

# R Programming for Research

Brooke Anderson and Rachel Severson

2017-08-28







To ...



# Contents

<b>Online course book, ERHS 535</b>	<b>v</b>
<b>Course information</b>	<b>1</b>
0.1 Course overview . . . . .	1
0.2 Time and place . . . . .	1
0.3 Detailed schedule . . . . .	1
0.4 Grading . . . . .	2
0.5 Course set-up . . . . .	5
0.6 Coursebook . . . . .	5
<b>I Part I: Preliminaries</b>	<b>7</b>
<b>1 R Preliminaries</b>	<b>9</b>
1.1 R and R Studio . . . . .	9
1.2 The “package” system . . . . .	14
1.3 Basic code conventions of R . . . . .	19
1.4 R’s most basic object types . . . . .	24
1.5 Using R functions . . . . .	28
1.6 R scripts . . . . .	31
1.7 In-course Exercise . . . . .	34
<b>II Part II: Basics</b>	<b>45</b>
<b>2 Entering and cleaning data #1</b>	<b>47</b>
2.1 Directories and pathnames . . . . .	47
2.2 Diversion: <code>paste</code> . . . . .	53
2.3 Reading data into R . . . . .	55
2.4 Data cleaning . . . . .	61
2.5 Dates in R . . . . .	68
2.6 In-course Exercise . . . . .	72
<b>3 Exploring data #1</b>	<b>87</b>

3.1	Data from a package . . . . .	87
3.2	Plots to explore data . . . . .	88
3.3	Simple statistics functions . . . . .	100
3.4	Logical vectors . . . . .	105
3.5	Regression models . . . . .	107
3.6	In-course exercise . . . . .	116
<b>4</b>	<b>Reporting data results #1</b>	<b>141</b>
4.1	Guidelines for good plots . . . . .	141
4.2	High data density . . . . .	142
4.3	Meaningful labels . . . . .	145
4.4	References . . . . .	146
4.5	Highlighting . . . . .	148
4.6	Order . . . . .	151
4.7	Small multiples . . . . .	152
4.8	Advanced customization . . . . .	155
4.9	To find out more . . . . .	159
4.10	In-course exercise . . . . .	160
<b>5</b>	<b>Reproducible research #1</b>	<b>179</b>
5.1	What is reproducible research? . . . . .	179
5.2	Markdown . . . . .	180
5.3	Literate programming in R . . . . .	182
5.4	Style guidelines . . . . .	185
5.5	More with knitr . . . . .	188
5.6	Saving graphics files . . . . .	191
5.7	In-course exercise . . . . .	192
<b>III</b>	<b>Part III: Intermediate</b>	<b>195</b>
<b>6</b>	<b>Entering and cleaning data #2</b>	<b>197</b>
6.1	Joining datasets . . . . .	197
6.2	Tidy data . . . . .	199
6.3	Gathering . . . . .	205
6.4	In-course exercise . . . . .	208
<b>7</b>	<b>Exploring data #2</b>	<b>223</b>
7.1	Parentheses . . . . .	223
7.2	Loops . . . . .	223
7.3	Other control structures . . . . .	232
7.4	Functions . . . . .	235
7.5	Regular expressions . . . . .	239
7.6	In-course exercise . . . . .	247
<b>A</b>	<b>Appendix A: Vocabulary</b>	<b>253</b>

A.1 Week 1 (Quiz 1) . . . . .	253
A.2 Week 2 (Quiz 2) . . . . .	254
A.3 Week 3 (Quiz 3) . . . . .	255
A.4 Week 4 (Quiz 4) . . . . .	256
A.5 Week 5 (Quiz 5) . . . . .	257
A.6 Week 6 (Quiz 6) . . . . .	257
A.7 Week 7 (Quiz 7) . . . . .	258
A.8 Week 8 (Quiz 8) . . . . .	258
<b>B Appendix B: Homework</b>	<b>259</b>
B.1 Homework #1 . . . . .	259
B.2 Homework #2 . . . . .	262
B.3 Homework #3 . . . . .	263
B.4 Homework #4 . . . . .	264
B.5 Homework #5 . . . . .	265
B.6 Homework #6 . . . . .	265



# Online course book, ERHS 535

```
fortunes::fortune("write a book")  
  
##  
## Have you ever wanted to write a book, but not known where to start? Now is  
## a very good time to jump in, because there is currently a very simple  
## recipe for success: just put R in the title and you will have to beat the  
## publishers off with a stick!  
## -- Paul Murrell  
## ASA Statistical Computing & Graphics Newsletter 17(2) (November 2006)
```

This is the online book for Colorado State University's ERHS 535 *R Programming for Research* course. 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. Because this is my first semester teaching the course with this online book, it will be evolving throughout the semester, as we get to new material.

“ “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 document provides the course notes for Colorado State University's ERHS 535 course for Fall 2017. The course offers in-depth instruction on data collection, data management, programming, and visualization, using data examples relevant to academic research.

## 0.2 Time and place

This course meets in Room 120 of the Environmental Health Building on Mondays and Wednesdays, 10:00 am–12:00 pm. Exceptions to these meeting times are:

- There will be no meeting on Labor Day (Monday, Sept. 4).
- There are no course meetings the week of Thanksgiving (week of Nov. 20).
- I will be away from Fort Collins on up to three course dates (Sept. 20, Nov. 6, and Nov. 8). I will update the course with plans for these dates. Potential plans include guest lectures, recorded lectures, and make-up sessions (for which attendance would not be counted towards the course attendance grade).

## 0.3 Detailed schedule

Here is a more detailed view of the schedule for this course for Fall 2016:

Dates	Level	Lecture content	Graded items
Aug. 21, 23	Preliminary	R Preliminaries	
Aug. 28, 30	Basic	Entering and cleaning data	
Sept. 6	Basic	Exploring data	Quiz (W)
Sept. 11, 13	Basic	Reporting data results	Quiz (M), HW #1 (W)
Sept. 18, 20	Basic	Reproducible Research	Quiz (M)
Sept. 25, 27	Intermediate	Entering and cleaning data	Quiz (M), HW #2 (W)
Oct. 2, 4	Intermediate	Exploring data	Quiz (M)
Oct. 9, 11	Intermediate	Reporting data results	Quiz (M), HW #3 (W)
Oct. 16, 18	Intermediate	Reproducible Research	Quiz (M), Group choices (M)
Oct. 23, 25	Advanced	Entering and cleaning data	Quiz (M), Project proposal (M), HW #4 (W)
Oct. 30, Nov. 1	Advanced	Exploring data	
Nov. 6, 8	Advanced	Reporting data results	HW #5 (W)
Nov. 13, 15	Advanced	Mapping in R	
Nov. 27, 29	Advanced	Package development 1	HW #6 (W)
Dec. 4, 6	Advanced	Package development 2	Project draft (M)
Week of Dec. 11		Group presentations	Final project (M)

## 0.4 Grading

Course grades will be determined by the following five components:

Assessment component	Percent of grade
Final group project	30
Weekly in-class quizzes, weeks 3-10	25
Homework	25
Attendance and class participation	10
Weekly in-course group exercises	10

### 0.4.1 Attendance and class participation

Because so much of the learning for this class is through interactive work in class, it is critical that you come to class. Out of a possible 10 points for class attendance, you will get:

- **10 points** if you attend all classes
- **8 points** if you miss one class
- **6 points** if you miss two classes
- **4 points** if you miss three classes
- **2 points** if you miss four classes
- **0 points** if you miss five or more classes

Exceptions:

- Attendance on the first day of class (Aug. 21) will not be counted.

- If you miss classes for “University-sanctioned” activities. These can include attending a conference, travel to collect data for your dissertation), For these absences, you must inform prior to the date that you will be absence. No points will be lost for attendance if you provide a signed letter from your research advisor by Dec. 11, 2017 (start of finals week), and you can make arrangements with me to make up any missed work. For more details, see CSU’s Academic Policies on Course Attendance.
- If we have any course meetings outside of our regular course time (M/W 10:00 AM–12:00 PM) to make up for dates when I am travelling, attendance will be optional.

#### 0.4.2 Weekly in-course group exercises

Part of each class will be spent doing in-course group exercises. Ten points of your final grade will be based on your participation in these exercises. As long as you are in class and participate in these exercises, you will get full credit for this component. If you miss a class, to get credit towards this component of your grade, you will need to turn in a one-page document describing what you learned from doing the in-course exercise on your own time. All in-class exercises are included in the online course book at the end of the chapter on the associated material.

#### 0.4.3 In-class quizzes

You will have eight total in-class quizzes. You will have one for each of the Week 2–10 class meetings. There will be *at least* 10 questions per quiz. You will get 1/3 point for each correct answer. If you do the math, you can get full credit for this if you get at least 75% of your answers right. You can not get more than the maximum of 25 points for this component—once you reach 25 points on quizzes, you will have achieved full credit for the quiz component of the course grade.

All quiz questions will be multiple choice, matching, or some other form of “close-answered” question (i.e., no open-response-style questions). You can not make up a quiz for a class period you missed. You can still get full credit on your total possible quiz points if you miss a class, but it means you will have to work harder and get more questions right for days you are in class.

Because grading format for these quizzes allows for you to miss some questions and still get the full quiz credit for the course, I will not ever re-consider the score you got on a previous quiz, give points back for a wrong answer on a poorly-worded question, etc. However, if a lot of people got a particular question wrong, I will be sure to cover it in the next class period. Also, especially if a question was poorly worded and caused confusion, I will work a similar question into a

future quiz– in addition to the 10 guaranteed questions for that quiz– so every student will have the chance to get an extra 1/3 point of credit for the question.

The “Vocabulary” appendix of our online book has the list of material for which you will be responsible for this quiz. Most of the functions and concepts will have been covered in class, but some may not. You are responsible for going through the list and, if there are things you don’t know or remember from class, learning them. To do this, you can use help functions in R, Google, StackOverflow, books on R, ask a friend, and any other resource you can find.

In general, using R frequently in your research or other coursework will help you to prepare and do well on these quizzes.

#### 0.4.4 Homework

There will be six homework assignments, starting a few weeks into the course and then due approximately every two weeks (see the detailed schedule in the online course book for exact due dates).

The first homework (HW #1) should be done individually. Other homeworks will be done in small groups of approximately three students, which I will randomly assign for each homework (and that will change for most separate assignments).

Homeworks will be graded for correctness, but some partial credit will be given for questions you try but fail to answer correctly. If you can’t complete a required task, be sure to show and explain what you tried to do to complete it to improve your chances of getting partial credit for the task. For each of the homeworks done in groups, 90% of each person’s grade will be based on the group’s homework submission (and so be the same for all group members), while 10% will be based on group participation, as based on input from your fellow group members. It is critical that each group member understand all elements of the final group homework submission, and quizzes following homework submissions are likely to include questions to gauge this comprehension.

Homework is due by the start of class on the due date. Your grade will be reduced by 10 points for each day it is late, and will receive no credit if it is late by over a week.

#### 0.4.5 Final group project

For the final project, you will work in small groups (3–4 people) on an R programming challenge. The final grade will be based on the resulting R software, as well as on a short group presentation and written report describing your work. You will be given a lot of in-class time during the last third of the semester to work with your group on this project, and you will also need to spend some time

working outside of class to complete the project. More details on this project will be provided later in the semester.

## 0.5 Course set-up

Please be sure you have the latest version of R and RStudio (Desktop version, Open Source edition) installed. Both are free for anyone to download. Also, be sure to sign up for a free GitHub account.

Here are useful links for this set-up:

- R: <https://cran.r-project.org>
- RStudio: <https://www.rstudio.com/products/rstudio/#Desktop>
- Sign-up for a GitHub account: <https://github.com>

## 0.6 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 the second half of the book, 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:

- Links to the slides presented in class 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 phenomenal 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.

### 0.6.1 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.

# **Part I**

## **Part I: Preliminaries**



# Chapter 1

## R Preliminaries

Download a pdf of the lecture slides covering this topic.

### 1.1 R and R Studio



One of the strengths of R is that many different programmers have developed and shared their own extensions of R, called “packages”. We’ll talk a lot more about packages later in this section, but we wanted to introduce one here. The `fortunes` package allows you to read some notable quotes about R. These quotes are often by leaders in the R community, and they were often originally from R help listservs. We’ll use this package throughout the book to add in some of the R “fortunes” as topics they cover come up.

#### 1.1.1 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 journalism / data analysis

```
fortune("good data analyst")

## 
## Can one be a good data analyst without being a half-good programmer? The
## short answer to that is, 'No.' The long answer to that is, 'No.'
##   -- Frank Harrell
##       1999 S-PLUS User Conference, New Orleans (October 1999)

fortune("most widely used piece of software for statistics")

## 
## Let's not kid ourselves: the most widely used piece of software for
## statistics is Excel.
##   -- Brian D. Ripley ('Statistical Methods Need Software: A View of
##       Statistical Computing')
##       Opening lecture RSS 2002, Plymouth (September 2002)
```

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. However, R is still the first choice of statisticians in most fields, so I would argue that R has a an 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

```
fortune("show it's as good as the others")

## 
## Rene M. Raupp: Does anybody know any work comparing R with other (charged)
## statistical software (like Minitab, SPSS, SAS)? [...] I have to show it's
## as good as the others.
## Kjetil Brinchmann Halvorsen: Sorry. That will be difficult. Couldn't it do
## to prove it is better?
##   -- Rene M. Raupp and Kjetil Brinchmann Halvorsen
##       R-help (May 2005)
```

### 1.1.2 Open-source software

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

```
fortune("the price")

## 
## It [the effort of learning how to use R] is the price paid, just as the
## dollars or euros for a commercial package would be. For that price, I've
## learnt a great deal - and not only about R. And I shall remember it when I
## next have to find a heavyweight solution for a big problem presented by a
## small charitable client with an invisible budget. It's a huge,
## awe-inspiring package - easier to perceive as such because the power is
## not hidden beneath a cosmetic veneer.
##     -- Felix Grant (in an article about free statistics software)
##         Scientific Computing World (November 2004)
```

R is open-source software. Many other popular statistical programming languages, conversely, are proprietary. 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

Open-source software software is the *libre* type of free (Figure 1.1). 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

Popular open source languages for R and R packages include GPL and MIT.

“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



**Figure 1.1:** 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 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.

```
fortune("how do you know that the results are trustable?")
```

```
##  
## Mingzhai Sun: When you use it [R], since it is written by so many authors,  
## how do you know that the results are trustable?  
## Bill Venables: 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.  
##     -- Mingzhai Sun and Bill Venables  
##          R-help (January 2004)
```

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

### 1.1.3 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.

The company that develops this IDE is a fantastic contributor to the global R community. RStudio currently:

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)

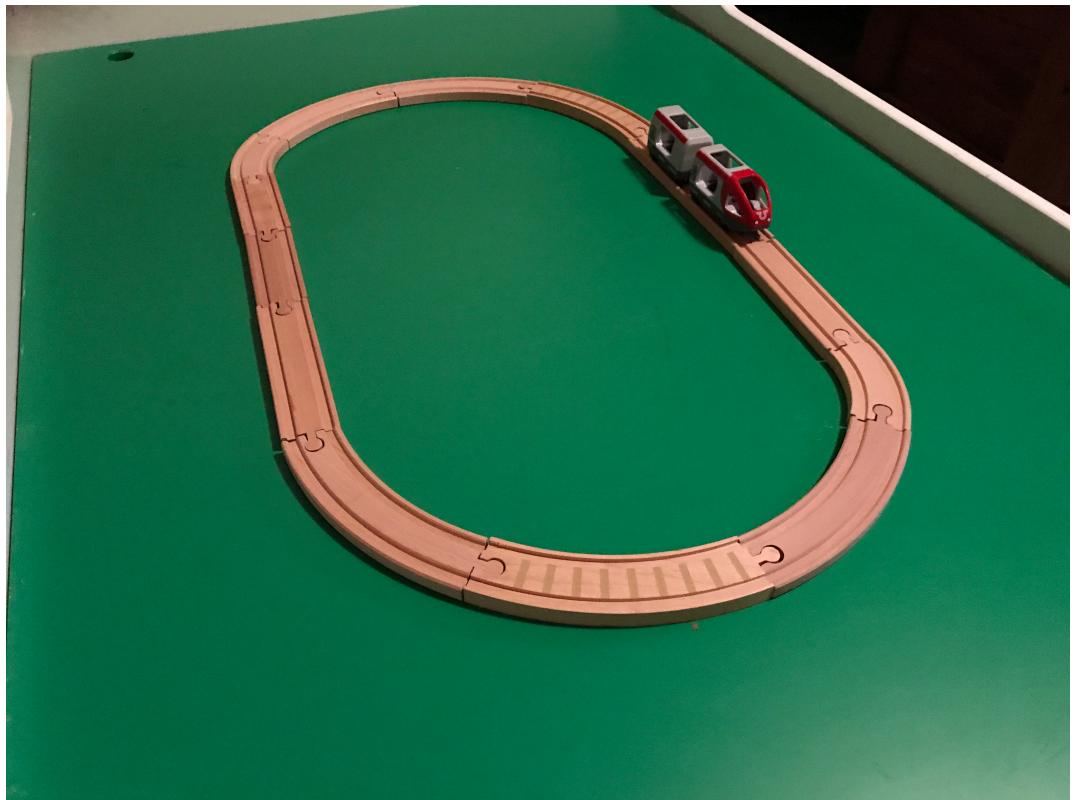
R has been advancing by leaps in bounds in terms of what it can do and the elegance with which it does it, in large part because of the enormous contributions of people involved with RStudio.

## 1.2 The “package” system

### 1.2.1 R packages

```
fortune("Take that, SAS!")  
  
##  
## 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 *packages*. 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 1.2.



**Figure 1.2:** 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 1.3.



**Figure 1.3:** The toy version of what your R set-up will look like once you find cool packages to use for your research.

Each package is basically a bundle of extra R functions. 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 “Advanced” section of this course, you will learn how to write your own R package.

### 1.2.2 Installing from CRAN

The most popular place from which to get packages is currently CRAN, which has over 10,000 R packages available. You can install packages from CRAN

using R code, with the `install.packages` function. For example, telephone keypads include letters for each number (Figure 1.4), which allow companies to have “named” phone numbers that are easier for people to remember, like 1-800-GO-FEDEX and 1-800-FLOWERS.



**Figure 1.4:** Telephone keypad with letters corresponding to each number.

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("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.

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.

```
fortune("pizza")
```

```
##  
## Roger D. Peng: I don't think anyone actually believes that R is designed  
## to make *everyone* happy. For me, R does about 99% of the things I need to
```

```

## do, but sadly, when I need to order a pizza, I still have to pick up the
## telephone.
## Douglas Bates: There are several chains of pizzerias in the U.S. that
## provide for Internet-based ordering (e.g. www.papajohnsonline.com) so,
## with the Internet modules in R, it's only a matter of time before you will
## have a pizza-ordering function available.
## Brian D. Ripley: Indeed, the GraphApp toolkit (used for the RGui interface
## under R for Windows, but Guido forgot to include it) provides one (for use
## in Sydney, Australia, we presume as that is where the GraphApp author
## hails from). Alternatively, a Padovian has no need of ordering pizzas with
## both home and neighbourhood restaurants ....
##     -- Roger D. Peng, Douglas Bates, and Brian D. Ripley
##     R-help (June 2004)

```

### 1.2.3 Loading an installed package

When you open RStudio, unless you reload the history of a previous R session (which we typically **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.

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(phonenumber)
```



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 above code, 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 (although, if you want, this function also works if you follow the rule and call `library("phonenumbers")` instead).

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 `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(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.

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("phonenumbers")
```

```
##
## To cite package 'phonenumbers' in publications use:
##
##   Steve Myles (2015). phonenumbers: Convert Letters to Numbers and
##   Back as on a Telephone Keypad. R package version 0.2.2.
##   https://CRAN.R-project.org/package=phonenumbers
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {phonenumbers: Convert Letters to Numbers and Back as on a Telephone Keypad},
##     author = {Steve Myles},
##     year = {2015},
```

```
##   note = {R package version 0.2.2},
##   url = {https://CRAN.R-project.org/package=phonenumbers},
## }
```



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., `phonenumbers::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.

## 1.3 Basic code conventions of R

### 1.3.1 R's MVP: The *gets arrow*

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 `<-`. The basic structure of a call with a gets arrow looks like this:

```
## Note: Generic code
[name of object] <- [thing I want to save]
```



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.

In R, objects are the way to save something to use again later in your code. If you do not assign something to an object with the gets arrow, R will just print it back out to you at the console. For example, if I just type `"GoFedEx"` at the R console, R will print that string back to me, but won't save it anywhere for me to use later:

```
"GoFedEx"
```

```
## [1] "GoFedEx"
```

However, if I assign "GoFedEx" to an object using a gets arrow, I can print it out or use it later by typing ("referencing") that object name:

```
fedex_number <- "GoFedEx"
fedex_number
```

```
## [1] "GoFedEx"
```

```
letterToNumber(fedex_number)
```

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

You can assign the output of a function call directly to an object. For example, if you wanted to save the output of the `letterToNumber` call in the above code to its own object, you could do that with the call:

```
fedex_actual_number <- letterToNumber(fedex_number)
fedex_actual_number
```

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

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"
```

If you would like to see all the R objects that are currently defined in your R session, you can do that with the `ls` command:

```
ls()
```

```
## [1] "a"                      "b"                      "fedex_actual_number"
## [4] "fedex_number"
```

If you want to see the contents of an object, just print its name. For example, to see what we have stored in the `fedex_number` object, you can run:

```
fedex_number
```

```
## [1] "GoFedEx"
```

In R, at heart everything is an object, so this printing even works for things like functions. For example, to see the source code with which the `letterToNumber` function is defined, you can run (notice the lack of parentheses):

```
letterToNumber
```

```
## function (value, qz = 1)
## {
##   value <- as.character(value)
##   value <- gsub("[^A-Za-z0-9]", "-", value)
##   value <- toupper(value)
##   valueSplit <- strsplit(value, "")[[1]]
##   numString <- as.character()
##   maphone <- function(char) {
##     if (qz == 0 && (char == LETTERS[17] | char == LETTERS[26])) {
##       "0"
##     }
##     else {
##       ifelse(is.element(char, LETTERS[1:3]), "2",
##              ifelse(is.element(char, LETTERS[4:6]), "3",
##                     ifelse(is.element(char, LETTERS[7:9]),
##                            "4",
##                            ifelse(is.element(char, LETTERS[10:12]),
##                                   "5",
##                                   ifelse(is.element(char, LETTERS[13:15]),
##                                         "6",
##                                         ifelse(is.element(char, LETTERS[16:19]),
##                                                "7",
##                                                ifelse(is.element(char, LETTERS[20:22]),
##                                                       "8",
##                                                       ifelse(is.element(char, "-") | suppressWarnings(!is.na(as.numeric
##                                                               char, "9")))))))))
##     }
##   }
##   numString <- lapply(valueSplit, maphone)
##   return(paste0(numString, collapse = ""))
## }
```

```
## }
## <bytecode: 0x3cdb5c0>
## <environment: namespace:phonenumbers>
```

### 1.3.2 Assignment operator wars: `<-` vs. `=`

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:

```
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”.

### 1.3.3 Naming objects

When you assign objects, you will need to choose names for them. This object name is what you will type later in your code to reference the object and use it in functions, figures, etc. For example, with the following code, I am assigning the character string “GoFedEx” to an object that I am naming `fedex_number`:

```
fedex_number <- "GoFedEx"
```

There are only two fixed rules for naming objects in R:

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

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 (`fedex_number`, not `FedExNumber`)
- Use an underscore as a separator (`fedex_number`, not `fedex.number` or `fedexNumber`)
- Avoid using names that are already defined in R (e.g., don't name an object `mean`, because a function named `mean` already exists)

```
fortune("call your matrix")
```

```
##  
## Firstly, 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., `fedex_number`) and later, when you start writing functions, name those after verbs (e.g., `call_fedex`). 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.

## 1.4 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).

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).

### 1.4.1 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. All values in a vector must be of the same class (i.e., all numbers, all characters, all dates). If you try to create a vector with elements from different classes (like "FedEx", which is a character, and 3, a number), R will coerce all of the elements to the most generic class 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).

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 a vector with the first five elements of the Fibonacci sequence, you can run:

```
fibonacci <- c(1, 1, 2, 3, 5)
fibonacci
```

```
## [1] 1 1 2 3 5
```

Here is an example of creating a vector using elements with the character class instead of numbers (note the quotation marks used around each element for character strings):

```
one_to_five <- c("one", "two", "three", "four", "five")
one_to_five
```

```
## [1] "one"    "two"    "three"   "four"   "five"
```

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"
```

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(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 elements you want to pull, with a numeric vector inside the brackets that lists the numbered positions of the elements you want to get:

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

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

```
fibonacci[c(1, 5)] # Get first and fifth values
```

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

```
fibonacci[1:3] # Get the first three values
```

```
## [1] 1 1 2
```

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 statements to index vectors later in the book.

### 1.4.2 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. You can create dataframes using the `data.frame` function. However, most often you will create a dataframe by reading in data from a file, using something like `read.csv`.

To create a dataframe using `data.frame`, in this case by sticking together vectors you already have saved as R objects, you can run:

```
fibonacci_seq <- data.frame(num_in_seq = one_to_five,
                             fibonacci_num = fibonacci)
fibonacci_seq
```

	num_in_seq	fibonacci_num
## 1	one	1
## 2	two	1
## 3	three	2
## 4	four	3
## 5	five	5

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

You can also create a dataframe using `data.frame` even if you don't have the vectors for the columns saved as objects. Instead, in this case, you can put the vector assignment directly within the `data.frame` call:

```
fibonacci_seq <- data.frame(num_in_seq = c("one", "two", "three",
                                             "four", "five"),
                             fibonacci_num = c(1, 1, 2, 3, 5))
fibonacci_seq
```

	num_in_seq	fibonacci_num
## 1	one	1
## 2	two	1
## 3	three	2
## 4	four	3
## 5	five	5



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 `data.frame` 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.

The general format for using `data.frame` is:

```
## Note: Generic code
[name of object] <- data.frame([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.

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 `fibonacci_seq` dataframe we created above:

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

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

```
fibonacci_seq[5, ] # Last row, all columns
```

```
##   num_in_seq fibonacci_num
## 5      five            5
```



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.

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 `read.csv` 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`”, you can read it into your R session with the following code:

```
daily_show <- read.csv("daily_show_guests.csv",
                       header = TRUE,
                       skip = 4)
```

In this code, the `read.csv` function is reading in the data from “`daily_show_guests`”, 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.

Once you've read in the data and saved the resulting dataframe as an object, you can use square-bracket indexing to look at the first two rows in the data:

```
daily_show[1:2, ]
```

```
##   YEAR GoogleKnowlege_Occupation Show Group Raw_Guest_List
## 1 1999                         actor 1/11/99 Acting Michael J. Fox
## 2 1999                         Comedian 1/12/99 Comedy Sandra Bernhard
```

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

```
dim(daily_show)
```

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

```
nrow(daily_show)
```

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

```
ncol(daily_show)
```

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

## 1.5 Using R functions

### 1.5.1 Function structure

In general, functions in R take the following structure:

```
## Generic code
function.name(parameter 1 = argument 1, parameter 2 = argument 2,
               parameter 3 = argument 3)
```

The result of the function will be output to your R session, unless you choose to save the output in an object:

```
## Generic code
new_object <- function.name(parameter 1 = argument 1,
                           parameter 2 = argument 2,
                           parameter 3 = argument 3)
```

Here are some example function calls, to give you examples of this structure:

```
head(daily_show)
```

```
##   YEAR GoogleKnowlege_Occupation Show Group Raw_Guest_List
## 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
```

```
head(daily_show, n = 3)
```

```
##   YEAR GoogleKnowlege_Occupation Show Group Raw_Guest_List
## 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
```

```
daily_show <- read.csv("daily_show_guests.csv",
                       skip = 4,
                       header = TRUE)
```

Within the function call, *parameters* allow you to customize the function to run in a certain way (e.g., use a certain dataframe as an input, give output in a certain format). Some function parameters will have *default arguments*, which means that you don't have to put a value for that parameter for the function to run, but you can if you want the function to do something other than the default.

### 1.5.2 Function help files

You can find out more about a function, include what parameters it has and what the default values, if any, are by using `?`  before the function name in the R console. For example, to find out more about the `read.csv` command, run:

```
?read.csv
```

From the “Usage” section of the help file, you can figure out that the only required parameter is `file`, the pathname of the file that you want to read in, since this is the only argument in the “Usage” example without an argument value:

```
read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

You can also see from this “Usage” section that the default value of `header` is `TRUE`, the default value of `sep` is a comma, etc.

The “Arguments” section explains each of the parameters, and possible arguments that each can take. For example, here is the explanation of the `nrows` parameter in the `read.csv` function:

```
integer: the maximum number of rows to read in. Negative and other
invalid values are ignored.
```

From this, you can determine that you should put in a whole number, 1 or higher, and the function will only read in that many lines of the dataframe when you run `read.csv`.

### 1.5.3 Function parameters

Each function parameter has a name (e.g., `nrows`, `header`, `file`). The safest way to call a function in R is to use the structure `parameter name = argument value` for every parameter, like this:

```
head(x = daily_show, n = 3)
```

However, you can also give argument values by position. For example, in the `head` function, the first parameter is `x`, the object you want to look at, and the second is `n`, the number of elements you want to include when you look at the object. If you know this, you can call `head` using the shorter call:

```
head(daily_show, 3)
```

If you use position alone, you will have problems if you don’t include arguments in exactly the right order. However, if you use parameter names to set each argument, it doesn’t matter what order you include arguments when calling a function:

```
# These two calls return the exact same object
head(x = daily_show, n = 3)
head(n = 3, x = daily_show)
```

Because code tends to be more robust to errors when you use parameter names to set arguments, we recommend against using position, rather than name, to give arguments when calling functions, at least while you’re learning R. It’s too easy to forget the exact order and get errors in your code. However, there is

one exception— the first argument to a function is almost always required (i.e., there’s not a default value), and you very quickly learn what the first parameter of most functions are as soon as you start using the function regularly. Therefore, it’s fine to use position alone to specify the first argument in a function, but for now always use the parameter name to set any later arguments:

```
head(daily_show, n = 3)
```



Using the full parameter names for arguments can take a bit more time, since it requires more typing. However, RStudio helps you out with that by offering *code completion*. Once you start typing the first few letters of a parameter name within a function call, try pressing the tab key. All possible arguments that start with those letters should show up, and you can scroll through to pick the right one, or keep typing until the argument you want is at the top of the list of choices, and then press the tab key again.

## 1.6 R scripts

This is a good point in learning R for you to start putting your work 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” -> “New File” -> “R Script”. Alternatively, you can use the keyboard shortcut Command-Shift-N. Figure 1.5 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

The screenshot shows an RStudio interface with a script editor containing 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",
8                               "Exploring Data 1"))
9
10 a <- 1:4 ; b <- rnorm(10)
11
```

Annotations with arrows point to specific parts of the code:

- 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.
- An arrow points from the text "One command" to the assignment statement `one_to_ten <- 1:10`.
- An arrow points from the text "Two commands on one line" to the line `a <- 1:4 ; b <- rnorm(10)`.

Figure 1.5: Example of an R script in RStudio.

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.

### 1.6.1 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

## 1.7 In-course Exercise

### 1.7.1 About the dataset

For 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:**

- Check out the Creative Commons license. What are we allowed to do with this data? What restrictions are there on using the data?
- 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.

If you have extra time:

- Check out the related article on FiveThirtyEight. What are some specific questions they used this data to answer for this article?
- Who is Nate Silver?

### 1.7.2 Manually creating vectors and a dataframe

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 to an object named `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 call 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 set, 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 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.
- Combine these two vectors 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 square bracket indexing 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:

```
# 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]
five_guests[c(3, 5)]
five_guests[2:4]

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

# 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(five_guests)
class(appearance_year)

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

# Use square bracket indexing to print subsets of the data frame
guest_list[1, 2]
guest_list[3:5, 1]
guest_list[, 1]
```

```
guest_list[c(1, 3), ]  
  
# Use `str` to check out the structure of the data frame you created  
str(guest_list)
```

### 1.7.3 Installing and using a package

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("stringr")  
  
# Load the package into your current R session  
library(stringr)  
  
# Open the package's vignette  
vignette("stringr")  
  
# Convert the `five_guests` strings to lowercase  
str_to_lower(five_guests)  
  
# Convert the `five_guests` strings to uppercase  
str_to_upper(five_guests)
```

### 1.7.4 Getting the data onto your computer

Next, we will work with the whole dataset. Download the data from GitHub onto your computer. Make a directory (folder) on your computer specifically for this course (I strongly recommend that you put it somewhere where the file path will not have any spaces in it— for example, putting it in your home directory, under the name “R\_Prog\_Course” would be great. Putting it in a directory called “R Prog Course” would not.). Put the *Daily Show* data in your directory for this course.

#### Take the following steps to get the data onto your computer

- Find out what your home directory is in R. To do this, make sure R is set to your home directory using `setwd("~/")`, and then get R to print that home directory path using `getwd()`.
- If you do not yet have a directory (folder) just for this course, make one someplace straightforward like in your home directory. Do not use any spaces in the directory name.
- 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.
- Outside of R, open Finder (for Macs), Windows Explorer (for Windows), or your system’s equivalent. Go to your home directory (mine, for example, is `/Users/brookeanderson`). Figure out how to get from your home directory to the directory you created for this course. If you create the course directory as a subdirectory in your home directory, this will just be one step from your home directory.
- Go back into R. Set R’s working directory to your directory for this class using the `setwd` command, now that you know the pathname for the directory. For example, I would use `setwd("~/RProgrammingForResearch/data")`.
- Use the `list.files` command to make sure that the “`daily_show_guests.csv`” file is in your current working directory.

The full R code for this task might look something like:

```
# Set the working directory to your home directory
setwd("~/")

# Print out the current working directory
getwd()

# Example code to re-set the working directory to your course directory
setwd("~/RProgrammingForResearch")

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

```
"daily_show_guests.csv" %in% list.files()
```

If you have extra time:

- See if you can figure out the last line of code in the example R code.

### 1.7.5 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 next week.) You can read csv files into R using the `read.csv` function.

#### Read the data into your R session

- Use the `read.csv` function to read the data into R and save it as the object `daily_show`.
- 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. Why are we using the option `header = TRUE`? Can you figure out why we're using the `skip` option, and why it's set to 4?
- 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")` rather than `daily_show <- read.csv("daily_show_guests.csv")`.)

Example R code:

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

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("daily_show_guests.csv", header = TRUE,
                                skip = 4, nrows = 10)
daily_show_first10
```

### 1.7.6 Checking out the data

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 (all the guests who came on The Daily Show with Jon Stewart), do you think it has about the right number of rows?
- Use indexing to look at the first two rows of the dataset. Based on this, what does each row “measure”? What information do you get for each “measurement”? As a reminder, indexing uses square brackets immediately after the object name. If the object has two dimensions, like a dataframe (rows and columns), you put the rows you want, then a comma, then the columns you want. If you want all rows (or columns), you leave that space blank. For example, if you wanted to get the first two rows and the first three columns, you'd use `daily_show[1:2, 1:3]`. If you wanted to get the first five rows and all the columns, you'd use `daily_show[1:5, ]`.
- 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?
- Use the `str` function to get more details about the `daily_show` dataframe you read in.

Example R code:

```
# Use indexing to see a subset of the data
daily_show[1:2, ]
```

```
# Check the dimensions of the data
dim(daily_show)
head(daily_show)
tail(daily_show)
```

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(daily_show, n = 10)
```

### 1.7.7 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

- The `subset` function 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 `subset(daily_show, YEAR == 1999)`. Check out the helpfile for `subset` and use the function to create a new dataframe that only has the rows of `daily_show` when Nate Silver was a guest. Put it in the object `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 indexing: `nate_silver[ , 3]` (since `Show` is the third column), `nate_silver[ , "Show"]`, or `nate_silver$Show`. Try these.

