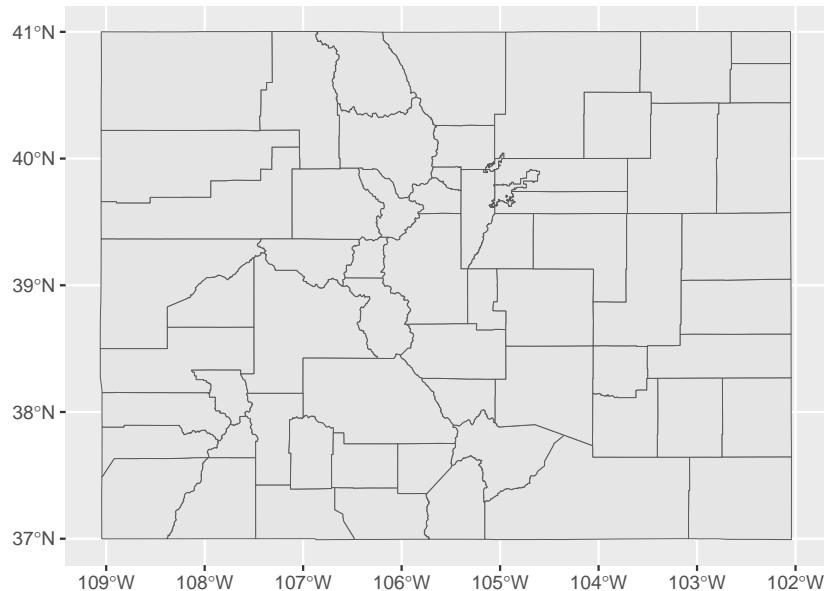
```
##           xmin         ymin         xmax         ymax
## -109.06025  36.99243 -102.04152  41.00344
```

```
st_bbox(co_counties$geometry[1]) # Just for first county
```

```
##           xmin         ymin         xmax         ymax
## -107.11395  39.34947 -106.17592  39.92525
```

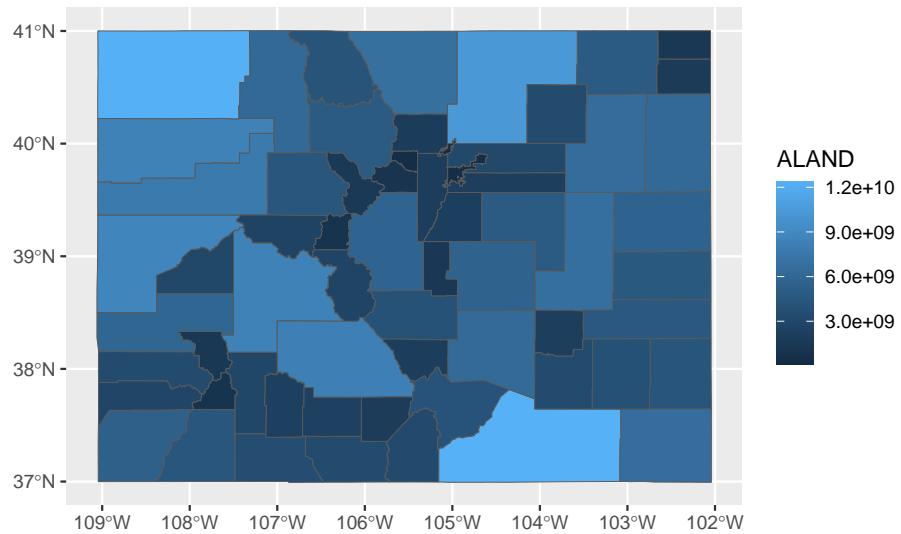
You can add `sf` objects to `ggplot` objects using `geom_sf`:

```
library(ggplot2)
ggplot() +
  geom_sf(data = co_counties)
```



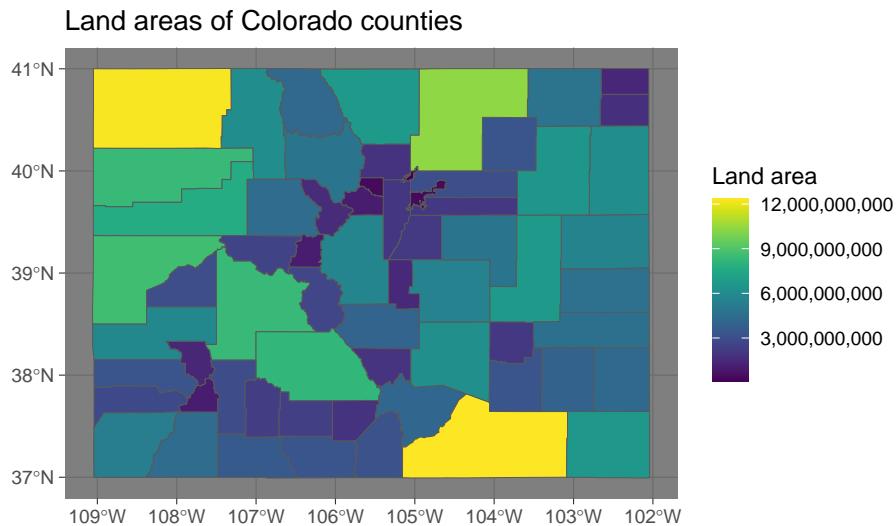
You can map one of the columns in the `sf` object to the fill aesthetic to make a **choropleth**:

```
ggplot() +
  geom_sf(data = co_counties, aes(fill = ALAND))
```



You can use all your usual ggplot tricks with this:

```
library(viridis)
ggplot() +
  geom_sf(data = co_counties, aes(fill = ALAND)) +
  scale_fill_viridis(name = "Land area", label = comma) +
  ggtitle("Land areas of Colorado counties") +
  theme_dark()
```



Because simple features are a special type of dataframe, you can also use a lot of `dplyr` tricks.

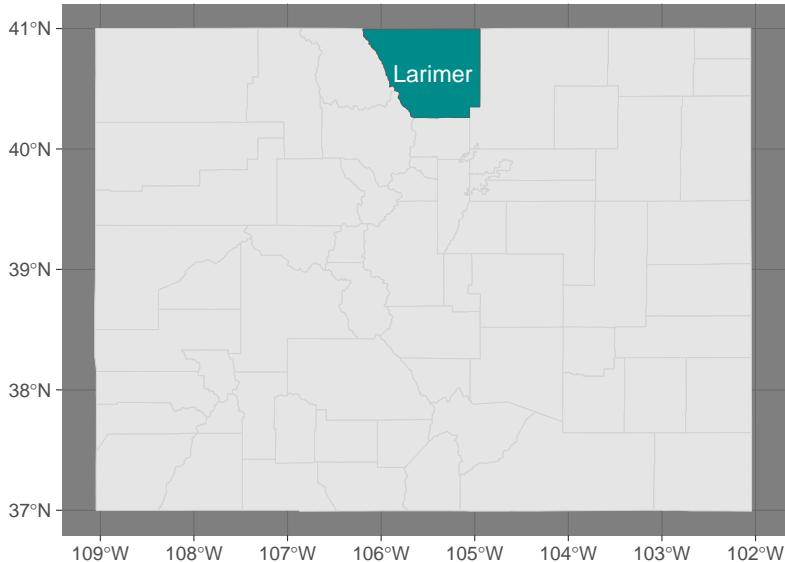
For example, you could pull out just Larimer County, CO:

```
larimer <- co_counties %>%
  filter(NAME == "Larimer")
larimer

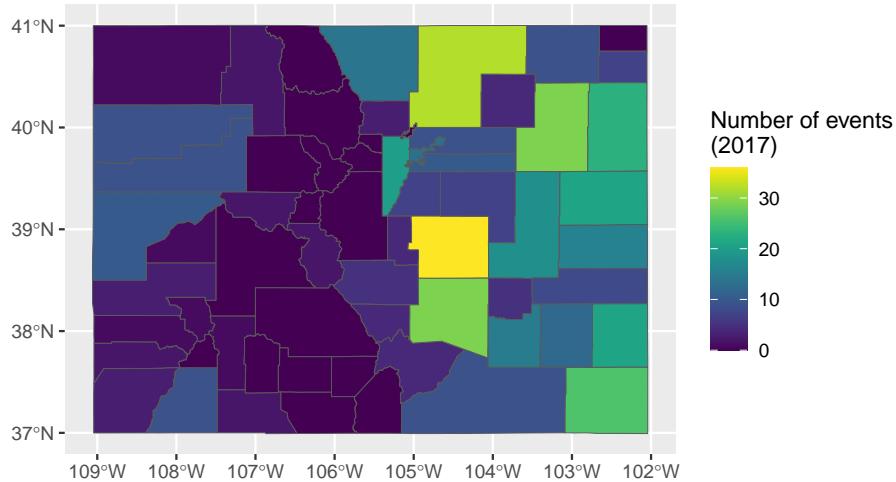
## Simple feature collection with 1 feature and 12 fields
## Geometry type: MULTIPOLYGON
## Dimension: XY
## Bounding box: xmin: -106.1954 ymin: 40.25788 xmax: -104.9431 ymax: 40.99821
## Geodetic CRS: NAD83
## STATEFP COUNTYFP COUNTYNS      AFFGEOID GEOID      NAME      NAMELSAD STUSPS
## 1      08      069 00198150 05000000US08069 08069 Larimer Larimer County      CO
## STATE_NAME LSAD      ALAND     AWATER                      geometry
## 1  Colorado   06 6723017943 98980718 MULTIPOLYGON (((-106.1954 4...
```

Note: You may need the development version of `ggplot2` for the next piece of code to work (`devtools::install_github("tidyverse/ggplot2")`).

```
ggplot() +  
  geom_sf(data = co_counties, color = "lightgray") +  
  geom_sf(data = larimer, fill = "darkcyan") +  
  geom_sf_text(data = larimer, aes(label = NAME), color = "white") +  
  theme_dark() + labs(x = "", y = "")
```



This operability with tidyverse functions means that you should now be able to figure out how to create a map of the number of events listed in the NOAA Storm Events database (of those listed by county) for each county in Colorado (for the code, see the in-course exercise):



8.4.2 State boundaries

The `tigris` package allows you to pull state boundaries, as well, but on some computers mapping these seems to take a really long time.

Instead, for now I recommend that you pull the state boundaries using base R's `maps` package and convert that to an `sf` object:

```
library(maps)

## 
## Attaching package: 'maps'

## The following object is masked from 'package:viridis':
## 
##     unemp

## The following object is masked from 'package:faraway':
## 
##     ozone

## The following object is masked from 'package:purrr':
## 
##     map
```

```
us_states <- map("state", plot = FALSE, fill = TRUE) %>%
  st_as_sf()
```

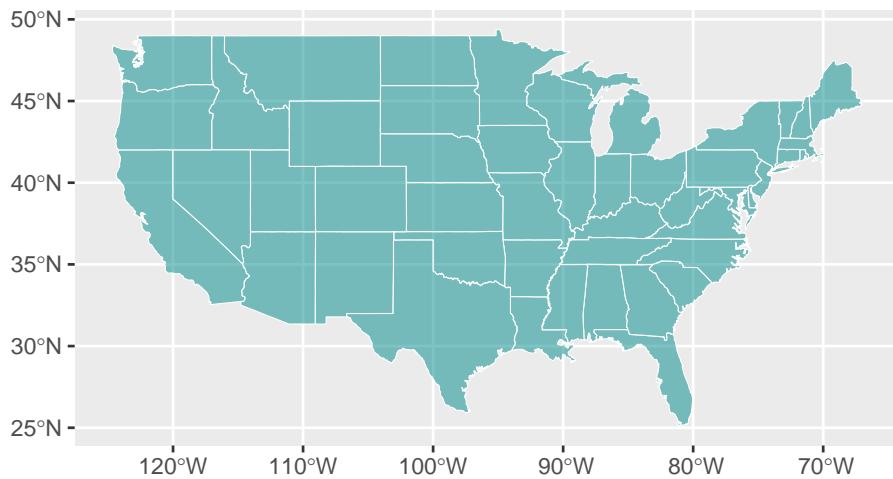
You can see these borders include an ID column that you can use to join by state:

```
head(us_states)
```

```
## Simple feature collection with 6 features and 1 field
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -124.3834 ymin: 30.24071 xmax: -71.78015 ymax: 42.04937
## Geodetic CRS:   +proj=longlat +ellps=clrk66 +no_defs +type=crs
##           ID                  geom
## 1    alabama MULTIPOLYGON (((-87.46201 3...
## 2    arizona MULTIPOLYGON (((-114.6374 3...
## 3    arkansas MULTIPOLYGON (((-94.05103 3...
## 4    california MULTIPOLYGON (((-120.006 42...
## 5    colorado MULTIPOLYGON (((-102.0552 4...
## 6 connecticut MULTIPOLYGON (((-73.49902 4...
```

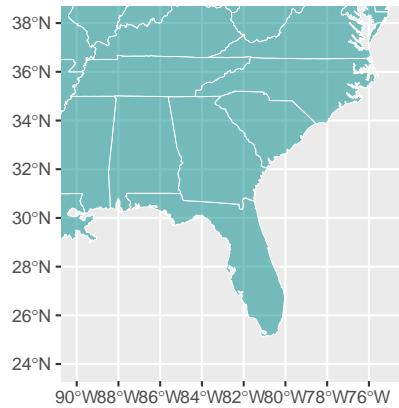
As with other `sf` objects, you can map these state boundaries using `ggplot`:

```
ggplot() +
  geom_sf(data = us_states, color = "white",
          fill = "darkcyan", alpha = 0.5)
```



As a note, you can use `xlim` and `ylim` with these plots, but remember that the x-axis is longitude in degrees West, which are negative:

```
ggplot() +
  geom_sf(data = us_states, color = "white",
          fill = "darkcyan", alpha = 0.5) +
  xlim(c(-90, -75)) + ylim(c(24, 38))
```



8.4.3 Creating an `sf` object

You can create an `sf` object from a regular dataframe.

You just need to specify:

1. The coordinate information (which columns are longitudes and latitudes)
2. The Coordinate Reference System (CRS) (how to translate your coordinates to places in the world)

For the CRS, if you are mapping the new `sf` object with other, existing `sf` objects, make sure that you use the same CRS for all `sf` objects.

Spatial objects can have different Coordinate Reference Systems (CRSs). CRSs can be *geographic* (e.g., WGS84, for longitude-latitude data) or *projected* (e.g., UTM, NAD83).

There is a website that lists projection strings and can be useful in setting projection information or re-projecting data: <http://www.spatialreference.org>

Here is an excellent resource on projections and maps in R from Melanie Frazier: <https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/OverviewCoordinateReferenceSystems.pdf>

Let's look at floods in Colorado. First, clean up the data:

```
co_floods <- storms_2017 %>%
  filter(state == "Colorado" &
         event_type %in% c("Flood", "Flash Flood")) %>%
  select(begin_date_time, event_id, begin_lat:end_lon) %>%
  gather(key = "key", value = "value",
```

```
-begin_date_time, -event_id) %>%
separate(key, c("time", "key")) %>%
spread(key = key, value = value)
```

There are now two rows per event, one with the starting location and one with the ending location:

```
co_floods %>%
  slice(1:5)
```

```
## # A tibble: 5 x 5
##   begin_date_time     event_id time    lat    lon
##   <dttm>           <dbl> <chr> <dbl> <dbl>
## 1 2017-05-08 16:00:00 693374 begin  40.3 -105.
## 2 2017-05-08 16:00:00 693374 end   40.5 -104.
## 3 2017-05-10 15:00:00 686479 begin  38.1 -105.
## 4 2017-05-10 15:00:00 686479 end   38.1 -105.
## 5 2017-05-10 15:20:00 686480 begin  38.2 -105.
```

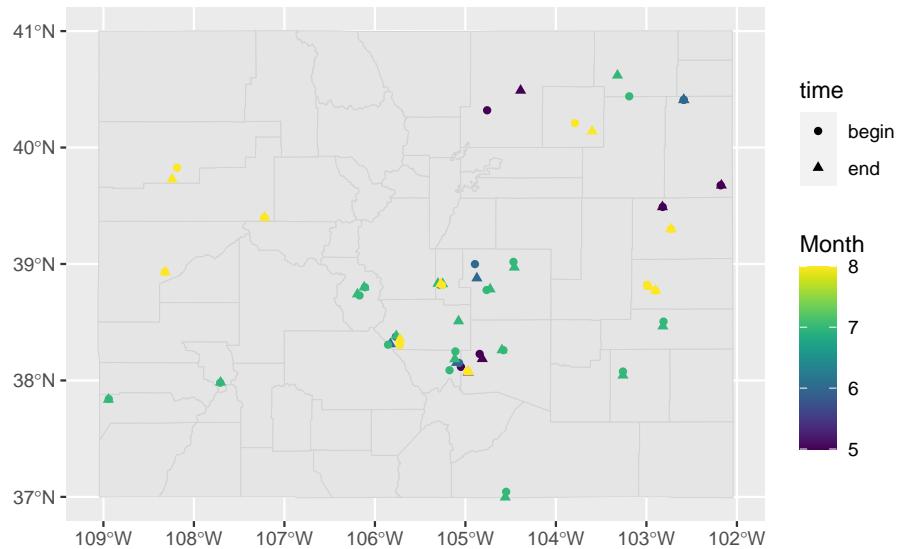
Change to an `sf` object by saying which columns are the coordinates and setting a CRS:

```
co_floods <- st_as_sf(co_floods, coords = c("lon", "lat")) %>%
  st_set_crs(4269)
co_floods %>% slice(1:3)
```

```
## Simple feature collection with 3 features and 3 fields
## Geometry type: POINT
## Dimension:      XY
## Bounding box:  xmin: -105.0496 ymin: 38.1167 xmax: -104.39 ymax: 40.49
## Geodetic CRS:  NAD83
## # A tibble: 3 x 4
##   begin_date_time     event_id time           geometry
##   <dttm>           <dbl> <chr>         <POINT [°]>
## 1 2017-05-08 16:00:00 693374 begin (-104.76 40.32)
## 2 2017-05-08 16:00:00 693374 end   (-104.39 40.49)
## 3 2017-05-10 15:00:00 686479 begin (-105.0496 38.1167)
```

Now you can map the data:

```
ggplot() +
  geom_sf(data = co_counties, color = "lightgray") +
  geom_sf(data = co_floods, aes(color = month(begin_date_time),
                                 shape = time)) +
  scale_color_viridis(name = "Month")
```



If you want to show lines instead of points, group by the appropriate ID and then summarize within each event to get a line:

```
co_floods <- co_floods %>%
  group_by(event_id) %>%
  summarize(month = month(first(begin_date_time)),
            do_union = FALSE) %>%
  st_cast("LINESTRING")
```

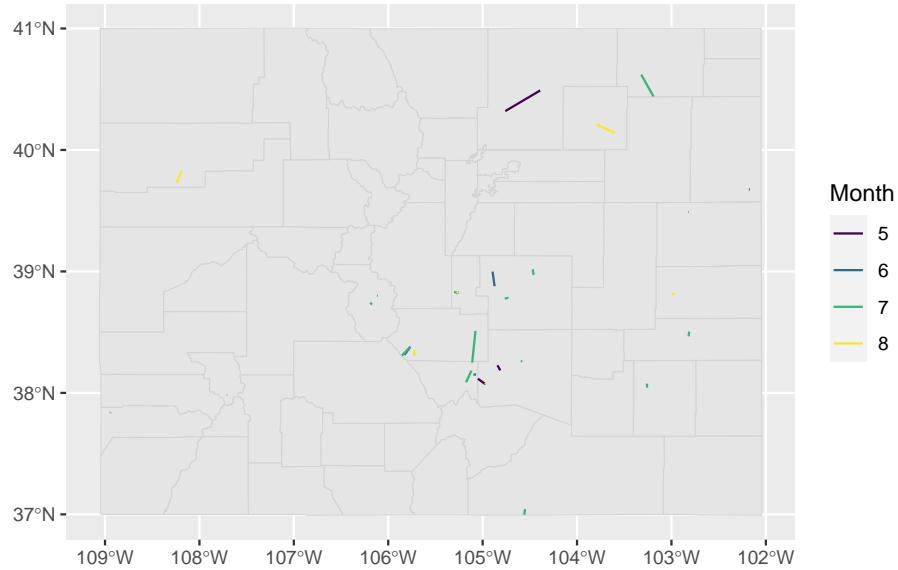
```
head(co_floods)
```

```
## Simple feature collection with 6 features and 2 fields
```

```
## Geometry type: LINESTRING
## Dimension: XY
## Bounding box: xmin: -105.8286 ymin: 38.0708 xmax: -104.39 ymax: 40.49
## Geodetic CRS: NAD83
## # A tibble: 6 x 3
##   event_id month               geometry
##       <dbl> <dbl> <LINESTRING [°]>
## 1     686479     5 (-105.0496 38.1167, -104.9687 38.0708)
## 2     686480     5 (-104.8425 38.2275, -104.8137 38.1854)
## 3     693306     6 (-104.8947 38.999, -104.8734 38.8783)
## 4     693374     5      (-104.76 40.32, -104.39 40.49)
## 5     693444     6 (-105.7688 38.3753, -105.8286 38.3127)
## 6     693449     6      (-105.07 38.15, -105.0973 38.1524)
```

Now this data will map as lines:

```
ggplot() +
  geom_sf(data = co_counties, color = "lightgray") +
  geom_sf(data = co_floods,
    aes(color = factor(month), fill = factor(month))) +
  scale_fill_viridis(name = "Month", discrete = TRUE) +
  scale_color_viridis(name = "Month", discrete = TRUE)
```



8.4.4 Reading in from GIS files

You can also create `sf` objects by reading in data from files you would normally use for GIS.

For example, you can read in an `sf` object from a shapefile, which is a format often used for GIS in which a collection of several files jointly store geographic data. The files making up a shapefile can include:

- “.shp”: The coordinates defining the shape of each geographic object. For a point, this would be a single coordinate (e.g., latitude and longitude). For lines and polygons, there will be multiple coordinates per geographic object.
- “.prf”: Information on the projection of the data (how to get from the coordinates to a place in the world).
- “.dbf”: Data that goes along with each geographical object. For example, earlier we looked at data on counties, and one thing measured for each county was its land area. Characteristics like that would be included in the “.dbf” file in a shapefile.

Often, with geographic data, you will be given the option to download a compressed file (e.g., a zipped file). When you unzip the folder, it will include a number of files in these types of formats (“.shp”, “.prf”, “.dbf”, etc.).

Sometimes, that single folder will include multiple files from each extension. For example, it might have several files that end with “.shp”. In this case, you have multiple **layers** of geographic information you can read in.

We've been looking at data on storms from NOAA for 2017. As an example, let's try to pair that data up with some from the National Hurricane Center for the same year.

The National Hurricane Center allows you to access a variety of GIS data through the webpage <https://www.nhc.noaa.gov/gis/?text>.

Let's pull some data on Hurricane Harvey in 2017 and map it with information from the NOAA Storm Events database.

On <https://www.nhc.noaa.gov/gis/?text>, go to the section called “Preliminary Best Track”. Select the year 2017. Then select “Hurricane Harvey” and download “al092017_best_track.zip”.

Depending on your computer, you may then need to unzip this file (many computers will unzip it automatically). Base R has a function called `unzip` that can help with this.

You'll then have a folder with a number of different files in it. Move this folder somewhere that is convenient for the working directory you use for class. For example, I moved it into the “data” subdirectory of the working directory I use for the class.

You can use `list.files` to see all the files in this unzipped folder:

```
list.files("data/al092017_best_track/")

## [1] "al092017_lin.dbf"           "al092017_lin.prj"
## [3] "al092017_lin.shp"          "al092017_lin.shp.xml"
## [5] "al092017_lin.shx"          "al092017_pts.dbf"
## [7] "al092017_pts.prj"          "al092017_pts.shp"
## [9] "al092017_pts.shp.xml"      "al092017_pts.shx"
## [11] "al092017_radii.dbf"        "al092017_radii.prj"
## [13] "al092017_radii.shp"        "al092017_radii.shp.xml"
## [15] "al092017_radii.shx"        "al092017_winds swath.dbf"
## [17] "al092017_winds swath.prj"  "al092017_winds swath.shp"
## [19] "al092017_winds swath.shp.xml" "al092017_winds swath.shx"
```

You can use `st_layers` to find out the available layers in a shapefile directory:

```
st_layers("data/al092017_best_track/")
```

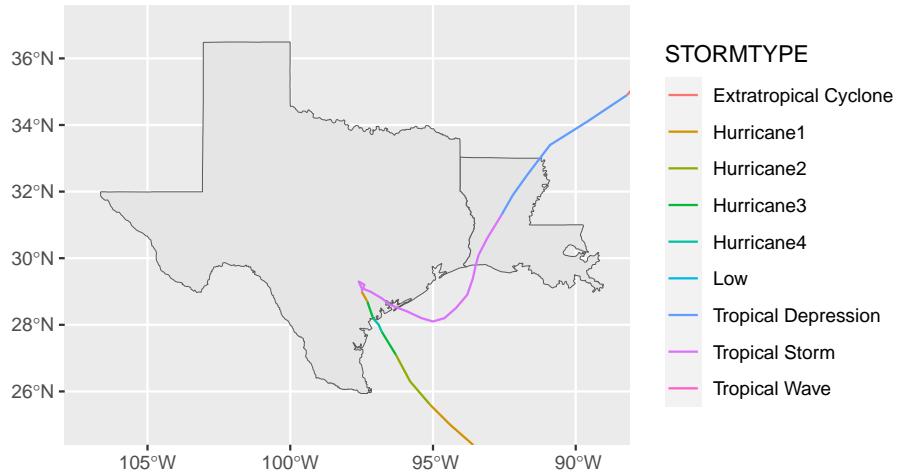
```
## Driver: ESRI Shapefile
## Available layers:
##   layer_name geometry_type features fields
## 1 al092017_lin   Line String      17      3
## 2 al092017_pts    Point          74     15
## 3 al092017_radii   Polygon        61      9
## 4 al092017_windswath Polygon        4      6
##                                     crs_name
## 1 Unknown datum based upon the Authalic Sphere
## 2 Unknown datum based upon the Authalic Sphere
## 3 Unknown datum based upon the Authalic Sphere
## 4 Unknown datum based upon the Authalic Sphere
```

Once you know which layer you want, you can use `read_sf` to read it in as an `sf` object:

```
harvey_track <- read_sf("data/al092017_best_track/",
                        layer = "al092017_lin")
head(harvey_track)
```

```
## Simple feature collection with 6 features and 3 fields
## Geometry type: LINESTRING
## Dimension: XY
## Bounding box: xmin: -92.3 ymin: 13 xmax: -45.8 ymax: 21.4
## Geodetic CRS: Unknown datum based upon the Authalic Sphere
## # A tibble: 6 x 4
##   STORMNUM STORMTYPE           SS                                geometry
##   <dbl> <chr>            <int>                            <LINESTRING [°]>
## 1 9     Low                0 (-45.8 13.7, -47.4 13.7, -49 13.6, -50.6 1~
## 2 9     Tropical Depression 0 (-52 13.4, -53.4 13.1, -55 13)
## 3 9     Tropical Storm     0 (-55 13, -56.6 13, -58.4 13, -59.6 13.1, --~
## 4 9     Tropical Depression 0 (-67.5 13.7, -69.2 13.8)
## 5 9     Tropical Wave      0 (-69.2 13.8, -71 14, -72.9 14.2, -75 14.4, ~
## 6 9     Low                0 (-89.7 20, -90.7 20.5, -91.6 20.9, -92.3 2~
```

```
ggplot() +
  geom_sf(data = filter(us_states, ID %in% c("texas", "louisiana"))) +
  geom_sf(data = harvey_track, aes(color = STORMTYPE)) +
  xlim(c(-107, -89)) + ylim(c(25, 37))
```

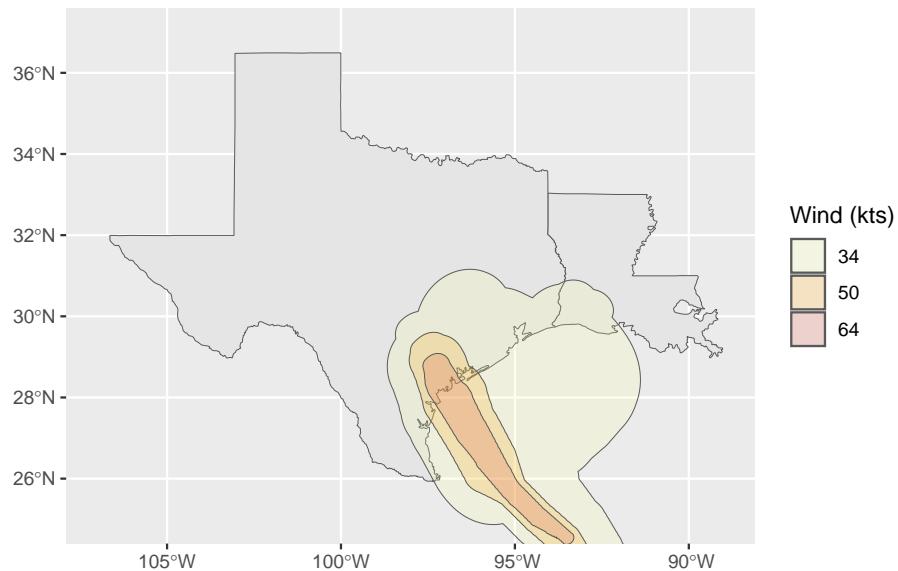


You can read in other layers:

```
harvey_windswath <- read_sf("data/al092017_best_track/",
                             layer = "al092017_windswath")
head(harvey_windswath)
```

```
## Simple feature collection with 4 features and 6 fields
## Geometry type: POLYGON
## Dimension:      XY
## Bounding box:  xmin: -98.66872 ymin: 12.94564 xmax: -54.58527 ymax: 31.15894
## Geodetic CRS:  Unknown datum based upon the Authalic Sphere
## # A tibble: 4 x 7
##   RADII STORMID BASIN STORMNUM STARTDTG     ENDDTG           geometry
##   <dbl> <chr>    <chr>    <dbl> <chr>    <chr>    <POLYGON [°]>
## 1     34 al092017 AL        9 2017081718 2017081906 ((-65.68199 14.50528,
## 2     34 al092017 AL        9 2017082318 2017083018 ((-96.22456 31.15752,
## 3     50 al092017 AL        9 2017082406 2017082618 ((-97.32707 29.60604,
## 4     64 al092017 AL        9 2017082418 2017082612 ((-97.20689 29.08475,
```

```
ggplot() +
  geom_sf(data = filter(us_states, ID %in% c("texas", "louisiana"))) +
  geom_sf(data = harvey_windswath,
          aes(fill = factor(RADII)), alpha = 0.2) +
  xlim(c(-107, -89)) + ylim(c(25, 37)) +
  scale_fill_viridis(name = "Wind (kts)", discrete = TRUE,
                      option = "B", begin = 0.6, direction = -1)
```



The `read_sf` function is very powerful and can read in data from lots of different formats.

