R Programming for Research

Colorado State University, ERHS 535

Brooke Anderson, Rachel Severson, and Nicholas Good

2019-08-26

Contents

Online course book, ERHS 535

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

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

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

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

0.2 Time and place

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

For the first five weeks of class, the course meets in the first-floor classroom of the Military Sciences building on Mondays and Wednesdays, 10:00 am–11:50 am. For the remaining weeks, the 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. 2).
- There are no course meetings the week of Thanksgiving (week of Nov. 25).
- Office hours will be 10:00–11:00 AM on Fridays in EH 120.

0.3 Detailed schedule

Here is a more detailed view of the schedule for this course for Fall 2019:

Week	Class dates	Level	Lecture content	Graded items
1	Aug. 26, 28	Preliminary	R Preliminaries	
2	Sept. 4	Basic	Entering and cleaning data	Quiz (W)
3	Sept. 9, 11	Basic	Exploring data	Quiz (W), HW #1 (F)
4	Sept. 16, 18	Basic	Reporting data results	Quiz (W)
5	Sept. 23, 25	Basic	Reproducible Research	Quiz (W), HW #2 (F)
6	Sept. 30, Oct. 2	Intermediate	Entering and cleaning data	Quiz (W)
7	Oct. 7, 9	Intermediate	Exploring data	Quiz (W)
8	Oct. 14, 16	Intermediate	Reporting data results	Quiz (W), HW #3 (W)
9	Oct. 21, 23	Intermediate	Reproducible Research	Quiz (W)
10	Oct. 28, 30	Advanced	Entering and cleaning data	Quiz (W), HW #4 (F)
11	Nov. 4, 6	Advanced	Exploring data	
12	Nov. 11, 13	Advanced	Exploring data (mapping)	HW #5 (F)
13	Nov. 18, 20	Advanced	Reporting data results	
14	Dec. 2, 4	Advanced	Reproducible Research	HW #6 (F)
15	Dec. 9, 11	Advanced	Continuing in R	Project draft (M)
16	Week of Dec. 16		Group presentations	Final project

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

0.4 Grading

0.4.1 Grading for ERHS 535

For ERHS 535, course grades will be determined by the following five components:

Assessment component	Percent of grade
Final group project	30
Weekly in-class quizzes, weeks 2-10	25
Six homework assignments	25
Attendance and class participation	10
Weekly in-course group exercises	10

0.4.2 Grading for ERHS 581A3

For ERHS 581A3, course grades will be determined by the following four components:

Assessment component	Percent of grade
Weekly in-class quizzes, weeks 2-5	40
Two homework assignments	30
Attendance and class participation	10
Weekly in-course group exercises	20

0.4.3 Grading for ERHS 581A4

For ERHS 581A4, course grades will be determined by the following five components:

Assessment component	Percent of grade
Final group project	30
Weekly in-class quizzes, weeks 1–5 (weeks 6–10 of the semester)	25
Four homework assignments	30
Attendance and class participation	5
Weekly in-course group exercises	10

0.4.4 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.

If you are in **ERHS 535**, out of a possible 10 points for class attendance, you will get:

- **10 points** if you miss two or fewer classes
- **8 points** if you miss three classes
- **6 points** if you miss four classes
- **4 points** if you miss five classes
- **2 points** if you miss six classes
- **0 points** if you miss seven or more classes

If you are in **ERHS 581A3** or **ERHS581A4**, out of a possible 10 points for class attendance, you will get:

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

Exceptions:

- Attendance on the first day of class (Aug. 26) 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 provide a signed letter from your research adviser. For more details, see CSU's Academic Policies on Course Attendance.
- If you have to miss class for a serious medical issue (e.g., operation, sickness severe enough to require a doctor's visit), the absence will be excused if you bring in a note from a doctor of other medical professional giving the date you missed and that it was for a serious medical issue.

For an absence to be excused, you must email me a copy of the letter by 5:00 pm the Friday afternoon of the week of the class you missed.

0.4.5 Weekly in-course group exercises

Part of each class will be spent doing in-course group 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 a few paragraphs describing what was covered in the exercise and what you learned. To get credit for this, you must submit it to me by email by 5:00 pm the Friday afternoon of the week of the class you missed.

All in-class exercises are included in the online course book at the end of the chapter on the associated material.

0.4.6 In-class quizzes

There will be weekly in-course quizzes for weeks 2–10 of the course. Students in ERHS 535 will take all these quizzes. Students in ERHS 581A3 will take quizzes in weeks 2–5. Students in ERHS 581A4 will take quizzes in weeks 6–10.

Each quiz will have at least 10 questions. Typically, a quiz will have more questions, usually 12–15 questions. The grading of the quizzes is structures so that you can get full credit for the quiz portion of the grade without getting 100% of quiz questions right. Instead, if you get ten questions right per quiz on average, you will get full credit for the quiz portion of the grade.

Once you reach the maximum possible points on quizzes, you can continue to take the quizzes for practice, or you can choose to skip any following quizzes.

Quiz questions will be multiple choice, matching, or very short answers. 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. The final version of the Vocabulary list you will be responsible will be posted by the Wednesday evening before each quiz. In general, using R frequently in your research or other coursework will help you to prepare and do well on these quizzes.

Except in very unusual situations, the only time you will be able to make up a quiz is during office hours of the same week when you missed the quiz. Note that 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.

0.4.6.1 Quiz grade calculations for ERHS 535

For students in ERHS 581A3, the **nine quizzes** in weeks 2–10 count for **25 points** of the final grade. The final quiz total for students in ERHS 535 will be calculated as:

$$\text{Quiz grade} = 25 * \frac{\text{Number of correct quiz answers}}{90}$$

0.4.6.2 Quiz grade calculations for ERHS 581A3

For students in ERHS 581A3, the **four quizzes** in weeks 2–5 count for **40 points** of the final grade. The final quiz total for students in ERHS 581A3 will be calculated as:

$$\text{Quiz grade} = 40 * \frac{\text{Number of correct quiz answers}}{40}$$

0.4.6.3 Quiz grade calculations for ERHS 581A4

For students in ERHS 581A4, the **five quizzes** in weeks 6–10 count for **25 points** of the final grade. The final quiz total for students in ERHS 581A3 will be calculated as:

$$\text{Quiz grade} = 25 * \frac{\text{Number of correct quiz answers}}{50}$$

0.4.7 Homework

There will be homework assignments due every two to three weeks during the course, starting the third week of the course (see the detailed schedule in the online course book for exact due dates).

The first two homeworks (HWs #1 and # 2) should be done individually. For later homeworks, you may be given the option to work in small groups of approximately three students.

Homeworks will be graded for correctness, but some partial credit will be given for questions you try but fail to answer correctly. Some of the exercises will not have "correct" answers, but instead will be graded on completeness.

For later homeworks, a subset of the full set of questions will be selected for which I will do a detailed grading of the code itself, with substantial feedback on coding. All other questions in the homework will be graded for completeness and based on the final answer produced.

Homework is due to me by email by 5:00 pm on the Friday it is due. 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.

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

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

Here are useful links for this set-up:

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

0.6 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 later weeks, which is currently in a rawer draft than the beginning of the book). We typically cover about a chapter of the book each week of the course.

This coursebook includes:

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

0.6.1 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 Objectives

After this chapter, you should:

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

1.2 R and R Studio

1.2.1 What is R?

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

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

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

- Statistics
- Machine learning
- Data analysis

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

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

1.2.2 Free and open-source software

> "Life is too short to run proprietary software." – Bdale Garbee

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

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

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

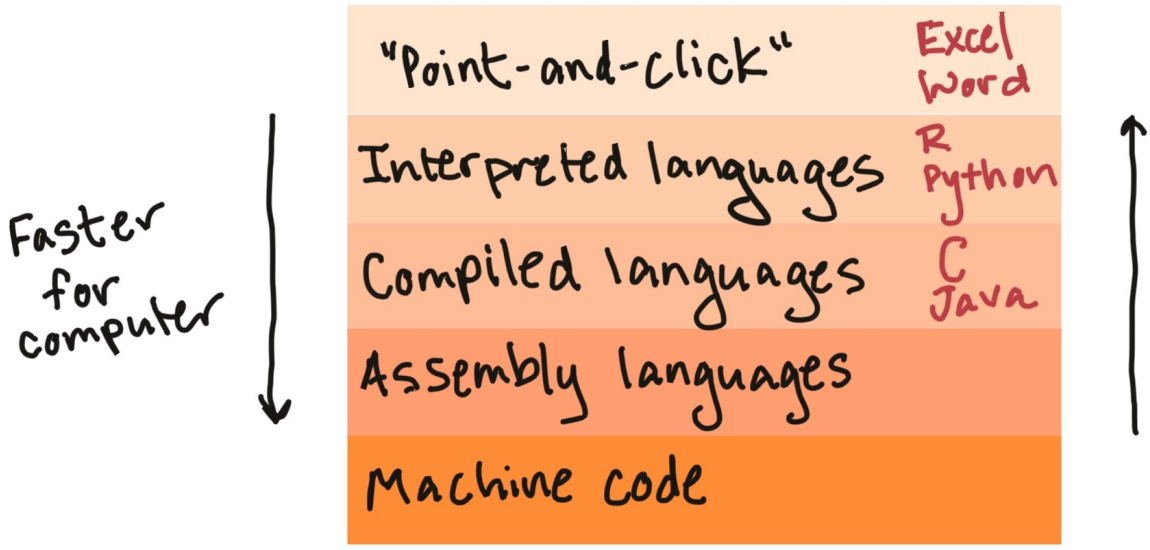


Figure 1.1: Broad types of software programs. R is an interpreted language. 'Point-and-click' programs, like Excel and Word, are often easiest for a new user to get started with, but are slower for the computer and are restricted in the functionality they offer. By contrast, compiled languages (like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

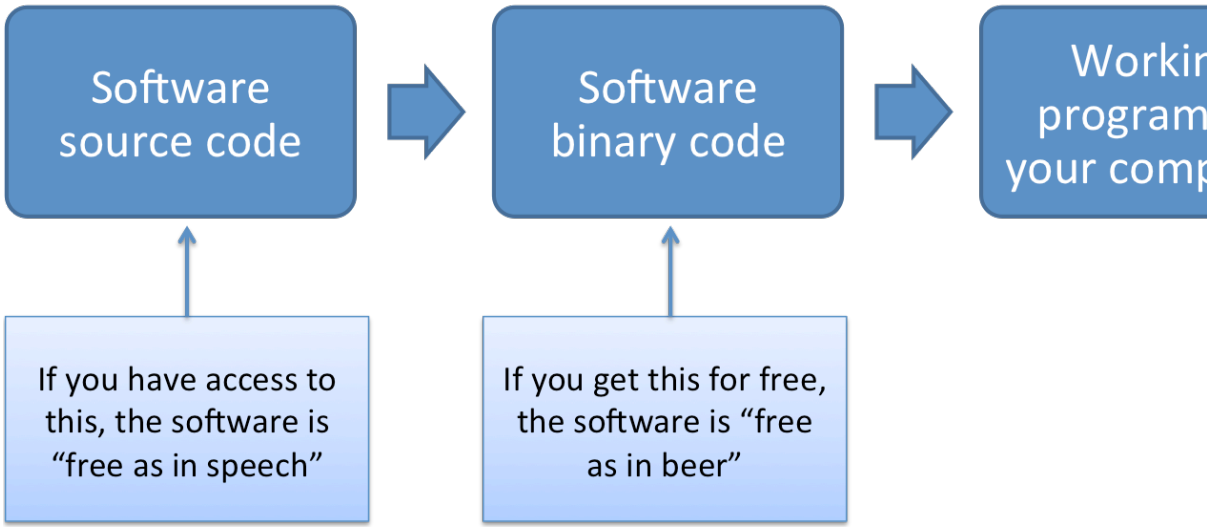


Figure 1.2: An overview of how software can be each type of free (beer and speech). For software programs developed using a compiled programming language, the final product that you open on your computer is run by machine-readable binary code. A developer can give you this code for free (as in beer)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

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

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

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

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

> "Making Linux GPL'd was definitely the best thing I ever did." – Linus Torvalds

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

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

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

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

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

> "Talk is cheap. Show me the code." - Linus Torvalds

1.2.3 What is RStudio?

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

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

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

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

- Develops and freely provides the RStudio IDE
- Provides excellent resources for learning and using R (e.g., cheatsheets, free online books)

- Is producing some of the most-used R packages
- Employs some of the top people in R development
- Is a key member of The R Consortium (others include Microsoft, IBM, and Google)

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

1.3 Communicating with R

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

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

1.3.1 R sessions, the console, and the command prompt

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

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

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

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

1.3.2 R expressions, function calls, and objects

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

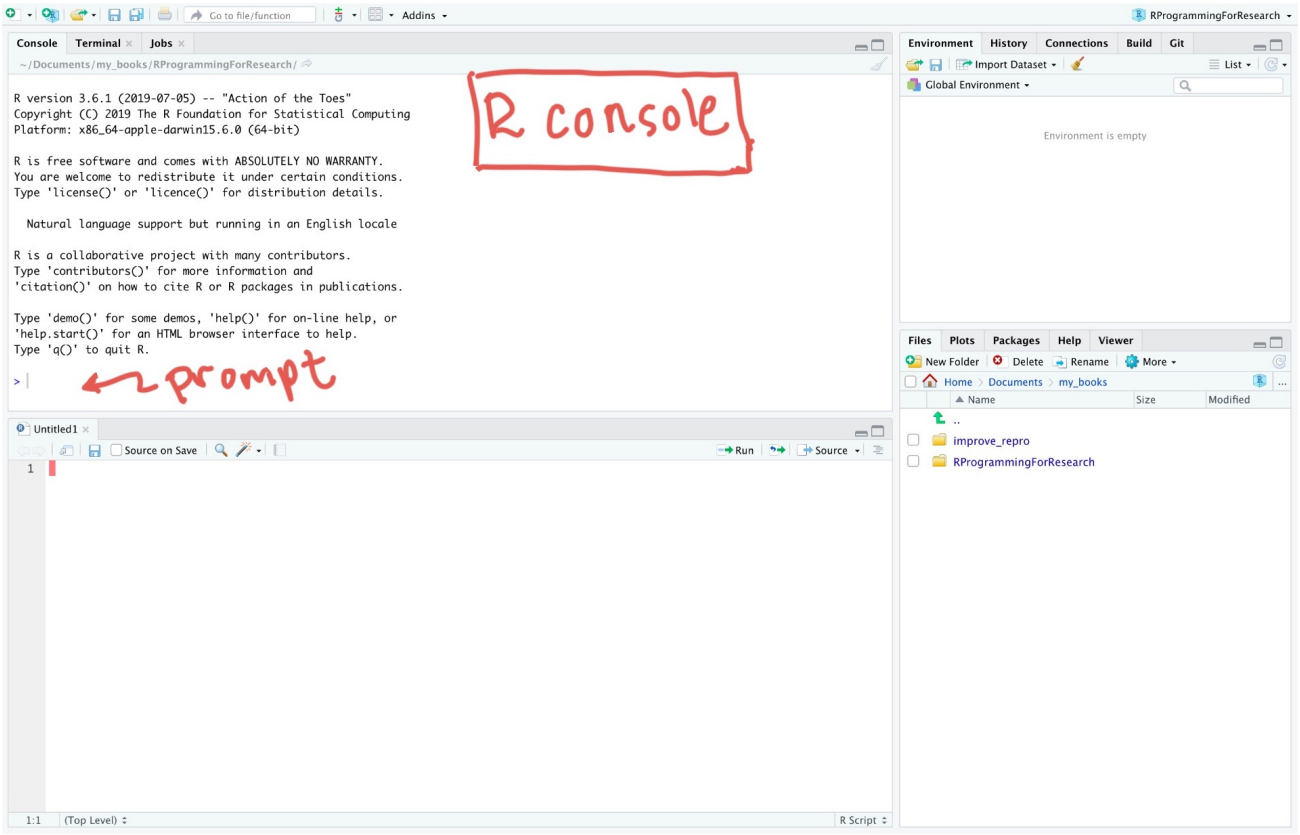


Figure 1.3: Finding the 'Console' pane and the command prompt in RStudio.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

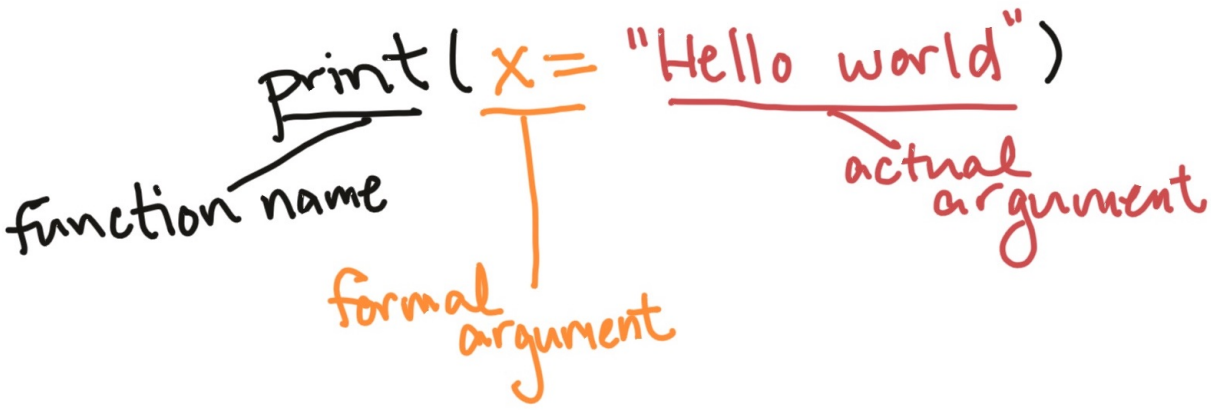


Figure 1.4: Main parts of a function call. This example is calling a function with the name 'print'. The function call has one argument, with a formal argument of 'x', which in this call is provided the named argument 'Hello world'.

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

```
getRversion()
```

```
## [1] '3.6.1'
```

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

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

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

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

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

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

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

```
print()
```

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

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

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

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

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

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

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

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

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

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

```
?mean
```

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

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

```
2 + 3
```

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

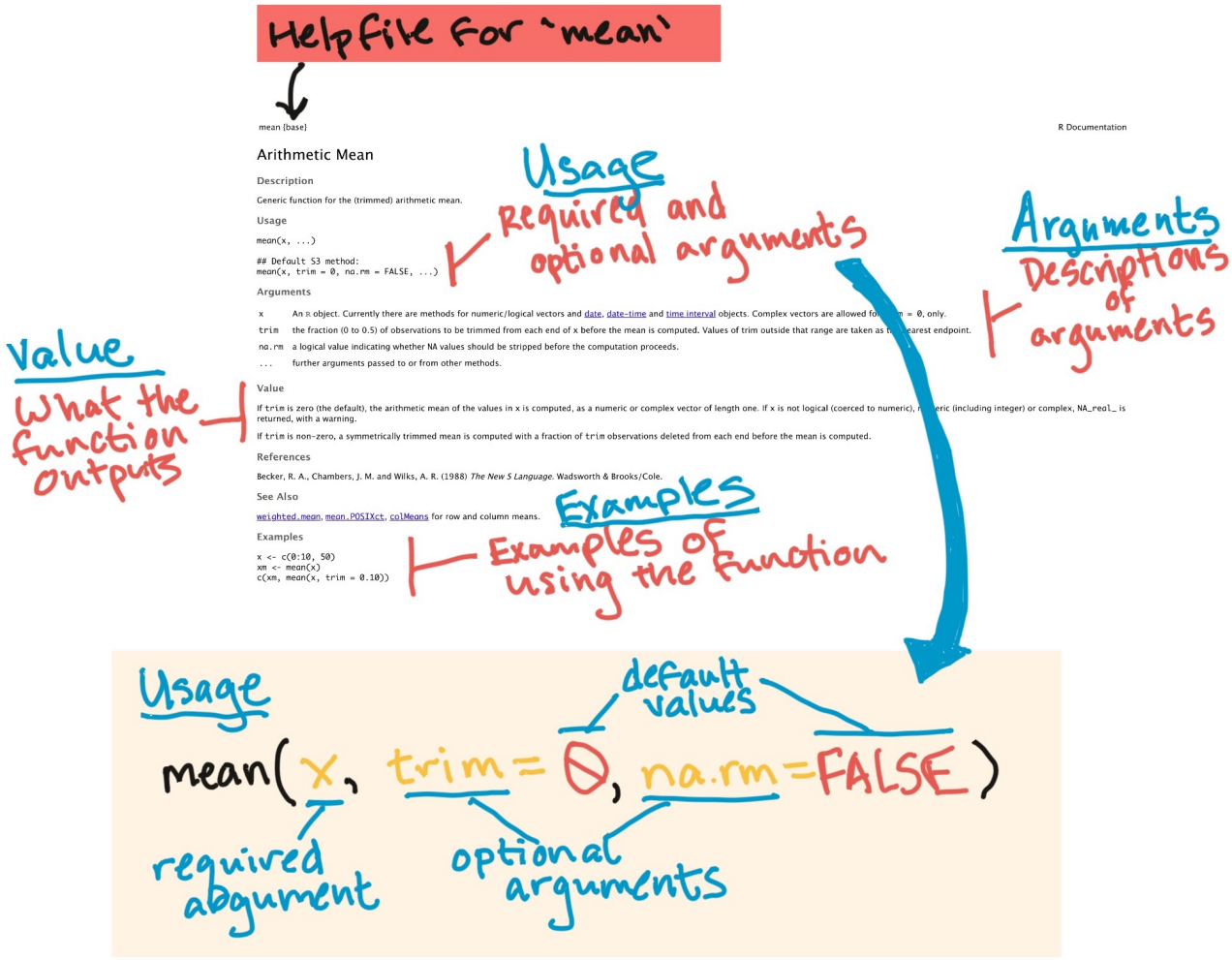


Figure 1.5: Navigating a helpfile. This example shows some key parts of the helpfile for the 'mean' function.

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

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

Function calls will produce an object. If you just call a function, as we've been doing, then R will respond by printing out that object. However, we'll often want to use that object some more. For example, we might want to use it as an argument later in our "conversation" with R, when we call another function later. If you want to re-use the results of a function call later, you can **assign** that **object** to an **object name**. This kind of expression is called an **assignment expression**.

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

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

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

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

```
"Hello world"
```

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

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

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

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

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

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

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

Here's are examples of these strategies:

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

```
message
```

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

2. Call the `ls` function:

```
ls()
```

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

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

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

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

```
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
```

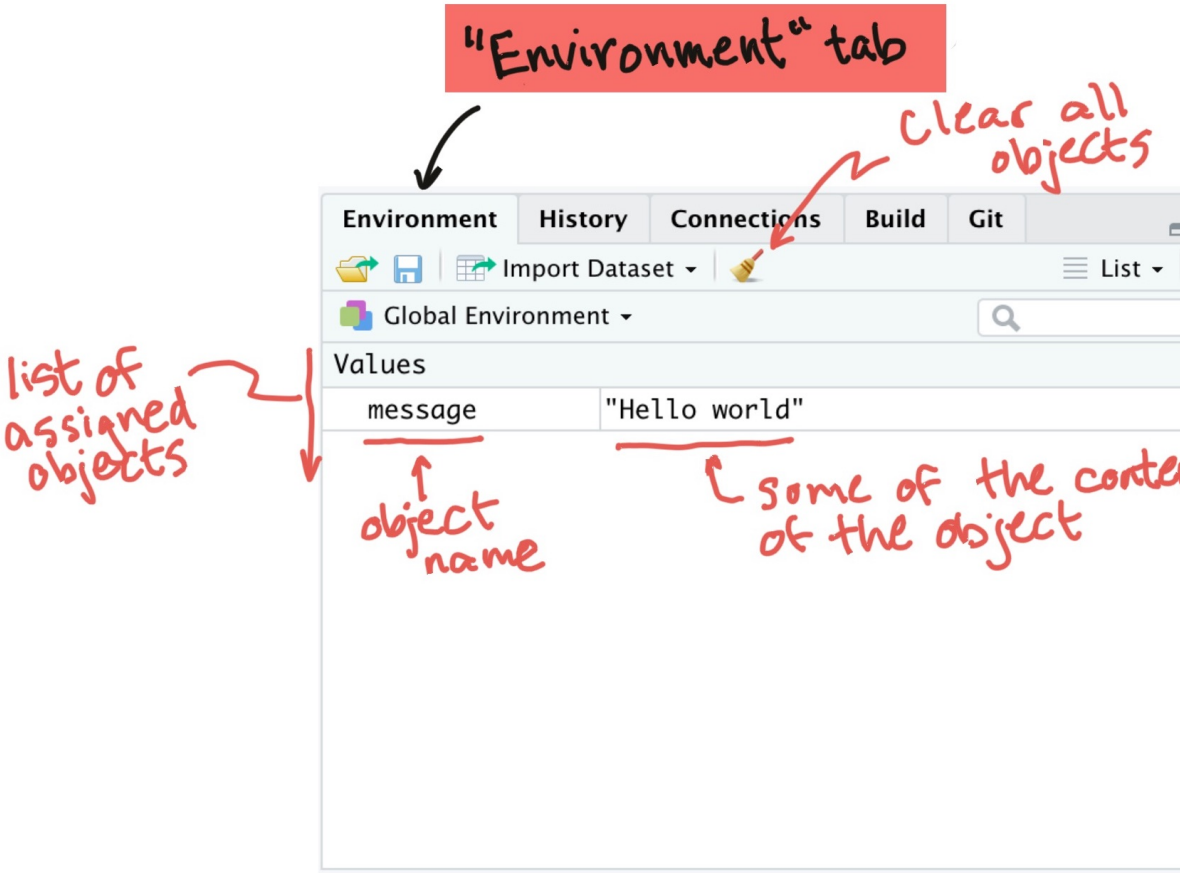


Figure 1.6: 'Environment' pane in RStudio. This shows the names and first few values of all objects that have been assigned to object names in the global environment.

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

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

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

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

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

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

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

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

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

Sometimes, you'll want to create an object that you won't want to keep for very long. For example, you might want to create a small object to test some code, but you plan to not need the object again once you've done that. You may want to come up with some short, generic object names that you use for these kinds of objects, so that you'll know that you can delete them without problems when you want to clean up your R session.

There are all kinds of traditions for these placeholder variable names in computer science. `foo` and `bar` are two popular choices, as are, evidently, `xyzzy`, `spam`, `ham`, and `norf`. There are different placeholder names in different languages: for

example, `toto`, `truc`, and `azerty` (French); and `pippo`, `pluto`, `paperino` (Disney character names; Italian). See the Wikipedia page on metasyntactic variables to find out more.