Example R code:

```
# Create a subset of the data with just Nate Silver appearances
nate_silver <- subset(daily_show,
```

```
Raw_Guest_List == "Nate Silver")

# Investigate this subset of the data
nate_silver
dim(nate_silver)
nrow(nate_silver)
nate_silver[ , 3]
nate_silver[ , "Show"]
```

If you have extra time:

- When you print out the Show column, why does it also print out something underneath about Levels? Hint: This has to do with the class that R has saved this column as. What class is it currently? What other classes might we want to consider converting it to as we continue working with the dataset? Check out the example code below to get some ideas.
- 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:

```
class(nate_silver>Show)
```

```
range(nate_silver>Show)
```

```
nate_silver>Show <- as.Date(nate_silver>Show,
                                format = "%m/%d/%y")
range(nate_silver>Show)
diff(range(nate_silver>Show)) # Time between Nate's first and last shows
```

```
statisticians <- subset(daily_show,
                        GoogleKnowlege_Occupation == "Statistician")
statisticians
```

```
num_visits <- table(daily_show[, 2])
head(num_visits) # Note: This is a vector rather than a data frame
names(num_visits[num_visits == 1])
subset(daily_show, GoogleKnowlege_Occupation == "chess player")
subset(daily_show, GoogleKnowlege_Occupation == "mathematician")
subset(daily_show, GoogleKnowlege_Occupation == "orca trainer")
subset(daily_show, GoogleKnowlege_Occupation == "Puzzle Creator")
subset(daily_show, GoogleKnowlege_Occupation == "Scholar")
```

### 1.7.8 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 (including the function to read the data into R). 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 results surprise you?



## **Part II**

### **Part II: Basics**



# Chapter 2

## Entering and cleaning data #1

Download a pdf of the lecture slides covering this topic.

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 (`read.csv`, `read.table`) 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.

### 2.1 Directories and pathnames

#### 2.1.1 Directory structure

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 illustrates the file directory structure on my computer. (Note that I have omitted many, many additional files and directories – this just shows an example of a few directories and files and how they are structured together). 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.

You will have a home directory somewhere near the top of your struc-



**Figure 2.1:** An example of file directory structure.

ture, although it's likely not your root directory. My 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.

### 2.1.2 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] "/home/travis/build/geanders/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". I have never had a compelling reason to change this on my own computer, as I find it very easy to just move around the directories and set a new working directory using pathnames and the `setwd()` function.

Finally, later in the course, we'll talk about using R Projects from within RStudio. 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).

### 2.1.3 File and directory pathnames

Once you get a picture of how your directories and files are organized, you can use pathnames, either absolute or relative, to move around the directories, set a different working directory, and 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 `read.csv`. 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("/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 changed your working directory (for example, if you used `setwd("CourseText")` to move into the `CourseText` directory), this relative pathname would no longer work.

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

Shorthand	Meaning
<code>~`</code>	Home directory
<code>..`</code>	Current working directory
<code>..`..`</code>	One directory up from current working directory
<code>..../..`</code>	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`. If I wanted to change my working directory to the `Downloads` directory, which is a direct subdirectory of my home directory, I could use:

```
setwd("~/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 `(..)`, 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 `CourseTest` directory would be:

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

These relative pathnames would “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 R scripts and data, 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.

You can use absolute or relative pathnames for a number of things:

- To set your working directory: `setwd("../Week2_Aug31")`, for example
- To read in files from a different directory (as shown in the previous examples)
- To list files in a different directory: for example, `list.files("../")` will list all files in the directory directly about your current working directory (the *parent directory* of your working directory)

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.

## 2.2 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(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(c("Sunday", "Monday", "Tuesday"))
## [1] 3

length(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(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.

## 2.3 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
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

*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"
```

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.

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.table` family of functions to read it in. This family of functions includes several specialized functions. All members of the `read.table` family are doing the same basic thing. The only difference is what defaults each function has for the separator (`sep`) and the decimal point (`dec`). Members of the `read.table` family include:

Function	Separator	Decimal point
'read.table'	comma	period
'read.csv'	comma	period
'read.csv2'	semi-colon	comma
'read.delim'	tab	period
'read.delim2'	tab	period

You can use `read.table` to read in any delimited file, regardless of the separator and the value used for the decimal point. However, you will need to specify these values using the `sep` and `dec` parameters if they differ from the defaults for `read.table` (a space for the delimiter and period for the decimal). If you remember the more specialized function call, therefore, you can save yourself some typing. There are a few other default values besides `sep` and `dec` that differ between different functions in this family: `header`, for example, specifies whether the first row should be used as column names.

For example, to read in the Ebola data, which is comma-delimited, you could

either use `read.table` with a `sep` argument specified or use `read.csv`, in which case you don't have to specify `sep`:

```
# These two calls do the same thing
ebola <- read.table("data/country_timeseries.csv", sep = ",",
                     header = TRUE)
ebola <- read.csv("data/country_timeseries.csv")
```

These functions have a number of different parameters to help you tell R how to read in data. For example, if the first few lines of the file aren't part of the tabular data, you can tell R how many rows of the file to skip before it starts reading in the data. If the data uses an unusual value for missing data (e.g., -999), you can specify that, as well. Some of the interesting parameters with the `read.table` family of functions are:

Option	Description
'sep'	What is the delimiter in the data?
'skip'	How many lines of the start of the file should you skip?
'header'	Does the first line you read give column names?
'as.is'	Should you bring in strings as characters, not factors?
'nrows'	How many rows do you want to read in?
'na.strings'	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.table` and `?read.fwf`. You can also use the help files to determine the default values of arguments for each function.

### 2.3.2 The `read_*` functions

The `read.table` family of functions are part of base R. There is a newer package called `readr` that has a family of `read_*` functions. These functions are very similar, but have some more sensible defaults. Compared to the `read.table` family of functions, the `read_*` functions:

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

Functions in the `read_*` family include:

- `read_csv`, `read_tsv` (specific delimiters)
- `read_delim`, `read_table` (generic)
- `read_fwf`
- `read_log`

- `read_lines`

These functions work very similarly to functions from the `read.table` family. For example, to read in the Daily Show guest data, you can call:

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

## Parsed with column specification:
## cols(
##   YEAR = col_integer(),
##   GoogleKnowlege_Occupation = col_character(),
##   Show = col_character(),
##   Group = col_character(),
##   Raw_Guest_List = col_character()
## )
```

The message that R prints after this call (“Parsed with column specification..”) lets you know what classes R used for each column (this function tries to guess the appropriate function and, unlike the `readr` functions, will assign characters to a character rather than factor class – this is usually what you want).



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. Most were developed in part or full by Hadley Wickham and others at RStudio. Many of these packages are only a few 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 use the `tidyverse` package to download all tidyverse packages at once.

### 2.3.3 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. With the `read.table` family of functions, you can read in online flat files from non-secure webpages, but not from secure ones.

For example, to read in the tab-separated file saved at this web address, which is non-secure, you can run:

```
url <- paste0("http://www2.unil.ch/comparativegenometrics",
              "/docs/NC_006368.txt")
ld_genetics <- read_tsv(url)
ld_genetics[1:5, 1:4]
```

```
## # A tibble: 5 x 4
##   pos     nA     nC     nG
##   <int> <int> <int> <int>
## 1 500    307    153    192
## 2 1500   310    169    207
## 3 2500   319    167    177
## 4 3500   373    164    168
## 5 4500   330    175    224
```

Similarly , to read in data from this GitHub repository of Ebola data, which is a secure website, you can run:

```
url <- paste0("https://raw.githubusercontent.com/cmrvrivers/",
              "ebola/master/country_timeseries.csv")
ebola <- read_csv(url)
ebola[1:3, 1:3]

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

### 2.3.4 Saving and loading R objects

You can save an R object you've created as an `.RData` file.

To save an R object in a `.RData` file, use the `save` function:

```
save(ebola, file = "Ebola.RData")
list.files()
```

```

## [1] "_book"           "_bookdown.yml"
## [3] "_build.sh"        "_deploy.sh"
## [5] "_output.yml"      "01-course_info.Rmd"
## [7] "02-prelim.Rmd"    "03-databasics.Rmd"
## [9] "book.bib"         "data"
## [11] "DESCRIPTION"      "Ebola.RData"
## [13] "figures"          "homework.Rmd"
## [15] "index.Rmd"        "LICENSE"
## [17] "packages.bib"     "preamble.tex"
## [19] "README.md"        "references.Rmd"
## [21] "RProgrammingForResearch.Rmd" "RProgrammingForResearch.Rproj"
## [23] "slides"            "style.css"
## [25] "toc.css"           "vocabulary.Rmd"

```

Notice that, once you save the object, a new file named “Ebola.RData” is listed in the files in your current working directory. The default is for R to save the R object in your current working directory; to save it elsewhere, use a full relative or absolute pathname for the `file` argument.

Once you’ve saved an R object, you can re-load it later using the `load` function with the object’s file path. For example, since I’ve saved this R object, I can remove it from my current R workspace using the `rm` function, after which it will not show up when I run `ls`:

```

rm(ebola)
ls()

```

```

## [1] "ld_genetics" "my_dir"      "url"

```

Then I can use the `load` command to re-load the object, after which it will again show up as an object in my R workspace:

```

load("Ebola.RData")
ls()

```

```

## [1] "ebola"       "ld_genetics" "my_dir"      "url"

```

There is one caveat for saving R objects: some people suggest you avoid this if possible, to make your research more reproducible. Imagine someone wants to look at your data and code in 30 years. R might not work the same way, so you might not be able to read an `.RData` file. Notice that, if you try to open an `.RData` file in a text edit, it won’t make any sense. However, you can open flat files (e.g., `.csv`, `.txt`) and R scripts (`.R`) in text editors – you should still be able to do this regardless of what happens to R. Some potential exceptions, when it might be useful to save an R object, include when:

- You have an object that you need to save that has a structure that won’t

- work well in a flat file (a list rather than a dataframe, for example); or
- Your starting dataset is very large, and it would take a long time for you to read in your data fresh every time. In this case it may make sense to do some data cleaning and then save the cleaned R object as a `.RData` file, but be sure to also save the script you used to clean the raw data.

### 2.3.5 Reading 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 Data cleaning

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	<code>'dplyr'</code> 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 four 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("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(daily_show)
```

```
## # A tibble: 6 x 5
##   YEAR GoogleKnowlege_Occupation Show Group Raw_Guest_List
##   <int> <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
```

### 2.4.1 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
- 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(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(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(daily_show,
                      year = YEAR,
                      job = GoogleKnowlege_Occupation,
                      date = Show,
                      category = Group,
                      guest_name = Raw_Guest_List)
head(daily_show, 3)

## # A tibble: 3 x 5
##   year           job     date category   guest_name
##   <int>         <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.

## 2.4.2 Selecting columns

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(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(daily_show, job, date, category, guest_name)
```

```

## # A tibble: 2,693 x 4
##   job     date category   guest_name
##   <chr>   <chr>    <chr>      <chr>

```

```

## <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
## # ... with 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(daily_show, -year)
head(daily_show, 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

```

### 2.4.3 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(dataframe, logical statement)
```

The `logical statement` 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(daily_show, category == "Science")
head(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 statement 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 NA	<code>is.na(job)</code>
<code>!is.na()</code>	is not NA	<code>!is.na(job)</code>
<code>&amp;</code>	and	<code>year == 2015 &amp; category == "Academic"</code>
<code> </code>	or	<code>year == 2015   category == "Academic"</code>

We'll use these logical operators 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.

#### 2.4.4 Add or change columns

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(dataframe,
       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(unique(daily_show$job), 10)
```

```
## [1] "actor"           "Comedian"        "television actress"
## [4] "film actress"    "Singer-lyricist"  "model"
## [7] "stand-up comedian" "actress"         "comedian"
## [10] "Singer-songwriter"
```

To make all the observations in the `job` column lowercase, use the `tolower` function within a `mutate` function:

```
mutate(daily_show, job = tolower(job))
```

```
## # 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
## # ... with 2,683 more rows
```

### 2.4.5 Piping

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(dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
select(dataframe, column_name_1, column_name_2)
filter(dataframe, logical statement)
mutate(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("data/daily_show_guests.csv",
                       skip = 4)
daily_show <- rename(daily_show,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
daily_show <- select(daily_show, -YEAR)
daily_show <- mutate(daily_show, job = tolower(job))
daily_show <- filter(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("data/daily_show_guests.csv",
                       skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
        date = Show,
        category = Group,
```

```

    guest_name = Raw_Guest_List) %>%
select(-YEAR) %>%
mutate(job = tolower(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.

#### 2.4.6 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.

## 2.5 Dates in R

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
<code>character</code>	“Chemistry”, “Physics”, “Mathematics”
<code>numeric</code>	10, 20, 30, 40
<code>factor</code>	Male [underlying number: 1], Female [2]
<code>Date</code>	“2010-01-01” [underlying number: 14,610]
<code>logical</code>	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, you can use the `as.Date` function. For example, to convert the `date` column in the `daily_show` data into a Date class, you can run:

```
daily_show <- mutate(daily_show,
                      date = as.Date(date, format = "%m/%d/%y"))
head(daily_show, 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(daily_show$date)
```

```
## [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$date)
diff(range(daily_show$date))
```

You can convert dates expressed in a number of different ways into a Date class in R, as long as you can explain to R how to parse the format that the date is in before you convert it. The only tricky thing in converting objects into a Date class is learning the abbreviations for the `format` option of the `as.Date` function. Here are some common ones:

Abbreviation	Meaning
%m	Month as a number (e.g., 1, 05)
%B	Full month name (e.g., August)
%b	Abbreviated month name (e.g., Aug)

Abbreviation	Meaning
%y	Two-digit year (e.g., 99)
%Y	Four-digit year (e.g., 1999)
%A	Full weekday (e.g., Monday)
%a	Abberviated weekday (e.g., Mon)

Here are some examples of what you would specify for the `format` argument of `as.Date` for some different original formats of date columns:

Your date	format
10/23/2008	“%m/%d%Y”
08-10-23	“%y-%m-%d”
Oct. 23 2008	“%b. %d %Y”
October 23, 2008	“%B %d, %Y”
Thurs, 23 October 2008	“%a, %d %B %Y”



You must use the `format` argument to specify what your date column looks like **before** it's converted to a Date class, not how you'd like it to look after its converted. Once an objects is in a date class, it will always be printed out using a common format, unless you change it back into a character class. (Confusingly, there is a `format` function that you can use to convert from a Date class to a character class and, in that case, the `format` argument does specify how the final date will look. This is mainly useful as a last step in data analysis, when you're creating plot labels of table columns, for example.)

There is also a function in the tidyverse, called `lubridate`, that helps in parsing dates. In many cases you can use functions from this package to parse dates much more easily, without having to specify specific starting formats.

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"
```

The `lubridate` package has similar functions for other date orders or for date-times, including:

- `dmy`
- `mdy`
- `ymd_h`
- `ymd_hm`

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

```
daily_show <- read_csv("data/daily_show_guests.csv",
                       skip = 4) %>%
  rename(job = GoogleKnowledge_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(date = mdy(date)) %>%
  filter(category == "Science")
head(daily_show, 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(daily_show,
       show_day = wday(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 Tues    Catherine Weitz
## 3 2004-06-15 Tues    Hassan Ibrahim
## 4 2005-09-06 Tues    Dr. Marc Siegel
## 5 2006-02-13 Mon    Astronaut Mike Mullane

```

## 2.6 In-course Exercise

### 2.6.1 Checking out directory structures

Download the whole directory for this week from Github ([https://github.com/geanders/week\\_2\\_data](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, choose “Clone or Download” and then choose “Download ZIP”. This will download a compressed file with the full directory of data, probably to your computer’s “Downloads” folder. Then move the directory into your course directory and “unzip” it (try double-clicking the file, or right click on the file and see if there’s a “decompress” or “unzip” option). Rename this subdirectory “data” (don’t do this in R, just use whatever technique you usually use on your computer to rename directories).

- Look through the structure of the “data” directory. What files are in the directory? What subdirectories? Sketch out the structure of this directory (this should look similar to the “directory structure” slide we went over in the course notes).
- Create a new R script to put all the code you use for this exercise. Create a subdirectory in your course directory called “R” and save this script there using a .R extension (e.g., “week\_2.R”).

### 2.6.2 Using relative and absolute file pathnames

Once you have the data, I’d like you to try using `setwd()` to move around the directories on your computer. For this section of the exercise, you’ll also try to open the same file from different working directories, to get practice using absolute and relative file pathnames.

Start by changing your working directory to the “data” subdirectory you just downloaded. For me, the absolute path to that is `/Users/brookeanderson/RProgrammingForResearch/data`, so to change my working directory to this one using an absolute file pathname, I would run:

```
setwd("/Users/brookeanderson/RProgrammingForResearch/data")
getwd()
```

The absolute pathname will be different for each person, based on the directories on his or her computer and where he or she saved this particular directory.



Since “`~`” is shorthand for my home directory (“`/Users/brookeanderson`”), I could also use the absolute pathname “`~/RProgrammingForResearch/data`” when setting this directory as my working directory.

Now, use the `list.files` function to make sure you have all the files that you just pulled in your current working directory:

```
list.files()
```

```
## [1] "accident.csv"           "App-1"
## [3] "App-2"                  "country_timeseries.csv"
## [5] "daily_show_guests.csv"   "deaths-weather.csv"
## [7] "example_choropleth.Rdata" "fars_colorado.RData"
## [9] "icd-10.xls"              "ICU_Data_Code_Sheet.pdf"
## [11] "icu.sas7bdat"            "ld_genetics.txt"
## [13] "measles_data"            "mexico_deaths.csv"
## [15] "mexico_exposure.csv"     "serial_map_data.csv"
## [17] "serial_phone_data.csv"    "titanic.csv"
```

Try the following tasks:

- Read in the ebola data in `country_timeseries.csv`. Assign it to an R object named `ebola`. How many rows and columns does it have? What are the names of the columns?
- Now try moving up one directory from your current working directory (which should be the “data” directory), into the directory you created for this course. What happens if you use the same code as before to read in the file? What do you need to change to be able to read the file in from here? Try to read in the data from here using:
  - A relative pathname
  - An absolute pathname
- Now move your working directory to be the subdirectory of “data” called “`measles_data`”. Try to read in the Ebola data from this working directory using:

- A relative pathname
- An absolute pathname
- Which method (absolute or relative pathnames) always used the same code, regardless of your current working directory? Which method used different code, depending on the starting working directory?

Example R code:

```
getwd() ## Make sure you're in the "data" directory to start

## [1] "/Users/brookeanderson/RProgrammingForResearch/data"

ebola <- read.csv("country_timeseries.csv", header = TRUE)

ebola[1:5, 1:5]

##           Date Day Cases_Guinea Cases_Liberia Cases_SierraLeone
## 1 1/5/2015    289        2776          NA       10030
## 2 1/4/2015    288        2775          NA       9780
## 3 1/3/2015    287        2769        8166       9722
## 4 1/2/2015    286          NA        8157          NA
## 5 12/31/2014   284        2730        8115       9633

dim(ebola) # To figure out number of rows and columns

## [1] 122 18

colnames(ebola) # To figure out column names

## [1] "Date"                  "Day"                   "Cases_Guinea"
## [4] "Cases_Liberia"         "Cases_SierraLeone" "Cases_Nigeria"
## [7] "Cases_Senegal"          "Cases_UnitedStates" "Cases_Spain"
## [10] "Cases_Mali"             "Deaths_Guinea"      "Deaths_Liberia"
## [13] "Deaths_SierraLeone"    "Deaths_Nigeria"    "Deaths_Senegal"
## [16] "Deaths_UnitedStates"   "Deaths_Spain"       "Deaths_Mali"

## Move up one directory
setwd("..")
getwd()

## Get the file using the relative pathname
ebola <- read.csv("data/country_timeseries.csv", header = TRUE)
```

```
ebola[1:5, 1:5]

##           Date Day Cases_Guinea Cases_Liberia Cases_SierraLeone
## 1  1/5/2015 289       2776        NA     10030
## 2  1/4/2015 288       2775        NA      9780
## 3  1/3/2015 287       2769      8166     9722
## 4  1/2/2015 286         NA      8157        NA
## 5 12/31/2014 284       2730      8115     9633

## Get the file using the absolute pathname
abs_path <- paste0("/Users/brookeanderson/RProgrammingForResearch/",
                   "data/country_timeseries.csv")
abs_path

## [1] "/Users/brookeanderson/RProgrammingForResearch/data/country_timeseries.csv"

ebola <- read.csv(abs_path, header = TRUE)

ebola[1:5, 1:5]

##           Date Day Cases_Guinea Cases_Liberia Cases_SierraLeone
## 1  1/5/2015 289       2776        NA     10030
## 2  1/4/2015 288       2775        NA      9780
## 3  1/3/2015 287       2769      8166     9722
## 4  1/2/2015 286         NA      8157        NA
## 5 12/31/2014 284       2730      8115     9633

## Reset your working directory as your directory for this course
## and then check that you're in the right place
getwd()

## [1] "/Users/brookeanderson/RProgrammingForResearch"

## Move into the measles_data subdirectory
setwd("data/measles_data")
getwd()

## [1] "/Users/brookeanderson/RProgrammingForResearch/data/measles_data"

## Get the file using the relative pathname
ebola <- read.csv("../country_timeseries.csv", header = TRUE)
```

```
ebola[1:5, 1:5]
```

```
##           Date Day Cases_Guinea Cases_Liberia Cases_SierraLeone
## 1 1/5/2015 289      2776          NA        10030
## 2 1/4/2015 288      2775          NA        9780
## 3 1/3/2015 287      2769        8166        9722
## 4 1/2/2015 286          NA        8157          NA
## 5 12/31/2014 284      2730        8115        9633
```

```
# Get the file using the absolute pathname (note: we've already assigned
# the absolute pathname in the `abs_path` object, so we can just use that
# object in the `read.csv` call here, rather than typing out the full
# absolute pathname again.)
ebola <- read.csv(abs_path, header = TRUE)
```

```
ebola[1:5, 1:5]
```

```
##           Date Day Cases_Guinea Cases_Liberia Cases_SierraLeone
## 1 1/5/2015 289      2776          NA        10030
## 2 1/4/2015 288      2775          NA        9780
## 3 1/3/2015 287      2769        8166        9722
## 4 1/2/2015 286          NA        8157          NA
## 5 12/31/2014 284      2730        8115        9633
```

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()` when your working directory is the “data” directory, 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?

### 2.6.3 Reading in different types of files

First, make sure you reset your working directory to your course directory:

```
setwd("~/RProgrammingForResearch/")
```

Now you'll try reading in data from a variety of types of file formats. All of these files are stored in the "data" subdirectory of your current working directory, so you'll use filenames throughout that start with "data/".

Try the following tasks:

- What type of flat file do you think the "ld\_genetics.txt" file is? See if you can read it in 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? Stay in the "data" directory and use a relative pathname to read the file in and save it as the R object `ca_measles`. 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?
- Read in the Excel file "icd-10.xls" and assign it to the object name `icd10`. 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(readr)

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

# Use `read_tsv` to read this file. Because the first line
# of the file is *not* the column names, you need to specify what the column
# names should be with the `col_names` parameter.
ca_measles <- read_tsv("data/measles_data/02-09-2015.txt",
                       col_names = c("city", "count"))
head(ca_measles)

## # A tibble: 6 x 2
##                 city count
##                 <chr> <int>
## 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 this file. Load that.
library(readxl)

# Use the `read_excel` function to read in the file.
icd10 <- read_excel("data/icd-10.xls")
head(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 this file. Load that.
library(haven)

```

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

If you have extra time:

- Is there a way to read the “ld\_genetics.txt” file in using `read_delim()` and specific options? If so, try to read the data in using that function. Why can you use both `read.delim` and `read.table` to read in this file?

Example R code:

```
## Using the read_delim function with the `delim` argument specified to `"\t"`.
ld_genetics <- read_delim("data/ld_genetics.txt", delim = "\t")
```

```
## Parsed with column specification:
## cols(
##   pos = col_integer(),
##   nA = col_integer(),
##   nC = col_integer(),
##   nG = col_integer(),
##   nT = col_integer(),
##   GCsk = col_integer(),
##   TAsk = col_integer(),
##   cGCsk = col_integer(),
##   cTAsk = col_integer()
## )
```

```
ld_genetics[1:5, 1:5]
```

```
## # A tibble: 5 x 5
##   pos    nA    nC    nG    nT
##   <int> <int> <int> <int> <int>
## 1 500    307   153   192   348
## 2 1500   310   169   207   314
## 3 2500   319   167   177   337
```

```
## 4 3500 373 164 168 295
## 5 4500 330 175 224 271
```

### 2.6.4 Cleaning up data

Try out the following tasks:

- You now have an R object called `ebola`. Create an object called `ebola_liberia` that only has the columns with the Date and cases and deaths in Liberia. How many observations do you have?
- Change the column names to `date`, `cases`, and `deaths`.
- What class is the `date` column currently in? Convert it to a Date object. If it's currently a factor, you will need to first convert it to a character (do you know why)? What are the starting and ending dates of this data?
- This data has earliest dates last and latest dates first. Often, we want our data in chronological order. Change the dataset so it's in chronological order. *Hint: You can use row indices to do this, if you use row indices to list the data from last row to first. For example, what do you get if you do 3:1? What about if you do ebola[3:1, ]? Think of how you can extend this, along with nrow() do this task. There are a number of different ways to do this step, and we'll talk about more ways later in the course.*

Example R code:

```
library(dplyr)
## Create a subset with just the Liberia columns and Date
ebola_liberia <- select(ebola, Date, Cases_Liberia, Deaths_Liberia)
head(ebola_liberia)

##           Date Cases_Liberia Deaths_Liberia
## 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 rows does the whole dataset have?
nrow(ebola_liberia)

## [1] 122

## Rename the columns
ebola_liberia <- rename(ebola_liberia,
```

```
date = Date,
cases = Cases_Liberia,
deaths = Deaths_Liberia)
ebola_liberia[1:3, ]
```

```
##      date cases deaths
## 1 1/5/2015    NA     NA
## 2 1/4/2015    NA     NA
## 3 1/3/2015  8166   3496
```

```
## What class is the `date` column?
class(ebola_liberia$date)
```

```
## [1] "factor"
```

```
## Use the `mdy` from `lubridate` to convert to Date class
library(lubridate)
ebola_liberia <- mutate(ebola_liberia,
                       date = mdy(date))
head(ebola_liberia$date)
```

```
## [1] "2015-01-05" "2015-01-04" "2015-01-03" "2015-01-02" "2014-12-31"
## [6] "2014-12-28"
```

```
## What are the starting and ending dates?
range(ebola_liberia$date)
```

```
## [1] "2014-03-22" "2015-01-05"
```

```
## Re-order the dataset from last to first
ebola_liberia <- ebola_liberia[nrow(ebola_liberia):1, ]
ebola_liberia[1:3, ]
```

```
##      date cases deaths
## 122 2014-03-22    NA     NA
## 121 2014-03-24    NA     NA
## 120 2014-03-25    NA     NA
```

If you have extra time:

- Limit the `ebola_liberia` dataset just to the days with non-missing case data. How many observations do you have now?

- Write a pipe chain for the full data cleaning process for creating `ebola_liberia` up to this point. *Hint: Try using the `arrange` function from `dplyr` to re-order the data by dates in this pipe.*
- Try using the basic plotting function, `plot()`, to plot the number of cases over time. Do you think that the `cases` variable is measuring the count of cases for that day, or the cumulative number of cases up to that day? See if you can figure out more on Caitlin Rivers' GitHub documentation. Do you notice any potential data quality issues in this data? *Hint: The `plot()` function takes, as required arguments, the vector you want to plot on the x-axis and then the vector you want to plot on the y-axis, like `plot([x vector], [y vector])`. If you are pulling the vectors from a dataset, you will need to use indexing to pull out the column you want as a vector, like `plot([dataframe name]$[column name for x], [dataframe]$[column name for y])`.*

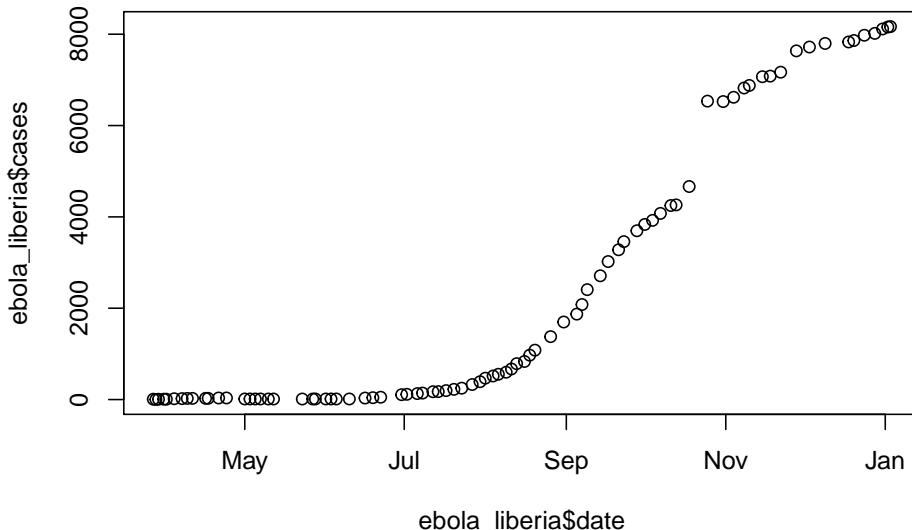
Example R code:

```
## Remove observations with missing cases
ebola_liberia <- filter(ebola_liberia, !is.na(cases))
nrow(ebola_liberia)
```

```
## [1] 83
```

```
## Rewrite the data cleaning for `ebola_liberia` using a pipe chain
ebola_liberia <- ebola %>%
  select(Date, Cases_Liberia, Deaths_Liberia) %>%
  rename(date = Date, cases = Cases_Liberia, deaths = Deaths_Liberia) %>%
  mutate(date = mdy(date)) %>%
  arrange(date) %>%
  filter(!is.na(cases))

## Plot the data
plot(ebola_liberia$date, ebola_liberia$cases)
```



### 2.6.5 A taste of what's to come...

Here's an example to give you a feel for why it's worth learning all these different things about directories, `list.files`, and pathnames.

The `measles_data` subdirectory includes counts of measles made at different times in different cities in California. Say that we wanted to read them all in and make put them into one long dataframe with the variables `city`, `count`, and `date`. You can put together the things you've learned so far, along with a few new ideas (including doing a loop), to do this very easily.

We'll talk more later about using loops and functions to make your programming more efficient, but for right now just look through this code and see if you can get a feel for how it's working (make sure your directory for this course is your working directory):

```
## Create a vector of all the file names in the `measles_data` subdirectory
measles_files <- list.files("data/measles_data")
measles_files

## [1] "02-09-2015.txt" "02-11-2015.txt" "02-13-2015.txt" "02-18-2015.txt"
## [5] "02-20-2015.txt" "02-23-2015.txt" "02-25-2015.txt" "02-27-2015.txt"
## [9] "03-02-2015.txt" "03-06-2015.txt" "03-13-2015.txt" "03-20-2015.txt"
## [13] "03-27-2015.txt" "04-03-2015.txt" "04-10-2015.txt" "04-17-2015.txt"

## Create a vector of all the dates for files by taking the
## ` `.txt` off each of these file names and change it into
```

```

## a date
measles_dates <- sub(".txt", "", measles_files)
measles_dates

## [1] "02-09-2015" "02-11-2015" "02-13-2015" "02-18-2015" "02-20-2015"
## [6] "02-23-2015" "02-25-2015" "02-27-2015" "03-02-2015" "03-06-2015"
## [11] "03-13-2015" "03-20-2015" "03-27-2015" "04-03-2015" "04-10-2015"
## [16] "04-17-2015"

class(measles_dates)

## [1] "character"

measles_dates <- mdy(measles_dates)
measles_dates

## [1] "2015-02-09" "2015-02-11" "2015-02-13" "2015-02-18" "2015-02-20"
## [6] "2015-02-23" "2015-02-25" "2015-02-27" "2015-03-02" "2015-03-06"
## [11] "2015-03-13" "2015-03-20" "2015-03-27" "2015-04-03" "2015-04-10"
## [16] "2015-04-17"

## Before I show the loop, let me talk you through some
## of the parts of it:
i <- 1 # I'm setting the index to 1

## Now I'll use `paste0` to create the first file name
## I want to read.
file_name <- paste0("data/measles_data/", measles_files[i])
file_name

## [1] "data/measles_data/02-09-2015.txt"

## Now I'll read in that tab-delimited file
df <- read_tsv(file_name, col_names = c("city", "count"))
head(df)

## # A tibble: 6 x 2
##       city   count
##       <chr> <int>
## 1 ALAMEDA     6
## 2 LOS ANGELES    20
## 3 City of Long Beach    2
## 4 City of Pasadena      4

```

```

## 5           MARIN      2
## 6           ORANGE    34

## Now I'll add on a column with the date for all the
## values from that file. Notice that I'm using `i` to
## index this, as well

df <- mutate(df, date = measles_dates[i])
head(df)

## # A tibble: 6 x 3
##       city   count      date
##   <chr>   <int>     <date>
## 1 ALAMEDA     6 2015-02-09
## 2 LOS ANGELES 20 2015-02-09
## 3 City of Long Beach    2 2015-02-09
## 4 City of Pasadena    4 2015-02-09
## 5 MARIN        2 2015-02-09
## 6 ORANGE      34 2015-02-09

## Loop through and read in files. After the first file,
## add on the new information to the data that's already
## been read in. Note that you can use `rbind` to add on
## new rows to a data frame as long as the new and old rows
## have the same number of columns and the same column names.
for(i in 1:length(measles_files)){
  file_name <- paste0("data/measles_data/", measles_files[i])
  df <- read.delim(file_name, header = FALSE,
                   col.names = c("city", "count"))
  df <- mutate(df, date = measles_dates[i])
  if(i == 1){
    ca_measles <- df
  } else {
    ca_measles <- rbind(ca_measles, df)
  }
}

dim(ca_measles)

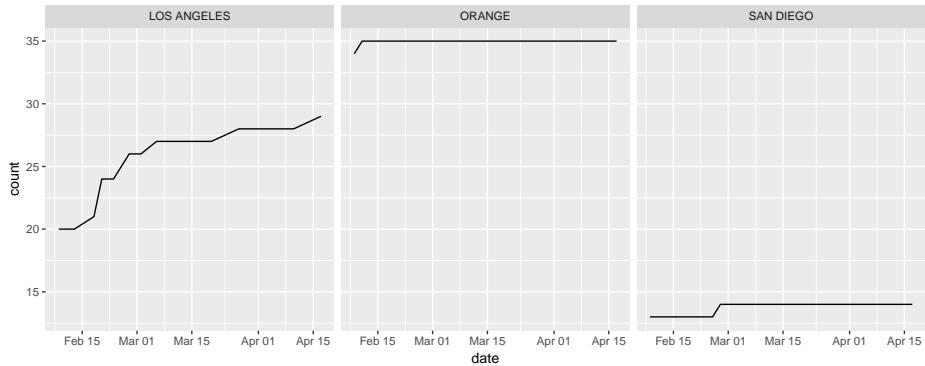
## [1] 232    3

summary(ca_measles)

## # A tibble: 232 x 3
##       city   count      date
##   <chr>   <int>     <date>
## 1 ALAMEDA     6 2015-02-09
## 2 LOS ANGELES 20 2015-02-09
## 3 City of Long Beach    2 2015-02-09
## 4 City of Pasadena    4 2015-02-09
## 5 MARIN        2 2015-02-09
## 6 ORANGE      34 2015-02-09
## # ... with 226 more rows, and 1 more variable:
## #   date: Date, format: "2015-02-09"
## #   .. default language: en_US.UTF-8 ...
## #   .. unit: UTC
## #   .. tz: UTC
## #   .. format: "%Y-%m-%d" 
```

```
## ALAMEDA : 16 Min. : 1.000 Min. :2015-02-09
## City of Long Beach: 16 1st Qu.: 2.000 1st Qu.:2015-02-20
## City of Pasadena : 16 Median : 4.000 Median :2015-03-02
## LOS ANGELES : 16 Mean : 8.698 Mean :2015-03-07
## MARIN : 16 3rd Qu.:12.000 3rd Qu.:2015-03-27
## ORANGE : 16 Max. :35.000 Max. :2015-04-17
## (Other) :136
```

```
library(ggplot2)
ggplot(subset(ca_measles, city %in% c("LOS ANGELES", "SAN DIEGO", "ORANGE")),
       aes(x = date, y = count)) +
  geom_line() +
  facet_grid(. ~ city)
```



# Chapter 3

## Exploring data #1

Download a pdf of the lecture slides covering this topic.