See Section 2 of the `sf` manual (<https://cran.r-project.org/web/packages/sf/vignettes/sf2.html>) for more on this function.

You can find (much, much) more on working with spatial data in R online:

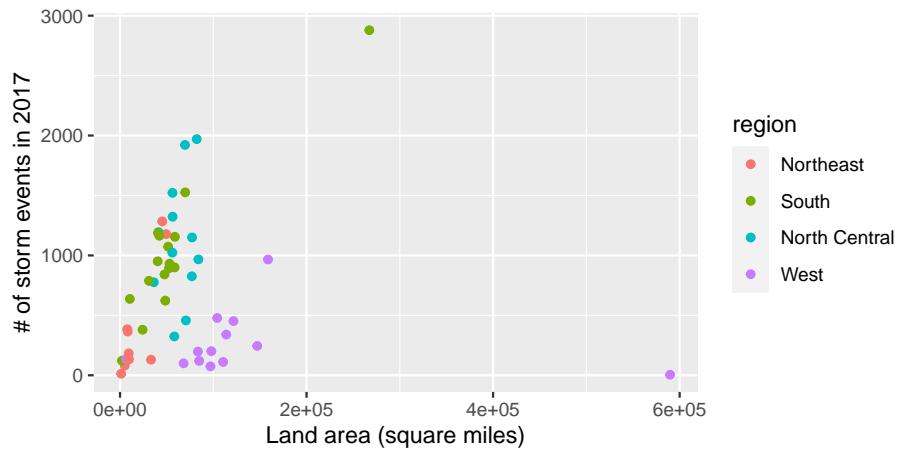
- **R Spatial:** <http://r-spatial.org/index.html>
- **Geocomputation with R:** <https://geocompr.robinlovelace.net>

8.5 In-course exercise Chapter 8

8.5.1 Getting and cleaning the example data

This week, we'll be using some example data from NOAA's Storm Events Database. This data lists major weather-related storm events during 2017. For each event, it includes information like the start and end dates, where it happened, associated deaths, injuries, and property damage, and some other characteristics. Each row is a separate **event**. However, often several events are grouped together within the same **episode**. Some of the event types are listed by their county ID (FIPS code) ("C"), but some are listed by a forecast zone ID ("Z"). Which ID is used is given in the column **CZ_TYPE**.

- Go to <https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/csvfiles/> and download the bulk storm details data for 2017, in the file that starts "StormEvents_details" and includes "d2017".
- Move this into a good directory for your current working directory and read it in using `read_csv` from the `readr` package.
- Limit the dataframe to: the beginning and ending dates and times, the episode ID, the event ID, the state name and FIPS, the "CZ" name, type, and FIPS, the event type, the source, and the begining latitude and longitude and ending latitude and longitude
- Convert the beginning and ending dates to a "date-time" class (there should be one column for the beginning date-time and one for the ending date-time)
- Change state and county names to title case (e.g., "New Jersey" instead of "NEW JERSEY")
- Limit to the events listed by county FIPS (`CZ_TYPE` of "C") and then remove the `CZ_TYPE` column
- Pad the state and county FIPS with a "0" at the beginning (hint: there's a function in `stringr` to do this) and then unite the two columns to make one `fips` column with the 5-digit county FIPS code
- Change all the column names to lower case (you may want to try the `rename_all` function for this)
- There is data that comes with R on U.S. states (`data("state")`). Use that to create a dataframe with the state name, area, and region
- Create a dataframe with the number of events per state in 2017. Merge in the state information dataframe you just created. Remove any states that are not in the state information dataframe
- Create the following plot:



8.5.1.1 Example R code

Read in the data using `read_csv`. Here's the code I used. Yours might be a bit different, depending on the current name of the file and where you moved it.

```
library(readr)
library(dplyr)

storms_2017 <- read_csv("data/StormEvents_details-ftp_v1.0_d2017_c20180918.csv")

## Rows: 56989 Columns: 51
## -- Column specification -----
## Delimiter: ","
## chr (26): STATE, MONTH_NAME, EVENT_TYPE, CZ_TYPE, CZ_NAME, WFO, BEGIN_DATE_T...
## dbl (25): BEGIN_YEARMONTH, BEGIN_DAY, BEGIN_TIME, END_YEARMONTH, END_DAY, EN...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Here's what the first few columns and rows should look like:

```
storms_2017 %>%
  select(1:3) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 3
##   BEGIN_YEARMONTH BEGIN_DAY BEGIN_TIME
##   <dbl>        <dbl>      <dbl>
## 1 201704         6       1509
## 2 201704         6       930
## 3 201704         5      1749
```

Once you've read the data in, here's the code that I used to clean the data:

```
library(lubridate)
library(stringr)
library(tidyr)

storms_2017 <- storms_2017 %>%
  select(BEGIN_DATE_TIME, END_DATE_TIME,
         EPISODE_ID:STATE_FIPS, EVENT_TYPE:CZ_NAME, SOURCE,
         BEGIN_LAT:END_LON) %>%
  mutate(BEGIN_DATE_TIME = dmy_hms(BEGIN_DATE_TIME),
         END_DATE_TIME = dmy_hms(END_DATE_TIME),
         STATE = str_to_title(STATE),
         CZ_NAME = str_to_title(CZ_NAME)) %>%
  filter(CZ_TYPE == "C") %>%
  select(-CZ_TYPE) %>%
  mutate(STATE_FIPS = str_pad(STATE_FIPS, 2, side = "left", pad = "0"),
         CZ_FIPS = str_pad(CZ_FIPS, 3, side = "left", pad = "0")) %>%
  unite(fips, STATE_FIPS, CZ_FIPS, sep = "") %>%
  rename_all(funs(str_to_lower(.)))

## Warning: `fun` was deprecated in dplyr 0.8.0.
## i Please use a list of either functions or lambdas:
##
## # Simple named list: list(mean = mean, median = median)
##
## # Auto named with `tibble::lst()`: tibble::lst(mean, median)
##
## # Using lambdas list(~ mean(.., trim = .2), ~ median(.., na.rm = TRUE))
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

Here's what the data looks like now:

```
storms_2017 %>%
  slice(1:3)

## # A tibble: 3 x 13
##   begin_date_time     end_date_time      episode_id event_id state    fips
##   <dttm>              <dttm>          <dbl>      <dbl> <chr>   <chr>
## 1 2017-04-06 15:09:00 2017-04-06 15:09:00      113355  678791 New Jersey 34015
## 2 2017-04-06 09:30:00 2017-04-06 09:40:00      113459  679228 Florida 12071
## 3 2017-04-05 17:49:00 2017-04-05 17:53:00      113448  679268 Ohio    39057
## # i 7 more variables: event_type <chr>, cz_name <chr>, source <chr>,
## #   begin_lat <dbl>, begin_lon <dbl>, end_lat <dbl>, end_lon <dbl>
```

There is data that comes with R on U.S. states (`data("state")`). Use that to create a dataframe with the state name, area, and region:

```
data("state")
us_state_info <- data.frame(state = state.name,
                             area = state.area,
                             region = state.region)
```

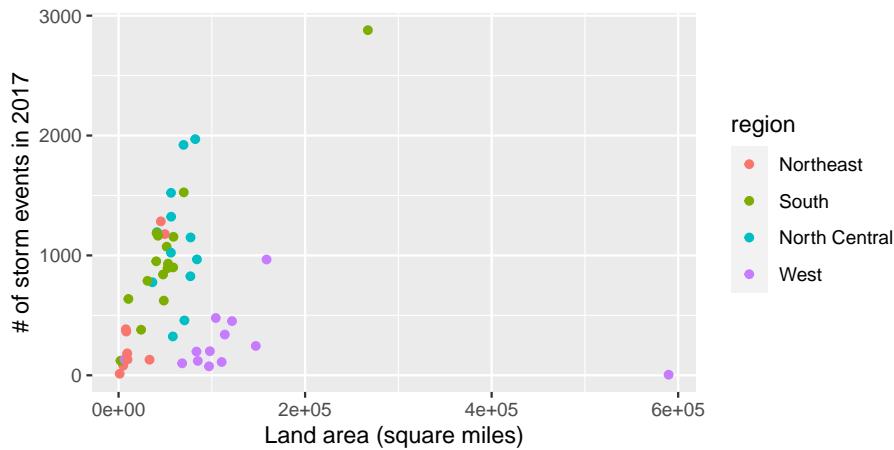
Create a dataframe with the number of events per state in 2017. Merge in the state information dataframe you just created. Remove any states that are not in the state information dataframe:

```
state_storms <- storms_2017 %>%
  group_by(state) %>%
  count() %>%
  ungroup() %>%
  right_join(us_state_info, by = "state")
```

To create the plot:

Ultimately, in this group exercise, you will create a plot of state land area versus the number of storm events in the state:

```
library(ggplot2)
storm_plot <- ggplot(state_storms, aes(x = area, y = n)) +
  geom_point(aes(color = region)) +
  labs(x = "Land area (square miles)",
       y = "# of storm events in 2017")
storm_plot
```

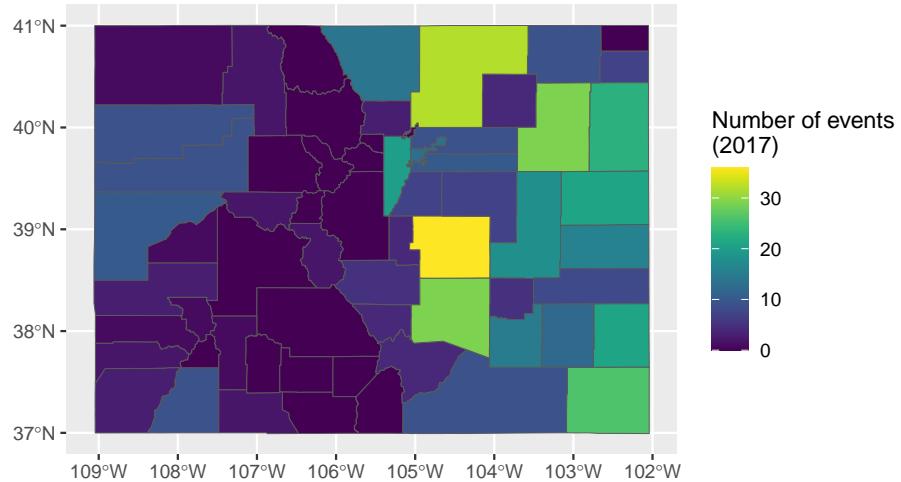


8.5.2 Trying out `ggplot2` extensions

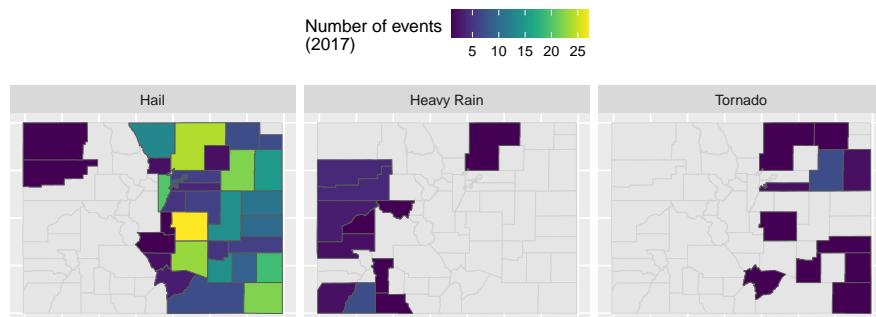
- Go back through the notes so far for this week. Pick your favorite plot that's been shown so far and recreate it. All the code is in the notes, but you'll need to work through it closely to make sure that you understand how to add code from the extension into the rest of the `ggplot2` code.

8.5.3 Using simple features to map

- Re-create the following map of the number of events listed in the NOAA Storm Events database (of those listed by county) for each county in Colorado:



- If you have time, try this one, too. It shows the number of three certain types of events by county. If a county had no events, it's shown in gray (as having a missing value when you count up the events that did happen).



8.5.3.1 Example R code

Here is some R code that could be used to create the figure. Note that the code to create `storms_2017` and `co_counties` is available in the course notes.

```

library(viridis)

co_event_counts <- storms_2017 %>%
  filter(state == "Colorado") %>%
  group_by(fips) %>%
  count() %>%
  ungroup()

co_county_events <- co_counties %>%
  mutate(fips = paste(STATEFP, COUNTYFP, sep = "")) %>%
  full_join(co_event_counts, by = "fips") %>%
  mutate(n = ifelse(!is.na(n), n, 0))

ggplot() +
  geom_sf(data = co_county_events, aes(fill = n)) +
  scale_fill_viridis(name = "Number of events\n(2017)")

```

Example code for second plot:

```

co_event_counts <- storms_2017 %>%
  filter(state == "Colorado") %>%
  filter(event_type %in% c("Tornado", "Heavy Rain", "Hail")) %>%
  group_by(fips, event_type) %>%
  count() %>%
  ungroup()

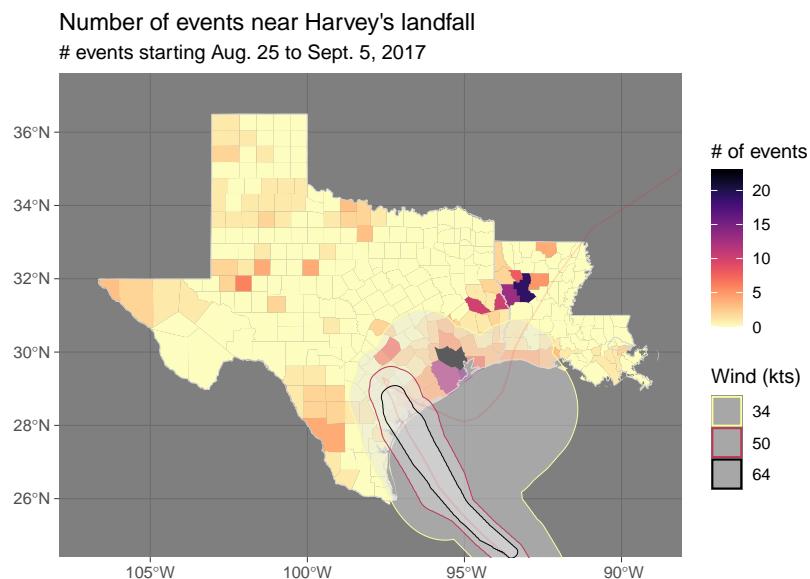
co_county_events <- co_counties %>%
  mutate(fips = paste(STATEFP, COUNTYFP, sep = "")) %>%
  right_join(co_event_counts, by = "fips")

ggplot() +
  geom_sf(data = co_counties, color = "lightgray") +
  geom_sf(data = co_county_events, aes(fill = n)) +
  scale_fill_viridis(name = "Number of events\n(2017)") +
  theme(legend.position = "top") +
  facet_wrap(~ event_type, ncol = 3) +
  theme(axis.line = element_blank(),
        axis.text = element_blank(),
        axis.ticks = element_blank())

```

8.5.4 More on mapping

- See if you can put everything we've talked about together to create the following map. Note that you can get the county boundaries using the `tigris` package and the hurricane data from the NHC website mentioned in the text. The storm events data is in the `storms_2017` dataframe created in code in the text. See how close you can get to this figure.



8.5.4.1 Example R code

Here is the code I used to create the figure:

```
harvey_events <- storms_2017 %>%
  filter(state %in% c("Texas", "Louisiana") &
         ymd_hms("2017-08-25 00:00:00") < begin_date_time &
         begin_date_time < ymd_hms("2017-10-05 00:00:00")) %>%
  group_by(fips) %>%
  count() %>%
  ungroup()

tx_la_counties <- counties(state = c("TX", "LA"), cb = TRUE, class = "sf") %>%
  mutate(fips = paste(STATEFP, COUNTYFP, sep = "")) %>%
  full_join(harvey_events, by = "fips") %>%
```

```
mutate(n = ifelse(is.na(n), 0, n))

ggplot() +
  geom_sf(data = tx_la_counties, color = NA,
          aes(fill = n)) +
  geom_sf(data = filter(us_states, ID %in% c("texas", "louisiana")),
          fill = NA, color = "lightgray") +
  geom_sf(data = harvey_windswath,
          aes(color = factor(RADII)), alpha = 0.4) +
  geom_sf(data = harvey_track, color = "red",
          alpha = 0.1, size = 1) +
  xlim(c(-107, -89)) + ylim(c(25, 37)) +
  scale_fill_viridis(name = "# of events", option = "A", direction = -1) +
  scale_color_viridis(name = "Wind (kts)", discrete = TRUE,
                      option = "B", direction = -1) +
  theme_dark() +
  ggtitle("Number of events near Harvey's landfall",
          subtitle = "# events starting Aug. 25 to Sept. 5, 2017")
```


Chapter 9

Reproducible research #2

Download slides for Nichole Monhait's guest lecture

Nichole's full repo for the lecture is at: https://github.com/nmonhait/erhs_535.

Download a pdf of the lecture slides covering this topic.

9.1 Templates

R Markdown **templates** can be used to change multiple elements of the style of a rendered document. You can think of these as being the document-level analog to the themes we've used with `ggplot` objects.

To do this, some kind of style file is applied when rendering document. For HTML documents, Cascading Style Sheets (CSS) (`.css`) can be used to change the style of different elements. For pdf files, LaTeX package (style) files (`.sty`) are used.

To open a new R Markdown file that uses a template, in RStudio, go to “File” -> “New File” -> “R Markdown” -> “From Template”.

Different templates come with different R packages. A couple of templates come with the `rmarkdown` package, which you likely already have.

Many of these templates will only render to pdf.

To render a pdf from R Markdown, you need to have a version of TeX installed on your computer. Like R, TeX is open source software. RStudio recommends the following installations by system:

- For Macs: MacTeX
- For PCs: MiKTeX

Links for installing both can be found at <http://www.latex-project.org/ftp.html>

Current version of TeX: 3.14159265.

The **tufte** package has templates for creating handouts typeset like Edward Tufte's books.

This package includes templates for creating both pdf and HTML documents in this style.

The package includes special functions like **newthought**, special chunk options like **fig.fullwidth**, and special knitr engines like **marginfigure**. Special features available in the **tufte** template include:

- Margin references
- Margin figures
- Side notes
- Full width figures

The **rticles** package has templates for several journals:

- *Journal of Statistical Software*
- *The R Journal*
- *Association for Computing Machinery*
- ACS publications (*Journal of the American Chemical Society, Environmental Science & Technology*)
- Elsevier publications

Some of these templates create a whole directory, with several files besides the .Rmd file. For example, the template for *The R Journal* includes:

- The R Markdown file in which you write your article
- “RJournal.sty”: A LaTeX package (style) file specific to *The R Journal*. This file tells LaTeX how to render all the elements in your article in the style desired by this journal.
- “RReferences.bib”: A BibTeX file, where you can save citation information for all references in your article.
- “Rlogo.png”: An example figure (the R logo).

Once you render the R Markdown document from this template, you'll end up with some new files in the directory:

- “[your file name].tex”: A TeX file with the content from your R Markdown file. This will be “wrapped” with some additional formatting input to create “RJwrapper.tex”.

- “RJwrapper.tex”: A TeX file that includes both the content from your R Markdown file and additional formatting input. Typically, you will submit this file (along with the BibTeX, any figure and image files, and possibly the style file) to the journal.
- “RJwrapper.pdf”: The rendered pdf file (what the published article would look like)

This template files will often require some syntax that looks more like LaTeX than Markdown.

For example, for the template for *The R Journal*, you need to use `\citet{}` and `\citep{}` to include citations. These details will depend on the style file of the template.

As a note, you can always use raw LaTeX in R Markdown documents, not just in documents you’re creating with a template. You just need to be careful not to mix the two. For example, if you use a LaTeX environment to begin an itemized list (e.g., with `begin{itemize}`), you must start each item with `item`, not `-`.

You can create your own template. You create it as part of a custom R package, and then will have access to the template once you’ve installed the package. This can be useful if you often write documents in a certain style, or if you ever work somewhere with certain formatting requirements for reports.

RStudio has full instructions for creating your own template: http://rmarkdown.rstudio.com/developer_document_templates.html

9.2 R Projects

9.2.1 Organization

So far, you have run much of your analysis within a single R script or R Markdown file. Often, any associated data are within the same working directory as your script or R Markdown file, but the files for one project are not separated from files for other projects.

As you move to larger projects, this kind of set-up won’t work as well. Instead, you’ll want to start keeping all materials for a project in a single and exclusive directory.

Often, it helps to organize the files in a project directory into subdirectories. Common subdirectories include:

- `data-raw`: Raw data and R scripts to clean the raw data.
- `data`: Cleaned data, often saved as `.RData` after being generated by a script in `data-raw`.

- **R:** Code for any functions used in analysis.
- **reports:** Any final products rendered from R Markdown and their original R Markdown files (e.g., paper drafts, reports, presentations).

9.2.2 Creating R Projects

RStudio allows you to create “Projects” to organize code, data, and results within a directory. When you create a project, RStudio adds a file with the extension “.Rproj” to the directory.

There are some advantages to setting a directory to be an R Project. The project:

- Automatically uses the directory as your current working directory when you open the project.
- Coordinates well with git version control and GitHub repository system.
- Opens a “Files” window for navigating project files in an RStudio pane when you open the project.

You can create a new project from scratch or from an existing directory.

To create an R project from a working directory, in RStudio go to “File” -> “New Project” -> “New Directory”. You can then choose where you want to save the new project directory.

9.3 git

Git is a version control system.

It saves information about all changes you make on all files in a repository. This allows you to revert back to previous versions and search through the history for all files in the repository.

Git is open source. You can download it for different operating systems here:

<https://git-scm.com/downloads>

You will need git on your computer to use git with RStudio and create local git repositories you can sync with GitHub repositories.

Before you use git, you should configure it. For example, you should make sure it has your name and email address.

You can configure git with commands at the shell. For example, I would run the following code at a shell to configure git to have my proper user name and email:

```
git config --global user.name "Brooke Anderson"  
git config --global user.email "brooke.anderson@colostate.edu"
```

Sometimes, RStudio will automatically find git (once you've installed git) when you start RStudio.

However, in some cases, you may need to take some more steps to activate git in RStudio. To do this, go to “RStudio” -> “Preferences” -> “Git/SVN”. Choose “Enable version control”. If RStudio doesn't find your version of git in the “Git executable” box, browse for it.

9.3.1 Initializing a git repository

You can initialize a git repository using commands from the shell. To do that, take the following steps (first check that it is not already a git repository):

1. Use a shell (“Terminal” on Macs) to navigate to that directory. You can use `cd` to do that (similar to `setwd` in R).
2. Once you are in the directory, run `git status`. If you get the message `fatal: Not a git repository (or any of the parent directories): .git`, it is not yet a git repository.
3. If you do not get an error from `git status`, the directory is already a repository. If you do get an error, run `git init` to initialize it as a repository.

For example, if I wanted to make the “`fars_analysis`” directory, which is a direct subdirectory of my home directory, a git repository, I could open a shell and run:

```
cd ~/fars_analysis  
git init
```

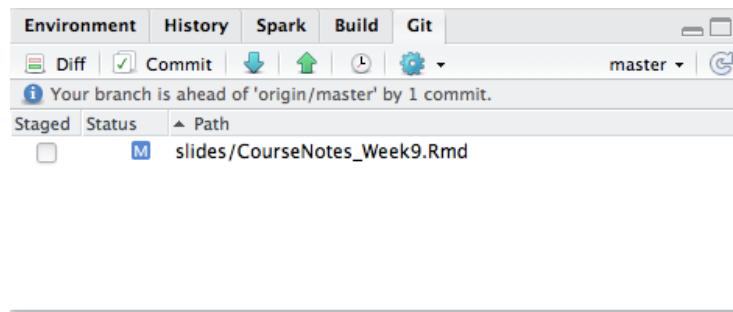
You can also initialize a git repository for a directory that is an R Project directory through R Studio.

1. Open the Project.
2. Go to “Tools” -> “Version Control” -> “Project Setup”.
3. In the box for “Version control system”, choose “Git”.

Note: If you have just installed git, and have not restarted RStudio, you'll need to do that before RStudio will recognize git. If you do not see “Git” in the box for “Version control system”, it means either that you do not have git installed on your computer or that RStudio was unable to find it.

Once you initialize the project as a git repository, you should have a “Git” window in one of your RStudio panes (top right pane by default).

