O'REILLY

Football Analytics with Python & R

Learning Data Science Through the Lens of Sports

Early Release

RAW & UNEDITED

Eric A. Eager & Richard A. Erickson

Football Analytics with Python and R

A Data Science Approach

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Eric A. Eager and Richard A. Erickson

Football Analytics with R and Python

by Eric A. Eager and Richard A. Erickson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Michelle Smith
- Development Editor: Corbin Collins
- Production Editor: Clare Jensen
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- October 2023: First Edition

Revision History for the Early Release

- 2022-07-21: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9781492099628 for release details.

978-1-492-09956-7

Chapter 1. Introducing to Python and R

Tools for Football Analytics

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Football analytics, and more broadly, data science, require a broad set of tools. Successful practitioners in these fields require an understanding of these tools. Statistical programming languages are a backbone of our data science toolbox. These programs allow us to clean our datasets, conduct our analyses, and readily reuse our methods. Although many people commonly use spreadsheets (such as Microsoft Excel or Google Sheets) for data cleaning and analysis, we find spreadsheets do not scale well. For example, when one has to work with large datasets like tracking data, which can contain thousands of rows of data per play, spreadsheets simply are not up to the task. Programming languages also allow for easy reuse because copy and pasting formulas in spreadsheets can be tedious and error prone. Lastly, spreadsheets allow undocumented errors. For example, spreadsheets do not having a method to catch a copying and pasting mistake.. Furthermore, modern data science tools allow code, data, and results to be blended together in easy-to-use interfaces. Common languages include Python, R, Julia, Matlab, and SAS. Additional languages continue to appear as computer science continues to advance.

As practitioners of data science, we use R and Python daily for our work, which has collectively spanned the space of applied mathematics, applied statistics, theoretical ecology and, of course, football analytics. Of the languages listed previously, Python and R offer the benefit of larger user bases (and hence likely contain the tools and models we need). Both R and Python (as well as Julia) are open source. *Open source* means two types of freedom. First, anybody can access all the code in the language, and this freedom is sometimes called *libre* freedom (think *free* like in *free speech*). This allows volunteers to help improve the languages, such ensuring that users can debug the code and extend the languages through add-on packages. Open source also offers the benefit of free to use for users, sometimes called *gratis* freedom (think *free* like in *free beer*). Hence, users do not need to pay thousands of dollars annually in licensing fees. We were initially trained in R, but have learned Python over the course of our jobs. Either language is well suited for football analytics (and sports analytics in general).

We encourage you to pick one language for the book and learn that language well. Should you need to learn a second programming language, it is easier if you understand the programming concepts behind a first language well. Then, you can relate the concepts back to your understanding of your original computer language. Although many people pick favorite languages and sometimes have arguments with each other over which coding language is better (similar to Coke versus Pepsi or Ford versus General Motors), we have seen both R and Python used in production and also used with large data and complex models. For example, we have used R with 100 GB files on servers with sufficient memory. Both of us began our careers coding almost exclusively in R, but have learned to use Python when the situation has called for it.

> **TIP**
>
> When picking a language, we suggest you “use what your friends use.” They can then help you debug and troubleshoot. If you still need help deciding, open up both languages and play around for a little bit. See which one you like better. Personally, the authors like R when working with data, because of R’s data manipulation tools, and Python when building and deploying new models because of Python’s cleaner syntax.

The Python Language

The Dutch computer scientist, Guido van Rossum, created Python as a programming language in 1991. The language is often considered clean and easy to understand because the code uses white space for formatting and grouping (for example, rather than using `{}` like R, Python uses blank space to group code). The language allows extensions through packages, although multiple package managers exist. Python can be used for everything from video game development to web-page hosting. The language is well designed with respect to computer science concepts, but can also be used as an interactive tool to explore data or scripting statistical methods.

Python’s numerical tools emerged as replacements for other languages. NumPy emerged as a replacement for MatLab’s matrix tools and other numerical methods. Matplotlib emerged as a plotting library inspired by MatLab’s plotting style and syntax. Pandas emerged as data frame tools, inspired by R’s `data.frame` objects. In contrast to a *matrix*, which typically only allows values of one type (such as only allowing numbers or characters), *data frames* allow columns to be different data types (similar to a spreadsheet with columns of text and columns of numbers). We use Miniconda for this book because it allows for more than Python to be installed and managed. For example, Miniconda can also be used to install R and R packages. Miniconda may be installed from the project’s page.