### 3.1 Data from a package

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

1. From flat files (either on your computer or online)
2. From files like SAS and Excel
3. From R objects (i.e., using `load()`)

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* (available as an ebook from the CSU library), 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.

## 3.2 Plots to explore data

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.

### 3.2.1 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))
```

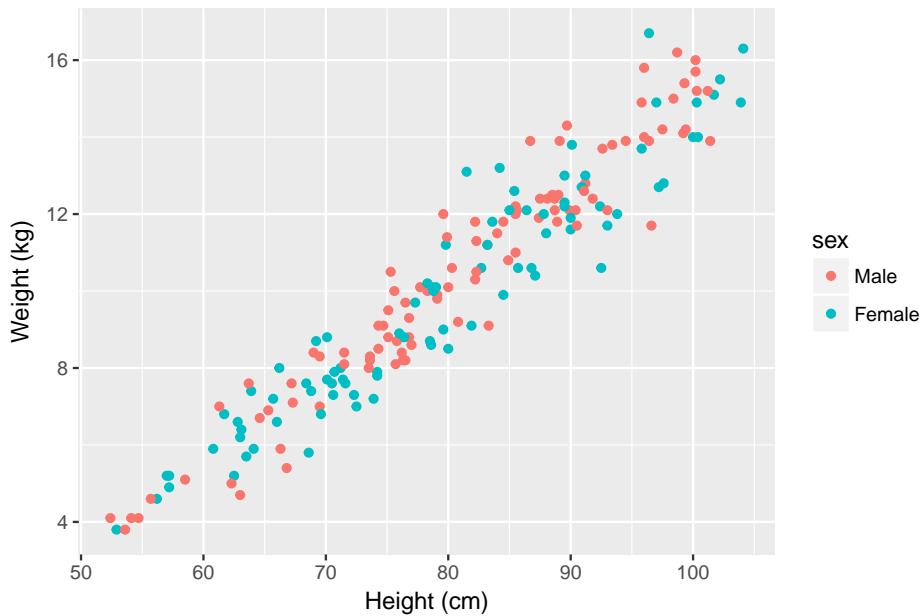
The data frame is the first parameter in a `ggplot` call and, if you like, you can use the parameter definition with that call (e.g., `data = dataframe`). Aesthetics are defined within an `aes` function call that typically is used within the `ggplot` call.



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.

### 3.2.2 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
'x'	Position on x-axis
'y'	Position on y-axis
'shape'	Shape
'color'	Color of border of elements
'fill'	Color of inside of elements
'size'	Size
'alpha'	Transparency (1: opaque; 0: transparent)
'linetype'	Type of line (e.g., solid, dashed)

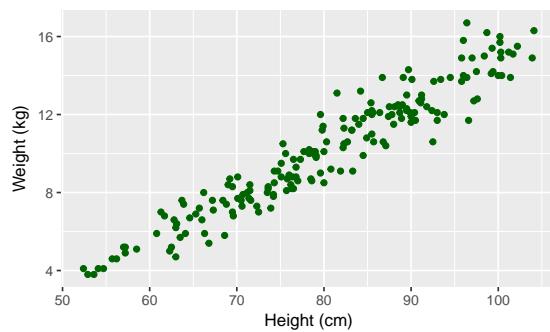
### 3.2.3 Adding geoms

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>

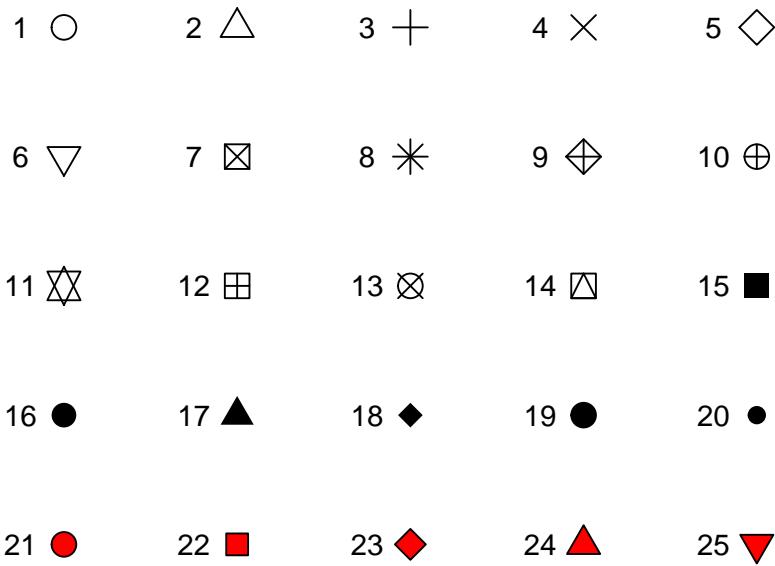
### 3.2.4 Constant aesthetics

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).



**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.

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”.

### 3.2.5 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

### 3.2.6 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.

- 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:

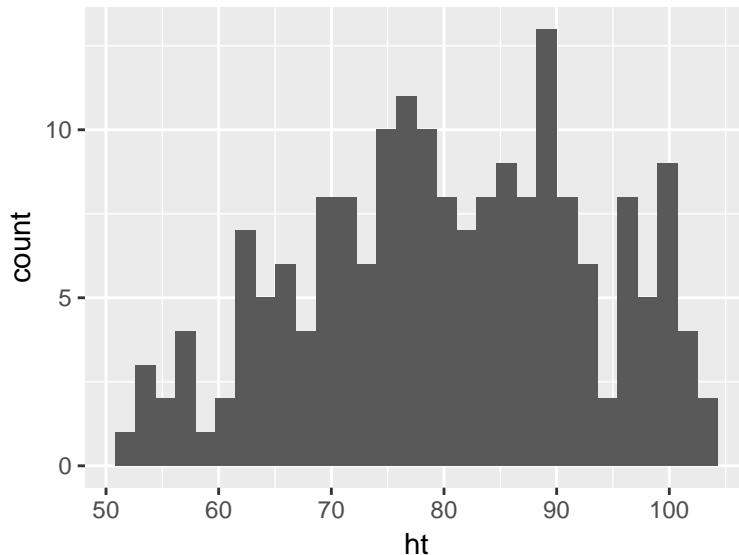
```
head(nepali)

##      id    sex   wt   ht age
## 1 120011  Male 12.8 91.2 41
## 2 120012 Female 14.9 103.9 57
## 3 120021 Female  7.7 70.1  8
## 4 120022 Female 12.1 86.4 35
## 5 120023  Male 14.2 99.4 49
## 6 120031  Male 13.9 96.4 46
```

### 3.2.7 Histograms

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()
```



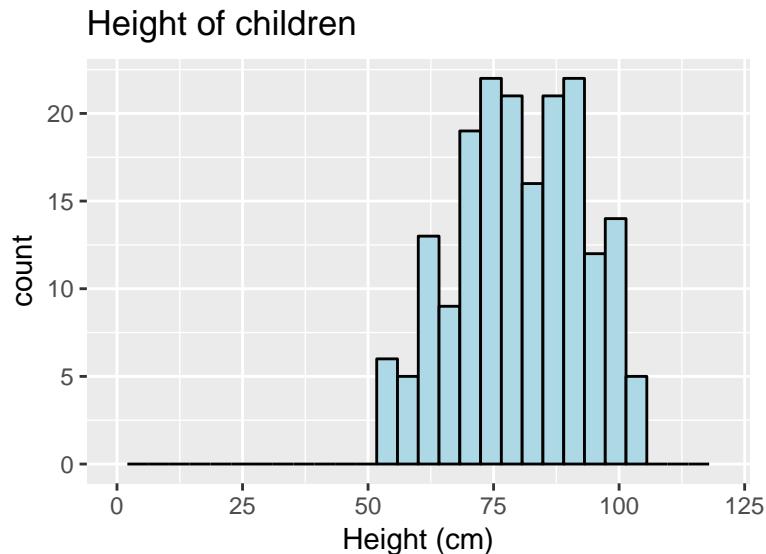
**Figure 3.4:** Basic example of plotting a histogram with ‘ggplot2’. This histogram shows the distribution of heights for the first recorded measurements of each child in the ‘nepali’ dataset.



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`).

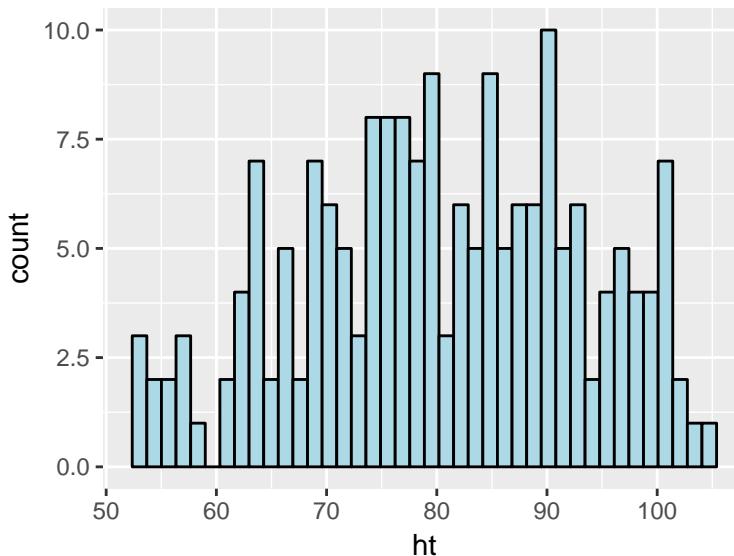
```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black") +
  ggtitle("Height of children") +
  xlab("Height (cm)") + xlim(c(0, 120))
```



**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)
```



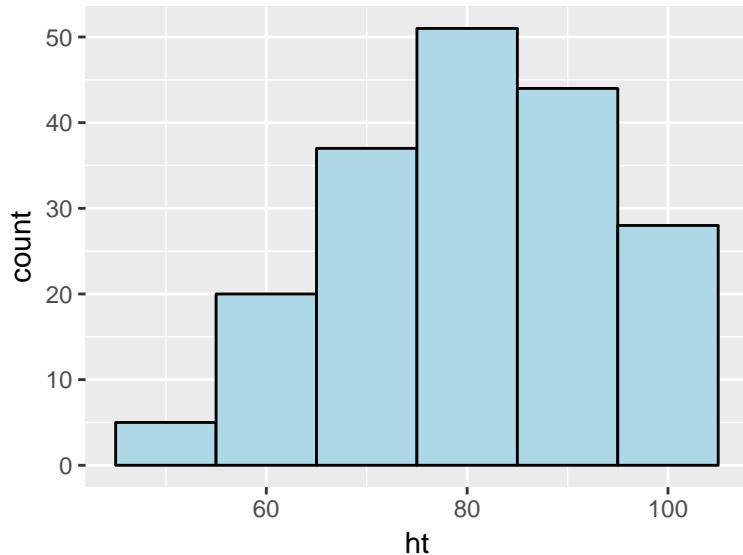
**Figure 3.6:** Example of using the ‘bins’ argument to change the number of bins used in a histogram.

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).

```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black",
  binwidth = 10)
```

### 3.2.8 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:



**Figure 3.7:** Example of using the ‘binwidth’ argument to set the width of each bin used in a histogram.

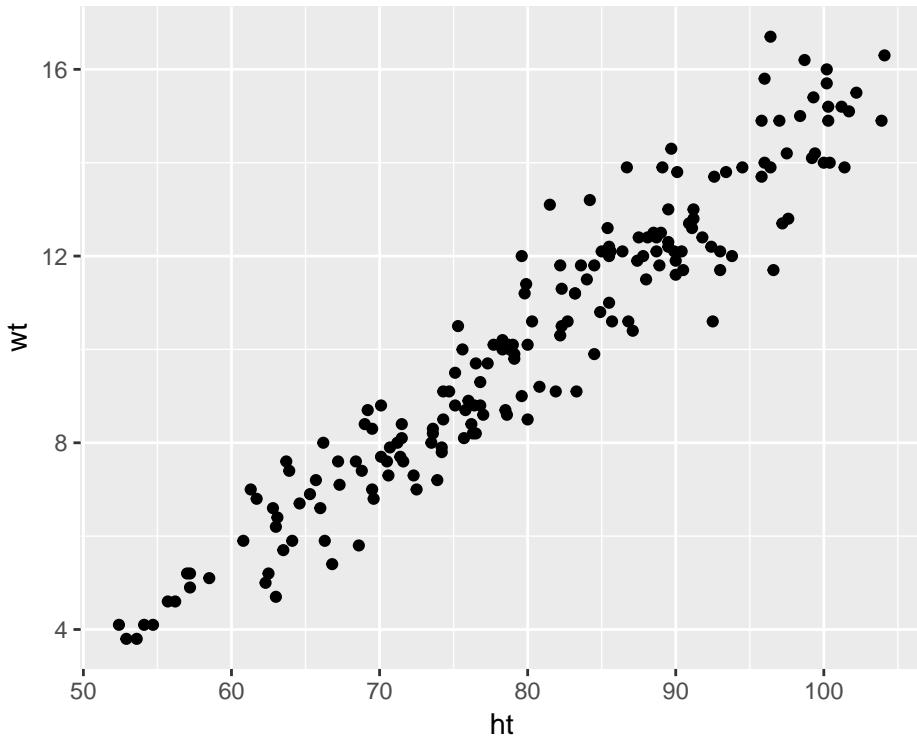
```
ggplot(nepali, aes(x = ht, y = wt)) +
  geom_point()
```

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:

```
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)")
```



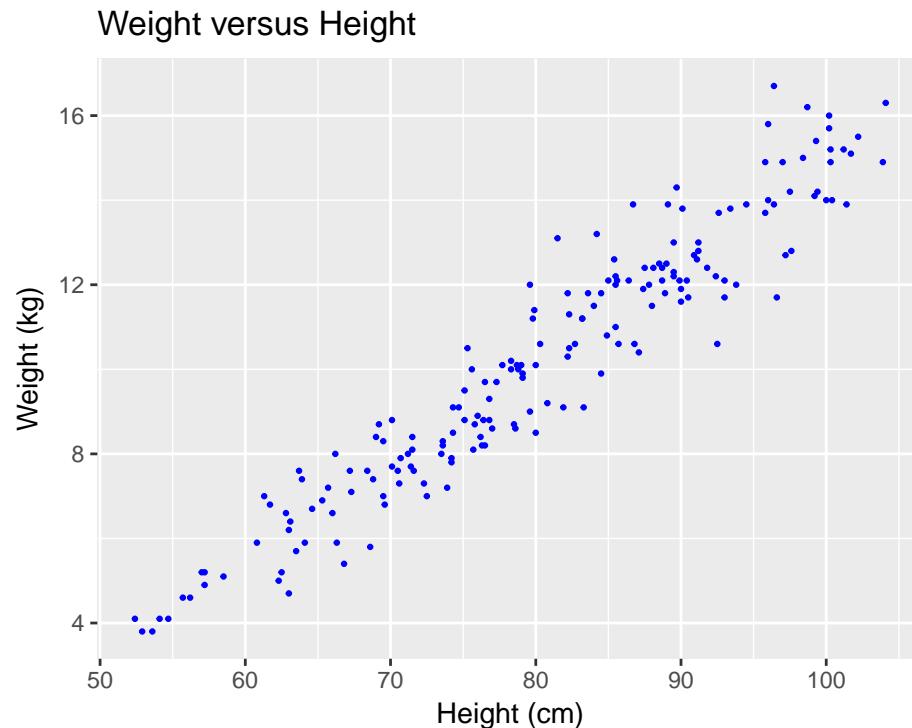
**Figure 3.8:** Example of creating a scatterplot. This scatterplot shows the relationship between children's heights and weights within the nepali dataset.

### 3.2.9 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 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)")
```

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.



**Figure 3.9:** Example of adding ggplot elements to customize a scatterplot.

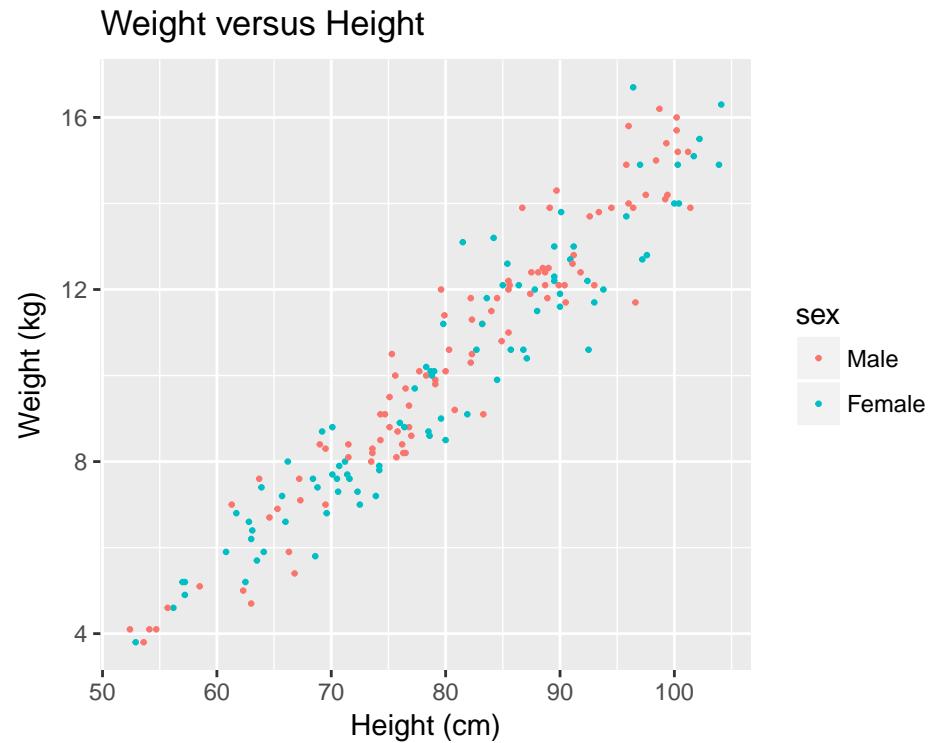
```
ggplot(nepali, aes(x = sex, y = ht, group = sex)) +
  geom_boxplot() +
  xlab("Sex") + ylab("Height (cm)")
```

### 3.2.10 Extensions of ggplot2

There are lots of R extensions for creating other interesting plots. For example, you can use the `ggpairs` function from the `GGally` package to plot all pairs of scatterplots for several variables (Figure ??).

```
library(GGally)
ggpairs(nepali %>% select(sex, wt, ht, age))
```

Notice how this output shows continuous and binary variables differently. For example, the center diagonal shows density plots for continuous variables, but a bar chart for the categorical variable.



**Figure 3.10:** Example of mapping color to an element of the data in a scatterplot.

See <https://www.ggplot2-exts.org> to find more ggplot2 extensions.

## 3.3 Simple statistics functions

### 3.3.1 Summary statistics

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

Function	Description
<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"`.

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

```
## [1] 10.18432
```

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

```
## [1] 52.4 104.1
```

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

```
## [1] 12.64529
```

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

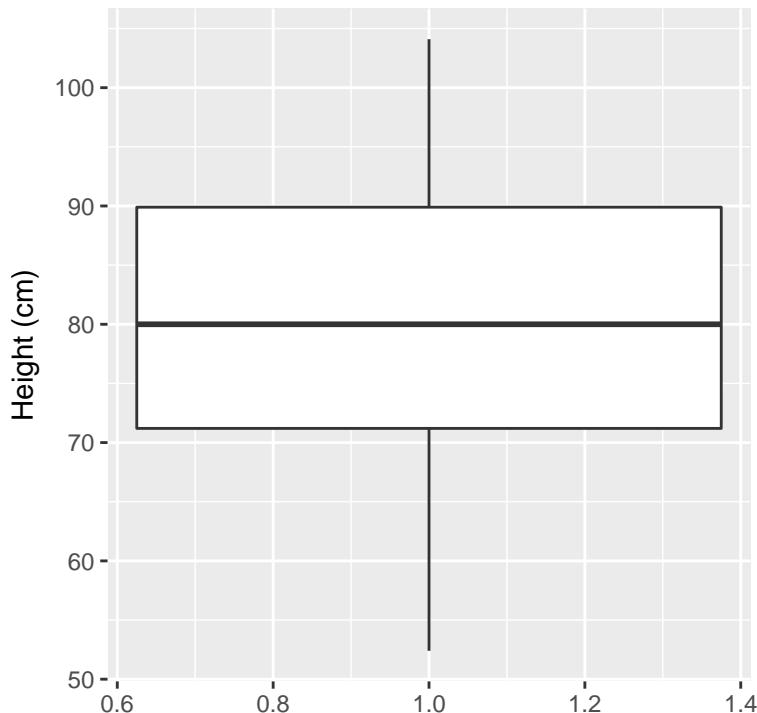
```
##
##   Male Female
##   107    93
```

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] 0.9571535
```

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



**Figure 3.11:** Example of creating a boxplot. The example shows the distribution of height data for children in the nepali dataset.

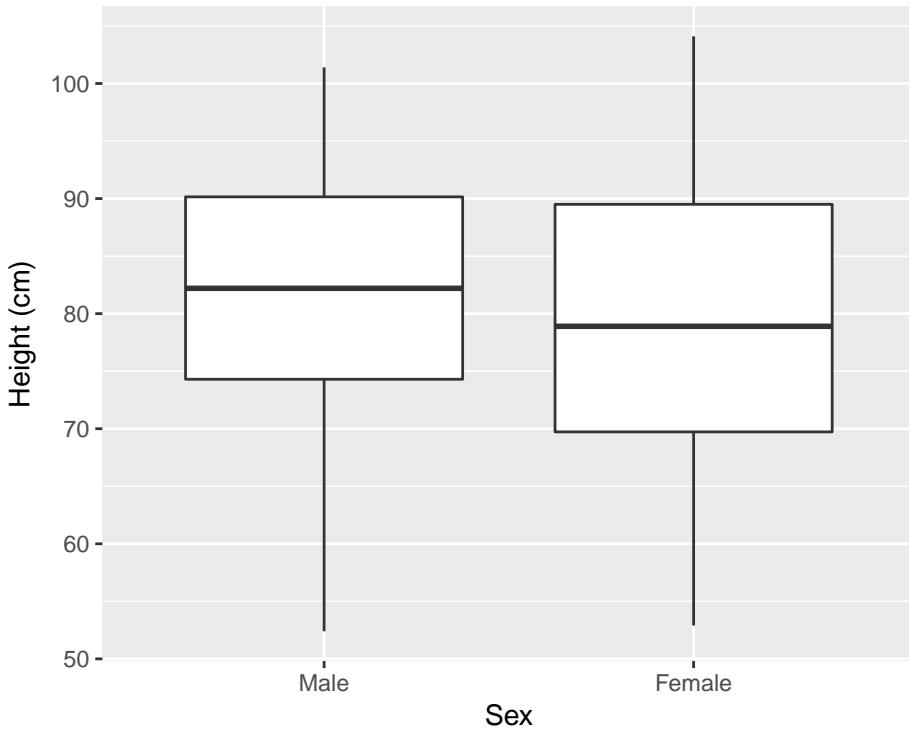
```
##           wt          ht          age
## wt  1.0000000 0.9571535 0.8931195
## ht  0.9571535 1.0000000 0.9287129
## age 0.8931195 0.9287129 1.0000000
```

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    7.90 10.10 10.18 12.40 16.70    15
```

However, if you submit a factor vector, like the `sex` column in `nepali`, the



**Figure 3.12:** Example of creating separate boxplots, divided by a categorical grouping variable in the data.

`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)
```

```
##   Male Female
##     107     93
```

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.

### 3.3.2 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:

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

##   mean_wt
## 1 10.18432
```

There are also some special functions that you can use with `summarize`. 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          200    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
##   <fctr>     <dbl>      <int>    <fctr>
## 1   Male 10.497980        107 120011
## 2 Female  9.823256        93 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.

## 3.4 Logical vectors

Last week, you learned a lot about logical statements and how to use them with the `filter` function. 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.

When you run a logical statement on a vector, you create a logical vector the same length as the original vector:

```
is_male <- nepali$sex == "Male"
length(nepali$sex)
```

```
## [1] 200
```

```
length(is_male)
```

```
## [1] 200
```

The logical vector (`is_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(is_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*, `!`:

```
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 with indexing to pull out just the rows of a dataframe where `is_male` is TRUE:

```
head(nepali[is_male, ])

##      id sex wt ht age
## 1 120011 Male 12.8 91.2 41
## 5 120023 Male 14.2 99.4 49
## 6 120031 Male 13.9 96.4 46
## 7 120051 Male  8.3 69.5  8
## 9 120053 Male 15.8 96.0 54
## 11 120062 Male 12.1 89.9 57
```

Or, with `!`, just the rows where `is_male` is FALSE:

```
head(nepali[!is_male, ])

##      id sex wt ht age
## 2 120012 Female 14.9 103.9 57
## 3 120021 Female  7.7  70.1  8
## 4 120022 Female 12.1  86.4 35
## 8 120052 Female 11.8  83.6 32
## 10 120061 Female  8.7  78.5 26
## 15 120082 Female 11.2  79.8 36
```

For these cases, the length of the logical vector and the number of rows in the dataframe will match.

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. 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` dataframe, then group by the new `is_male` column and summarize, using the `n` function to count the number of observations in each group:

```
nepali %>%
  mutate(is_male = sex == "Male") %>%
  group_by(is_male) %>%
  summarize(n_children = n())
```

```
## # A tibble: 2 x 2
##   is_male n_children
##   <lgl>     <int>
## 1 FALSE       93
## 2 TRUE        107
```

## 3.5 Regression models

### 3.5.1 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 dataframe as an independent variables (e.g., <code>y ~ . - x1</code> ); usually used in conjunction with <code>.</code> or <code>1</code>

### 3.5.2 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 dataframe from which to get the vectors.

```
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  
##      -8.6948        0.2351
```

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:

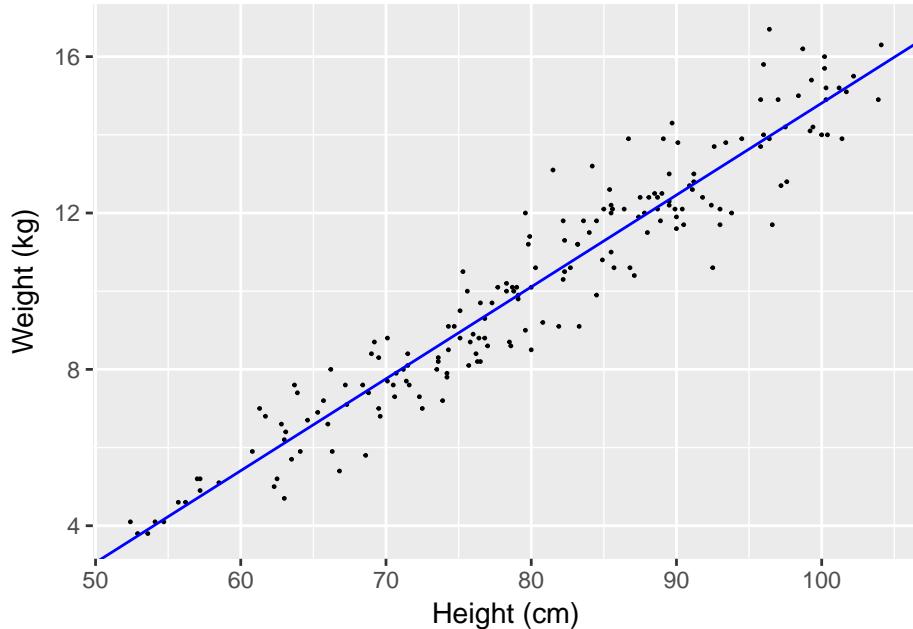
```
coefficients(mod_a)
```

```
## (Intercept)          ht
## -8.694768    0.235050
```

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  $\beta_1$ , 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 3.13).

```
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 3.13:** 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           5           6
## 0.05820415 -0.82693141 -0.08223993  0.48644436 -0.46920621 -0.06405608
```

The result of a `residuals` call is a vector with one element for each of the non-missing observations (rows) in the dataframe you used to fit the model. Each value gives the difference between the model fitted value and the observed value for each of these observations, in the same order the observations show up in the dataframe. The residuals are in the same order as the observations in the original dataframe.



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
## -2.44736 -0.55708  0.01925  0.49941  2.73594
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -8.694768  0.427398 -20.34  <2e-16 ***
## ht          0.235050  0.005257  44.71  <2e-16 ***
## ---
##
```

```
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9017 on 183 degrees of freedom
##   (15 observations deleted due to missingness)
## Multiple R-squared:  0.9161, Adjusted R-squared:  0.9157
## F-statistic: 1999 on 1 and 183 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"
```

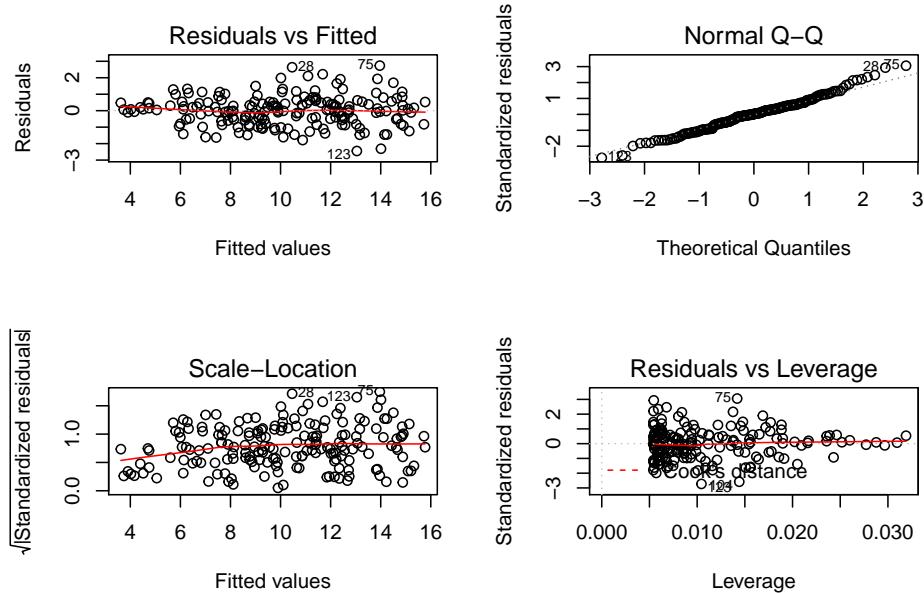
You can use the `$` operator to pull out any element of the list. For example, to pull out the table with information on the estimated model coefficients, you can run:

```
summary(mod_a)$coefficients
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -8.694768 0.427397843 -20.34350 7.424640e-49
## ht           0.235050 0.005256822  44.71334 1.962647e-100
```

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 3.14):

```
plot(mod_a)
```



**Figure 3.14:** 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)	10.497980	0.3110957	33.745177	1.704550e-80

```
## sexFemale -0.674724 0.4562792 -1.478752 1.409257e-01
```

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 dataframe 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 10.5, while the estimated mean weight of females is  $10.5 + -0.7 = 9.8$ .

If you would prefer that a different level of the factor be the baseline (for example, "Female" rather than "Male" for the previous regression), you can do that by using the `levels` argument in the `factor` function to reset factor levels. For example:

```
nepali_reset <- nepali %>%
  mutate(sex = factor(sex, levels = c("Female", "Male")))
mod_b_reset <- lm(wt ~ sex, data = nepali_reset)
summary(mod_b_reset)$coef
```

	Estimate	Std. Error	t value	Pr(> t )
## (Intercept)	9.823256	0.3337816	29.430189	2.626719e-71
## sexMale	0.674724	0.4562792	1.478752	1.409257e-01

Now, (`Intercept`) gives the estimated mean weight for females, while the second estimated coefficient gives the estimated mean difference for males compared to the expected value for females.

### 3.5.3 Generalized linear models (GLMs)

You can fit a variety of models, including linear models, logistic models, and Poisson models, using generalized linear models (GLMs).

For linear models, the only difference between `lm` and `glm` are the mechanics of how they estimate the model coefficients (`lm` uses least squares while `glm` uses maximum likelihood). You will (almost always) get exactly the same estimated coefficients regardless of whether you use `glm` or `lm` to fit a linear regression.

For example, here is the code to fit a linear regression model for weight regressed on height from the `nepali` dataset:

```
mod_c <- glm(wt ~ ht, data = nepali)
```

This call fits the same regression model I fit earlier with the `lm` function and saved as `mod_a`. You can see that the two methods give exactly the same coefficient estimates:

```
coef(mod_c)
```

```
## (Intercept)          ht
## -8.694768     0.235050
```

```
coef(mod_a)
```

```
## (Intercept)          ht
## -8.694768     0.235050
```

Unlike the `lm` function, however, the `glm` function also allows you to fit other model types, including logistic and Poisson models. You can specify the model type using the `family` argument to the `glm` call:

Model type	'family' argument
Linear	'family = gaussian(link = 'identity')'
Logistic	'family = binomial(link = 'logit')'
Poisson	'family = poisson(link = 'log')'

For example, say we wanted to fit a logistic regression for the `nepali` data of whether the probability of a child weighing more than 13 kg is associated with the child's sex.

First, create a binary variable in the `nepali` dataset, `wt_over_13`, that is TRUE if a child weighed more than 13 kilograms and FALSE otherwise. You can use the `mutate` function from `dplyr` to add this new column (which, as a note, is a logical vector):

```
nepali <- nepali %>%
  mutate(wt_over_13 = wt > 13)
head(nepali)
```

```
##      id   sex   wt    ht age wt_over_13
## 1 120011 Male 12.8 91.2  41      FALSE
## 2 120012 Female 14.9 103.9  57       TRUE
## 3 120021 Female  7.7 70.1   8      FALSE
## 4 120022 Female 12.1  86.4  35      FALSE
## 5 120023 Male 14.2 99.4  49       TRUE
## 6 120031 Male 13.9  96.4  46       TRUE
```

Now you can fit a logistic regression of `wt_over_13` regressed on `sex`, using a logistic model:

```
mod_d <- glm(wt_over_13 ~ sex, data = nepali,
              family = binomial(link = "logit"))
```

Elements of a GLM can be pulled out in the same way that we looked at elements from the linear model fit with `lm`. For example, to see a table of estimated model coefficients, you can run:

```
summary(mod_d)$coef
```

```
##             Estimate Std. Error   z value   Pr(>|z|)  
## (Intercept) -1.3121864  0.2458445 -5.337465 9.425485e-08  
## sexFemale    -0.4133237  0.3886659 -1.063442 2.875815e-01
```

Because this model was a logistic model, fit with a log link, here the model coefficient estimate for `sexFemale` gives an estimate of the **log odds** of weight higher than 13 kg associated with females versus males. The p-value for this estimate ( $\text{Pr}(>|z|) = 0.29$ ) isn't very small, suggesting that the difference between male and female children in the odds of weighing more than 13 kg is not statistically significant.

### 3.5.4 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

## 3.6 In-course exercise

### 3.6.1 Loading data from an R package

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`, `colnames`, `str`, and `summary` to check out the data a bit. 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?