As you make and save changes to files, they will show up in this window for you to commit. For example, this is what the Git window for our coursebook looks like when I have changes to the slides for week 9 that I need to commit:

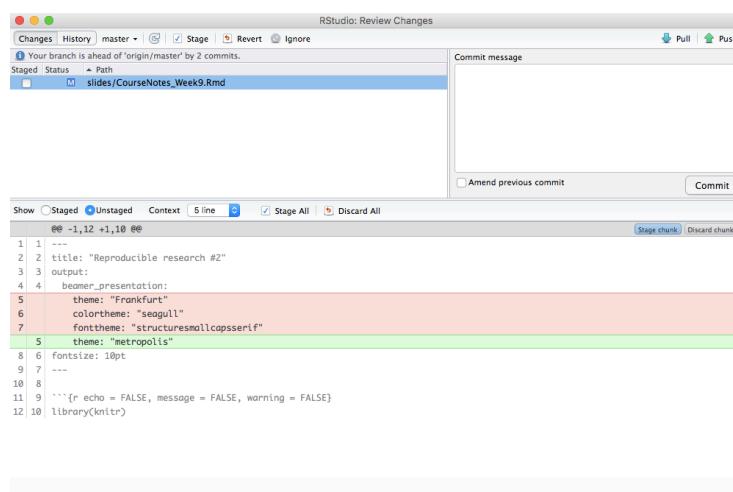


9.3.2 Committing

When you want git to record changes, you *commit* the files with the changes. Each time you commit, you have to include a short commit message with some information about the changes.

You can make commits from a shell. However, in this course we'll just make commits from the RStudio environment.

To make a commit from RStudio, click on the “Commit” button in the Git window. That will open a separate commit window that looks like this:



In this window, to commit changes:

1. Click on the files you want to commit to select them.
2. If you'd like, you can use the bottom part of the window to look through the changes you are committing in each file.
3. Write a message in the "Commit message" box. Keep the message to one line in this box if you can. If you need to explain more, write a short one-line message, skip a line, and then write a longer explanation.
4. Click on the "Commit" button on the right.

Once you commit changes to files, they will disappear from the Git window until you make and save more changes in them.

9.3.3 Browsing history

On the top left of the Commit window, you can toggle to "History". This window allows you to explore the history of commits for the repository.

RStudio: Review Changes

Changes History master > (all commits) | Pull

| Subject | Author | Date | SHA |
|---|---|-------------------|-----------------|
| Move some figure files | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-16 | 20e32e94 |
| Add in some figures for slides | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-16 | 58f43de6 |
| origin/master < origin/HEAD Make a small change to in-course exercise | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-16 | 2293e8f8 |
| Try adding coursebook material for week 9 | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-16 | e4201f72 |
| Some final changes | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-11 | 5f633f71 |
| Try adding back in choropleth code | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-11 | d8cf0aaa |
| A bit more work on in-course exercise | Brooke Anderson <brooke.anderson@colorado.edu> | 2017-10-11 | e401016b |

Commits 1-100 of 584

View file @ 2293e8f8

09-reproducibleresearch2.Rmd

```

@@ -399,7 +399,7 @@ In this part of the group exercise, you will set up an R Project to use for the
399 399 - The 'data-row' directory will ultimately have your raw data as well as some R scripts with code for cleaning up the raw
you are saving this file for 1999. In the form of a csv, you could name the file 'person_1999.csv'.
400 400 - Download FARS data from the years 1999 to 2010. From each year, pull out the 'person' file. Save these yearly 'person'
files in the 'yearly_person_data' subdirectory you created. As a file name, use 'person.' and then the year. For example, if
you are saving this file for 1999, the form of a csv you could name the file 'person_1999.csv'.
401 401 - Open the RStudio project you will have in the 'R' folder in 'Markdown' and add a new R Markdown file ('File' -> 'New
File' -> 'R Markdown') and save it to this subdirectory. You can change the name and data in the file if you'd like. Delete
all the text that comes as a default. Write a piece of code that lists the files you saved in 'data-row/yearly_person_data'.
Remember that the working directory for an R Markdown file is the directory in which it's saved, so you'll need to use a
relative pathname that goes up one directory (...) and then goes into 'data-row' and 'yearly_person_data'.
402 402 - If you have time, go to the [FARS documentation](https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812316) and
find out more about which variables are included in this data set and which values can have.
403 403
404 404 ### Linking your project with a GitHub repository
405 405

```

GitHub allows you to host git repositories online. This allows you to:

- Work collaboratively on a shared repository
- Fork someone else's repository to create your own copy that you can use and change as you want
- Suggest changes to other people's repositories through pull requests

To push local repositories to GitHub and fork other people's repositories, you will need a GitHub account.

You can sign up at <https://github.com>. A free account is fine.

The basic unit for working in GitHub is the repository. You can think of a repository as very similar to an R Project—it's a directory of files with some supplemental files saving some additional information about the directory.

While R Projects have this additional information saved as an “.RProj” file, git repositories have this information in a directory called “.git”. Because this pathname starts with a dot, it won't show up in many of the ways you list files in a directory. From a shell, you can see files that start with . by running `ls -a` from within that directory.

9.3.4 Linking local repo to GitHub repo

If you have a local directory that you would like to push to GitHub, these are the steps to do it.

First, you need to make sure that the directory is under git version control. See the notes on initializing a repository.

Next, you need to create an empty repository on GitHub to sync with your local repository. Do that by:

1. In GitHub, click on the “+” in the upper right corner (“Create new”).
2. Choose “Create new repository”.
3. Give your repository the same name as the local directory you'd like to connect it to. For example, if you want to connect it to a directory called “fars_analysis” on your computer, name the repository “fars_analysis”.
4. Leave everything else as-is (unless you'd like to add a short description in the “Description” box). Click on “Create repository” at the bottom of the page.

Now you are ready to connect the two repositories.

First, you'll want to change some settings in RStudio so GitHub will recognize that your local repository belongs to you, rather than asking for you password every time.

- In RStudio, go to “RStudio” -> “Preferences” -> “Git / svn”. Choose to “Create RSA key”.
- Click on “View public key”. Copy what shows up.
- Go to your GitHub account and navigate to “Settings”. Click on “SSH and GPG keys”.
- Click on “New SSH key”. Name the key something like “RStudio” (you might want to include the device name if you'll have SSH keys from RStudio on several computers). Paste in your public key in the “Key box”.

9.3.5 Syncing RStudio and GitHub

Now you’re ready to push your local repository to the empty GitHub repository you created.

1. Open a shell and navigate to the directory you want to push. (You can open a shell from RStudio using the gear button in the Git window.)
2. Add the GitHub repository as a remote branch with the following command (this gives an example for adding a GitHub repository named “ex_repo” in my GitHub account, “geanders”):

```
git remote add origin git@github.com:geanders/ex_repo.git
```

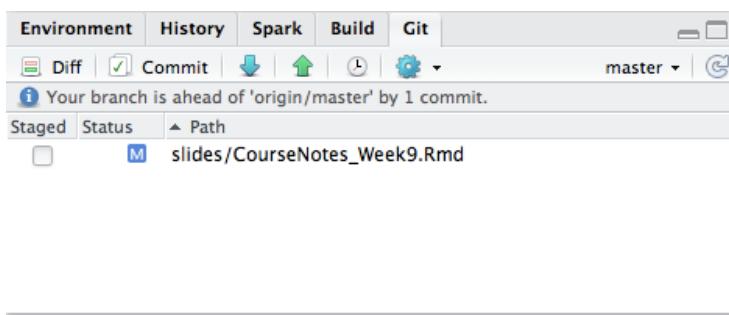
3. Push the contents of the local repository to the GitHub repository.

```
git push -u origin master
```

To pull a repository that already exists on GitHub and to which you have access (or that you’ve forked), first use `cd` to change a shell into the directory where you want to put the repository then run `git clone` to clone the repository locally. For example, if I wanted to clone a GitHub repository called “ex_repo” in my GitHub account, I would run:

```
git clone git@github.com:geanders/ex_repo.git
```

Once you have linked a local R project with a GitHub repository, you can push and pull commits using the blue down arrow (pull from GitHub) and green up arrow (push to GitHub) in the Git window in RStudio.



GitHub helps you work with others on code. There are two main ways you can do this:

- **Collaborating:** Different people have the ability to push and pull directly to and from the same repository. When one person pushes a change to the repository, other collaborators can immediately get the changes by pulling the latest GitHub commits to their local repository.
- **Forking:** Different people have their own GitHub repositories, with each linked to their own local repository. When a person pushes changes to GitHub, it only makes changes to his own repository. The person must issue a pull request to another person’s fork of the repository to share the changes.

9.3.6 Issues

Each original GitHub repository (i.e., not a fork of another repository) has a tab for “Issues”. This page works like a Discussion Forum.

You can create new “Issue” threads to describe and discuss things that you want to change about the repository.

Issues can be closed once the problem has been resolved. You can close issues on the “Issue” page with the “Close issue” button.

If a commit you make in RStudio closes an issue, you can automatically close the issue on GitHub by including “Close #[issue number]” in your commit message and then pushing to GitHub.

For example, if issue #5 is “Fix typo in section 3”, and you make a change to fix that typo, you could make and save the change locally, commit that change with the commit message “Close #5”, and then push to GitHub, and issue #5 in “Issues” for that GitHub repository will automatically be closed, with a link to the commit that fixed the issue.

9.3.7 Pull request

You can use a *pull request* to suggest changes to a repository that you do not own or otherwise have the permission to directly change.

You can also use pull requests within your own repositories. Some people will create a pull request every time they have a big issue they want to fix in one of their repositories.

In GitHub, each repository has a “Pull requests” tab where you can manage pull requests (submit a pull request to another fork or merge in someone else’s pull request for your fork).

Take the following steps to suggest changes to someone else’s repository:

1. Fork the repository

2. Make changes (locally or on GitHub)
3. Save your changes and commit them
4. Submit a pull request to the original repository
5. If there are not any conflicts and the owner of the original repository likes your changes, he or she can merge them directly into the original repository. If there are conflicts, these need to be resolved before the pull request can be merged.

9.3.8 Merge conflicts

At some point, you will get *merge conflicts*. These happen when two people have changed the same piece of code in two different ways at the same time.

For example, say Rachel and I are both working on local versions of the same repository, and I change a line to `mtcars[1,]` while Rachel changes the same line to `head(mtcars, 1)`. Rachel pushes to the GitHub version of the repository before I do.

When I pull the latest commits to the GitHub repository, I will have a merge conflict for this line. To be able to commit a final version, I'll need to decide which version of the code to use and commit a version of the file with that code.

Merge conflicts can come up in a few situations:

- You pull in commits from the GitHub branch of a repository you've been working on locally.
- Someone sends a pull request for one of your repositories.

If there are merge conflicts, they'll show up like this in the file:

```
<<<<< HEAD
mtcars[1, ]
=====
head(mtcars, 1)
>>>> remote-branch
```

To fix them, search for all these spots in files with conflicts, pick the code you want to use, and delete everything else.

For the example conflict, I might change the file from this:

```
<<<<< HEAD
mtcars[1, ]
=====
head(mtcars, 1)
>>>> remote-branch
```

To this:

```
head(mtcars, 1)
```

Then you can save and commit the file.

9.3.9 Find out more

If you'd like to find out more, Hadley Wickham has a great chapter on using git and GitHub with RStudio in his *R Packages* book:

<http://r-pkgs.had.co.nz/git.html>

9.4 In-course exercise Chapter 9

9.4.1 Organizing a project

In this part of the group exercise, you will set up an R Project to use for the next homework assignment. You will want to set up a similar project for your final group project.

- First, you need to create a new project. In RStudio, go to “File” -> “New Project” -> “New Directory”. Choose where you want to save this directory and what you want to name it.
- Once you open the project, one of the RStudio panes should have a tab called “Files”. This shows the files in this project directory and allows you to navigate through them. Currently, you won’t have any files other than the R project file (“.Rproj”). As a next step, create several subdirectories. We’ll use these to structure the data and R Markdown files for your homework. Create the following subdirectories (you can use the “New Folder” button in the RStudio “Files” pane):
 - **data**
 - **writing**
- Download the data for the homework from the *Washington Post’s* GitHub page: <https://github.com/washingtonpost/data-homicides>. Save this data inside your R project in the **data** subdirectory

9.4.2 Initializing git for an R Project

- If you do not already have one, sign up for a GitHub account. The free option is fine.

- If you do not already have git installed on your computer, install it: <https://git-scm.com/downloads>
- Restart RStudio. go to “RStudio” -> “Preferences” ->“Git/SVN”. Choose “Enable version control”. If RStudio doesn’t find your version of git in the “Git executable” box, browse for it.
- Open your homework project in RStudio. Change your Project settings to initialize git for this project (see the course notes for tips on how to do that).
- Open a shell from R using the gear symbol in the “Git” pane you should now see in RStudio. Configure git from this shell. For example, I would open a shell and run:

```
git config --global user.name "Brooke Anderson"  
git config --global user.email "brooke.anderson@colostate.edu"
```

Note that you only need to do this once (until you get a new computer or, maybe, update git). - Go to the “Commit” window. Click on all of the files you see there, and then make an initial commit using “Initial commit” as your commit message. - The `writing` subdirectory will have your R Markdown file and its output. Create a new R Markdown file (“File” -> “New File” -> “R Markdown”) and save it to this subdirectory. You can change the name and date for the file if you’d like. Delete all the text that comes as a default. Write a piece of code that lists the files you saved in the `data` subdirectory. Remember that the working directory for an R Markdown file is the directory in which it’s saved, so you may need to use a relative pathname that goes up one directory (`..`) and then goes into `data` subdirectory. - Commit this change using the “Commit” window. After you commit the changes, look at the “History” window to see the history of your commits.

9.4.3 Linking your project with a GitHub repository

(See course notes for more on these steps.)

- Login to your GitHub account online. Create an empty GitHub repository for the project. Give it the same name as the name of your R project directory.
- If you do not already have an RSA key, create one in RStudio and add it as an SSH key in your GitHub settings. If you already have a key (you almost certainly know if you do), see if you can copy it and submit it in GitHub.
- Set this empty online GitHub repository as the remote branch of your local git repository for the project.
- Push your local repository to this GitHub repository.
- Go to your GitHub account and make sure the repository was pushed.

- Try making some more changes to your local repository. Commit the changes, then use the green up arrow in the Git window to push the changes to your GitHub repository.

Part IV

Part IV: Advanced

Chapter 10

Entering and cleaning data #3

Download a pdf of the lecture slides covering this topic.

10.1 Pulling online data

10.1.1 APIs

APIs are “Application Program Interfaces”. An API provides the rules for software applications to interact. In the case of open data APIs, they provide the rules you need to know to write R code to request and pull data from the organization’s web server into your R session. Often, an API can help you avoid downloading all available data, and instead only download the subset you need.

The basic strategy for using APIs from R is:

- Figure out the API rules for HTTP requests
- Write R code to create a request in the proper format
- Send the request using GET or POST HTTP methods
- Once you get back data from the request, parse it into an easier-to-use format if necessary

Start by reading any documentation available for the API. This will often give information on what data is available and how to put together requests.

EPIC About Page'."/>

Source: <https://api.nasa.gov/api.html#EONET>

Many organizations will require you to get an API key and use this key in each of your API requests. This key allows the organization to control API access, including enforcing rate limits per user. API rate limits restrict how often you can request data (e.g., an hourly limit of 1,000 requests per user for NASA APIs).

You should keep this key private. In particular, make sure you do not include it in code that is posted to GitHub.

The `riem` package, developed by Maelle Salmon and an ROpenSci package, is an excellent and straightforward example of how you can use R to pull open data through a web API. This package allows you to pull weather data from airports around the world directly from the Iowa Environmental Mesonet.

To get a certain set of weather data from the Iowa Environmental Mesonet, you can send an HTTP request specifying a base URL, “<https://mesonet.agron.iastate.edu/cgi-bin/request/asos.py/>”, as well as some parameters describing the subset of dataset you want (e.g., date ranges, weather variables, output format).

Once you know the rules for the names and possible values of these parameters (more on that below), you can submit an HTTP GET request using the `GET` function from the `httr` package.

```
#DEBUG: Format Typ      -> comma
#DEBUG: Time Period    -> 2016-01-01 00:00:00+00:00 2016-10-25 00:00:00
#DEBUG: Time Zone      -> Etc/UTC
#DEBUG: Data Contact   -> daryl herzmann akrherz@iastate.edu 515
#DEBUG: Entries Found  -> -1
station,valid, sped
DEN,2016-01-01 00:53,6.9
DEN,2016-01-01 01:53,10.4
DEN,2016-01-01 02:53,12.7
DEN,2016-01-01 03:53,10.4
DEN,2016-01-01 04:53,8.1
DEN,2016-01-01 05:53,10.4
DEN,2016-01-01 06:53,9.2
DEN,2016-01-01 07:53,8.1
DEN,2016-01-01 08:53,6.9
DEN,2016-01-01 09:53,11.5
DEN,2016-01-01 10:53,11.5
DEN,2016-01-01 11:53,5.8
DEN,2016-01-01 12:53,6.9
DEN,2016-01-01 13:53,9.2
```

https://mesonet.agron.iastate.edu/cgi-bin/request/asos.py?station=DEN&data=sknt&year1=2016&month1=6&day1=1&year2=2016&month2=6&day2=30&tz=America%2FDenver&format=comma&latlon=no&direct=no&report_type=1&report_type=2

When you are making an HTTP request using the `GET` or `POST` functions from the `httr` package, you can include the key-value pairs for any query parameters as a list object in the `query` argument of the function.

```
library(httr)
meso_url <- paste0("https://mesonet.agron.iastate.edu/",
                     "cgi-bin/request/asos.py/")
denver <- GET(url = meso_url,
               query = list(station = "DEN", data = "sped",
```

```
year1 = "2016", month1 = "6",
day1 = "1", year2 = "2016",
month2 = "6", day2 = "30",
tz = "America/Denver",
format = "comma"))
```

The `GET` call will return a special type of list object with elements that include the url you queried and the content of the page at that url:

```
str(denver, max.level = 1, list.len = 6)
```

```
## List of 10
## $ url      : chr "https://mesonet.agron.iastate.edu/cgi-bin/request/asos.py?sta...
## $ status_code: int 200
## $ headers   :List of 6
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
## $ all_headers:List of 1
## $ cookies   :'data.frame': 0 obs. of 7 variables:
## $ content    : raw [1:239499] 23 44 45 42 ...
##   [list output truncated]
## - attr(*, "class")= chr "response"
```

The `httr` package includes functions to pull out elements of this list object, including:

- `headers`: Pull out the header information
- `content`: Pull out the content returned from the page
- `status_code`: Pull out the status code from the `GET` request (e.g., 200: okay; 404: not found)

Note: For some fun examples of 404 pages, see <https://www.creativebloq.com/web-design/best-404-pages-812505>

You can use `content` from `httr` to retrieve the contents of the HTTP request we made. For this particular web data, the requested data is a comma-separated file, so you can convert it to a dataframe with `read_csv`:

```
denver %>% content() %>%
  read_csv(skip = 5, na = "M") %>%
  slice(1:3)
```

```

## Rows: 9108 Columns: 3
## -- Column specification -----
## Delimiter: ","
## chr (1): station
## dbl (1): sped
## dttm (1): valid
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 3 x 3
##   station valid           sped
##   <chr>    <dttm>        <dbl>
## 1 DEN      2016-06-01 00:00:00  9.2
## 2 DEN      2016-06-01 00:05:00  9.2
## 3 DEN      2016-06-01 00:10:00  6.9

```

The `riem` package wraps up this whole process, so you can call a single function to get in the data you want from the API:

```

library(riem)
denver_2 <- riem_measures(station = "DEN",
                           date_start = "2016-06-01",
                           date_end = "2016-06-30")
denver_2 %>% slice(1:3)

## # A tibble: 3 x 32
##   station valid           lon   lat  tmpf  dwpf  relh  drct  sknt p01i
##   <chr>    <dttm>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 DEN      2016-06-01 00:00:00 -105.  39.8   NA     NA     70     7     NA
## 2 DEN      2016-06-01 00:05:00 -105.  39.8   NA     NA     80     8     NA
## 3 DEN      2016-06-01 00:10:00 -105.  39.8   NA     NA     80     9     NA
## # i 22 more variables: alti <dbl>, mslp <dbl>, vsby <dbl>, gust <dbl>,
## #   skyc1 <chr>, skyc2 <chr>, skyc3 <chr>, skyc4 <chr>, skyl1 <dbl>,
## #   skyl2 <dbl>, skyl3 <dbl>, skyl4 <dbl>, wxcodes <chr>,
## #   ice_accretion_1hr <lgl>, ice_accretion_3hr <lgl>, ice_accretion_6hr <lgl>,
## #   peak_wind_gust <dbl>, peak_wind_drct <dbl>, peak_wind_time <chr>,
## #   feel <dbl>, metar <chr>, snowdepth <lgl>

```

10.2 Example R API wrapper packages

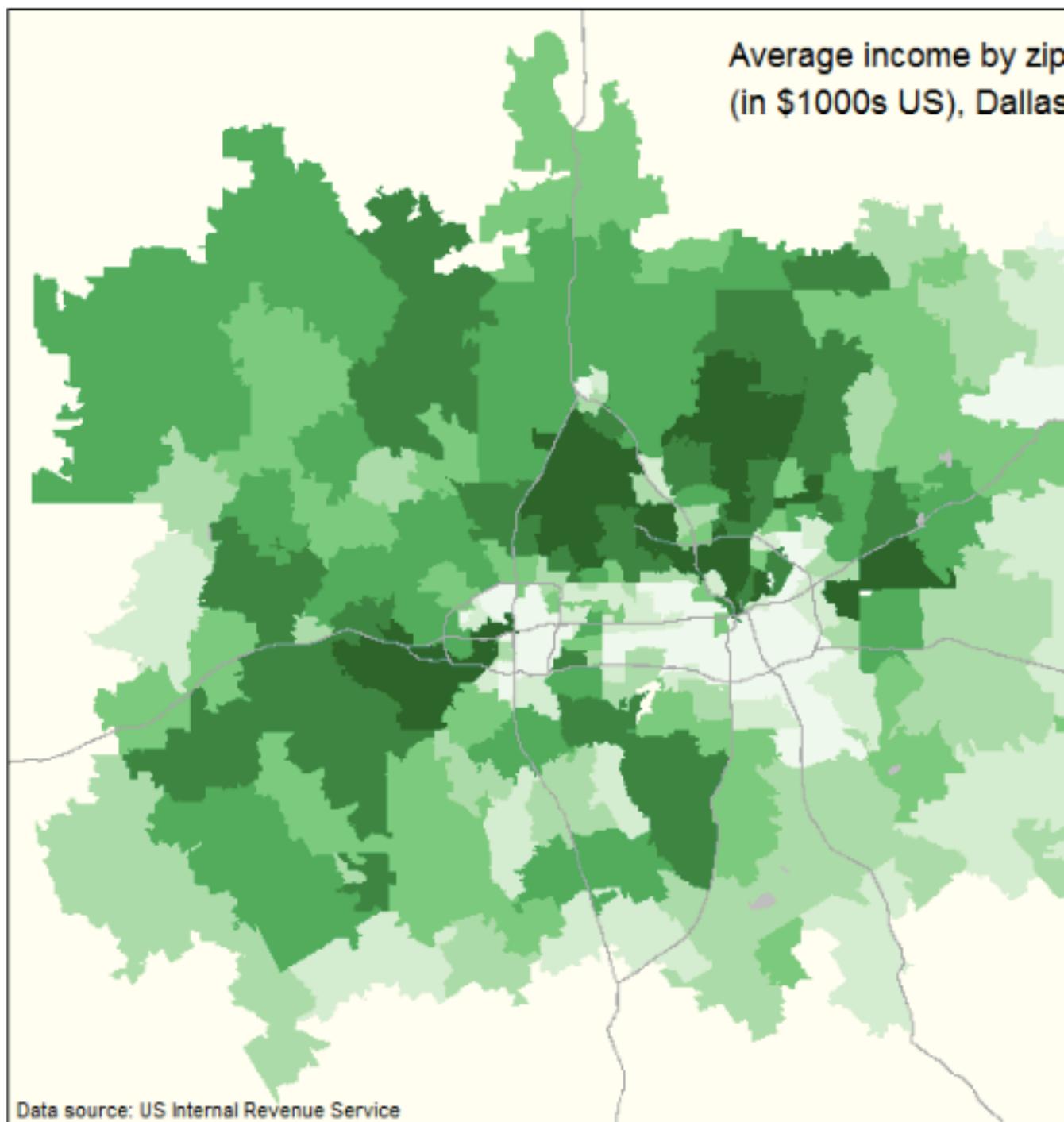
The `tigris` package is a very useful example of an API wrapper. It retrieves geographic boundary data from the U.S. Census for a number of different ge-

ographies:

- Location boundaries
 - States
 - Counties
 - Blocks
 - Tracks
 - School districts
 - Congressional districts
- Roads
 - Primary roads
 - Primary and secondary roads
- Water
 - Area-water
 - Linear-water
 - Coastline
- Other
 - Landmarks
 - Military

10.3 tigris package

The following plot is an example of the kinds of maps you can create using the **tigris** package. This map comes from: *Kyle Walker. 2016. “tigris: An R Package to Access and Work with Geographic Data from the US Census Bureau”.* *The R Journal.* This is a great article to read to find out more about **tigris**.



A number of other R packages also help you access and use data from the U.S. Census:

- **acs**: Download, manipulate, and present American Community Survey and Decennial data from the US Census (see “Working with the American Community Survey in R: A Guide to Using the `acs` Package”, a book available free online through the CSU library)
- **USABoundaries**: Historical and contemporary boundaries of the United States of America
- **idbr**: R interface to the US Census Bureau International Data Base API (e.g., populations of other countries)

The organization rOpenSci (<https://ropensci.org>) has the following mission:

“At rOpenSci we are creating packages that allow access to data repositories through the R statistical programming environment that is already a familiar part of the workflow of many scientists. Our tools not only facilitate drawing data into an environment where it can readily be manipulated, but also one in which those analyses and methods can be easily shared, replicated, and extended by other researchers.”

rOpenSci collects a number of packages for tapping into open data for research. These are listed at <https://ropensci.org/packages>. Many of these packages are wrappers for APIs with data useful for scientific research

Some examples (all descriptions from rOpenSci):

- **AntWeb**: Access data from the world’s largest ant database
- **chromer**: Interact with the chromosome counts database (CCDB)
- **gender**: Encodes gender based on names and dates of birth
- **musemeta**: R Client for Scraping Museum Metadata, including The Metropolitan Museum of Art, the Canadian Science & Technology Museum Corporation, the National Gallery of Art, and the Getty Museum, and more to come.
- **rusda**: Interface to some USDA databases
- **webchem**: Retrieve chemical information from many sources. Currently includes: Chemical Identifier Resolver, ChemSpider, PubChem, and Chemical Translation Service.