The R Language

R was created by Ross Ihaka and Robert Gentleman as a teaching language in 1993 in New Zealand. The R language is based upon the S language, which was first developed by the famous Bell Laboratories in 1975. Like Python, R has extendable packages. Unlike Python, base R natively supports many data types such as data frames and matrices, and R includes many basic statistical tools. R has been developed by statisticians, and many computer programmers feel the language is not as well polished as other languages such as Python or Java from a computer science perspective. We use R on a daily basis and like its ability to work with different data types.

R is also the *lingua franca* for statisticians, especially academics. In fact, *Significance*, the joint magazine published by the Royal Statistical Society, the Statistical Society of Australia, and the American Statistical Association published an issue in August 2018 titled the *R Generation* to celebrate the 25th anniversary of the R language and its emergence as subcultural phenomenon. Because of this prevalence and widespread use by academic statisticians, many cutting edge statistical methods are first developed as R packages by the researchers who develop the methods. Historically, statistical tool availability and the ability to work with diverse data types were strengths R had over Python. However, as Python has continued to become more widely used (and, arguably has become the most common language used in data science), this gap has narrowed. Likewise, R and Python can call easily call code from the other language, further leveling the playing field between the two languages.

Within R, three population sub-languages are emerging. First, *base R* (the default R packages) remains popular and stable. However, limitations exist with these methods and functions, and new ideas have emerged. The `data.table` package works quickly and with large data. We have used the `data.table` package to load 100 GB files on remote servers with ease and to program high-throughput data processing when we have days worth of data to process. The `tidyverse` set of packages emerged as Hadley Wickham and others committed much of their academic life to the question of how R should be written. We use the `tidyverse` on a daily basis because it is easy to read and works quicker than base R, and is more than quick enough (compared to `data.table`) for our daily needs. Like Python, R is used in daily production by some companies, including Eric's employer, PFF. However, many people prefer Python for production because Python can be used for everything from data analysis to web page development.

First Steps in Python and R

Opening a computer terminal may be intimidating for many people. For example, our spouses will walk by our computers, see code up on the screens, and immediately turn their heads in disgust. One of the author's spouses won't even allow him to open any terminals on her Chromebook. However, terminals are quite powerful and allow more to be done with less, once you learn the language. This section will help you get started using Python or R.

Different options exist for installing Python and R and then using the programs on your computer. You may download the programs directly from their project homepages, www.python.org for Python and www.r-project.org for R. However, you will then still need a program to work in as you program. We recommend using the miniconda program to manage Python and R on your computer because doing so allows you to easily use Jupyter Notebooks with your code and Jupyter Lab for editing. Furthermore, miniconda and the related Anaconda program are probably the most commonly used programs by data scientists for managing Python. We describe the how and why this program works in [Link to Come].

> **TIP**
>
> Historically, many if not most developers used a Unix- or Linux-based operating system, including macOS (which is based upon Unix). More recently, tools such as conda, Docker, and Windows Sub-system for Linux (WSL) allow people to develop on Windows as well. Likewise, Chromebooks now have developer modes that give full access to Linux tools on which the Chromebooks are built. However, we have observed that many companies are now moving to the cloud, which enables people to use any operating system (including the iPad-based iPadOS). Hence, operating system is becoming less important than the ability to use core tools that work across OS.

For you first steps in Python & R, do the following to obtain the program and get the initial add-on packages you will need for this book:

1. Download Miniconda. As of 2022, the homepage is *https://docs.conda.io/en/latest/miniconda.html*.
2. Open the Miniconda terminal if you are on Windows. If you are on Linux (including Chromebook) or macOS, open your Bash terminal.
3. Install the required core Python packages for the book by typing `conda install -c conda-forge scipy pandas seaborn jupyterlab`. Type `y` to confirm you want to install the required dependencies. . Install the required R packages for the book by typing `conda install -c r r-recommended r-tidyverse r-irkernel`. Type `y` to confirm you want to install the required dependencies.
4. Run `R -e 'IRkernel::installspec()'` to add R-kernel to Jupyter. This tells Jupyter to recognize R.
5. Open Python by typing `python` or R by typing `R`.

If you need additional help, online video tutorials exist on sites such as YouTube. For example, a search for “install mincoda video” on www.duckduckgo.com links to several helpful videos (we used DuckDuckGo as our example search engine because others such as Google customize results based upon individuals, thus your research results probably differ from ours).