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

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

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

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

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

a
```

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

```
b
```

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

```
a <- b
a
```

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

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

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

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

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

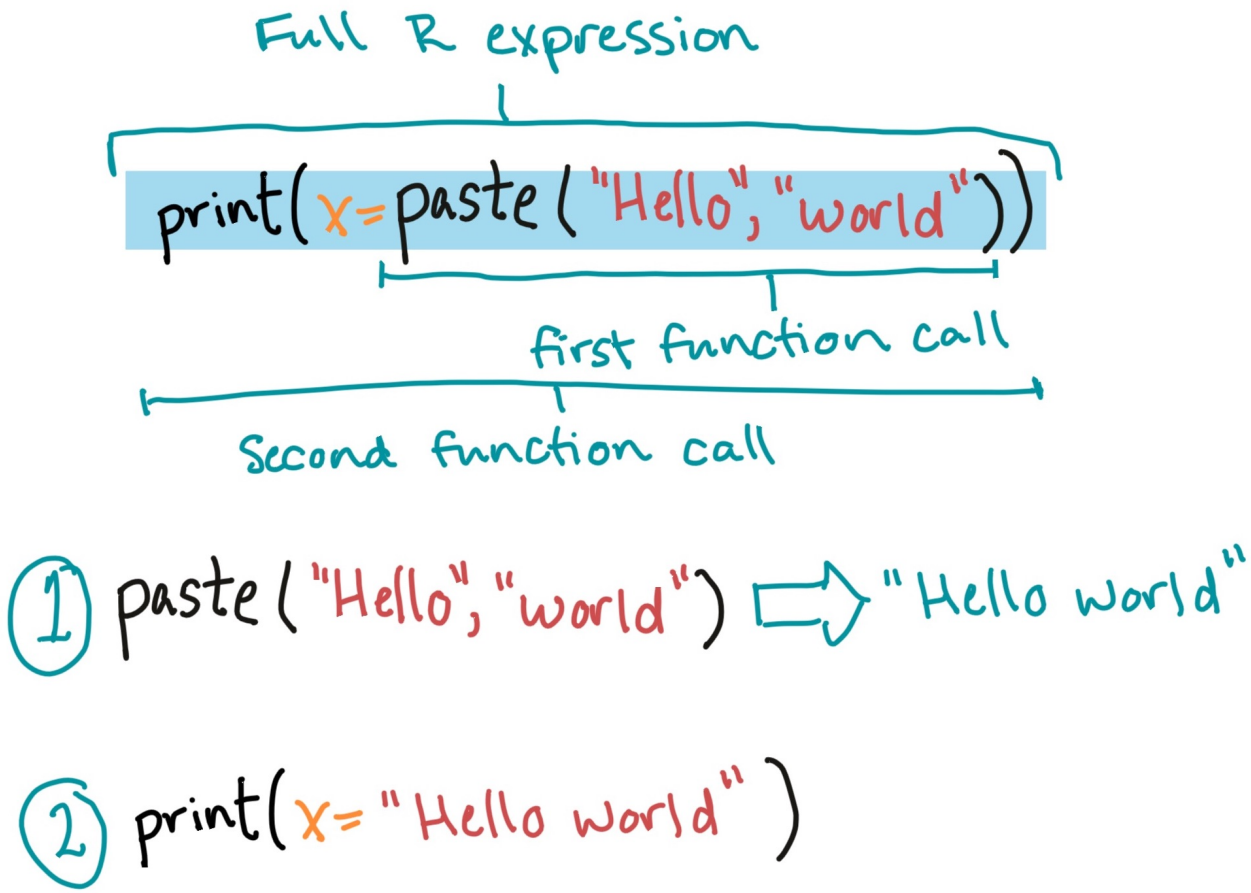


Figure 1.7: 'Environment' pane in RStudio. This shows the names and first few values of all objects that have been assigned to object names in the global environment.

1.4 R scripts

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

An R script is a plain text file where you can save a series of R commands. You can save the script and open it up later to see (or re-do) what you did earlier, just like you could with something like a Word document when you're writing a paper. To open a new R script in RStudio, go to the menu bar and select "File" -> "New File" -> "R Script". Alternatively, you can use the keyboard shortcut Command-Shift-N. Figure 1.8 gives an example of an R script file opened in RStudio and points out some interesting elements.

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

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

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

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

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

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

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

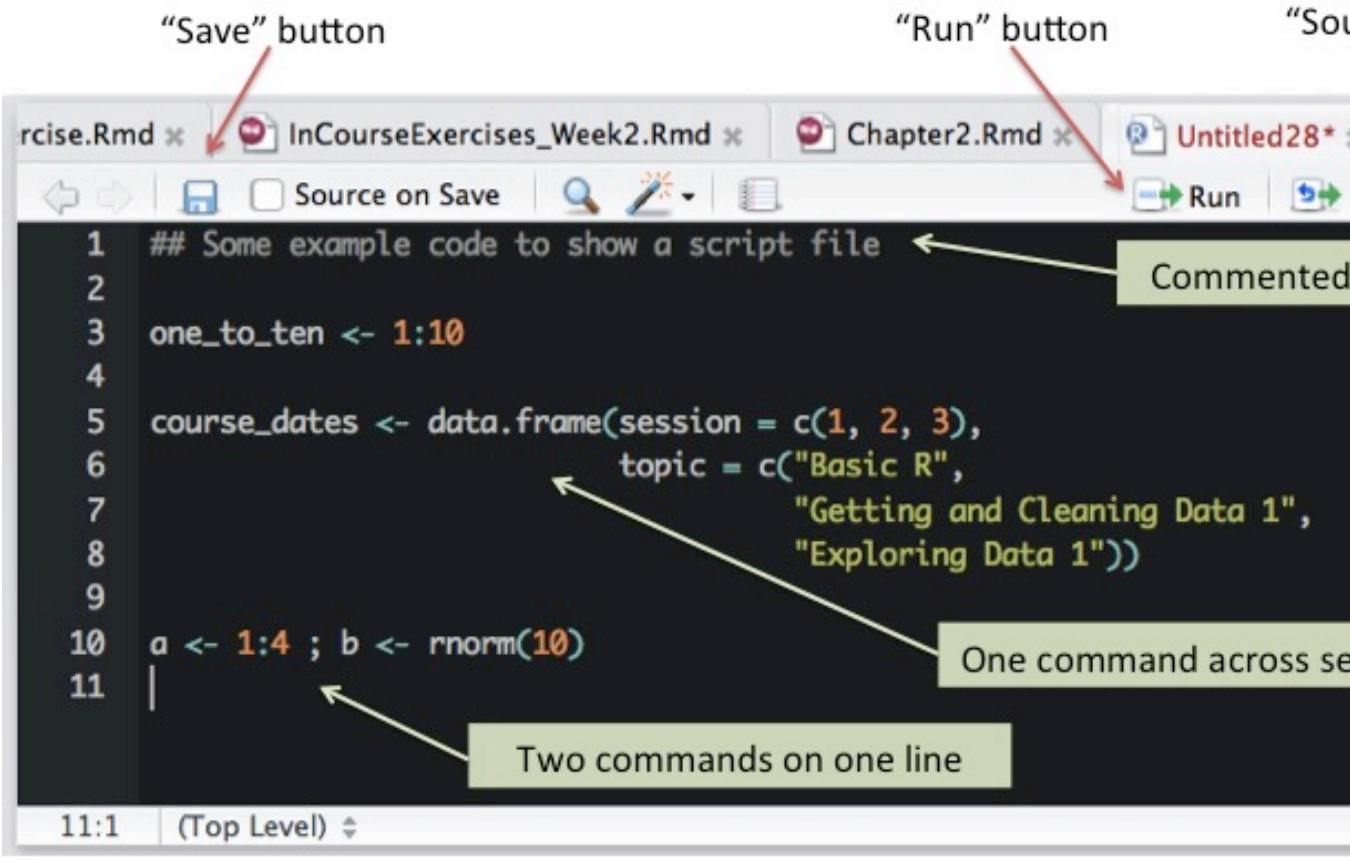


Figure 1.8: Example of an R script in RStudio.

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.

1.4.1 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

1.5 The "package" system

1.5.1 R packages

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

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

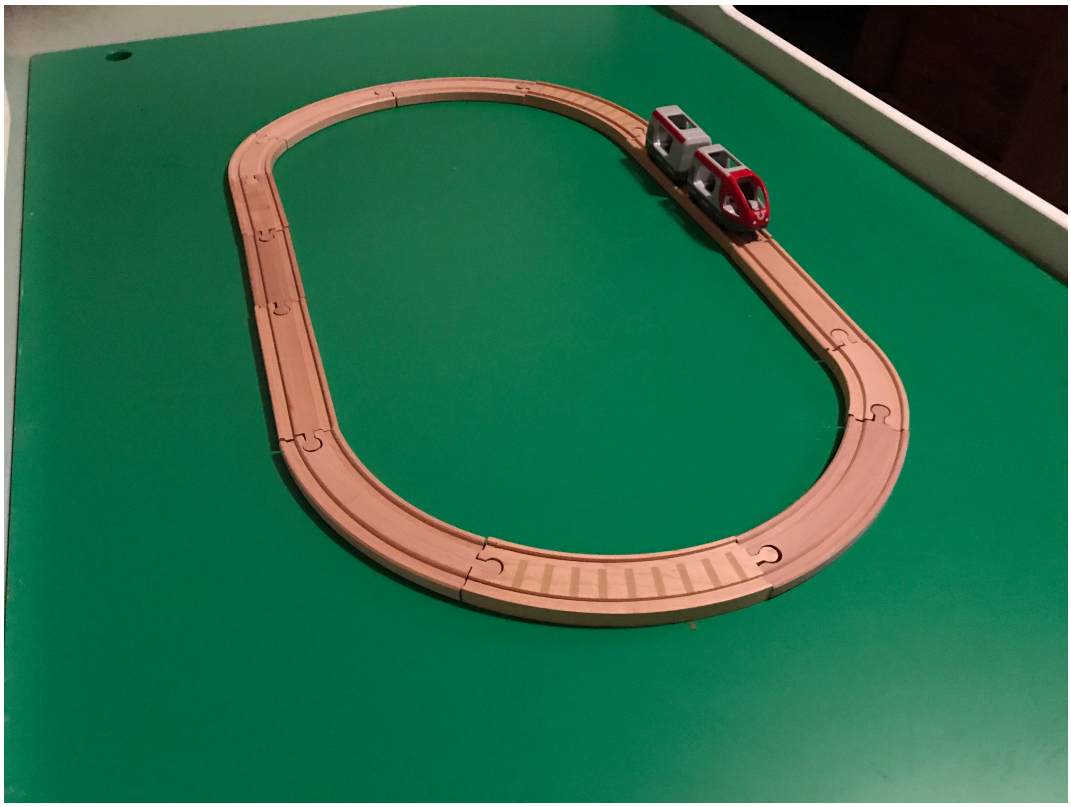


Figure 1.9: The toy version of base R.

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

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

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

You can get these "add-on" packages in a number of ways. The main source for installing packages for R remains the Comprehensive R Archive Network, or CRAN. However, GitHub is growing in popularity, especially for packages that are still in development. You can also create and share packages among your collaborators or co-workers, without ever posting them publicly. In the "Advanced" section of this course, you will learn some about writing your own R package.

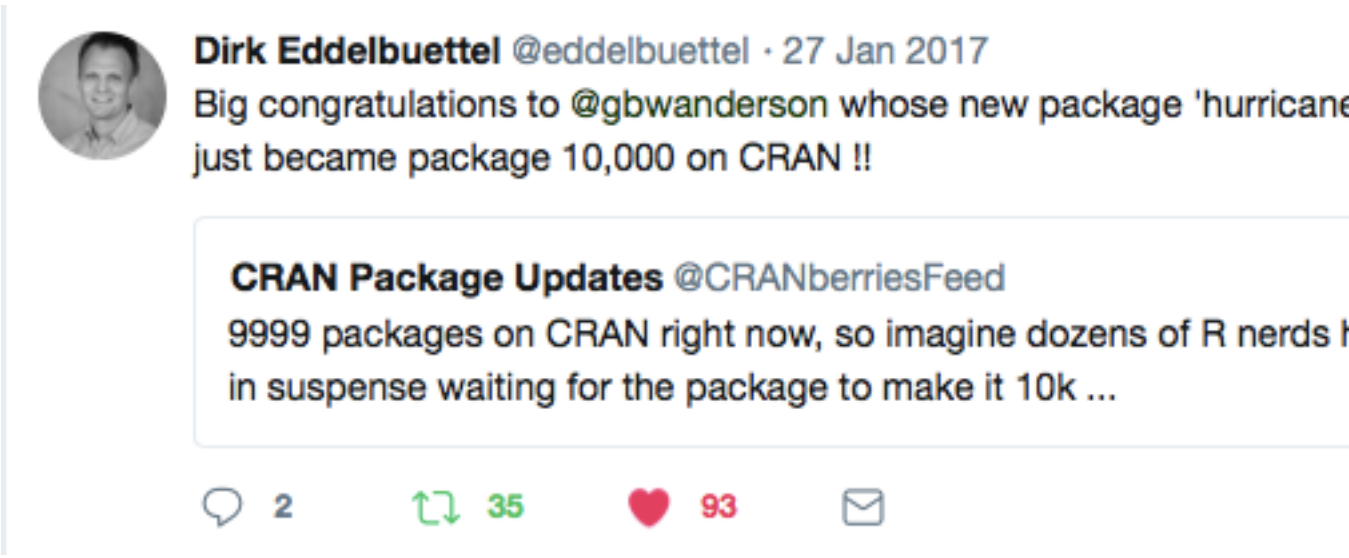


Figure 1.11: Celebrating CRAN's 10,000th package.

1.5.2 Installing from CRAN

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

The `phonenumber` 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("phonenumber")
```

This downloads the package from CRAN and saves it in a special location on your computer where R can load it when you're ready to use it. Once you've installed a package to your computer this way, you don't need to re-run this `install.packages` for the package ever again (unless the package maintainer posts an updated version).

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

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

1.5.3 Loading an installed package

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

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

```
library("phonenumber")
```

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

Once a package is loaded, you can use all its exported (i.e., public) functions by calling them directly. For example, the `phonenumber` has a function called `letterToNumber` that converts a character string to a number. If you have not loaded the `phonenumber` package in your current R session and try to use this function, you will get an error. However, once you've loaded `phonenumber` using the `library` function, you can use this function in your R session:

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

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

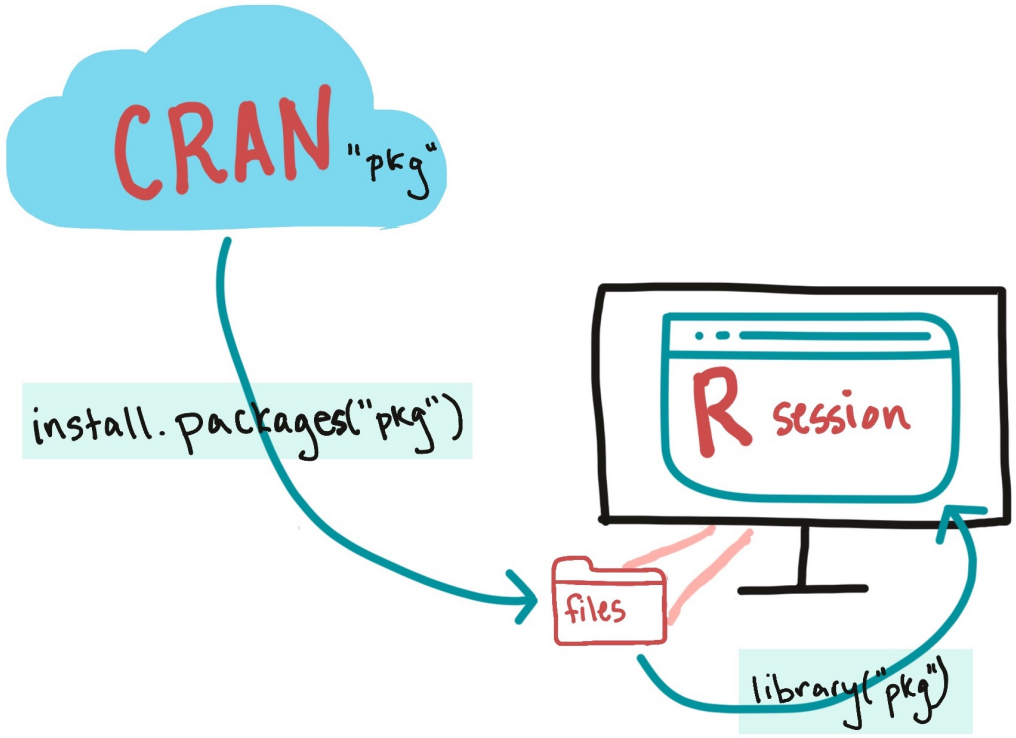


Figure 1.13: Install a package (with 'install.packages') to get it onto your computer. Load it (with 'library') to get it into your R session.

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

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

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

```
citation("phonenumber")
```

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

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., `phonenumber::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 occassionally see it used to distinguish between two functions from different packages that have the same name, as this format makes the desired function

unambiguous. One example where this syntax often is needed is when both `plyr` and `dplyr` packages are loaded in an R session, since these share functions with the same name.

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

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

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

```
vignette(package = "phonenumber")
```

From the output of this, you can call any of the package's vignettes directly. For example, the previous call tells you that this package only has one vignette, and that vignette has the same name as the package ("phonenumber"). Once you know the name of the vignette you would like to open, you can also use `vignette` to open it:

```
vignette("phonenumber")
```

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

```
?letterToNumber
```

1.6 R's most basic object types

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

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

There are a variety of different object types in R, shaped to fit different types of objects ranging from the simple to complex. In this section, we'll start by describing two object

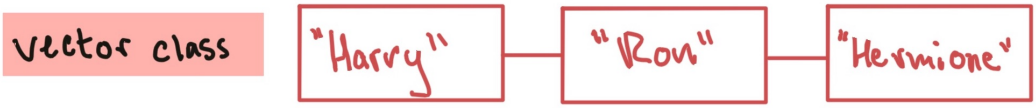


Figure 1.14: An example of the structure of an R object with the vector class. This object class contains data as a string of values, all with the same data type.

types that you will use most often in basic data analysis, **vectors** (1-dimensional objects) and **dataframes** (2-dimensional objects).

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

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

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

1.6.1 Vectors

To get an initial grasp of the *vector* object type in R, think of it as a 1-dimensional object, or a string of values. Figure 1.14 provides an example of the structure for a very simple vector, one that holds the names of the three main characters in the *Harry Potter* book series.

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

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

```
c("Harry", "Ron", "Hermione")
```

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

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

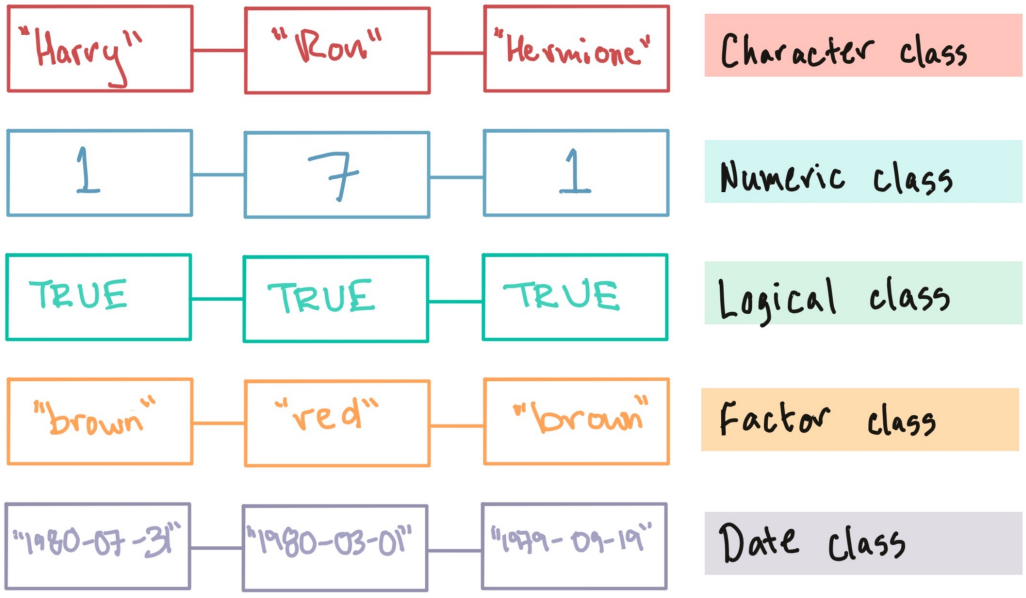


Figure 1.15: Examples of vectors of different classes. All the values in a vector must be of the same type (e.g., all numbers, all characters). There are different classes of vectors depending on the type of data they store.

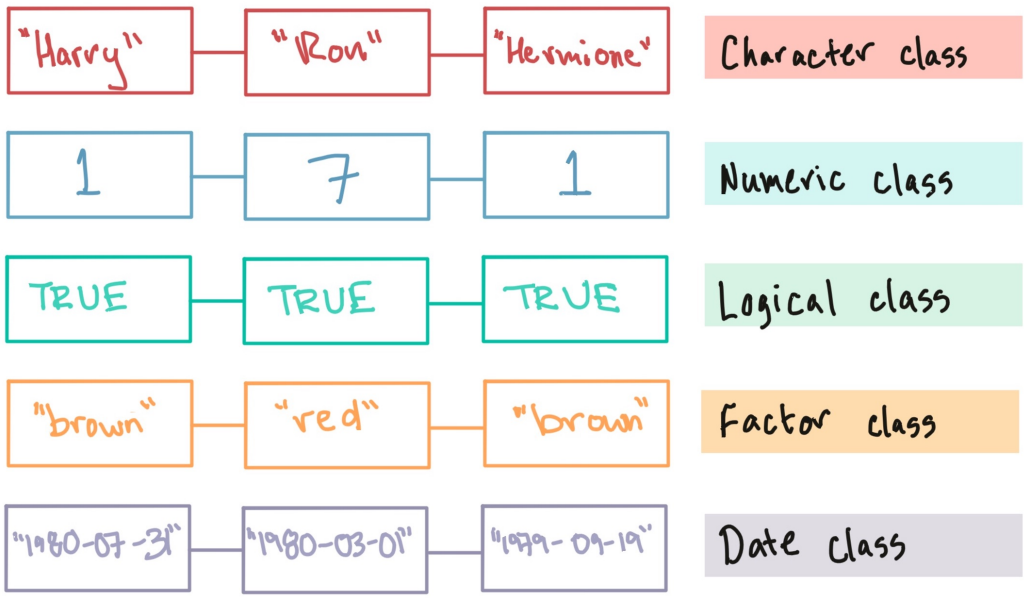


Figure 1.16: Elements of the assignment expression for creating a vector and assigning it an object name.

```
main_characters <- c("Harry", "Ron", "Hermione")
print(x = main_characters)
```

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

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

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

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

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

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

```
## [1] "1"    "3"    "five"
```

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

```
class(mixed_classes)
```

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

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

```
length(mixed_classes)
```

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

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

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

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

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

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

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

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

```
1:10
```

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

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

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

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

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

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

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

1.6.2 Dataframes

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

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

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

This dataframe is arranged in rows and columns, with column names for each column (Figure 1.18). Note that each row of this dataframe gives a different observation (in this

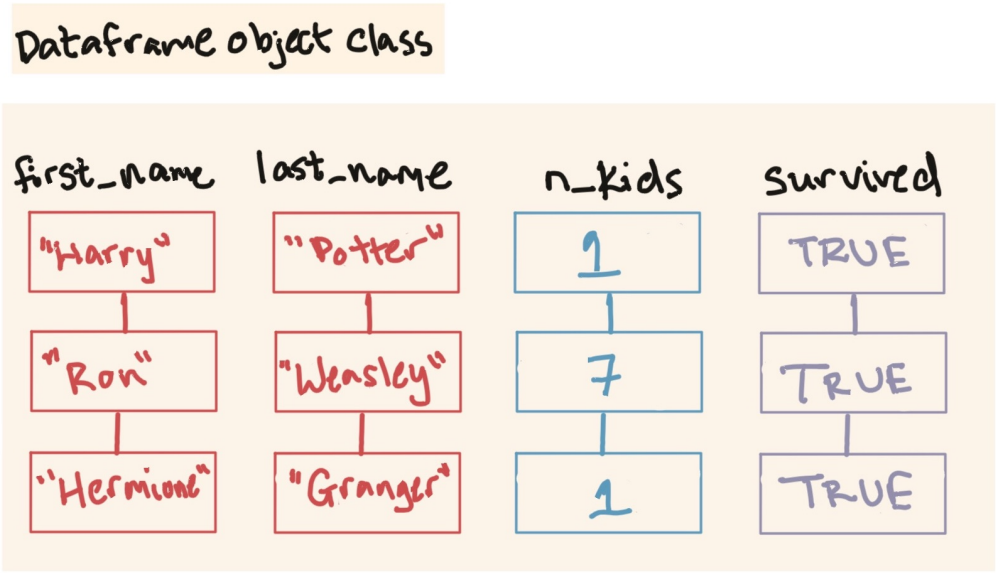


Figure 1.17: An example dataframe, created from several vectors of the same length and with observations aligned across vector positions (for example, the first value in each vector provides a value for Harry, the second for Ron).

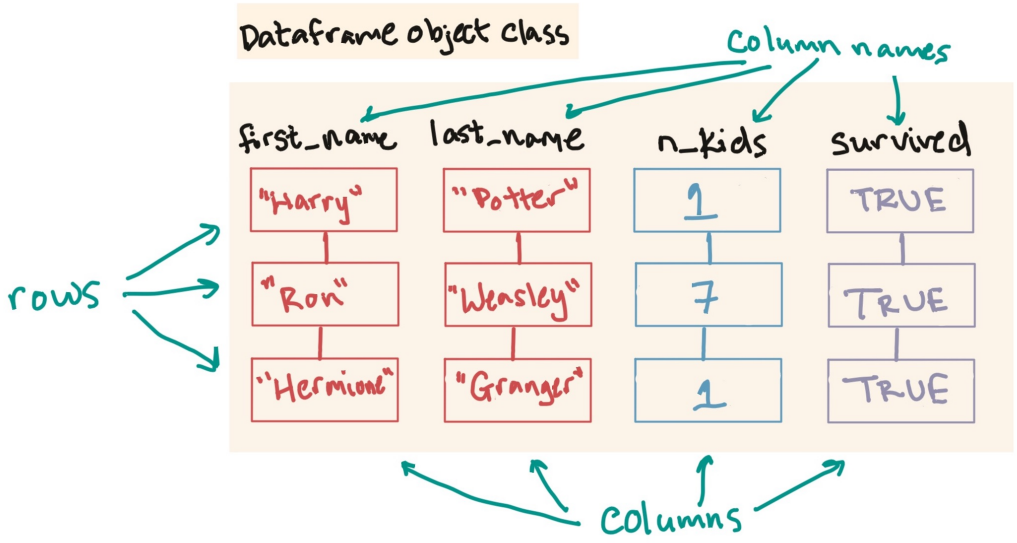


Figure 1.18: The elements of a dataframe: columns, rows, and column names.

case, our unit of observation is a Harry Potter character). Each column gives a different type of information (first name, last name, birth year, and whether they're still alive) for each of the observations (Beatles). Notice that the number of elements in each of the columns must be the same in this dataframe, but that the different columns can have different classes of data (e.g., character vectors for `first_name` and `last_name`, logical value of TRUE or FALSE for `alive`).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
dim(daily_show)
```

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

```
nrow(daily_show)
```

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

```
ncol(daily_show)
```

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

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

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

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

```
str(daily_show)
```

```
## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 2693 obs. of  5 variables:
##  $ YEAR                    : num  1999 1999 1999 1999 1999 ...
##  $ GoogleKnowlege_Occupation: chr  "actor" "Comedian" "television actress" "film actress" ...
##  $ Show                    : chr  "1/11/99" "1/12/99" "1/13/99" "1/14/99" ...
##  $ Group                   : chr  "Acting" "Comedy" "Acting" "Acting" ...
##  $ Raw_Guest_List          : chr  "Michael J. Fox" "Sandra Bernhard" "Tracey Ullman" "Gillian
##  - attr(*, "spec")=
##   .. cols(
##   ..   YEAR = col_double(),
##   ..   GoogleKnowlege_Occupation = col_character(),
##   ..   Show = col_character(),
##   ..   Group = col_character(),
##   ..   Raw_Guest_List = col_character()
##   .. )
```

```
summary(daily_show)
```

```
##       YEAR      GoogleKnowlege_Occupation     Show
##  Min.   :1999   Length:2693               Length:2693
##  1st Qu.:2003   Class :character          Class :character
##  Median :2007   Mode  :character          Mode  :character
##  Mean   :2007
##  3rd Qu.:2011
##  Max.   :2015
##     Group           Raw_Guest_List
##  Length:2693        Length:2693
##  Class :character   Class :character
##  Mode  :character   Mode  :character
##
##
##
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.7 In-course Exercise

1.7.1 Trying out the code in slides so far

To start, you'll try running some simple code in R, using examples from the course slides up to this point. Take the following steps:

1. Open an R session and find the "Console" pane.
2. Go through the slides we've covered so far. Find any examples of R expressions and try them out at the prompt in the console.
3. Once you've run an assignment expression, find the "Environment" pane. Check that the object name that you assigned now appears there.

1.7.2 Writing your code as an R script

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

- Open a new R script and save it to your current working directory (i.e., wherever you saved the data you downloaded for this exercise).
- Take some of the code that you wrote for this exercise. Put it in the R script. Do not put more than one function call per line (but it's fine to have longer function calls span a few lines).
- Use the "Run" button to run a single line of this code. Check the console to see what happens when you do.
- Highlight a few lines of the code and use "Run" to run them.
- Try using the keyboard shortcut (Command-Return) to run the line of code your cursor is currently on. Try doing this with a function call that runs across several lines of the R script file– what do you see at the console?
- Try running the whole script using "Source". Again, look at the console after you "source" the script.
- Close your R session (and save any changes to your R script). Do **not** save your R session history. Re-open R and see if you can re-open your R script and re-run

it. Try using `ls()` to list the objects in your R session before and after you re-run your script. Does anything about the result surprise you?

1.7.3 About the dataset

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

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

- 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?

1.7.4 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 functions from the `dplyr` package to print out the following subsets of this dataframe (you'll have one R call per subset): (1) The appearance year of the first guest; (2) Names of the third through fifth guests; (3) Names of all guests; (4) Both names and appearance years of the first and third guests.
- The `str` function can be used to figure out the structure of a dataframe. Run this command on the `guest_list` dataframe you created. What information does this give you? Use the helpfile for `str` to help you figure this out (which you can access by running `?str`). Do you see anything that surprises you?
- Use the `ls` function to list all the objects you currently have defined in your R session. Compare this list to the "Environment" pane in RStudio.

Example R code:

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

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

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

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

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

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

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

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

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

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

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

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

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

```
class(appearance_year)
```

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

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

```
## # A tibble: 5 x 2
##   Raw_Guest_List   YEAR
##   <chr>           <dbl>
## 1 Miss Piggy       1999
## 2 Stanley Tucci    2000
## 3 Kermit the Frog  2001
## 4 Hank Azaria      2001
## 5 Al Gore          2002
```

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

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

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

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

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

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

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

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

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

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    5 obs. of  2 variables:
##  $ Raw_Guest_List: chr  "Miss Piggy" "Stanley Tucci" "Kermit the Frog" "Hank Azaria" ...
##  $ YEAR          : num  1999 2000 2001 2001 2002
```

1.7.5 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 stings. 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 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)
```

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

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

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

1.7.6 Getting the data onto your computer

Next, we will work with the whole dataset. Download the data from GitHub onto your computer. In class, we created an R Project for you to use for this class. Put the *Daily*

Show data in that directory.

Take the following steps to get the data onto your computer

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

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

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

1.7.7 Getting the data into R

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

Read the data into your R session

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

Example R code:

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

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

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

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

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

If you have extra time:

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

Example R code:

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

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

```
# Check the dataframe
daily_show_first10
```

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

1.7.8 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 (that it includes all the guests who came on The Daily Show with Jon Stewart), do you think it has about the right number of rows?

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

Example R code:

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

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

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

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

```
head(daily_show)
```

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

```
tail(daily_show)
```

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

```
## 1  2015 actor                    7/28/15 Acting Tom Cruise
## 2  2015 biographer               7/29/15 Media  Doris Kearns Goodwin
## 3  2015 director                 7/30/15 Media  J. J. Abrams
## 4  2015 stand-up comedian        8/3/15  Comedy Amy Schumer
## 5  2015 actor                    8/4/15  Acting Denis Leary
## 6  2015 comedian                 8/5/15  Comedy Louis C.K.
```

If you have extra time:

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

Example R code:

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

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

1.7.9 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 (`Raw_Guest_List == "Nate Silver"`). Save this as an object named `nate_silver`.
- Print out the full `nate_silver` dataframe by typing `nate_silver`. (You could just use this to answer both questions, but still try the next steps. They would be important with a bigger dataset.)

- To count the number of times Nate Silver was a guest, you'll need to count the number of rows in the new dataset. You can either use the `dim` function or the `nrow` function to do this. What additional information does the `dim` function give you?
- To get the dates when Nate Silver was a guest, you can print out just the `Show` column of the dataframe. There are a few ways you can do this using the `select` function from the `dplyr` package.

Example R code:

```
# 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
```

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

```
dim(nate_silver)
```

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

```
nrow(nate_silver)
```

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

```
select(.data = nate_silver, 3)
```

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

If you have extra time:

- Was Nate Silver the only statistician to be a guest on the show?

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

Example R code:

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

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

```
num_visits <- table(daily_show[ , 2])
head(num_visits) # Note: This is a vector rather than a dataframe
```

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

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

```
## [1] "-"                "accountant"       "administrator"
## [4] "advocate"         "aei president"    "afghan politician"
```

```
subset(daily_show, GoogleKnowlege_Occupation == "chess player")
```

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

```
subset(daily_show, GoogleKnowlege_Occupation == "mathematician")
```

```
## # A tibble: 1 x 5
```

```
##    YEAR GoogleKnowlege_Occupation Show    Group    Raw_Guest_List
##   <dbl> <chr>                     <chr>   <chr>    <chr>
## 1  2005 mathematician             9/14/05 Academic Dr. William A. Dembski
```

```
subset(daily_show, GoogleKnowlege_Occupation == "orca trainer")
```

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

```
subset(daily_show, GoogleKnowlege_Occupation == "Puzzle Creator")
```

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

```
subset(daily_show, GoogleKnowlege_Occupation == "Scholar")
```

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

Part II

Part II: Basics

Chapter 2

Entering and cleaning data #1

Download a pdf of the lecture slides covering this topic.

2.1 Objectives

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

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

2.2 Overview

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

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

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

2.3 Reading data into R

The following video covers the slides in this section on "Reading data into R":

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

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

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

2.3.1 Reading local flat files

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

1. Fixed width files; and
2. Delimited files:

- ".csv": Comma-separated values
- ".tab", ".tsv": Tab-separated values
- Other possible delimiters: colon, semicolon, pipe ("|")

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

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

```
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"
```

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

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

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

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

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

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

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

```
library(readr)

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

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

```
## Parsed with column specification:
## cols(
##   Date = col_character(),
##   Day = col_double(),
##   Cases_Guinea = col_double(),
##   Cases_Liberia = col_double(),
##   Cases_SierraLeone = col_double(),
##   Cases_Nigeria = col_double(),
##   Cases_Senegal = col_double(),
##   Cases_UnitedStates = col_double(),
##   Cases_Spain = col_double(),
##   Cases_Mali = col_double(),
##   Deaths_Guinea = col_double(),
##   Deaths_Liberia = col_double(),
##   Deaths_SierraLeone = col_double(),
##   Deaths_Nigeria = col_double(),
##   Deaths_Senegal = col_double(),
##   Deaths_UnitedStates = col_double(),
##   Deaths_Spain = col_double(),
##   Deaths_Mali = col_double()
## )
```

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

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

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

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

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

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

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

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

- `read.csv`

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

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

2.3.2 Reading in other file types

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

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

File type	Function	Package
Excel	'read_excel'	'readxl'
SAS	'read_sas'	'haven'
SPSS	'read_spss'	'haven'
Stata	'read_stata'	'haven'

2.4 Directories and pathnames

The following video covers this section on "Directories and pathnames":

2.4.1 Directory structure

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

However, you'll often want to read in files that are located somewhere else on your computer, or even files that are saved on another computer (for example, data files that are posted online). Doing this is very similar to reading in a file that is in your current

working directory; the only difference is that you need to give R some directions so it can find the file.

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

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

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

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

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

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

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

You can notice a few interesting things from Figure 2.1. First, you might notice the structure includes a few of the directories that you use a lot on your own computer, like `Desktop`, `Documents`, and `Downloads`. Next, the directory at the very top is the computer's root directory, `/`. For a PC, the root directory might something like `C:`; for Unix and Macs, it's usually `/`. Finally, if you look closely, you'll notice that it's possible to have different files in different locations of the directory structure with the same file name. For

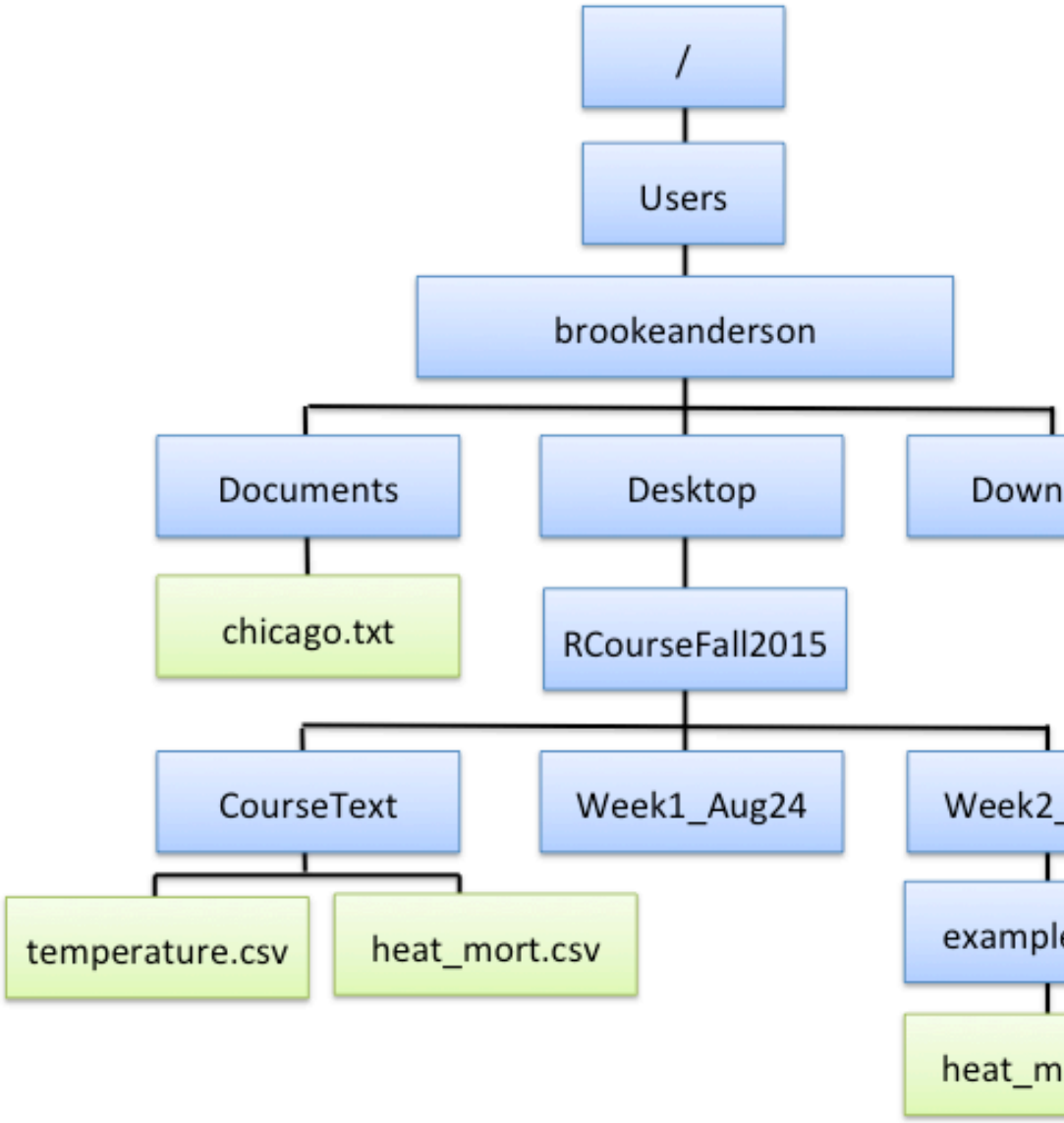


Figure 2.1: An example of file directory structure.

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

2.4.2 Working directory

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

```
getwd()
```

```
## [1] "/Users/georgianaanderson/Documents/my_books/RProgrammingForResearch"
```

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

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

2.4.3 File and directory pathnames

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

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

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

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

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

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

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

```
heat_mort <- read_csv("/Users/brookeanderson/Desktop/RCourseFall2015/CourseText/heat_m
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relative pathnames "break" as soon as you tried them from a different working directory—this fact might make it seem like you would never want to use relative pathnames, and would always want to use absolute ones instead, even if they're longer. If that were the only consideration (length of the pathname), then perhaps that would be true. However, as you do more and more in R, there will likely be many occasions when you want to use relative pathnames instead. They are particularly useful if you ever want to share a whole directory, with all subdirectories, with a collaborator. In that case, if you've used relative pathnames, all the code should work fine for the person you share with, even though

they're running it on their own computer. Conversely, if you'd used absolute pathnames, none of them would work on another computer, because the "top" of the directory structure (i.e., for me, `/Users/brookeanderson/Desktop`) will almost definitely be different for your collaborator's computer than it is for yours.