As an example, one ROpenSci package, **rnoaa**, allows you to:

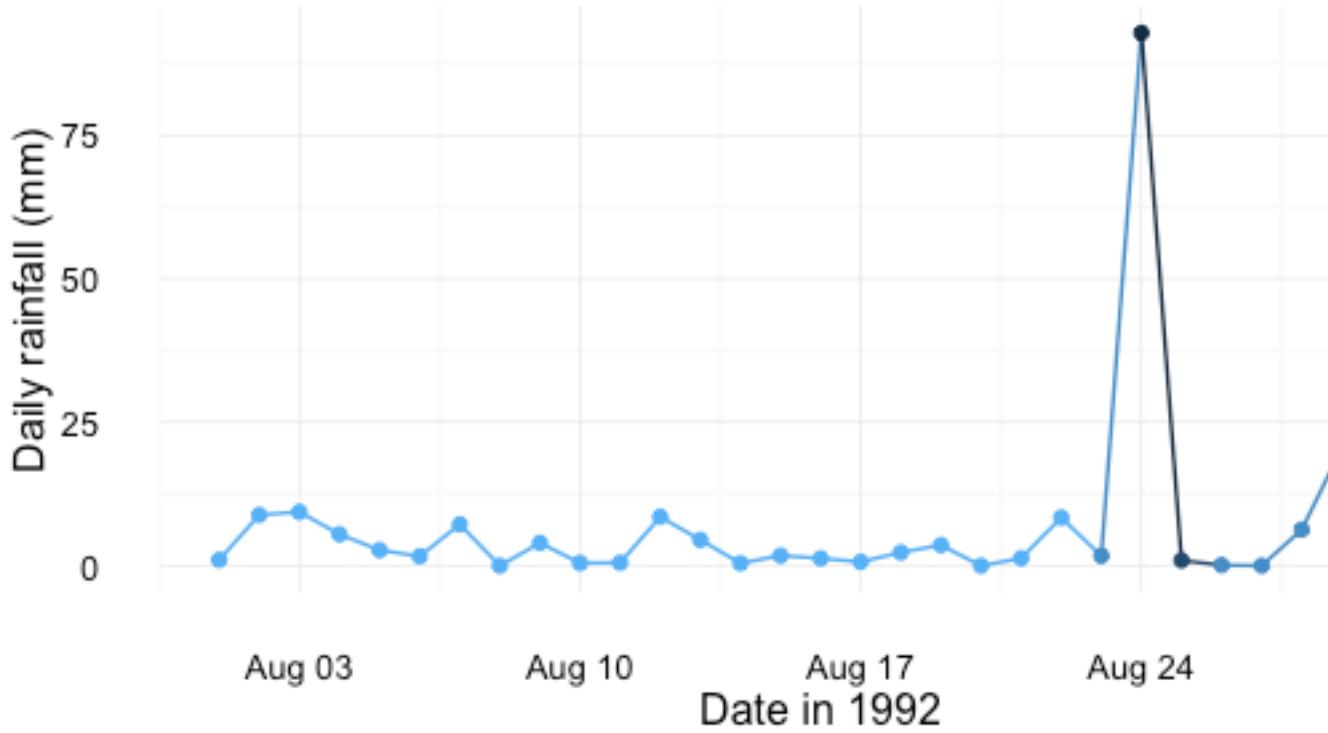
“Access climate data from NOAA, including temperature and precipitation, as well as sea ice cover data, and extreme weather events”

It includes access to:

- Buoy data from the National Buoy Data Center
- Historical Observing Metadata Repository (HOMR)— climate station metadata
- National Climatic Data Center weather station data
- Sea ice data
- International Best Track Archive for Climate Stewardship (IBTrACS)— tropical cyclone tracking data
- Severe Weather Data Inventory (SWDI)

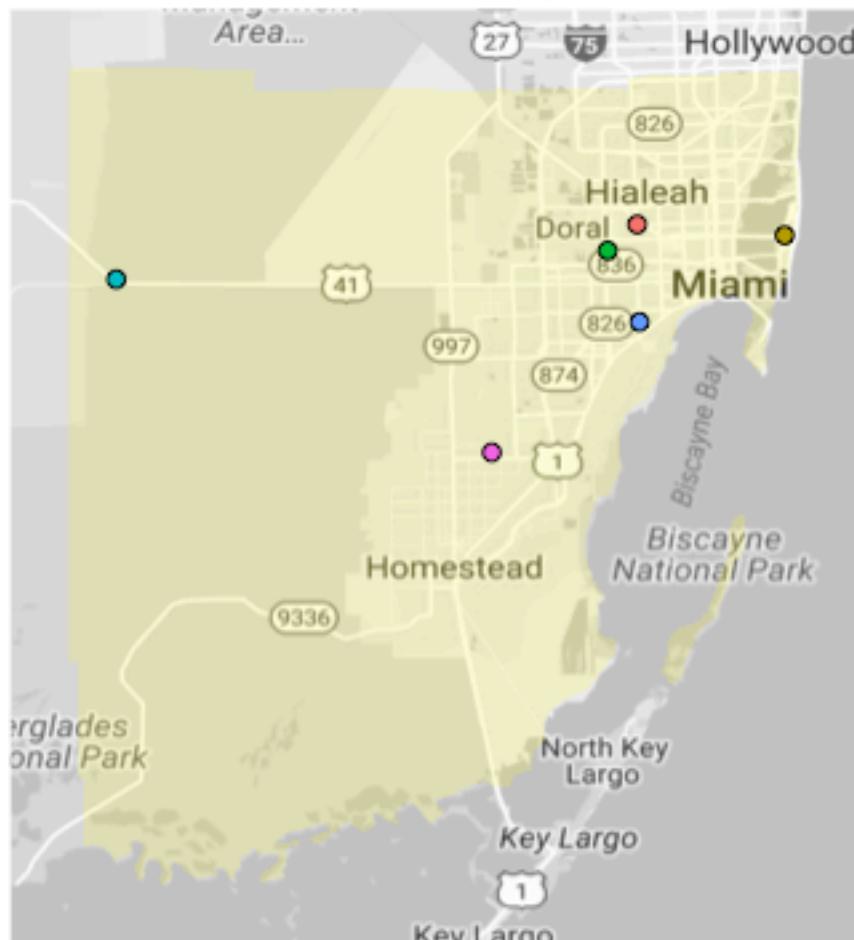
10.4 countyweather

The `countyweather` package, developed by a student here at CSU, wraps the `rnoaa` package to let you pull and aggregate weather at the county level in the U.S. For example, you can pull all data from Miami during Hurricane Andrew:



When you pull the data for a county, the package also maps the contributing weather stations:

Miami-Dade County, Florida



The USGS also has a very nice collection of R packages that wrap USGS open data APIs, which can be accessed through: <https://owi.usgs.gov/R/>

"USGS-R is a community of support for users of the R scientific programming language. USGS-R resources include R training ma-

terials, R tools for the retrieval and analysis of USGS data, and support for a growing group of USGS-R developers.”

USGS R packages include:

- **dataRetrieval**: Obtain water quality sample data, streamflow data, and metadata directly from either the USGS or EPA
- **EGRET**: Analysis of long-term changes in water quality and streamflow, including the water-quality method Weighted Regressions on Time, Discharge, and Season (WRTDS)
- **laketemps**: Lake temperature data package for Global Lake Temperature Collaboration Project
- **lakeattributes**: Common useful lake attribute data
- **soilmoisturetools**: Tools for soil moisture data retrieval and visualization

Here are some examples of other R packages that facilitate use of an API for open data:

- **twitteR**: Twitter
- **Quandl**: Quandl (financial data)
- **RGoogleAnalytics**: Google Analytics
- **WDI**, **wbstats**: World Bank
- **GuardianR**, **rdian**: The Guardian Media Group
- **blsAPI**: Bureau of Labor Statistics
- **rtimes**: New York Times

Find out more about writing API packages with this vignette for the `httr` package: <https://cran.r-project.org/web/packages/httr/vignettes/api-packages.html>. This document includes advice on error handling within R code

that accesses data through an open API.

10.5 Cleaning very messy data

One version of Atlantic basin hurricane tracks is available here: <https://www.nhc.noaa.gov/data/hurdat/hurdat2-1851-2017-050118.txt>. The data is not in a classic delimited format:

This data is formatted in the following way:

- Data for many storms are included in one file.
 - Data for a storm starts with a shorter line, with values for the storm ID, name, and number of observations for the storm. These values are comma separated.
 - Observations for each storm are longer lines. There are multiple observations for each storm, where each observation gives values like the location and maximum winds for the storm at that time.

Strategy for reading in very messy data:

1. Read in all lines individually.
 2. Use regular expressions to split each line into the elements you'd like to use to fill columns.
 3. Write functions and / or `map` calls to process lines and use the contents to fill a data frame.
 4. Once you have the data in a data frame, do any remaining cleaning to create a data frame that is easy to use to answer research questions.

Because the data is not nicely formatted, you can't use `read_csv` or similar functions to read it in.

However, the `read_lines` function from `readr` allows you to read a text file in one line at a time. You can then write code and functions to parse the file one line at a time, to turn it into a dataframe you can use.

Note: Base R has `readLines`, which is very similar.

The `read_lines` function from `readr` will read in lines from a text file directly, without trying to separate into columns. You can use the `n_max` argument to specify the number of lines to read it.

For example, to read in three lines from the hurricane tracking data, you can run:

```
tracks_url <- paste0("http://www.nhc.noaa.gov/data/hurdat/",  
                      "hurdat2-1851-2017-050118.txt")  
hurr_tracks <- read_lines(tracks_url, n_max = 3)  
hurr_tracks
```

```
## [1] "AL011851,           UNNAMED,      14,"  
## [2] "18510625, 0000, , HU, 28.0N, 94.8W, 80, -999, -999, -999, -999, -999, -999,  
## [3] "18510625, 0600, , HU, 28.0N, 95.4W, 80, -999, -999, -999, -999, -999, -999, -999,
```

The data has been read in as a vector, rather than a dataframe:

```
class(hurr_tracks)
```

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

```
length(hurr_tracks)
```

```
## [1] 3
```

```
hurr_tracks[1]
```

```
## [1] "AL011851,           UNNAMED,      14,"
```

You can use regular expressions to break each line up. For example, you can use `str_split` from the `stringr` package to break the first line of the hurricane track data into its three separate components:

```
library(stringr)  
str_split(hurr_tracks[1], pattern = ",")
```

```
## [[1]]
## [1] "AL011851"           "UNNAMED" "14"
## [4] "
```

You can use this to create a list where each element of the list has the split-up version of a line of the original data. First, read in all of the data:

```
tracks_url <- paste0("http://www.nhc.noaa.gov/data/hurdat/",
                      "hurdat2-1851-2017-050118.txt")
hurr_tracks <- read_lines(tracks_url)
length(hurr_tracks)
```

```
## [1] 52151
```

Next, use `map` with `str_split` to split each line of the data at the commas:

```
library(purrr)
hurr_tracks <- purrr::map(hurr_tracks, str_split,
                           pattern = ",", simplify = TRUE)
hurr_tracks[[1]]
```

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] "AL011851" "UNNAMED" "14" "
```

```
hurr_tracks[[2]][1:2]
```

```
## [1] "18510625" "0000"
```

Next, you want to split this list into two lists, one with the shorter “meta-data” lines and one with the longer “observation” lines. You can use `map_int` to create a vector with the length of each line. You will later use this to identify which lines are short or long.

```
hurr_lengths <- map_int(hurr_tracks, length)
hurr_lengths[1:17]
```

```
## [1] 4 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 4 21
```

```
unique(hurr_lengths)
```

```
## [1] 4 21
```

You can use bracket indexing to split the `hurr_tracks` into two lists: one with the shorter lines that start each observation (`hurr_meta`) and one with the storm observations (`hurr_obs`). Use bracket indexing with the `hurr_lengths` vector you just created to make that split.

```
hurr_meta <- hurr_tracks[hurr_lengths == 4]
hurr_obs <- hurr_tracks[hurr_lengths == 21]
```

```
hurr_meta[1:3]
```

```
## [[1]]
##      [,1]      [,2]      [,3]      [,4]
## [1,] "AL011851" "UNNAMED" "14"   ""
##
## [[2]]
##      [,1]      [,2]      [,3]      [,4]
## [1,] "AL021851" "UNNAMED" "1"    ""
##
## [[3]]
##      [,1]      [,2]      [,3]      [,4]
## [1,] "AL031851" "UNNAMED" "1"    ""
```

```
hurr_obs[1:2]
```

```
## [[1]]
##      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]      [,9]
## [1,] "18510625" "0000" " " "HU" "28.0N" "94.8W" "80" "-999" "-999"
##      [,10]     [,11]     [,12]     [,13]     [,14]     [,15]     [,16]     [,17]     [,18]
## [1,] "-999" "-999" "-999" "-999" "-999" "-999" "-999" "-999" "-999"
##      [,19]     [,20]     [,21]
## [1,] "-999" "-999" ""
##
## [[2]]
```

```
##      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]      [,9]
## [1,] "18510625" " 0600" "   " " HU" " 28.0N" " 95.4W" " 80" " -999" " -999"
##      [,10]     [,11]     [,12]     [,13]     [,14]     [,15]     [,16]     [,17]     [,18]
## [1,] " -999" " -999" " -999" " -999" " -999" " -999" " -999" " -999" " -999"
##      [,19]     [,20]     [,21]
## [1,] " -999" " -999" "
```

Now, you can use `bind_rows` from `dplyr` to change the list of metadata into a dataframe. (You first need to use `as_tibble` with `map` to convert all elements of the list from matrices to dataframes.)

```
library(dplyr); library(tibble)
hurr_meta <- hurr_meta %>%
  purrr::map(as_tibble) %>%
  bind_rows()
```

```
## Warning: The `x` argument of `as_tibble.matrix()` must have unique column names if
## `name_repair` is omitted as of tibble 2.0.0.
## i Using compatibility `name_repair`.
## i The deprecated feature was likely used in the purrr package.
## Please report the issue at <https://github.com/tidyverse/purrr/issues>.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
hurr_meta %>% slice(1:3)
```

```
## # A tibble: 3 x 4
##   V1      V2          V3      V4
##   <chr>    <chr>      <chr>    <chr>
## 1 AL011851 "UNNAMED" "14"   ""
## 2 AL021851 "UNNAMED" "1"    ""
## 3 AL031851 "UNNAMED" "1"    ""
```

You can clean up the data a bit more.

- First, the fourth column doesn't have any non-missing values, so you can get rid of it:

```
unique(hurr_meta$V4)
```

```
## [1] "
```

- Second, the second and third columns include a lot of leading whitespace:

```
hurr_meta$V2[1:2]
```

```
## [1] "UNNAMED" "UNNAMED"
```

- Last, we want to name the columns.

```
hurr_meta <- hurr_meta %>%
  select(-V4) %>%
  rename(storm_id = V1, storm_name = V2, n_obs = V3) %>%
  mutate(storm_name = str_trim(storm_name),
         n_obs = as.numeric(n_obs))
hurr_meta %>% slice(1:3)
```

```
## # A tibble: 3 x 3
##   storm_id storm_name n_obs
##   <chr>     <chr>      <dbl>
## 1 AL011851 UNNAMED      14
## 2 AL021851 UNNAMED       1
## 3 AL031851 UNNAMED       1
```

Now you can do the same idea with the hurricane observations. First, we'll want to add storm identifiers to that data. The “meta” data includes storm ids and the number of observations per storm. We can take advantage of that to make a `storm_id` vector that will line up with the storm observations.

```
storm_id <- rep(hurr_meta$storm_id, times = hurr_meta$n_obs)
head(storm_id, 3)
```

```
## [1] "AL011851" "AL011851" "AL011851"
```

```
length(storm_id)
```

```
## [1] 50303
```

```
length(hurr_obs)
```

```
## [1] 50303
```

```
hurr_obs <- hurr_obs %>%
  purrr::map(as_tibble) %>%
  bind_rows() %>%
  mutate(storm_id = storm_id)
hurr_obs %>% select(V1:V2, V5:V6, storm_id) %>% slice(1:3)
```

```
## # A tibble: 3 x 5
##   V1      V2      V5      V6      storm_id
##   <chr>   <chr>   <chr>   <chr>   <chr>
## 1 18510625 " 0000" " 28.0N" " 94.8W" AL011851
## 2 18510625 " 0600" " 28.0N" " 95.4W" AL011851
## 3 18510625 " 1200" " 28.0N" " 96.0W" AL011851
```

10.6 In-course exercise Chapter 10

10.6.1 Working with an API wrapper package

The `rplos` package provides a wrapper to the Public Library of Science (PLoS)'s API. PLOS has a collection of academic journals spanning a variety of topics.

- Check out this page of documentation for this API: <http://api.plos.org/solr/search-fields/> Look through the potential search terms.
- Use the `searchplos` function to search articles in the PLoS collection for the term “West Nile”. Pull the publication date, title, abstract, article type, subject, and journal of each matching article and save the result to an R object called `wn_papers`. You may find it helpful to look at the examples in the helpfile for `searchplos` or the tutorial available at: https://ropensci.org/tutorials/rplos_tutorial/

- The object returned by `searchplos` will be a list with two top levels, `meta` and `data`. Confirm that this is true for the `wn_papers` object you created. Look at the `meta` part of the list (you can use `$` indexing to pull this out). How many articles were found with “West Nile” in them? Does the query seem case-sensitive (i.e., do you get the same number of papers when you query “west nile” rather than “West Nile”)?
- Re-run your query (save the results to `wn_papers_titles`) looking only for papers with “West Nile” in the title of the paper. How many papers are returned by this query?
- By default the limit of the number of papers returned by a query will be 10. You can change this (to a certain degree) by using the `limit` option. For the call you ran to create `wn_papers_titles`, set the limit to the number of articles that match this query, as identified in the `meta` element of the first run of the call. Check the number of rows in the `data` element that was returned to make sure it has the same number of rows as the number of articles that match the query.
- Create a plot of the number of articles published per year. Use color to show which articles are Research Articles versus other types of articles.
- Determine which journals have published these articles and the number of articles published in each journal. You may notice that sometimes “PLoS” is used and sometimes “PLOS”. See if you can fix this in R to get the count of articles per journal without this capitalization difference causing problems.
- Explore the list of packages on ROpenSci, those through the USGS, and those related to the U.S. Census. See if you can identify any other packages that might provide access to data relevant to West Nile virus.

10.6.1.1 Example R code

```
library(rplos)
wn_papers <- searchplos(q = "West Nile",
                           fl = c("publication_date", "title", "journal", "subject",
                                  "abstract", "article_type"))
```

Confirm the structure of the returned object:

```
str(wn_papers)

## List of 2
## $ meta: tibble [1 x 2] (S3: tbl_df/tbl/data.frame)
```

```

##   ..$ numFound: int 4121
##   ..$ start   : int 0
## $ data: tibble [10 x 6] (S3: tbl_df/tbl/data.frame)
##   ..$ journal      : chr [1:10] "PLOS Neglected Tropical Diseases" "PLOS Pathogen...
##   ..$ publication_date: chr [1:10] "2021-05-06T00:00:00Z" "2019-10-31T00:00:00Z" "20...
##   ..$ article_type   : chr [1:10] "Review" "Review" "Review" "Research Article" ...
##   ..$ abstract       : chr [1:10] "\nAfter the unexpected arrival of West Nile virus ...
##   ..$ title         : chr [1:10] "A 20-year historical review of West Nile virus a...
##   ..$ subject        : chr [1:10] "/Biology and life sciences/Anatomy/Body fluids/...
##   ..- attr(*, "numFound")= int 4121
##   ..- attr(*, "start")= int 0

```

Look at the meta data:

```
wn_papers$meta
```

```

## # A tibble: 1 x 2
##   numFound start
##       <int> <int>
## 1      4121     0

```

Query for just articles with “West Nile” in the title:

```
wn_papers_titles <- searchplos(q = "title:West Nile",
                                fl = c("publication_date", "title", "journal", "subject",
                                       "abstract", "article_type"))
```

Look at the metadata:

```
wn_papers_titles$meta
```

```

## # A tibble: 1 x 2
##   numFound start
##       <int> <int>
## 1      248     0

```

Re-run the query using an appropriate limit to get all the matching articles:

```
wn_papers_titles <- searchplos(q = "title:West Nile",
                                fl = c("publication_date", "title", "journal", "subject",
                                       "abstract", "article_type"),
                                limit = 190)
```

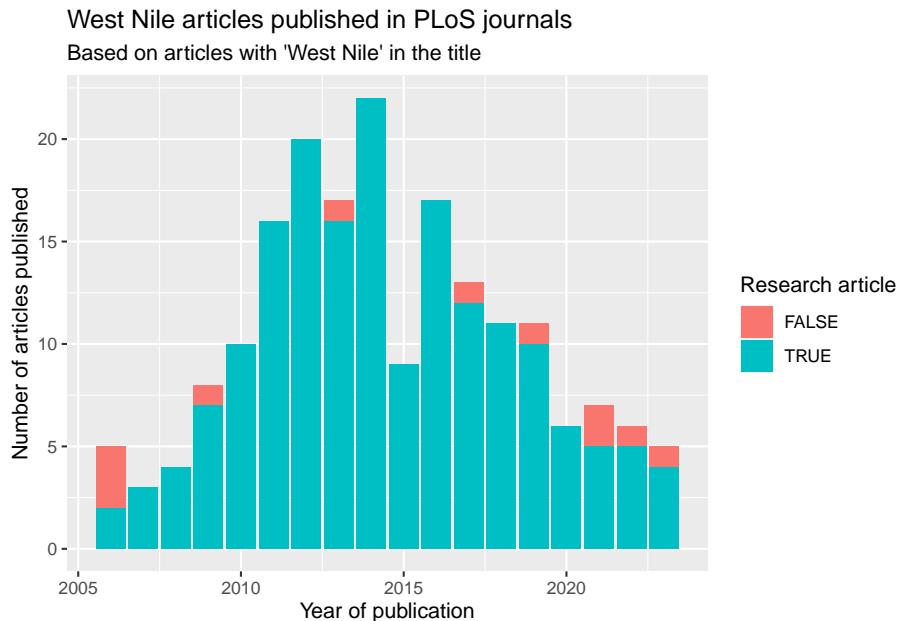
Check the number of rows:

```
nrow(wn_papers_titles$data)
```

```
## [1] 190
```

Create a plot of the number of articles in the PLoS journals with “West Nile” in the title by year:

```
library(lubridate)
library(dplyr)
library(ggplot2)
wn_papers_titles$data %>%
  mutate(publication_date = ymd_hms(publication_date),
         pub_year = year(publication_date),
         research_article = article_type == "Research Article") %>%
  group_by(pub_year, research_article) %>%
  count() %>%
  ggplot(aes(x = pub_year, y = n, fill = research_article)) +
  geom_col() +
  labs(x = "Year of publication",
       y = "Number of articles published",
       fill = "Research article") +
  ggtitle("West Nile articles published in PLoS journals",
          subtitle = "Based on articles with 'West Nile' in the title")
```



Determine which journals have published these articles and the number of articles published in each journal:

```
library(forcats)
library(stringr)
wn_papers_titles$data %>%
  mutate(journal = str_replace(journal, "PLOS", "PLoS")) %>%
  group_by(journal) %>%
  count() %>%
  ungroup() %>%
  filter(!is.na(journal)) %>%
  arrange(desc(n))
```

```
## # A tibble: 7 x 2
##   journal                      n
##   <chr>                         <int>
## 1 PLoS ONE                       98
## 2 PLoS Neglected Tropical Diseases 48
## 3 PLoS Pathogens                   28
## 4 PLoS Computational Biology      3
## 5 PLoS Medicine                   3
## 6 PLoS Biology                     2
## 7 PLoS Climate                     1
```

10.6.2 Cleaning very messy data

With your groups, create an R script that does all the steps described so far to pull the messy hurricane tracks data from online and clean it.

Then try the following further cleaning steps:

- Select only the columns with date, time, storm status, location (latitude and longitude), maximum sustained winds, and minimum pressure and renames them
- Create a column with the date-time of each observation, in a date-time class
- Clean up the latitude and longitude so that you have separate columns for the numeric values and for the direction indicator (e.g., N, S, E, W)
- Clean up the wind column, so it gives wind speed as a number and `NA` in cases where wind speed is missing
- If you have time, try to figure out what the `status` abbreviations stand for. Create a new factor column named `status_long` with the status spelled out.

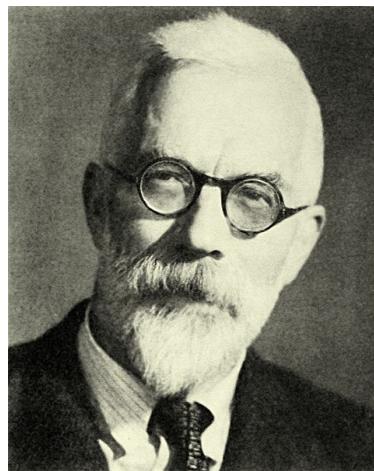
Chapter 11

Exploring data #3

Download a pdf of the lecture slides covering this topic.

11.1 Simulations

11.1.1 The lady tasting tea



Source: Flickr commons, <https://www.flickr.com/photos/internetarchivebookimages/20150531109/>

“Dr. Muriel Bristol, a colleague of Fisher’s, claimed that when drinking tea she could distinguish whether milk or tea was added to the

cup first (she preferred milk first). To test her claim, Fisher asked her to taste eight cups of tea, four of which had milk added first and four of which had tea added first.” — Agresti, *Categorical Data Analysis*, p.91

Research questions:

- If she just guesses, what is the probability she will get all cups right?
- What if more or fewer cups are used in the experiment?

One way to figure this out is to run a *simulation*.

In R, `sample` can be a very helpful function for simulations. It lets you randomly draw values from a vector, with or without replacement.

```
## Generic code
sample(x = [vector to sample from],
       size = [number of samples to take],
       replace = [logical-- should values in the
                  vector be replaced?],
       prob = [vector of probability weights])
```

Create vectors of the true and guessed values, in order, for the cups of tea:

```
n_cups <- 8
cups <- sample(rep(c("milk", "tea"), each = n_cups / 2))
cups
```

```
## [1] "milk" "milk" "tea"   "milk" "milk" "tea"   "tea"   "tea"
```

```
guesses <- sample(rep(c("milk", "tea"), each = n_cups / 2))
guesses
```

```
## [1] "tea"   "tea"   "milk" "milk" "tea"   "milk" "tea"   "milk"
```

For this simulation, determine how many cups she got right (i.e., guess equals the true value):

```
cup_results <- cups == guesses
cup_results

## [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

```
n_right <- sum(cup_results)
n_right
```

```
## [1] 2
```

Write a function that will run one simulation. It takes the argument `n_cups`—in real life, they used eight cups (`n_cups = 8`). Note that this function just wraps the code we just walked through.

```
sim_null_tea <- function(n_cups){
  cups <- sample(rep(c("milk", "tea"), each = n_cups / 2))
  guesses <- sample(rep(c("milk", "tea"), each = n_cups / 2))
  cup_results <- cups == guesses
  n_right <- sum(cup_results)
  return(n_right)
}
sim_null_tea(n_cups = 8)
```

```
## [1] 2
```

Now, we need to run a lot of simulations, to see what happens on average if she guesses. You can use the `replicate` function to do that.

```
## Generic code
replicate(n = [number of replications to run],
          eval = [code to replicate each time])
```

```
tea_sims <- replicate(5, sim_null_tea(n_cups = 8))
tea_sims
```

```
## [1] 2 4 0 2 2
```

This call gives a vector with the number of cups she got right for each simulation. You can replicate the simulation many times to get a better idea of what to expect if she just guesses, including what percent of the time she gets all cups right.

```
tea_sims <- replicate(1000, sim_null_tea(n_cups = 8))
mean(tea_sims)
```

```
## [1] 3.994
```

```
quantile(tea_sims, probs = c(0.025, 0.975))
```

```
##   2.5% 97.5%
##      2       6
```

```
mean(tea_sims == 8)
```

```
## [1] 0.013
```

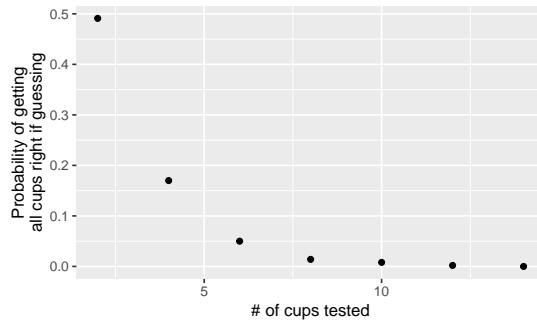
Now we'd like to know, for different numbers of cups of tea, what is the probability that the lady will get all cups right?