### 3.6.1.1 Example R code:

Load the `faraway` package using `load()` 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",...: 1 16 9 9 5 32 11 11 18 20 ...
## $ Position: Factor w/ 4 levels "Defender","Forward",...: 4 4 1 4 2 2 1 2 4 1 ...
## $ 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"
## [7] "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.   : 0.0000   Min.   : 0.0000
## 1st Qu.:29.00   1st Qu.: 1.0000   1st Qu.: 0.0000
## Median :61.00   Median : 3.0000   Median : 0.0000
## Mean   :84.52   Mean   : 4.192   Mean   : 0.6672
## 3rd Qu.:115.50  3rd Qu.: 6.0000  3rd Qu.: 0.0000
## Max.   :563.00  Max.   :34.0000  Max.   :20.0000
##
```

### 3.6.2 Basic plots of the data

Use some basic plots to check out this data. Try the following:

- Plot histograms of all the numeric variables (`Time`, `Shot`, `Passes`, `Tackles`, `Saves`)
- Plot scatterplots of different combinations of numeric variables (e.g., `Time` vs. `Shots`). Try doing this using the `geom_point()` geom from `ggplot2`. Also try doing it using the `ggpairs()` function from the `GGally` package,

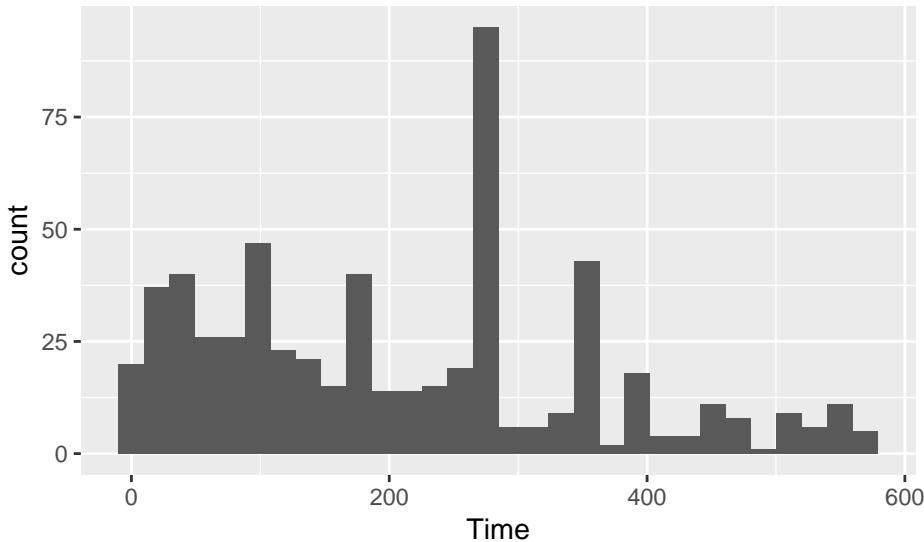
to plot several of these at the same time. Try using different constant or mapped values with the `color` aesthetic.

- Create boxplots of `Time`, `Shots`, `Passes` and `Saves` by position.
- Go online and find out which teams were the top four teams in this World Cup (i.e., first through fourth places). Create a `top_teams` subset with just these teams. Plot boxplots of `Shots` and `Saves` by team for just these teams.
- Did you notice any interesting features of the data when you did any of the graphs in this section?

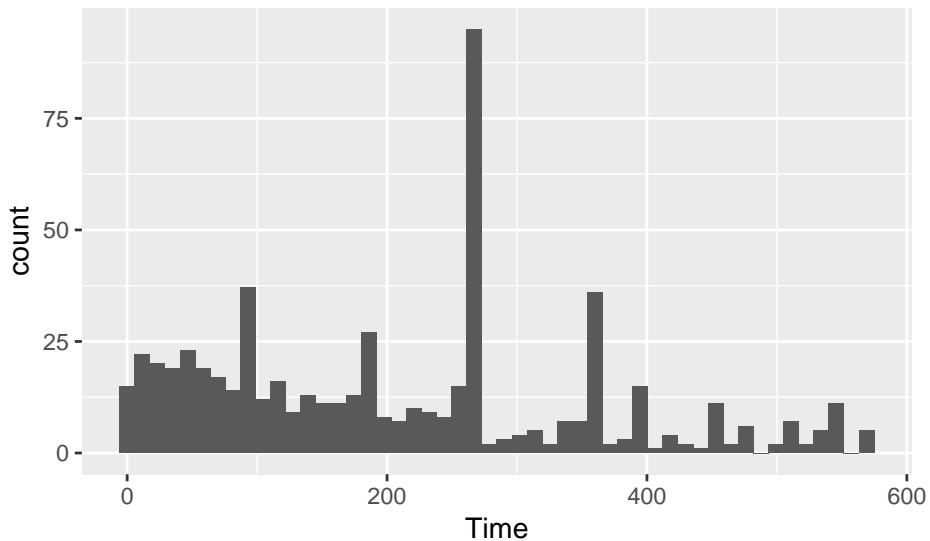
### 3.6.2.1 Example R code:

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.

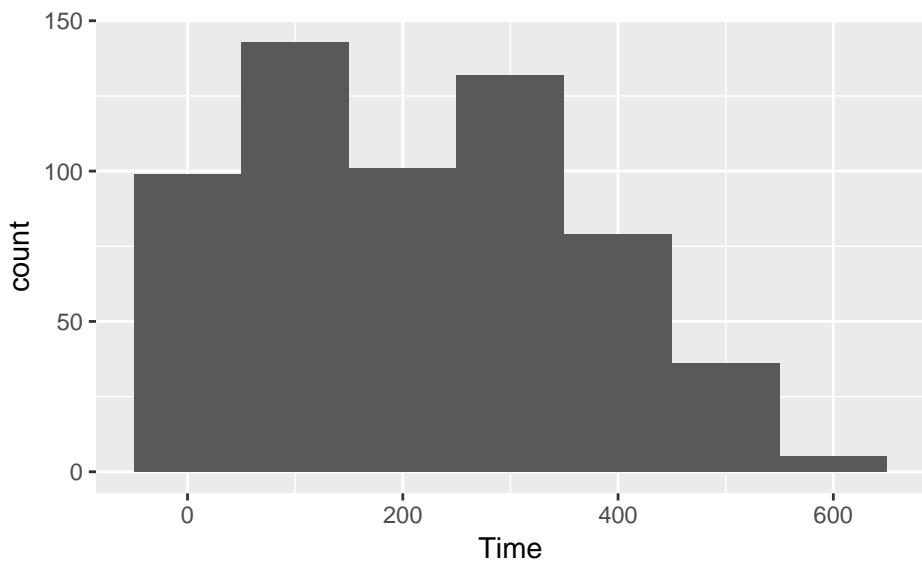
```
library(ggplot2)
ggplot(worldcup, aes(x = Time)) +
  geom_histogram()
```



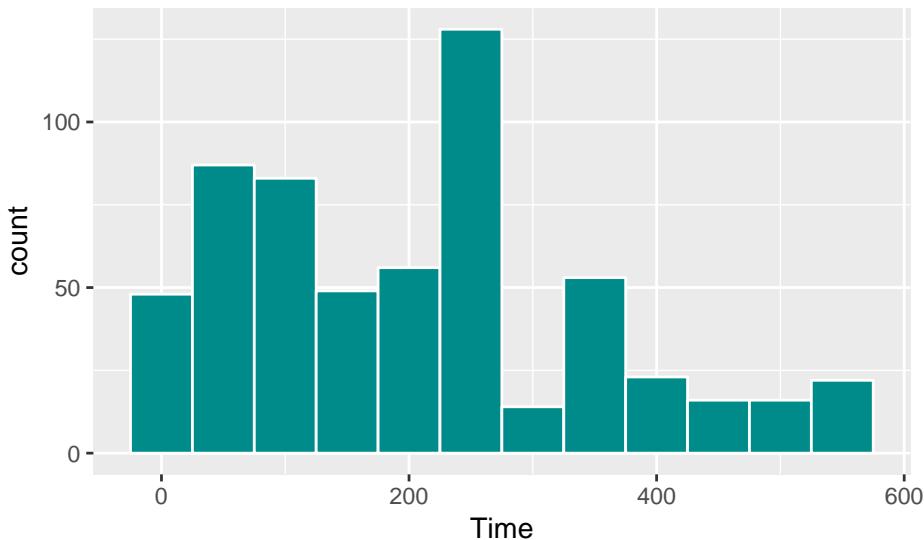
```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram(bins = 50)
```



```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram(binwidth = 100)
```

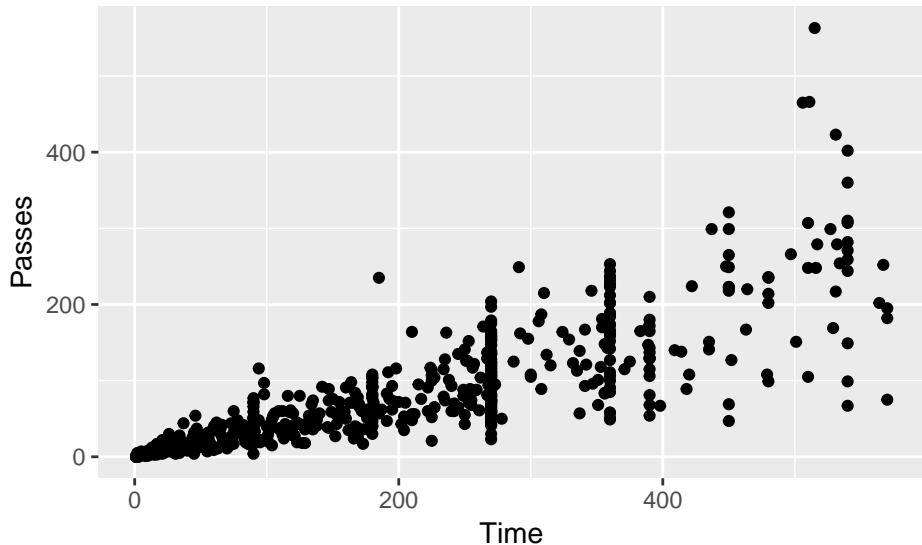


```
ggplot(worldcup, aes(x = Time)) +  
  geom_histogram(binwidth = 50, color = "white", fill = "cyan4")
```

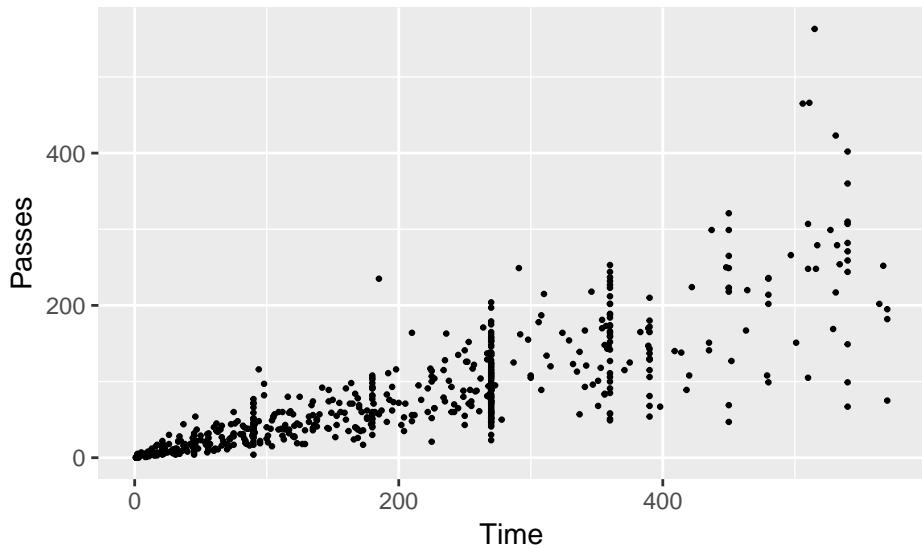


Create a scatterplot of `Time` versus `Passes`. To change the size of the points, use the `size` argument (use a number lower than 1 for smaller points, higher than 1 for larger points). Try changing the color and transparency of the points using the aesthetics `color` and `alpha`. Try using color to show each player's position by mapping `Position` to the `color` aesthetic.

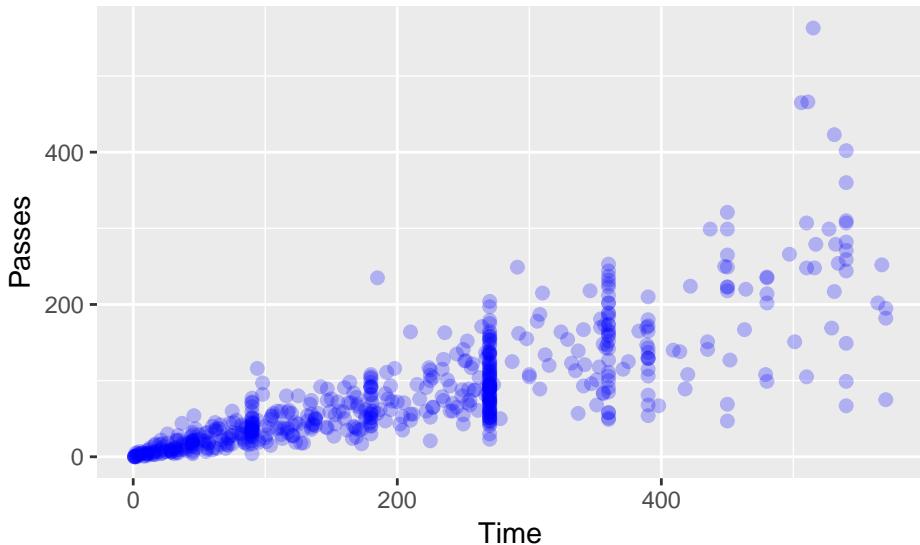
```
ggplot(worldcup, aes(x = Time, y = Passes)) +  
  geom_point()
```



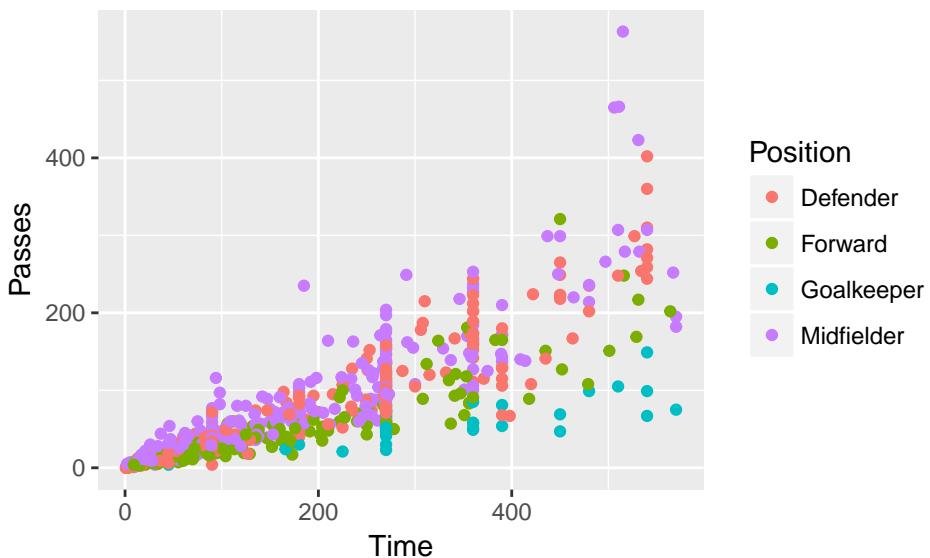
```
ggplot(worldcup, aes(x = Time, y = Passes)) +  
  geom_point(size = 0.5)
```



```
ggplot(worldcup, aes(x = Time, y = Passes)) +  
  geom_point(size = 2, color = "blue", alpha = 0.25)
```



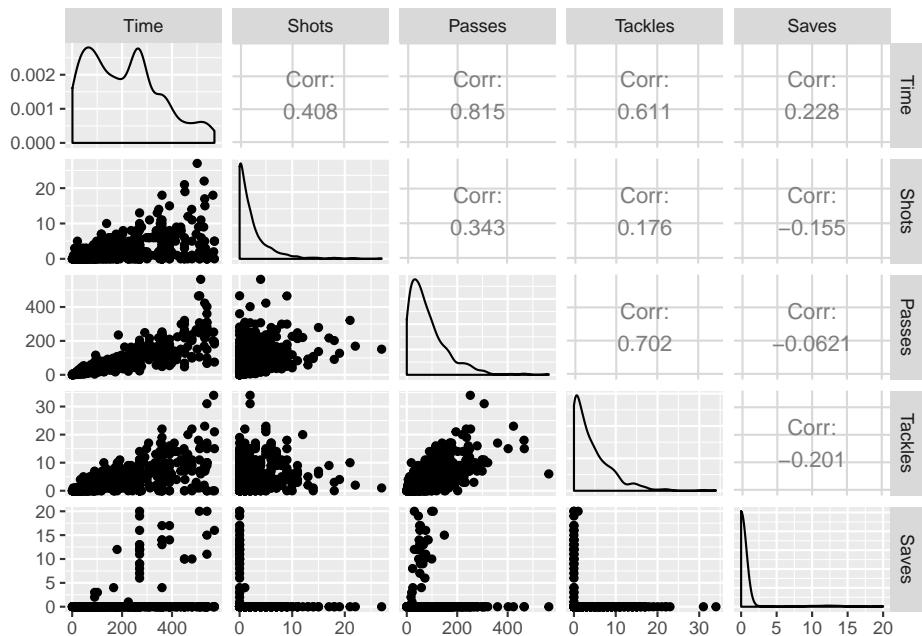
```
ggplot(worldcup, aes(x = Time, y = Passes, color = Position)) +  
  geom_point()
```



Use the `ggpairs` function from the `GGally` package to plot scatterplots of all combinations of several numeric variables.

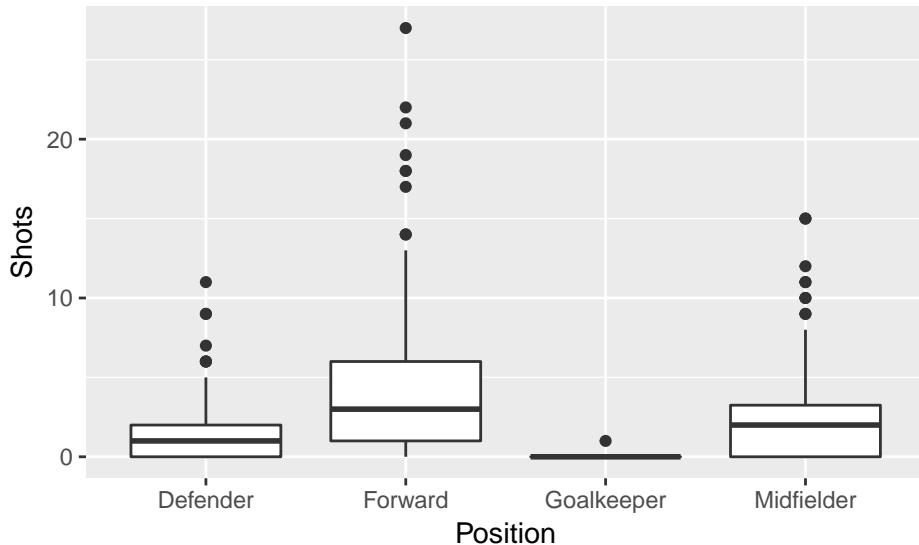
```
library(GGally)
```

```
library(dplyr)
ggpairs(select(worldcup, Time, Shots, Passes, Tackles, Saves))
```



To create a boxplot of `Shots` by `Position`, you can use `geom_boxplot`:

```
ggplot(worldcup, aes(x = Position, y = Shots)) +
  geom_boxplot()
```



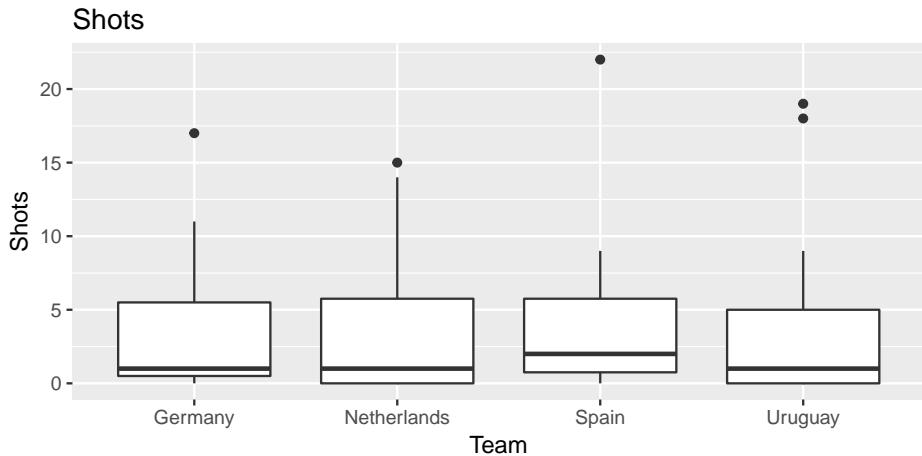
The top four teams in this World Cup were Spain, the Netherlands, Germany, and Uruguay. Create a subset with just the data for these four teams:

```
top_teams <- worldcup %>%
  filter(Team %in% c("Spain", "Netherlands", "Germany", "Uruguay")) %>%
  mutate(Team = factor(Team))
```

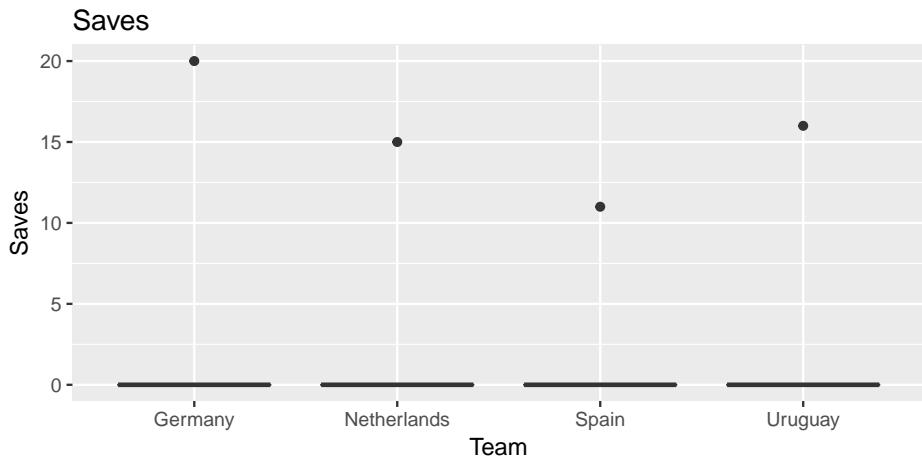
This dataset will still have all the levels saved for the `Team` factor, even though it isn't using them all. You can re-set this by resetting `Team` as a factor, which is what I've done with the `mutate` line. When R creates a factor from a vector, its default is to only use as levels the values that show up in the vector.

Now, you can plot the boxplots, mapping `Team` to the `x` aesthetic and `Shots` or `Saves` to the `y` aesthetic:

```
ggplot(top_teams, aes(x = Team, y = Shots)) +
  geom_boxplot() +
  ggtitle("Shots")
```



```
ggplot(top_teams, aes(x = Team, y = Saves)) +
  geom_boxplot() +
  ggtitle("Saves")
```



### 3.6.2.2 If you have extra time:

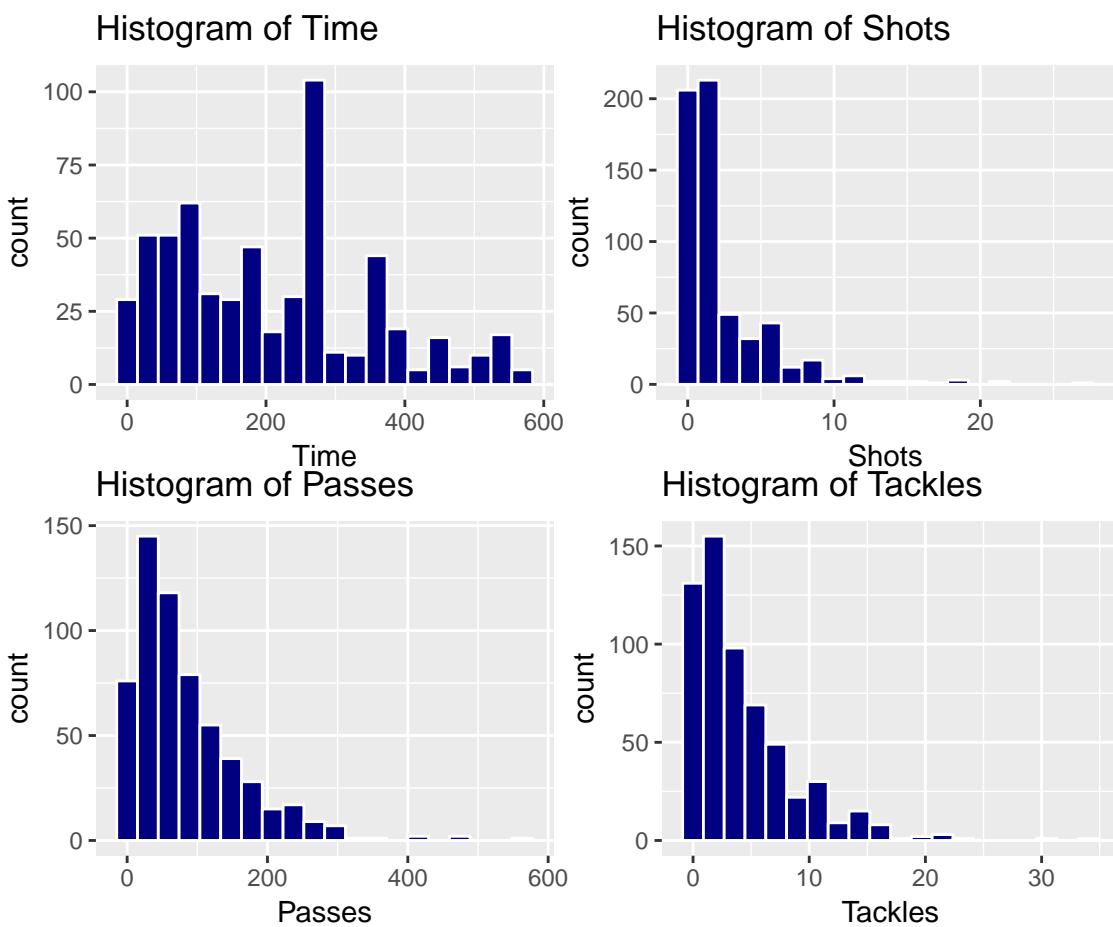
If you wanted to do the same plot for several different variables, you could loop through your code (we'll be covering more about loops in a few weeks). For example, you could create histograms for all of the numeric variables (if you do this in RStudio, you'll need to use the arrows on the plot window to move through and see all the different plots once you've created them):

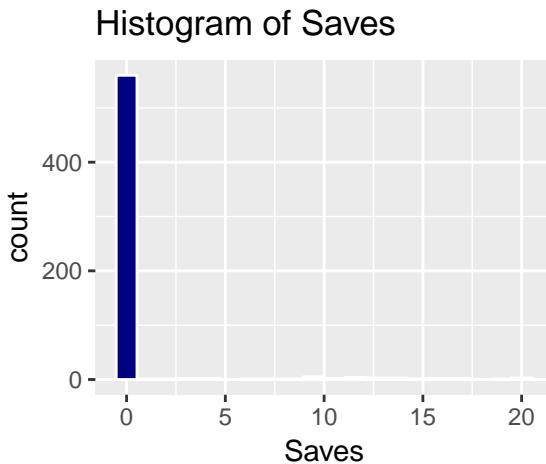
```

## Create an object with the column names for all of the numeric variables
my_vars <- colnames(worldcup)[3:7]

## Loop through all of those variables. Print out a histogram with the
## variable, and have it print on the plot, as the main title, the
## column name for that variable
for(var in my_vars){
  worldcup$to_plot <- worldcup[, var]
  a <- ggplot(worldcup, aes(x = to_plot)) +
    geom_histogram(bins = 20, color = "white", fill = "navy") +
    xlab(var) +
    ggtitle(paste("Histogram of", var))
  plot(a)
}

```





A few things to note in this example:

- To map an element of the data to an aesthetic, it's easiest if that element is saved in a column in the data frame. Within this loop, I'm making an extra column called `to_plot`, where I'm copying the column of the variable I want to plot each time the loop runs. That way, I can always use `x = to_plot` in the aesthetic mapping for the `ggplot` object.
- If you run code to create a `ggplot` object within a loop, it won't automatically print. Instead, you need to use `print` to get the object to print out. One way to do that is to save the final `ggplot` object as an R object (here I'm saving it to `a`) and then use the `print` function to print that object.
- Next week, we'll talk some about facetting, which can create multiple plots by variable like this in a lot less code. However, it's useful at this point to start thinking about how to extend code to use in loops, to save yourself time when you need to repeat something similar many times.

### 3.6.3 Exploring the data using simple statistics and logical statements

Next, try checking out the data using some basic commands for simple statistics, like `mean()`, `range()`, `max()`, and `min()`. Use these, along with some logical statements, to help you answer the following questions:

- What is the range of time that players spent in the game? Who played the most World Cup time in this World Cup? For the minimum of the range of `Time`, how many players played this amount of time?
- What is the mean number of saves that players made? What is the mean number of saves just among the goalkeepers? How many of the players are goalkeepers? Did any non-goalkeeper make a save?

### 3.6.3.1 Example R code:

Use `range()` to find out the range of time these players played in the World Cup.

```
range(worldcup$Time)
```

```
## [1] 1 570
```

To figure out who played the most time, you need to subset out the rows of the dataset where the `Time` variable equals the maximum of the `Time` variable for the whole dataset. There are a few ways to do that. Here I'm showing two: (1) using logic within the “square-bracket indexing”, to pull out just rows where it is TRUE that the `Time` for that row equals `max(worldcup$Time)` and (2) using `filter` from the `dplyr` package to filter down to rows where where it is TRUE that the `Time` for that row equals `max(Time)` for the whole dataset.

```
max(worldcup$Time)
```

```
## [1] 570
```

```
head(worldcup$Time == max(worldcup$Time))
```

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

```
worldcup[worldcup$Time == max(worldcup$Time), ]
```

```
##           Team Position Time Shots Passes Tackles Saves to_plot
## 1 Arevalo Rios Uruguay Midfielder 570      5    195      21      0      0
## 2 Maxi Pereira Uruguay Midfielder 570      5    182      15      0      0
## 3 Muslera     Uruguay Goalkeeper 570      0     75       0     16     16
```

```
worldcup %>%
  filter(Time == max(Time))
```

```
##           Team Position Time Shots Passes Tackles Saves to_plot
## 1 Uruguay Midfielder 570      5    195      21      0      0
## 2 Uruguay Midfielder 570      5    182      15      0      0
## 3 Uruguay Goalkeeper 570      0     75       0     16     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, use `mutate` to move those names from the rownames of the data into a column in the dataframe:

```
worldcup %>%
  mutate(Name = rownames(worldcup)) %>%
  filter(Time == max(Time))

##      Team  Position Time Shots Passes Tackles Saves to_plot      Name
## 1 Uruguay Midfielder  570     5    195      21      0       0 Arevalo Rios
## 2 Uruguay Midfielder  570     5    182      15      0       0 Maxi Pereira
## 3 Uruguay Goalkeeper  570     0     75      0      16      16 Muslera
```

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
```

For the next parts of the question, it will be convenient to have a logical vector for whether each player is a goalkeeper, so here's how you would create that:

```
goalie <- worldcup$Position == "Goalkeeper"
```

This new object, `goalie`, is a vector the same length as `worldcup$Position`. Each element of `goalie` says whether it is TRUE or FALSE that `worldcup$Position` is equal to "Goalkeeper" at that spot on the `worldcup$Position` vector.

```
head(goalie)

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

The `summary()` function will count up the total number of times that `goalie` is TRUE and FALSE.

```
summary(goalie)
```

```
##      Mode   FALSE    TRUE
## logical    559     36
```

There are a few ways to use this vector to figure out how many players were goalkeepers. First, you could use `summary` (which I just showed) or `table`, and just read how many times this vector has the value `TRUE`. Second, since R saves logical vectors with `TRUE` as 1 and `FALSE` as 0, you could just the `sum` function to add up the vector to find out how often it's `TRUE` (`sum` adds up every value in the vector).

```
table(goalie)
```

```
## goalie
## FALSE  TRUE
## 559    36
```

```
sum(goalie)
```

```
## [1] 36
```

You could also answer this question by using `summarize` from `dplyr`. You need to `group_by` player position and then you can use the `n` function in `summarize` to count up the total number of observations in each group:

```
worldcup %>%
  group_by(Position) %>%
  summarize(n_players = n())
```

```
## # A tibble: 4 x 2
##       Position n_players
##       <fctr>     <int>
## 1 Defender      188
## 2 Forward       143
## 3 Goalkeeper     36
## 4 Midfielder    228
```

Now, you can answer the questions about mean saves for goalies and max saves for non-goalies. First, try doing that using the `goalie` logical vector you created. If you put `goalie` in the square bracket indexing for the dataframe as the rows value (i.e., the index before the comma), R will subset out just the rows where `goalie` is equal to `TRUE`. If you put `!goalie` in the square bracket indexing as the rows value, R will just subset out the rows where `goalie` is equal to `FALSE`. You can use this index subsetting to figure out the mean number of saves per goalie and also whether any non-goalie made a save (by checking the maximum value or range of saves for non-goalies).

```

head(worldcup[goalie, ])

##          Team Position Time Shots Passes Tackles Saves to_plot
## Barry    Ivory Coast Goalkeeper 270     0    23      0     8     8
## Benaglio Switzerland Goalkeeper 270     0    75      0    11    11
## Bravo    Chile Goalkeeper    360     0    58      0     4     4
## Buffon   Italy Goalkeeper    45      0     4      0     0     0
## Casillas Spain Goalkeeper   540     0    67      0    11    11
## Chaouchi Algeria Goalkeeper  90      0    17      0     2     2

mean(worldcup[goalie, "Saves"])

## [1] 11.02778

range(worldcup[!goalie, "Saves"])

## [1] 0 0

```

You could also answer this question using a `dplyr` pipe chain to summarize the data after grouping it by position:

```

worldcup %>%
  group_by(Position) %>%
  summarise(number_players = n(),
            mean_saves = mean(Saves),
            max_saves = max(Saves))

## # A tibble: 4 x 4
##       Position number_players mean_saves max_saves
##       <fctr>           <int>      <dbl>     <dbl>
## 1   Defender            188     0.00000      0
## 2   Forward             143     0.00000      0
## 3 Goalkeeper            36     11.02778     20
## 4 Midfielder           228     0.00000      0

```

### 3.6.4 Using regression models to explore data

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 dataframe to something shorter, like `chic`. 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`) on temperature (`temp`). (Bonus points: Notice anything that seems unusual about these two variables in this dataset? You can find out with `summary`, but it helps if you know a bit about what dewpoint temperature measures.) Save this model as the object `mod_1`.
- Based on this regression, does there seem to be a relationship between temperature and dewpoint temperature in Chicago? (Hint: Try using `summary()` on the model object to get more information about the model you fit.) 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.
- Use `plot()` on the model object to check if some of the assumptions for the regression model seem appropriate.
- Try fitting the regression as a GLM, using `glm()`. 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, and then run `anova()` on that model, and R will compare it 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` variable to do this—just pull out the subset where the month is 6, 7, or 8, for June, July, and 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?

#### 3.6.4.1 Example R code:

Install and load the `dlnm` package and then load the `chicagoNMMAPS` data. Change the name of the dataframe 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 `summary()` to check out a bit more about the model you fit.

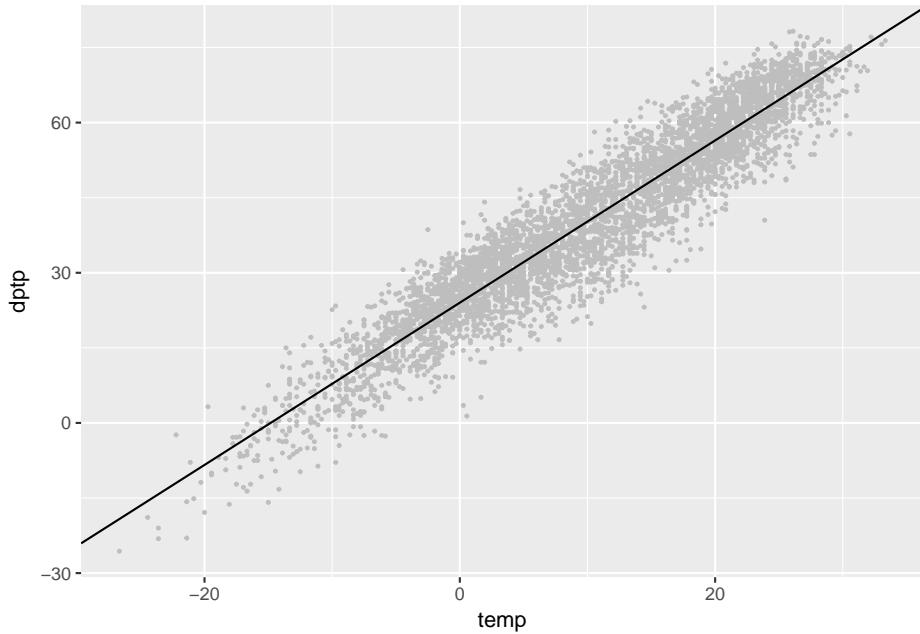
```
summary(mod_1)
```

```
##
## Call:
## lm(formula = dptp ~ temp, data = chic)
##
## Residuals:
##    Min     1Q   Median     3Q    Max 
## -24.3093 -3.7470  0.4687  4.0738 18.6518 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 24.024869  0.112933 212.7   <2e-16 ***
## temp        1.620650  0.007631 212.4   <2e-16 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
## 
## Residual standard error: 5.899 on 5112 degrees of freedom
## Multiple R-squared:  0.8982, Adjusted R-squared:  0.8982 
## F-statistic: 4.511e+04 on 1 and 5112 DF, p-value: < 2.2e-16
```

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  $<2\text{e}-16$ . 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.