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

2.4.4 Diversion: `paste`

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

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

```
my_file <- "/Users/brookeanderson/Desktop/RCourseFall2015/CourseText/heat_mort.csv"
heat_mort <- read_csv(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 stings.) You separate all the different things you want to paste together using with 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.

2.4.5 Reading online flat files

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

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

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

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

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

2.5 Data cleaning

The following video covers the section on "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	'dplyr' function
Renaming columns	'rename'
Filtering to certain rows	'filter'
Selecting certain columns	'select'
Adding or changing columns	'mutate'

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

2.5.1 Renaming columns

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

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

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

```
colnames(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
##   <dbl> <chr>              <chr>   <chr>    <chr>
## 1  1999 actor              1/11/99 Acting   Michael J. Fox
## 2  1999 Comedian           1/12/99 Comedy   Sandra Bernhard
## 3  1999 television actress 1/13/99 Acting   Tracey Ullman
```

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

2.5.2 Selecting columns

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>
##  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
```

2.5.3 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 `str_to_lower` function from the `stringr` package within a `mutate` function:

```
library(stringr)
mutate(daily_show, job = str_to_lower(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
```

2.5.4 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.

2.6 Dates and filtering

The following video covers the lecture material on working with dates in R and on filtering a dataframe to a subset of rows using logical operators:

2.6.1 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
`character`	"Chemistry", "Physics", "Mathematics"
`numeric`	10, 20, 30, 40
`factor`	Male [underlying number: 1], Female [2]
`Date`	"2010-01-01" [underlying number: 14,610]

Class	Example
`logical`	TRUE, FALSE

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

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

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

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

To convert a vector to the `Date` class, if you'd like to only use base R, you can use the `as.Date` function. I'll walk through how to use `as.Date`, since it's often used in older R code. However, I recommend in your own code that you instead use the `lubridate` package, which I'll talk about later in this section.

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 actor              1999-01-11 Acting   Michael J. Fox
## 2 Comedian           1999-01-12 Comedy   Sandra Bernhard
## 3 television actress 1999-01-13 Acting   Tracey Ullman
```

```
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)
`%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 package 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 = GoogleKnowlege_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 Tue      Catherine Weitz
## 3 2004-06-15 Tue      Hassan Ibrahim
## 4 2005-09-06 Tue      Dr. Marc Siegel
## 5 2006-02-13 Mon      Astronaut Mike Mullane
```

R functions tend to use the timezone of **YOUR** computer's operating system by default, or UTC, or GMT. You need to be careful when working with dates and times to either specify the time zone or convince yourself the default behavior works for your application.

2.6.2 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>          <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
## 4 doctor         2005-09-06 Science  Dr. Marc Siegel
## 5 astronaut      2006-02-13 Science  Astronaut Mike Mullane
## 6 Astrophysicist 2007-01-30 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
`==`	equals	`category == "Acting"`
`!=`	does not equal	`category != "Comedy`
`%in%`	is in	`category %in% c("Academic", "Science")`
`is.na()`	is missing	`is.na(job)`
`!is.na()`	is not missing	`!is.na(job)`
`&`	and	`year == 2015 & category == "Academic"`
`|`	or	`year == 2015 | category == "Academic"`

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.

2.7 Piping

The following video covers the lecture material on 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.

2.8 In-course Exercise

2.8.1 Downloading and checking out the example data

Download the whole directory for this week from Github (https://github.com/geanders/week_2_data). To do that, go the the GitHub page with data for this week's exercise and, in the top right, 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 Project 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). All the files will be in a subdirectory—move them into the main Project directory (don't do this in R, just use whatever technique you usually use on your computer to move files between directories).

- Look through the structure of the "data" directory. What files are in the directory? Which files are **flat files**? Which are **delimited** (one category of flat files), and what are their delimiters?
- 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").

2.8.2 Reading in different types of files

Now you'll try reading in data from a variety of types of file formats.

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? Since it's in a subdirectory, you'll need to tell R how to get to it from the project directory, using something called a **relative pathname** (we'll talk about this a lot more in the next section of the lecture). Read this file into R as an object named `ca_measles`, using the relative pathname ("measles_data/02-09-2015.txt") in place of the file name in the `read_tsv` function call. Use the `col_names` option to name the columns "city" and "count". What would the default column names be if you didn't use this option (try this out by running `read_csv` without the `col_names` option)?
- Read in the Excel file "icd-10.xls" and assign it to the object name `idc10`. Use the `readxl` package to do that (examples are at the bottom of the linked page).
- Read in the SAS file `icu.sas7bdat`. To do this, use the `haven` package. Read the file into the R object `icu`.

Example R code:

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

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

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

```
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("measles_data/02-09-2015.txt",
                       col_names = c("city", "count"))
```

```
## Parsed with column specification:
## cols(
##   city = col_character(),
##   count = col_double()
## )
```

```
head(ca_measles)
```

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

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

```
# Use the `read_excel` function to read in the file.
icd10 <- read_excel("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 01, biovar cholerae
## 5 A00.1   Cholera due to Vibrio cholerae 01, biovar eltor
## 6 A00.9   Cholera, unspecified
```

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

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

2.8.3 Directory structure

Once you have the data, I'd like you to try using `getwd()` to figure out your current working directory and `list.files()` to figure out which files you have in the directories near that current working directory.

Start by creating a new subdirectory called "data" in your R Project directory for this class (if you don't already have that subdirectory). Move the data you downloaded in the start of this In-Course Exercise into that "data" subdirectory. (For this, use whatever tools you would normally use on your computer to move files from one directory to another—you don't have to do this part in R.) Keep the "measles" data in its own subdirectory (so, the "data" subdirectory of your project will have its own "measles" subdirectory, which will have those files).

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

```
getwd()
```

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

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

```
list.files()
```

Try the following tasks:

- Read in the ebola data in `country_timeseries.csv` from your current working directory. This will require you to use a relative filename to direct R to how to find that file in the "data" subdirectory of your current working directory.
- Assign the data you read in to an R object named `ebola`. How many rows and columns does it have? What are the names of the columns?
- While staying in the same working directory, used `list.files()` to print the names of the available files in the "data" subdirectory. How about in the "R" subdirectory (if you have one)?
- Try to list the files in your "data" subdirectory using:
  - A relative pathname
  - An absolute pathname
- Now use a relative pathname along with `list.files()` to list all the files in the "measles_data" subdirectory.
- Then try to read in the Ebola data using the appropriate `readr` function and a relative 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?

Read in the ebola data in `country_timeseries.csv` from your current working directory. This will require you to use a relative filename to direct R to how to find that file in the "data" subdirectory of your current working directory.

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

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

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

While staying in the same working directory, used `list.files()` to print the names of the available files in the "data" subdirectory. How about in the "R" subdirectory (if you have one)?

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

Try to list the files in your "data" subdirectory using: + A relative pathname + An absolute pathname

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

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

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

Then try to read in the Ebola data using the appropriate `readr` function and a relative pathname

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

If you have extra time:

- Find out some more about this Ebola dataset by checking out Caitlin Rivers' Ebola data GitHub repository. Who is Caitlin Rivers? How did she put this dataset together?
- Search for R code related to Ebola research on GitHub. Go to the GitHub home page and use the search bar to search for "ebola". On the results page, scroll down and use the "Language" sidebar on the left to choose repositories with R code. Did you find any interesting projects?
- When you `list.files()` for the "data" subdirectory, almost everything listed has a file extension, like `.csv`, `.xls`, `.sas7bdat`. One thing does not. Which one? Why does this listing not have a file extension?

2.8.4 Cleaning up data #1

Try out the following tasks:

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

```
# Copy this code to an R script and add comments describing what each line is doing
library(haven)
icu <- read_sas("data/icu.sas7bdat")

icu <- select(icu, ID, AGE, GENDER)

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

icu <- mutate(icu,
              gender = factor(gender, levels = c(0, 1), labels = c("Male", "Female")),
              id = as.character(id))
```

```
icu
```

- Following previous parts of the in-course exercise, you have an R object called `ebola` (if you need to, use some code from earlier in this in-course exercise to read in the data and create that object). Create an object called `ebola_liberia` that only has the columns with the date and the number of cases and deaths in Liberia. How many columns does this new dataframe have? How many observations?
- Change the column names to `date`, `cases`, and `deaths`.
- Add a column called `ratio` that has the ratio of deaths to cases for each observation (i.e., death counts divided by case counts).

Example R code:

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

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

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

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

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

```
nrow(ebola_liberia)
```

```
## [1] 122
```

```
## Rename the columns
ebola_liberia <- rename(ebola_liberia,
                        date = Date,
                        cases = Cases_Liberia,
```

```
                        deaths = Deaths_Liberia)
head(ebola_liberia)
```

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

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

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

2.8.5 Cleaning up data #2

- For the `ebola_liberia` dataframe, what class is the `date` column currently in? Convert it to a Date class. What are the starting and ending dates of observations in this dataframe?
- 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, from the observation with the earliest date to the one with the latest date.
- Filter out all rows from the `ebola_liberia` dataframe that are missing death counts for Liberia. How many rows are in the dataframe now?
- Create a new object called `first_five` that has only the five observations with the highest death counts in Liberia. What date in this dataset had the most deaths?

Example R code:

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

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

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

```
## [1] "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 first date to last date
ebola_liberia <- arrange(ebola_liberia, date)
head(ebola_liberia)
```

```
## # A tibble: 6 x 4
##   date       cases deaths ratio
##   <date>     <dbl>  <dbl> <dbl>
## 1 2014-03-22    NA     NA NA
## 2 2014-03-24    NA     NA NA
## 3 2014-03-25    NA     NA NA
## 4 2014-03-26    NA     NA NA
## 5 2014-03-27     8      6  0.75
## 6 2014-03-28     3      3  1
```

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

```
## # A tibble: 6 x 4
##   date       cases deaths ratio
##   <date>     <dbl>  <dbl> <dbl>
## 1 2014-03-27     8      6 0.75
## 2 2014-03-28     3      3 1
## 3 2014-03-29     7      2 0.286
```

```
## 4 2014-03-31     8      4 0.5
## 5 2014-04-01     8      5 0.625
## 6 2014-04-04    18      7 0.389
```

```
nrow(ebola_liberia)
```

```
## [1] 81
```

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

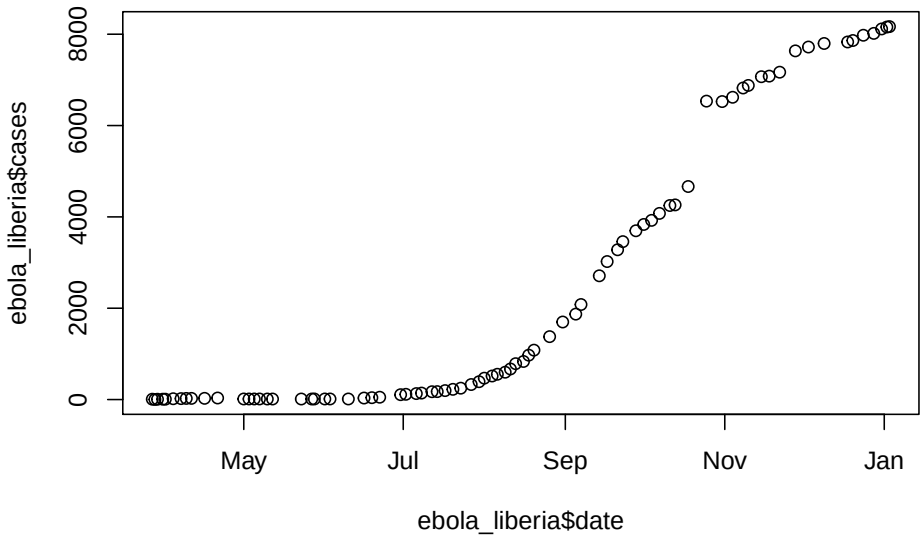
```
## # A tibble: 5 x 4
##   date       cases deaths ratio
##   <date>     <dbl>  <dbl> <dbl>
## 1 2015-01-02  8157   3496 0.429
## 2 2015-01-03  8166   3496 0.428
## 3 2014-12-31  8115   3471 0.428
## 4 2014-12-28  8018   3423 0.427
## 5 2014-12-24  7977   3413 0.428
```

If you have extra time:

- Try using the basic plotting function, `plot()`, to plot the number of cases of Ebola over time in Liberia from this dataframe. 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:

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



2.8.6 Piping

Try the following tasks:

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

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

icu
```

- In previous sections of the in-course exercise, you have created code to read in and clean the Ebola dataset to create `ebola_liberia`. This included the following cleaning steps: (1) selecting certain columns, (2) renaming those columns, (3) adding a `ratio` column, (4) mutating the `date` column to a Date class, (5) re-ordering the observations from earliest to latest date, and (6) removing observations for which the count of deaths in Liberia is missing. Re-write this code to create and clean `ebola_liberia` as "piped" code. Start from reading in the raw data.

Example R code:

```
ebola_liberia <- read_csv("data/country_timeseries.csv") %>%
  select(Date, Cases_Liberia, Deaths_Liberia) %>%
  rename(date = Date,
         cases = Cases_Liberia,
         deaths = Deaths_Liberia) %>%
  mutate(ratio = deaths / cases) %>%
  mutate(date = mdy(date)) %>%
  arrange(date) %>%
  filter(!is.na(cases))

head(ebola_liberia)
```

```
## # A tibble: 6 x 4
##   date       cases deaths ratio
##   <date>     <dbl>  <dbl> <dbl>
## 1 2014-03-27     8      6 0.75
## 2 2014-03-28     3      3 1
## 3 2014-03-29     7      2 0.286
## 4 2014-03-31     8      4 0.5
## 5 2014-04-01     8      5 0.625
## 6 2014-04-04    18      7 0.389
```

Chapter 3

Exploring data #1

Download a pdf of the lecture slides covering this topic.

3.1 Objectives

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

- Be able to load and use datasets from R packages
- Be able to describe and use logical vectors
- Understand how logical vectors check logical statements against other R vector(s) and store TRUE / FALSE values as 0 / 1 at a deeper level
- Be able to use the `dplyr` function `mutate` to create a logical vector as a new column in a dataframe and the `dplyr` function `filter` with that new column to filter a dataframe to a subset of rows
- Be able to use the bang operator (!) to reverse a logical vector
- Know what the "tidyverse" is and name some of its packages
- Be able to use some simple statistical functions (e.g., `min`, `max`, `mean`, `median`, `cor`, `summary`), including how to handle missing values when using these
- Be able to use the `dplyr` function `summarize` to summarize data, with and without grouping using `group_by`, including with special functions `n`, `n_distinct`, `first`, and `last`
- Understand the three basic elements of `ggplot` plots: data, aesthetics, and geoms
- Be able to create a `ggplot` object, set its data using `data = ...` and its aesthetics using `mapping = aes(...)`, and add on layers (including `geoms`) with +
- Be able to create some basic plots (e.g., scatterplots, boxplots, histograms) using `ggplot2` functions
- Understand the difference between setting an aesthetic by mapping it to a column of the dataframe versus setting it to a constant value
- Understand the difference between "statistical" geoms (e.g., histograms, boxplots) and geoms that add one geom element per dataframe observation (row)

3.2 Data from a package

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

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

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

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

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

```
?worldcup
```

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

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

```
data()
```

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

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

```
library(dplyr)
library(faraway)
```

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

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

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

```
##       id    sex   wt    ht age
## 1 120011   Male 12.8  91.2  41
## 2 120012 Female 14.9 103.9  57
## 3 120021 Female  7.7  70.1   8
## 4 120022 Female 12.1  86.4  35
```

3.3 Logical vectors

Last week, you learned a lot about logical statements and how to use them with the `filter` function from the `dplyr` package. You can also use logical vectors, created with these logical statements, for a lot of other things. For example, you can use them directly in the square bracket indexing (`[..., ...]`) to pull out just the rows of a dataframe that meet a certain condition. For using logical statements in either context, it is helpful to understand a bit more about logical vectors.

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

```
length(nepali$sex)
```

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

```
length(nepali$sex == "Male")
```

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

The logical vector (`nepali$sex == "Male"` in this example) will have the value `TRUE` at any position where the original vector (`nepali$sex` in this example) met the logical

condition you tested, and FALSE anywhere else:

```
head(nepali$sex)
```

```
## [1] Male   Female Female Female Male   Male
## Levels: Male Female
```

```
head(nepali$sex == "Male")
```

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

You can "flip" this logical vector (i.e., change every TRUE to FALSE and vice-versa) using the *bang operator*, !:

```
is_male <- nepali$sex == "Male" # Save this logical vector as the object named `is_mal

head(is_male)
```

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

```
head(!is_male)
```

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

The bang operator turns out to be very useful. You will often find cases where it's difficult to write a logical vector to get what you want, but fairly easy to write the inverse (find everything you don't want). One example is filtering down to non-missing values—the is.na function will return TRUE for any value that is NA, so you can use !is.na() to identify any non-missing values.

You can do a few cool things with a logical vector. For example, you can use it inside a filter function to pull out just the rows of a dataframe where is_male is TRUE:

```
nepali %>%
  filter(is_male) %>%
  head()
```

```
##       id  sex   wt   ht age
## 1 120011 Male 12.8 91.2  41
## 2 120023 Male 14.2 99.4  49
## 3 120031 Male 13.9 96.4  46
## 4 120051 Male  8.3 69.5   8
## 5 120053 Male 15.8 96.0  54
## 6 120062 Male 12.1 89.9  57
```

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

```
nepali %>%
  filter(!is_male) %>%
  head()
```

```
##       id    sex   wt    ht age
## 1 120012 Female 14.9 103.9  57
## 2 120021 Female  7.7  70.1   8
## 3 120022 Female 12.1  86.4  35
## 4 120052 Female 11.8  83.6  32
## 5 120061 Female  8.7  78.5  26
## 6 120082 Female 11.2  79.8  36
```

You can also use `sum()` and `table()` with a logical vector to find out how many of the values in the vector are `TRUE` AND `FALSE`. You can use `sum` because R saves logical vectors at a basic level as `0` for `FALSE` and `1` for `TRUE`. Therefore, if you add up all the values in a logical vector, you're adding up the number of observations with the value `TRUE`.

In the example, you can use these functions to find out how many males and females are in the dataset:

```
sum(is_male)
```

```
## [1] 107
```

```
sum(!is_male)
```

```
## [1] 93
```

```
table(is_male)
```

```
## is_male
## FALSE  TRUE
##    93   107
```

Note that you could also achieve the same thing with `dplyr` functions. For example, you could use `mutate` with a logical statement to create an `is_male` column in the `nepali` dataframe, then group by the new `is_male` column and count the number of observations in each group using `count`:

```
library(dplyr)
nepali %>%
```

```
  mutate(is_male = sex == "Male") %>%
  group_by(is_male) %>%
  count()
```

```
## # A tibble: 2 x 2
## # Groups:   is_male [2]
##   is_male     n
##   <lgl>   <int>
## 1 FALSE      93
## 2 TRUE      107
```

We will cover using `summarize`, including with data that has been grouped with `group_by`, later in this chapter.

3.4 Simple statistics functions

3.4.1 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
`range()`	Range (minimum and maximum) of vector
`min()`, `max()`	Minimum or maximum of vector
`mean()`, `median()`	Mean or median of vector
`sd()`	Standard deviation of vector
`table()`	Number of observations per level for a factor vector
`cor()`	Determine correlation(s) between two or more vectors
`summary()`	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")
```

```
##            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 `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.

3.4.2 summarize function

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

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

```
##    mean_wt
## 1 10.18432
```

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

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

```
##   n_children first_id
## 1        200   120011
```

See the "summary function" section of the the RStudio Data Wrangling cheatsheet 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
##   <fct>    <dbl>      <int> <fct>
## 1 Male     10.5         107 120011
## 2 Female    9.82         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.

3.5 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.

3.5.1 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.

3.5.2 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.

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.

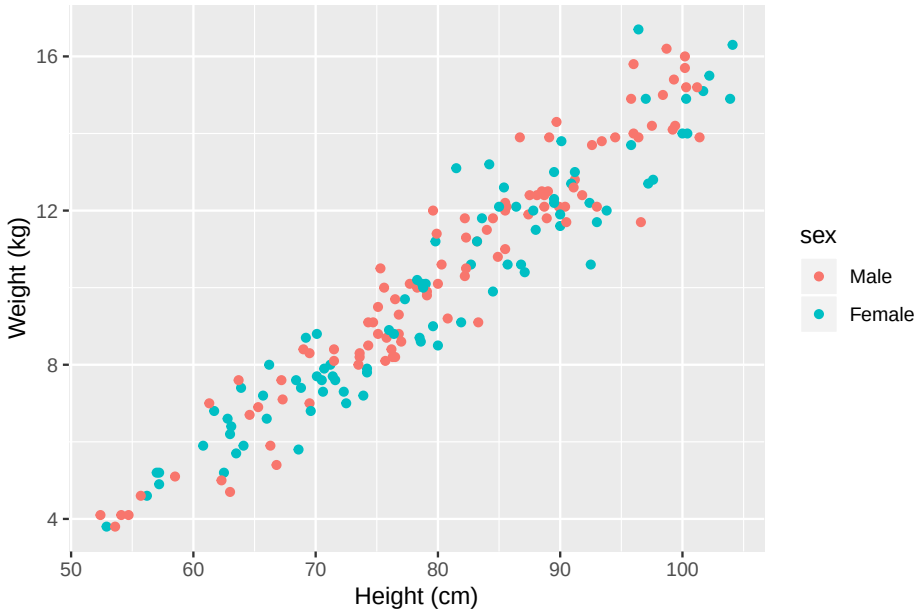


Figure 3.1: Example of how different properties of a plot can show different elements to the data. Here, color indicates gender, position along the x-axis shows height, and position along the y-axis shows weight. This example is a subset of data from the 'nepali' dataset in the 'faraway' package.

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)

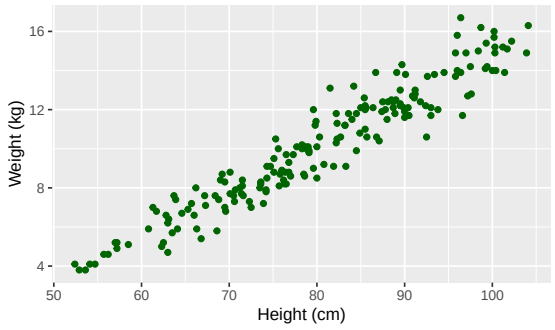
3.5.3 Adding geoms

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)	'geom_histogram'
Scatterplot (2 numeric variables)	'geom_point'
Boxplot (1 numeric variable, possibly 1 factor variable)	'geom_boxplot'
Line graph (2 numeric variables)	'geom_line'

3.5.4 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

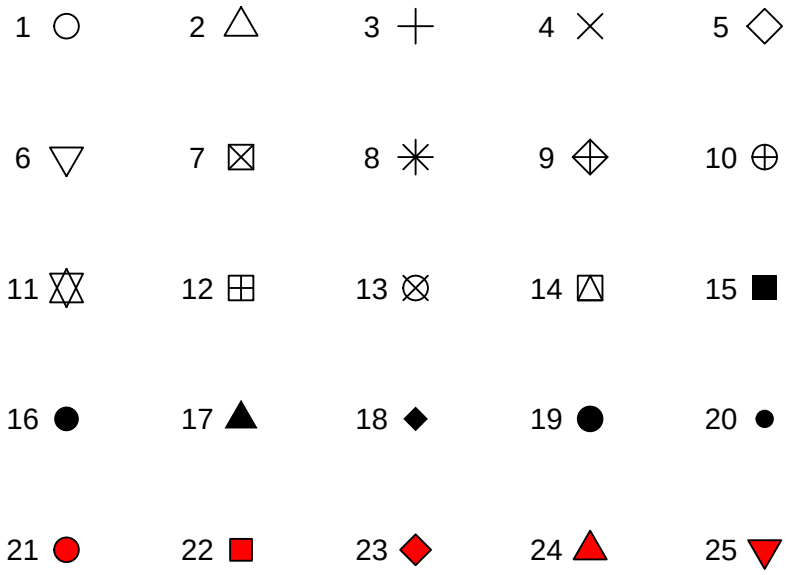


Figure 3.2: Examples of the shapes corresponding to different numeric choices for the 'shape' aesthetic. For all examples, 'color' is set to black and 'fill' to red.

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).

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".

3.5.5 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
'ggtitle'	Plot title
'xlab', 'ylab'	x- and y-axis labels
'xlim', 'ylim'	Limits of x- and y-axis

3.5.6 Example dataset

For the example plots, I'll use a dataset in the `faraway` package called `nepali`. This gives data from a study of the health of a group of Nepalese children.

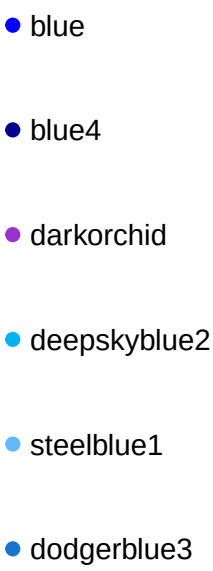


Figure 3.3: Example of available shades of blue in R.

```
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 chile, 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:

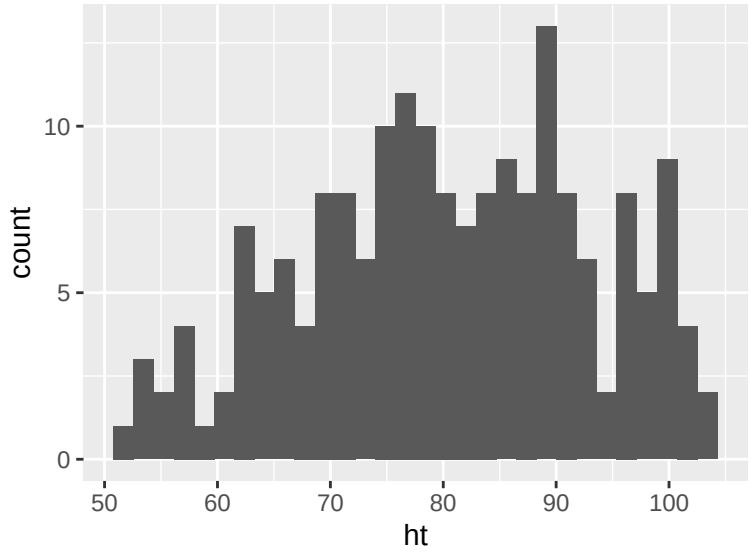


Figure 3.4: Basic example of plotting a histogram with 'ggplot2'. This histogram shows the distribution of heights for the first recorded measurements of each child in the 'nepali' dataset.

```
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
```

3.5.7 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()
```

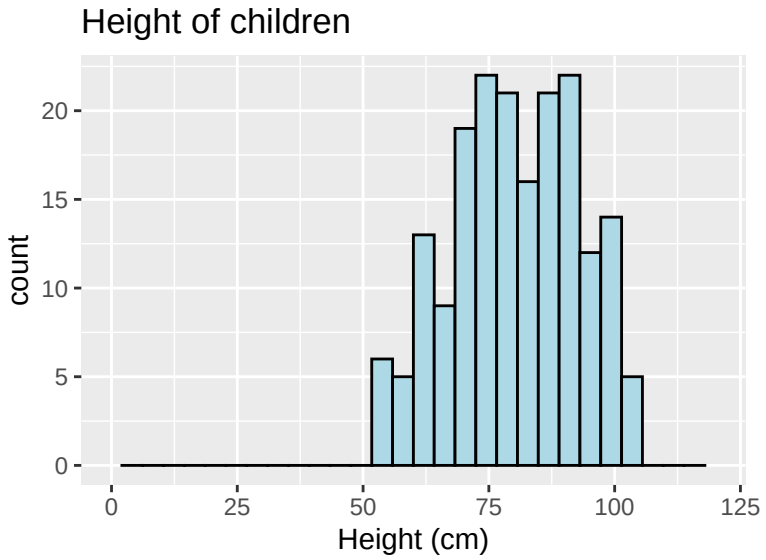


Figure 3.5: Example of adding ggplot elements to customize a histogram.

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))
```

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.

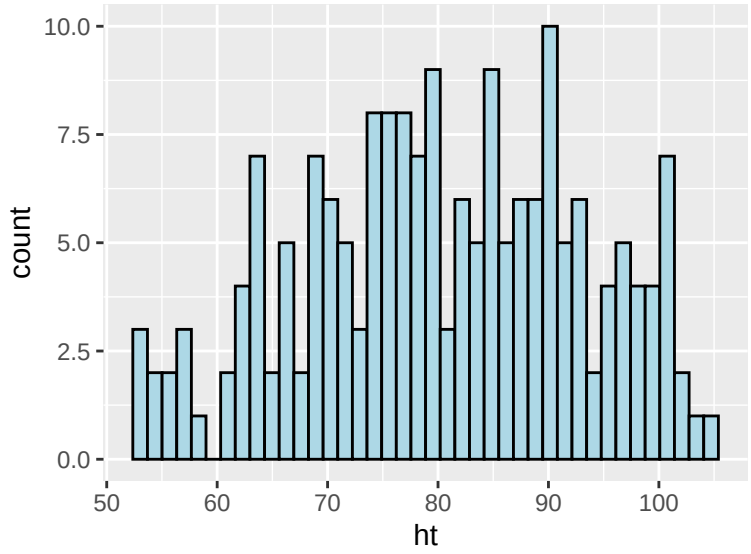


Figure 3.6: Example of using the 'bins' argument to change the number of bins used in a histogram.