NOTE

Both Python and R have flourished because they readily allow add-on packages. Conda exists as one tool for managing these add-on. [Link to Come] covers Conda and other add-ons. In general, packages in Python can be installed by typing `pip install <package name>` or `conda install <package name>` in the terminal outside of Python. Sometimes, you will need to use `pip3`, depending upon your operating system's configuration if you are using the `pip` package manager system. For a concrete example, to install the `seaborn` package, you could type `conda install seaborn` in your terminal. In general, packages in R can be installed by opening R and then typing **`install.packages("<package name>")`**. For example, to install the `tidyverse`, open R and type **`install.packages("tidyverse")`**

Now, that you have R or Python installed, you have an expensive graphing calculator (i.e., your computer). In fact, both of your authors, in lieu of using an actually calculator, will often calculate silly things like point spreads or totals in the console if in need of a quick calculation. Let's see some things we can do. Type `2+2`.

```
2 + 2
```

```
4
```

NOTE

People use comments to leave notes to themselves and others in code. Both Python and R use the `#` symbol for comments (the *pound symbol* for the authors or *hash-tag* for younger readers). Comments are code that the computer does not read. We use two comment symbols to tell you if a code block is Python (`## Python`) or R (`## R`)

Try other math operations such as multiplication (for example, `2*2`). What do you see? How might you take 2 to the third power, (2^3)? What happens if you try typing `2^3`? In R, you get something you probably expect:

```
## R
2^3
```

```
[1] 8
```

but in Python, you get

```
## Python
2^3
```

```
1
```

Python, you did not take an exponent; instead you took a bitwise `XOR` operator. To take an exponent in Python, use `2 ** 3`. This also works in R because the old S language, which R is based upon, included it as an undocumented feature.

You may also save numbers as variables. In Python, you could define `z` to be `2` and then re-use `z` and divide by 3.

```
## Python
z = 2
z / 3
```

```
0.6666666666666666
```

> **TIP**
>
> In R, either `<-` or `=` may be used to create variables. We use `<-` for two reasons. First, in this book this helps you see the difference between R and Python code. Second, we use this style in our day-to-day programming as well. [Link to Come] talks about code styles. Regardless of which operator you use, be consient with your programming style in R. Your future self will thank you.

In R, you can also define `z` to be `2` and then re-use `z` and divide by 3.

```
## R
z <- 2
z / 3
```

```
[1] 0.6666667
```

Next, we will create a data frame and then use this to create simple scatterplots. In Python, we first load the required packages, `pandas` and `seaborn`. Each time you want to use functions from a package, you need to use the package's name. To simplify our typing, we use an *alias* or nickname for the package. `pandas` commonly uses the nickname `pd`, which makes sense as a shorter version of `pandas`. `seaborn` commonly uses the nickname `sns`, which is a joke references to the character Samuel Norman Seaborn ("SNS") from the TV drama *The West Wing*.

```
import pandas as pd
import seaborn as sns
```

Next, we create two lists of values, `x` and `y`. In Python, lists are created with square brackets, `[ ]`, that are separated by commas, `,`.

```
## Python
x = [1, 2, 3]
y = [10, 11, 12]
```

We then put both of these into a dataframe. We need to put `x` and `y` into dictionaries. Python dictionaries consist of a pair and key. For example, `{"x", x}` takes our existing variable, `x` and creates a key with the name "x". We could also use any name such as `"Fred`, `football`, or `Green Bay"`.

TIP

Using single quotes around a name, such as `'x'`, or double quotes, such as `"x"`, are both acceptable to languages such as Python or R. Make sure you open and close the quotes with the same type. For example, `'x"` would not be acceptable to the languages.

You may use both single and double quotes to place quotes inside of quotes. For example, in a figure caption you might write `"Panther's score"` or `'Air temperature ("true temperature")'`. Or, in Python, you can use a combination of quotes later for inputs such as `"team == 'GB'"` because we need to nest quotes inside of quotes.

NOTE

Typing `print(...)` around objects is not required for R or Python much of the time. However, calling the function will ensure outputs are printed, which can sometimes be important. If in doubt, be explicit with the use of the `print()` function. We tend to include `print()` so that Python does not format the outputs in the Jupyter Notebooks used to create this book.

Next, let's create a data frame. We use a dictionary with the keys `x-axis` and `y-axis` with our previously saved `x` and `y` lists. We can the print this to the screen:

```
## Python
dat_python = pd.DataFrame({"x-axis": x, "y-axis": y})
print(dat_python)
```

```
   x-axis  y-axis
0       1      10
1       2      11
2       3      12
```

Finally, we plot our data using the `scatterplot()` function from `seaborn`. Inside the function, we tell Python to use the `data` from the `dat_python` data frame that we just created. Likewise, we tell Python to use the `"x-axis` data for the x-axis (horizontal axis) and `y-axis"` data for the y-axis (vertical axis). These variable names come from the column names of `dat_python`. this results in Figure Figure 1-1:

```
## Python
sns.scatterplot(data = dat_python, x = "x-axis", y = "y-axis")
```