For this, we can apply the replication code across different values of `n_cups`:

```
n_cups <- seq(from = 2, to = 14, by = 2)
perc_all_right <- sapply(n_cups, FUN = function(n_cups){
  cups_right <- replicate(1000, sim_null_tea(n_cups))
  out <- mean(cups_right == n_cups)
  return(out)
})
perc_all_right
```

```
## [1] 0.491 0.170 0.050 0.014 0.008 0.002 0.000
```

```
tea_sims <- data_frame(n_cups, perc_all_right)
ggplot(tea_sims, aes(x = n_cups, y = perc_all_right)) +
  geom_point() + xlab("# of cups tested") +
  ylab("Probability of getting\\nall cups right if guessing")
```



You can answer this question analytically using the hypergeometric distribution:

$$P(n_{11} = t) = \frac{\binom{n_1}{t} \binom{n_{2+}}{n_1-t}}{\binom{n}{n_1}}$$

| | Guessed milk | Guessed tea | Total |
|-------------|----------------|----------------|----------------|
| Really milk | \$n_{11} | \$n_{12} | \$n_{1+} = 4\$ |
| Really tea | \$n_{21} | \$n_{22} | \$n_{2+} = 4\$ |
| Total | \$n_{+1} = 4\$ | \$n_{+2} = 4\$ | \$n = 8\$ |

In R, you can use `dhyper` to get the density of the hypergeometric function:

```
dhyper(x = [# of cups she guesses have milk first that do],
       m = [# of cups with milk first],
       n = [# of cups with tea first],
       k = [# of cups she guesses have milk first])
```

Probability she gets three “milk” cups right if she’s just guessing and there are eight cups, four with milk first and four with tea first:

```
dhyper(x = 3, m = 4, n = 4, k = 4)
```

```
## [1] 0.2285714
```

Probability she gets three or more “milk” cups right if she’s just guessing:

```
dhyper(x = 3, m = 4, n = 4, k = 4) +
  dhyper(x = 4, m = 4, n = 4, k = 4)
```

```
## [1] 0.2428571
```

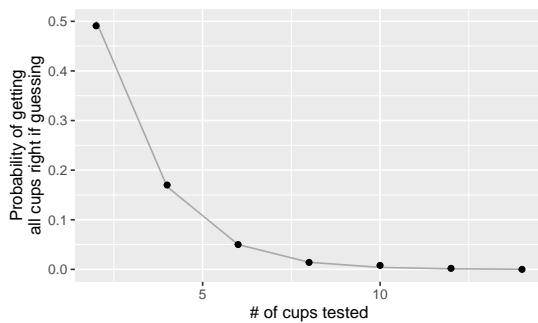
Other density functions:

- `dnorm`: Normal
- `dpois`: Poisson
- `dbinom`: Binomial
- `dchisq`: Chi-squared
- `dt`: Student’s t
- `dunif`: Uniform

You can get the analytical result for each of the number of cups we simulated and compare those values to our simulations:

```
analytical_results <- data_frame(n_cups = seq(2, 14, 2)) %>%
  mutate(perc_all_right = dhyper(x = n_cups / 2,
                                 m = n_cups / 2,
                                 n = n_cups / 2,
                                 k = n_cups / 2))
```

```
ggplot(analytical_results, aes(x = n_cups, y = perc_all_right)) +
  geom_line(color = "darkgray") +
  geom_point(data = tea_sims) + xlab("# of cups tested") +
  ylab("Probability of getting all cups right if guessing")
```



For more on this story (and R.A. Fisher), see:

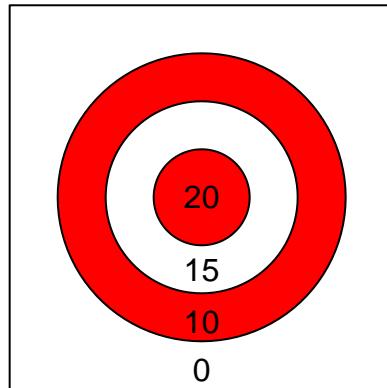
- *The Lady Tasting Tea: How Statistics Revolutionized Science in the Twentieth Century.* David Salsburg.
- *The Design of Experiments.* Ronald Fisher.
- <https://priceconomics.com/why-the-father-of-modern-statistics-didnt-believe/>

11.1.2 Playing darts

Research question: Is a person skilled at playing darts?

Here's our dart board— the numbers are the number of points you win for a hit in each area.

```
##  
## Attaching package: 'plotrix'  
  
## The following object is masked from 'package:scales':  
##  
##     rescale
```



First, what would we expect to see if the person we test has no skill at playing darts?

Questions to consider:

- *What would the dart board look like under the null (say the person throws 20 darts for the experiment)?*
- *About what do you think the person's mean score would be if they had no skill at darts?*
- *What are some ways to estimate or calculate the expected mean score under the null?*

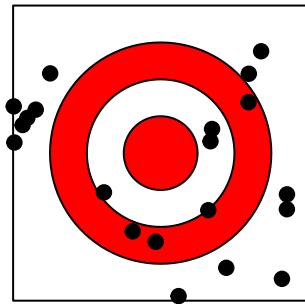
Let's use R to answer the first question: what would the null look like?

First, create some random throws (the square goes from -1 to 1 on both sides):

```
n.throws <- 20
throw.x <- runif(n.throws, min = -1, max = 1)
throw.y <- runif(n.throws, min = -1, max = 1)
head(cbind(throw.x, throw.y))
```

```
##           throw.x     throw.y
## [1,] -0.7486924  0.5402985
## [2,]  0.3227515 -0.3867515
## [3,] -0.9956136  0.3168441
## [4,]  0.8555210 -0.3801982
## [5,] -0.1894219 -0.5303237
## [6,]  0.4446527 -0.7771359
```

```
plot(c(-1, 1), c(-1,1), type = "n", asp=1,
      xlab = "", ylab = "", axes = FALSE)
rect( -1, -1, 1, 1)
draw.circle( 0, 0, .75, col = "red")
draw.circle( 0, 0, .5, col = "white")
draw.circle( 0, 0, .25, col = "red")
points(throw.x, throw.y, col = "black", pch = 19)
```



Next, let's tally up the score for this simulation of what would happen under the null.

To score each throw, we calculate how far the point is from $(0, 0)$, and then use the following rules:

- **20 points:** $0.00 \leq \sqrt{x^2 + y^2} \leq .25$
- **15 points:** $.25 < \sqrt{x^2 + y^2} \leq .50$
- **10 points:** $.50 < \sqrt{x^2 + y^2} \leq .75$
- **0 points:** $.75 < \sqrt{x^2 + y^2} \leq 1.41$

Use these rules to “score” each random throw:

```
throw.dist <- sqrt(throw.x^2 + throw.y^2)
head(throw.dist)
```

```
## [1] 0.9232891 0.5037313 1.0448142 0.9361981 0.5631375 0.8953526
```

```
throw.score <- cut(throw.dist,
                     breaks = c(0, .25, .5, .75, 1.5),
                     labels = c("20", "15", "10", "0"),
                     right = FALSE)
head(throw.score)
```

```
## [1] 0 10 0 0 10 0
## Levels: 20 15 10 0
```

Now that we've scored each throw, let's tally up the total:

```
table(throw.score)
```

```
## throw.score
## 20 15 10 0
## 0 3 4 13
```

```
mean(as.numeric(as.character(throw.score)))
```

```
## [1] 4.25
```

So, this just showed *one* example of what might happen under the null. If we had a lot of examples like this (someone with no skill throwing 20 darts), what would we expect the mean scores to be?

Questions to consider:

- *How can you figure out the expected value of the mean scores under the null (that the person has no skill)?*
- *Do you think that 20 throws will be enough to figure out if a person's mean score is different from this value, if he or she is pretty good at darts?*
- *What steps do you think you could take to figure out the last question?*
- *What could you change about the experiment to make it easier to tell if someone's skilled at darts?*

How can we figure this out?

- **Theory.** Calculate the expected mean value using the expectation formula.

- **Simulation.** Simulate a lot of examples using R and calculate the mean of the mean score from these.

The expected value of the mean, $E[\bar{X}]$, is the expected value of X , $E[X]$. To calculate the expected value of X , use the formula:

$$E[X] = \sum_x xp(x)$$

$$E[X] = 20 * p(X = 20) + 15 * p(X = 15) + 10 * p(X = 10) + 0 * p(X = 0)$$

So we just need to figure out $p(X = x)$ for $x = 20, 15, 10$.

(In all cases, we're dividing by 4 because that's the area of the full square, 2^2 .)

- $p(X = 20)$: Proportional to area of the smallest circle, $(\pi * 0.25^2)/4 = 0.049$
- $p(X = 15)$: Proportional to area of the middle circle minus area of the smallest circle, $\pi(0.50^2 - 0.25^2)/4 = 0.147$
- $p(X = 10)$: Proportional to area of the largest circle minus area of the middle circle, $\pi(0.75^2 - 0.50^2)/4 = 0.245$
- $p(X = 0)$: Proportional to area of the square minus area of the largest circle, $(2^2 - \pi * 0.75^2)/4 = 0.558$

As a double check, if we've done this right, the probabilities should sum to 1:

$$0.049 + 0.147 + 0.245 + 0.558 = 0.999$$

$$E[X] = \sum_x xp(x)$$

$$E[X] = 20 * 0.049 + 15 * 0.147 + 10 * 0.245 + 0 * 0.558$$

$$E[X] = 5.635$$

Remember, this also gives us $E[\bar{X}]$.

Now it's pretty easy to also calculate $var(X)$ and $var(\bar{X})$:

$$Var(X) = E[(X - \mu)^2] = E[X^2] - E[X]^2$$

$$E[X^2] = 20^2 * 0.049 + 15^2 * 0.147 + 10^2 * 0.245 + 0^2 * 0.558 = 77.18$$

$$Var(X) = 77.175 - (5.635)^2 = 45.42$$

$$Var(\bar{X}) = \sigma^2/n = 45.42/20 = 2.27$$

Note that we can use the Central Limit Theorem to calculate a 95% confidence interval for the mean score when someone with no skill (null hypothesis) throws 20 darts:

```
5.635 + c(-1, 1) * qnorm(.975) * sqrt(2.27)
```

```
## [1] 2.682017 8.587983
```

We can check our math by running simulations— we should get the same values of $E[\bar{X}]$ and $Var(\bar{X})$ (which we can calculate directly from the simulations using R).

```
n.throws <- 20
n.sims <- 10000

x.throws <- matrix(runif(n.throws * n.sims, -1, 1),
                     ncol = n.throws, nrow = n.sims)
y.throws <- matrix(runif(n.throws * n.sims, -1, 1),
                     ncol = n.throws, nrow = n.sims)
dist.throws <- sqrt(x.throws^2 + y.throws^2)
score.throws <- apply(dist.throws, 2, cut,
                       breaks = c(0, .25, .5, .75, 1.5),
                       labels = c("20", "15", "10", "0"),
                       right = FALSE)
```

```
dist.throws[1:3,1:5]
```

```
##          [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 1.3134651 1.1082144 0.9241944 0.8097362 0.8041162
## [2,] 0.5190076 1.2699996 0.9267038 1.1626113 0.9247417
## [3,] 0.7869685 0.6322061 0.7960028 0.8003027 0.7774120
```

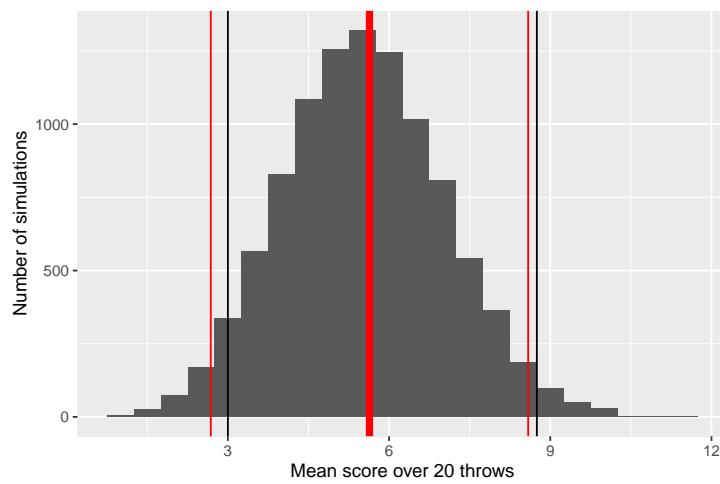
```
score.throws[1:3,1:5]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "0"  "0"  "0"  "0"  "0"
## [2,] "10" "0"  "0"  "0"  "0"
## [3,] "0"  "10" "0"  "0"  "0"

mean.scores <- apply(score.throws, MARGIN = 1,
                      function(x){
                        out <- mean(as.numeric(
                                      as.character(x)))
                        return(out)
                      })
head(mean.scores)

## [1] 3.50 2.50 4.75 3.50 5.00 5.25

## Warning: `qplot()` was deprecated in ggplot2 3.4.0.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



Let's check the simulated mean and variance against the theoretical values:

```
mean(mean.scores) ## Theoretical: 5.635
```

```
## [1] 5.6527
```

```
var(mean.scores) ## Theoretical: 2.27
```

```
## [1] 2.240119
```

11.1.3 Simulations in research

Simulations in the wild (just a few examples):

- The Manhattan Project
- US Coast Guard search and rescue
- Infectious disease modeling

11.2 Other computationally-intensive approaches

11.2.1 Bootstrap and friends

- **Bootstrapping:** Sample the dataset with replacement and reestimate the statistical parameter(s) each time.
- **Jackknifing:** Take out one observation at a time and reestimate the statistical parameter(s) with the rest of the data.
- **Permutation tests:** See how unusual the result from the data is compared to if you shuffle your data (and so remove any relationship in observed data between variables).
- **Cross-validation:** See how well your model performs if you pick a subset of the data, build the model just on that subset, and then test how well it predicts for the rest of the data, and repeat that many times.

11.2.2 Bayesian analysis

Suggested books for learning more about Bayesian analysis in R:

- *Doing Bayesian Data Analysis, Second Edition: A Tutorial with R, JAGS, and Stan.* John Kruschke.

- *Statistical Rethinking: A Bayesian Course with Examples in R and Stan.* Richard McElreath.
- *Bayesian Data Analysis, Third Edition.* Andrew Gelman et al.

R can tap into software for Bayesian analysis:

- BUGS
- JAGS
- STAN

11.2.3 Ensemble models and friends

- **Bagging:** Sample data with replacement and build a tree model. Repeat many times. To predict, predict from all models and take the majority vote.
- **Random forest:** Same as bagging, for picking each node of a tree, only consider a random subset of variables.
- **Boosting:** Same as bagging, but “learn” from previous models as you build new models.
- **Stacked models:** Build many different models (e.g., generalized linear regression, Naive Bayes, k-nearest neighbors, random forest, ...), determine weights for each, and predict using weighted predictions combined from all models.

For more on these and other machine learning topics, see:

- *An Introduction to Statistical Learning.* Gareth James, Robert Tibshirani, and Trevor Hastie.
- The `caret` package: <http://topepo.github.io/caret/index.html>
- For many examples of predictive models like this built with R (and Python): <https://www.kaggle.com>

Chapter 12

Reporting data results #3

Download a pdf of the lecture slides covering this topic.

Slides for the second half of these week are available to download in an HTML format. Save this file as HTML, and then you should be able to open the file in a web browser to see the presentation.

12.1 Shiny web apps

12.1.1 Resources for learning Shiny

There is an excellent tutorial to get you started here at RStudio. There are also several great sites that show you both Shiny examples and their code, here and here.

Many of the examples and ideas in the course notes this week come directly or are adapted from RStudio's Shiny tutorial.

To start, Shiny has several example apps that you can try out. These are all available through your R session once you install the Shiny package. You can make them available to your R session using the command `system.file()`:

```
install.packages("shiny")
library(shiny)
system.file("examples", package = "shiny")
```

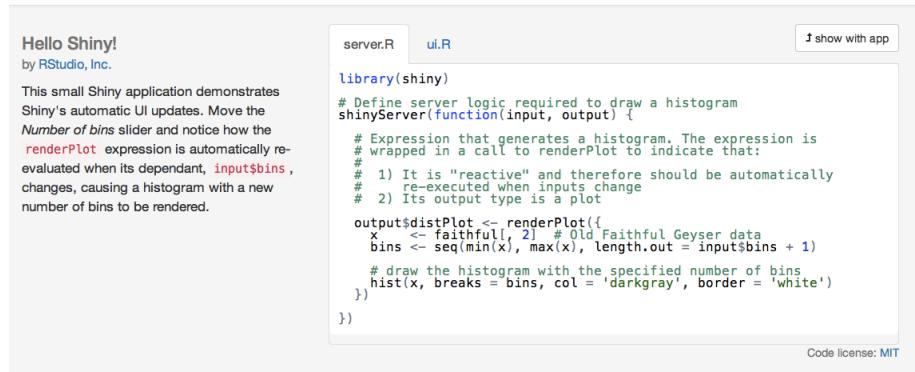
12.1.2 Basics of Shiny apps

Once you have Shiny installed, you can run the examples using the `runExample()` command. For example, to run the first example, you would run:

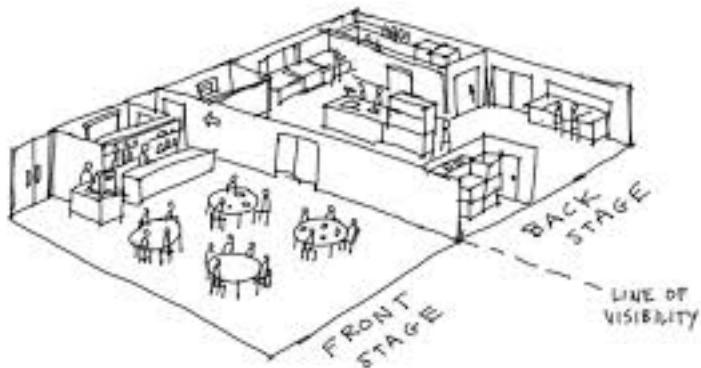
```
runExample("01_hello")
```

This is a histogram that lets you adjust the number of bins using a slider bar. Other examples are: `02_text`, `03_reactivity`, `04_mpg`, `05_sliders`, `06_tabsets`, `07_widgets`, `08_html`, `09_upload`, `10_download`, and `11_timer`.

When you run any of these, a window will come up in your R session that shows the Shiny App, and your R session will pay attention to commands it gets from that application until you close the window. Notice that if you scroll down, you'll be able to see the code that's running behind the application:

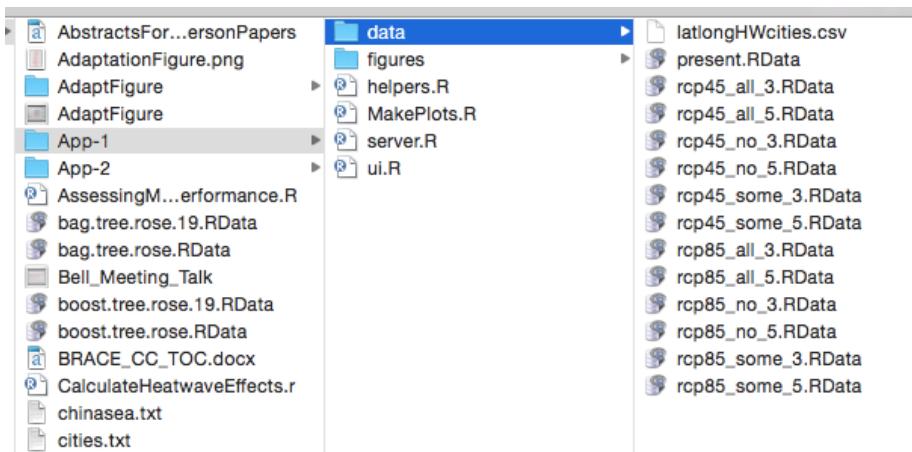


Generally, each application takes two files: (1) a user interface file and (2) a server file. You can kind of think of the two elements of an R shiny app, the user interface and the server, as two parts of a restaurant.



The user interface is the dining area. This is the only place the customer every sees. It's where the customer makes his order, and it's also where the final product comes out for him to consume. The server is the kitchen. It takes the order, does all the stuff to make it happen, and then sends out the final product back to the dining area.

At its heart, an R shiny app is just a directory on your computer or a server with these two files (as well as any necessary data files) in it. For example, here's a visual of an App I wrote to go with a paper:



This has the heart of the application (`server.R` and `ui.R`) plus a couple of R helper files and subdirectories with some figures and data that I'm using in the that application. If I open either of the main files in RStudio, I can run the

application locally using a button at the top of the file called “Run App”. Once I have the App running, if I have an account for the Shiny server, I can choose to “Publish” the application to the Shiny server, and then anyone can access and use it online (this service is free up to a certain number of visitors per time—unless you make something that is very popular, you should be well within the free limit).

12.1.3 server.R file

The server file will be named `server.R`. This file tells R what code to run with the inputs it gets from a user making certain selections. For example, for the histogram example, this file tells R how to re-draw a histogram of the data with the number of bins that the user specified on the slider on the application. Once you get through all the code for what to do, this file also will have code telling R what to send back to the application for the user to see (in this case, a picture of a histogram made with the specified number of bars). Here is the code in the `server.R` file for the histogram example:

```
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  # 1) It is "reactive" and therefore should be automatically
  #    re-executed when inputs change
  # 2) Its output type is a plot

  output$distPlot <- renderPlot({
    x      <- faithful[, 2] # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
})
```

Notice that some of the “interior” code here looks very familiar and should remind you of code we’ve been learning about this class. For example, this file has within it some code to figure out the breaks for histogram bins, based on how many total bins you want, and draw a histogram with those bin breaks:

```

x   <- faithful[, 2] # Old Faithful Geyser data
bins <- seq(min(x), max(x), length.out = input$bins + 1)

# draw the histogram with the specified number of bins
hist(x, breaks = bins, col = 'darkgray', border = 'white')

```

This code is then “wrapped” in two other functions. First, this code is generating a plot that will be posted to the application, so it’s wrapped in a `renderPlot` function to send that plot as output back to the application:

```

output$distPlot <- renderPlot({
  x   <- faithful[, 2] # Old Faithful Geyser data
  bins <- seq(min(x), max(x), length.out = input$bins + 1)

  # draw the histogram with the specified number of bins
  hist(x, breaks = bins, col = 'darkgray', border = 'white')
})

```

Notice that this code is putting the results of `renderPlot` into a slot of the object `output` named `distPlot`. We could have used any name we wanted to here, not just `distPlot`, for the name of the slot where we’re putting this plot, but it is important to put everything into an object called `output`. Now that we’ve rendered the plot and put it in that slot of the `output` object, we’ll be able to refer to it by its name in the user interface file, when we want to draw it somewhere there.

All of this is wrapped up in another wrapper:

```

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  # 1) It is "reactive" and therefore should be automatically
  #    re-executed when inputs change
  # 2) Its output type is a plot

  output$distPlot <- renderPlot({
    x   <- faithful[, 2] # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
  })
}

```

```

# draw the histogram with the specified number of bins
hist(x, breaks = bins, col = 'darkgray', border = 'white')
})

})

```

The `server.R` file also has a line to load the `shiny` package. You should think of apps as being like Rmd files— if there are any packages or datasets that you need to use in the code in that file, you need to load it within the file, because R won't check in your current R session to find it when it runs the file.

12.1.4 ui.R file

The other file that a Shiny app needs is the user interface file (`ui.R`). This is the file that describes how the application should look. It will write all the buttons and sliders and all that you want for the application interface. This is also where you specify what you want to go where and put in any text that you want to show up.

For example, here is the `ui.R` file for the histogram example:

```

library(shiny)

# Define UI for application that draws a histogram
shinyUI(fluidPage(

  # Application title
  titlePanel("Hello Shiny!"),

  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                 "Number of bins:",
                 min = 1,
                 max = 50,
                 value = 30)
    ),
    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
)

```

```

    )
)
))
```

There are a few things to notice with this code. First, there is some code that tells the application to show the results from the `server.R` code. For example, the following code tells R to show the histogram that we put into the `output` object in the `distPlot` slot and to put that graph in the main panel of the application:

```

mainPanel(
  plotOutput("distPlot")
)
```

Other parts of the `ui.R` code will tell the application what kinds of choice boxes and sliders to have on the application, and what default value to set each to. For example, the following code tells the application that it should have a slider bar that can take a minimum value of 1 and a maximum value of 50. When you first open the application, its default value should be 30. It should be annotated with the text “Number of bins:”. Whatever value is selected should be saved to the `bins` slot of the `input` object (just like we’re using the `output` object to get things out of the server and printed to the application, we’re using the `input` object to get things that the user chooses from the application interface to the server where we can run R code).