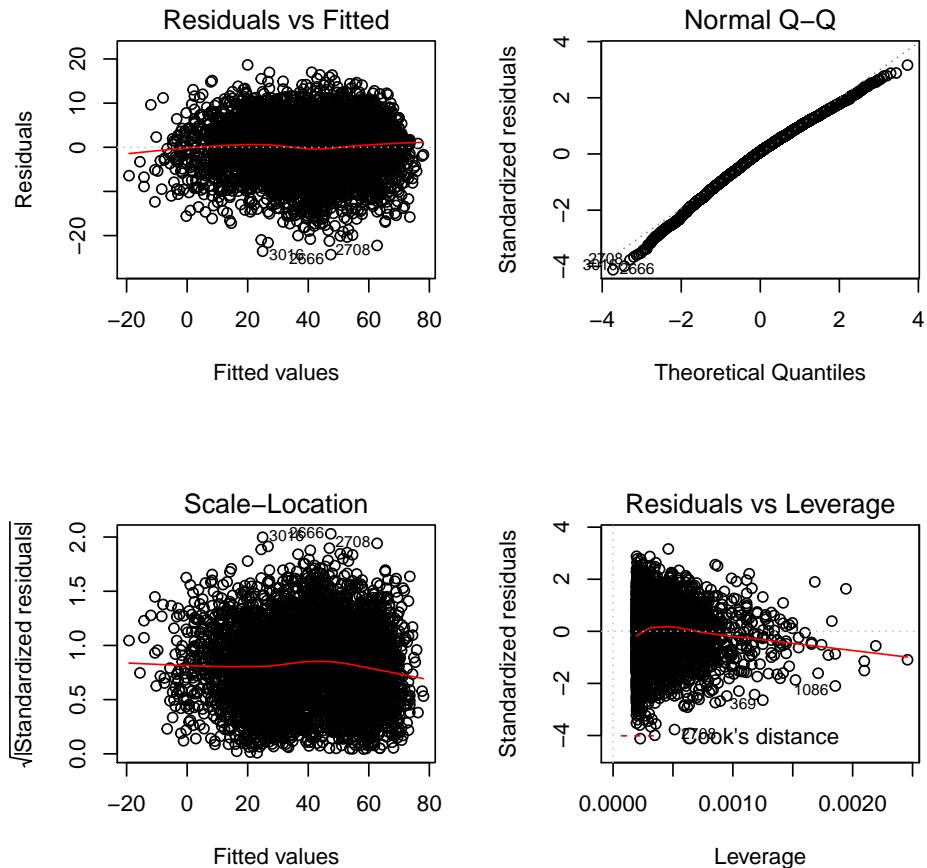
Plot these two variables and add in the regression line from the model (note: I've used the `color` option to make the color of the points gray). Use the values from `coef` with a `geom_abline` to add the regression line for the model you fit.

```
mod_coefs <- coef(mod_1)
ggplot(chic, aes(x = temp, y = dptp)) +
  geom_point(size = 0.5, col = "gray") +
  geom_abline(aes(intercept = mod_coefs[1], slope = mod_coefs[2]))
```



Plot some plots to check model assumptions for the model you fit using the `plot()` function on your model object:

```
par(mfrow = c(2, 2)) # Set to four plots per panel -- 2 rows, 2 columns
plot(mod_1)
```



```
par(mfrow = c(1, 1)) # Reset to one plot per panel
```

Try fitting the model using `glm()`. Call it `mod_1a`. Compare the coefficients for the two models. You can use the `coef()` function on an `lm` or `glm` object to pull out just the model coefficients.

```
mod_1a <- glm(dptp ~ temp, data = chic)

coef(mod_1)

## (Intercept)      temp
## 24.02487     1.62065

coef(mod_1a)

## (Intercept)      temp
```

```
##      24.02487    1.62065
```

The results from the two models are identical.

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)
summary(mod_2)
```

```
##
## Call:
## lm(formula = pm10 ~ dow, data = chic)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -39.05 -12.55 - 3.34   8.80 328.66 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 27.5217    0.7303  37.684 < 2e-16 ***
## dowMonday    6.1322    1.0340   5.931 3.22e-09 ***
## dowTuesday   6.7954    1.0269   6.617 4.05e-11 ***
## dowWednesday 8.4768    1.0262   8.261 < 2e-16 ***
## dowThursday  8.8047    1.0240   8.598 < 2e-16 ***
## dowFriday    9.4816    1.0262   9.240 < 2e-16 ***
## dowSaturday  3.6602    1.0269   3.564 0.000368 *** 
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 19.07 on 4856 degrees of freedom
## (251 observations deleted due to missingness)
## Multiple R-squared:  0.02588,    Adjusted R-squared:  0.02467 
## F-statistic:  21.5 on 6 and 4856 DF,  p-value: < 2.2e-16
```

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).

```
anova(mod_2)
```

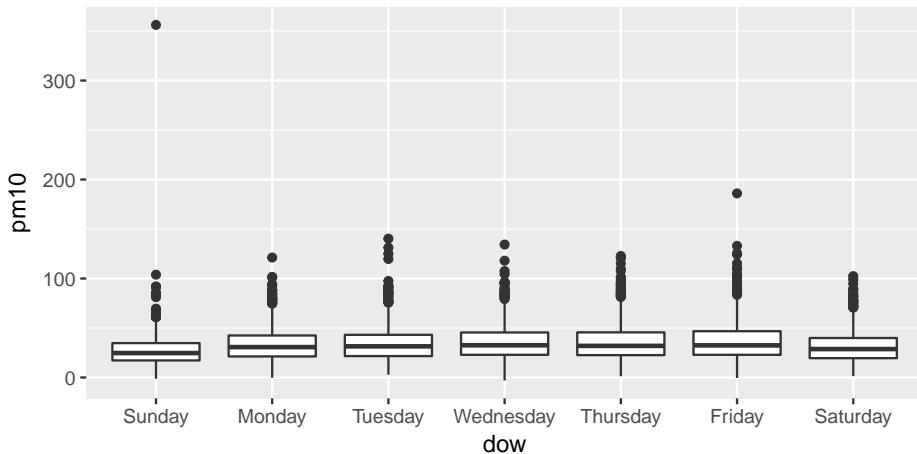
```
## Analysis of Variance Table
##
## Response: pm10
##           Df  Sum Sq Mean Sq F value    Pr(>F)    
## dow       6   46924  7820.6   21.5 < 2.2e-16 ***
## Residuals 4856 1766407   363.8
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The p-value for an ANOVA of the model 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. (Note, too, that the F value and  $\text{Pr}(>\text{F})$  for the `anova()` call are identical to the F-statistic information given in the `summary()` of the model object. This will always be true when you're using `anova()` to compare a model to a model with just an intercept.)

Use a boxplot to visually compare PM10 by day of week.

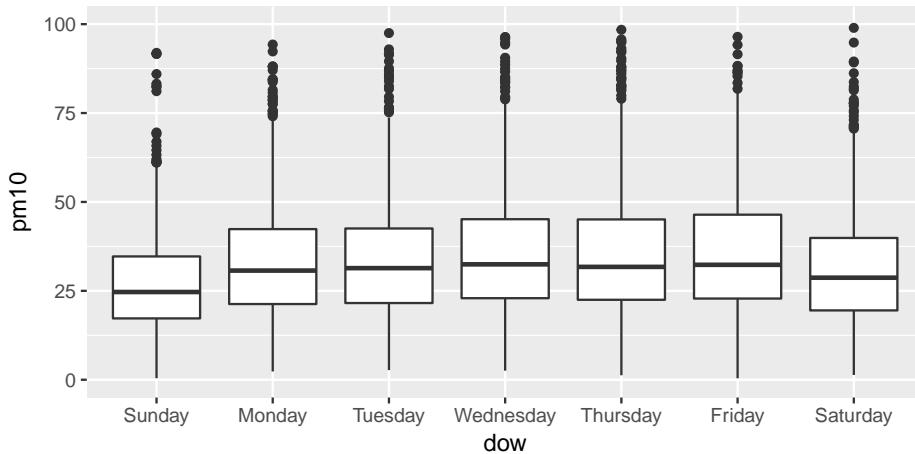
```
ggplot(chic, aes(x = dow, y = pm10)) +
  geom_boxplot()
```



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:

```
summer <- chic %>%
  filter(month %in% 6:8)
```

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"))
summary(mod_3)

##
## Call:
## glm(formula = resp ~ temp, family = poisson(link = "log"), data = summer)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -3.9755   -0.7162   -0.1807    0.6927    3.6555
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 1.910317  0.058373 32.726   <2e-16 ***
## temp        0.006137  0.002581  2.378   0.0174 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
```

```
##      Null deviance: 1499.4  on 1287  degrees of freedom
## Residual deviance: 1493.8  on 1286  degrees of freedom
## AIC: 6425.4
##
## Number of Fisher Scoring iterations: 4
```

Use the fitted model coefficient to determine the relative risk for a one degree Celcius increase in temperature. First, remember that you can use the `coef()` 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.

```
coef(mod_3)
```

```
## (Intercept)      temp
## 1.910316958 0.006136743
```

```
coef(mod_3)[2]
```

```
##      temp
## 0.006136743
```

```
exp(coef(mod_3)[2])
```

```
##      temp
## 1.006156
```

# Chapter 4

# Reporting data results #1

Download a pdf of the lecture slides covering this topic.

## 4.1 Guidelines for good plots

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

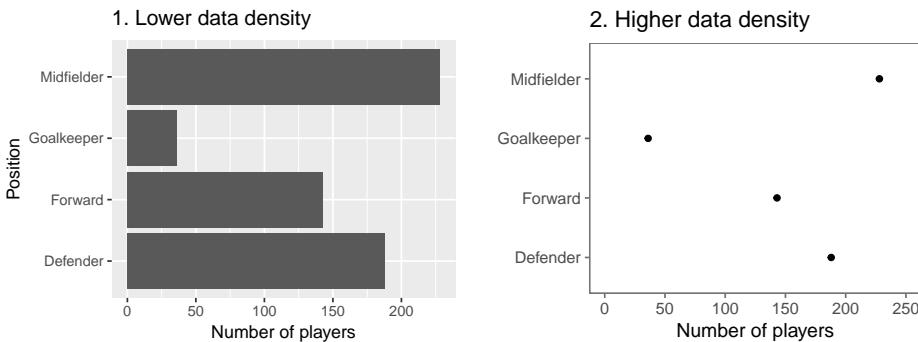
### Guideline 1: Aim for high data density.

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.

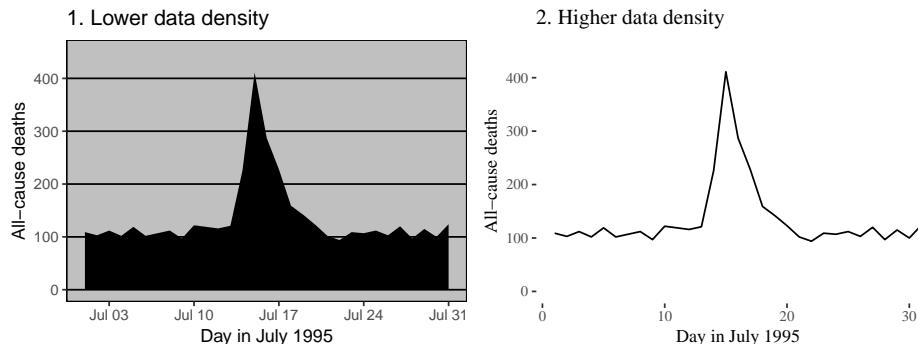
The two graphs in Figure 4.1 show the same information, but use very different amounts of ink. Each shows the number of players in each of four positions in the `worldcup` dataset. Notice how, in the plot on the right, a single dot for each category shows the same information that a whole filled bar 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` 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.

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



**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.



**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.

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.

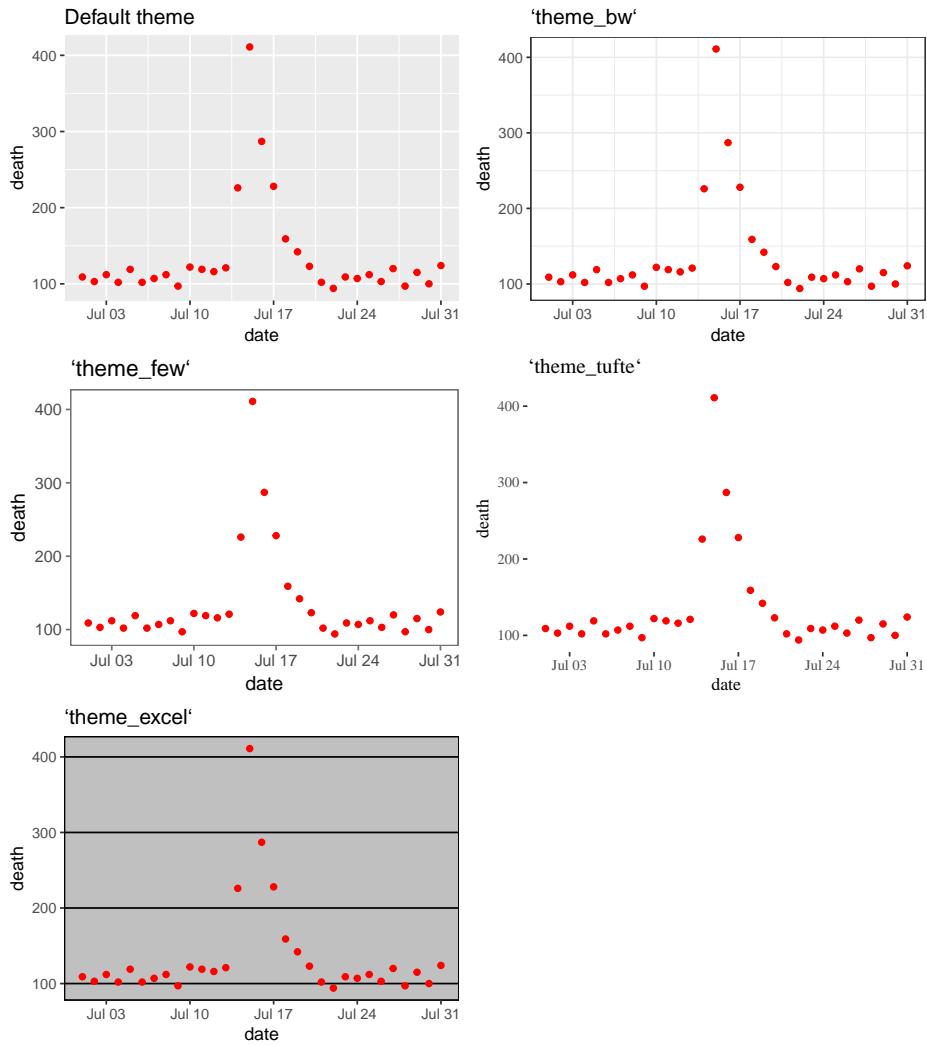
One quick way to increase data density in `ggplot2` is to change the *theme* for the plot. The theme specifies a number of the “background” elements to a plot, including elements like the plot grid, background color, and the font used for labeling. Some themes come with `ggplot2`, including:

- `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).



**Figure 4.3:** Daily mortality in Chicago, IL, in July 1995. This figure gives an example of the plot using different themes.

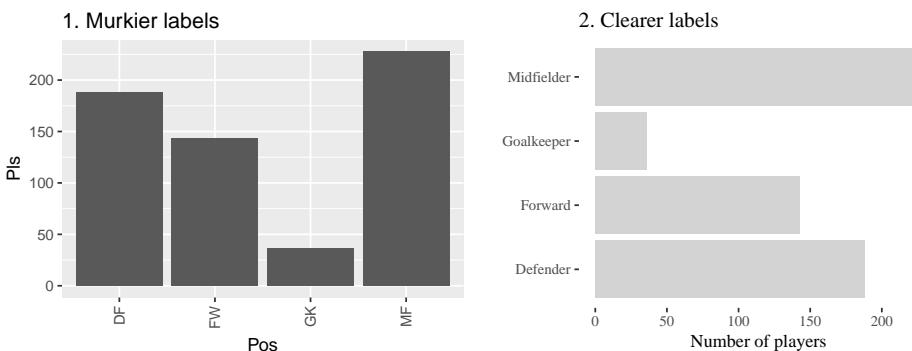
## 4.3 Meaningful labels

Guideline 2: **Use clear, meaningful labels.**

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 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.

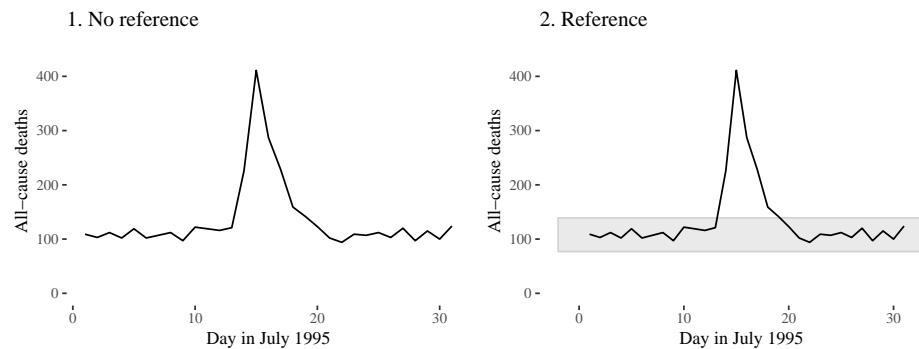
- Include units of measurement in axis titles when relevant. If units are dollars or percent, check out the `scales` package, which allows you to add labels directly to axis elements by including arguments like `labels = percent` in `scale` elements. See the helpfile for `scale_x_continuous` for some 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.

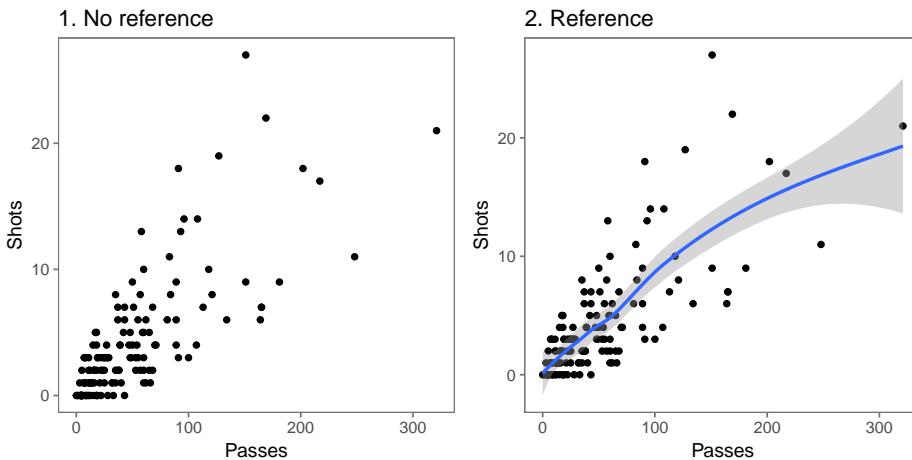
Figure 4.5 shows daily mortality during July 1995 in Chicago, IL. The graph on the right has added shading showing the range of daily death counts in July in Chicago for neighboring years (1990–1994 and 1996–2000). This added reference helps clarify for viewers how unusual the number of deaths during 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.



**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.

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"`.
- **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

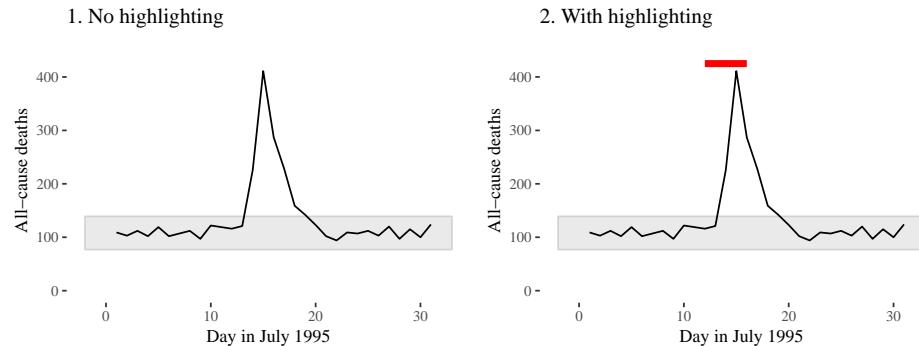
You want these references to support the main data shown in the plot, but not overwhelm it. When adding these references:

- 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

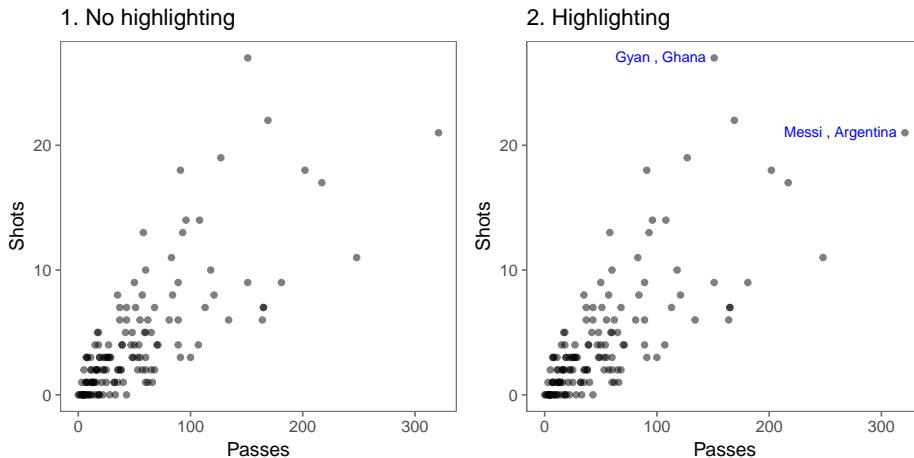
### Guideline 4: Highlight interesting aspects.

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.



**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.

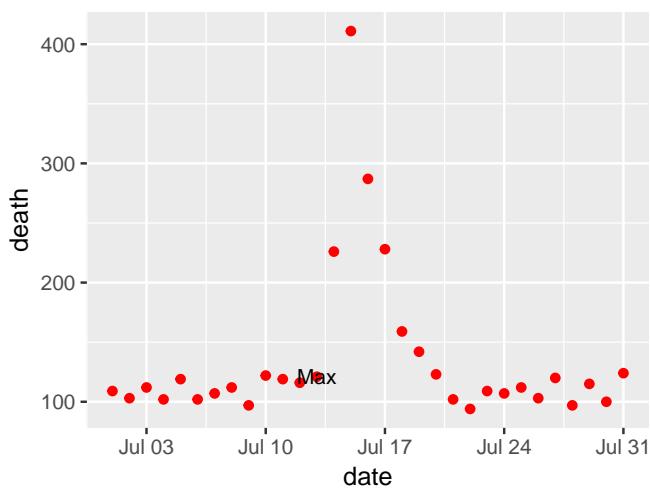


One helpful way to annotate is with text, using `geom_text()`. For this, you'll first need to create a dataframe with the hottest day in the data:

```
hottest_day <- chic_july %>%
  filter(temp == max(temp))
hottest_day[ , 1:6]

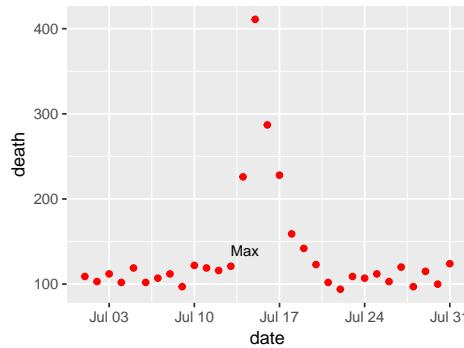
##           date time year month doy      dow
## 1 1995-07-13 3116 1995      7 194 Thursday
```

```
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)
```

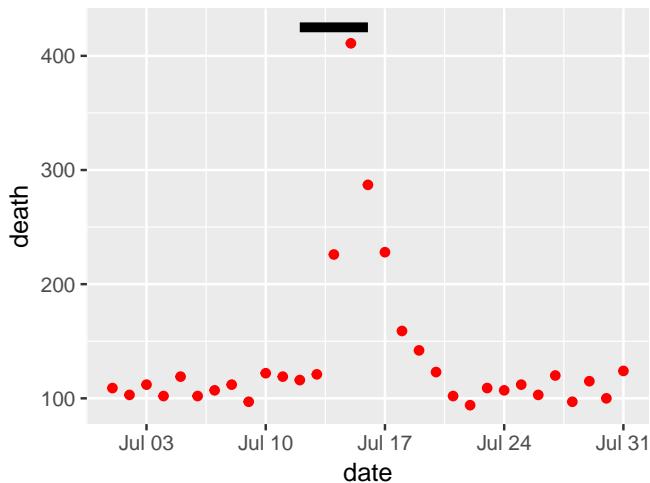


You can also use lines to highlight. For this, it is often useful to create a new dataframe with data for the reference. To add a line for the Chicago heat wave, I've added a dataframe called `hw` with the relevant date range. I'm setting the y-value to be high enough (425) to ensure the line will be 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)
```

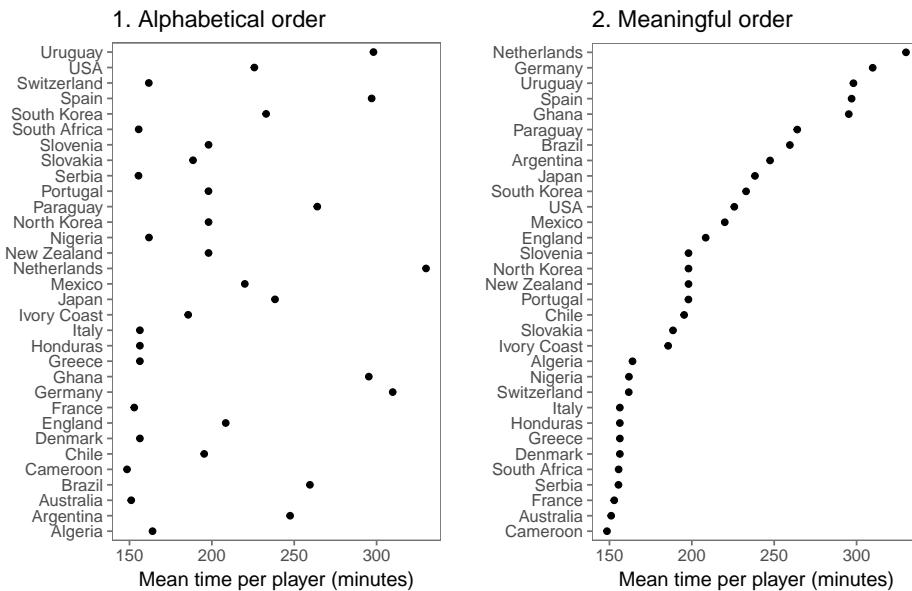
b



## 4.6 Order

Guideline 5: **Make order meaningful.**

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`

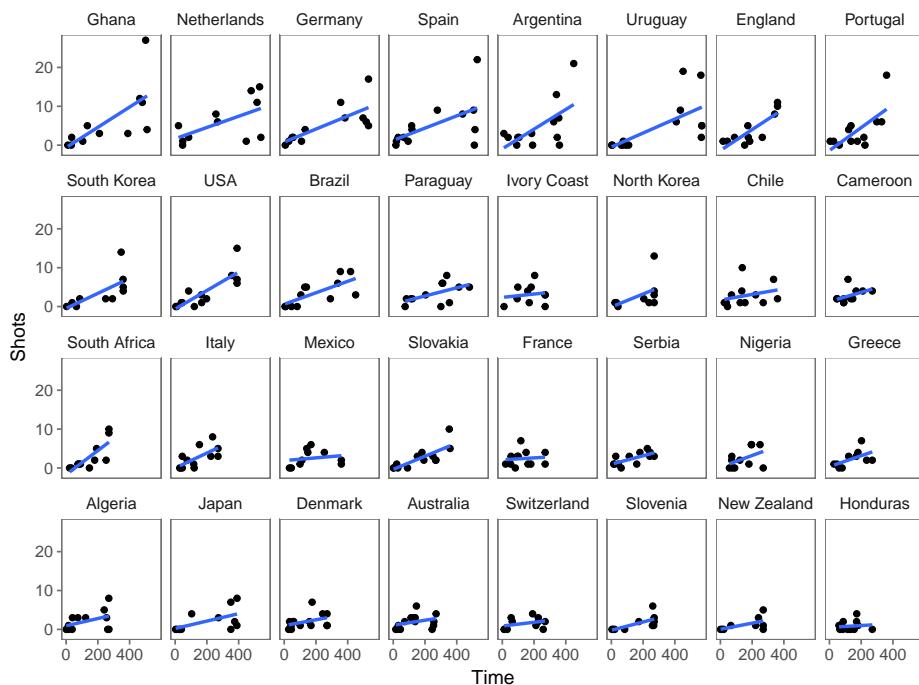
parameter.

## 4.7 Small multiples

Guideline 6: **When possible, use small multiples.**

*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.



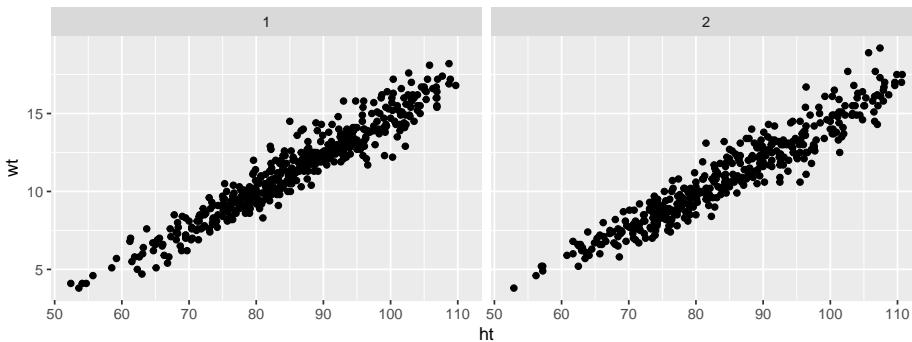
You can use the `facet` functions to create small multiples. This separates the graph into several small graphs, one for each level of a factor.