```
ggplot(nepali, aes(x = ht)) +
  geom_histogram(fill = "lightblue", color = "black",
                 bins = 40)
```

Similarly, the `binwidth` argument can be used to set the width of bins. Figure 3.7 shows an example of using this function to create a histogram of the Nepali children's heights with binwidths of 10 centimeters (note that this argument is set in the same units as the x variable).

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

3.5.8 Scatterplots

A scatterplot shows how one variable changes as another changes. You can use the `geom_point` geom to create a scatterplot. For example, to create a scatterplot of height versus age for the Nepali data (Figure 3.8), you can run the following code:

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

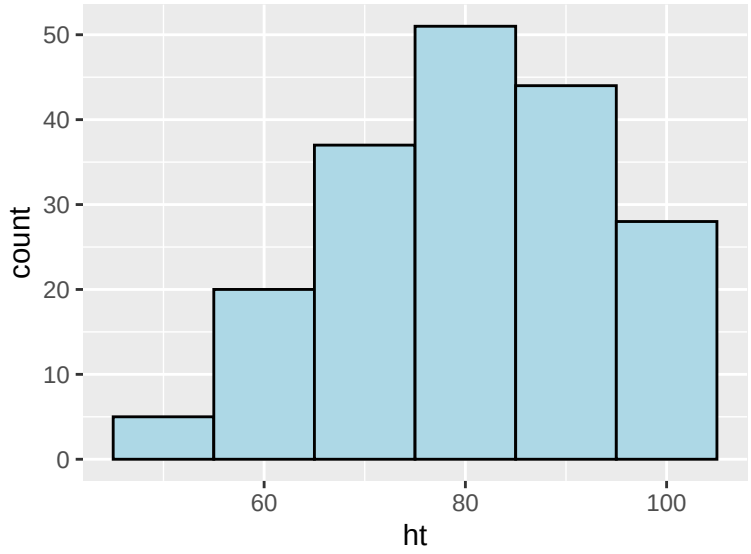


Figure 3.7: Example of using the 'binwidth' argument to set the width of each bin used in a histogram.

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)")
```

3.5.9 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

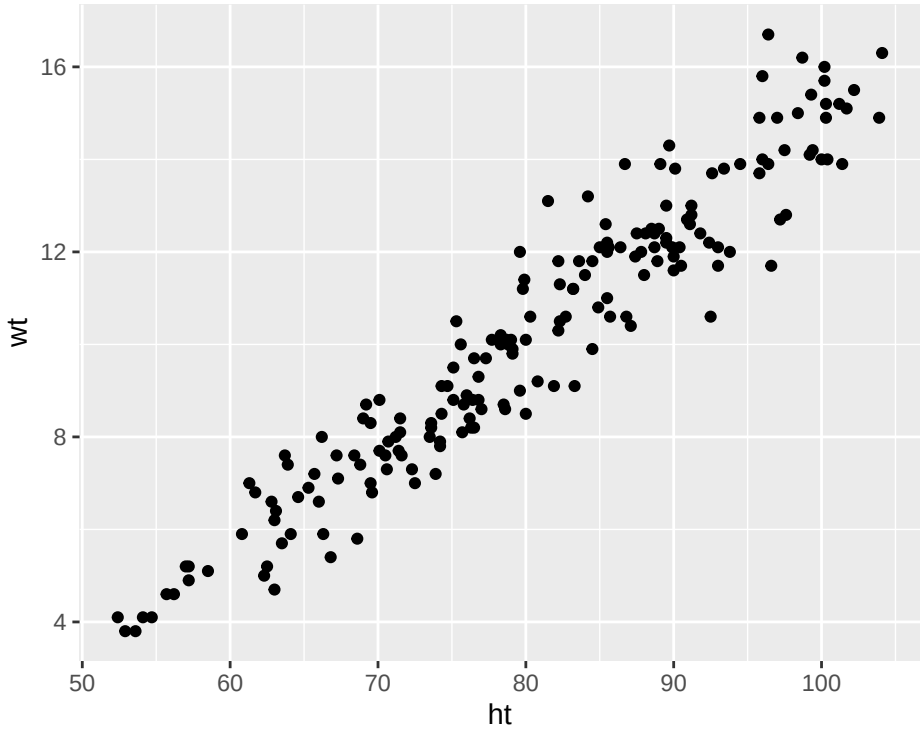


Figure 3.8: Example of creating a scatterplot. This scatterplot shows the relationship between children's heights and weights within the nepali dataset.

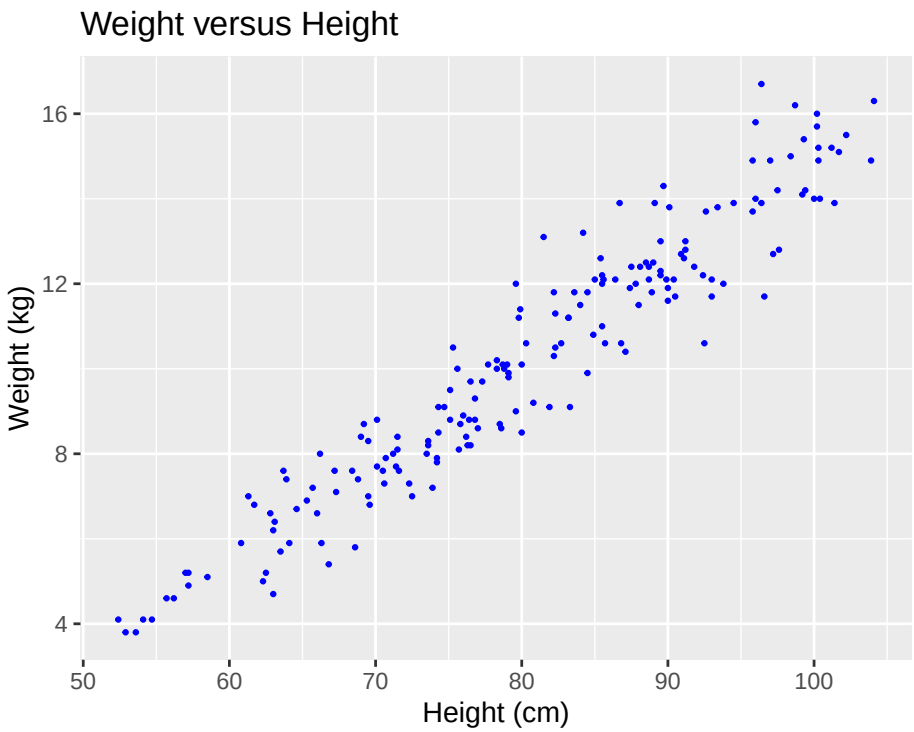


Figure 3.9: Example of adding ggplot elements to customize a scatterplot.

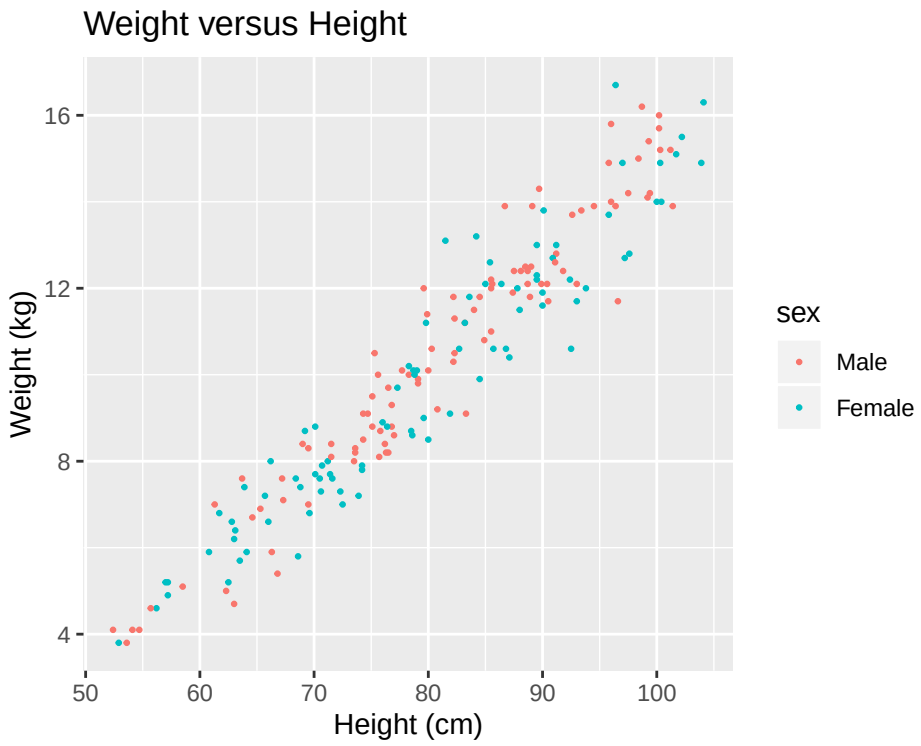


Figure 3.10: Example of mapping color to an element of the data in a scatterplot.

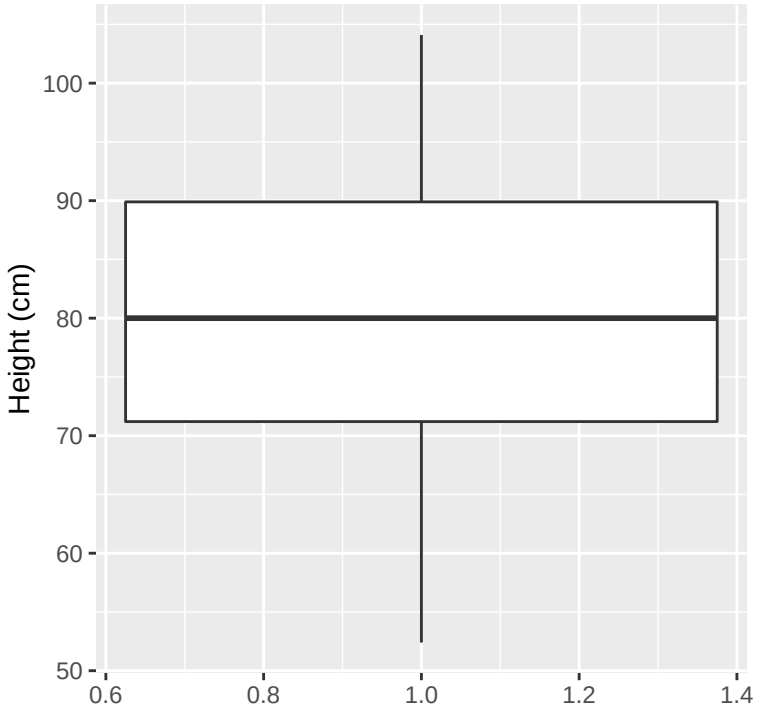


Figure 3.11: Example of creating a boxplot. The example shows the distribution of height data for children in the nepali dataset.

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.

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

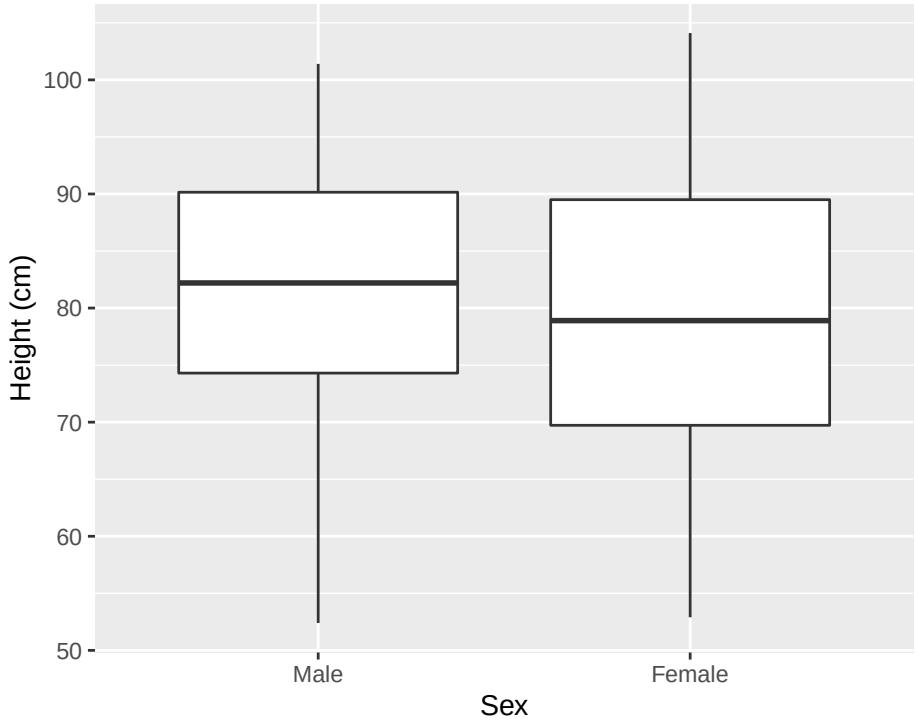


Figure 3.12: Example of creating separate boxplots, divided by a categorical grouping variable in the data.

3.6 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 `library()` and then load the data using `data()`:

```
## Uncomment the next line if you need to install the package
# install.packages("faraway")
library(faraway)
data("worldcup")
```

Check out the help file for the `worldcup` dataset to find out more about the data. (Note: Only datasets that are parts of packages will have help files.)

```
?worldcup
```

Check out the data a bit:

```
str(worldcup)
```

```
## 'data.frame':    595 obs. of  7 variables:
##  $ Team    : Factor w/ 32 levels "Algeria","Argentina",..: 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.   : 0.000   Min.   : 0.0000
##  1st Qu.: 29.00   1st Qu.: 1.000   1st Qu.: 0.0000
##  Median : 61.00   Median : 3.000   Median : 0.0000
##  Mean   : 84.52   Mean   : 4.192   Mean   : 0.6672
##  3rd Qu.:115.50   3rd Qu.: 6.000   3rd Qu.: 0.0000
##  Max.   :563.00   Max.   :34.000   Max.   :20.0000
##
```

3.6.2 Exploring the data using logical statements

Next, try checking out the data using logical statements and some of the `dplyr` code we covered last week (`filter` and `arrange`, for example), to help you answer the following questions:

- What is the range of time that players spent in the game?
- Which player or players played the most time in this World Cup?
- How many players are goalies in this dataset?
- Create a new R object named `brazil_players` that is limited to the players in this dataset that are (1) on the Brazil team and (2) not goalies.

If you have additional time, look over the "Data Manipulation" cheatsheet available in RStudio's Help section. Make a list of questions you would like to figure out from this example data, and start to plan out how you might be able to answer those questions using functions from `dplyr`. Write the related code and see if it works.

3.6.2.1 Example R code:

To figure out the range of time, you could use `arrange` twice, once with `desc` and once without, to figure out the maximum and minimum values

```
# Minimum time
arrange(worldcup, Time) %>%
  select(Time) %>%
  slice(1)
```

```
##   Time
## 1    1
```

```
# Maximum time
arrange(worldcup, desc(Time)) %>%
  select(Time) %>%
  slice(1)
```

```
##   Time
## 1  570
```

Later, we will learn about the `n()` function, which you can use within piped code to represent the total number of rows in the dataframe. If you'd like to get the full range of the `Time` column in one pipeline of code, you can use `n()` as a reference within `slice`, to pull both the first and last rows of the dataframe:

```
arrange(worldcup, Time) %>%
  select(Time) %>%
  slice(c(1, n()))
```

```
##   Time
## 1    1
## 2  570
```

Finally, as a further hint at things you'll learn later in the lecture, you could also use `min()` and `max()` functions to get the minimum and maximum values of the `Time` column in

the `worldcup` dataframe (remember that you can use the `dataframe$column_name` notation to pull a column from a dataframe). Similarly, you could 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 which player or players played for the most time, there are a few approaches you can take. Here I'm showing two: (1) using `filter` from the `dplyr` package to filter down to rows where where the `Time` for that row equals the maximum play time that you determined from an earlier task (570 minutes); and (2) using the `top_n` function from `dplyr` to pick out the rows with the maximum value (`n = 1`) of the `Time` column (see the help file for `top_n` if you are unfamiliar with this function; we have not covered it in class yet).

```
worldcup %>%
  filter(Time == 570)
```

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

```
worldcup %>%
  top_n(n = 1, wt = Time)
```

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

Note: You may have noticed that you lost the players names when you did this using the `dplyr` pipechain. That's because `dplyr` functions convert the data to a dataframe format that does not include rownames. If you want to keep players' names, you can use a function from the `tibble` package called `rownames_to_column` to move those names from the rownames of the data into a column in the dataframe. Use the `var` parameter of this function to specify what you want the new column to be named. For example:

```
library(tibble)
worldcup %>%
  rownames_to_column(var = "Name") %>%
  filter(Time == 570)
```

```
##           Name    Team   Position Time Shots Passes Tackles Saves
```

```
## 1 Arevalo Rios Uruguay Midfielder  570     5    195       21      0
## 2 Maxi Pereira Uruguay Midfielder  570     5    182       15      0
## 3      Muslera Uruguay Goalkeeper  570     0     75        0     16
```

There are a few ways to figure out how many players are goalies in this dataset. One way is to use `sum()` on a logical vector of whether the player's position is "Goalkeeper":

```
is_goalie <- worldcup$Position == "Goalkeeper"
sum(is_goalie)
```

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

Another way is to use `filter` from `dplyr`, along with a logical statement, to filter the data to only players with the position of "Goalkeeper", and then pipe that filtered subset into the `nrow` function to count the number of rows in the filtered dataframe:

```
worldcup %>%
  filter(Position == "Goalkeeper") %>%
  nrow()
```

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

Next, create a new R object named `brazil_players` that is limited to the players in this dataset that are (1) on the Brazil team and (2) not goalies. You can use a logical statement to filter to rows that meet both these conditions by joing two logical statements in the `filter` function with an `&`:

```
brazil_players <- worldcup %>%
  filter(Team == "Brazil" & Position != "Goalkeeper")
head(brazil_players)
```

```
##     Team   Position Time Shots Passes Tackles Saves
## 1 Brazil Midfielder   82     0     42       1     0
## 2 Brazil   Defender  310    11    215       6     0
## 3 Brazil Midfielder  140     5     57       6     0
## 4 Brazil    Forward  418     9     89       4     0
## 5 Brazil   Defender   33     0      6       4     0
## 6 Brazil Midfielder  450     3    299      11     0
```

3.6.3 Exploring the data using simple statistics and `summarize`

Next, try checking out the data using some basic commands for simple statistics, like `mean()`, `range()`, `max()`, and `min()`, as well as the `summarize` and `group_by` functions from the `dplyr` package. Try to answer the following questions:

- What is the mean number of saves that players made?

- What is the mean number of saves just among the goalkeepers?
- Did players from any position other than goalkeeper make a save?
- How many players were there in each position?
- How many forwards were there on each team? Which team had the most shots in total among all its forwards?
- Which team(s) had the defender with the most tackles?

If you have extra time, continuing using the "Data Wrangling" cheatsheet to come up with some other ideas for how you can explore this data, and write up and test code to do that.

3.6.3.1 Example R code:

To calculate the mean number of saves among all the players, use the `mean` function, either by itself or within a `summarize` call:

```
mean(worldcup$Saves)
```

```
## [1] 0.6672269
```

```
worldcup %>%
  summarize(mean_saves = mean(Saves))
```

```
##   mean_saves
## 1  0.6672269
```

There are a few ways to figure out the mean number of saves just among the goalkeepers. One way is to filter the dataset to only goalies and then use `summarize` to calculate the mean number of saves in this filtered subset of the data:

```
worldcup %>%
  filter(Position == "Goalkeeper") %>%
  summarize(mean_saves = mean(Saves))
```

```
##   mean_saves
## 1   11.02778
```

The next question is if players from any position other than goalkeeper made a save. One way to figure this out is to group the data by position and then summarize the maximum number of saves. Based on this, it looks like there were not saves from players in any position except goalie:

```
worldcup %>%
  group_by(Position) %>%
  summarize(max_saves = max(Saves))
```

```
## # A tibble: 4 x 2
##   Position   max_saves
##   <fct>          <int>
## 1 Defender           0
## 2 Forward            0
## 3 Goalkeeper        20
## 4 Midfielder         0
```

To figure out how many players were there in each position, you can can group the data by position and then use the `n` function from `dplyr` to count the number of observations in each group:

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

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

For the next set of questions, you can filter the data to only forwards, then group by team to use `summarize` to count up the number of forwards on each team. You can also use the same `summarize` call to figure out the total number of shots by all forwards on each team. To figure out which team had the most shots in total among all its forwards, you can use the `arrange` function to re-order the data from the team with the most total shots to the least. It turns out that Uruguay had the most shots by forwards on its team, with a total of 46 shots.

```
worldcup %>%
  filter(Position == "Forward") %>%
  group_by(Team) %>%
  summarize(n_forwards = n(),
            total_forward_shots = sum(Shots)) %>%
  arrange(desc(total_forward_shots))
```

```
## # A tibble: 32 x 3
##    Team        n_forwards total_forward_shots
##    <fct>            <int>               <int>
##  1 Uruguay              5                  46
##  2 Argentina            6                  45
##  3 Germany              6                  41
##  4 Netherlands          5                  34
```

```
##  5 Spain                3                  33
##  6 Ghana                5                  32
##  7 Portugal             4                  28
##  8 Paraguay             5                  25
##  9 Brazil               4                  23
## 10 USA                  5                  21
## # ... with 22 more rows
```

To figure out which team(s) had the defender with the most tackles, you can filter to only defenders and then use the `top_n` function to identify the players with the top number of tackles. It turns out these players were on the England, Germany, and Chile teams.

```
worldcup %>%
  filter(Position == "Defender") %>%
  top_n(n = 1, wt = Tackles)
```

```
##      Team Position Time Shots Passes Tackles Saves
## 1 England Defender  357     3    173      17     0
## 2 Germany Defender  540     0    360      17     0
## 3   Chile Defender  306     6    178      17     0
```

3.6.4 Exploring the data using basic plots #1

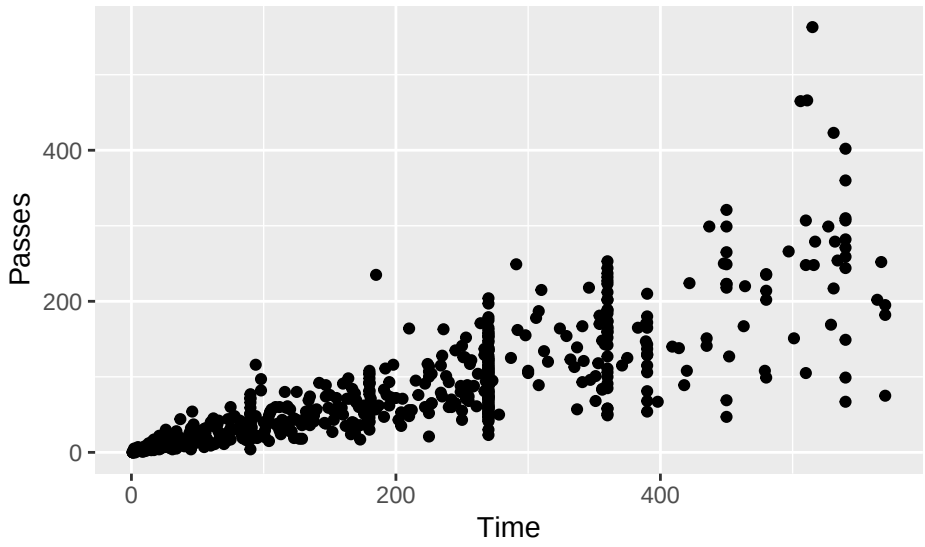
Use some basic plots to check out this data. Try the following:

- Create a scatterplot of the `worldcup` data, where each point is a player, the x-axis shows the amount of time the player played in the World Cup, and the y-axis shows the number of passes the player had. Try writing the code both with and without "piping in" the data you want to plot into the `ggplot` function.
- Create the same scatterplot, but have each point in the scatterplot show that player's position using some aesthetic besides the x or y position (e.g., color, point shape). Add "rug plots" to the margins.
- Create a scatterplot of number of shots (x-axis) versus number of tackles (y-axis) for **just** players on one of the four teams that made the semi-finals (Spain, Netherlands, Germany, Uruguay). Use color to show player's position and shape to show player's team. (Hint: you will want to use some `dplyr` code to clean the data before plotting to do this.)
- Create a scatterplot of player time versus passes. Use color to show whether the player was on one of the top 4 teams or not. (Hint: Again, you'll want to use some `dplyr` code before plotting to do this.) For an extra challenge, also try adding each player's name on top of each point. (Hint: check out the `rownames_to_column` function from the `tibble` package to help with this.)
- Did you notice any interesting features of the data when you did any of the graphs in this section?

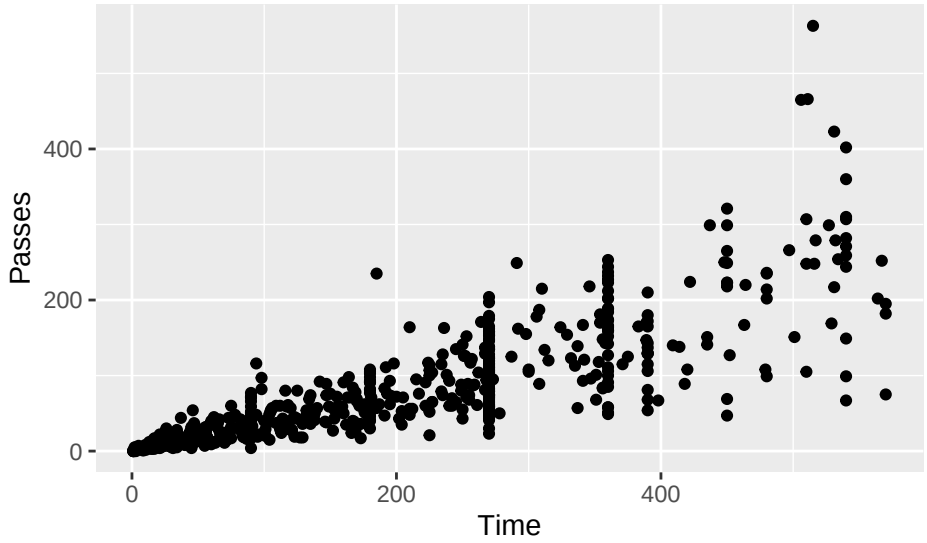
3.6.4.1 Example R code:

Create a scatterplot of `Time` versus `Passes`.

```
# Without piping
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes))
```

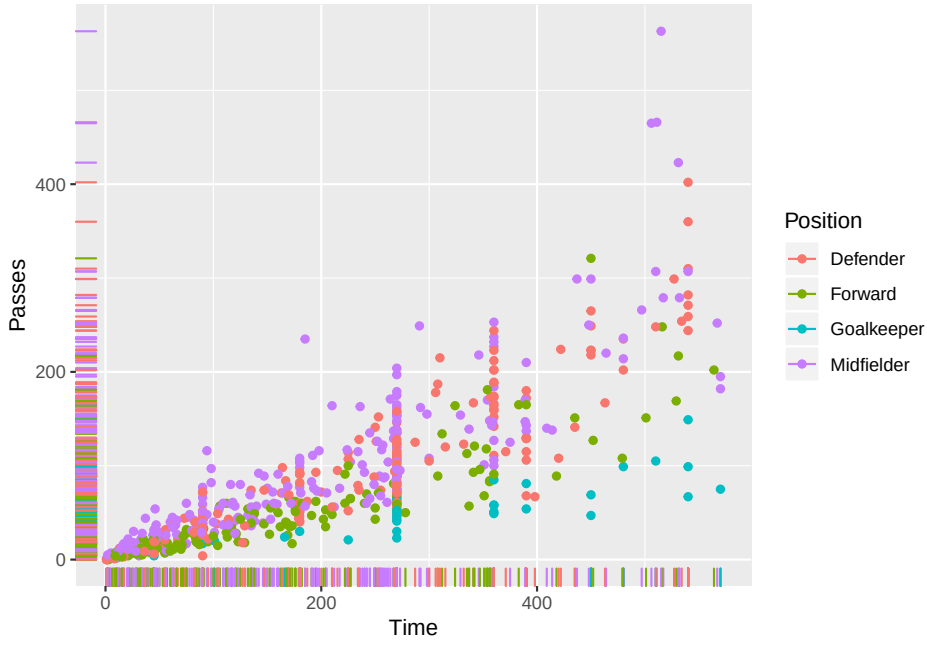



```
# With piping
worldcup %>%
  ggplot() +
  geom_point(mapping = aes(x = Time, y = Passes))
```



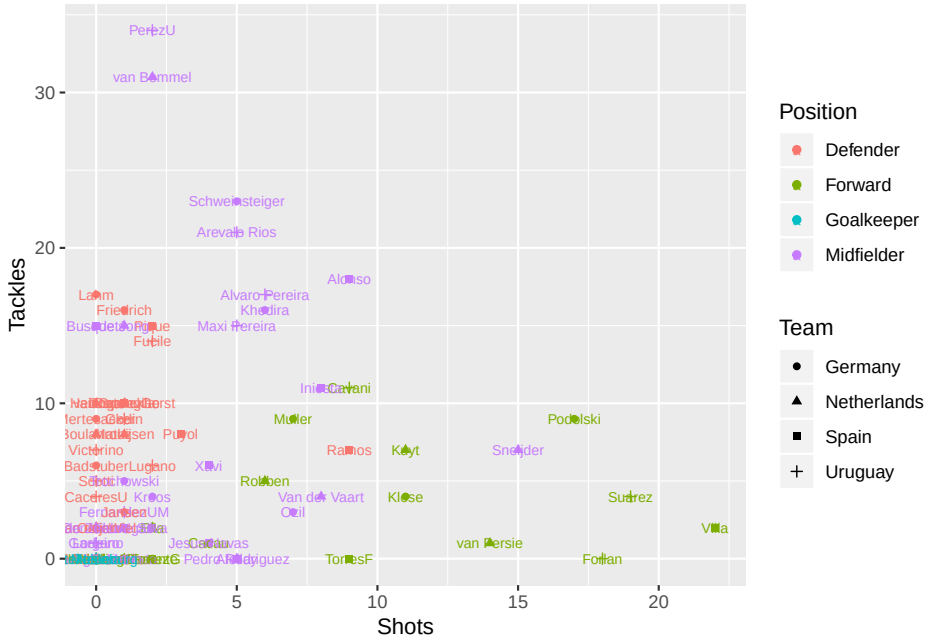
Create the same scatterplot, but have each point in the scatterplot show that player's position.

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



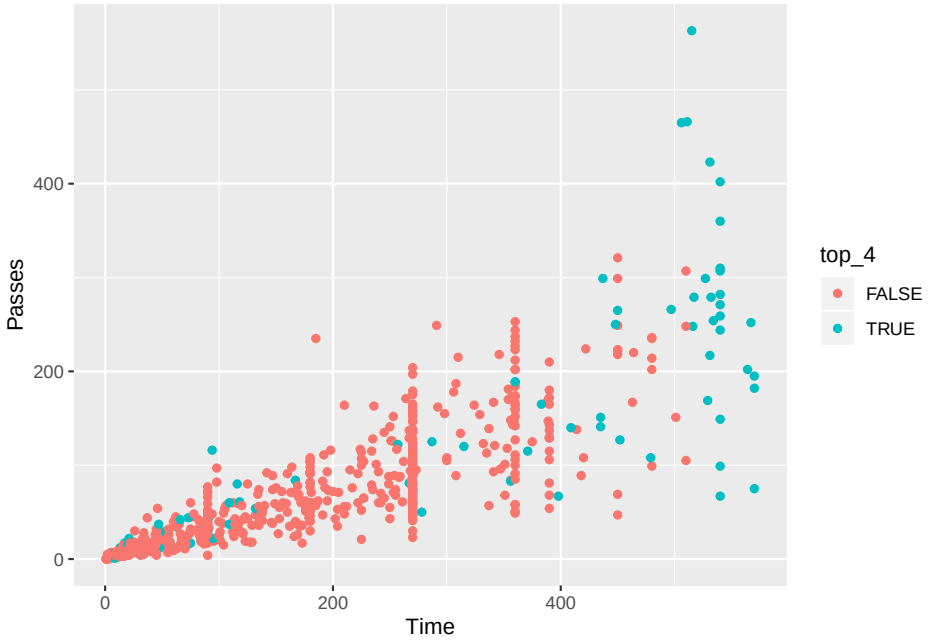
Create a scatterplot of number of shots (x-axis) versus number of tackles (y-axis) for **just** players on one of the four teams that made the semi-finals (Spain, Netherlands, Germany, Uruguay). Use color to show player's position and shape to show player's team. For an extra challenge, also try adding each player's name on top of each point.

```
worldcup %>%
  rownames_to_column(var = "Name") %>%
  filter(Team %in% c("Spain", "Netherlands", "Germany", "Uruguay")) %>%
  ggplot() +
  geom_point(aes(x = Shots, y = Tackles, color = Position, shape = Team)) +
  geom_text(mapping = aes(x = Shots, y = Tackles,
                          color = Position, label = Name),
            size = 2.5)
```



Create a scatterplot of player time versus passes. Use color to show whether the player was on one of the top 4 teams or not.

```
worldcup %>%
  mutate(top_4 = Team %in% c("Spain", "Netherlands", "Germany", "Uruguay")) %>%
  ggplot() +
  geom_point(aes(x = Time, y = Passes, color = top_4))
```



3.6.5 Exploring the data using basic plots #2

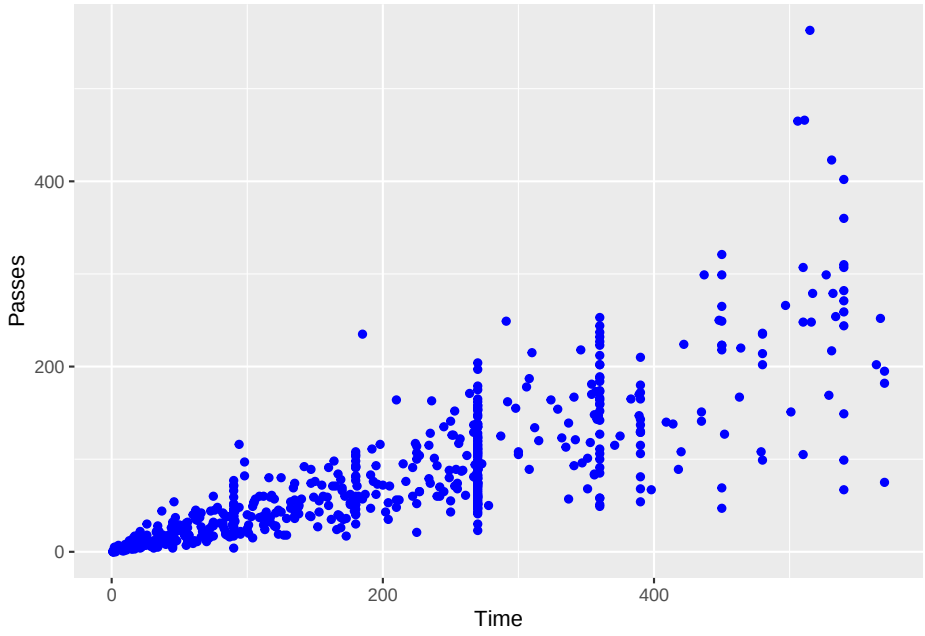
Go back to the code you used in the previous section to create a scatterplot of the `worldcup` data, where each point is a player, the x-axis shows the amount of time the player played in the World Cup, and the y-axis shows the number of passes the player had. Try the following modifications:

- Make all the points blue.
- Google "R colors" to find a list of color names in R. Pick your favorite and make all the points in the scatterplot that color.
- Change the size of the points to make them smaller (hint: check out the `size` aesthetic).
- Make it so the color of the points shows the player's position and all the points are slightly transparent.
- Change the title of the x-axis to "Time (minutes)" and the y-axis to "Number of passes".
- Add the title "World Cup statistics" and the subtitle "2010 World Cup".