```

sliderInput("bins",
  "Number of bins:",
  min = 1,
  max = 50,
  value = 30)
```

12.1.5 Making a Shiny app

The first step in making a Shiny app is to make a new directory somewhere and to create R scripts for that directory called `server.R` and `ui.R`. You can just make these two files the normal way—within RStudio, do “New File”, “R Script”, and then just save them with the correct names to the directory you created for the App. Once you save a file as `ui.R`, notice that you’ll have a button in the top right of the file called “Run App”. When you’re ready to run your application, you can either use this button or use the command `runApp`.

12.1.6 Starting with the ui.R file

Next, you'll need to put code in these files. I would suggest starting with the `ui.R` files. This file is where you get to set up how the application looks and how people will be able to interact with it. That means that this is a good place to start because it's both quickly fulfilling (I made something pretty!) and also because you need to have an idea of what inputs and outputs you need before you can effectively make the server file to tell R what to do. In truth, though, you'll be going back and forth quite a bit between these two files as you edit your application.

In the `ui.R` file, everything needs to be wrapped in a `shinyUI()` function, and then most things will be wrapped in other functions within that to set up different panels. For example, here's a very basic `ui.R` file (adapted directly from the RStudio tutorial) that shows a very basic set up for a user interface:

```
shinyUI(fluidPage(
  titlePanel("Tweets during Paris Attack"),
  sidebarLayout(
    sidebarPanel("Select hashtag to display"),
    mainPanel("Map of tweets")
  )
))
```

Notice that everything that I want to go in certain panels of the page are wrapped in functions like `sidebarPanel` and `mainPanel` and `titlePanel`. Everything in this file will be divided up by the place you want it to go in the final version.

As a note, the sidebar layout (a sidebar on one side and one main panel) is the simplest possible Shiny layout. You can do fancier layouts if you want by using different functions like `fluidRow()` and `navBarPage()`. RStudio has a layout help page with very detailed instructions and examples to help you figure out how to do other layouts.

If I run this `ui.R`, even if my `server.R` file only includes the line `shinyServer(function(input, output) { })`, I'll get the following version application:

The screenshot shows a Shiny application window. At the top, there's a header bar with three colored dots (red, yellow, green), the URL 'http://127.0.0.1:5157', a 'Open in Browser' button, and a 'Publish' dropdown. The main title of the app is 'Tweets during Paris Attack'. Below the title, there's a sidebar with a dropdown menu labeled 'Select hashtag to display'. To the right of the sidebar, the main content area is titled 'Map of tweets'.

This doesn't have anything interactive on it, and it isn't using R at all, but it shows the basics of how the syntax of the `ui.R` file works. As a note, I don't have all of the functions for this, like `fluidPage` and `titlePanel` memorized. When I'm working on this file, I'll either look to example code from other Shiny apps or look at RStudio's help for Shiny applications until I can figure out what syntax to use to do what I want to do.

12.1.7 Adding in widgets

Next, I'll add in some cool things that will let the user interact with the application. In this case, I'd like to have a slider bar so people can chose the range of time for the tweets that are shown. I'd also like to have a selection box so that users can look at maps of specific hashtags or terms. To add these on (they won't be functional, yet, but they'll be there!), I can edit the `ui.R` script to the following:

```
shinyUI(fluidPage(
  titlePanel("Tweets during Paris Attack"),
  sidebarLayout(position = "right",
    sidebarPanel("Choose what to display",
      sliderInput(inputId = "time_range",
        label = "Select the time range: ",
        value = c(as.POSIXct("2015-11-13 00:00:00",
          tz = "CET"),
        as.POSIXct("2015-11-14 12:00:00",
          tz = "CET")),
        min = as.POSIXct("2015-11-13 00:00:00", tz = "CET"),
        max = as.POSIXct("2015-11-14 12:00:00", tz = "CET"),
        step = 60,
        timeFormat = "%dth %H:%M",
        timezone = "+0100")),
      mainPanel("Map of tweets")
    )
  )))

```

The important part of this is the new `sliderInput` call, which sets up a slider bar that users can use to specify certain time ranges to look at. Here is what the interface of the app looks like now:



If I open this application, I can move the slider bar around, but I it isn't actually sending any information to R yet.

The heart of this new addition to the `ui.R` file is this:

```
sliderInput(inputId = "time_range",
            label = "Select the time range: ",
            value = c(as.POSIXct("2015-11-13 00:00:00", tz = "CET"),
                      as.POSIXct("2015-11-14 12:00:00", tz = "CET")),
            min = as.POSIXct("2015-11-13 00:00:00", tz = "CET"),
            max = as.POSIXct("2015-11-14 12:00:00", tz = "CET"),
            step = 60,
            timeFormat = "%dth %H:%M",
            timezone = "+0100")
```

This is all within the function `sliderInput` and is all being used to set up the slider bar. `"time_range"` is the name I'm giving the input I get from this. Later, when I write my server code, I'll be able to pull the values that the user suggested from the `time_range` slot of the `input` object. The next thing is the label. This is what I want R to print right before it gives the slider bar. The `value` object says which values I want to be the defaults on the slider. This is where the slider positions will be when someone initially opens the application. `min` and `max` give the highest and lowest values that will show up on the slider bar. For this application, I'm calling them as `POSIXct` objects because I want to do this for times rather than numbers. `step` says how big of an increment I want the slider to advance by when someone is pulling it. If the values of the slider are times, then the default unit for this is seconds, so I'm saying to have a step size of one minute. The `timeFormat` says how I want the time to print

out at the interface and the `timezone` says what time zone I want time values to display in.

Things like this slider bar are called “Control Widgets”, and there’s a whole list of them in the third lesson of RStudio’s Shiny tutorial. There are also examples online in the Shiny Gallery.

12.1.8 Creating output in `server.R`

Next, I’ll put some R code in the `server.R` file to create a figure and pass it through to the `ui.R` file to print out to the application interface. At first, I won’t make this figure “reactive”; that is, it won’t change at all when the user changes the slider bar. However, I will eventually add in that reactivity so that the plot changes everytime a user changes the slider bar.

I am going to create a map of all the Tweets that included certain hashtags or phrases and that were Tweeted (and geolocated) from within a five-mile radius of the center of Paris during the attacks last Friday. For this, I’m going to use data on Tweets I pulled using the `TwitteR` package, which syncs up with Twitter’s API. Here’s an example of what the data looks like (I’ve cleared out some of the extra columns I won’t use):

```
paris_twitter <- read.csv("data/App-1/data/final_tweets.csv", as.is = TRUE) %>%
  mutate(tag = factor(tag),
        created = ymd_hms(created, tz = "Europe/Paris"),
        text = iconv(text, to='ASCII//TRANSLIT'))
paris_twitter[1:2, ]
```

```
## 
## 1 RT @forza_will2006: My heart aches for the people of France. #PorteOuverte #PrayForParis #Do
## 2                           Ensemble contre la haine #jesuisparis #porteouverte #paris
##   created longitude latitude      tag
## 1 2015-11-15 01:27:56       NA       NA #PorteOuverte
## 2 2015-11-14 22:19:47  2.364184 48.86747 #PorteOuverte
```

About an equal number of these have and don’t have location data:

```
table(!is.na(paris_twitter$longitude))
```

```
## 
## FALSE  TRUE
## 10478 10683
```

Here is a table of the number of tweets under the five most-tweeted tags:

```
head(paris_twitter)

## 
## 1 RT @forza_will2006: My heart aches for the people of France. #PorteOuverte #PrayForParis
## 2                                         Ensemble contre la haine #jesuisparis #porteouverte
## 3                               Liberte, egalite, fraternite #Paris #TodosSomosParis #JeSuisParis
## 4             Stay With My French #prayforparis #parisattacks #porteouverte #france
## 5
## 6                                         RT @bodoi_music: J'ai perdu ma place dans le monde

##           created longitude latitude      tag
## 1 2015-11-15 01:27:56        NA       NA #PorteOuverte
## 2 2015-11-14 22:19:47  2.364184 48.86747 #PorteOuverte
## 3 2015-11-14 21:36:06  2.350800 48.85670 #PorteOuverte
## 4 2015-11-14 21:00:49  2.350800 48.85670 #PorteOuverte
## 5 2015-11-14 20:11:01  2.350800 48.85670 #PorteOuverte
## 6 2015-11-14 19:52:43        NA       NA #PorteOuverte

tweet_sum <- paris_twitter %>%
  dplyr::group_by_(~ tag) %>%
  dplyr::summarize_(n = ~ n(),
                    example = ~ gsub("[[:punct:]]", " ",
                                     base::sample(text, 1))) %>%
  dplyr::arrange_(~ dplyr::desc(n))

## Warning: `arrange_()` was deprecated in dplyr 0.7.0.
## i Please use `arrange()` instead.
## i See vignette('programming') for more help
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

## Warning: `summarise_()` was deprecated in dplyr 0.7.0.
## i Please use `summarise()` instead.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

## Warning: `group_by_()` was deprecated in dplyr 0.7.0.
## i Please use `group_by()` instead.
## i See vignette('programming') for more help
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
knitr::kable(tweet_sum[1:5, ],
             col.names = c("Tag", "# of Tweets", "Example Tweet"))
```

| Tag | # of Tweets | Example Tweet |
|---------------|-------------|--|
| #Paris | 7502 | RT Vince66240 Tirs a la Kalash au petit Cambodge dans le 10 eme a Paris plusie |
| #PrayForParis | 7250 | NA |
| #13novembre | 1255 | RT taimaz Situation tres tendue devant le Bataclan les curieux degages manu mi |
| #PorteOuverte | 1147 | RT fake rebel Si jamais quelqu un a besoin porteouverte Metro Maraichers L app |
| #fusillade | 765 | RT jacques lefort Coups de feu et haute tension a l angle rues de Charonne amp |

For the Tweets that are geolocated, it's possible to map the tweet locations using the following code:

```
paris_map <- get_map("paris", zoom = 12, color = "bw")

paris_locations <- c("Stade de France", "18 Rue Alibert",
                      "50 Boulevard Voltaire", "92 Rue de Charonne",
                      "Place de la Republique")
paris_locations <- paste(paris_locations, "paris france")
paris_locations <- cbind(paris_locations, geocode(paris_locations))

# Plot contour map of tweet locations
plot_map <- function(tag = "all", df = paris_twitter){
  library(ggmap)
  library(dplyr)
  df <- dplyr::select(df, tag, latitude, longitude) %>%
    filter(!is.na(longitude)) %>%
    mutate(tag = as.character(tag))

  if(tag != "all"){
    if(!(tag %in% df$tag)){
      stop(paste("That tag is not in the data. Try one of the following tags in"))
    }
    to_plot <- df[df$tag == tag, ]
  } else {
    to_plot <- df
  }

  hotel_de_ville <- to_plot$latitude == 48.85670 &
    to_plot$longitude == 2.350800
  n_hotel_de_ville <- sum(hotel_de_ville)

  if(n_hotel_de_ville == max(table(to_plot$latitude))){
```

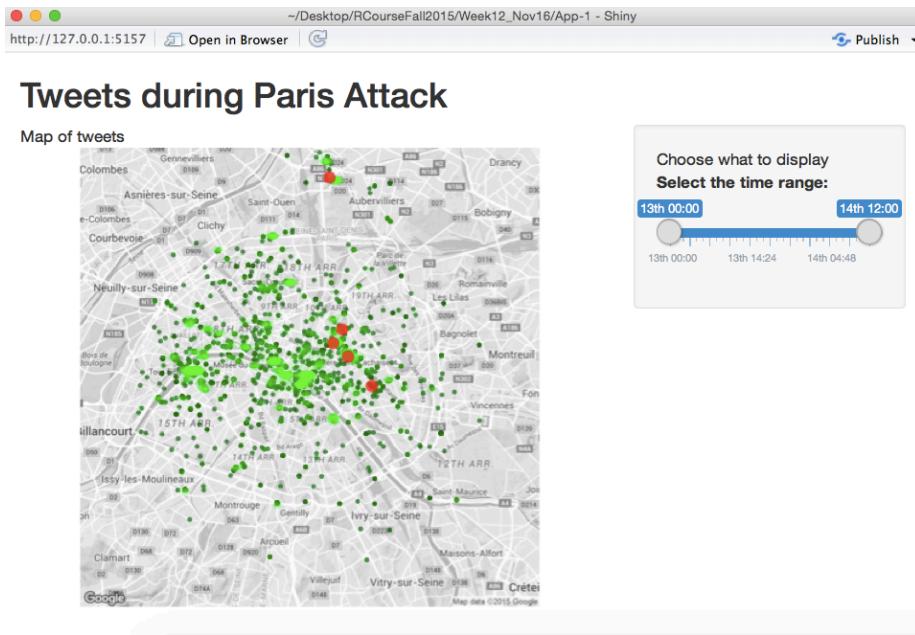
```

        hdv_index <- sample(1:nrow(to_plot))[hotel_de_ville],
                    round(n_hotel_de_ville / 2))
      to_plot <- to_plot[-hdv_index, ]
    }

my_map <- ggmap(paris_map, extent = "device") +
  geom_point(data = to_plot, aes(x = longitude, y = latitude),
             color = "darkgreen", alpha = 0.75) +
  geom_density2d(data = to_plot,
                 aes(x = longitude, latitude), size = 0.3) +
  stat_density2d(data = to_plot,
                 aes(x = longitude, y = latitude,
                     fill = ..level.., alpha = ..level..),
                 size = 0.01, bins = round(nrow(to_plot) / 3.3),
                 geom = "polygon") +
  scale_fill_gradient(low = "green", high = "yellow", guide = FALSE) +
  scale_alpha(guide = FALSE) +
  geom_point(data = paris_locations, aes(x = lon, y = lat),
             color = "red", size = 5, alpha = 0.75)
return(my_map)
}

```

To print this out in the application, I'll put all the code for the mapping function in a file called `helper.R`, source this file in the `server.R` file, and then I can just call the function within the server file. The application will look as follows after this step:



To complete this, I first changed the `server.R` file to look like this:

```
library(dplyr)
library(lubridate)

source("helper.R")

paris_twitter <- read.csv("data/final_tweets.csv", as.is = TRUE) %>%
  mutate(tag = factor(tag),
        created = ymd_hms(created, tz = "Europe/Paris"))

shinyServer(function(input, output) {
  output$twitter_map <- renderPlot({ plot_map() })
})
```

Notice a few things here:

1. I'm loading the packages I'll need for the code.
2. I'm running all the code in the `helper.R` file (which includes the function I created to plot this map) using the `source()` command.
3. I put the code to plot the map (`plot_map()`) inside the `renderPlot({})` function.
4. I'm putting the plot in a `twitter_map` slot of the `output` object.

5. All of this is going inside the call `shinyServer(function(input, output){ })`.

One other change is necessary to get the map to print on the app. I need to add code to the `ui.R` file to tell R where to plot this map on the final interface. The full file now looks like this:

```
shinyUI(fluidPage(
  titlePanel("Tweets during Paris Attack"),
  sidebarLayout(position = "right",
    sidebarPanel("Choose what to display",
      sliderInput(inputId = "time_range",
        label = "Select the time range: ",
        value = c(as.POSIXct("2015-11-13 00:00:00",
          tz = "CET"),
        as.POSIXct("2015-11-15 06:00:00",
          tz = "CET")),
        min = as.POSIXct("2015-11-13 00:00:00", tz =
        max = as.POSIXct("2015-11-14 12:00:00", tz =
        step = 60,
        timeFormat = "%dth %H:%M",
        timezone = "+0100")),
      mainPanel("Map of tweets",
        plotOutput("twitter_map")))
    )
  )))

```

The new part is where I've added the code:

```
mainPanel("Map of tweets",
  plotOutput("twitter_map"))
```

This tells R to put the plot output `twitter_map` from the `output` object in the main panel of the Shiny app.

12.1.9 Making the output reactive

Now almost all of the pieces are in place to make this graphic reactive. First, I added some options to the function in `helper.R` to let it input time ranges and

only plot the tweets within that range. Next, I need to use the values that the user selects from the slider in the call for plotting the map. To do this, I can use the values passed from the slider bar in the `input` object into the code in the `server.R` file. Here is the new code for the `server.R` file:

```
library(ggmap)
library(ggplot2)
library(dplyr)
library(lubridate)

source("helper.R")

paris_twitter <- read.csv("data/final_tweets.csv", as.is = TRUE) %>%
  mutate(tag = factor(tag))
paris_twitter$created <- as.POSIXct(paris_twitter$created, tz = "CET")

shinyServer(function(input, output) {
  output$twitter_map <- renderPlot({
    plot_map(start.time = input$time_range[1],
              end.time = input$time_range[2])
  })
})
```

The only addition from before is to use the `start.time` and `end.time` options in the `plot_map` function and to set them to the first, [1], and second, [2], values in the `time_range` slot of the `input` object. Remember that we chose to label the input from the slider bar `time_range` when we set up the `ui.R` file.

This app is saved in the directory `App-1` in this week's directory if you'd like to play around with the code. I've deployed it on shinyapps here.

12.1.10 Fancier version

I've also created a (much) fancier version of a Shiny App looking at this Twitter data that you can check out here. All the code for that is here.

12.2 htmlWidgets

12.2.1 Overview of htmlWidgets

Very smart people have been working on creating interactive graphics in R for a long time. So far, nothing coded in R has taken off in a big way.

JavaScript has developed a number of interactive graphics libraries that can be for documents viewed in a web browser. There is now a series of R packages that allow you to create plots from these JavaScript libraries from within R.

There is a website with much more on these `htmlWidgets` at <http://www.htmlwidgets.org>.

Some of the packages available to help you create interactive graphics from R using JavaScript graphics libraries:

- `leaflet`: Mapping (we'll cover this next week)
- `dygraphs`: Time series
- `plotly`: A variety of plots, including maps
- `rbokeh`: A variety of plots, including maps
- `networkD3`: Network data
- `d3heatmap`: Heatmaps
- `DT`: Data tables
- `DiagrammeR`: Diagrams and flowcharts

These packages can be used to make some pretty cool interactive visualizations for HTML output from R Markdown or Shiny (you can also render any of them in RStudio).

There are, however, a few limitations:

- Written by different people. The different packages have different styles as well as different interfaces. Learning how to use one package may not help you much with other of these packages.
- Many are still in development, often in early development.

12.2.2 `plotly` package

From the package documentation:

“Easily translate `ggplot2` graphs to an interactive web-based version and / or create custom web-based visualizations directly from R.”

- Like many of the packages today, draws on functionality external to R, but within a package that allows you to work exclusively within R.
- Allows you to create interactive graphs from R. Some of the functions extend the `ggplot2` code you've learned.
- Interactivity will only work within RStudio or on documents rendered to HTML.

The `plotly` package allows an interface to let you work with `plotly.js` code directly using R code.

`plotly.js` is an open source library for creating interactive graphs in JavaScript. This JavaScript library is built on `d3.js` (Data-Driven Documents), which is a key driver in interactive web-based data graphics today.

There are two main ways of create plots within `plotly`:

- Use one of the functions to create a customized interactive graphic:
 - `plot_ly`: Workhorse of `plotly`, renders most non-map types of graphs
 - `plot_geo`, `plot_mapbox`: Specific functions for creating `plotly` maps.
- Create a `ggplot` object and then convert it to a `plotly` object using the `ggplotly` function.

```
library(faraway)
data(worldcup)
library(plotly)
plot_ly(worldcup, type = "scatter", x = ~ Time, y = ~ Shots)
```

Just like with `ggplot2`, the mappings you need depend on the type of plot you are creating.

For example, scatterplots (`type = "scatter"`) need `x` and `y` defined, while a surface plot (`type = "surface"`) can be created with a single vector of elevation (we'll see an example in a few slides).

The help file for `plot_ly` includes a link with more documentation on the types of plots that can be made and the required mappings for each.

```
plot_ly(worldcup, type = "scatter", x = ~ Time, y = ~ Shots,
        color = ~ Position)
```

The `plotly` package is designed so you can pipe data into `plot_ly` and add elements by piping into `add_*` functions (this idea is similar to adding elements to a `ggplot` object with `+`).

```
worldcup %>%
  plot_ly(x = ~ Time, y = ~ Shots, color = ~ Position) %>%
  add_markers()
```

Some of the `add_*` functions include:

- `add_markers`
- `add_lines`
- `add_paths`
- `add_polygons`
- `add_segments`
- `add_histogram`

If you pipe to the `rangeslider` function, it allows the viewer to zoom in on part of the x range. (This can be particularly nice for time series.)

You should have a dataset available through your R session named `USAccDeaths`. This gives a monthly count of accidental deaths in the US for 1973 to 1978. This code will plot it and add a range slider on the lower x-axis.

```
plot_ly(x = time(USAccDeaths), y = USAccDeaths) %>%
  add_lines() %>%
  rangeslider()
```

For a 3-D scatterplot, add a mapping to the `z` variable:

```
worldcup %>%
  plot_ly(x = ~ Time, y = ~ Shots, z = ~ Passes,
          color = ~ Position, size = I(3)) %>%
  add_markers()
```

The `volcano` data comes with R and is in a matrix format. Each value gives the elevation for a particular pair of x- and y-coordinates.

```
dim(volcano)
```

```
## [1] 87 61
```

```
volcano[1:4, 1:4]
```

```
##      [,1] [,2] [,3] [,4]
## [1,] 100 100 101 101
## [2,] 101 101 102 102
## [3,] 102 102 103 103
## [4,] 103 103 104 104
```

```
plot_ly(z = ~volcano, type = "surface")
```

Mapping with `plotly` can build on some data that comes with base R or other packages you've likely added (or can add easily, as with the `map_data` function from `ggplot2`). For example, we can map state capitals and cities with > 40,000 people using data in the `us.cities` dataframe in the `maps` package:

```
head(maps::us.cities, 3)
```

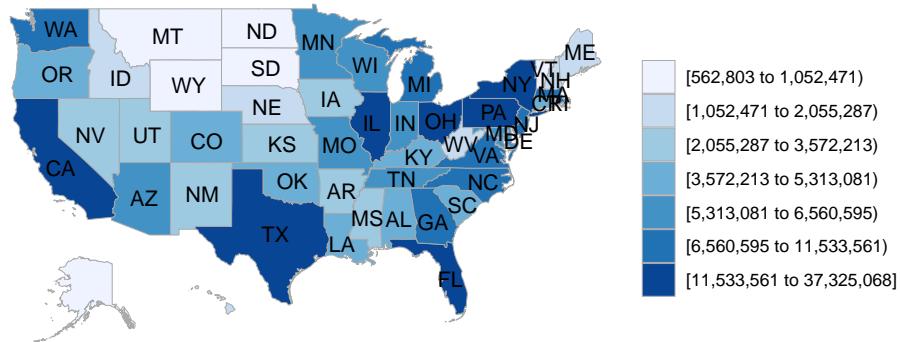
```
##      name country.etc    pop     lat     long capital
## 1 Abilene TX 113888 32.45 -99.74      0
## 2 Akron OH 206634 41.08 -81.52      0
## 3 Alameda CA  70069 37.77 -122.26      0
```

Here is code you can use to map all of these cities on a US map:

```
ggplot2::map_data("world", "usa") %>%
  group_by(group) %>% filter(-125 < long & long < -60 &
                                25 < lat & lat < 52) %>%
  plot_ly(x = ~long, y = ~lat) %>%
  add_polygons(hoverinfo = "none") %>%
  add_markers(text = ~paste(name, "<br />", pop), hoverinfo = "text",
              alpha = 0.25,
              data = filter(maps::us.cities, -125 < long & long < -60 &
                            25 < lat & lat < 52)) %>%
  layout(showlegend = FALSE)
```

You can also make choropleths interactive. Remember that we earlier created a choropleth of US state populations with the following code:

```
library(choroplothr)
data(df_pop_state)
state_choropleth(df_pop_state)
```



You can use the following code with `plotly` to make an interactive choropleth instead:

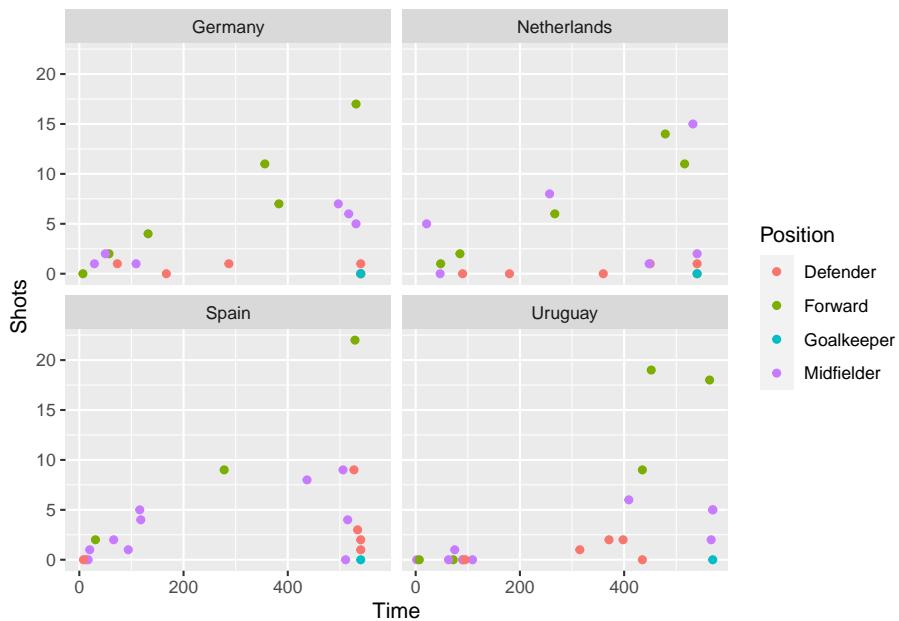
```
us_map <- list(scope = 'usa',
                 projection = list(type = 'albers usa'),
                 lakecolor = toRGB('white'))
plot_geo() %>%
  add_trace(z = df_pop_state$value[df_pop_state$region != "district of columbia"],
            text = state.name, locations = state.abb,
            locationmode = 'USA-states') %>%
  add_markers(x = state.center[["x"]], y = state.center[["y"]],
              size = I(2), symbol = I(8), color = I("white"), hoverinfo = "none") %>%
  layout(geo = us_map)
```

The other way to create a `plotly` graph is to first create a `ggplot` object and then transform it into an interactive graphic using the `ggplotly` function.

The following code can be used to plot Time versus Shots for the World Cup date in a regular, non-interactive plot:

```
shots_vs_time <- worldcup %>%
  mutate(Name = rownames(worldcup)) %>%
```

```
filter(Team %in% c("Netherlands", "Germany", "Spain", "Uruguay")) %>%
  ggplot(aes(x = Time, y = Shots, color = Position, group = Name)) +
  geom_point() +
  facet_wrap(~ Team)
shots_vs_time
```



To make the plot interactive, just pass the `ggplot` object to `ggplotly`:

```
ggplotly(shots_vs_time)
```

With R, not only can you pull things from another website using an API, you can also upload or submit things.

There is a function in the `plotly` library, `plotly_POST`, that lets you post a plot you create in R to <https://plot.ly>.

You need a `plot.ly` account to do that, but there are free accounts available.

The creator of the R `plotly` package has written a bookdown book on the package that you can read here. It provides extensive details and examples for using `plotly`.

Getting Started with D3 by Mike Dewar (a short book on D3 in JavaScript) is available for free [here](#).

12.2.3 rbokeh package

The `rbokeh` package provides an R interface to a Python interactive visualization library, Bokeh.

There is a website with many more details on using the `rbokeh` package: <https://hafen.github.io/rbokeh/>

You can find out more about the original Python library, Bokeh, at <http://bokeh.pydata.org/en/latest/>.

Here is an example of an interactive scatterplot of the World Cup data made with the `rbokeh` package:

```
library(rbokeh)
figure(width = 600, height = 300) %>%
  ly_points(Time, Shots, data = worldcup,
            color = Position, hover = list(Time, Shots))
```

This package can also be used to create interactive maps. For example, the following dataset has data on Oregon climate stations, including locations:

```
orstationc <- read.csv("data/orstationc.csv")
head(orstationc, 3)
```

```
##   station      lat      lon elev tjan tjul tann pjan pjul pann  idnum
## 1     ANT 44.917 -120.717  846  0.0 20.2  9.6   41    9  322 350197
## 2     ARL 45.717 -120.200   96  0.9 24.6 12.5   40    6  228 350265
## 3     ASH 42.217 -122.717  543  3.1 20.8 11.1   70    7  480 350304
##                               Name
## 1 ANTELOPE 1 N USA-OR
## 2 ARLINGTON USA-OR
## 3 ASHLAND 1 N USA-OR
```

You can use the following code to create an interactive map of these climate stations:

```
gmap(lat = 44.1, lng = -120.767, zoom = 5, width = 500, height = 428) %>%
  ly_points(lon, lat, data = orstationc, alpha = 0.8, col = "red",
            hover = c(station, Name, elev, tann))
```

You can get very creative with this package. The following code comes directly from the help documentation for the package and shows how to use this to create an interactive version of the periodic table:

```
# prepare data
elements <- subset(elements, !is.na(group))
elements$group <- as.character(elements$group)
elements$period <- as.character(elements$period)

# add colors for groups
metals <- c("alkali metal", "alkaline earth metal", "halogen",
           "metal", "metalloid", "noble gas", "nonmetal", "transition metal")
colors <- c("#a6cee3", "#1f78b4", "#fdbf6f", "#b2df8a", "#33a02c",
           "#bbbb88", "#baa2a6", "#e08e79")
elements$color <- colors[match(elements$metal, metals)]
elements$type <- elements$metal

# make coordinates for labels
elements$symx <- paste(elements$group, ":0.1", sep = "")
elements$numbery <- paste(elements$period, ":0.8", sep = "")
elements$massy <- paste(elements$period, ":0.15", sep = "")
elements$namey <- paste(elements$period, ":0.3", sep = "")

# create figure
p <- figure(title = "Periodic Table", tools = c("resize", "hover"),
             ylim = as.character(c(7:1)), xlim = as.character(1:18),
             xgrid = FALSE, ygrid = FALSE, xlab = "", ylab = "",
             height = 445, width = 800) %>%

# plot rectangles
ly_crect(group, period, data = elements, 0.9, 0.9,
          fill_color = color, line_color = color, fill_alpha = 0.6,
          hover = list(name, atomic.number, type, atomic.mass,
                      electronic.configuration)) %>%

# add symbol text
ly_text(symx, period, text = symbol, data = elements,
        font_style = "bold", font_size = "10pt",
        align = "left", baseline = "middle") %>%

# add atomic number text
ly_text(symx, numbery, text = atomic.number, data = elements,
        font_size = "6pt", align = "left", baseline = "middle") %>%
```

```
# add name text
ly_text(symx, namey, text = name, data = elements,
       font_size = "4pt", align = "left", baseline = "middle") %>%
# add atomic mass text
ly_text(symx, massy, text = atomic.mass, data = elements,
       font_size = "4pt", align = "left", baseline = "middle")

p
```

12.2.4 dygraphs package

The `dygraphs` package lets you create interactive time series plots from R using the `dygraphs` JavaScript library.

The main function syntax is fairly straightforward. Like many of these packages, it allows piping.

There is a website with more information on using `dygraphs` available at <http://rstudio.github.io/dygraphs/index.html>.

For example, here is the code to plot monthly deaths from lung diseases in the UK in the 1970s.