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:

```
ggplot(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:

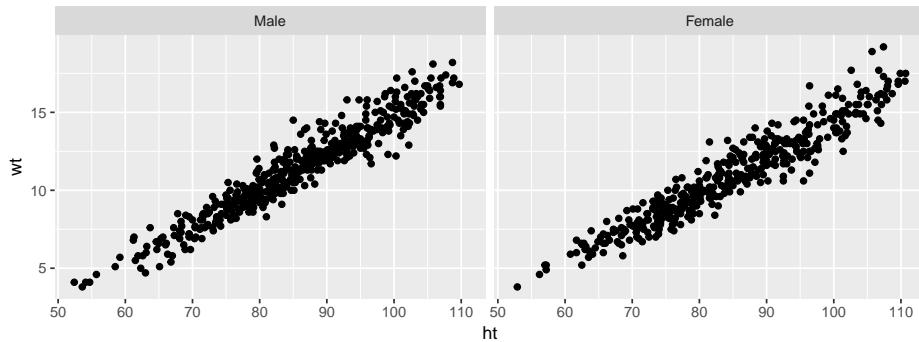
```
## Generic code
facet_wrap(~ [factor for faceting], ncol = [number of columns])
```

Often, when you do faceting, 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)
```

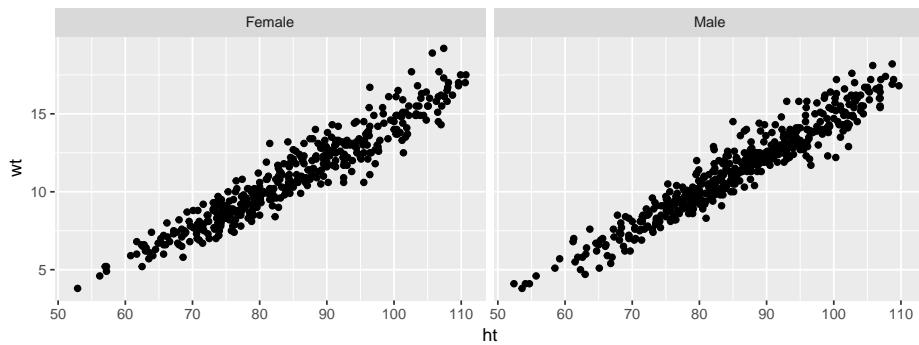


To re-order the factor, and show the plot for “Female” first, you can use `factor` to change the order of the levels:

```
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

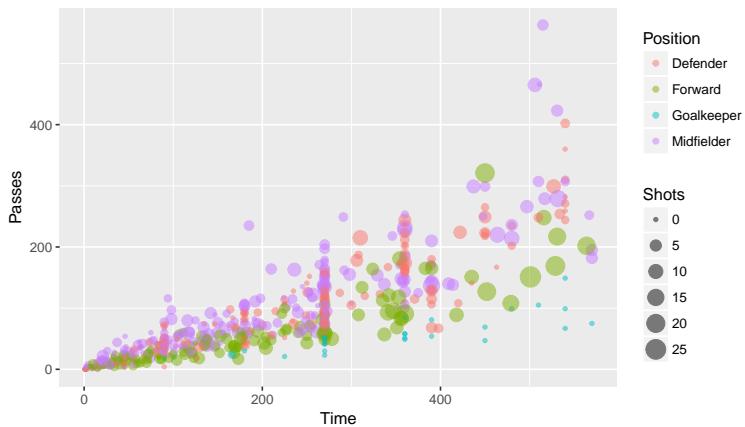
There are a number of different functions for adjusting scales. These follow the following convention:

```
## Generic code
scale_[aesthetic]_[vector type]
```

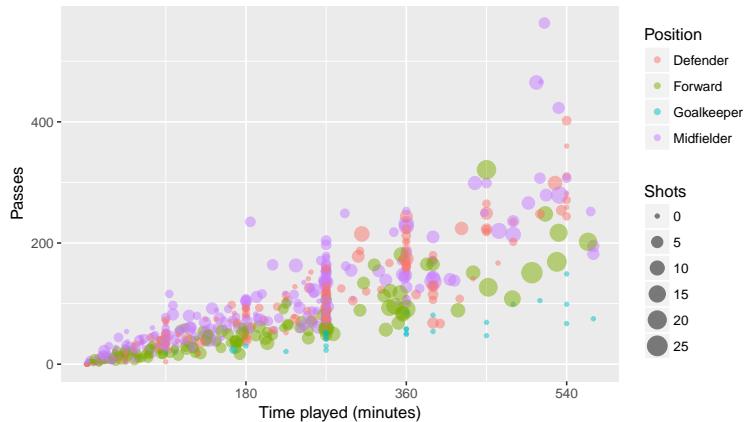
For example, to adjust the x-axis scale for a continuous variable, you'd use `scale_x_continuous`. You can use a `scale` function for an axis to change things like the axis label (which you could also change with `xlab` or `ylab`) as well as position and labeling of breaks.

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))
```

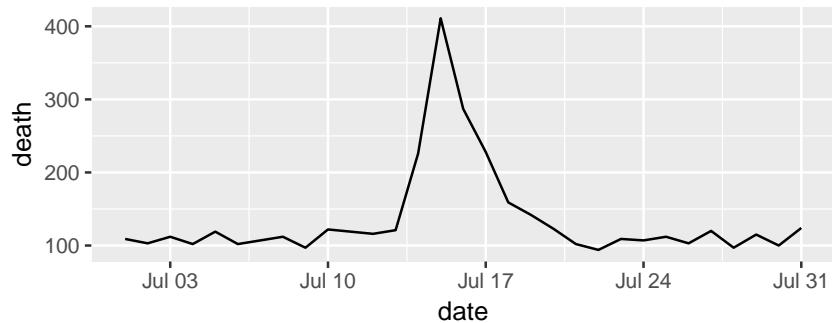


Parameters you might find useful in `scale` functions include:

Parameter	Description
<code>name</code>	Label or legend name
<code>breaks</code>	Vector of break points
<code>minor_breaks</code>	Vector of minor break points
<code>labels</code>	Labels to use for each break
<code>limits</code>	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:

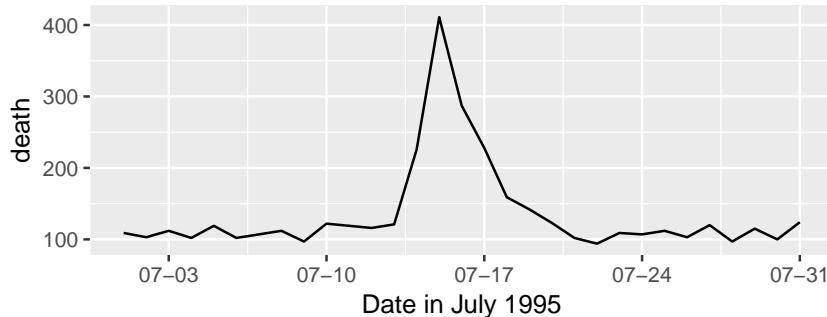
```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line()
```



And here's an example of changing the formating and name of the x-axis:

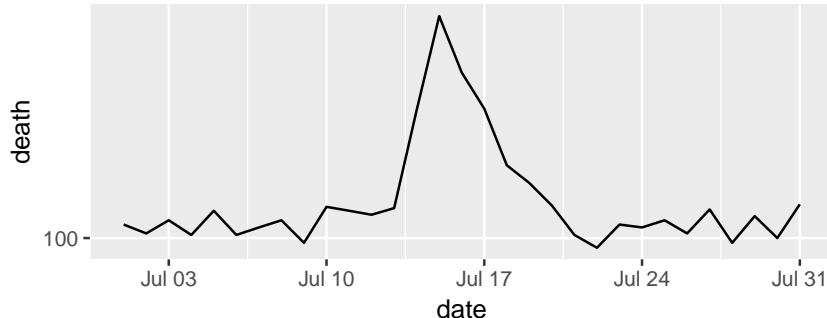
```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line()
```

```
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)
```



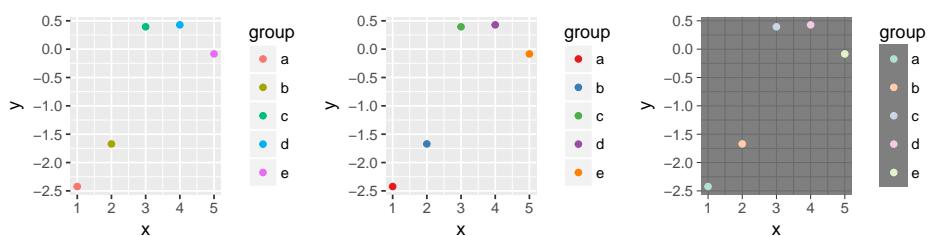
Set1 (qualitative)

PRGn (divergent)

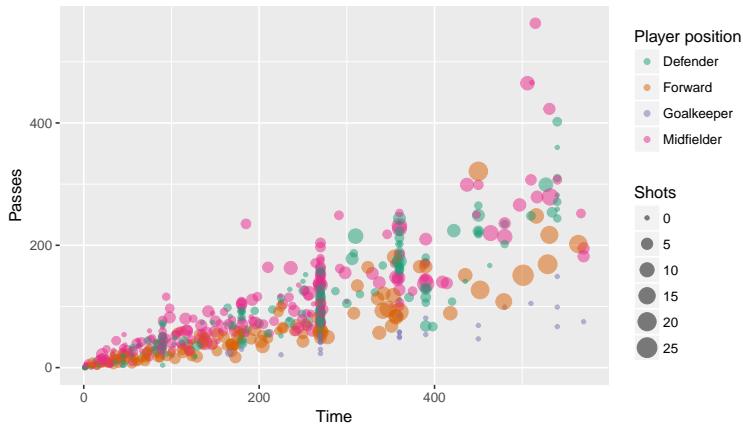
PuBuGn (sequential)

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)
```

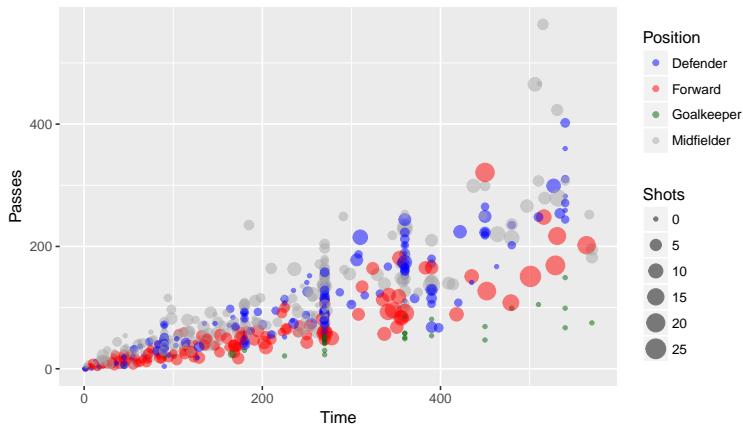


```
ggplot(worldcup, aes(x = Time, y = Passes,
                      color = Position, size = Shots)) +
  geom_point(alpha = 0.5) +
  scale_color_brewer(palette = "Dark2",
                     name = "Player position")
```



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

### 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.

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

```
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 %>%
  mutate(Player = rownames(worldcup))
head(worldcup, 2)
```

```
##           Team   Position Time Shots Passes Tackles Saves Player
## 1 Algeria Midfielder 16     0      6      0      0 Abdoun
## 2 Japan   Midfielder 351    0     101     14      0 Abe
```

### 4.10.2 Basic scatterplots with R base graphics and ggplot

Perform the following tasks:

- Install and load the `ggplot2` and `ggthemes` packages.
- For the `worldcup` data, plot a scatterplot of Time (on the x-axis) versus Shots.
- Add some ggplot elements to make this plot a bit more attractive. (For example, change the x- and y-axis labels and add a title.)

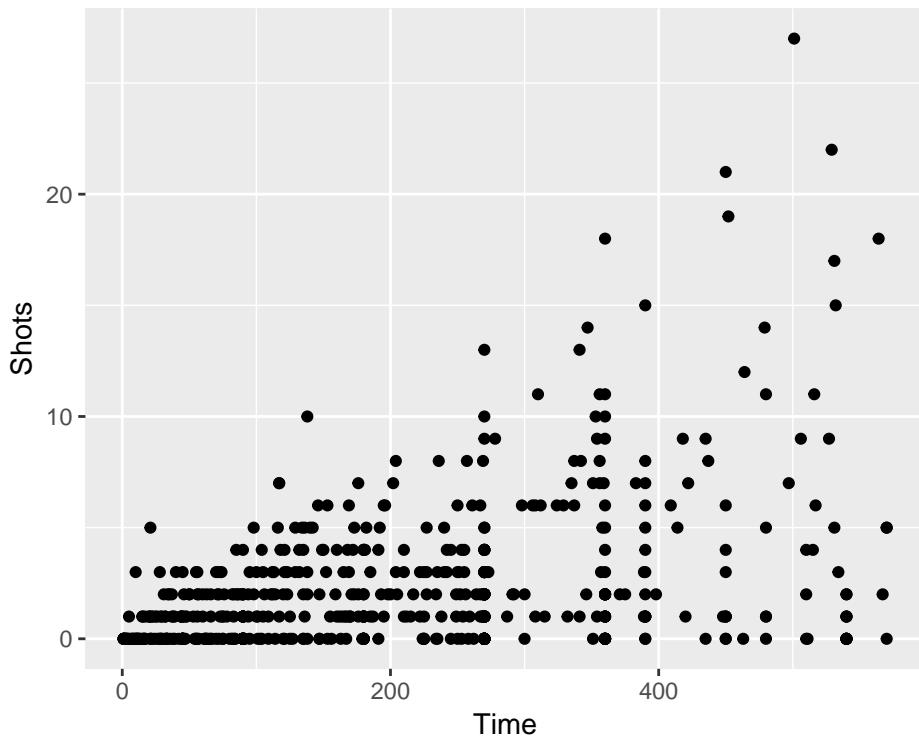
#### 4.10.2.1 Example R code

Install and load the `ggplot2` package:

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

Use the R base graphics to create a scatterplot of Time versus Shots:

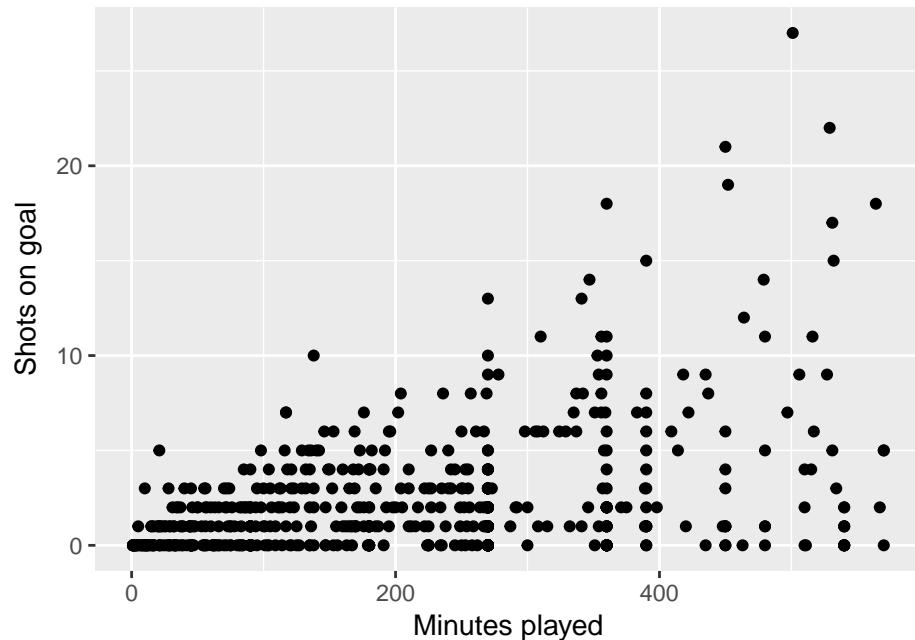
```
ggplot(worldcup, aes(x = Time, y = Shots)) +
  geom_point()
```



Use some of the plotting options to improve the graph's appearance:

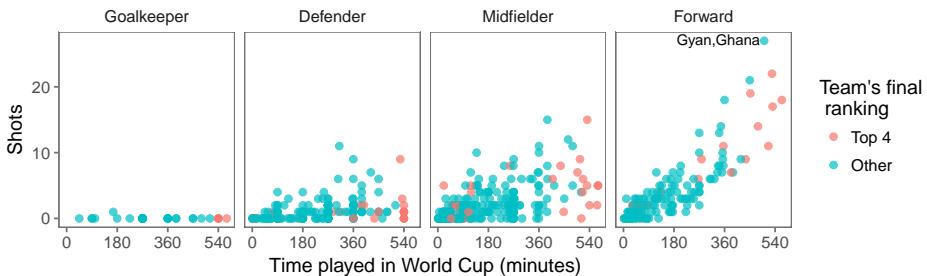
```
ggplot(worldcup, aes(x = Time, y = Shots)) +  
  geom_point() +  
  xlab("Minutes played") +  
  ylab("Shots on goal") +  
  ggtitle("Time played vs. shots taken\nWorld Cup 2010")
```

### Time played vs. shots taken World Cup 2010



#### 4.10.3 Fancier graphs

In this section, we'll work on creating a plot like this:



Try the following tasks:

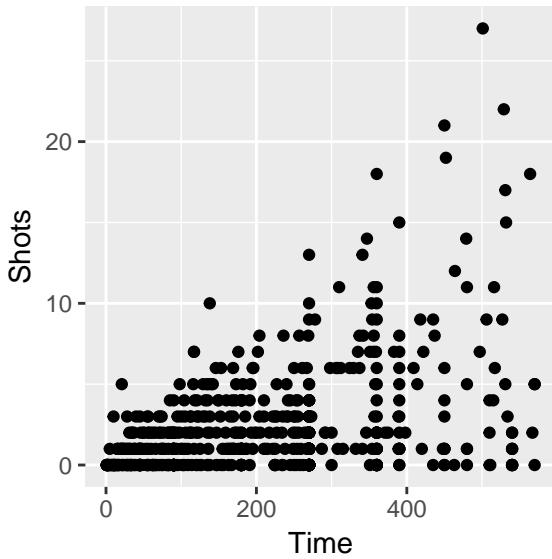
- First, before you start coding, talk with your group members about how this graph is different from the simple one you created with `ggplot` in the last section. Also discuss what you can figure out from this new graph that was less clear from the simpler one you created before.
- Use the `xlab()` function to make a clearer title for the x-axis. (You may have already written this code in the last section of this exercise.)

- 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; (b) make the points somewhat transparent; and (c) jitter the points. Try doing the first two with the simple ggplot scatterplot you created in the previous section of Shots by Time.
- 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.
- Create small multiples. The relationship between time played and shots taken is probably different by the players' positions. Use facetting to create different graphs for each position.
- Make order count: 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.
- 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.
- Increase data density: Try changing the theme, to come up with a graph with a bit less non-data ink. From the `ggthemes` package (you'll need to install it if you don't already have it), try some of the following themes: `theme_few()`, `theme_tufte()`, `theme_stata()`, `theme_fivethirtyeight()`, `theme_economist_white()`, and `theme_wsj()`.

#### 4.10.3.1 Example R code

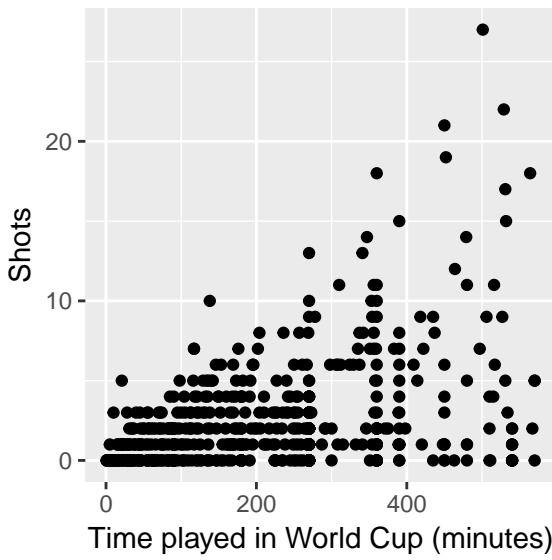
As a reminder, here's the code to do a simple scatterplot of Shots by Time for the `worldcup` data:

```
ggplot(worldcup, aes(x = Time, y = Shots)) +  
  geom_point()
```



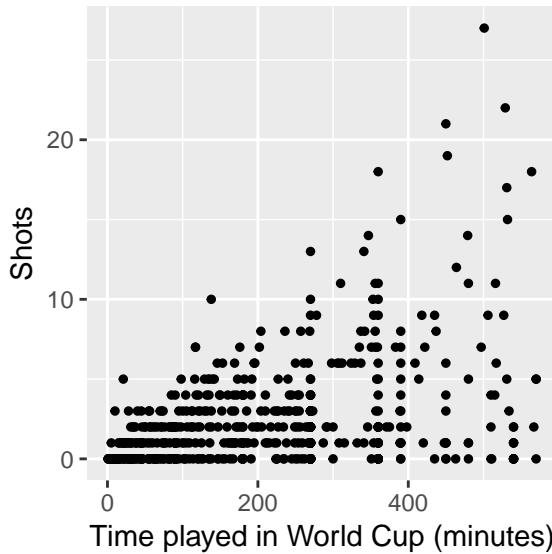
To add a clearer x-axis label than the current one, use `xlab()`:

```
ggplot(worldcup, aes(x = Time, y = Shots)) +
  geom_point() +
  xlab("Time played in World Cup (minutes)")
```



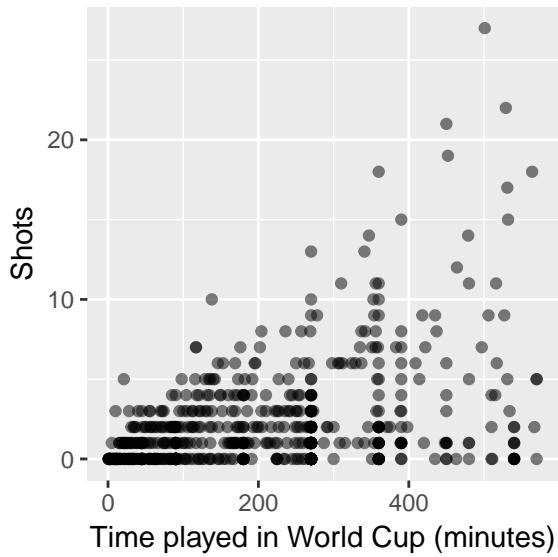
To make the points smaller, use the `size` option in `geom_point()` (smaller than about 2 = smaller than default, larger than about 2 = larger than default):

```
ggplot(worldcup, aes(x = Time, y = Shots)) +
  geom_point(size = 1) +
  xlab("Time played in World Cup (minutes)")
```



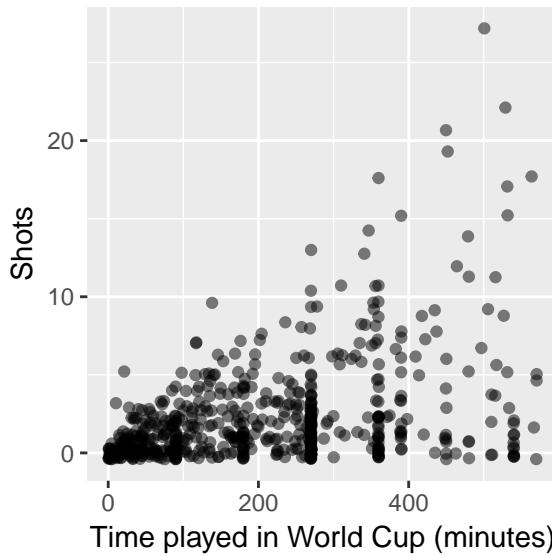
To make the points semi-transparent, use the `alpha` option in `geom_point()` (closer to 0 = more transparent, closer to 1 = more opaque):

```
ggplot(worldcup, aes(x = Time, y = Shots)) +
  geom_point(alpha = 0.5) +
  xlab("Time played in World Cup (minutes)")
```



To jitter the points some, use the `position = "jitter"` option in `geom_point()`:

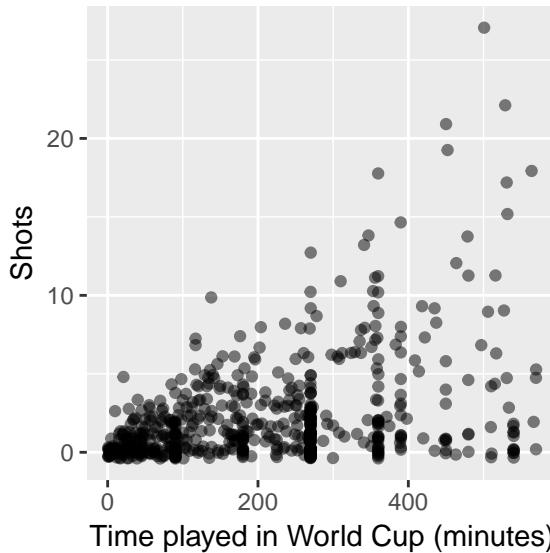
```
ggplot(worldcup, aes(x = Time, y = Shots)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)")
```



As an alternative, you could also jitter the points by using `geom_jitter` rather

than `geom_point`:

```
ggplot(worldcup, aes(x = Time, y = Shots)) +
  geom_jitter(size = 1.5, alpha = 0.5, width = 0.25) +
  xlab("Time played in World Cup (minutes)")
```



“Jittering” the points means adding some extra random noise in either the x- or y-direction, or in both directions. This technique can be particularly useful when you are trying to plot points for which one dimension is categorical, rather than continuous. You can specify which direction (x and / or y) is jittered, as well as the amount of noise to add. To find out more, check out the helpfile for the `geom_jitter` function.

To create a new column called `top_four`, first create vector that lists those top four teams, then create a logical vector in the dataframe for whether the team for that observation is in one of the top four teams:

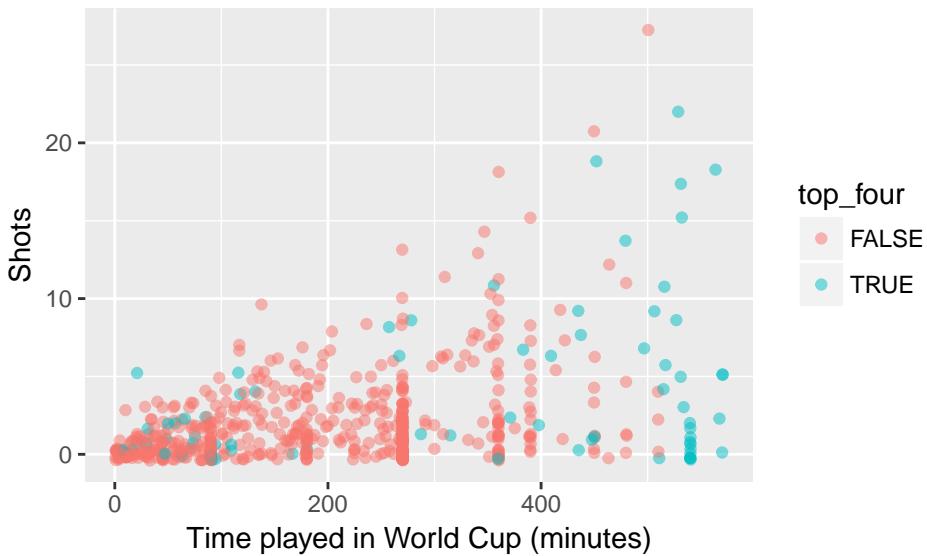
```
worldcup <- worldcup %>%
  mutate(top_four = Team %in% c("Spain", "Germany",
                                "Uruguay", "Netherlands"))
summary(worldcup$top_four)

##      Mode   FALSE    TRUE
## logical     517      78
```

To color points by this variable, use `color =` in the `aes()` part of the `ggplot()`

call:

```
ggplot(worldcup, aes(x = Time, y = Shots,
                      color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)")
```

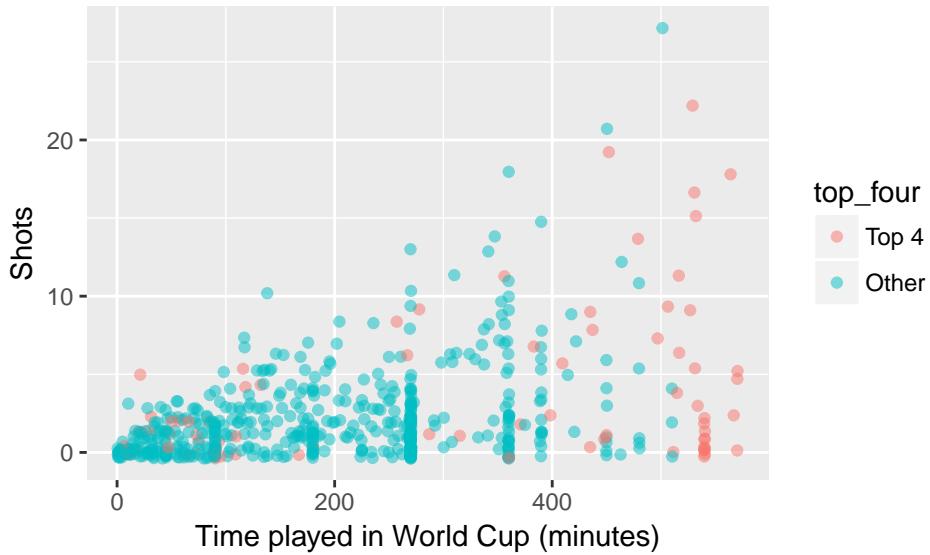


To create nicer labels for the legend for color, convert the `top_four` column into the factor class, with the labels you want to use in the figure legend:

```
worldcup <- worldcup %>%
  mutate(top_four = factor(top_four, levels = c(TRUE, FALSE),
                          labels = c("Top 4", "Other")))
summary(worldcup$top_four)
```

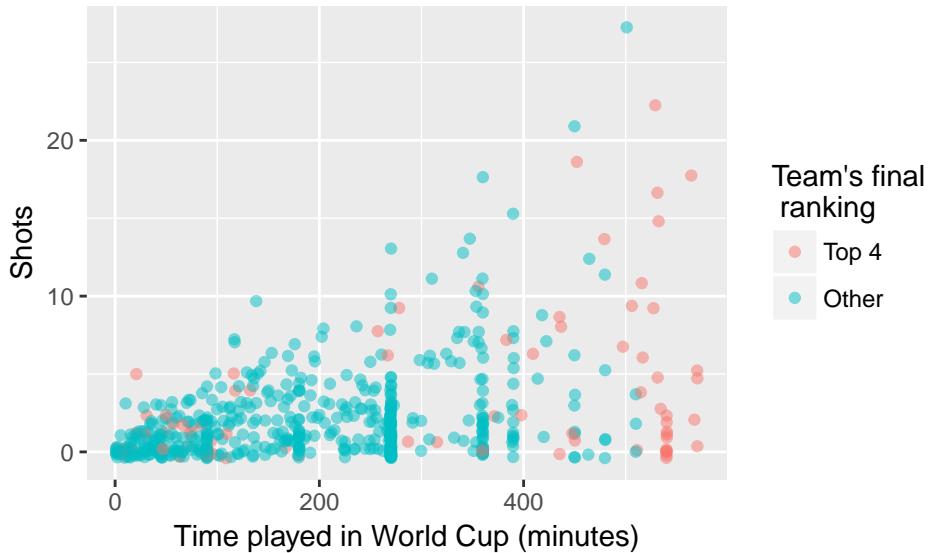
```
## Top 4 Other
##      78    517
```

```
ggplot(worldcup, aes(x = Time, y = Shots,
                      color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)")
```



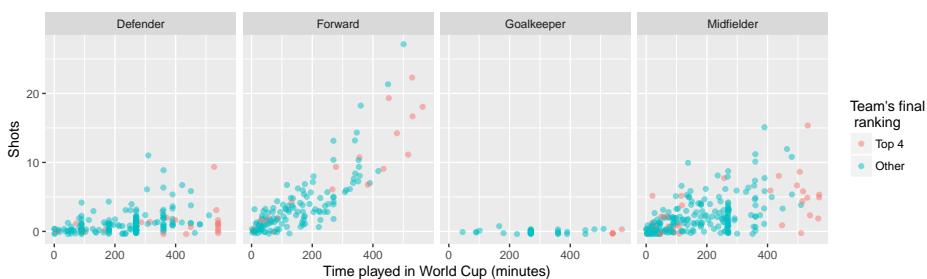
As a note, you can use the `scale_color_discrete()` function to put in a nicer legend title:

```
ggplot(worldcup, aes(x = Time, y = Shots,
                      color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)") +
  scale_color_discrete(name = "Team's final\nn ranking")
```



To create small multiples, use the `facet_grid()` command:

```
ggplot(worldcup, aes(x = Time, y = Shots,
                      color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)") +
  scale_color_discrete(name = "Team's final\nn ranking") +
  facet_grid(. ~ Position)
```



To re-order the `Position` column of the data frame, use the `levels` option of the `factor()` function. This re-sets how R saves the order of the levels of this factor.

```
worldcup <- worldcup %>%
  mutate(Position = factor(Position,
                           levels = c("Goalkeeper", "Defender",
                                      "Midfielder", "Forward")))
levels(worldcup$Position)

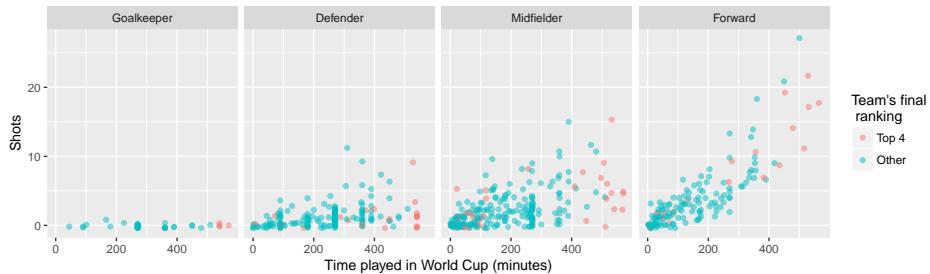
## [1] "Goalkeeper" "Defender"     "Midfielder"   "Forward"
```



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.

Then use the same code from before for your plot:

```
ggplot(worldcup, aes(x = Time, y = Shots,
                      color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)") +
  scale_color_discrete(name = "Team's final\nn ranking") +
  facet_grid(. ~ Position)
```



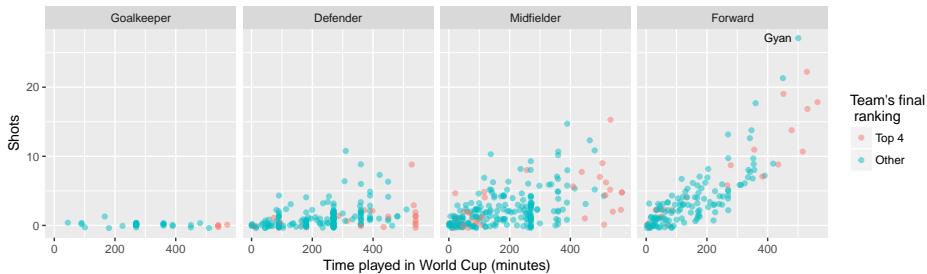
You can use the `filter` function with a logical statement comparing `Shots` to the maximum value of `Shots` (`max(Shots)`) to filter down to the row or rows of the player or players with the most shots:

```
most_shots <- worldcup %>%
  filter(Shots == max(Shots))
most_shots
```

```
##      Team Position Time Shots Passes Tackles Saves Player top_four
## 1 Ghana  Forward  501     27     151       1      0   Gyan    Other
```

Use `geom_text()` to label his 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. Also, I pasted on an extra space at the end of the player's name, to add some padding so the label wouldn't be right on top of the point.

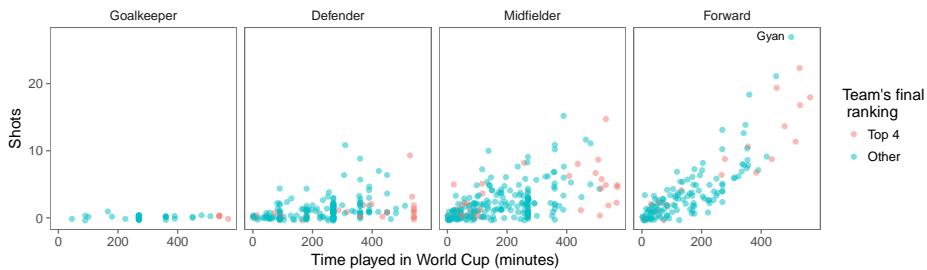
```
ggplot(worldcup, aes(x = Time, y = Shots,
                      color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)") +
  scale_color_discrete(name = "Team's final\nranking") +
  facet_grid(. ~ Position) +
  geom_text(data = most_shots,
            aes(label = paste(Player, " ")),
            colour = "black", size = 3,
            hjust = 1, vjust = 0.4)
```



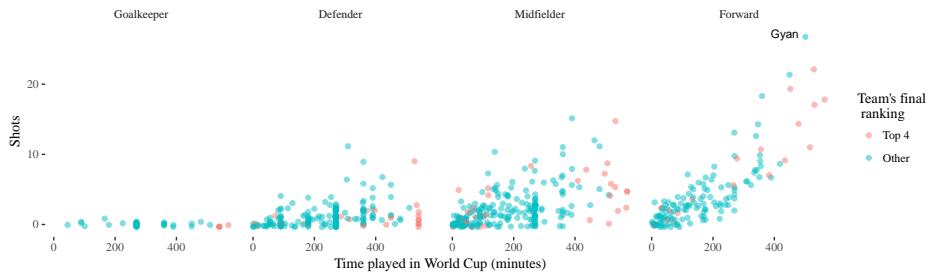
Try out different themes for the plot. First, I'll save everything we've done so far as the object `shot_plot`, then I'll try adding different themes:

```
shot_plot <- ggplot(worldcup, aes(x = Time, y = Shots,
                                     color = top_four)) +
  geom_point(size = 1.5, position = "jitter",
             alpha = 0.5) +
  xlab("Time played in World Cup (minutes)") +
  scale_color_discrete(name = "Team's final\nranking") +
  facet_grid(. ~ Position) +
  geom_text(data = most_shots,
            aes(label = paste(Player, " ")),
            colour = "black", size = 3,
            hjust = 1, vjust = 0.4)

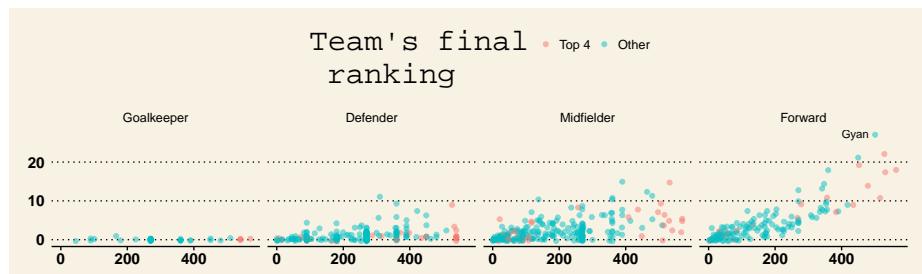
shot_plot + theme_few()
```



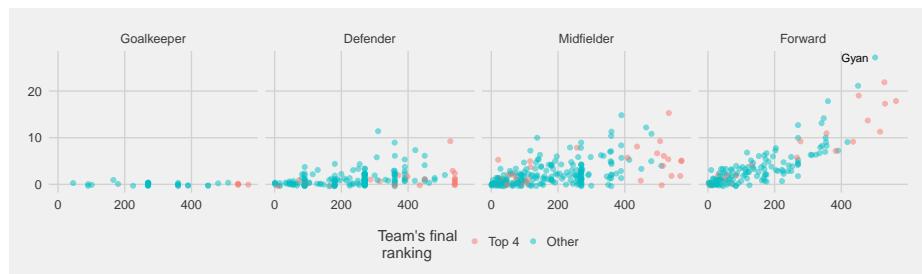
```
shot_plot + theme_tufte()
```



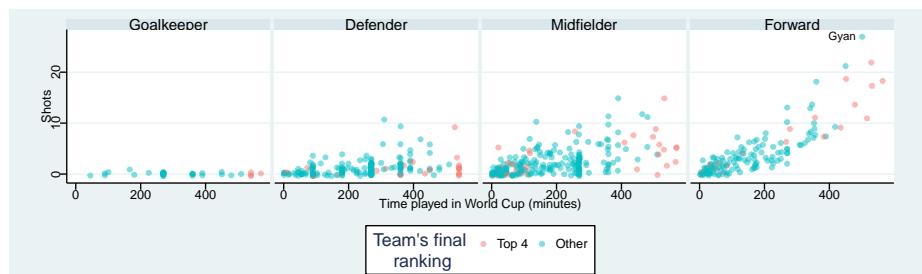
```
shot_plot + theme_wsj()
```



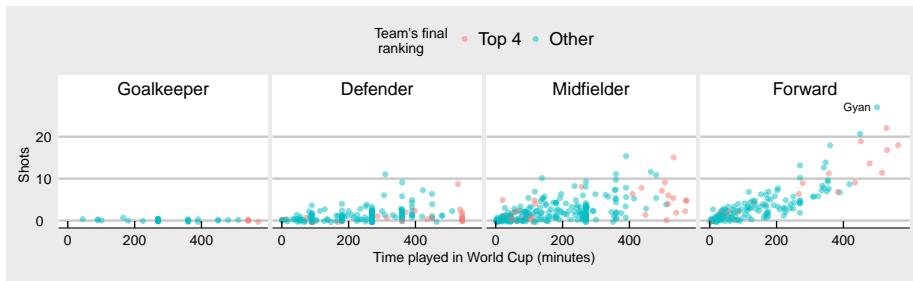
```
shot_plot + theme_fivethirtyeight()
```



```
shot_plot + theme_stata()
```



```
shot_plot + theme_economist_white()
```



#### 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

Download a pdf of the lecture slides covering this topic.

### 5.1 What is reproducible research?

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

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).

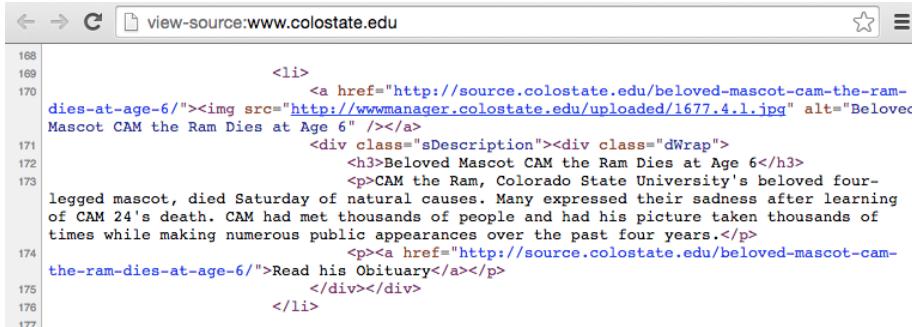
In markup languages, on the other hand, you markup the document directly to show what formatting the final version should have (e.g., you type **bold** in the file to end up with a document with **bold**).

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.