3.6.5.1 Example R code:

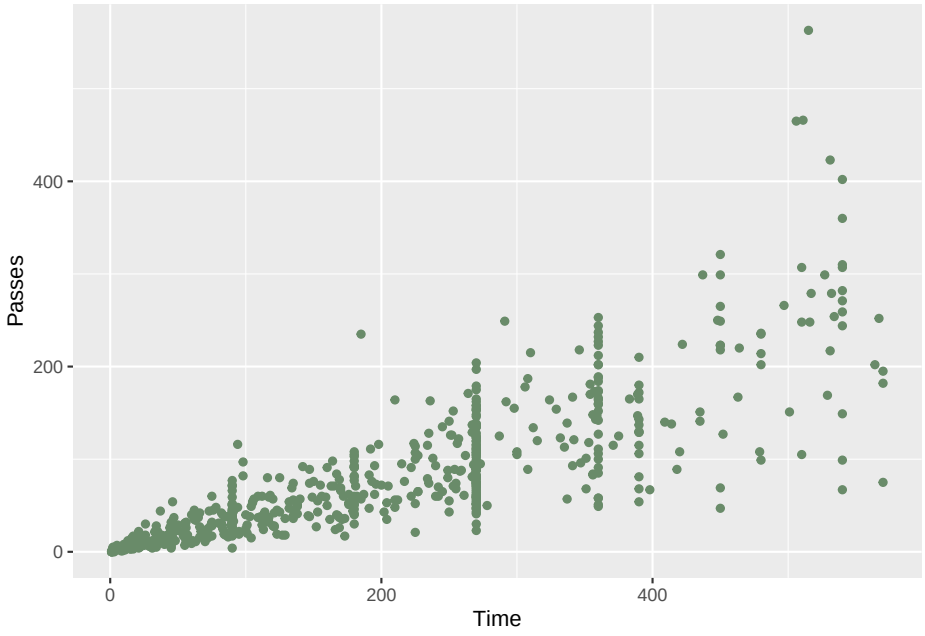
Make all the points blue.

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes),
             color = "blue")
```



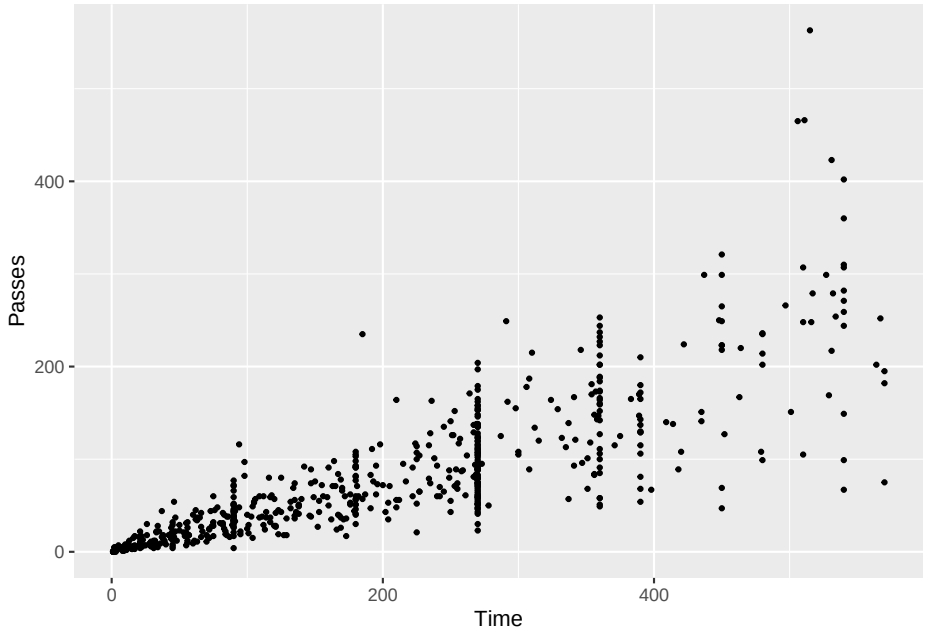
Google "R colors" to find a list of color names in R. Pick your favorite and make all the points in the scatterplot that color.

```
# Make the points "darkseagreen4"
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes),
             color = "darkseagreen4")
```



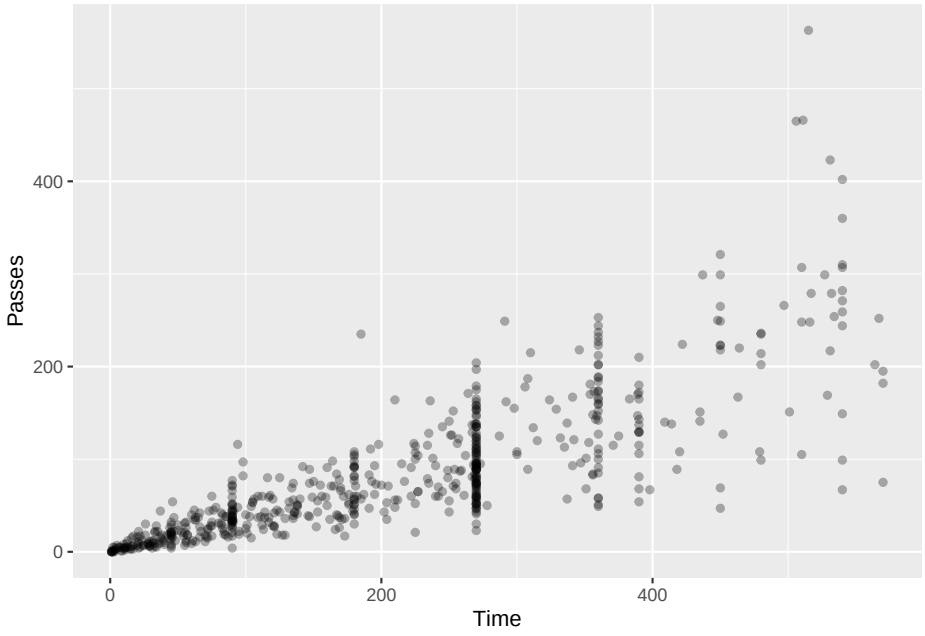
Change the size of the points to make them smaller (hint: check out the `size` aesthetic).

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes),
             size = 0.8)
```



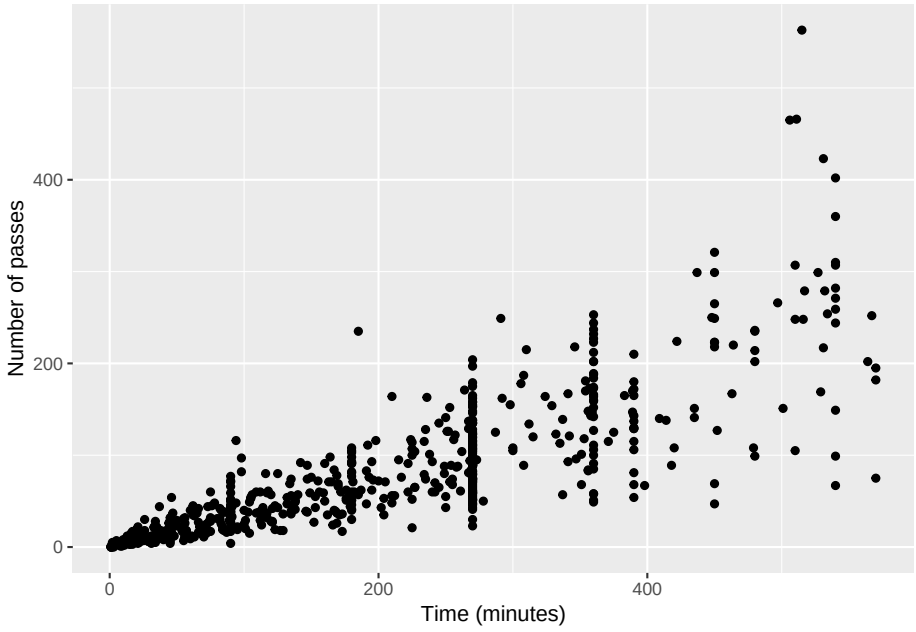
Make it so the color of the points shows the player's position and all the points are slightly transparent.

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes),
             alpha = 0.3)
```



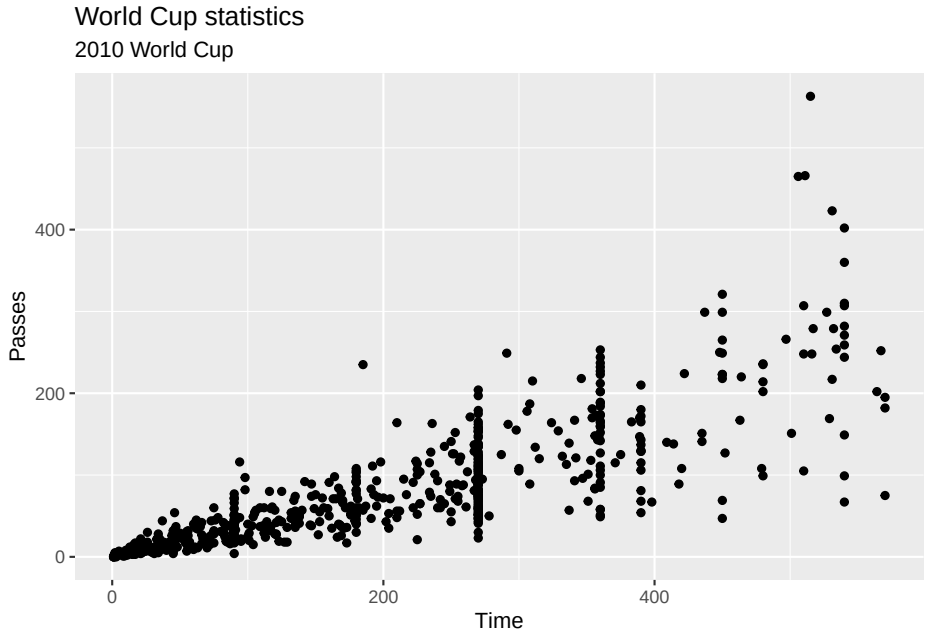
Change the title of the x-axis to "Time (minutes)" and the y-axis to "Number of passes".

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes)) +
  labs(x = "Time (minutes)", y = "Number of passes")
```



Add the title “World Cup statistics” and the subtitle “2010 World Cup”.

```
ggplot(worldcup) +
  geom_point(mapping = aes(x = Time, y = Passes)) +
  ggtitle("World Cup statistics",
          subtitle = "2010 World Cup")
```



3.6.6 Exploring the data using basic plots #3

Try out creating some plots using the "statistical" geoms to check out this data. Try the following:

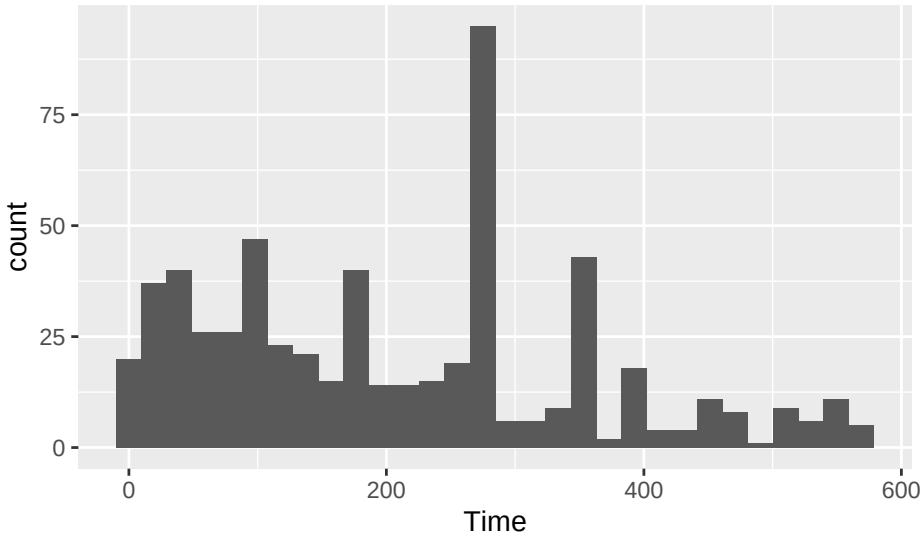
- Plot histograms of all the numeric variables (`Time`, `Shot`, `Passes`, `Tackles`, `Saves`) in the dataset.
- Try customizing the number of bins used for one of the histograms plotted in the previous step.
- Try using constant values for some of the aesthetics (e.g., customize the color and the fill) of the histogram created in the previous step.
- Create a boxplot of `Shots` by position.
- Create a `top_teams` subset with just the four teams that made the semi-finals in the 2010 World Cup (Spain, the Netherlands, Germany, and Uruguay). Plot boxplots of `Shots` and `Saves` by team for just these teams.
- Create a histogram using data only from the four top teams for the amount of time each player played. Use the color aesthetic of the histogram to show team.

3.6.6.1 Example R code

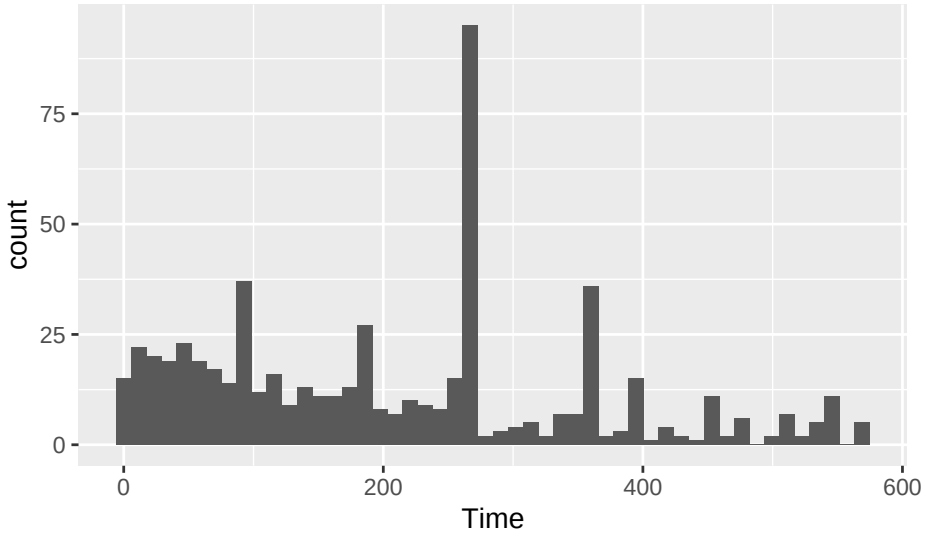
Use histograms to explore the distribution of different variables. If you want to change the number of bins in the histogram, try playing around with the `bins` and `binwidth` arguments. You can use the `bins` argument to say how many bins you want (e.g., `bins = 50`). You can use the `binwidth` argument to say how wide you want the bins to be (e.g., `binwidth = 10` if you wanted bins to be 10 units wide, in the units of the variable

mapped to the `x` aesthetic. Try using `fill` and `color` to change the appearance of the plot. Google "R colors" and search the images to find links to listings of different R colors.

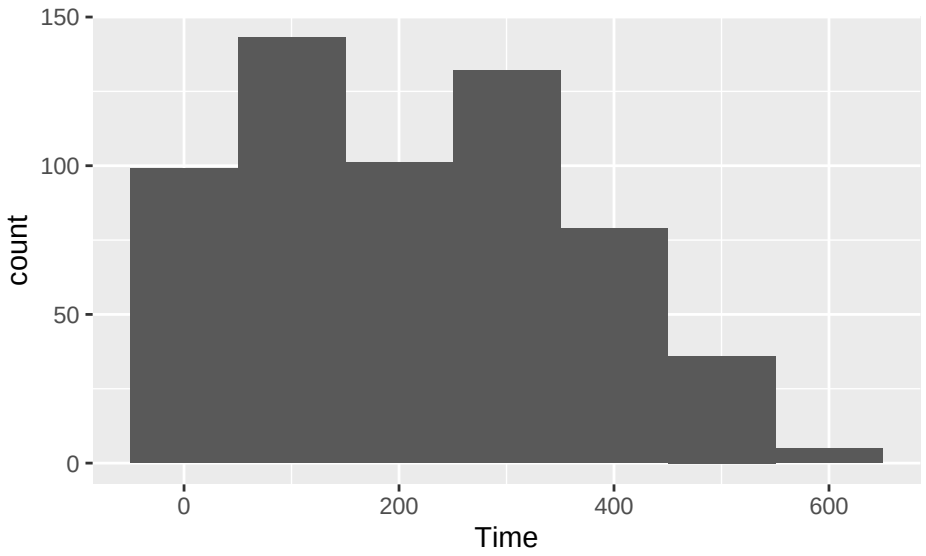
```
ggplot(worldcup, aes(x = Time)) +
  geom_histogram()
```



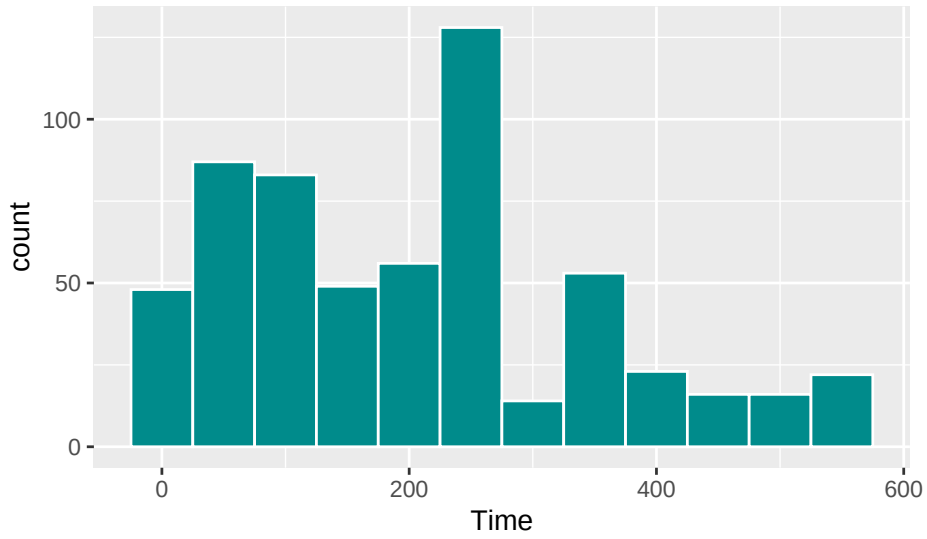

```
ggplot(worldcup, aes(x = Time)) +
  geom_histogram(bins = 50)
```




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

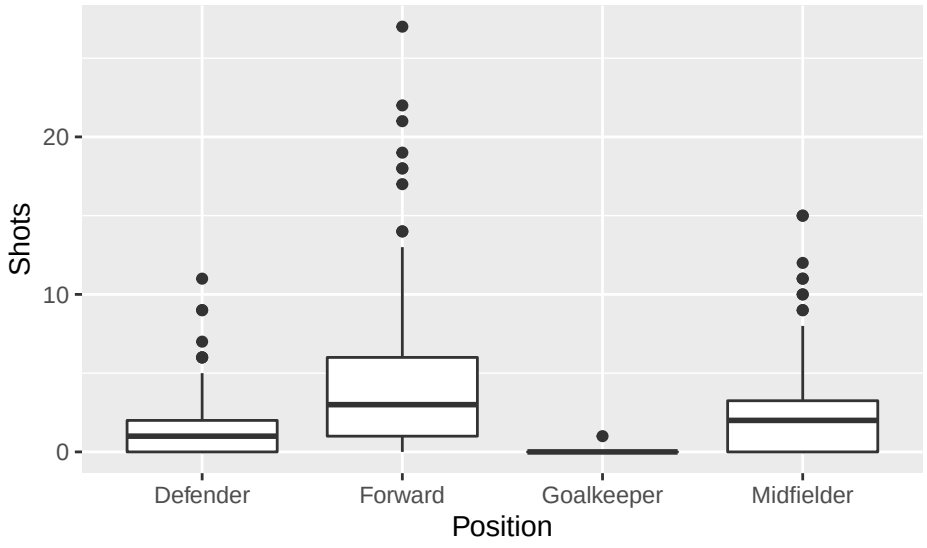



```
ggplot(worldcup, aes(x = Time)) +
  geom_histogram(binwidth = 50, color = "white", fill = "cyan4")
```



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

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

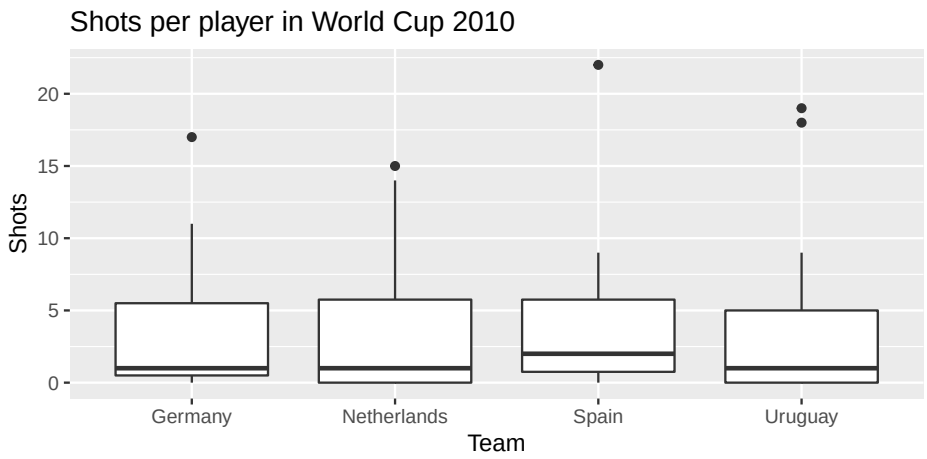


The top four teams in this World Cup were Spain, the Netherlands, Germany, and Uruguay. Create a subset with just the data for these four teams:

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

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

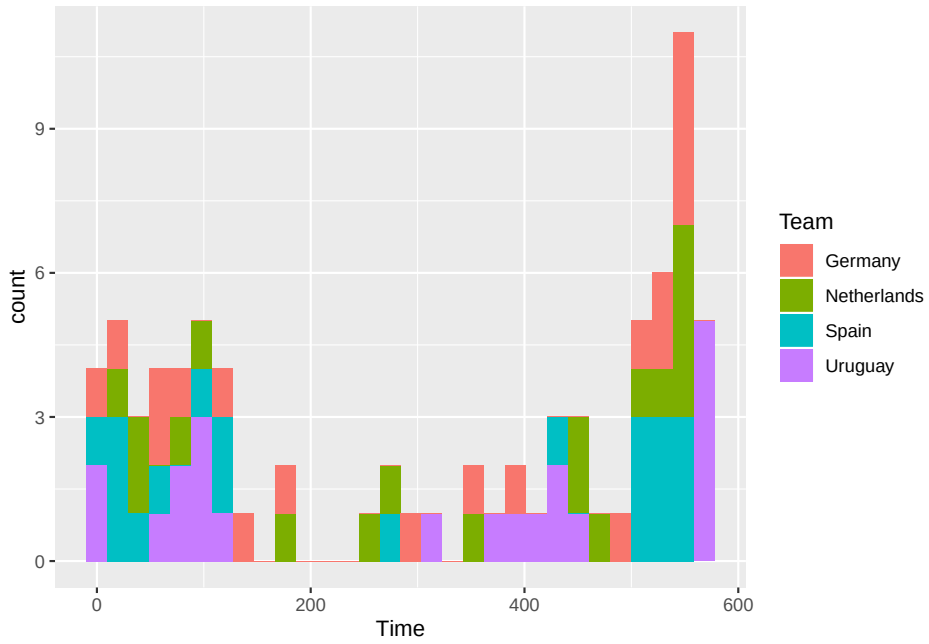
```
ggplot(top_teams, aes(x = Team, y = Shots)) +
  geom_boxplot() +
  ggtitle("Shots per player in World Cup 2010")
```



Create a histogram using data only from the four top teams for the amount of time each player played. Use the color aesthetic of the histogram to show team.

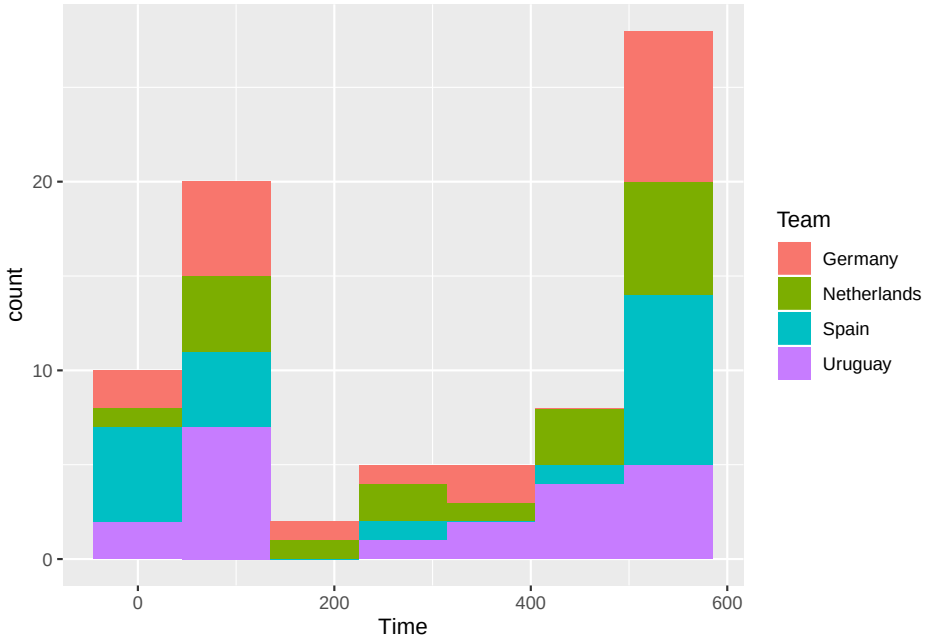
```
ggplot(data = top_teams) +
  geom_histogram(aes(x = Time, fill = Team))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Note that you can also explore other values for `geom_histogram` arguments. For example, you could change the binwidths to be 90 minutes (since games are 90 minutes).