```
library(dygraphs)
lungDeaths <- cbind(mdeaths, fdeaths)
dygraph(lungDeaths) %>%
  dySeries("mdeaths", label = "Male") %>%
  dySeries("fdeaths", label = "Female")
```

12.2.5 DT package

The `DT` package provides a way to create interactive tables in R using the JavaScript `DataTables` library.

We've already seen some examples of this output in some of the Shiny apps I showed last week. You can also use this package to include interactive tables in R Markdown documents you plan to render to HTML.

There is a website with more information on this package at <http://rstudio.github.io/DT/>.

```
library(DT)
datatable(worldcup)
```

12.2.6 Creating your own widget

If you find a JavaScript visualization library and would like to create bindings to R, you can create your own package for a new htmlWidget.

There is advice on creating your own widget for R available at http://www.htmlwidgets.org/develop_intro.html.

Chapter 13

Reproducible research #3

Download a pdf of the lecture slides covering this topic.

13.1 Overview of R packages

13.1.1 What is an R package?

- From Writing R Extensions: “A directory of files which extend R”.
- Files bundled together using `tar` and compressed using `gzip`. The file extension is `.tar.gz`. These are the *source files* for the package, which then must be installed from this source code locally prior to use.
- Sometimes also called an *extension* of R.

Example R package:

| Folders | Documents | Developer |
|---|--|--|
| <p>weathermetrics</p> <p>PDF Documents</p> <p>weathermetrics.pdf</p> <p>Other</p> <p>weathermetrics_1.2.0.tar.gz</p> <p>weathermetrics_1.2.2.tar.gz</p> | <p>cran-comments.md</p> <p>NEWS.md</p> <p>README.md</p> <p>Folders</p> <p>data</p> <p>inst</p> <p>man</p> <p>R</p> <p>vignettes</p> <p>Other</p> <p>DESCRIPTION</p> <p>NAMESPACE</p> <p>README.Rmd</p> <p>weathermetrics.Rproj</p> | <p>data.R</p> <p>heat_index.R</p> <p>moisture_conversions.R</p> <p>rainmeasure_conversion.R</p> <p>temperature_conversions.R</p> <p>weathermetrics.R</p> <p>wind_conversions.R</p> |

You can also have “binary packages” for a certain operating system. From Writing R Extensions:

A binary package is “a zip file or tarball containing the files of an installed package which can be unpacked rather than installing from sources.”

Consider developing software when:

- You have developed a new method you want to share
- You have data you’d like to make publicly available
- You find yourself doing the same task repeatedly

Why create an R package?

- Share some functions broadly
- Share some functions with a small group
- Create a version of code for yourself that’s more organized and easier to use
 - Includes documentation (vignettes, help files)
 - Function names linked to package namespace
 - Once library is installed, can load easily

13.1.2 Example: NMMAPS package



Source: www.ihapss.jhsph.edu

Contents of NMMAPS package:

NMMAPSdata pack

Data

- akr
- albu
- Anch
and 105 other US cities
- *Meta-data on cities
(population,
location, counties,
Census variables)*

Functions

- readCity
- getMetaData
*and various other
functions for different
versions of the package*

Research impacts of NMMAPS package (*Source: Barnett, Huang, and Turner, “Benefits of Publicly Available Data”, Epidemiology 2012*):

- As of November 2011, 67 publications had been published using this data, with 1,781 citations to these papers
- Research using NMMAPS has been used by the US EPA in creating regulatory impact statements for air pollution (particulates and ozone)
- “Thanks to NMMAPS, there is probably no other country in the world with a greater understanding of the health effects of air pollution and heat waves in its population.”

13.1.3 Sharing an R package

If you want to share your R package, there are a number of ways you can do that:

- CRAN
- GitHub
- Bioconductor: “Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data.” (from the Bioconductor website.)
- Other repositories
 - Private(-ish) repositories: e.g., ROpenSci’s repository (for more, see <https://ropensci.org/blog/blog/2015/08/04/a-drat-repository-for-ropensci>)
 - `drat` repository: Make your own R package repository, including through GitHub pages.
- Compressed file: You can save a source tarball or binary package file with others without posting to a repository.

Sharing on CRAN:

- Traditional way to share an R package widely
- Easiest way for others to get your package (`install.packages`)
- Some barriers:
 - Size constraint on packages (5 MB)
 - Must follow CRAN policies
 - All packages must pass a submission process. This is not a guarantee that a package does what it says, just a check that required files are where they should be and that the package more or less doesn’t break things.

Sharing on GitHub:

GitHub is becoming more and more common as a place to share R packages, both development packages that eventually are posted to CRAN and packages that are never submitted to CRAN.

- No restrictions / submission requirements
- GitHub repository size restrictions (1 GB, no files over 100 MB) much larger than CRAN package size restrictions (5 MB)
- GitHub packages can be installed using `install_github` from the `devtools` package
 - Requires `devtools` package, which has some set-up requirements (XCode for Mac, Rtools for Windows)
- Packages on CRAN cannot depend on packages available only on GitHub

13.1.4 Package names

The format requirements for a package name are, based on Writing R Extensions:

“This should contain only (ASCII) letters, numbers and dot, have at least two characters and start with a letter and not end in a dot.”

Hadley Wickham’s additional guidelines:

- Make it easy to Google.
- Make it all uppercase or all lower case
- Base it on a word that’s easy to remember, but then tweak the spelling to make it unique (and easier to Google).
- Abbreviate.
- Add an “r”.

13.1.5 Package maintainer

A package can have many authors, but only one maintainer. The maintainer is in charge of fixing any problems that come up with CRAN checks over time to keep the package on CRAN. The maintainer is also the person who will be emailed about bugs, etc., by other users.

The package can have other authors, as well as people in other roles (e.g., contributor). See the helpfile for the `person` function for more on the codes used for different roles.

13.1.6 Find out more

To find out more about writing R packages, useful sources are:

- Writing R Extensions: Guidelines for R packages from the R Core Team.
- R Packages by Hadley Wickham
- R package development cheatsheet

13.2 Basic example package: `weathermetrics`

`weathermetrics`: Functions to Convert Between Weather Metrics

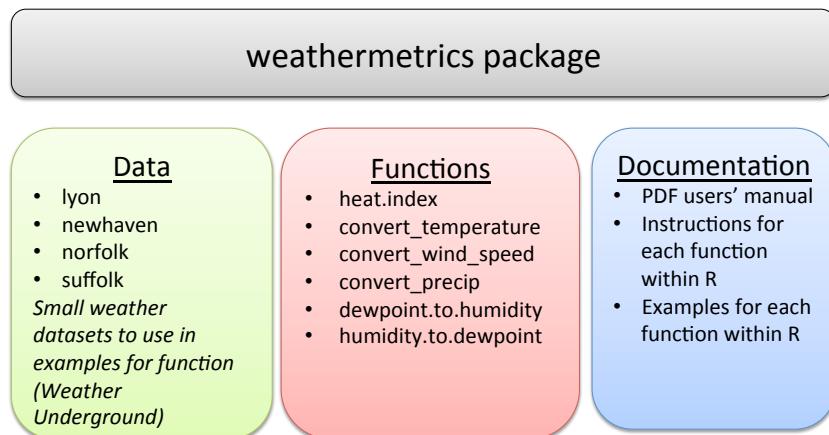
Functions to convert between weather metrics, including conversions for metric and imperial units of temperature, wind speed, and precipitation. This package also includes functions to calculate the heat index from air temperature and a measure of air moisture.

| | |
|-------------------|---|
| Version: | 1.2.2 |
| Depends: | R (\geq 2.10) |
| Suggests: | knitr , rmarkdown |
| Published: | 2016-05-19 |
| Author: | Brooke Anderson [aut, cre], Roger Peng [aut], Joshua Ferrell [aut] |
| Maintainer: | Brooke Anderson <brooke.anderson at colostate.edu> |
| BugReports: | https://github.com/geanders/weathermetrics/issues |
| License: | GPL-2 |
| URL: | https://github.com/geanders/weathermetrics/ |
| NeedsCompilation: | no |
| Citation: | weathermetrics citation info |
| Materials: | NEWS |
| CRAN checks: | weathermetrics results |

The key functions in this package are:

- `convert_temperature`: Convert between temperature metrics
- `convert_precip`: Convert between precipitation metrics
- `convert_wind_speed`: Convert between wind speed metrics
- `heat.index`: Calculates heat index from air temperature and a measure of air moisture (dew point temperature or relative humidity)

13.2.1 Contents of weathermetrics package



Equation to convert from Celsius to Fahrenheit:

$$T_F = \frac{9}{5}T_C + 32$$

Here is the associated R function:

```
celsius.to.fahrenheit
```

```
## function (T.celsius, round = 2)
## {
##   T.fahrenheit <- (9/5) * T.celsius + 32
##   T.fahrenheit <- round(T.fahrenheit, digits = round)
##   return(T.fahrenheit)
## }
## <bytecode: 0x000001338c6d2b60>
## <environment: namespace:weathermetrics>
```

Here is part of the help file for this function:

```
convert_temperature {weathermetrics}
```

Convert from one temperature metric to

Description

This function allows you to convert a vector of temperature values between degrees Kelvin.

Usage

```
convert_temperature(temperature, old_metric, new_metric, round)
```

Arguments

temperature A numeric vector of temperatures to be converted.

old_metric The metric from which you want to convert. Possible options:

- `fahrenheit`, `f`
- `kelvin`, `k`
- `celsius`, `c`

new_metric The metric to which you want to convert. The same options as `old_metric`.

round An integer indicating the number of decimal places to round to.

Here is the start of the function to calculate the heat index:

```
head(heat.index.algorithm, 10)

## 
## 1  function (t = NA, rh = NA)
## 2  {
## 3      if (is.na(rh) | is.na(t)) {
## 4          hi <- NA
## 5      }
## 6      else if (t <= 40) {
## 7          hi <- t
## 8      }
## 9      else {
## 10         alpha <- 61 + ((t - 68) * 1.2) + (rh * 0.094)
```

Here is an example of using this function:

```
data(suffolk)
suffolk %>%
  mutate(heat_index = heat.index(t = TemperatureF,
                                 rh = Relative.Humidity)) %>%
  slice(1:5)

##       Date TemperatureF Relative.Humidity heat_index
## 1 1998-07-12        72            69        72
## 2 1998-07-13        73            66        73
## 3 1998-07-14        74            74        75
## 4 1998-07-15        78            86        80
## 5 1998-07-16        78           100       81
```

13.3 Elements of an R package

13.3.1 Basic elements

Things you edit directly:

- DESCRIPTION file: The package’s “Title page”. Metadata on the package, including names and contacts of authors, package name, and description. This file also lists all the package *dependencies* (other packages with functions this package uses).

- **R folder:** R code defining functions in the package. All code is included in one or more R scripts. If you use Roxygen for help with documentation, all of that is also included in these files.

Things that are automatically written:

- **man folder:** Help documentation for each function. These files are automatically rendered if you use Roxygen.
- **NAMESPACE file:** Helps R find functions in your package you want others to use.

13.3.2 DESCRIPTION file

Required elements:

- Package: Name of the package
- Version: Number of the current version of the package (e.g., 0.1.0)
- Title: Short title for the package, in title case and in 65 characters or less.
- Author and Maintainer (these two sections can be replaced with Authors@R section that uses the `person` function)
- Description: Paragraph describing the package
- License: Name of the license the package is under. If necessary, you can also refer to a LICENSE file included as another file in the package. Only some licenses are easily accepted by CRAN.

Other elements that are common but not required:

- Date: Release date of this version of the package.
- Imports: A list of the packages on which this package depends: other packages with functions used by the code in this package.
- URL: If there is a webpage associated with the package, the address for it. Often, this is the web address of the package's GitHub repository.
- BugReports: Where users can submit problems they've had. Often, the web address of the "Issues" page of the GitHub repository for the package.

```
Package: weathermetrics
Type: Package
Title: Functions to Convert Between Weather Metrics
Version: 1.2.2
Date: 2016-05-19
Authors@R: c(person("Brooke", "Anderson",
  email = "brooke.anderson@colostate.edu",
  role = c("aut", "cre")),
```

```
person("Roger", "Peng",
       email = "rdpeng@gmail.com", role = c("aut")),
person("Joshua", "Ferreri",
       email = "joshua.m.ferreri@gmail.com", role = c("aut")))
Description: Functions to convert between weather metrics,
             including conversions for metrics of temperature, air
             moisture, wind speed, and precipitation. This package also
             includes functions to calculate the heat index from
             air temperature and air moisture.
URL: https://github.com/geanders/weathermetrics/
BugReports: https://github.com/geanders/weathermetrics/issues
License: GPL-2
LazyData: true
RoxygenNote: 5.0.1
Depends:
  R (>= 2.10)
Suggests: knitr,
           rmarkdown
VignetteBuilder: knitr
```

13.3.3 R folder

The R folder of the package includes:

- R scripts with code defining all functions for the package
- Help documentation for each function (if using Roxygen)
- Help documentation for the package data in “data.R”

| Folders | Documents | Developer |
|---|--|--|
| <p>weathermetrics</p> <p>PDF Documents</p> <p>weathermetrics.pdf</p> <p>Other</p> <p>weathermetrics_1.2.0.tar.gz</p> <p>weathermetrics_1.2.2.tar.gz</p> | <p>cran-comments.md</p> <p>NEWS.md</p> <p>README.md</p> <p>Folders</p> <p>data</p> <p>inst</p> <p>man</p> <p>R</p> <p>vignettes</p> <p>Other</p> <p>DESCRIPTION</p> <p>NAMESPACE</p> <p>README.Rmd</p> <p>weathermetrics.Rproj</p> | <p>data.R</p> <p>heat_index</p> <p>moisture_c</p> <p>rainmeasur</p> <p>temperatur</p> <p>weatherme</p> <p>wind_conve</p> |

You define functions in the R scripts just as you would anytime you want to define a function in R. For example, “temperature_conversions.R” includes the following code to define converting from Celsius to Fahrenheit:

```
celsius.to.fahrenheit <- function (T.celsius, round = 2) {
  T.fahrenheit <- (9/5) * T.celsius + 32
  T.fahrenheit <- round(T.fahrenheit, digits = round)
  return(T.fahrenheit)
}
```

Only exception: use `package::function` syntax to call functions from other packages (e.g., `dplyr::mutate()`).

Using `roxygen2`, you put all information for the help files directly into a special type of code comments right before defining the function.

- Start each line with '#!'.
- To render into help files, use the `document` function from the `devtools` package.
- This will write out help files in the `man` folder of the package.
- Use these comments to specify which functions should be *exported* from the package using the `@export` tag. This information will be used to render the `NAMESPACE` file for the package.

```

#! Convert from Celsius to Fahrenheit.
#'
#' \code{celsius.to.fahrenheit} creates a numeric vector of
#'   temperatures in Fahrenheit from a numeric vector of
#'   temperatures in Celsius.
#'
#' @param T.celsius Numeric vector of temperatures in Celsius.
#' @inheritParams convert_temperature
#'
#' @return A numeric vector of temperature values in Fahrenheit.
#'
#' @note Equations are from the source code for the US National
#'       Weather Service's
#'       \ href{http://www.wpc.ncep.noaa.gov/html/heatindex.shtml}
#'       {online heat index calculator}.
#' @author
#' Brooke Anderson \email{brooke.anderson@colostate.edu},
#' Roger Peng \email{rdpeng@gmail.com}
#'
#' @seealso \code{\link{fahrenheit.to.celsius}}
#'
#' @examples # Convert from Celsius to Fahrenheit.
#' data(lyon)
#' lyon$TemperatureF <- celsius.to.fahrenheit(lyon$TemperatureC)
#' lyon
#'
#' @export

```

Some of the most common tags you'll use for `roxygen2` are:

- `@param`: Use to explain parameters for the function.
- `@inheritParam`: If you have already explained a parameter for the help file for a different function, you can use this tag to use the same definition for this function.
- `@return`: Explanation of the object returned by the function.
- `@examples`: One or more examples of using the function.
- `@export`: Export the function, so it's available when users load the package.

By default, the first line in the `roxygen2` comments is the function title and the next section is the function description. For more on `roxygen2`, see: <https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html>

Once you run `document`, this is all rendered as a help file. Now, when you run `?celsius.to.fahrenheit`, you'll get:

```
celsius.to.fahrenheit {weathermetrics}
```

Convert from Celsius to Fahrenheit.

Description

`celsius.to.fahrenheit` creates a numeric vector of temperatures in Fahrenheit from a numeric vector of temperatures in Celsius.

Usage

```
celsius.to.fahrenheit(T.celsius, round = 2)
```

Arguments

`T.celsius` Numeric vector of temperatures in Celsius.

`round` An integer indicating the number of decimal places to round the result to.

Value

A numeric vector of temperature values in Fahrenheit.

Note

Equations are from the source code for the US National Weather Service.

Author(s)

Brooke Anderson brooke.anderson@colostate.edu, Roger Peng [rdpeng](#)

The start of the NAMESPACE file will be automatically written when you run `document` and will look like:

```
# Generated by roxygen2: do not edit by hand

export(celsius.to.fahrenheit)
export(celsius.to.kelvin)
export(convert_precip)
export(convert_temperature)
export(convert_wind_speed)
export(dewpoint.to.humidity)
```

If you are automating helpfile documentation, you must also include an R script with the documentation for each data set that comes with the package.

This file will include `roxygen2` documentation for each data set, followed by the name of the dataset in quotation marks.

As an example, the next slide has the documentation in the “data.R” file for the “lyon” data set.

```
 #' Weather in Lyon, France
#
#' Daily values of mean temperature (Celsius) and mean dew
#' point temperature (Celsius) for the week of June 18, 2000,
#' in Lyon, France.
#
#' @source \href{http://www.wunderground.com/}
#'          {Weather Underground}
#
#' @format A data frame with columns:
#'   \describe{
#'     \item{Date}{Date of weather observation}
#'     \item{TemperatureC}{Daily mean temperature in Celsius}
#'     \item{DewpointC}{Daily mean dewpoint temperature in
#'                   Celsius}
#'   }
#"lyon"
```

13.3.4 Other common elements

Some other elements, while not required, are common in many R packages:

- **data** folder: R objects with data that goes with the package. Often, these are small-ish data files for examples of how to use package functions.

However, more “scientific” packages may include more substantive data in this folder. Some packages are created solely to deliver data.

- **vignettes** folder: One or more tutorials on why the package was created and how to use it. These can be written in RMarkdown.
- NEWS file: Information about changes in later versions of the package.
- .Rbuildignore file: Lists files and directories that should not be included in the package build
- LICENSE file: With certain licenses (MIT is a common example), you need a separate LICENSE file, to supplement the license information in the DESCRIPTION file.

13.3.5 Less common elements

- **src** folder: Sources and headers for compiled code (e.g., C++).
- **demo** folder: R scripts that give demonstrations of using the package.
- **tests** folder: Test code for the package. Currently, the best way to create tests for a package are with the **testthat** package.
- **inst** folder: Various and sundries, including a CITATION file to tell others how to cite your package and executable scripts not in R (e.g., shell scripts, Perl or Python code).

13.4 Creating an R package

Invaluable tools when creating an R package:

- The **devtools** package: Various utility functions that help you develop an R package.
- *R Packages* by Hadley Wickham. Available from O'Reilly or free online at <http://r-pkgs.had.co.nz>
- GitHub: When in doubt of how to structure something, look for examples in code for other R packages. GitHub is currently the easiest way to browse through the code for many R packages.

13.4.1 Initializing an R package

The easiest way to start a new R package project is through R Studio. Go to “File” -> “New Project” -> “Empty Directory”. One of the options is “R Package”.

New Project

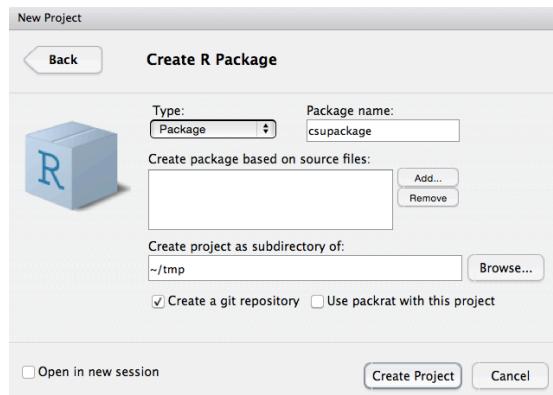
Back Project Type

 **Empty Project**
Create a new project in an empty directory

 **R Package**
Create a new R package

 **Shiny Web Application**
Create a new Shiny web application

You'll need to specify where you want to save the directory and the package name. You can also select if you'd like to use git (you'll still need to set-up and sync with GitHub if you want to post the package to GitHub).



Once you choose this, R Studio will create a new “skeleton” directory for you, with some of the default files and directories you need (kind of like how it starts with a template for RMarkdown documents). You can add and edit files within this structure to create your package.

The screenshot shows the RStudio interface. The top bar includes standard OS X window controls (red, yellow, green) and application icons for file operations, a search bar labeled "Go to file/function", and a dropdown menu for "Addins". The main area is divided into two panes. The left pane is the "Console" showing the R startup message, which includes the R license, natural language support, collaborative nature, documentation resources, and how to quit. The right pane is a code editor titled "hello.R" containing the following R code:

```
1 # Hello, world!-
2 #
3 # This is an example function named 'hello'-
4 # which prints 'Hello, world!'-
5 #
6 # You can learn more about package authoring with RStudio at:-
7 #
8 # ...http://r-pkgs.had.co.nz/-
9 #
```

The code editor has a toolbar with icons for back, forward, save, and search, and a "Source on Save" checkbox. The status bar at the bottom indicates "1:1 (Top Level)".

13.4.2 Working on an R package

Once you set-up the package, most of your work will be in writing the code for the package’s functions and creating documentation. The `devtools` package has some functions that are very useful for this process:

- `load_all`: Loads the last saved version of all functions in the package. You can use this to change and check functions without rebuilding the whole package and restarting R each time.
- `document`: Parse all `roxygen2` comments to create the helpfiles in the `man` directory and the `NAMESPACE` file. As soon as you’ve loaded an documented the last saved version of your package, you can access the help file for each function using `?`, as with other R functions.
- `Control-. :` This is a keyboard shortcut rather than a function, but it allows you to search the package for the code where a certain function is defined. As a package grows larger, this functionality is very useful for navigating the R code in the package.

The `devtools` package also has some functions that set up useful infrastructure for the package. For example, if you want to include a vignette written in RMarkdown, you need to do a few things:

1. Add a new folder called `vignettes`.
2. Add `inst/doc` to the `.gitignore` file. (The built pdf is written into this folder, but typically you don’t want to include rendered files in git, just the code with which they were generated.)
3. Make a few changes to the `DESCRIPTION` file.

Rather than having to remember how to do all this yourself, you can use the `use_vignette` function, which adds this infrastructure to the package at once.

Typically, you will only use these infrastructure calls once per package. Other useful infrastructure functions are:

- `use_cran_comments`: Add a text file with comments for the people who check the package when it’s submitted to CRAN.
- `use_readme_rmd`: Create an RMarkdown “README” file that you can use to provide information on the package (similar to the vignette, but this will show up on the first page of the GitHub repo if you have one for the package).
- `use_news_md`: Create a text file to provide details of changes in later package versions.
- `use_travis`: Add the infrastructure needed to check the package on Travis when you push to GitHub.

- `use_rcpp`: Add an `src` directory and other infrastructure needed to use C++ code within the package.
- `use_testthat`: Add infrastructure for using package tests based on `testthat`.

There are a few infrastructure-type functions you might use more often:

- `use_data`: Save data currently in an R object in your working session to use as data within the package. This function saves that data as an `.rda` file in the `data` folder.
- `use_build_ignore`: Add a file or files to the “`Rbuildignore`” file, so they won’t cause an error with CRAN checks (one of the checks is that there aren’t any unrecognized files or directories in the top level of the package).
- `use_package`: Add a package that your package depends on to the `DESCRIPTION` file.

13.4.3 Finalizing an R package

Once you have included all the functions and documentation for a package, there are a few more steps before that version is ready to be shared:

- Create a vignette and / or `README` file to explain how others can use the package.
- Run the package through CRAN checks and resolve all ERRORS, WARNINGS, and NOTES. This is required if you are submitting to CRAN. It’s usually a good idea and improves the package even if you’re not.
- Change the version number to a stable version (typically, development versions end in `.9000`, like `0.0.0.9000`). When you have a stable version of the package, you’ll change this to a three-part number (e.g., `0.1.0`).
- Build the package locally. For this, you can use the “Build” tab in the upper right RStudio tab (it will show up once you have a package project open).
- Build the package on other systems. You can use Travis (Unix / Linux) and `build_win` (Windows) to do this for those systems.
- Create a pdf of all help files to proofread. To do this, open a bash shell in the parent directory of the package and run `R CMD Rd2pdf <packagename>` (for example, if the package were `cuspackage`, you’d run `R CMD Rd2pdf cuspackage`). This will create a pdf in that directory with all helpfiles for all functions.
- If you want the package to be on CRAN, submit to CRAN. There is a function called `submit_cran` that will build the package and submit it to CRAN.

Appendix A

Appendix A: Vocabulary

You will be responsible for knowing the following functions and vocabulary for the weekly quizzes.

A.1 Quiz 1—R Preliminaries

- Grading policies for the course
- Course requirements / policies for in-class quizzes
- Free and open source software
- “Free as in beer” versus “free as in speech”
- Advantages and disadvantages of interpreted languages compared to “Point-and-click” software
- Advantages and of interpreted languages compared to compiled languages and assembly languages
- Difference between R and RStudio
- R session
- R console
- Function arguments (including required versus optional arguments)
- Accessing a function’s helpfile using ?
- Mathematical operators: +, -, *, -
- R objects and object names
- “gets arrow”: <-
- Rules and style guidelines for naming objects
- ls()
- R scripts
- # comment character
- R packages
- CRAN
- Installing packages

- `install.packages()`
- Loading a package
- `library()`
- Types of package documentation: vignettes and helpfiles
- `vignette()`, option `package =`
- Vectors
- `c()`
- Two of the basic classes of vectors: character and numeric
- `class()`
- Square bracket indexing for vectors: `[...]`
- Dataframes
- `tibble()` function from the `dplyr` package
- `select` and `slice` functions to extract values from dataframes
- `read_csv`, option `skip =`
- `str()`
- `summary()`
- `dim()`
- `ncol()`
- `nrow()`
- Nate Silver
- FiveThirtyEight
- NA for missing values
- `$` to get a column from a dataframe
- `paste()`, option `sep =`
- `paste0()`
- `=` vs. `<-` for assignment expressions
- `package::function()` notation

A.2 Quiz 2—Entering / cleaning data #1

- What kinds of data can be read into R?
- delimited files (csv, tsv)
- fixed width files
- delimiter
- `read_delim`, options `delim =`, `skip =`, `n_max =`, `col_names =`
- `read_fwf`
- `read_csv`, options `skip =`, `n_max =`, `col_names =`
- `readxl` package and its `read_excel()` function
- `haven` package and its `read_sas()` function
- NA
- Computer directory structure
- working directory
- `getwd()`
- `list.files()`

- relative pathnames
- absolute pathnames
- shorthand for pathnames: `..`, `..`, `../data`, etc.
- Reading in data from either a local or online flat file
- `paste()`, option `sep =`
- `paste0()`
- How to read flat files of data that are online directly into R
- `dplyr` package
- `rename()`
- Why you might want to rename column names (e.g., uppercase, long, unusual characters)
- `select()`
- `slice()`
- `mutate()`
- `filter()`
- `arrange()`, including with `desc()`
- `%>%`, advantages of piping

A.3 Quiz 3

- Main types of vector classes in R: character, numeric, factor, date, logical
- `lubridate` functions, including `ymd`, `ymd_hm`, `mdy`, `wday`, and `mday`
- Common logical operators in R (`==`, `!=`, `%in%`, `is.na()`, `&`, `|`)
- `data()` (with and without the name of a dataset as an option)
- `library()` (with and without an argument in the parentheses)
- logical vectors, including running `sum` on a logical vector
- What the bang operator (`!`) does to a logical operator
- The tidyverse
- `min()`
- `max()`
- `mean()`
- `median()`
- `summarize()`
- Special functions to use with `summarize()`: `n()`, `n_distinct()`, `first()`, `last()`
- Using `group_by()` before using `summarize()`
- The three basic elements of a `ggplot` plot: data, aesthetics, and geoms
- `aes` function and common aesthetics, including `color`, `shape`, `x`, `y`, `alpha`, `size`, and `fill`
- Mapping an aesthetic to a column in the data versus setting it to a constant value
- Some common geoms: `geom_histogram`, `geom_points`, `geom_lines`, `geom_boxplot()`
- The difference between “statistical” geoms (e.g., `geom_boxplot`,

- `geom_smooth`) and “non-statistical” (e.g., `geom_point`, `geom_line`)
- Common additions to `ggplot` objects: `ggtitle`, `labs`, `xlim`, `ylim`, `expand_limits`

A.4 Quiz 4

- Guidelines for good graphics
- Data density / data-to-ink ratio
- Small multiples
- Edward Tufte
- Hadley Wickham
- Where to put the `+` in `ggplot` statements to avoid problems (ends of lines instead of starts of new lines)
- Can you save a `ggplot` object as an R object that you can reference later? If so, how would you add elements on to that object? How would you print it when you were ready to print the graph to your RStudio graphics window?
- `geom_hline()`, `geom_vline()`
- `geom_text()`
- `facet_grid()`, `facet_wrap()`
- `grid.arrange()` from the `gridExtra` package
- `ggthemes` package, including `theme_few()` and `theme_tufte()`
- Setting point color for `geom_point()` both as a constant (all points red) and as a way to show the level of a factor for each observation
- `size`, `alpha`, `color`
- Re-naming and re-ordering factors
- Note:** If you read this and find and bring in an example of a “small multiples” graph (from a newspaper, a website, an academic paper), you can get one extra point on this quiz

A.5 Quiz 5

- Reproducible research, including what it is and advantages to aiming to make your research reproducible
- R style guidelines on variable names, `<-` vs. `=`, line length, spacing, semi-colons, commenting, indentation, and code grouping
- Markup languages (concept and examples)
- Basic conventions for Markdown (bold, italics, links, headers, lists)
- Literate programming
- What working directory R uses for code in an .Rmd document
- Basic syntax for RMarkdown chunks, including how to name them
- Options for RMarkdown chunks: `echo`, `eval`, `messages`, `warnings`, `include`, `fig.width`, `fig.height`, `results`

- Difference between global options and chunk options, and which takes precedence
- What inline code is and how to write it in RMarkdown
- How to set global options
- Why style is important in coding
- RPubs

A.6 Quiz 6

- Three characteristics of tidy data
- Five common problems with tidy data and how to resolve them (make sure you understand the examples shown, which you can find out more about in the Hadley Wickham paper I reference in the slides)
- `group_by` with `mutate`, `slice`, and `arrange`
- `separate` and `unite`
- `pivot_longer` and `pivot_wider`
- The `*_join` family of functions (`left_join`, `right_join`, `inner_join`, `full_join`, `anti_join`, `semi_join`)

A.7 Quiz 7

- regular expressions
- difference in the output between `str_extract` and `str_detect`
- `str_to_lower`, `str_to_upper`, `str_to_title`, `str_trim`
- lists
- indexing from lists ([[and \$)
- exploring lists (`class`, `names`, `str` functions)
- exploring lists with `jsoned` from the `listviewer` package
- `tidy` function from `broom` applied to the output from a statistical test function
- `forcats` package, including `fct_recode`, `fct_infreq`, `fct_reorder`, and `fct_lump`
- Bioconductor
- “accessor” functions for Bioconductor (e.g., `get_variable`)
- `select` with `starts_with`, `select_if`

A.8 Quiz 8

- Review of previous weeks (especially tidyverse packages: `dplyr`, `tidyr`, `ggplot2`, `stringr`, and `forcats`)
- Writing a formula with ~ syntax

- Regression modeling with `lm`, `glm`
- Using functions from `broom` to tidy model output (`augment`, `tidy`, `glance`)

A.9 Quiz 9

- Nesting with `group_by` and `nest` and unnesting with `unnest`
- A list-column in a dataframe
- Review of previous quizzes

Appendix B

Appendix B: Homework

This section provides the homework assignments for the course.

B.1 Homework #1

Due date: Sept. 12 by midnight

For this assignment, you will submit the assignment to me **by email** by the due date. You should include one file in your submission:

1. A Word document with seven paragraphs. Each paragraph should be headed with the name of one swirl lesson and the body of the paragraph should describe that lesson and what you learned from it.

For this homework assignment, you'll be working through a few swirl lessons that are relevant to the material we've covered so far. Swirl is a platform that helps you learn R **in R**—you can complete the lessons right in your R console.

Depending on your familiarity with R, you can either work through seven lessons of your choice in the **R Programming: The basics of programming in R** and **Getting and Cleaning Data** courses (suggested lessons are listed further below) (**Option #1**), or you can work through seven lessons of your choice taken from any number of swirl's available courses (**Option #2**).

For each lesson completed, please write a few sentences that cover: 1. A summary of the topic(s) covered in that lesson, and 2. The most interesting thing that you learned from that lesson. Turn in a hardcopy of this (with your first and last name at the top) during class on the due date.

To begin, you'll first need to install the swirl package:

```
install.packages("swirl")
```

If you've never run `swirl()` before, you will be prompted to install a course. You can do that with the `install_course` function. For example, to install the `R Programming` course, you would run:

```
library(swirl)
install_course("R Programming")
```

Once you've installed a course, every time you enter the swirl environment with `swirl()`, `R Progamming` should show up as a course option to select. You can enter `R Programming` to start lessons in that course by typing the number in front of it when you run `swirl()`.

Once you have at least one course installed, you call the `swirl()` function to enter the interactive platform in RStudio. The console will take you through a few prompts: you'll give swirl a name to call you, and take a look at some commands that are useful in the swirl environment. Those commands are listed further below.

```
library(swirl)
swirl()
```



After calling `swirl()`, you may be prompted to clear your workspace variables by running `rm=list=ls()`. Running this code will clear any variables you already have saved in your global environment. While swirl recommends that you do this, it's not necessary.

Some of these lessons complement online courses through Coursera, so sometimes you will be asked after you complete a lesson if you want to report your results to Coursera. You should select “No” for that option each time.

B.1.1 Option 1

For **Option 1** of this homework, you will need to work through seven of the 15 available lessons in the `R Programming` course. Here are some suggestions for particularly uesful lessons that you could choose (the lesson number within the course is in parentheses):

R Programming course:

- Basic Building Blocks (1)
- Sequences of Numbers (3)
- Vectors (4)
- Missing Values (5)
- Subsetting Vectors (6)
- Logic (8)
- Looking at Data (12)
- Dates and Times (14)

Getting and Cleaning Data course:

- Manipulating Data with dplyr (1)
- Grouping and Chaining with dplyr (2)
- Dates and Times with lubridate (4)

Each lesson should take at most 10–15 minutes, but some are much shorter. You can complete the lessons in any order you want, but you may find it easiest to start with the lowest-numbered lessons and work your way up, in the order we've listed the lessons here.

You'll be able to get started on some of these lessons after your first day in class ("Basic Building Blocks", for example), but others cover topics that we'll get to in weeks 2 and 3. Whether or not we've covered a swirl topic in class, you should be able to successfully work through the lesson. At the end of each lesson, you may be asked if you would like to receive credit for completing this course on Coursera.org. Always choose "no" for this option.

Again, you'll need to compose and turn in a few sentences for each lesson. Make sure to include a summary of what each lesson was about, and the most interesting thing about that lesson.

B.1.2 Option 2

If you're already somewhat familiar with R, you might want to choose your seven lessons from other swirl courses instead of or in addition to those available in the **R Programming** and **Getting and Cleaning Data** courses.

Check out the list of available Swirl Courses to see which ones you would like to install and check out available lessons for. For example, to choose a lesson in the **Exploratory Data Analysis** course, you would run:

```
library(swirl)
install_course("Exploratory Data Analysis")
swirl()
```

After entering the `Exploratory Data Analysis` course, you could choose from any one of its available lessons.

In your written summary for each lesson (again, a few sentences that cover a summary of the lesson and the most interesting thing you learned), make sure to specify which course each lesson you completed was from.

B.1.3 Special swirl commands

In the swirl environment, knowing about the following commands will be helpful:

- Within each lesson, the prompt `...` indicates that you should hit Enter to move on to the next section.
- `skip()`: skip the current question.
- `play()`: temporarily exit swirl. It can be useful during a swirl lesson to play around in the R console to try things out.
- `nxt()`: regain swirl's attention after `play()`ing around in the console.
- `main()`: return to swirl's main menu.
- `bye()`: exit swirl. Swirl will save your progress if you exit in the middle of a lesson. You can also hit the Esc. key to exit. (To re-enter swirl, run `swirl()`. In a new R session you will have to first load the swirl library: `library(swirl)`.)

B.1.3.1 For fun

While they aren't required for class, you should consider trying out some other swirl lessons later in the course. You can look through the course directory to see what other courses and lessons are available. For the first part of our course, you might find the "Exploratory Data Analysis" course helpful. If you would like to learn more about using R for statistical analysis, you might find the "Regression Models" course helpful.

B.2 Homework #2

Due date: Sept. 28 by midnight

For this assignment, you will submit the assignment to me **by email** by the due date. You should include three files in your submission:

1. The final rendered Word document (rendered from an RMarkdown file).
2. The original RMarkdown file used to create that final document.
3. The dataset you used for the assignment.

For this assignment, start by picking a dataset either from your own research or something interesting available online (if you're struggling to find something, check out Five Thirty Eight's GitHub data repository). You will then use this dataset to practice what you've learned with R so far. This will also be a chance, if you're using a dataset from your own research for me to get an idea of how you might be planning to use R in future research.

Very important note: Some research datasets have privacy constraints. This includes any datasets collected from human subjects, but can also include other datasets. For some projects, the principal investigator may prefer to keep the data private until publication of results. Before using a dataset for this project, please confirm with your research advisor that there are no constraints on the data. I do not plan to make the results of this assignment public, but I also do not want to us to be emailing back and forth a dataset with any constraints.

Using your dataset, create an RMarkdown document with the content listed below. **Be sure to set the echo option to TRUE so all of your code will also print out to the Word document.** Include in the RMarkdown document: (1) your full name; (2) the due date of the assignment; and (3) a title that includes "Homework 2". These should all be set in RMarkdown, **not** changed in the Word output. Each section I've listed below should be included in the RMarkdown document as its own section, with a section header created using Markdown code.

- **Section 1: Description of the data:** Describe the dataset you are using, both in terms of the **content** (what is this data measuring? how was it collected? what kinds of research questions are you hoping to use it to answer?) and in terms of its **format** (what type of file is it saved in? what if it is in a flat file, is it fixed width or delimited? if it is delimited, what is the delimiter? if it is binary, what is the program that would normally be used to open it?). (20 points)
- **Section 2: Reading the data into R:** Include code that reads the data into R and assigns it to a dataframe object that you can use later in the document. Explain in the text which R function you used to read in the data (e.g., `read_csv`) and which package it came from (if it was not a base R function). If there were any special options you needed to use (e.g., `skip` to skip some rows without data), list those and explain why you used them. Next, include some code to clean the data (e.g., rename columns, convert any dates into a "Date" format). You can filter to certain rows if you would like, but do **not** filter out missing values, as we'll want to learn more about those later. (20 points)
- **Section 3: Characteristics of the data:** Describe the dataframe you just read in. How many rows does it have? How many columns? (Use inline code in the RMarkdown document to put this information into a sentence that reads "This dataframe has ... rows and ... columns.") What are the names of the columns? What does each row measure (i.e., what is the unit of observation)? Include a table (using Markdown directly or

- `kable`) explaining what each column measures. This table should have three columns: (1) the column name in the R dataframe;
- (2) a very brief description of what each column measures; and (3) the units (if any) of the measurement in the column. (20 points)
- **Section 4: Summary statistics:** Pick three columns of the dataframe. Use the `summarize` function to get the following summaries of these columns: (1) minimum value; (2) maximum value; (3) mean value; (4) number of missing values. If there are missing values, make sure you use the appropriate options in summarizing these values to exclude those when calculating the minimum, maximum, and mean. Assign the result of this `summarize` call to a new R object, and print it out, so these summaries show up in your final, rendered Word document. (20 points)
 - **Section 5: Visualizing the data:** Create two plots of your dataframe. One should use a “statistical” geom (e.g., histogram, bar chart, boxplot) and one a “non-statistical” geom (e.g., scatterplot, line plot for time series). Explain why these plots help you learn more about this data and about the interesting research questions you’re hoping to explore with the data. Be sure to customize the final size of each plot in the Word document using the `fig.width` and `fig.height` commands. For each plot, also be sure to customize the x- and y-axis labels. Finally, explain how each plot is following at least two of the principles of “good graphics” covered in week 4 of the course (Chapter 4 of the book)—if necessary, use `ggplot` functions and options to make the plots comply with some of these principles. (20 points)

B.3 Homework #3

Due date: Oct. 19 by midnight

This homework includes several parts. Make sure you complete each part for full credit (100 points).

1. Complete your homework in an RMarkdown document and knit it to either Word or PDF to submit. Change your global options so that no warnings, messages, or errors are printed out, but that so all code and results are printed. (10 points)
2. Select one of the figures you created for the last homework. You will be improving that figure in this part of the homework.
 - Go through the six guidelines for good graphics we discussed in Chapter 6. Re-create the same plot you created in the last homework and, in a paragraph or two, list which of these elements you currently have in the plot and how these elements help make the plot more effective. (10 points)

- Select three of these guidelines for which you either aren't using them in the current graph or could add to what you currently have in the plot. Create a new version of the plot based on your choice. Include a paragraph explaining how these changes make the plot more effective and how you added them. (20 points)
3. In the `titanic` package (available on CRAN), there's a `titanic_train` dataset we'll use for this question.
- Load that dataset and create a dataframe object limited to the columns `Survived` and `Age`. Change the labeling for the `Survived` column so that it says "Survived" instead of "1" and "Died" instead of "0". Print out the oldest five people who survived and who died (so, you'll be printing out 10 rows total). (10 points)
 - Create a histogram of the ages of people in this dataset, faceted by whether or not they survived. Arrange these so that they are vertically aligned rather than side by side (Hint: check the `nrow` option when faceting). Change the label on the y-axis to be something that more precisely describes what's being shown (rather than "count") and add a title to the plot. (10 points)
 - Determine the mean age for each group (survived and died) for everyone who has a non-missing age, as well as the total number of people in the group and the number of people with age information missing for the group. Print out a two-row dataframe with this summary information (it should have columns for (1) which group (survived / died); (2) the mean age of people in the group with non-missing age data; (3) the total number of people in the group; and (4) the number of people in the group with missing age data). (15 points)
 - Run a statistical test of whether there is a difference in the mean ages between the two groups (Hint: you might want to google something like "test difference two means"). You can run this test either using a simple statistical test or by fitting a linear regression—you may take either approach. (15 points)
4. There's a radio show on National Public Radio called "Weekend Edition", and each week they have a "Sunday Puzzle". The puzzle for the week ending August 5, 2018 was this: "I said think of a familiar two-word phrase in eight letters with four letters in each word. The first word starts with M. And I said move the first letter of the second word to the end, and you get a regular eight-letter word, which, amazingly, other than the M, doesn't share any sounds with the original two-word phrase. What phrase is it?". The winner had this to say about how he solved the problem: "I went to the National Puzzlers' League website and pulled up a list of eight-letter words that start with the letter M and just started scanning through it...'. You can solve this puzzle using R (some of the `stringr` functions, like `str_count` and `str_sub`, will be particularly helpful). Write R code to solve this puzzle. (10 points)

- Hint 1: Work backwards. Start with eight-letter words, and apply a backwards transform to the one described to transform them into two four-letter words.
- Hint 2: This link might prove very useful: <https://github.com/dwyl/english-words>.

B.4 Homework #4

Due date: Nov. 2 by midnight

For this homework, you will use a data set collected by the *Washington Post* on homicides in 50 large U.S. cities. The data is available through their GitHub repository: <https://github.com/washingtonpost/data-homicides>. You can read their accompanying article at <https://www.washingtonpost.com/graphics/2018/investigations/where-murders-go-unsolved/>.

Submit your homework by email. You should include the Rmarkdown file (.Rmd) and the output Word document.

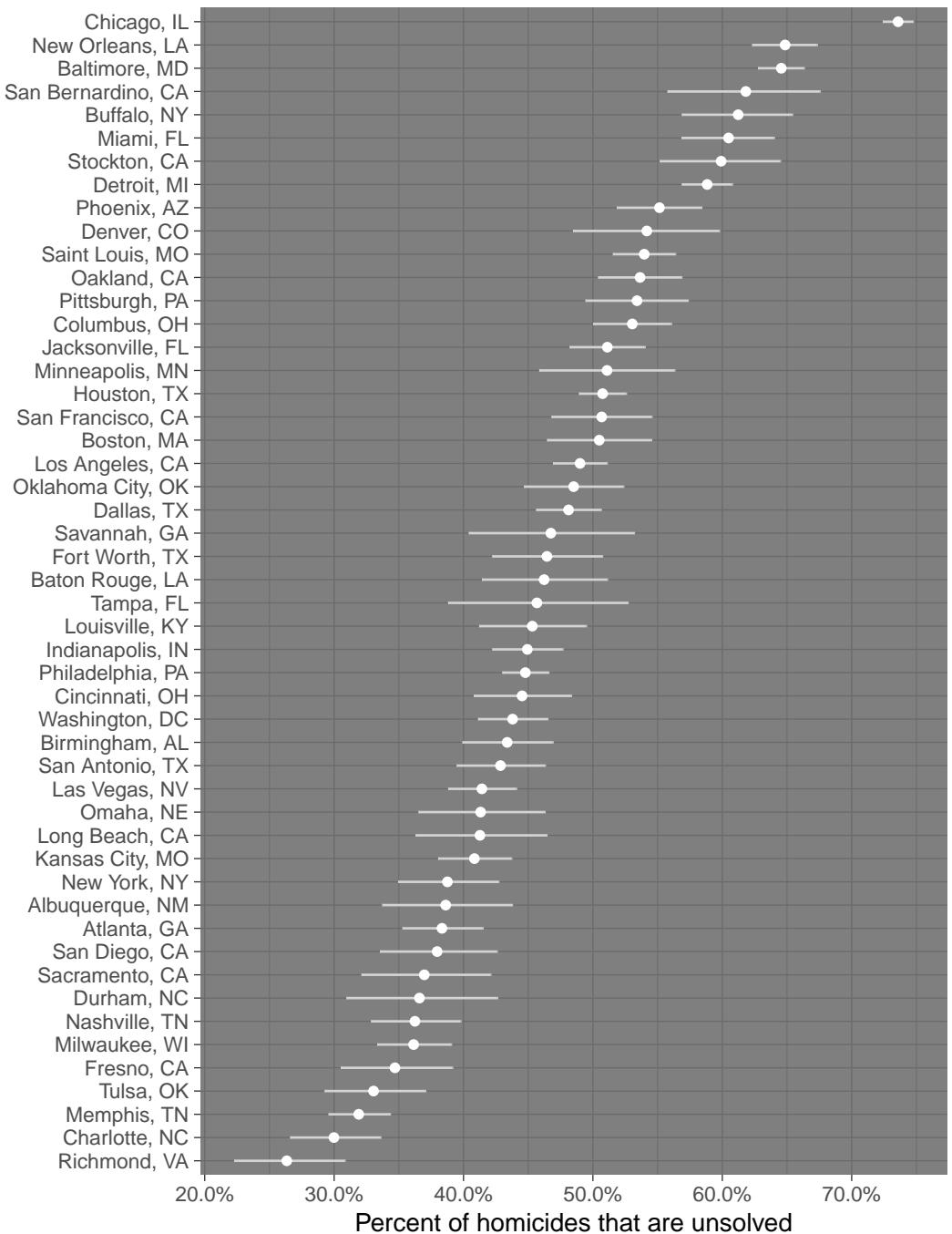
1. Read in the data as an R object named `homicides` and create a new column called `city_name` that combines the city and state like this “Baltimore, MD”. (15 points)
2. Create a dataframe called `unsolved` with one row per city that gives the total number of homicides for the city and the number of unsolved homicides (those for which the disposition is “Closed without arrest” or “Open/No arrest”). (15 points)
3. For the city of Baltimore, MD, use the `prop.test` function to estimate the proportion of homicides that are unsolved, as well as the 95% confidence interval for this proportion (we’re assuming that the data from the years covered by this dataset is a representative sample of Baltimore in a larger set of years). Print the output of the `prop.test` directly in your RMarkdown, and then save the output of `prop.test` as an R object and apply the `tidy` function from the `broom` package to this object and pull the estimated proportion and confidence intervals from the resulting tidy dataframe. (20 points)
4. Now use what you learned from running `prop.test` for one city to run `prop.test` for all the cities. Your goal is to create the dataframe you need to create the figure shown below, where the points show the estimated proportions of unsolved homicides in each city and the horizontal lines show the estimated 95% confidence intervals. Do this all within a “tidy” pipeline, starting from the `unsolved` dataframe that you created for step 3. Use `map2` from `purrr` to apply `prop.test` within each city and then `map` from `purrr` to apply `tidy` to this output. Use the `unnest` function from the `tidyverse` package on the resulting list-column (from mapping `tidy` to the `prop.test` output list-column), with the option `.drop = TRUE`, to

get your estimates back into a regular tidy data frame before plotting. (25 points)

5. Create the plot shown below. Hint: Check out the `geom_errorbarh` geom with the `height = 0` option to get the horizontal lines for the confidence intervals. To get full points, be very careful to make sure that all labeling, color choices, figure dimensions, etc., in your figure match the figure shown here. (25 points)

Percent of unsolved homicides by city

Bars show 95% confidence interval



B.5 Homework #5

Due date: Nov. 16 by midnight

For this homework, you will continue using the dataset used for Homework #4. You will submit this project by sending me the link to a GitHub repository. For the write-up of this homework, you'll need to write an RMarkdown document and render it to a pdf file within the R Project for the GitHub repository.

B.5.1 Setting up a GitHub repository for this project

Take the following steps to set up your GitHub repository for the homework (we will work on this as an in-course exercise):

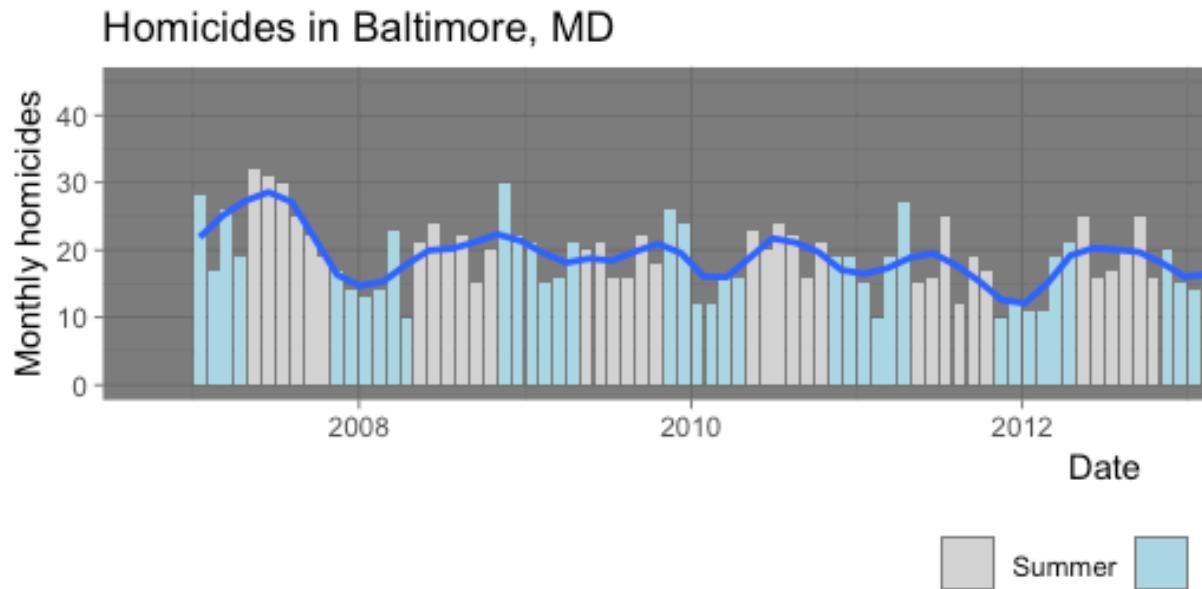
1. Set up a local (i.e., on your computer) R project with subdirectories for the data, writing (this is where you'll put your Rmarkdown file and the output file), and figures.
2. Download the Washington Post data and save it in an appropriate place in this R project directory.
3. Initialize git for the R project and make your initial commit.
4. Create an RMarkdown file for your homework answers in the appropriate subdirectory and save it. Commit this change with an appropriate commit message. (At this stage, you can look at your “History” in the git commit window to see the changes you've made so far.)
5. Login to your GitHub account and create a new, blank repo with the same name as your R project. (If you have not already set up an SSH key to make it easier to push back and forth between your computer and GitHub, do so now.)
6. On your computer, specify your GitHub repository as the remote for your project and do an initial push to send all the files to the GitHub repo. Look at your repository on GitHub to make sure it worked.
7. Add a README.md file to the top level of your R project (on your local computer) using Markdown syntax and then commit the change locally and push to your GitHub repository.

For the README.md file, you can create a text file in RStudio and save it as “README.md”. You can then write this file using Markdown syntax (like RMarkdown, but without any code chunks).

As you work on your homework, make sure you commit regularly (with helpful commit messages). You should have **at least 15 commit messages** in your history for the repo by the time you turn in the homework.

Select **one** of the following two figures to create for the homework:

1. **Choice 1:** Pick one city in the data. Create a map showing the locations of the homicides in that city, using the `sf` framework discussed in class. Use `tigris` to download boundaries for some sub-city geography (e.g., tracts, block groups, county subdivisions) to show as a layer underneath the points showing homicides. Use different facets for solved versus unsolved homicides and different colors to show the three race groups with the highest number of homicides for that city (you may find the `fct_lump` function from `forcats` useful for this).
2. **Choice 2:** Recreate the graph shown below. It shows monthly homicides in Baltimore, with a reference added for the date of the arrest of Freddie Gray and color used to show colder months (November through April) versus warmer months (May through October). There is a smooth line added to help show seasonal and long-term trends in this data.



B.6 Homework #6

Due date: Wednesday, Dec. 7 by midnight

1. Read the article *Good Enough Practices in Scientific Computing* by Wilson et al. (available here). In a half page, describe which of these “pretty good practices” you are incorporating for your group project. Also list one or two practices that you are not following but that would have made sense and how you could change your practices to follow them.

2. Find an article in *The R Journal* that describes an R package that you could use in your own research or otherwise find interesting. Describe why the package was created and what you think it's most interesting features are. In an R Markdown document, run one or two of the R examples included in the article.