```

168
169
170      <li>
171          <a href="http://source.colostate.edu/beloved-mascot-cam-the-ram-
dies-at-age-6/"></a>
172          <div class="sDescription"><div class="dWrap">
173              <h3>Beloved Mascot CAM the Ram Dies at Age 6</h3>
174              <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>
175          <p><a href="http://source.colostate.edu/beloved-mascot-cam-
the-ram-dies-at-age-6/">Read his Obituary</a></p>
176      </div></div>
177  </li>

```

**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
**text**	**text**	boldface
*text*	*text*	italicized
'[text](www.google.com)'	[text](www.google.com)	hyperlink
# text		first-level header
## text		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

**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
<code>'echo'</code>	Print out the R code?
<code>'eval'</code>	Run the R code?
<code>'messages'</code>	Print out messages?
<code>'warnings'</code>	Print out warnings?
<code>'include'</code>	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
<code>results</code>	How to print results (e.g., <code>hide</code> runs the code, but doesn't print the results)
<code>fig.width</code>	Width to print your figure, in inches (e.g., <code>fig.width = 4</code> )
<code>fig.height</code>	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

writting 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.”

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

**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 (`today_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

### 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/eqnedit.html) 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`

## 5.6 Saving graphics files

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.

### 5.6.1 Tables in R Markdown

If you want to create a nice, formatted table from an R dataframe, 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 dataframe)   |
| <code>align</code>    | A vector giving the alignment for each column ('l', 'c', 'r')  |
| <code>caption</code>  | Table caption  |
| <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"))
```

**Table 5.3:** My new table

| First 3 letters | First 3 numbers |
|-----------------|-----------------|
| a               | -0.83           |
| b               | -0.42           |
| c               | 0.86            |

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.

## 5.7 In-course exercise

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.

### 5.7.1 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

### 5.7.2 Adding in R code

Now incorporate the R code from last week's exercise 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.

### 5.7.3 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?

#### 5.7.4 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.

#### 5.7.5 Trying out `knitr` with your own data

Pick 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).

- Create an R Markdown document. Add some text to describe the data you’re using.
- Write some code to read in the data (you can save it to your computer and read it from a file or, if the data’s online, read it in directly).
- Use `dplyr` functions (especially `summarize`) to create a dataframe with some summary statistics for the data. Print this out (just as R output, not as a formatted table, for now).
- Try using `kable` to create a formatted version of the summary table you created.
- Create at least one plot using the data. Try using `fig.width` and `fig.height` chunk options to change the size of the figure in the output.
- Find an image online related to your data. Save it to your computer and use `include_graphics` from the `knitr` package to include it in your R Markdown document.

## **Part III**

### **Part III: Intermediate**



# Chapter 6

## Entering and cleaning data #2

Download a pdf of the lecture slides covering this topic.

### 6.1 Joining datasets

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, 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, 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, by = "course"
##   course grade      day
## 1     w    NA    Tues
## 2     x    90 Mon / Fri
## 3     y    82      Tue
```

`full_join` keeps everything from both datasets:

```
full_join(x, y)

## Joining, by = "course"
##   course grade      day
## 1     x    90 Mon / Fri
## 2     y    82      Tue
## 3     z    78    <NA>
## 4     w    NA    Tues
```

## 6.2 Tidy data

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

## 6.3 Gathering

There are two functions from the `tidyverse` package (another member of the tidyverse) that you can use to change between wide and long data: `gather` and `spread`. Here is a description of these two functions:

- `gather`: Take several columns and gather them into two columns, one with the former column names, and one with the former cell values.
- `spread`: Take two columns and spread them into multiple columns. Column names for the new columns will come from one of the two original columns, while cell values will come from the other of the original columns.

The following examples are from `tidyverse` help files and show the effects of gathering and spreading a dataset.

Here is some simulated wide data:

```
wide_stocks[1:3, ]
```

```
##          time         X         Y         Z
## 1 2009-01-01 -1.2310944 -3.196428 -1.792398
```

```
## 2 2009-01-02 2.0017059 -3.347798 1.839397
## 3 2009-01-03 -0.8079211 -4.399936 6.748484
```

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, ]
```

```
##           time      X      Y      Z
## 1 2009-01-01 -1.2310944 -3.196428 -1.792398
## 2 2009-01-02  2.0017059 -3.347798  1.839397
## 3 2009-01-03 -0.8079211 -4.399936  6.748484
```

If you want to convert the dataframe to have all stock values in a single column, you can use `gather` to convert wide data to long data:

```
long_stocks <- gather(wide_stocks, key = stock,
                      value = price, -time)
long_stocks[1:5, ]
```

```
##           time stock      price
## 1 2009-01-01     X -1.2310944
## 2 2009-01-02     X  2.0017059
## 3 2009-01-03     X -0.8079211
## 4 2009-01-04     X -1.4787862
## 5 2009-01-05     X  0.3075898
```

In this "long" 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, ]
```

```
##           time stock      price
## 1 2009-01-01     X -1.2310944
## 2 2009-01-02     X  2.0017059
## 3 2009-01-03     X -0.8079211
## 4 2009-01-04     X -1.4787862
## 5 2009-01-05     X  0.3075898
```

The format for a `gather` call is:

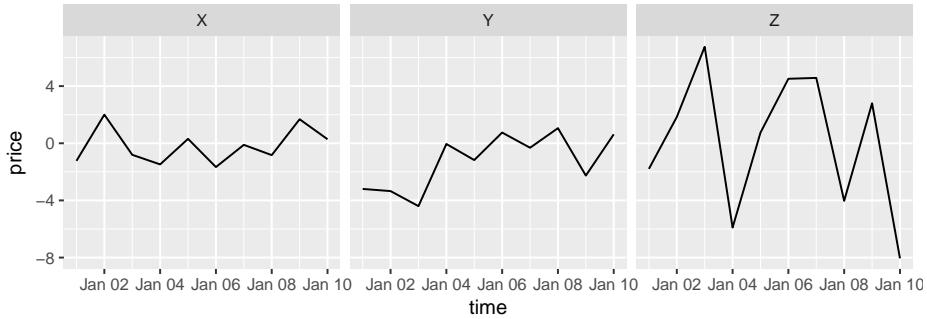
```
## Generic code
new_df <- gather(old_df,
  key = [name of column with old column names],
  value = [name of column with cell values],
  - [name of column(s) you want to
    exclude from gather])
```

Three important notes:

- Everything is gathered into one of two columns – one column with the old column names, and one column with the old cell values
- With the `key` and `value` arguments, you are just providing column names for the two columns that everything's gathered into.
- If there is a column you don't want to gather (`date` in the example), use `-` to exclude it in the `gather` call.

Notice how easy it is, now that the data is gathered, to use `stock` for aesthetics of faceting in a `ggplot2` call:

```
ggplot(long_stocks, aes(x = time, y = price)) +
  geom_line() +
  facet_grid(. ~ stock)
```



If you have data in a “long” format and would like to spread it out, you can use `spread` to do that:

```
stocks <- spread(long_stocks, key = stock, value = price)
stocks[1:5, ]
```

```
##       time      X      Y      Z
## 1 2009-01-01 -1.2310944 -3.1964276 -1.792398
## 2 2009-01-02  2.0017059 -3.3477980  1.839397
## 3 2009-01-03 -0.8079211 -4.3999365  6.748484
## 4 2009-01-04 -1.4787862 -0.0501619 -5.897630
```

```
## 5 2009-01-05 0.3075898 -1.1750339 0.746340
```

Notice that this reverses the action of `gather`.

“Spread” data is typically not tidy, so you often won’t want to use `spread` when you are preparing data for analysis. However, `spread` can be very helpful in creating clean 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 `spread` 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  -0.1852102 1.268802
## 2 Y       10  -1.2314339 1.951851
## 3 Z       10   0.1444207 4.915781

stock_summary %>%
  mutate("Mean (Std.dev.)" = paste0(round(mean, 2), " (",
                                         round(sd, 2), ")")) %>%
  select(-mean, -sd) %>%
  gather(key = "Statistic", value = "Value", -stock) %>%
  spread(key = stock, value = Value) %>%
  knitr::kable()
```

| Statistic       | X            | Y            | Z           |
|-----------------|--------------|--------------|-------------|
| Mean (Std.dev.) | -0.19 (1.27) | -1.23 (1.95) | 0.14 (4.92) |
| N               | 10           | 10           | 10          |

## 6.4 In-course exercise

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 |

---

| File name                        | Description   |
|----------------------------------|---|
| <code>mexico_exposure.csv</code> | Daily death counts and environmental measurements for Mexico City, Mexico, for 2008 |
| <code>mexico_deaths.csv</code>   |   |
| <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)
```

### 6.4.1 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.

### 6.4.2 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`.
- Merge the two datasets together to create the dataframe `mexico`. Exclude all columns except the outcome (`deaths`), date, and mean temperature. Convert the date to a date class.
- Try combining all the steps in the previous task into one “chained” command using the pipe operator, `%>%`.
- Use this new dataframe to plot deaths by date using `ggplot`.

#### 6.4.2.1 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>   <int>
## 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
##   <chr>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 1/1/08      7.8    17.8  11.770833  53.45833  2.6625000 0.009250000
## 2 1/2/08      2.6     9.8   6.637500  61.70833  3.3458333 0.005416667
## 3 1/3/08      1.1    15.6   7.041667  59.91667  1.8857143 0.015958333
## 4 1/4/08      3.1    20.6  10.862500  57.54167  1.1958333 0.040833333
```

```
## 5 1/5/08      6.0    21.3 13.404167 45.70833 0.9875000 0.046916667
## 6 1/6/08      7.2    22.1 14.341667 40.75000 0.8541667 0.028583333
## # ... with 7 more variables: NO2 <dbl>, NOX <dbl>, O3 <dbl>, CO <dbl>,
## #   SO2 <dbl>, PM10 <dbl>, PM25 <dbl>
```

Merge the two datasets together to create the dataframe `mexico`. Exclude all columns except the outcome (deaths), date, and mean temperature. Convert the date to a date class.

```
mexico <- full_join(mex_deaths, mex_exp, by = "day")
mexico <- select(mexico, day, deaths, temp_mean)

library(lubridate) ## For parsing dates
mexico <- mutate(mexico, day = mdy(day))
```

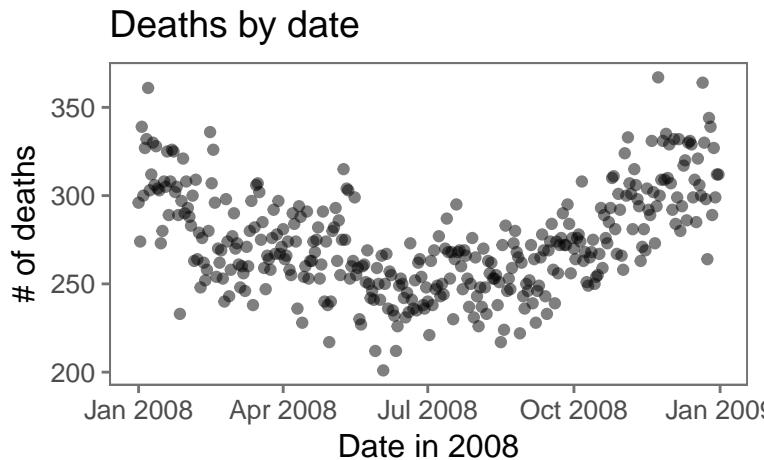
Try combining all the steps in the previous task into one “chained” command:

```
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>  <int>     <dbl>
## 1 2008-01-01    296 11.770833
## 2 2008-01-02    274  6.637500
## 3 2008-01-03    339  7.041667
## 4 2008-01-04    300 10.862500
## 5 2008-01-05    327 13.404167
## 6 2008-01-06    332 14.341667
```

Use this new dataframe to plot deaths by date using `ggplot`:

```
ggplot(mexico, aes(x = day, y = deaths)) +
  geom_point(size = 1.5, alpha = 0.5) +
  xlab("Date in 2008") + ylab("# of deaths") +
  ggtitle("Deaths by date") +
  theme_few()
```



### 6.4.3 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`.
- Run the following code to create new columns named `type` and `country` that split up the `variable` column into type (“Cases” or “Deaths”) and country (“Guinea”, “Liberia”, etc.). (This type of code is moving towards using regular expressions to clean up really messy data, which we’ll talk about some in the third section.)

```
foo <- strsplit(as.character(ebola$variable), split = "_")
bar <- matrix(unlist(foo), ncol = 2, byrow = TRUE)

ebola$type <- factor(bar[, 1])
ebola$country <- factor(bar[, 2])
```

- Use `dplyr` functions and piping to remove `Day` and `variable` (now that you’ve split it into `type` and `country`) and to convert `Date` to a date class.
- Use the `dplyr` function `spread()` to convert the data so you have separate columns for the two variables of numbers of `Cases` and `Deaths`.
- Remove any observations where counts of cases or deaths are missing for that country.
- Challenge question (you can do the next step without doing this, but your graphs won’t be in order): Create a dataframe called `case_sum` that gives the total number of cases recorded for each country. (Hint: Use the `dplyr` functions `group_by()` and `summarize()`.) Use `arrange()` to re-order this

dataset by the order of the number of cases, and then use this arrangement to re-order the levels in `country` in your main `ebola` dataset, so that your graphs in the next step will be ordered from the country with the most ebola cases to the one with the least.

- 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 (in the `gridExtra` package) and see how that changes these graphs.
- Based on these plots, what would your next questions be about this data before you used it for an analysis?
- Super-challenge question: Can you put all of the steps of this cleaning process into just a few “chaining” calls?

#### 6.4.3.1 Example R code

Read the data in using `read_csv`. (Review question: Why can't you read it directly 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
##       <chr> <int>      <int>        <int>          <int>
## 1 1/5/2015    289      2776         NA        10030
## 2 1/4/2015    288      2775         NA        9780
## 3 1/3/2015    287      2769       8166        9722
## 4 1/2/2015    286        NA       8157          NA
## 5 12/31/2014   284      2730       8115        9633
## 6 12/28/2014   281      2706       8018        9446
## # ... with 13 more variables: Cases_Nigeria <int>, Cases_Senegal <int>,
## #   Cases_UnitedStates <int>, Cases_Spain <int>, Cases_Mali <int>,
## #   Deaths_Guinea <int>, Deaths_Liberia <int>, Deaths_SierraLeone <int>,
## #   Deaths_Nigeria <int>, Deaths_Senegal <int>, Deaths_UnitedStates <int>,
## #   Deaths_Spain <int>, Deaths_Mali <int>
```

Change the data to long data using the `gather()` function from `dplyr`:

```
ebola <- gather(ebola, variable, count, -Date, -Day)
head(ebola)
```

```
## # A tibble: 6 x 4
##       Date     Day   variable count
##   <chr> <int>    <chr> <int>
## 1 1/5/2015    289 Cases_Guinea 2776
## 2 1/4/2015    288 Cases_Guinea 2775
## 3 1/3/2015    287 Cases_Guinea 2769
## 4 1/2/2015    286 Cases_Guinea NA
## 5 12/31/2014   284 Cases_Guinea 2730
## 6 12/28/2014   281 Cases_Guinea 2706
```

Split variable into type and country:

```
foo <- strsplit(as.character(ebola$variable), split = "_")
ebola[ , c("type", "country")] <- matrix(unlist(foo), ncol = 2, byrow = TRUE)

head(ebola)
```

```
## # A tibble: 6 x 6
##       Date     Day   variable count type country
##   <chr> <int>    <chr> <int> <chr> <chr>
## 1 1/5/2015    289 Cases_Guinea 2776 Cases Guinea
## 2 1/4/2015    288 Cases_Guinea 2775 Cases Guinea
## 3 1/3/2015    287 Cases_Guinea 2769 Cases Guinea
## 4 1/2/2015    286 Cases_Guinea NA Cases Guinea
## 5 12/31/2014   284 Cases_Guinea 2730 Cases Guinea
## 6 12/28/2014   281 Cases_Guinea 2706 Cases Guinea
```

Use dplyr functions and piping to remove Day and variable and to convert Date to a date class:

```
ebola <- select(ebola, -Day, -variable) %>%
  mutate(Date = mdy(Date))

head(ebola)
```

```
## # A tibble: 6 x 4
##       Date count type country
##   <date> <int> <chr> <chr>
## 1 2015-01-05 2776 Cases Guinea
## 2 2015-01-04 2775 Cases Guinea
## 3 2015-01-03 2769 Cases Guinea
## 4 2015-01-02 NA Cases Guinea
## 5 2014-12-31 2730 Cases Guinea
## 6 2014-12-28 2706 Cases Guinea
```

Convert the data so you have separate columns for the two variables of numbers of Cases and Deaths:

```
ebola <- spread(ebola, type, count)
head(ebola)
```

```
## # A tibble: 6 x 4
##       Date   country Cases Deaths
##   <date>   <chr>   <int>  <int>
## 1 2014-03-22 Guinea     49     29
## 2 2014-03-22 Liberia    NA     NA
## 3 2014-03-22 Mali      NA     NA
## 4 2014-03-22 Nigeria    NA     NA
## 5 2014-03-22 Senegal    NA     NA
## 6 2014-03-22 SierraLeone 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 4
##       Date   country Cases Deaths
##   <date>   <chr>   <int>  <int>
## 1 2014-03-22 Guinea     49     29
## 2 2014-03-24 Guinea     86     59
## 3 2014-03-25 Guinea     86     60
## 4 2014-03-26 Guinea     86     62
## 5 2014-03-27 Guinea    103     66
## 6 2014-03-27 Liberia     8      6
```

Create a dataframe called `case_sum` that gives the total number of cases recorded for each country. Use `arrange()` to re-order this dataset by the order of the number of cases:

```
case_sum <- group_by(ebola, country) %>%
  summarize(Cases = sum(Cases, na.rm = TRUE))
case_sum

## # A tibble: 8 x 2
##       country Cases
##   <chr>   <int>
## 1 Guinea 82099
## 2 Liberia 191351
## 3 Mali    42
## 4 Nigeria 636
## 5 Senegal 24
```

```
## 6 SierraLeone 211169
## 7 Spain 16
## 8 UnitedStates 59
```

Use this arrangement to re-order the levels in `country` in your main `ebola` dataset, so that your graphs in the next step will be ordered from the country with the most ebola cases to the one with the least:

```
case_sum <- arrange(case_sum, desc(Cases))
case_sum
```

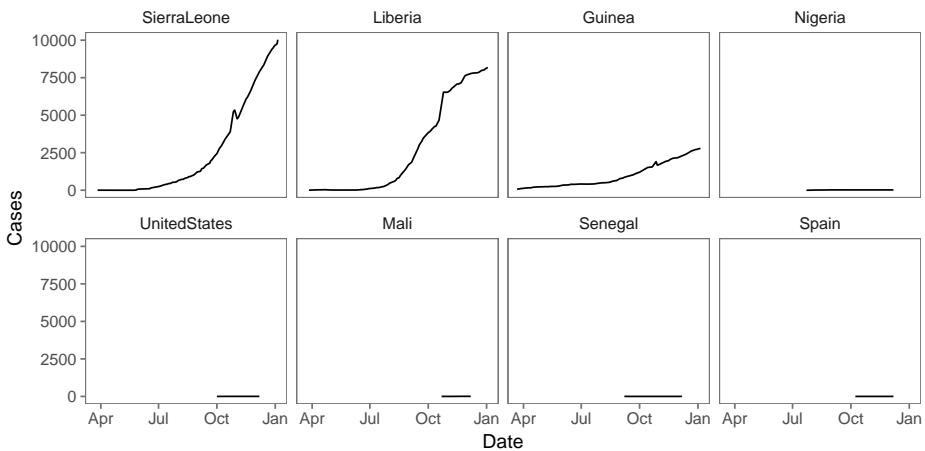
```
## # A tibble: 8 x 2
##       country   Cases
##       <chr>     <int>
## 1 SierraLeone 211169
## 2 Liberia    191351
## 3 Guinea     82099
## 4 Nigeria    636
## 5 UnitedStates 59
## 6 Mali        42
## 7 Senegal     24
## 8 Spain        16
```

```
ebola <- mutate(ebola,
                 country = factor(country, levels = case_sum$country))
levels(ebola$country)
```

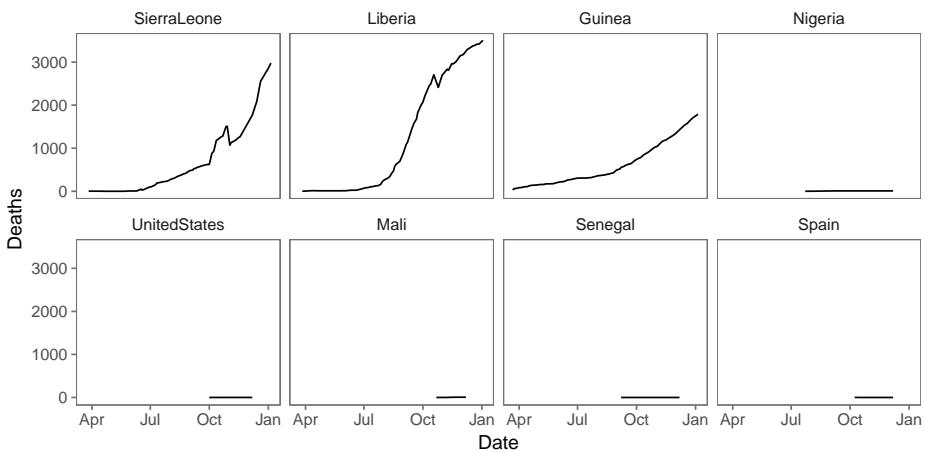
```
## [1] "SierraLeone"  "Liberia"       "Guinea"        "Nigeria"
## [5] "UnitedStates" "Mali"          "Senegal"       "Spain"
```

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_few()
```

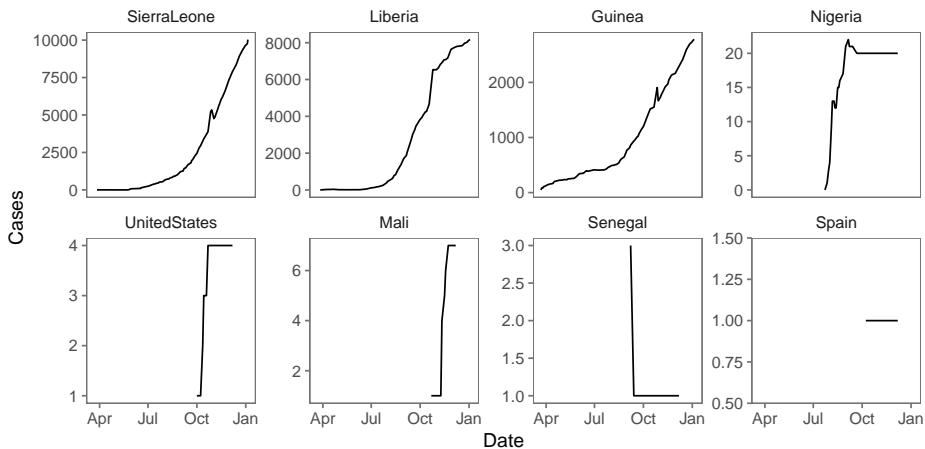


```
ggplot(ebola, aes(x = Date, y = Deaths)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4) +
  theme_few()
```

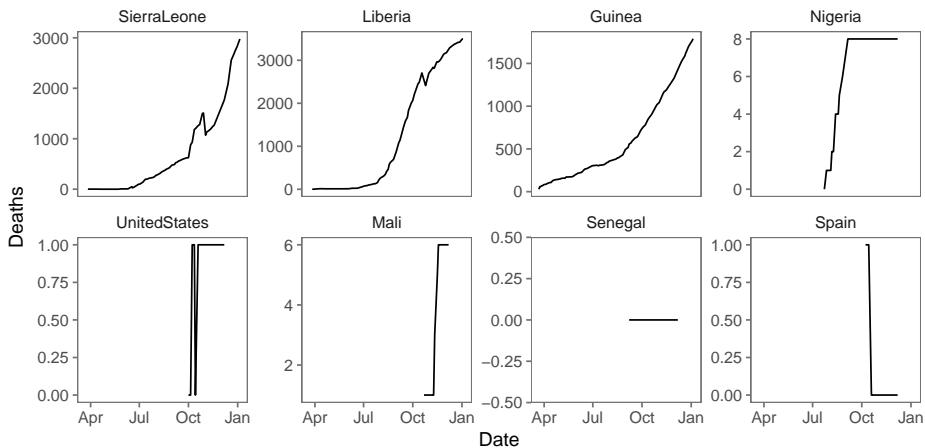


Try using the option `scales = "free_y"` in the `facet_wrap()` function (in the `gridExtra` package) and see how that changes these graphs:

```
ggplot(ebola, aes(x = Date, y = Cases)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4, scales = "free_y") +
  theme_few()
```



```
ggplot(ebola, aes(x = Date, y = Deaths)) +
  geom_line() +
  facet_wrap(~ country, ncol = 4, scales = "free_y") +
  theme_few()
```



Put all of the steps of this cleaning process into just a few “chaining” calls. (Note: I’m using `sub` here instead of `strsplit` for the variable-splitting step, just to keep the code a bit cleaner. Again, this is using regular expressions, which we’ll cover more later in the course.)

```
ebola <- read_csv(ebola_url) %>%
  gather(variable, count, -Date, -Day) %>%
  mutate(type = sub(".*", "", variable),
        country = sub(".*_", "", variable)) %>%
  select(-Day, -variable) %>%
```

```

    mutate(Date = mdy(Date)) %>%
    spread(type, count) %>%
    filter(!is.na(Cases) & !is.na(Deaths))

case_sum <- group_by(ebola, country) %>%
    summarize(Cases = sum(Cases, na.rm = TRUE)) %>%
    arrange(desc(Cases))

ebola <- mutate(ebola,
    country = factor(country, levels = case_sum$country))

case_sum

## # A tibble: 8 x 2
##       country   Cases
##       <chr>     <int>
## 1 SierraLeone 211169
## 2 Liberia      191351
## 3 Guinea       82099
## 4 Nigeria      636
## 5 UnitedStates  59
## 6 Mali          42
## 7 Senegal       24
## 8 Spain          16

head(ebola)

## # A tibble: 6 x 4
##       Date   country   Cases   Deaths
##       <date> <fctr>     <int>    <int>
## 1 2014-03-22 Guinea      49      29
## 2 2014-03-24 Guinea      86      59
## 3 2014-03-25 Guinea      86      60
## 4 2014-03-26 Guinea      86      62
## 5 2014-03-27 Guinea     103      66
## 6 2014-03-27 Liberia      8       6

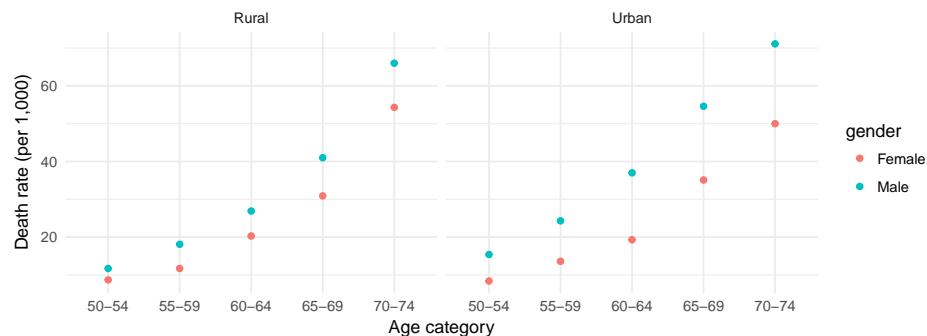
```

#### 6.4.4 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:



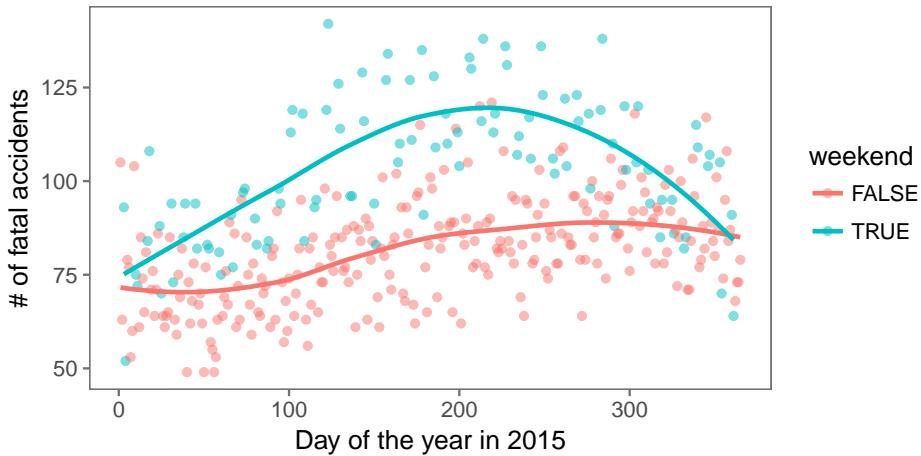
There is no example R code for this – try to figure out the code yourselves. We will go over a solution in class. You may find the RStudio Data Wrangling cheatsheet helpful for remembering which tidyverse functions do what.

#### 6.4.5 Exploring Fatality Analysis Reporting System (FARS) data

- Explore the interactive visualization at <http://metrocosm.com/10-years-of-traffic-accidents-mapped.html>. This was created by Max Galka using this dataset.
- Go to FARS web page. We want to get the raw data on fatal accidents. Navigate this page to figure out how you can get this raw data for the whole county for 2015. Save 2015 data to your computer. What is the structure of how this data is saved (e.g., directory structure, file structure)?
- On the FARS web page, find the documentation describing this raw data. Look through both this documentation and the raw files you downloaded to figure out what information is included in the data.
- Read the `accident.csv` file for 2015 into R (this is one of the files you’ll get if you download the raw data for 2015). Use the documentation to figure out what each column represents.
- 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. (Hint: Functions from the `lubridate` package will be very helpful,

including `yday` and `wday`).

- Discuss which of the variables in this dataset could be used to merge the dataset with other appropriate data, either other datasets in the FARS raw data, or outside datasets.
- 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.





# Chapter 7

## Exploring data #2

Download a pdf of the lecture slides covering this topic.

### 7.1 Parentheses

If you put parentheses around an entire code statement, it will both run the code and print out the answer.

```
study_months <- c("Jan", "Feb", "Mar")
study_months

## [1] "Jan" "Feb" "Mar"

(study_months <- c("Jan", "Feb", "Mar"))

## [1] "Jan" "Feb" "Mar"
```

### 7.2 Loops

Loops allow you to “walk through” and repeat the same code for different values of an index. For each run of the loop, R is told that, for **some index** in **some vector**, do **some code**. For example, the following loop specifies: For **i** in **1:3**, **print(i)**.

```
for(i in c(1, 2, 3)){
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

Note that this code is equivalent to:

```
i <- 1
print(i)
```

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

```
i <- 2
print(i)
```

```
## [1] 2
```

```
i <- 3
print(i)
```

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

Often, the index will be set to a number for each cycle of the loop, and then the index will be used within the code to index vectors or dataframes:

```
study_months <- c("Jan", "Feb", "Mar")
for(i in c(1, 3)){
    print(study_months[i])
}
```

```
## [1] "Jan"
## [1] "Mar"
```

Often, you want to set the index to sequential numbers (e.g., 1, 2, 3, 4). In this case, you can save time by using the : notation to create a vector of a sequence of numbers:

```
for(i in 1:3){
    print(i)
}
```

```
## [1] 1  
## [1] 2  
## [1] 3
```

With this notation, sometimes it may be helpful to use the `length` function to set the largest index value for the loop as the length of a vector (or `nrow` for indexing a dataframe). For example:

```
study_months <- c("Jan", "Feb", "Mar")  
for(i in 1:length(study_months)){  
    print(study_months[i])  
}
```

```
## [1] "Jan"  
## [1] "Feb"  
## [1] "Mar"
```

Sometimes, you want to set the index for each cycle of the loop to something that is not a number. You can set the index to any class of vector.

Remember that a loop works by saying for **some index** in **some vector**, do **some code**. For example, you may want to run: for `study_month` in `study_months`, `print(study_month)`:

```
study_months <- c("Jan", "Feb", "Mar")  
for(study_month in study_months){  
    print(study_month)  
}
```

```
## [1] "Jan"  
## [1] "Feb"  
## [1] "Mar"
```

Note that this is equivalent to:

```
study_month <- "Jan"  
print(study_month)
```

```
## [1] "Jan"
```

```
study_month <- "Feb"  
print(study_month)
```

```
## [1] "Feb"
```

```
study_month <- "Mar"
print(study_month)
```

```
## [1] "Mar"
```

What would this loop do?

```
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
    var_mean <- mean(worldcup[, vars[i]])
    print(var_mean)
}
```

```
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
    var_mean <- mean(worldcup[, vars[i]])
    print(var_mean)
}
```

```
## [1] 208.8639
## [1] 2.304202
## [1] 84.52101
## [1] 4.191597
## [1] 0.6672269
```

What would this loop do?