```
ggplot(data = top_teams) +
  geom_histogram(aes(x = Time, fill = Team), binwidth = 90)
```


count
20
10
0
0
200
400
600
Time
Team
Germany
Netherlands
Spain
Uruguay

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 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.

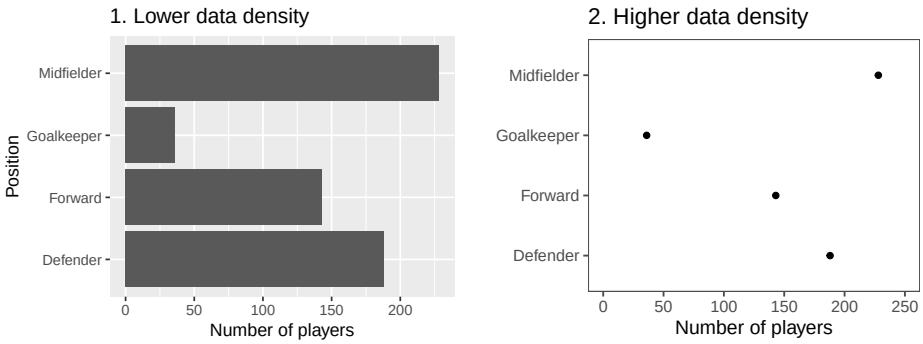


Figure 4.1: Example of plots with lower (left) and higher (right) data-to-ink ratios. Each plot shows the number of players in each position in the worldcup dataset from the faraway package.

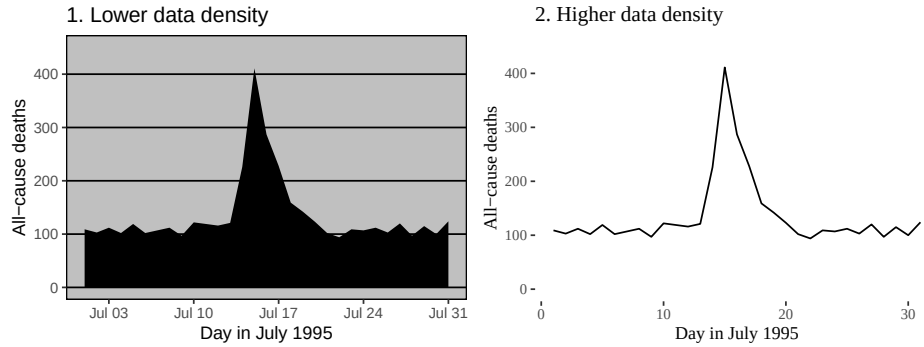


Figure 4.2: Example of plots with lower (left) and higher (right) data-to-ink ratios. Each plot shows daily mortality in Chicago, IL, in July 1995 using the chicagoNMMAPS data from the dlnm package.

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

- `theme_bw`
- `theme_minimal`
- `theme_void`

You can find more themes in packages that extend `ggplot2`. The `ggthemes` package, in particular, has some excellent additional themes.

Figures 4.3 shows some examples of the effects of using different themes. All show the same information– a plot of daily deaths in Chicago in July 1995. The top left graph shows the graph with the default theme. The other plots show the effects of adding different themes, including the black-and-white theme that comes with `ggplot2` (top right) and various themes from the `ggthemes` package. You can even use themes to add some questionable choices for different elements, like the Excel theme (bottom left).

4.3 Meaningful labels

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

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.

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

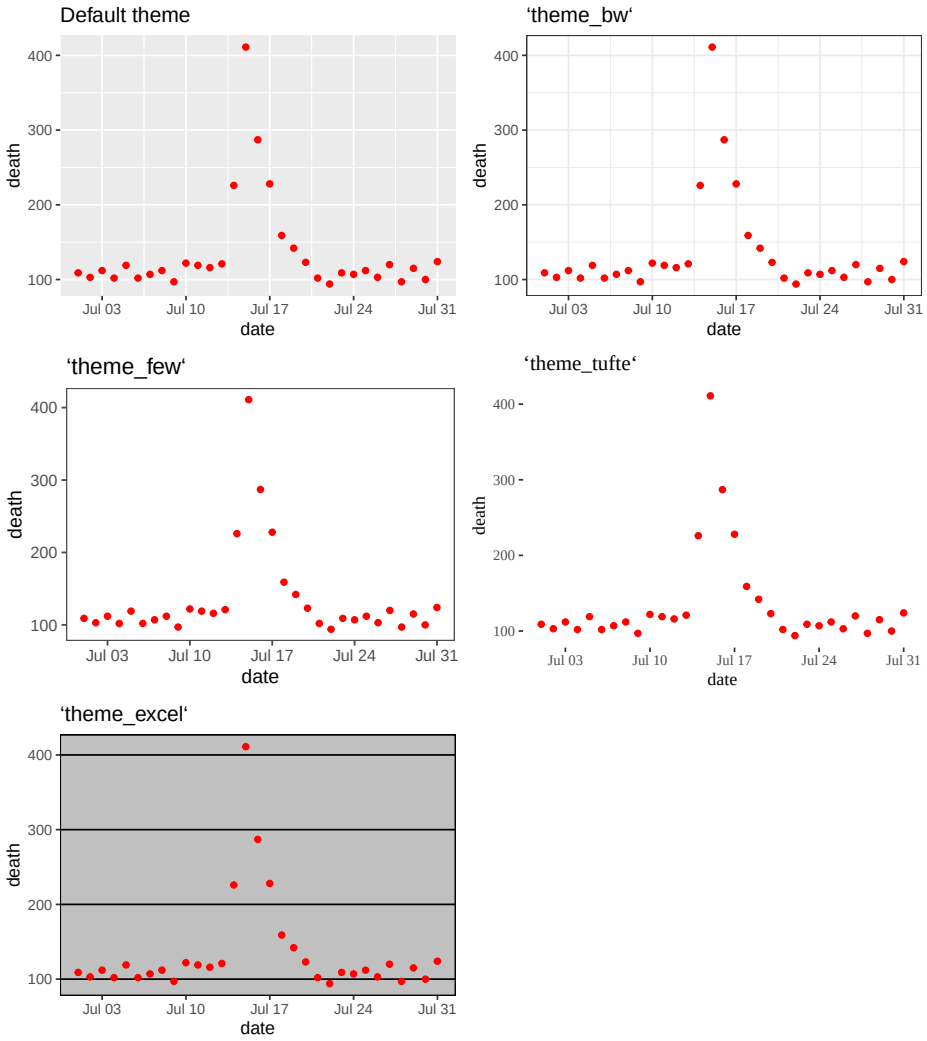


Figure 4.3: Daily mortality in Chicago, IL, in July 1995. This figure gives an example of the plot using different themes.

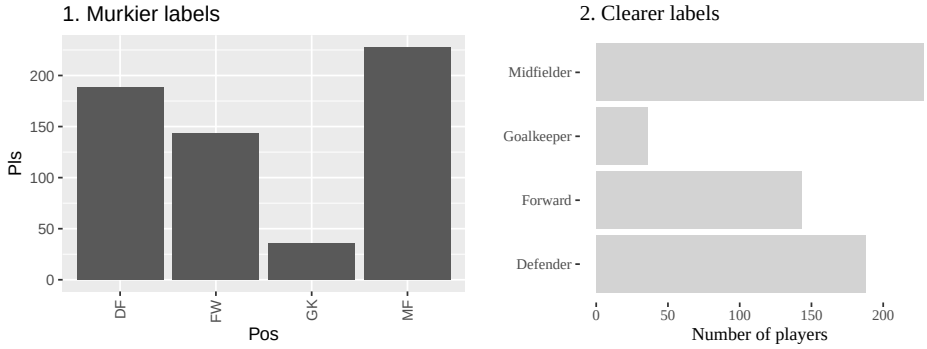


Figure 4.4: The number of players in each position in the worldcup data from the faraway package. Both graphs show the same information, but the left graph has murkier labels, while the right graph has labels that are easier to read and interpret.

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 it 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.

Another useful way to add references is to add a linear or smooth fit to the data, to help clarify trends in the data. Figure 4.6 shows the relationship between passes and shots for Forwards in the `worldcup` dataset. The plot on the right has added a smooth function of the relationship between these two variables.

For scatterplots created with `ggplot2`, you can use the function `geom_smooth` to add a smooth or linear reference line. Here is the code that produces Figure 4.6:

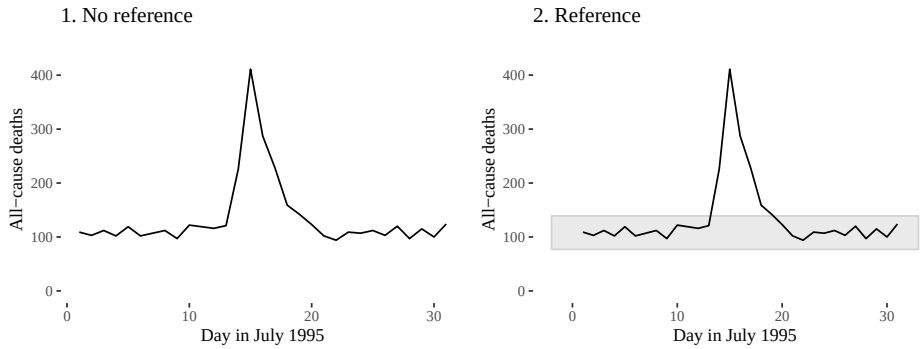


Figure 4.5: Daily mortality during July 1995 in Chicago, IL. In the graph on the right, I have added a shaded region showing the range of daily mortality counts for neighboring years, to show how unusual this event was.

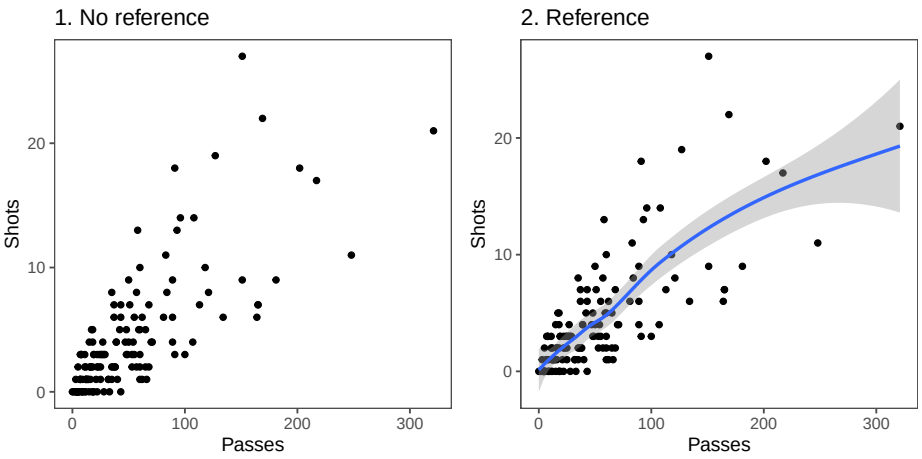


Figure 4.6: Relationship between passes and shots taken among Forwards in the worldcup dataset from the faraway package. The plot on the right has a smooth function added to help show the relationship between these two variables.

```
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.

In the below graphs, the names of the players with the most shots and passes have been added to highlight these unusual points.

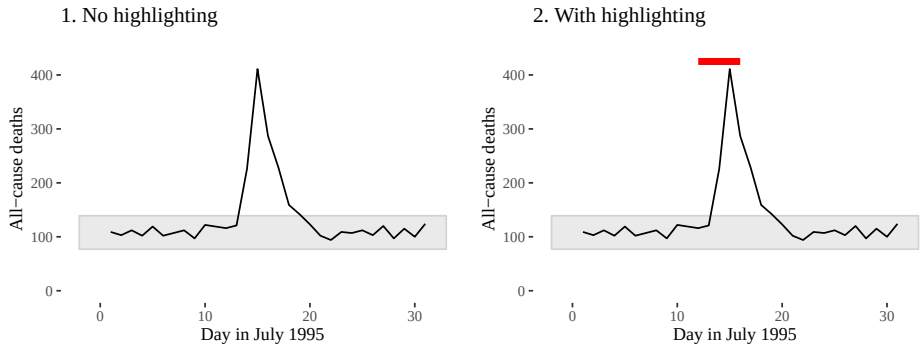
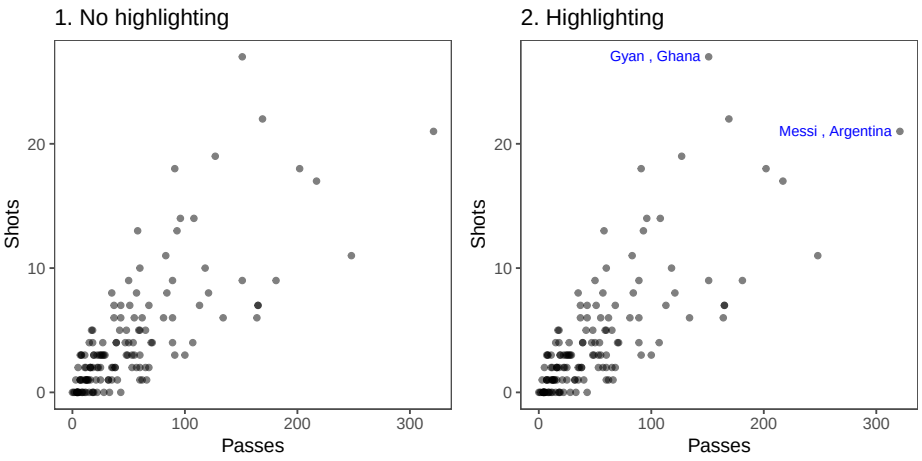


Figure 4.7: Mortality in Chicago, July 1995. In the plot on the right, a thick red line has been added to show the dates of a heat wave.

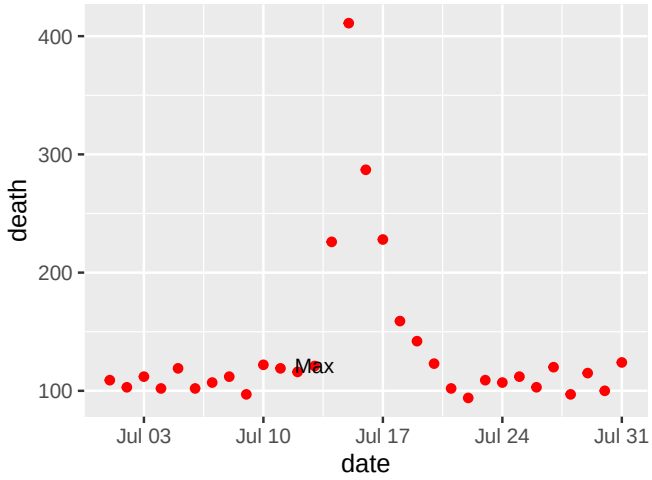


One helpful way to annotate is with text, using `geom_text()`. For this, you'll first need to create a dataframe with the hottest day in the data:

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

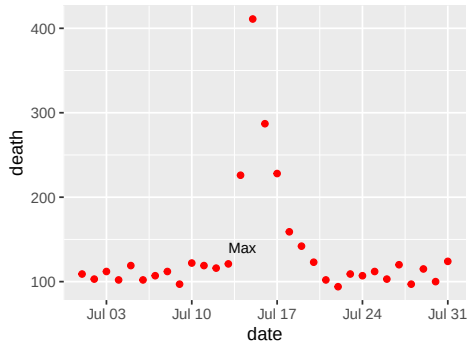
```
##         date time year month doy      dow
## 1 1995-07-13 3116 1995     7 194 Thursday
```

```
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)
```



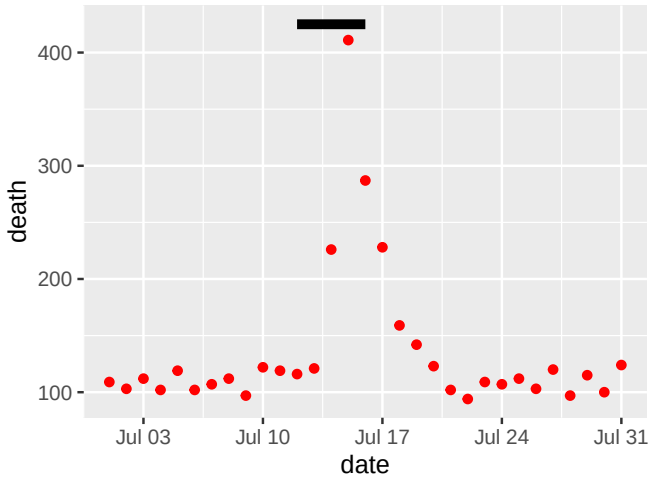
You can also use lines to highlight. For this, it is often useful to create a new dataframe with data for the reference. To add a line for the Chicago heat wave, I've added a dataframe called `hw` with the relevant date range. I'm setting the *y*-value to be high enough (425) to ensure the line will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

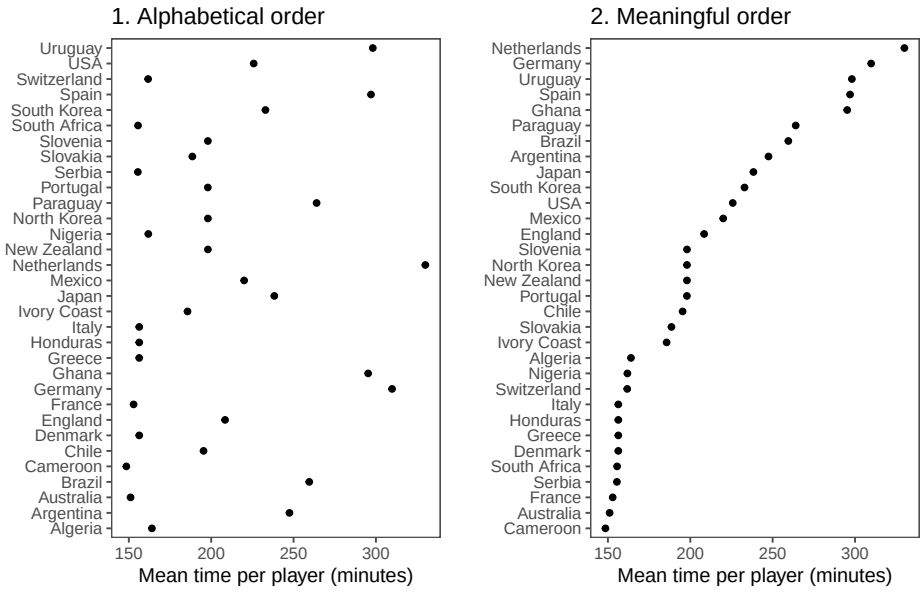
```
b
```



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.



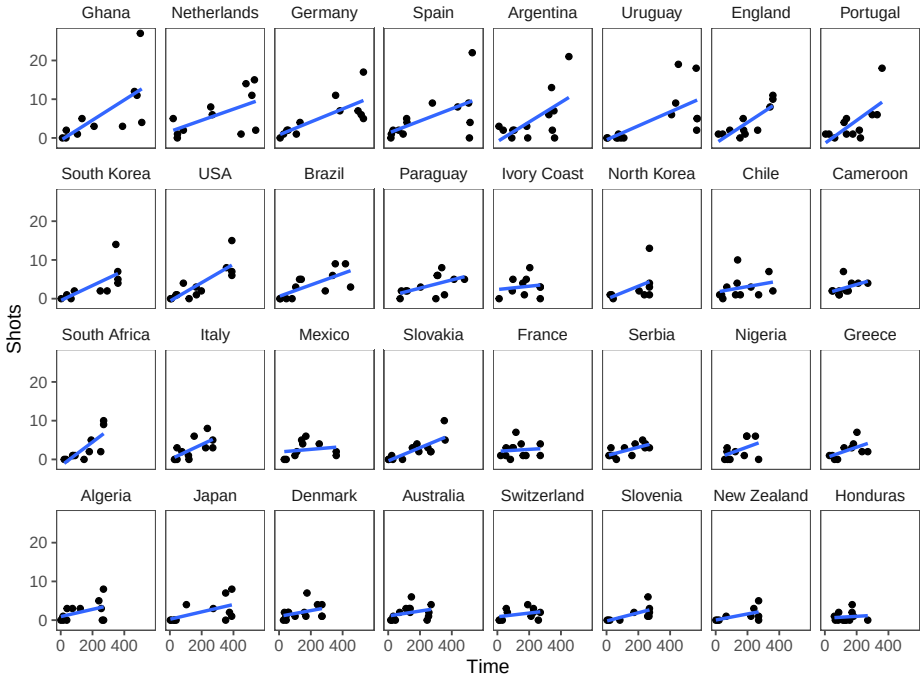
You can re-order factor variables in a graph by resetting the factor using the `factor` function and changing the order that levels are included in the `levels` parameter.

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.



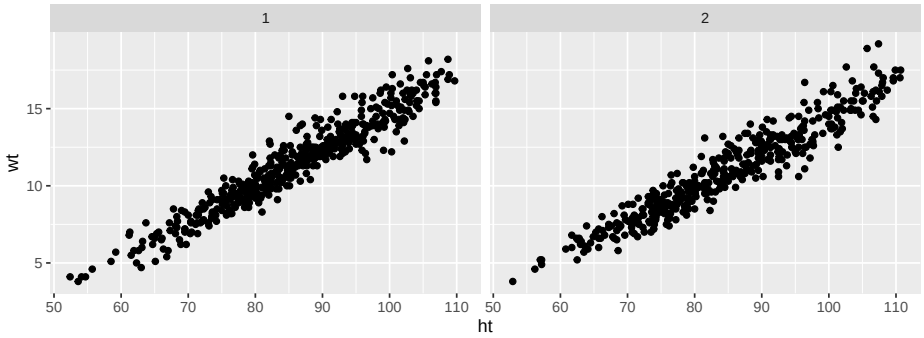
You can use the `facet` functions to create small multiples. This separates the graph into several small graphs, one for each level of a factor.

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:

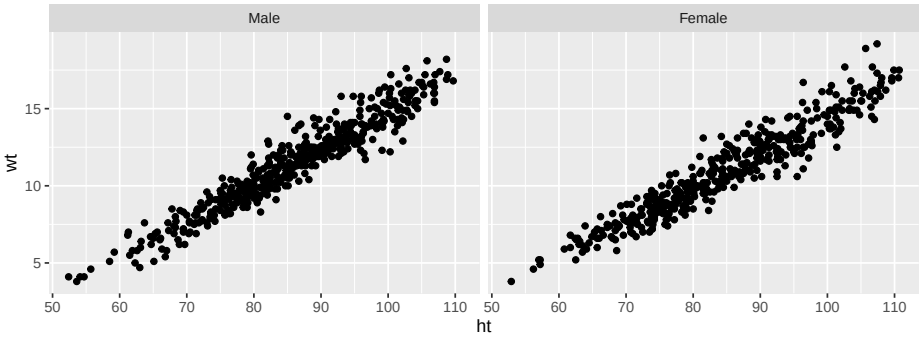
```
## Generic code
facet_wrap(~ [factor for faceting], ncol = [number of columns])
```

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)
```

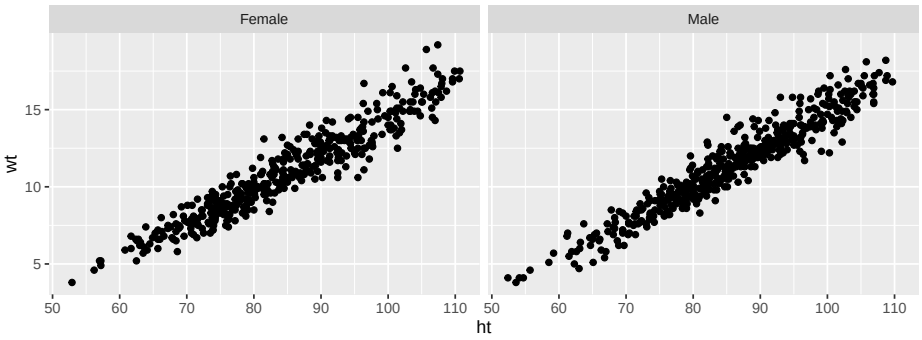


To re-order the factor, and show the plot for "Female" first, you can use `factor` to change the order of the levels:

```
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

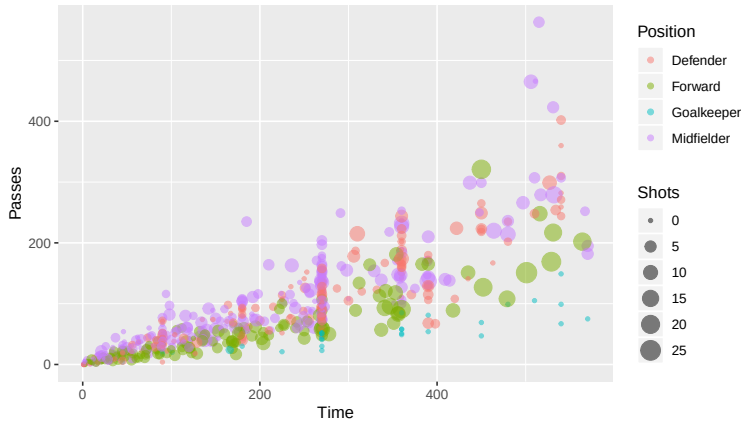
There are a number of different functions for adjusting scales. These follow the following convention:

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

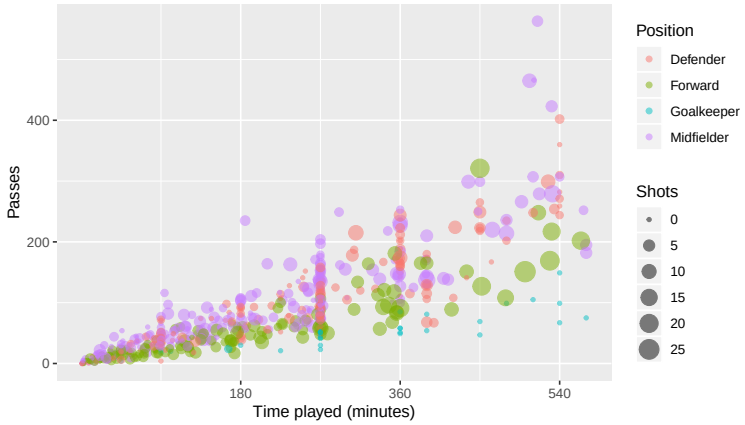
For example, to adjust the x-axis scale for a continuous variable, you'd use `scale_x_continuous`. You can use a `scale` function for an axis to change things like the axis label (which you could also change with `xlab` or `ylab`) as well as position and labeling of breaks.

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))
```

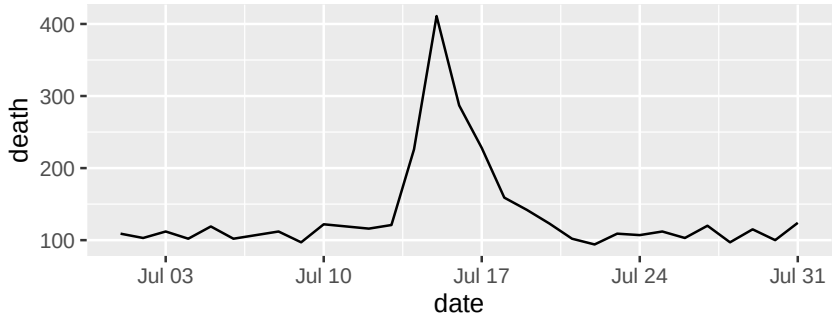


Parameters you might find useful in `scale` functions include:

Parameter	Description
name	Label or legend name
breaks	Vector of break points
minor_breaks	Vector of minor break points
labels	Labels to use for each break
limits	Limits to the range of the axis

For dates, you can use `scale` functions like `scale_x_date` and `scale_x_datetime`. For example, here's a plot of deaths in Chicago in July 1995 using default values for the x-axis:

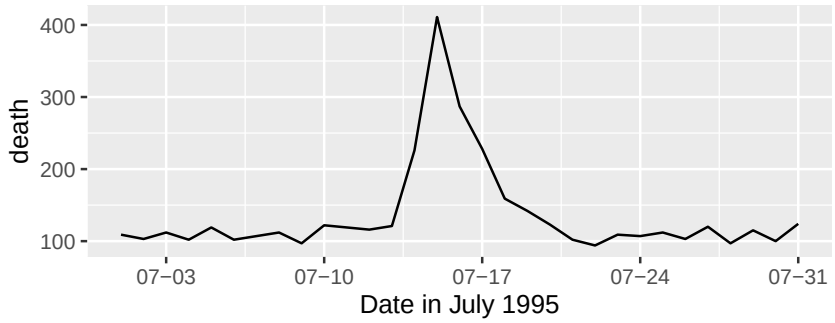
```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line()
```



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

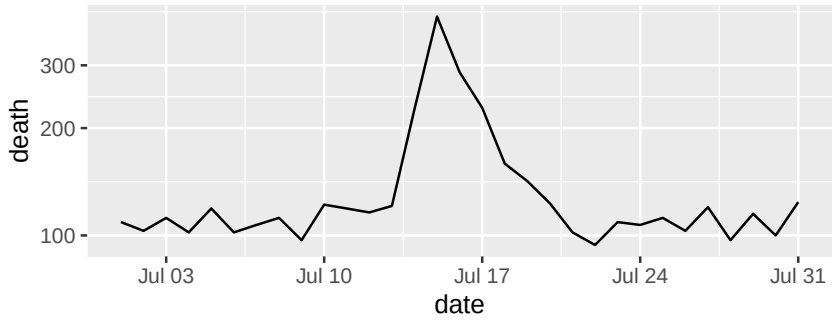
```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line() +
```

```
  scale_x_date(name = "Date in July 1995",
               date_labels = "%m-%d")
```



You can also use the `scale` functions to transform an axis. For example, to show the Chicago plot with "deaths" on a log scale, you can run:

```
ggplot(chic_july, aes(x = date, y = death)) +
  geom_line() +
  scale_y_log10()
```



For colors and fills, the conventions for the names of the `scale` functions can vary. For example, to adjust the color scale when you're mapping a discrete variable (i.e., categorical, like gender or animal breed) to color, you'd use `scale_color_hue`. To adjust the color scale for a continuous variable, like age, you'll use `scale_color_gradient`.

For any color scales, consider starting with `brewer` first (e.g., `scale_color_brewer`, `scale_color_distiller`). Scale functions from `brewer` allow you to set colors using different palettes. You can explore these palettes at http://colorbrewer2.org/.

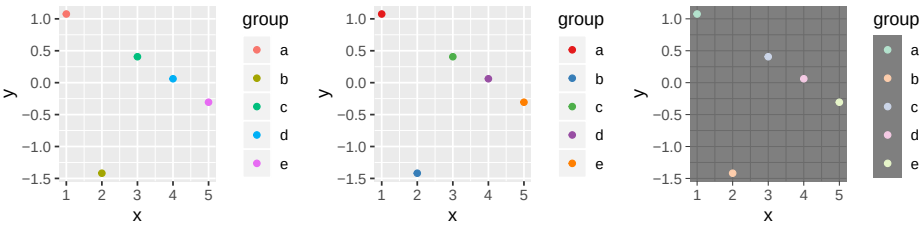
The Brewer palettes fall into three categories: sequential, divergent, and qualitative. You should use sequential or divergent for continuous data and qualitative for categorical data. Use `display.brewer.pal` to show the palette for a given number of colors.

```
library(RColorBrewer)
display.brewer.pal(name = "Set1", n = 8)
display.brewer.pal(name = "PRGn", n = 8)
display.brewer.pal(name = "PuBuGn", n = 8)
```

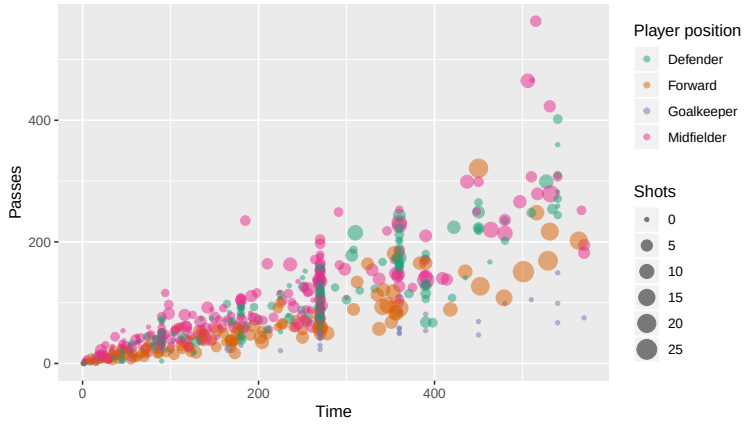
Set1 (qualitative) PRGn (divergent) PuBuGn (sequential)

Use the `palette` argument within a `scales` function to customize the palette:

```
a <- ggplot(data.frame(x = 1:5, y = rnorm(5),
                       group = letters[1:5]),
            aes(x = x, y = y, color = group)) +
  geom_point()
b <- a + scale_color_brewer(palette = "Set1")
c <- a + scale_color_brewer(palette = "Pastel2") +
  theme_dark()
grid.arrange(a, b, c, ncol = 3)
```

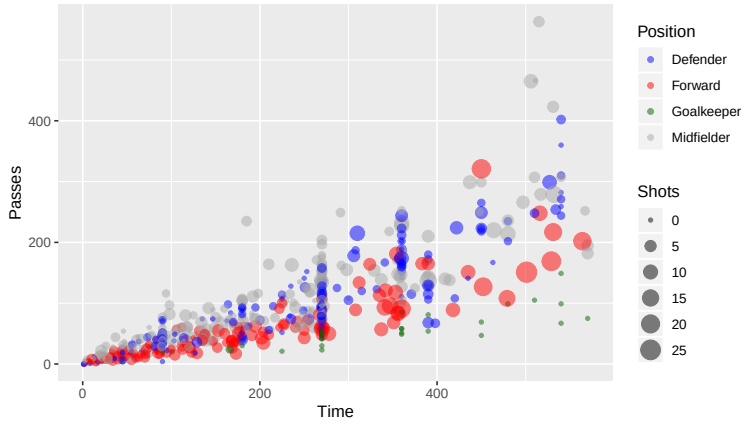



```
ggplot(worldcup, aes(x = Time, y = Passes,
                     color = Position, size = Shots)) +
  geom_point(alpha = 0.5) +
  scale_color_brewer(palette = "Dark2",
                     name = "Player position")
```



You can also set colors manually:

```
ggplot(worldcup, aes(x = Time, y = Passes,
                     color = Position, size = Shots)) +
  geom_point(alpha = 0.5) +
  scale_color_manual(values = c("blue", "red",
                                "darkgreen", "darkgray"))
```



4.9 To find out more

Some excellent further references for plotting are:

- R Graphics Cookbook (book and website)
- Google images

For more technical details about plotting in R:

- ggplot2: Elegant Graphics for Data Analysis, Hadley Wickham

- R Graphics, Paul Murrell

4.10 In-course exercise

4.10.1 Designing a plot

For today's exercise, you'll be building a plot using the `worldcup` data from the `faraway` package. First, load in that data. The name of each player is in the rownames of this data. Use the `tibble::rownames_to_column()` function to move those rownames into a new column named `Player`. Also install and load the `ggplot2` and `ggthemes` packages.

Next, say you want to look at the relationship between the number of minutes that a player played in the 2010 World Cup (`Time`) and the number of shots the player took on goal (`Shots`). On a sheet of paper, and talking with your partner, decide how the two of you would design a plot to explore and present this relationship. How would you incorporate some of the principles of creating good graphs?

4.10.1.1 Example R code

For this section, the only code needed is code to load the required packages, load the data, and move the rownames to a column named `Player`.