```
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
    var_mean <- mean(worldcup[, vars[i]])
    var_mean <- round(var_mean, 1)
    out <- paste0("mean of ", vars[i], ": ", var_mean)
    print(out)
}
```

To figure out, you can set `i <- 1` and then walk through the loop:

```
i <- 1
(var_mean <- mean(worldcup[, vars[i]]))

## [1] 208.8639
```

```
(var_mean <- round(var_mean, 1))

## [1] 208.9

(out <- paste0("mean of ", vars[i], ": ", var_mean))

## [1] "mean of Time: 208.9"

vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
for(i in 1:length(vars)){
  var_mean <- mean(worldcup[, vars[i]])
  var_mean <- round(var_mean, 1)
  out <- paste0("mean of ", vars[i], ": ", var_mean)
  print(out)
}

## [1] "mean of Time: 208.9"
## [1] "mean of Shots: 2.3"
## [1] "mean of Passes: 84.5"
## [1] "mean of Tackles: 4.2"
## [1] "mean of Saves: 0.7"
```

Often, it's convenient to create a dataset to fill up as you loop through:

```
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
my_df <- data.frame(variable = vars, mean = NA)
for(i in 1:nrow(my_df)){
  var_mean <- mean(worldcup[, vars[i]])
  my_df[i, "mean"] <- round(var_mean, 1)
}
```

```
vars <- c("Time", "Shots", "Passes", "Tackles", "Saves")
(my_df <- data.frame(variable = vars, mean = NA))
```

```
##   variable mean
## 1      Time  NA
## 2     Shots  NA
## 3    Passes  NA
## 4   Tackles  NA
## 5    Saves  NA
```

```

i <- 1
(var_mean <- mean(worldcup[ , vars[i]]))

## [1] 208.8639

my_df[i , "mean"] <- round(var_mean, 1)
my_df

##   variable  mean
## 1      Time 208.9
## 2     Shots    NA
## 3    Passes    NA
## 4   Tackles    NA
## 5    Saves    NA

for(i in 1:nrow(my_df)){
  var_mean <- mean(worldcup[ , vars[i]])
  my_df[i , "mean"] <- round(var_mean, 1)
}
my_df

##   variable  mean
## 1      Time 208.9
## 2     Shots  2.3
## 3    Passes 84.5
## 4   Tackles  4.2
## 5    Saves  0.7

```

Note: This is a pretty simplistic example. There are some easier ways to have done this:

```

worldcup %>%
  summarize(Time = mean(Time), Passes = mean(Passes),
            Shots = mean(Shots), Tackles = mean(Tackles),
            Saves = mean(Saves)) %>%
  gather(key = var, value = mean) %>%
  mutate(mean = round(mean, 1))

##       var  mean
## 1      Time 208.9
## 2    Passes 84.5
## 3     Shots  2.3
## 4   Tackles  4.2
## 5    Saves  0.7

```

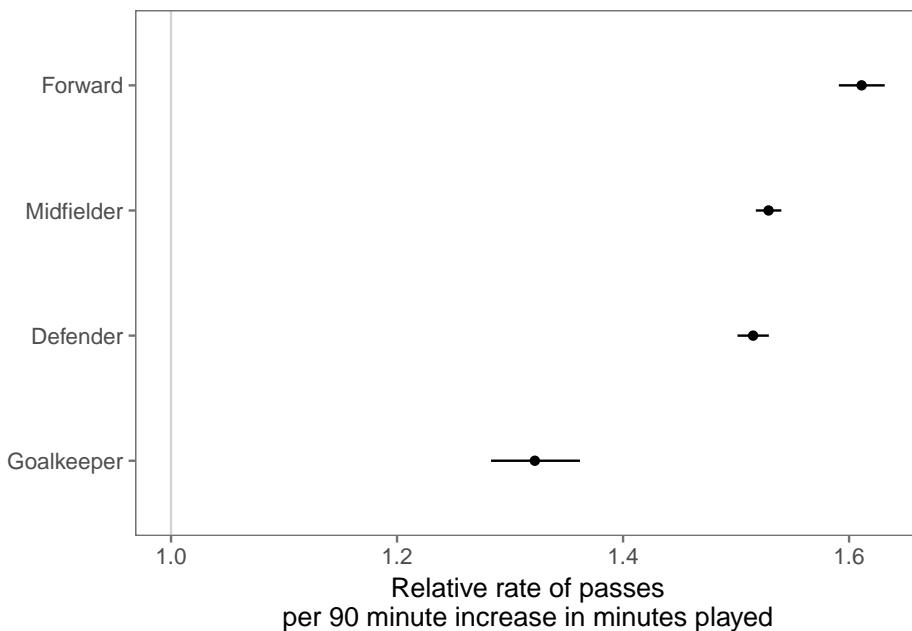
Another way to have done this is with `apply`:

```
means <- apply(worldcup[ , vars], 2, mean)
(means <- round(means, 1))
```

```
##      Time    Shots   Passes Tackles    Saves
##  208.9     2.3    84.5     4.2     0.7
```

However, you can use this same looping process for much more complex tasks that you can't do as easily with `apply` or `dplyr` tools.

Loops can be very useful for more complex repeated tasks. For example:



Creating this graph requires that you:

- Create a subset limited to each of the four positions
- Fit a Poisson regression of Passes on Time within each subset
- Pull the regression coefficient and standard error from each model
- Use those values to calculate 95% confidence intervals
- Convert everything from log relative rate to relative rate
- Plot everything

Create a vector with the names of all positions. Create an empty dataframe to store regression results:

```
(positions <- unique(worldcup$Position))

## [1] Midfielder Defender Forward Goalkeeper
## Levels: Defender Forward Goalkeeper Midfielder

(pos_est <- data.frame(position = positions,
                        est = NA, se = NA))

##      position est se
## 1 Midfielder   NA NA
## 2   Defender   NA NA
## 3    Forward   NA NA
## 4 Goalkeeper   NA NA
```

Loop through and fit a Poisson regression model for each subset of data. Save regression coefficients in the empty dataframe:

```
for(i in 1:nrow(pos_est)){
  pos_df <- worldcup %>%
    filter(Position == positions[i])
  pos_mod <- glm(Passes ~ Time,
                  data = pos_df,
                  family = poisson(link = "log"))
  pos_coefs <- summary(pos_mod)$coefficients[2, 1:2]
  pos_est[i, c("est", "se")] <- pos_coefs
}
pos_est[1:2, ]
```

## position est se
## 1 Midfielder 0.004716096 4.185925e-05
## 2 Defender 0.004616260 5.192736e-05

Calculate 95% confidence intervals for log relative risk values:

```
pos_est <- pos_est %>%
  mutate(lower_ci = est - 1.96 * se,
        upper_ci = est + 1.96 * se)

pos_est %>%
  select(position, est, lower_ci, upper_ci)

##      position      est      lower_ci      upper_ci
## 1 Midfielder 0.004716096 0.004634052 0.004798140
## 2   Defender 0.004616260 0.004514483 0.004718038
```

```
## 3    Forward 0.005299009 0.005158945 0.005439074
## 4 Goalkeeper 0.003101124 0.002770562 0.003431687
```

Calculate relative risk per 90 minute increase in minutes played:

```
pos_est <- pos_est %>%
  mutate(rr_est = exp(90 * est),
        rr_low = exp(90 * lower_ci),
        rr_high = exp(90 * upper_ci))
pos_est %>%
  select(position, rr_est, rr_low, rr_high)
```

```
##      position    rr_est    rr_low    rr_high
## 1 Midfielder 1.528747 1.517501 1.540077
## 2 Defender   1.515073 1.501258 1.529015
## 3 Forward    1.611090 1.590908 1.631527
## 4 Goalkeeper 1.321941 1.283192 1.361861
```

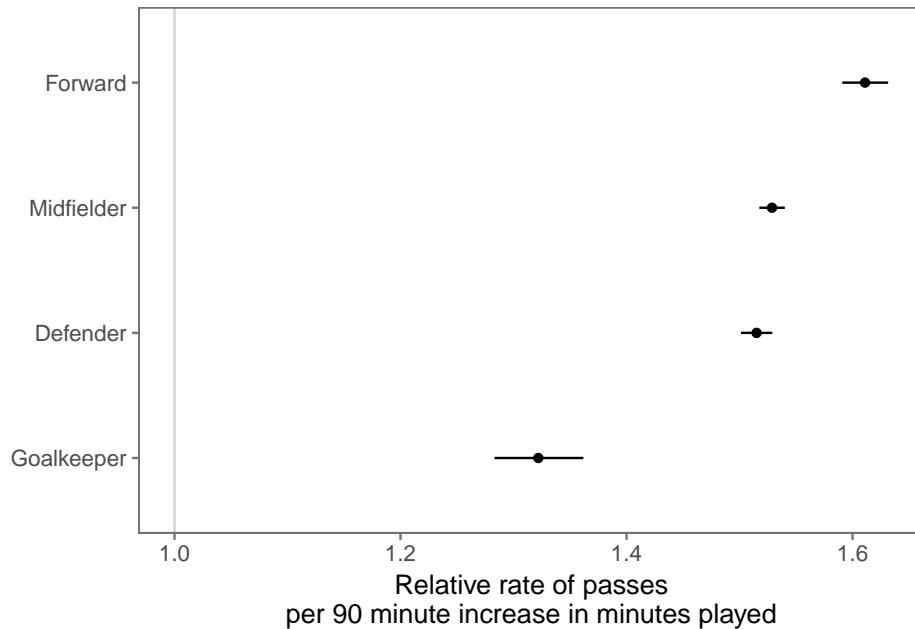
Re-level the position factor so the plot will be ordered from highest to lowest estimates:

```
pos_est <- arrange(pos_est, rr_est) %>%
  mutate(position = factor(position,
                            levels = position))
pos_est %>% select(position, est)
```

```
##      position      est
## 1 Goalkeeper 0.003101124
## 2 Defender   0.004616260
## 3 Midfielder 0.004716096
## 4 Forward    0.005299009
```

Create the plot:

```
ggplot(pos_est, aes(x = rr_low, y = position)) +
  geom_segment(aes(xend = rr_high, yend = position)) +
  geom_point(aes(x = rr_est, y = position)) +
  theme_few() +
  ylab("") +
  scale_x_continuous(paste("Relative rate of passes\nper",
                           "90 minute increase in minutes played"),
                     limits = c(1.0, max(pos_est$rr_high))) +
  geom_vline(aes(xintercept = 1), color = "lightgray")
```



## 7.3 Other control structures

### 7.3.1 if / else loops

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`).

### 7.3.2 Some other control structures

The control structures you are most likely to use in data analysis with R are “for” loops and “if / else” statements. 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:

```
for(i in 1:5){
  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 3, so it will only print the numbers 1 through 3:

```
for(i in 1:5){
  if(i > 3){
    break
  }
  print(i)
}

## [1] 1
## [1] 2
## [1] 3
```

```
my_sum <- 1
while(my_sum < 10){
  my_sum <- my_sum * 2
  print(my_sum)
}
```

```
## [1] 2
## [1] 4
## [1] 8
## [1] 16
```

## 7.4 Functions

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
- There are ways to check for errors in the arguments a user inputs to 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_impl(.data, quo): Evaluation error: argument "child_id" is missing, with no de
```

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
```

The `return` function can also be used to return an object other than the last defined object (although doesn't tend to be something you need to do very often). For example, 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

```

You can use `stop` to stop execution of the function and give the user an error message. For example, the `subset_nepali` function will fail if the user inputs a data frame that does not have a column named “id”:

```

subset_nepali(datafr = data.frame(wt = rnorm(10)),
              child_id = "12011")

```

```
Error: comparison (1) is possible only for
atomic and list types
```

You can rewrite the function to stop if the input `datafr` does not have a column named “id”:

```

subset_nepali <- function(datafr = nepali, child_id){
  if(!("id" %in% colnames(datafr))){
    stop("`datafr` must include a column named `id`")
  }
  datafr <- datafr %>%
    filter(id == child_id)
  return(datafr)
}

```

```
subset_nepali(datafr = data.frame(wt = rnorm(10)),
               child_id = "12011")
```

```
Error in subset_nepali(datafr = data.frame(wt = rnorm(10)),
                       child_id = "12011") :
  `datafr` must include a column named `id`
```

The `stop` function is particularly important if the function would keep running with the wrong input, but would result in the wrong output.

You can also output warnings and messages using the functions `warning` and `message`.

## 7.5 Regular expressions

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)
```

```
## # A tibble: 3 x 1
##   Name
##   <chr>
## 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 = ", ")
```

```
## # A tibble: 3 x 2
##   last_name           first_name
##   <chr>                 <chr>
## 1 Braund                Mr. Owen Harris
## 2 Cumings Mrs. John Bradley (Florence Briggs Thayer)
## 3 Heikkinen              Miss. Laina
```

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)

## # A tibble: 3 x 1
##   Name
##   <chr>
```

```
## 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 `grep1` that does something very similar (although note that the order of the arguments is reversed).

```
titanic_train %>%
  filter(grep1("Mrs.", Name)) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##   Name
##   <chr>
## 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)
```

```
## # A tibble: 3 x 2
##   Name      mrs
##   <chr>     <chr>
## 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)
```

```
## # A tibble: 3 x 2
##   Name     title
##   <chr>   <chr>
## 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 `"."`.

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)
```

```
## # A tibble: 3 x 2
##   Name     title
##   <chr>   <chr>
## 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) %>%
```

```
select(Name, title) %>%
  slice(1:3)

## # A tibble: 3 x 2
##   Name      title
##   <chr>    <chr>
## 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)

## # A tibble: 3 x 1
##   title
##   <chr>
## 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."
## [2,] "Mrs."
## [3,] "Miss."
## [4,] "Mrs."
## [5,] "Mr."
## [6,] "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)

## # A tibble: 3 x 2
##   Name           title
##   <chr>          <chr>
## 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 (“ $\wedge$ ”) the letter “A”:

```
titanic_train %>%
  filter(str_detect(Name, "^A")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##   Name
##   <chr>
## 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)
```

```
## # A tibble: 2 x 1
##   Name
##   <chr>
## 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 (“ $\wedge$ ” outside of brackets) the letters “A” and “B”:

```
titanic_train %>%
  filter(str_detect(Name, "^[AB]")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##   Name
##   <chr>
## 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                  |

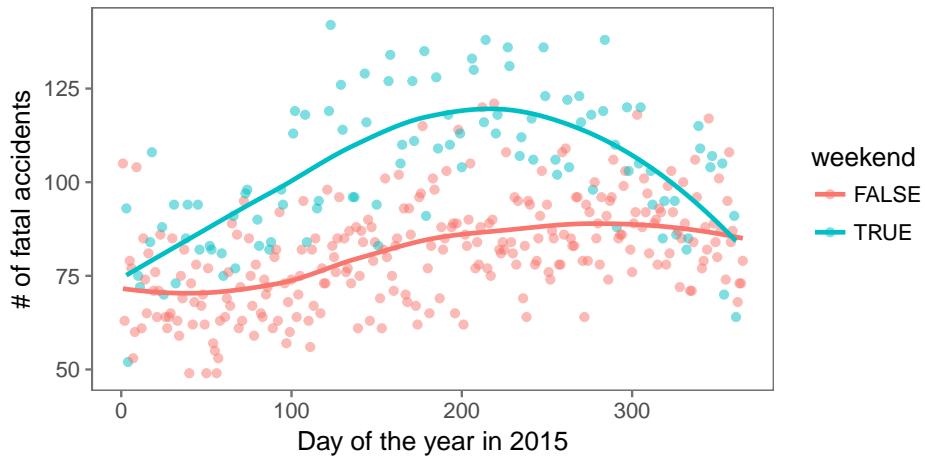
For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Introduction to `stringr` by Hadley Wickham
- Handling and Processing Strings in R by Gaston Sanchez (seven chapter ebook)
- <http://gskinner.com/RegExr> and <http://www.txt2re.com>: Interactive tools for helping you build regular expression pattern strings

## 7.6 In-course exercise

### 7.6.1 Exploring Fatality Analysis Reporting System (FARS) data

- Visit <http://metrocosm.com/10-years-of-traffic-accidents-mapped.html> and explore the interactive visualization created by Max Galka using this public dataset on US fatal motor vehicle accidents.
- Go to FARS web page to find and download the data for this week's exercise. We want to get the raw data on fatal accidents. Navigate this page to figure out how you can get this raw data for the whole county for 2015. Download the 2015 data to your computer. What is the structure of how this data is saved (e.g., directory structure, file structure)?
- On the FARS web page, find the documentation describing this raw data. Look through both this documentation and the raw files you downloaded to figure out what information is included in the data.
- Read the `accident.csv` file for 2015 into R (this is one of the files you'll get if you download the raw data for 2015). Use the documentation to figure out what each column represents.
- 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. (Hint: Functions from the `lubridate` package will be very helpful, including `yday` and `wday`).
- Discuss which of the variables in this dataset could be used to merge the dataset with other appropriate data, either other datasets in the FARS raw data, or outside datasets.
- Try to write the code to create the plot below. This will include some code for cleaning the data and some code for plotting. There is an example answer below, but I'd like you to try to figure it out yourselves first.



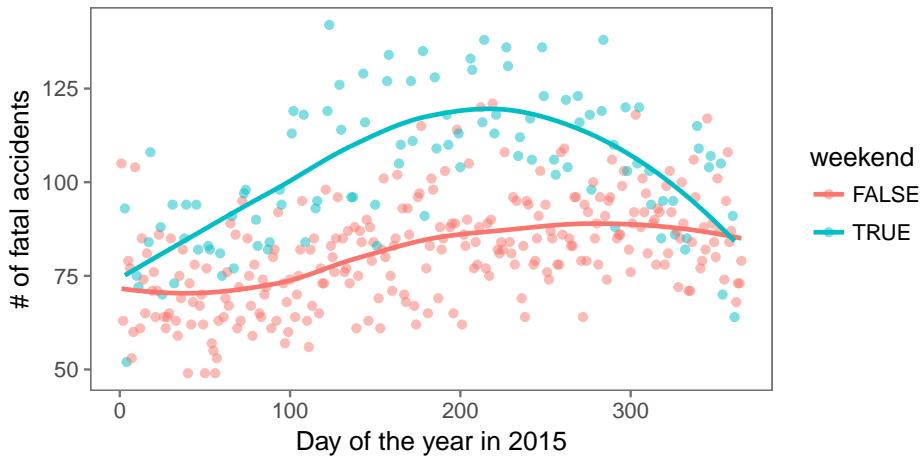
### 7.6.1.1 Example R code

Here is example code for the section above:

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

accident <- read_csv("data/accident.csv") %>%
  select(DAY:MINUTE) %>%
  select(-DAY_WEEK) %>%
  unite(date, DAY:MINUTE, sep = "-", remove = FALSE) %>%
  mutate(date = dmy_hm(date),
        yday = yday(date),
        weekday = wday(date, label = TRUE, abbr = FALSE),
        weekend = weekday %in% c("Saturday", "Sunday"))

accident %>%
  filter(!is.na(yday)) %>%
  group_by(yday) %>%
  summarize(accidents = n(),
            weekend = first(weekend)) %>%
  ggplot(aes(x = yday, y = accidents, color = weekend)) +
  geom_point(alpha = 0.5) +
  xlab("Day of the year in 2015") +
  ylab("# of fatal accidents") +
  theme_few() +
  geom_smooth(se = FALSE)
```



### 7.6.2 Using a loop to create state-specific plots

Next, you will write a loop to write a four-page pdf file with state-specific plots for the states of Colorado, Texas, California, and New York.

- The data has a column called `STATE`, but it gives state as a one- or two-digit code, rather than by name. These codes are the state Federal Information Processing Standard (FIPS) codes. A dataset with state names and FIPS codes is available at <http://www2.census.gov/geo/docs/reference/state.txt>. Read that data into an R object called `state_fips` and clean it so the first few lines look like this:

```
## # A tibble: 5 x 2
##   state state_name
##   <int>     <chr>
## 1     1    Alabama
## 2     2     Alaska
## 3     4    Arizona
## 4     5   Arkansas
## 5     6  California
```

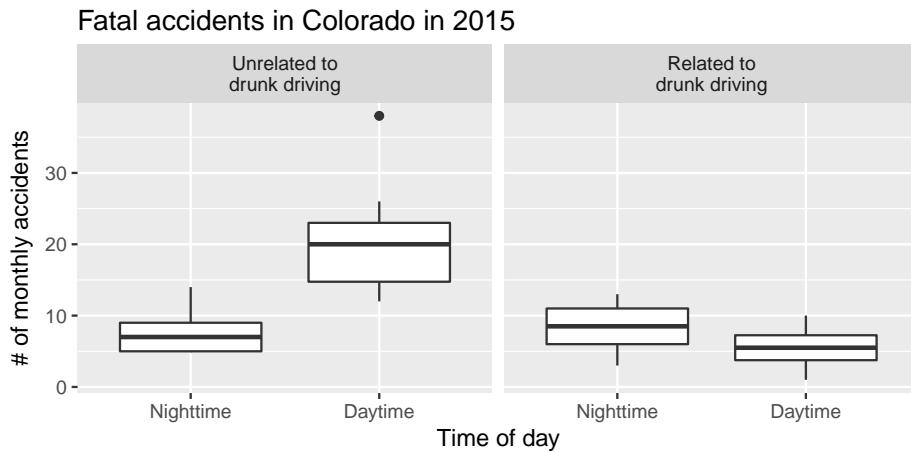
- Read the 2015 FARS data into an R object named `accident`. Use all the date and time information to create a column named `date` with the date and time of the accident. Include information on whether the accident was related to drunk driving (FALSE if there were 0 drunk drivers, TRUE if there were one or more), and create columns that give whether the accident was during the day (7 AM to 7 PM) or not as well as the month of the accident (for this last column, you can either retain it from the original data or recalculate it based on the new `date` variable). Filter out any values where the date-time does not render (i.e., `date` is a missing value). The first few rows of the cleaned dataframe should look like this:

```
## # A tibble: 5 x 5
##   state           date  drunk_dr daytime month
##   <int>       <dttm>    <lgl>    <lgl> <dbl>
## 1     1 2015-01-01 02:40:00     TRUE   FALSE     1
## 2     1 2015-01-01 22:13:00    FALSE   FALSE     1
## 3     1 2015-01-01 01:25:00     TRUE   FALSE     1
## 4     1 2015-01-04 00:57:00     TRUE   FALSE     1
## 5     1 2015-01-07 07:09:00    FALSE    TRUE     1
```

- Join the information from `state_fips` into the accident dataframe. There may be a few locations in the `state_fips` dataframe that are not included in the `accident` dataframe (e.g., Virgin Islands), so when you join keep all observations in `accident` but only the observations in `state_fips` that match at least one row of `accident`. The first few rows of the joined dataset should look like this:

```
## # A tibble: 5 x 6
##   state           date  drunk_dr daytime month state_name
##   <int>       <dttm>    <lgl>    <lgl> <dbl>      <chr>
## 1     1 2015-01-01 02:40:00     TRUE   FALSE     1  Alabama
## 2     1 2015-01-01 22:13:00    FALSE   FALSE     1  Alabama
## 3     1 2015-01-01 01:25:00     TRUE   FALSE     1  Alabama
## 4     1 2015-01-04 00:57:00     TRUE   FALSE     1  Alabama
## 5     1 2015-01-07 07:09:00    FALSE    TRUE     1  Alabama
```

- Write the code to create boxplots for Colorado of the distribution of total accidents within each month. Create separate boxplots for daytime and nighttime accidents, and facet by whether the accident was related to drunk driving. The plot should look like the plot below.



- Write a loop to create a plot like the one you just made for Colorado for four different states – Colorado, Texas, California, and New York. The

code should write the plots out to a pdf, with one plot per page of the pdf.  
*(Hint:* To get a loop to print out a plot created with ggplot, you must explicitly print the plot object. For example, you could assign the plot to a, and then you would run print(a) within your loop as the last step.)

### 7.6.2.1 Example R code

Here is example R code for creating state-specific plots using a loop:

```
state_fips <- read_delim("http://www2.census.gov/geo/docs/reference/state.txt",
  delim = "|") %>%
  rename(state = STATE,
         state_name = STATE_NAME) %>%
  select(state, state_name) %>%
  mutate(state = as.integer(state))

accident <- read_csv("data/accident.csv") %>%
  select(STATE, DAY:MINUTE, DRUNK_DR) %>%
  rename(state = STATE,
         drunk_dr = DRUNK_DR) %>%
  select(-DAY_WEEK) %>%
  unite(date, DAY:MINUTE, sep = "-") %>%
  mutate(date = dmy_hm(date),
         drunk_dr = drunk_dr >= 1,
         daytime = hour(date) %in% c(7:19),
         month = month(date)) %>%
  filter(!is.na(date)) %>%
  left_join(state_fips, by = "state")

pdf("StateFARSPlots.pdf", height = 3.5, width = 7)
state_list <- c("Colorado", "Texas", "California", "New York")
for(which_state in state_list){
  a <- accident %>%
    filter(state_name == which_state) %>%
    mutate(drunk_dr = factor(drunk_dr, labels =
      c("Unrelated to\ndrunk driving",
        "Related to\ndrunk driving")),
      daytime = factor(daytime, labels = c("Nighttime", "Daytime"))) %>%
    group_by(daytime, month, drunk_dr) %>%
    summarize(accidents = n()) %>%
    ggplot(aes(x = daytime, y = accidents, group = daytime)) +
    geom_boxplot() +
    facet_wrap(~ drunk_dr, ncol = 2) +
    xlab("Time of day") + ylab("# of monthly accidents") +
```

```
    ggtile(paste("Fatal accidents in", which_state, "in 2015"))
  print(a)
}
dev.off()
```

## Appendix A

# Appendix A: Vocabulary

You will be responsible for knowing the following functions and vocabulary for the weekly quizzes.

### A.1 Week 1 (Quiz 1)

- `c()`
- `data.frame()`
- `dim()`
- `ncol()`
- `nrow()`
- `head()`, option `n =`
- `read.csv`, options `head =`, `skip =`, `nrow =`
- `[...], [..., ...]`
- `getwd()`
- `setwd()`, including `setwd("~/")`
- `list.files()`
- `install.packages()`
- `library()`
- `<-`
- `=`
- `subset()`
- `length()`
- open source software
- “free as in beer”
- “free as in speech”
- CRAN
- GitHub
- R packages

- R working directory
- How to download a csv file from GitHub
- Nate Silver
- FiveThirtyEight
- Grading policies for the course
- Course requirements / policies for in-class quizzes and weekly journal entries
- Style rules for naming R objects
- Difference between R and RStudio
- Vectors
- Dataframes
- Note: Pay attention in the course notes and exercise to where the code uses quotation marks and where it does not– this will help you in the quiz

## A.2 Week 2 (Quiz 2)

- `source()`
- `setwd()`, including `setwd("~/")`, `setwd("../")`, `setwd("../\..")`
- `list.files()`, option `path =`
- functions in the `read.table()` family, including `read.csv()` and `read.delim()`. What are defaults of the `sep =` and `dec =` options for each? For all, the options `header =`, `sep =`, `as.is =`, `na.strings =`, `nrows =`, `skip =`, and `col.names =`.
- The tidyverse
- functions in the `read_*` family (e.g., `read_csv`)
- Advantages of the `read_*` family of functions compared to their base R analogues (the `read.table` functions)
- `paste()`, option `sep =`
- `paste0()`
- `readxl` package and its `read_excel()` function
- `haven` package and its `read_sas()` function
- `$`
- `class()`
- `str()`
- `as.Date()`, option `format =`
- `lubridate` functions, include `ymd`, `ymd_hm`, and `mdy`
- `range()`
- `dplyr` package
- `rename()`
- `mutate()`
- `arrange()`
- `%>%`, advantages of piping
- `filter()`
- Reading in data from either a local or online flat file

- `save()`, option `file` =
- `load()`
- `rm()`
- `ls()`
- Main types of vector classes in R: character, numeric, factor, date, logical
- Which classes of vectors don't always look like numbers, but R assigns an underlying numeric value to? (Hint: This include the logical class, which R saves with an underlying number, with TRUE = 1 and FALSE = 0.)
- Common abbreviations for telling R date formats (e.g., "%m", "%y")
- Common logical expressions to use in `filter()`
- relative pathnames
- absolute pathnames
- delimited files
- fixed width files
- R script file (How would you make a new one? What file extension would it have? Why is it important to use? How do you run code from a script file in RStudio?)
- What kinds of data can be read into R?
- How to read flat files of data that are online directly into R if they are on:
  - A "http:" site
  - A "https:" site
- When you might want to save an R object as a `.RData` file and when (and why) you might not want to

## A.3 Week 3 (Quiz 3)

- `data()` (with and without the name of a dataset as an option)
- `library()` (with and without an argument in the parentheses)
- common addition for all these plotting functions: `ggtitle`, `xlab`, `ylab`, `xlim`, `ylim`
- `aes` function and common aesthetics, including `color`, `shape`, `x`, `y`, and `fill`
- Some common geoms: `geom_histogram`, `geom_points`, `geom_lines`, `geom_boxplot()` (both for a single numeric variable and for a numeric vector stratified by a factor)
- `ggpairs()` from the `GGally` package
- `range()`
- `min()`
- `max()`
- `mean()`
- `median()`
- `table()`
- `cor()`, both for two variables in a dataframe, and to get the correlation matrix for several variables in a dataframe

- `summary()`, as applied to: different classes of vectors (numeric, factor, logical), dataframes, `lm` objects, and `glm` objects
- `lm()`, `data=` option
- `glm()`, options `data=`, `family=`
- Functions to apply to a `lm` or `glm` object: `summary()`, `coef()`, `residuals()`, `fitted()`, `plot()`, `abline()`
- The following elements that you can pull from the `summary` of a `lm` call:  
`summary(mod_1)$call`, `summary(mod_1)$coef`, `summary(mod_1)$r.squared`,  
`summary(mod_1)$cov.unscaled`
- How to create a logical vector and how to use one to (1) index a data frame and (2) count the number of times a certain condition is true in a vector
- What the bang operator (!) does to a logical operator
- What to do if you want to apply a summary statistic function to a vector with missing values (you do not need to know every option name for all the functions, just know that you would need to include an option like `na.rm=` or `use=`, and that you can use the help file for a function to figure out the option call for that function).
- The following about object-oriented programming: In R, it means that some functions, like `summary()`, will do different things depending on what type of object you call it on.
- The basic structure of regression formulae in R (for example, `y ~ x1 + x2`)
- Difference between using `lm()` and `glm()` to fit a linear regression model
- Difference between the code you would use to fit a linear, Poisson, or logistic model using `glm()`

## A.4 Week 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 Week 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 Week 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)
- `select`
- `filter`
- `mutate`
- `summarize`
- `group_by`
- `arrange`
- `gather`

- `spread`
- The `*_join` family of functions
- `%>%`
- Go through the examples from in-course exercises where we chained together several functions to clean up a dataset and make sure you can follow through these chained examples

## A.7 Week 7 (Quiz 7)

- `for` loops
- basics of writing a function
- figuring out the output of a loop based on its code
- figuring the the output of a function based on its code
- parentheses around a full assignment statement (e.g., `(ex <- 1)`)
- `kable()` from the `knitr` package

## A.8 Week 8 (Quiz 8)

- `apply` family of functions
- `matrix` objects, including how to subset
- `list` objects, including how to subset

## Appendix B

# Appendix B: Homework

The following are six homework assignments for the course.

### B.1 Homework #1

**Due date: Sept. 13**

For your first 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** course (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.

To begin, you'll first need to install the swirl package:

```
install.packages("swirl")
```

Load the swirl package, then 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.

### B.1.1 Option 1

If you've never run `swirl()` before, you will be prompted to install a course. Type 1 to install R Programming: The basics of programming in R. Now, every time you enter the swirl environment with `swirl()`, R Progammimg should show up as a course option to select. You can enter R Programming to start lessons in that course by typing 1 again.

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 is in parentheses):

- Basic Building Blocks (1)
- Vectors (4)
- Missing Values (5)
- Subsetting Vectors (6)
- Logic (8)
- Looking at Data (12)
- Dates and Times (14)

Each lesson should take about 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. Type 2 for no.

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` course. To install a course other than `R Programming`, you can use the following steps:

```
library(swirl)
install_course("Course Name Here")
swirl()
```

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 `Getting and Cleaning Data` course, you would run:

```
library(swirl)
install_course("Getting and Cleaning Data")
swirl()
```

After entering the `Getting and Cleaning Data` course, you could choose from any one of its four 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. In R Programming, the Functions lesson, as well as lapply and sapply and vapply and tapply could be particularly useful. You can also look through the course directory to see what other courses and lessons are available.

## B.2 Homework #2

### Due date: Sept. 27

For Homework 2, recreate the R Markdown document that you can download from here.

Here are some initial tips:

- Your goal is to create an R Markdown document that you can compile to create a Word document that looks just like the example document we've linked above.
- You will turn in (by email) both the compiled Word document and the .Rmd original file.
- Add your name as "Author" and the due date of the assignment as "Date". You should add these within the R Markdown document, rather than changing them in the final, compiled Word document.
- If you want to get started before you know how to use R Markdown, you can go ahead and write all of the necessary code to replicate the output and figures in the document in an R script.
- The code chunks here have been hidden with the option `echo = FALSE`, but you should include your code in your final document.
- Set the chunk options `warning = FALSE` and `message = FALSE` to prevent warnings and messages from being printed out. You will get some messages and warnings in the code from things like missing values and from loading packages, but you want to hide all of those messages in your final document.
- For things like templates, colors, level of transparency, and point size, you will receive full credit if you create figures that are visually similar to the ones shown in the example document. In other words, if the example document shows some transparency in points, you will get full credit if you also include some transparency in the points in your plot, but you do not have to include the exact same value of `alpha`.
- In R, there are often many different ways to achieve something. As long as your code *works*, it's fine if you haven't coded it exactly like we have in our version. However, your output should look identical to ours (or, in the case of color, transparency, point size, and themes, visually similar).

- You will not lose points if you cannot recreate the table in the document (although you should try to!).
- The last section, under the heading “Extra challenge– not graded”, is not graded. However, if you’d like an extra challenge, you’re welcome to try it out and include it in your final submission!

If you need them, here are some further tips:

- Functions from the tidyverse (especially from `dplyr`, `readr`, and `ggplot` packages) will make your life much easier for this exercise. You can now install and load the `tidyverse` package to load them all at once.
- To rename column names with “special” characters in them, wrap the whole old column name in backticks. For example, to change a column name that has a dollar sign in it, you would use something like `“rename(new_col_name = ‘old_col_name$’)`.
- To change the size of a figure in a report, use the “`fig.width`” and “`fig.height`” chunk options.
- You will want to use `scale_fill_brewer` in several of the figures.
- Don’t forget that, within functions like `scale_x_continuous`, you can use the argument `breaks` to set where the axis has breaks, and then `labels` to set what will actually be shown at each break.
- The string “`\n`” can be included in legends and labels to include a carriage return.
- Coordinates can be flipped in a graph with the `coord_flip` geom. So, if you can figure out a way to make a graph with the coordinates flipped, use that code and just add `coord_flip` at the end.

## B.3 Homework #3

**Due date: Oct. 11**

For Homework 3, recreate the R Markdown document that you can download from here.

Here are some initial tips:

- Your goal is to create an R Markdown document that you can compile to create a Word document that looks just like the target document we’ve linked above. The only difference is that you will use `echo = TRUE` to show your code within the rendered Word document. All formating within the text should be similar or identical to the target document.
- You will turn in (by email) both the compiled Word document and the `.Rmd` original file.
- Add your name as “Author” and the due date of the assignment as “Date”. You should add these within the R Markdown document, rather than

changing them in the final, compiled Word document.

- Set the chunk options `warning = FALSE` and `message = FALSE` to prevent warnings and messages from being printed out. You will get some messages and warnings in the code from things like missing values and from loading packages, but you want to hide all of those messages in your final document.
- For things like templates, colors, level of transparency, and point size, you will receive full credit if you create figures that are visually similar to the ones shown in the example document. In other words, if the example document shows some transparency in points, you will get full credit if you also include some transparency in the points in your plot, but you do not have to include the exact same value of `alpha`.
- In R, there are often many different ways to achieve something. As long as your code *works*, it's fine if you haven't coded it exactly like we have in our version. However, your output should look identical to ours (or, in the case of color, transparency, point size, and themes, visually similar).
- There is one formated table in the target document. Be sure that you render this as a formated table, not as raw R output.

If you need them, here are some further tips:

- Functions from the tidyverse (especially from `dplyr`, `readr`, and `ggplot` packages) will make your life much easier for this exercise. You can now install and load the `tidyverse` package to load them all at once.
- To reference column names with “special” characters in them, like dollar signs or spaces, wrap the whole old column name in backticks. For example, to change a column name that has a dollar sign in it, you would use something like `“rename(new_col_name = ‘old_col_name$’)`”.
- To change the size of a figure in a report, use the “`fig.width`” and “`fig.height`” chunk options.
- Don't forget that there are functions in the `scale` family that allow you to use log-scale axes.

## B.4 Homework #4

### Optional due date: Oct. 25

All instructions for this homework can be downloaded here. The example “`fars_analysis.pdf`” document you will try to recreate is here.

You have the option to turn in parts of this homework (up through creating a clean dataset) by Oct. 28. If you do so, I will email you the code I used to clean the data, so you can check your own code and be sure you have a reasonable version of the clean data as you do the final parts of the assignment.

## B.5 Homework #5

**Due date:** Nov. 8

All instructions for this homework can be downloaded here. The example “fars\_analysis.pdf” document you will try to recreate is here.

You will submit this homework by posting a repo with your project directory on GitHub. We will work on setting that up during an in-course exercise.

## B.6 Homework #6

**Due date:** Nov. 29

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” your last homework incorporated. Also list one or two practices that you did not follow in your last homework but that would have made sense and how you could have followed them.
2. Read the article *Science Isn’t Broken* on FiveThirtyEight. This article includes an interactive graphic. In a half page, give your opinion on whether this interactive graphic helps convey the main message of the article. Also, describe in general details how you might be able to create a graphic like this in R.
3. 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.



# Bibliography