```
library(faraway)
data(worldcup)
head(worldcup, 2)
```

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

This dataset has the players' names as rownames, rather than in a column. Once we start using `dplyr` functions, we'll lose these rownames. Therefore, start by converting the rownames to a column called `Player`:

```
library(dplyr)
worldcup <- worldcup %>%
  tibble::rownames_to_column(var = "Player")
head(worldcup, 2)
```

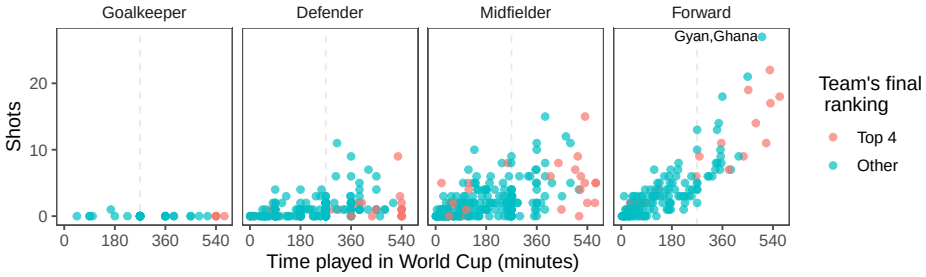
```
##   Player    Team   Position Time Shots Passes Tackles Saves
## 1 Abdoun Algeria Midfielder   16     0      6       0     0
## 2    Abe   Japan Midfielder  351     0    101      14     0
```

Install and load the `ggplot2` package:

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

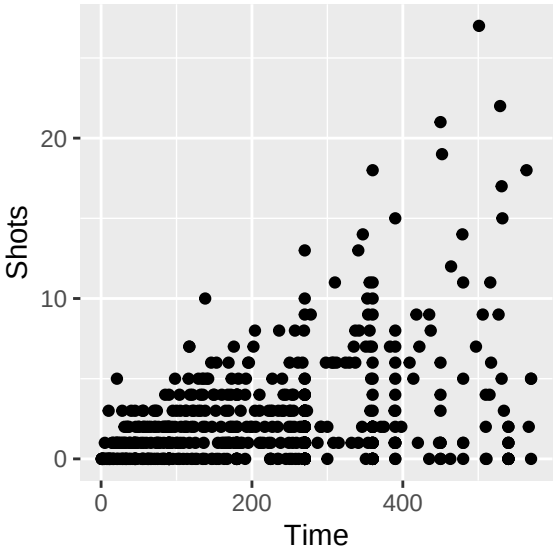
4.10.2 Implementing plot guidelines #1

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



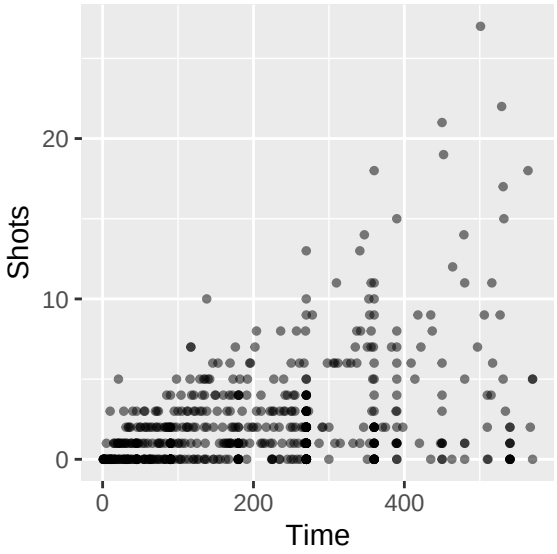
Do the following tasks:

- Create a simple scatterplot of Time versus Shots for the World Cup data. It should look like this:

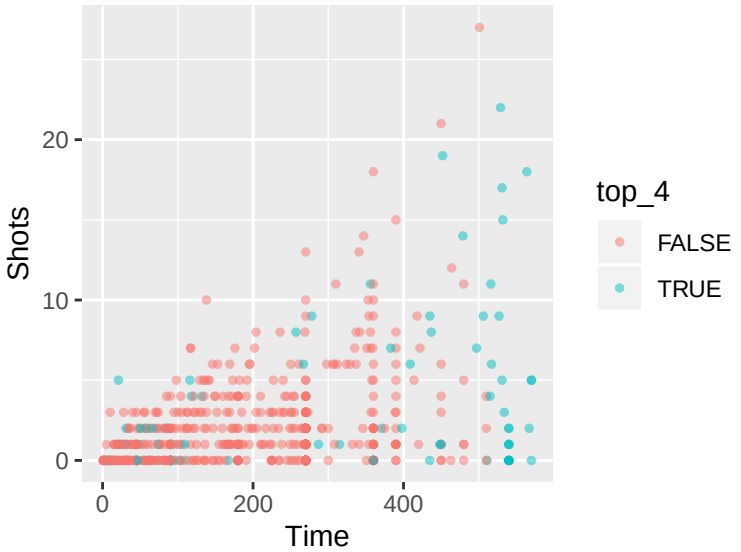


- Next, before any more coding, talk with your group members about how this graph is different from the simple one you created with `ggplot` in the last section. Also discuss what you can figure out from this new graph that was less clear from a simpler scatterplot of Time versus Shots for this data.

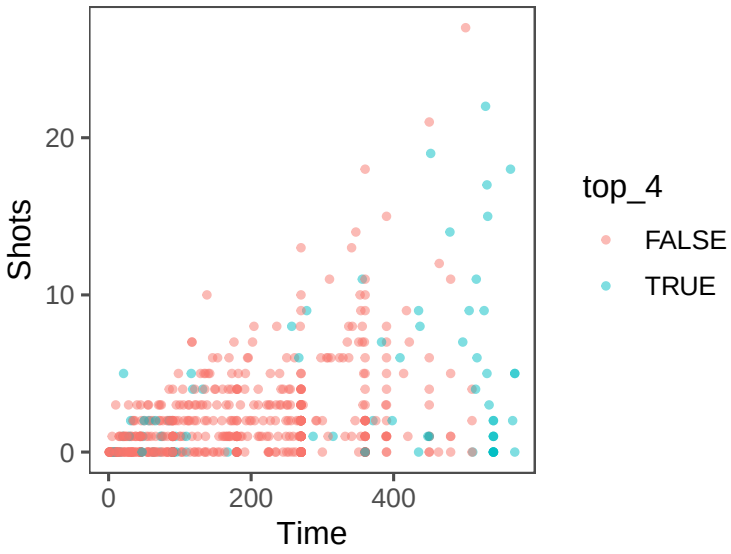
- Often, in graphs with a lot of points, it's hard to see some of the points, because they overlap other points. Three strategies to address this are: (a) make the points smaller; and (b) make the points somewhat transparent. Try doing these first two with the scatterplot you're creating. At this point, the plot should look something like this:



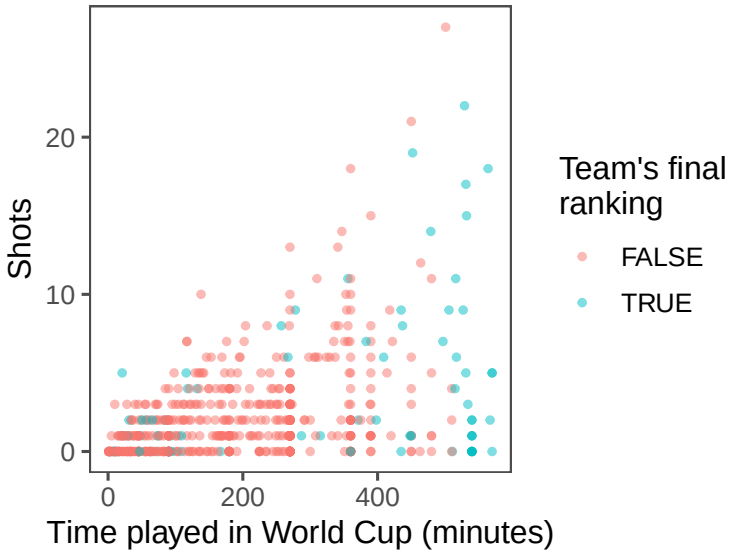
- Create a new column in the `worldcup` data called `top_four` that specifies whether or not the `Team` for that observation was one of the top four teams in the tournament (Netherlands, Uruguay, Spain, and Germany). Make the colors of the points correspond to whether the team was a top-four team. At this point, the plot should look something like this:



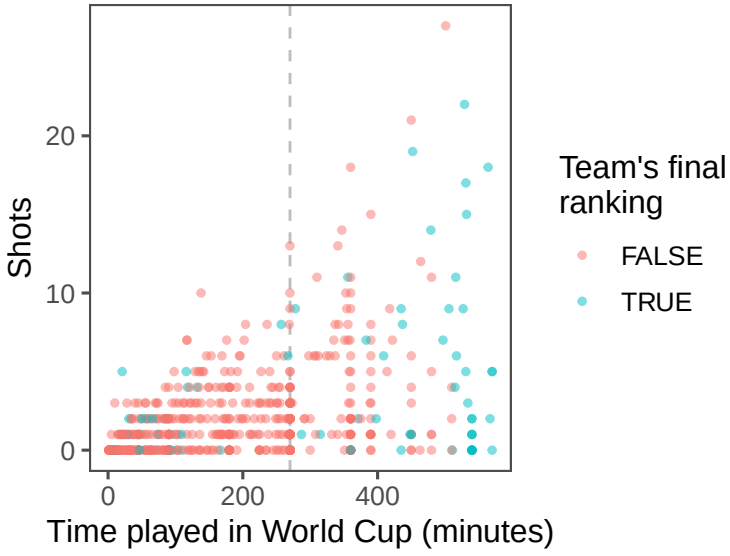
- Increase data density: Try changing the theme, to come up with a graph with a bit less non-data ink. From the `ggthemes` package, try some of the following themes: `theme_few()`, `theme_tufte()`, `theme_stata()`, `theme_fivethirtyeight()`, `theme_economist_white()`, and `theme_wsj()`. Pick a theme that helps increase the graph's data density. At this point, the plot should look something like this:



- Use meaningful labels: Use the `labs()` function to make a clearer title for the x-axis. (You may have already written this code in the last section of this exercise.) In addition to setting the x-axis title with the `labs` function, you can also set the title for the color scale (use `color =` within the `labs` function). You may want to make a line break in the color title– you can use the linebreak character (`\n`) inside the character string with the title to do that. At this point, the plot should look something like this:



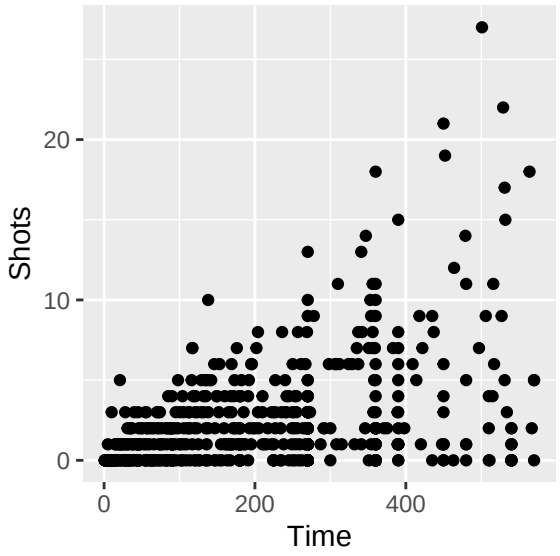
- Provide useful references: The standard time for a soccer game is 90 minutes. In the World Cup, all teams play at least three games, and then the top teams continue and play more games. Add a reference line at 270 minutes (i.e., the amount of standard time played for the three games that all teams play). At this point, the plot should look something like this:



4.10.2.1 Example R code

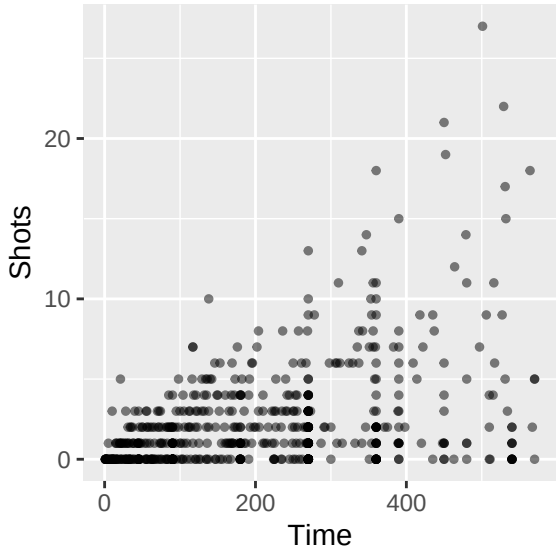
As a reminder, here's the code to do a simple scatterplot ot Shots by Time for the `worldcup` data:

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



Next, try to make it clearer to see the points by making them smaller and somewhat transparent. This can be done with the `size` and `alpha` aesthetics for `geom_points`. For the `size` aesthetic, a value smaller than about 2 = smaller than default, larger than about 2 = larger than default. For the `alpha` aesthetic, closer to 0 = more tranparent, closer to 1 = more opaque. As a reminder, in this case you are changing all of the points in the same way, so you will be setting those aesthetics to constant values. That means that you should specify the values **outside** of an `aes` call. This code could make these changes:

```
ggplot(data = worldcup) +
        geom_point(mapping = aes(x = Time, y = Shots),
                   size = 1, alpha = 0.5)
```



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

```
worldcup <- worldcup %>%
  mutate(top_4 = Team %in% c("Spain", "Germany",
                             "Uruguay", "Netherlands"))
head(worldcup)
```

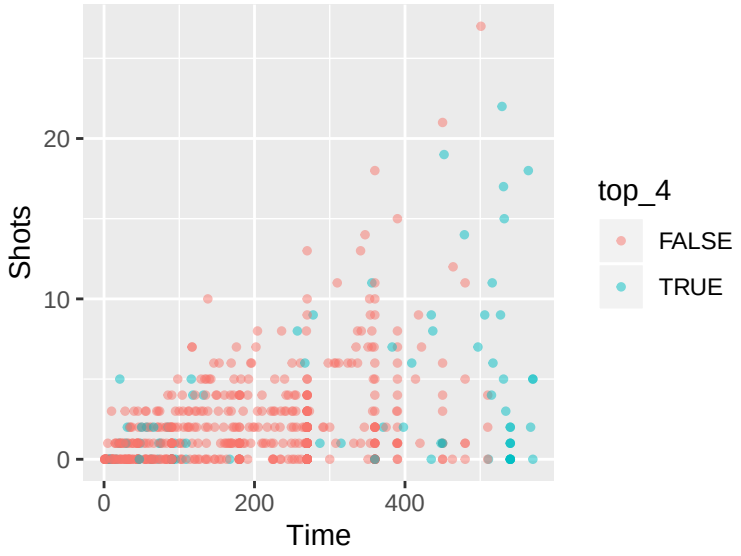
```
##       Team   Position Time Shots Passes Tackles Saves     Player top_4
## 1  Algeria Midfielder   16     0      6       0     0     Abdoun FALSE
## 2    Japan Midfielder  351     0    101      14     0        Abe FALSE
## 3   France   Defender  180     0     91       6     0     Abidal FALSE
## 4   France Midfielder  270     1    111       5     0 Abou Diaby FALSE
## 5 Cameroon    Forward   46     2     16       0     0  Aboubakar FALSE
## 6  Uruguay    Forward   72     0     15       0     0      Abreu  TRUE
```

```
summary(worldcup$top_4)
```

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

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

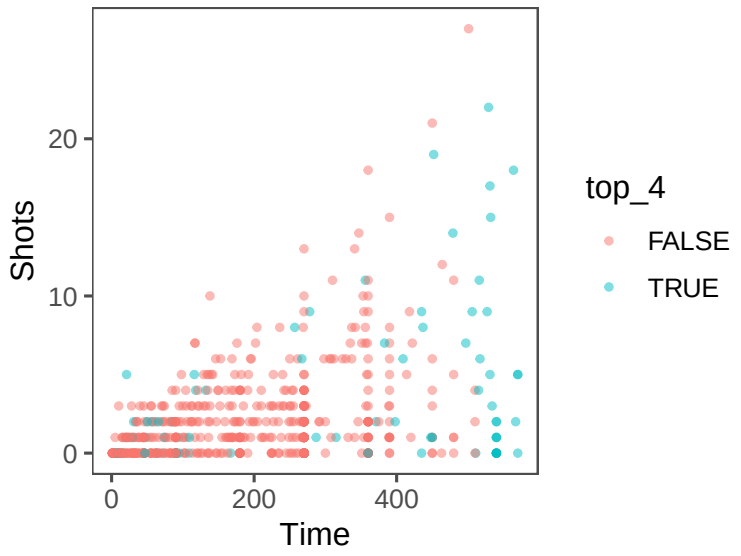
```
ggplot(data = worldcup) +
        geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
                   size = 1, alpha = 0.5)
```



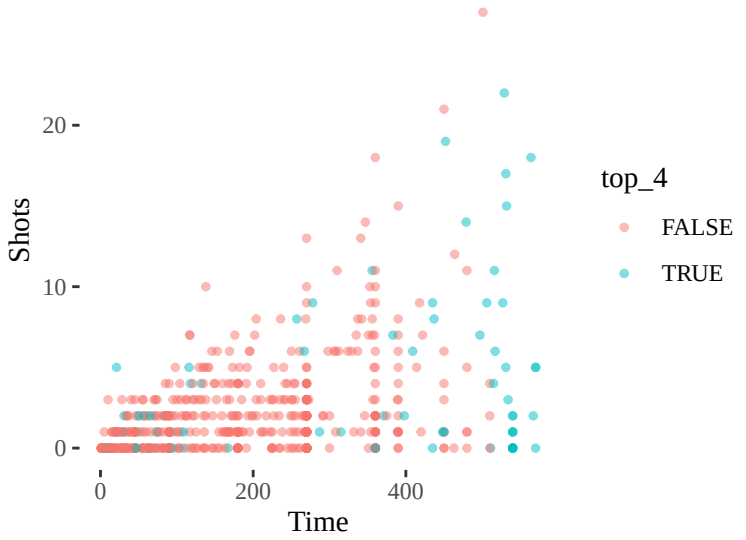
To increase the data density, try out different themes for the plot. First, I'll save everything we've done so far as the object `shot_plot`, then I'll try adding different themes:

```
shot_plot <- ggplot(data = worldcup) +
        geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
                   size = 1, alpha = 0.5)

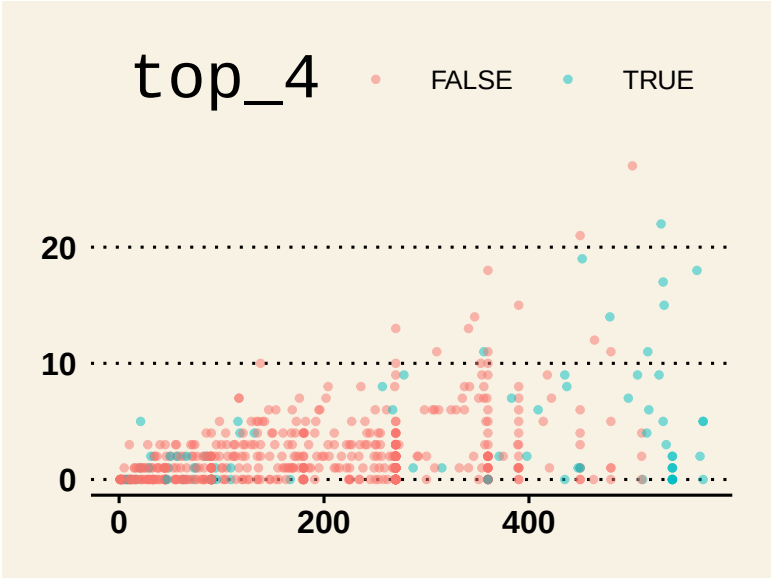
shot_plot + theme_few()
```



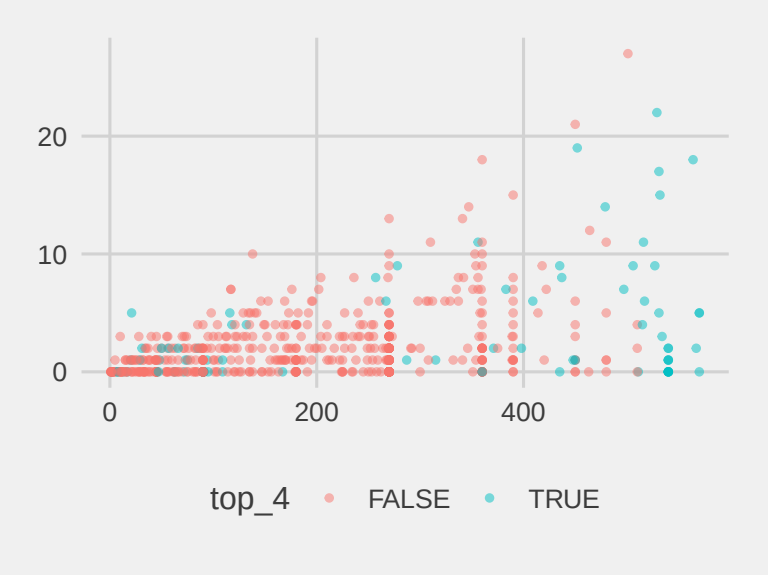
```
shot_plot + theme_tufte()
```



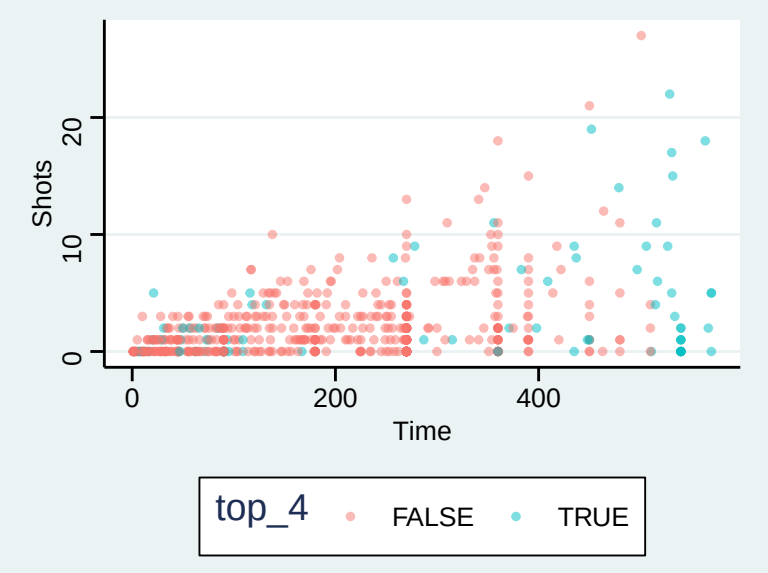

```
shot_plot + theme_wsj()
```



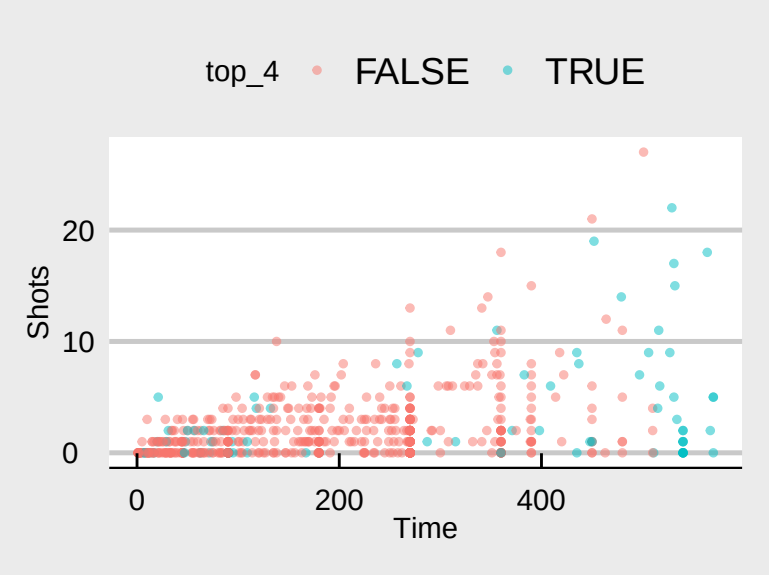

```
shot_plot + theme_fivethirtyeight()
```




```
shot_plot + theme_stata()
```

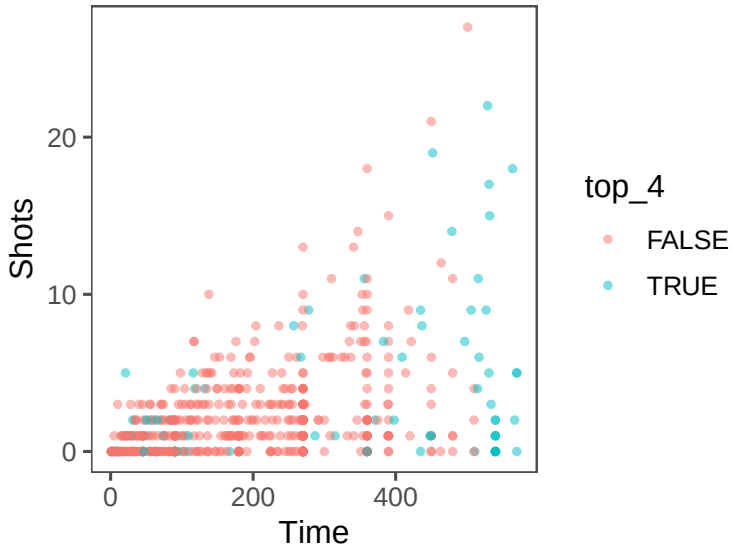


```
shot_plot + theme_economist_white()
```



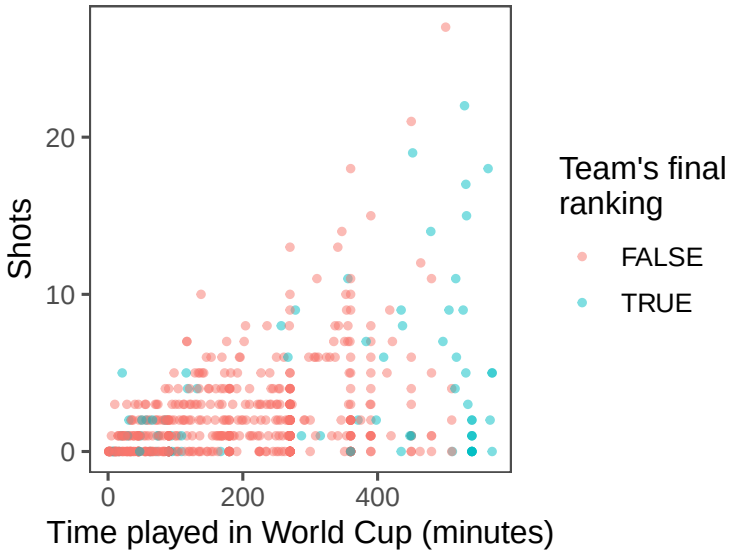
The data density is increased with the `theme_few()` theme, so I'll use that:

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few()
```



To change the titles for some of the scales (the x-axis and color scale), you can use the `labs()` function. Note that you can use `\n` to add a line break inside one of these titles (I've done that for the title for the color scale):

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few() +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking")
```

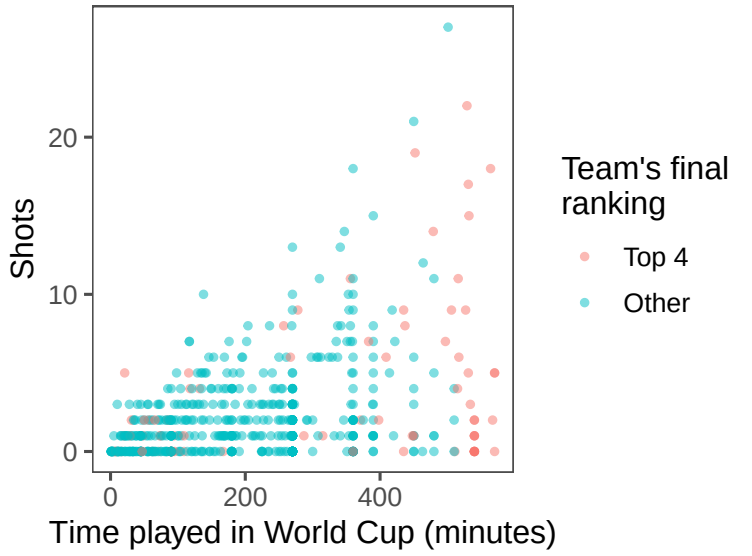


As an extra note, if you want to create nicer labels for the legend for color, convert the `top_four` column into the factor class, with the labels you want to use in the figure legend:

```
worldcup <- worldcup %>%
  mutate(top_4 = factor(top_4, levels = c(TRUE, FALSE),
                        labels = c("Top 4", "Other")))
summary(worldcup$top_4)
```

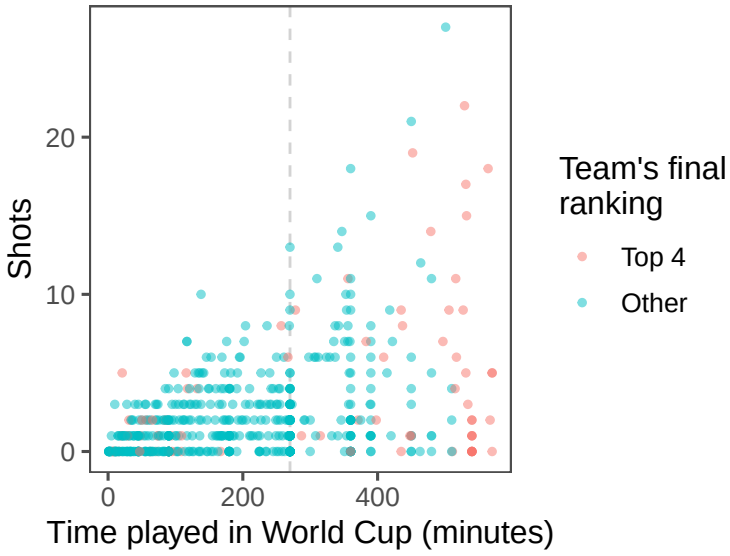
```
## Top 4 Other
##    78   517
```

```
ggplot(data = worldcup) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few() +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking")
```



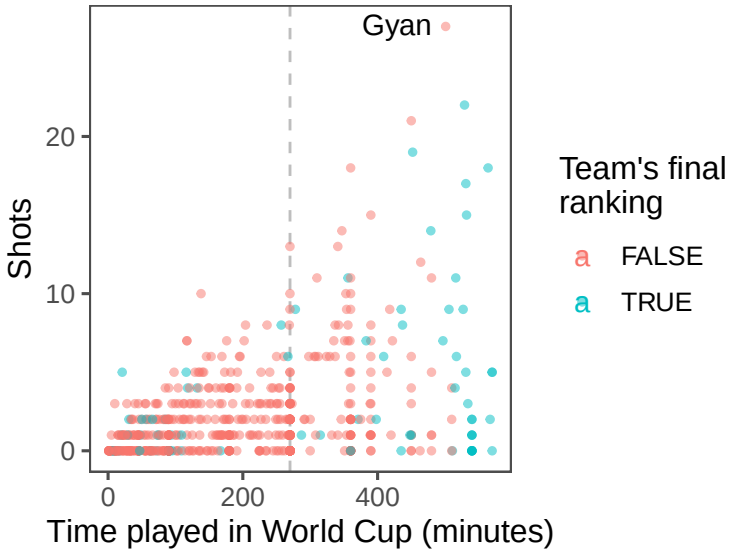
To add a reference line at 270 minutes of time, use the `geom_vline()` function. You'll want to make it a light color (like light gray) and dashed or dotted (`linetype` of 2 or 3), so it won't be too prominent on the graph:

```
ggplot(data = worldcup) +
  geom_vline(xintercept = 270, color = "lightgray", linetype = 2) +
  geom_point(mapping = aes(x = Time, y = Shots, color = top_4),
             size = 1, alpha = 0.5) +
  theme_few() +
  labs(x = "Time played in World Cup (minutes)",
       color = "Team's final\nranking")
```



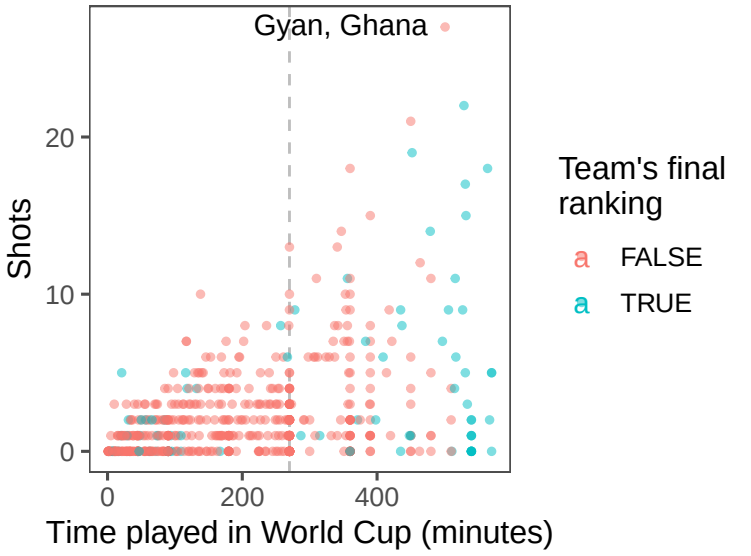
4.10.3 Implementing plot guidelines #2

- Highlighting interesting data: Who had the most shots in the 2010 World Cup? Was he on a top-four team? Use `geom_text()` to label his point on the graph with his name (try out some different values of `hjust` and `vjust` in this function call to get the label in a place you like). At this point, the plot should look something like this:

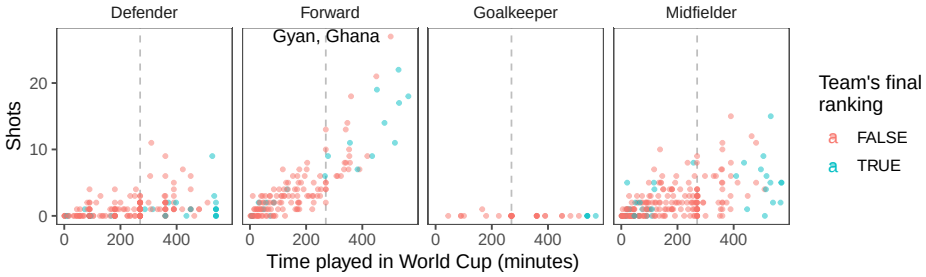


- For labeling the player with the top number of shots, instead of only using the player's name, use the following format: "[Player's name], [Player's team]". (Hint:

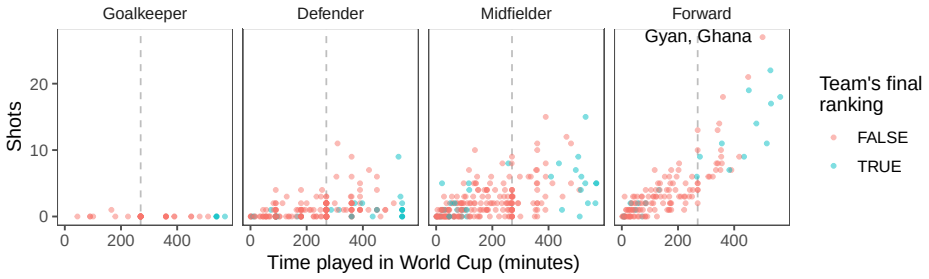
You may want to use `mutate` to add a new column, where you used `paste0` to paste together the player's name, `", "`, and the team name.) At this point, the plot should look something like this:



- Create small multiples. The relationship between time played and shots taken is probably different by the players' positions. Use faceting to create different graphs for each position. At this point, the plot should look something like this:



- Make order meaningful: What order are the faceted graphs currently in? Offensive players have more chances to take shots than defensive players, so that might be a useful ordering for the facets. Re-order the `Position` factor column to go from nearest your own goal to nearest the opponents goal, and then re-plot the graph from the previous step.

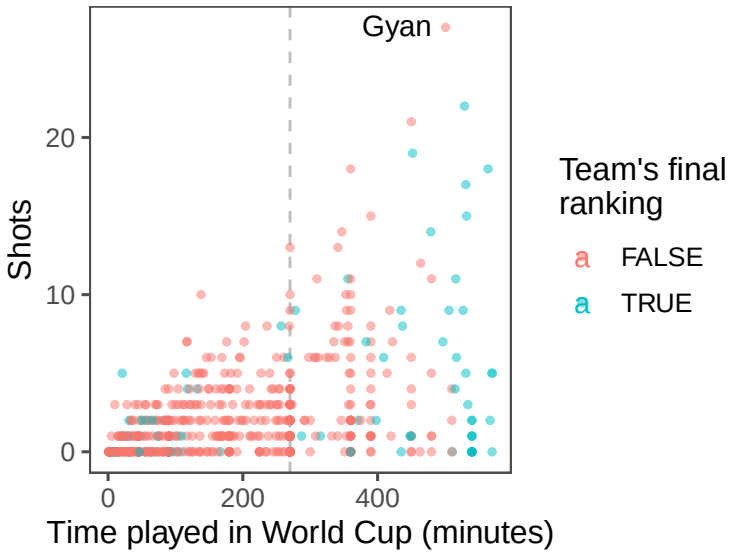


4.10.3.1 Example R code

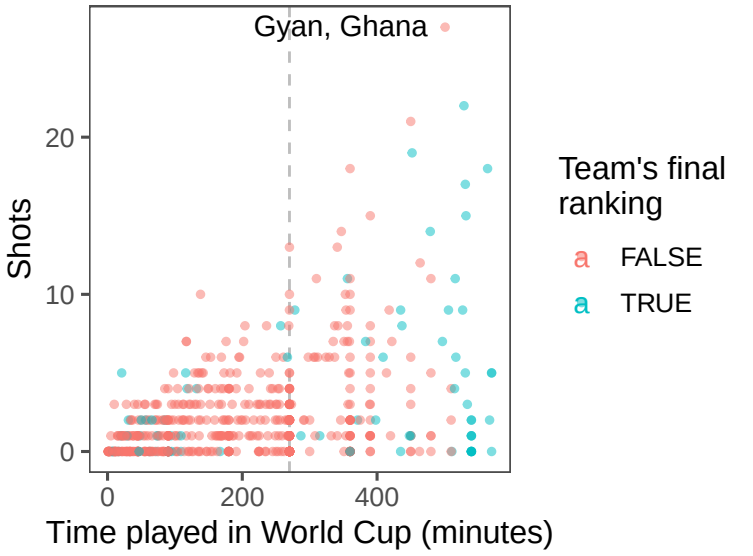
To add a text label with just the player with the most shots, you'll want to create a new dataframe with just the top player. You can use the `top_n` function to do that (the `wt` option is specifying that we want the top player in terms of values in the `Shots` column):

```
top_player <- worldcup %>%
  top_n(n = 1, wt = Shots)
```

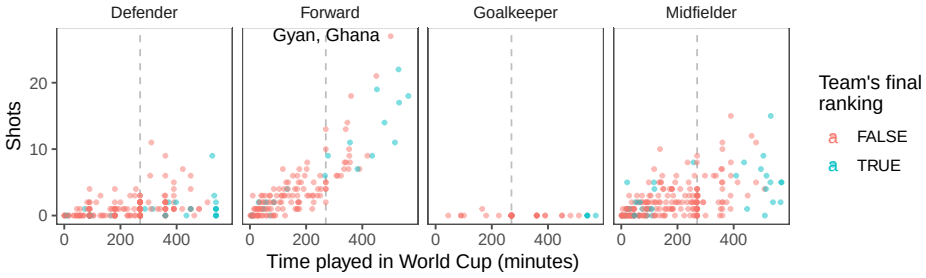
Now you can use `geom_text()` to label this player's point on the graph with his name. You may need to mess around with some of the options in `geom_text()`, like `size`, `hjust`, and `vjust` (`hjust` and `vjust` say where, in relation to the point location, to put the label), to get something you're happy with.



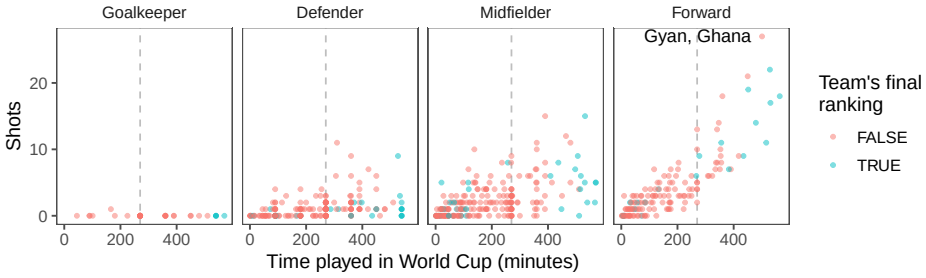
If you want to put both the player's name and his team, you can add a `mutate()` function when you create the new dataframe with just the top player, and then use this for the label:



To create small multiples, use the `facet_wrap()` command (you'll probably want to use `ncol` to specify to use four columns):



To re-order the `Position` column of the dataframe, add a `mutate` statement before you pipe into the plotting code. Use the `levels` option of the `factor()` function– whatever order you put the factors in for this argument will be the new order in which R saves the levels of this factor.



Note from this code example that you can use the `levels` function to find out the levels and their order for a factor-class vector.

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

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

4.10.4 Data visualization cheatsheet

RStudio comes with some excellent cheatsheets, which provide quick references to functions and code you might find useful for different tasks. For this part of the group exercise, you'll explore their cheatsheet for data visualization, both to learn some new `ggplot2` code and to become familiar with how to use this cheatsheet as you do your own analysis.

- Open the data visualization cheatsheet. You can do this from RStudio by going to "Help" -> "Cheatsheets" -> "Data Visualization with ggplot2".
- Notice that different sections give examples with some datasets that come with either base R or ggplot2. For example, under the "Graphical Primitives" section, there is code defining the object `a` as a ggplot object using the "seals" dataset: `a <- ggplot(seals, aes(x = long, y = lat))`.
- Go through the cheatsheet and list all of the example datasets that are used in this cheatsheet. Open their helpfiles to learn more about the data.
- Create the example datasets `a` through `l` and `s` through `t` using the code given on the cheatsheet.
- Pick at least one example to try out from each of the following sections: "Graphical Primitives", "One Variable", at least three subsections of "Two Variables", "Three Variables", "Scales", "Faceting", and "Position Adjustments". As you try these, try to figure out any aesthetics that you aren't familiar with (e.g., `ymin`, `ymax`). Also, use helpfiles for the geoms to look up parameters you aren't familiar with (e.g., `stat` for `geom_area`). If you can't figure out how to interpret a plot, check the helpfile for the associated geom. **Note**: For the `n` geom used in "scales", it should be defined as `n <- d + geom_bar(aes(fill = fl))`.

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(fl))
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(fl, 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(fl)`). 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 = fl))
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 ~ fl)
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 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.

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:

view-source:www.colostate.edu

```
168
169                         <li>
170                             <a href="http://source.colostate.edu/beloved-mascot-cam-the-ram-
dies-at-age-6/"><img src="http://wwwmanager.colostate.edu/uploaded/1677.4.1.jpg" alt="Beloved
Mascot CAM the Ram Dies at Age 6" /></a>
171                             <div class="sDescription"><div class="dWrap">
172                                 <h3>Beloved Mascot CAM the Ram Dies at Age 6</h3>
173                                 <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>
174                                 <p><a href="http://source.colostate.edu/beloved-mascot-cam-
the-ram-dies-at-age-6/">Read his Obituary</a></p>
175                             </div></div>
176                         </li>
177
```

Figure 5.1: Example of the source of an HTML file.

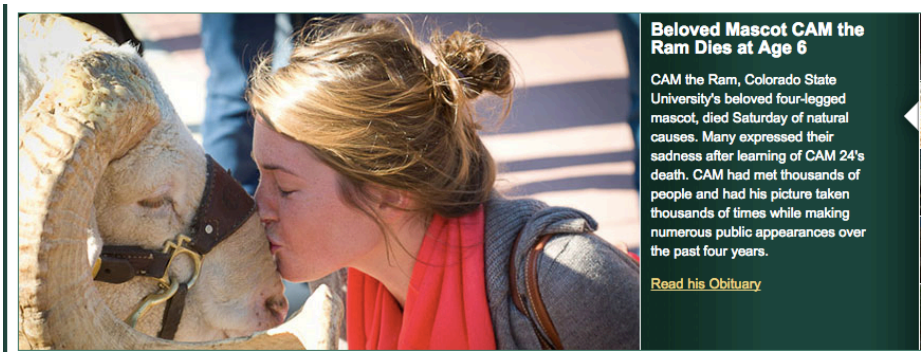


Figure 5.2: Example of a rendered HTML file.

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
'echo'	Print out the R code?
'eval'	Run the R code?
'messages'	Print out messages?
'warnings'	Print out warnings?
'include'	If FALSE, run code, but don't print code or results

Other chunk options take values other than TRUE / FALSE. Some you might want to include are:

Option	Action
`results`	How to print results (e.g., `hide` runs the code, but doesn't print the results)
`fig.width`	Width to print your figure, in inches (e.g., `fig.width = 4`)
`fig.height`	Height to print your figure

Add these options in the opening brackets and separate multiple ones with commas:

````
```{r  messages = FALSE, echo = FALSE}
````
````

````
nepali[1, 1:3]
```
````

I will cover other chunk options later, once you've gotten the chance to try 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 (`todays_groups` for an object with group assignments)
  - Verbs for functions (`make_groups` for the function to assign groups)

5.4.1 Line length

Google: **Keep lines to 80 characters or less**

To set your script pane to be limited to 80 characters, go to "RStudio" -> "Preferences" -> "Code" -> "Display", and set "Margin Column" to 80.

```
# Do
my_df <- data.frame(n = 1:3,
                    letter = c("a", "b", "c"),
                    cap_letter = c("A", "B", "C"))

# Don't
my_df <- data.frame(n = 1:3, letter = c("a", "b", "c"), cap_letter = c("A", "B", "C"))
```

This guideline helps ensure that your code is formatted in a way that you can see all of the code without scrolling horizontally (left and right).

5.4.2 Spacing

- Binary operators (e.g., <−, +, −) should have a space on either side
- A comma should have a space after it, but not before.
- Colons should not have a space on either side.
- Put spaces before and after = when assigning parameter arguments

```
# Do
shots_per_min <- worldcup$Shots / worldcup$Time
#Don't
shots_per_min<-worldcup$Shots/worldcup$Time

#Do
ave_time <- mean(worldcup[1:10 , "Time"])
#Don't
ave_time<-mean(worldcup[1 : 10 ,"Time"])
```

5.4.3 Semicolons

Although you can use a semicolon to put two lines of code on the same line, you should avoid it.

```
# Do
a <- 1:10
b <- 3

# Don't
a <- 1:10; b <- 3
```

5.4.4 Commenting

- For a comment on its own line, use #. Follow with a space, then the comment.
- You can put a short comment at the end of a line of R code. In this case, put two spaces after the end of the code, one #, and one more space before the comment.
- If it helps make it easier to read your code, separate sections using a comment character followed by many hyphens (e.g., #----------). Anything after the comment character is "muted".

```
# Read in health data ---------------------------

# Clean exposure data ---------------------------
```

5.4.5 Indentation

Google:

- Within function calls, line up new lines with first letter after opening parenthesis for parameters to function calls:

Example:

```
# Relabel sex variable
nepali$sex <- factor(nepali$sex,
                     levels = c(1, 2),
                     labels = c("Male", "Female"))
```

5.4.6 Code grouping

- Group related pieces of code together.
- Separate blocks of code by empty spaces.

```r
# 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 helpful.

Note: Equations denoted this way will always compile for pdf documents, but won't always come through on Markdown files (for example, GitHub won't compile math equations).

For example, writing this in your R Markdown file:

```
$$
E(Y_{t}) \sim \beta_{0} + \beta_{1}X_{1}
$$
```

will result in this rendered equation:

$$E(Y_t) \sim \beta_0 + \beta_1 X_1$$

To put math within a sentence (**inline equation**), just use one $ on either side of the math. For example, writing this in a R Markdown file:

```
"We are trying to model $E(Y_{t})$."
```

The rendered document will show up as:

"We are trying to model $E(Y_t)$."

5.5.2 Figures from file

You can include not only figures that you create with R, but also figures that you have saved on your computer.

The best way to do that is with the `include_graphics` function in `knitr`:

```
library(knitr)
include_graphics("figures/CSU_ram.png")
```

This example would include a figure with the filename "MyFigure.png" that is saved in the "figures" sub-directory of the parent directory of the directory where your .Rmd is saved. Don't forget that you will need to give an absolute pathway or the relative pathway **from the directory where the .Rmd file is saved**.

5.5.3 Saving graphics files

You can save figures that you create in R. Typically, you won't need to save figures for an R Markdown file, since you can include figure code directly. However, you will sometimes want to save a figure from a script. You have two options:

- Use the "Export" choice in RStudio
- Write code to export the figure in your R script

To make your research more reproducible, use the second choice.

To use code export a figure you created in R, take three steps:

1. Open a graphics device (e.g., `pdf("MyFile.pdf")`).
2. Write the code to print your plot.
3. Close the graphics device using `dev.off()`.

For example, the following code would save a scatterplot of time versus passes as a pdf named "MyFigure" in the "figures" subdirectory of the current working directory:

```
pdf("figures/MyFigure.pdf", width = 8, height = 6)
ggplot(worldcup, aes(x = Time, y = Passes)) +
        geom_point(aes(color = Position)) +
        theme_bw()
dev.off()
```

If you create multiple plots before you close the device, they'll all save to different pages of the same pdf file.

You can open a number of different graphics devices. Here are some of the functions you can use to open graphics devices:

- `pdf`
- `png`
- `bmp`
- `jpeg`
- `tiff`
- `svg`

You will use a device-specific function to open a graphics device (e.g., `pdf`). However, you will always close these devices with `dev.off`.

Most of the functions to open graphics devices include parameters like `height` and `width`. These can be used to specify the size of the output figure. The units for these depend on the device (e.g., inches for `pdf`, pixels by default for `png`). Use the helpfile for the function to determine these details.

5.5.4 Tables in R Markdown

If you want to create a nice, formatted table from an R 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
`colnames`	Column names (default: column name in the dataframe)
`align`	A vector giving the alignment for each column ('l', 'c', 'r')

Table 5.3: My new table

First 3 letters	First 3 numbers
a	-0.78
b	0.10
c	2.13

arg	expl
caption	Table caption
digits	Number of digits to round to. If you want to round columns different amounts, use a vector with one element for each column.

```
my.df <- data.frame(letters = c("a", "b", "c"),
                    numbers = rnorm(3))
kable(my.df, digits = 2, align = c("r", "c"),
      caption = "My new table",
      col.names = c("First 3 letters",
                    "First 3 numbers"))
```

From Yihui:

> "**Want more features?** No, that is all I have. You should turn to other packages for help. I'm not going to reinvent their wheels."

If you want to do fancier tables, you may want to explore the `xtable` and `pander` packages. As a note, these might both be more effective when compiling to pdf, rather than html.

5.6 In-course exercise

For all of today's tasks, you'll use the code from last week's in-course exercise to do the exercises. This week we are not focusing on writing new code, but rather on how to take R code and put it in an R Markdown file, so we can create reports from files that include the original code.

5.6.1 Creating a Markdown document

First, you'll create a Markdown document, without any R code in it yet.

In RStudio, go to "File" -> "New File" -> "R Markdown". From the window that brings up, choose "Document" on the left-hand column and "HTML" as the output format. A new file will open in the script pane of your RStudio session. Save this file (you may pick the name and directory). The file extension should be ".Rmd".

First, before you try to write your own Markdown, try rendering the example that the script includes by default. (This code is always included, as a template, when you first open a new RMarkdown file using the RStudio "New file" interface we used in this example.) Try rendering this default R Markdown example by clicking the "Knit" button at the top of the script file.

For some of you, you may not yet have everything you need on your computer to be able to get this to work. If so, let me know. RStudio usually includes all the necessary tools when you install it, but there may be some exceptions.

If you could get the document to knit, do the following tasks:

- Look through the HTML document that was created. Compare it to the R Markdown script that created it, and see if you can understand, at least broadly, what's going on.
- Look in the directory where you saved the R Markdown file. You should now also see a new, .html file in that folder. Try opening it with a web browser like Safari.
- Go back to the R Markdown file. Delete everything after the initial header information (everything after the 6th line). In the header information, make sure the title, author, and date are things you're happy with. If not, change them.
- Using Markdown syntax, write up a description of the data (`worldcup`) we used last week to create the fancier figure. Try to include the following elements:
  - Bold and italic text
  - Hyperlinks
  - A list, either ordered or bulleted
  - Headers

5.6.2 Adding in R code

Now incorporate the R code from previous weeks' exercises into your document. Once you get the document to render with some basic pieces of code in it, try the following:

- Try some different chunk options. For example, try setting `echo = FALSE` in some of your code chunks. Similarly, try using the options `results = "hide"` and `include = FALSE`.
- You should have at least one code chunk that generates figures. Try experimenting with the `fig.width` and `fig.height` options for the chunk to change the size of the figure.
- Try using the global commands. See if you can switch the `echo` default value for this document from TRUE (the usual default) to FALSE.

5.6.3 Working with R Markdown documents

Finally, try the following tasks to get some experience working with R Markdown files in RStudio:

- Go to one of your code chunks. Locate the small gray arrow just to the left of the line where you initiate the code chunk. Click on it and see what happens. Then

click on it again.

- Put your cursor inside one of your code chunks. Try using the "Run" button (or Ctrl-Return) to run code in that chunk at your R console. Did it work?
- Pick a code chunk in your document. Put your cursor somewhere in the code in that chunk. Click on the "Run" button and choose "Run All Chunks Above". What did that do? If it did not work, what do you think might be going on? (Hint: Check `getwd()` and think about which directory you've used to save your R Markdown file.)
- Pick another chunk of code. Put the cursor somewhere in the code for that chunk. Click on the "Run" button and choose "Run Current Chunk". Then try "Run Next Chunk". Try to figure out all the options the "Run" button gives you and when each might be useful.
- Click on the small gray arrow to the right of the "Knit HTML" button. If the option is offered, select "Knit Word" and try it. What does this do?

5.6.4 R style guidelines

Go through all the R code in your R Markdown file. Are there are places where your code is not following style conventions for R? Clean up your code to correct any of these issues.

Appendix A

Appendix A: Vocabulary

You will be responsible for knowing the following functions and vocabulary for the weekly quizzes.

A.1 Quiz 1—R Preliminaries (Updated for 2019)

- Grading policies for the course
- Course requirements / policies for in-class quizzes
- Free and open source software
- "Free as in beer" versus "free as in speech"
- Advantages and disadvantages of interpreted languages compared to "Point-and-click" software
- Advantages and of interpreted languages compared to compiled languages and assembly languages
- Difference between R and RStudio
- R session
- R console
- Function arguments (including required versus optional arguments)
- Accessing a function's helpfile using ?
- Mathematical operators: +, −, *, ^
- R objects and object names
- "gets arrow": <−
- Rules and style guidelines for naming objects
- `ls()`
- R scripts
- # comment character
- R packages
- CRAN
- Installing packages
- `install.packages()`

- Loading a package
- `library()`
- Types of package documentation: vignettes and helpfiles
- `vignette()`, option `package =`
- Vectors
- `c()`
- Two of the basic classes of vectors: character and numeric
- `class()`
- Square bracket indexing for vectors: `[...]`
- Dataframes
- `tibble()` function from the `dplyr` package
- `select` and `slice` functions to extract values from dataframes
- `read_csv`, option `skip =`
- `str()`
- `summary()`
- `dim()`
- `ncol()`
- `nrow()`
- Nate Silver
- FiveThirtyEight
- `NA` for missing values
- `$` to get a column from a dataframe
- `paste()`, option `sep =`
- `paste0()`
- = vs. <- for assignment expressions
- `package::function()` notation

A.2 Quiz 2—Entering / cleaning data #1 (Updated for 2018)

- What kinds of data can be read into R?
- delimited files (csv, tsv)
- fixed width files
- delimiter
- `read_delim`, options `delim =`, `skip =`, `n_max =`, `col_names =`
- `read_fwf`
- `read_csv`, options `skip =`, `n_max =`, `col_names =`
- `readxl` package and its `read_excel()` function
- `haven` package and its `read_sas()` function
- `NA`
- Computer directory structure
- working directory
- `getwd()`
- `list.files()`

- relative pathnames
- absolute pathnames
- shorthand for pathnames: `.`, `..`, `../data`, etc.
- Reading in data from either a local or online flat file
- `paste()`, option `sep =`
- `paste0()`
- How to read flat files of data that are online directly into R
- `dplyr` package
- `rename()`
- Why you might want to rename column names (e.g., uppercase, long, unusual characters)
- `select()`
- `slice()`
- `mutate()`
- `filter()`
- `arrange()`, including with `desc()`
- Main types of vector classes in R: character, numeric, factor, date, logical
- `lubridate` functions, include `ymd`, `ymd_hm`, `mdy`, `wday`, and `mday`
- %>%, advantages of piping
- Common logical operators in R (==, `!=`, `%in%`, `is.na()`, `&`, `|`)

A.3 Quiz 3 (Updated for 2018)

- `data()` (with and without the name of a dataset as an option)
- `library()` (with and without an argument in the parentheses)
- logical vectors, including running `sum` on a logical vector
- What the bang operator (`!`) does to a logical operator
- The tidyverse
- `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) and dataframes
- 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.

- `summarize()`
- Special functions to use with `summarize()`: `n()`, `n_distinct()`, `first()`, `last()`
- Using `group_by()` before using `summarize()`
- The three basic elements of a `ggplot` plot: data, aesthetics, and geoms
- `aes` function and common aesthetics, including `color`, `shape`, `x`, `y`, `alpha`, `size`, and `fill`
- Mapping an aesthetic to a column in the data versus setting it to a constant value
- Some common geoms: `geom_histogram`, `geom_points`, `geom_lines`, `geom_boxplot()`
- The difference between "statistical" geoms (e.g., `geom_boxplot`, `geom_smooth`) and "non-statistical" (e.g., `geom_point`, `geom_line`)
- Common additions to `ggplot` objects: `ggtitle`, `labs`, `xlim`, `ylim`, `expand_limits`

A.4 Quiz 4 (Updated for 2018)

- 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

Appendix B

Appendix B: Homework

This section provides the homework assignments for the course.

B.1 Homework #1

Due date: Sept. 11 by 5:00 pm

For this assignment, you will submit the assignment to me **by email** by the due date. You should include three files in your submission:

1. A Word document with seven paragraphs. Each paragraph should be headed with the name of one swirl lesson and the body of the paragraph should describe that lesson and what you learned from it.

For this homework assignment, you'll be working through a few swirl lessons that are relevant to the material we've covered so far. Swirl is a platform that helps you learn R **in** R—you can complete the lessons right in your R console.

Depending on your familiarity with R, you can either work through seven lessons of your choice in the `R Programming: The basics of programming in R` and `Getting and Cleaning Data` courses (suggested lessons are listed further below) (**Option #1**), or you can work through seven lessons of your choice taken from any number of swirl's available courses (**Option #2**) .

For each lesson completed, please write a few sentences that cover: 1. A summary of the topic(s) covered in that lesson, and 2. The most interesting thing that you learned from that lesson. Turn in a hardcopy of this (with your first and last name at the top) during class on the due date.

To begin, you'll first need to install the swirl package:

```
install.packages("swirl")
```

If you've never run `swirl()` before, you will be prompted to install a course. You can do that with the `install_course` function. For example, to install the `R Programming` course, you would run:

```
library(swirl)
install_course("R Programming")
```

Once you've installed a course, every time you enter the swirl environment with `swirl()`, `R Progamming` should show up as a course option to select. You can enter `R Programming` to start lessons in that course by typing the number in fromt of it when you run `swirl()`.

Once you have at least one course installed, you call the `swirl()` function to enter the interactive platform in RStudio. The console will take you through a few prompts: you'll give swirl a name to call you, and take a look at some commands that are useful in the swirl environment. Those commands are listed further below.

```
library(swirl)
swirl()
```

After calling `swirl()`, you may be prompted to clear your workspace variables by running `rm=(list=ls())`. Running this code will clear any variables you already have saved in your global environment. While swirl recommends that you do this, it's not necessary.

Some of these lessons complement online courses through Coursera, so sometimes you will be asked after you complete a lesson if you want to report your results to Coursera. You should select "No" for that option each time.

B.1.1 Option 1

For **Option 1** of this homework, you will need to work through seven of the 15 available lessons in the `R Programming` course. Here are some suggestions for particularly uesful lessons that you could choose (the lesson number within the course is in parentheses):

R Programming course:

- Basic Building Blocks (1)
- Sequences of Numbers (3)
- Vectors (4)
- Missing Values (5)
- Subsetting Vectors (6)

- Logic (8)
- Looking at Data (12)
- Dates and Times (14)

Getting and Cleaning Data course:

- Manipulating Data with dplyr (1)
- Grouping and Chaining with dplyr (2)
- Dates and Times with lubridate (4)

Each lesson should take at most 10–15 minutes, but some are much shorter. You can complete the lessons in any order you want, but you may find it easiest to start with the lowest-numbered lessons and work your way up, in the order we've listed the lessons here.

You'll be able to get started on some of these lessons after your first day in class ("Basic Building Blocks"", for example), but others cover topics that we'll get to in weeks 2 and 3. Whether or not we've covered a swirl topic in class, you should be able to successfully work through the lesson. At the end of each lesson, you may be asked if you would like to receive credit for completing this course on Coursera.org. Always choose"no" for this option.

Again, you'll need to compose and turn in a few sentences for each lesson. Make sure to include a summary of what each lesson was about, and the most interesting thing about that lesson.

B.1.2 Option 2

If you're already somewhat familiar with R, you might want to choose your seven lessons from other swirl courses instead of or in addition to those available in the `R Programming` and `Getting and Cleaning Data` courses.

Check out the list of available Swirl Courses to see which ones you would like to install and check out available lessons for. For example, to choose a lesson in the `Exploratory Data Analysis` course, you would run:

```
library(swirl)
install_course("Exploratory Data Analysis")
swirl()
```

After entering the `Exploratory Data Analysis` course, you could choose from any one of its available lessons.

In your written summary for each lesson (again, a few sentences that cover a summary of the lesson and the most interesting thing you learned), make sure to specify which course each lesson you completed was from.

B.1.3 Special swirl commands

In the swirl environment, knowing about the following commands will be helpful:

- Within each lesson, the prompt `...` indicates that you should hit Enter to move on to the next section.
- `skip()`: skip the current question.
- `play()`: temporarily exit swirl. It can be useful during a swirl lesson to play around in the R console to try things out.
- `nxt()`: regain swirl's attention after `play()`ing around in the console.
- `main()`: return to swirl's main menu.
- `bye()`: exit swirl. Swirl will save your progress if you exit in the middle of a lesson. You can also hit the Esc. key to exit. (To re-enter swirl, run `swirl()`. In a new R session you will have to first load the swirl library: `library(swirl)`.)

B.1.3.1 For fun

While they aren't required for class, you should consider trying out some other swirl lessons later in the course. You can look through the course directory to see what other courses and lessons are available. For the first part of our course, you might find the "Exploratory Data Analysis" course helpful. If you would like to learn more about using R for statistical analysis, you might find the "Regression Models" course helpful.

B.2 Homework #2

Due date: Sept. 27 by 5:00 p.m.

For this assignment, you will submit the assignment to me **by email** by the due date. You should include three files in your submission:

1. The final rendered Word document (rendered from an RMarkdown file).
2. The original RMarkdown file used to create that final document.
3. The dataset you used for the assignment.

For this assignment, start by picking a dataset either from your own research or something interesting available online (if you're struggling to find something, check out Five Thirty Eight's GitHub data repository). You will then use this dataset to practice what you've learned with R so far. This will also be a chance, if you're using a dataset from your own research for me to get an idea of how you might be planning to use R in future research.

Very important note: Some research datasets have privacy constraints. This includes any datasets collected from human subjects, but can also include other datasets. For some projects, the principal investigator may prefer to keep the data private until publication of results. Before using a dataset for this project, please confirm with your research advisor that there are no constraints on the data. I do not plan to make the results of this assignment public, but I also do not want to us to be emailing back and forth a dataset with any constraints.

Using your dataset, create an RMarkdown document with the content listed below. Each section I've listed below should be included in the RMarkdown document as its own section, with a section header created using Markdown code.

- **Section 1: Description of the data**: Describe the dataset you are using, both in terms of the **content** (what is this data measuring? how was it collected? what kinds of research questions are you hoping to use it to answer?) and in terms of its **format** (what type of file is it saved in? what if it is in a flat file, is it fixed width or delimited? if it is delimited, what is the delimiter? if it is binary, what is the program that would normally be used to open it?).
- **Section 2: Reading the data into R**: Include code that reads the data into R and assigns it to a dataframe object that you can use later in the document. Explain in the text which R function you used to read in the data (e.g., `read_csv`) and which package it came from (if it was not a base R function). If there were any special options you needed to use (e.g., `skip` to skip some rows without data), list those and explain why you used them. Next, include some code to clean the data (e.g., rename columns, convert any dates into a "Date" format). You can filter to certain rows if you would like, but do **not** filter out missing values, as we'll want to learn more about those later.
- **Section 3: Characteristics of the data**: Describe the dataframe you just read in. How many rows does it have? How many columns? (Use inline code in the RMarkdown document to put this information into a sentence that reads "This dataframe has … rows and … columns.".) What are the names of the columns? What does each row measure (i.e., what is the unit of observation)? Include a table (using Markdown directly or `kable`) explaining what each column measures. This table should have three columns: (1) the column name in the R dataframe;

(2) a very brief description of what each column measures; and (3) the units (if any) of the measurement in the column.

- **Section 4: Summary statistics**: Pick three columns of the dataframe. Use the `summarize` function to get the following summaries of these columns: (1) minimum value; (2) maximum value; (3) mean value; (4) number of missing values. If there are missing values, make sure you use the appropriate options in summarizing these values to exclude those when calculating the minimum, maximum, and mean. Assign the result of this `summarize` call to a new R object, and print it out, so these summaries show up in your final, rendered Word document.
- **Section 5: Visualizing the data**: Create two plots of your dataframe. One should use a "statistical" geom (e.g., histogram, bar chart, boxplot) and one a "non-statistical" geom (e.g., scatterplot, line plot for time series). Explain why these plots help you learn more about this data and about the interesting research questions you're hoping to explore with the data. Be sure to customize the final size of each plot in the Word document using the `fig.width` and `fig.height` commands. For each plot, also be sure to customize the x- and y-axis labels. Finally, explain how each plot is following at least two of the principles of "good graphics" covered in week 4 of the course (Chapter 4 of the book)—if necessary, use `ggplot` functions and options to make the plots comply with some of these principles.