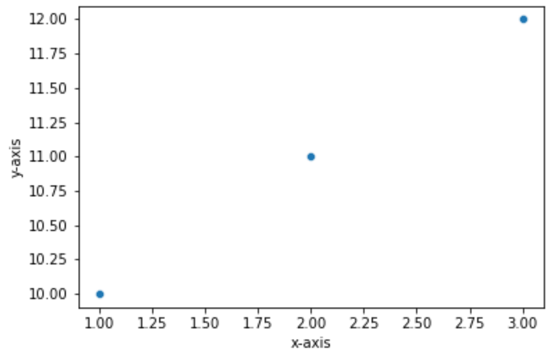

Figure 1-1. Example scatterplot with `seaborn` *in Python.*

We need to use quotes around `"x-axis` and `y-axis"` so that Python knows we want to use the names of the columns of `dat_python`. Without quotes, we could pass a variable to be the `x` or `y` input. For example, we could write the following:

```
x_name = "x-axis"
y_name = "y-axis"
sns.scatterplot(data = dat_python, x = x_name, y = y_name)
```

The power of passing objects in computer languages is confusing at first, but turns out to be quite powerful. For example, if you had to create plots for all teams in the NFL, you might read `x_name` from a list `["Green Bay", "Chicago",...]` and update inside of a loop or similar command.

We can use similar steps for R. First, we load the `ggplot2` package for plotting (the `ggplot2` package is included within the `tidyverse` set of packages, hence you already installed it, likely without realizing it if you followed our conda direction earlier).

```
## R
library(ggplot2)
```

Then, create `x` and `y` vectors. R uses the `c()` function to combine or concatenate items into a vector. Notice we use `<-` to define or save variables:

```
## R
x <- c(1, 2, 3)
y <- c(11, 12, 13)
```

Next, create a data frame in R. Notice R does not require us to use a package to have access to data frames. R also drops the dash, `-`, and replaces it with a period, `.`. R, especially `base R`, does not like special characters to be used in column names.

> **NOTE**
>
> Unlike the `DataFrame` from `pandas` in Python, the `data.frame` in `base R` does not easily allow special characters or spaces to be part of column names. Although this can be done using the backtick, for example `` \`x-axis\` ``, we find it best to avoid this use in most situations. Shorter column names are also easier to type and avoid cumbersome uses of backticks in our code. The backticks key is found to the left of the "1" key on standard US keyboards.

```
## R
dat_r <- data.frame("x-axis" = x, "y-axis" = y)
print(dat_r)
```

```
  x.axis y.axis
1      1     11
2      2     12
3      3     13
```

We can then plot this using `ggplot2`'s `ggplot()` function. `ggplot2` has its own language, based upon the *Grammar of Graphics* by Leland Wilkinson (Springer 2005) and implemented in R by Hadley Wickham during his doctoral studies at Iowa State University. The base function, `ggplot()` tells R we are using `ggplot2`. We tell `ggplot()` what `data` we are using as well as the `aesthetics` of our plot, in this case, the `x` and `y` axes. We then add a geometry of points, `geom_point()` to the plot. This results in Figure 1-2. Although confusing at first, `ggplot2` provides a powerful syntax for describing data graphically. Pedagogically, we tend to agree with David Robinson, who describes his reasons for teaching plotting with `ggplot2` over base R in a blog post titled Don’t teach built-in plotting to beginners (teach ggplot2).

```
ggplot(data = dat_r,
       aes(x = x.axis, y = y.axis)) +
geom_point()
```

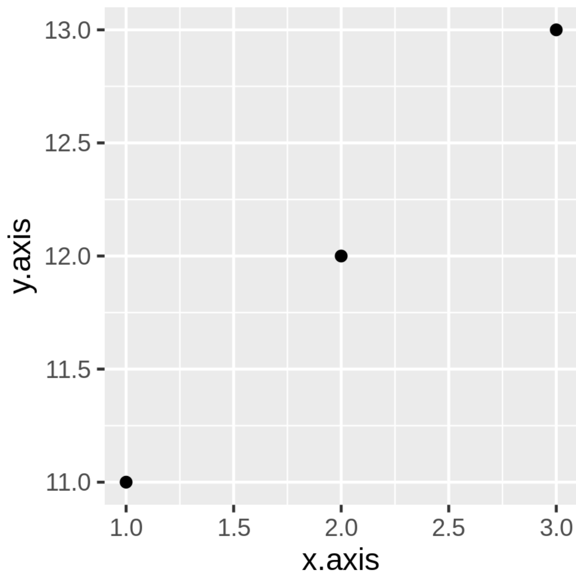

Figure 1-2. Example scatterplot with `ggplot2` in R.

Congratulations, you have likely now created your first plot using a scripting language!

Scripts and Integrated Development Environments

But, what if we want to save our inputs? We can write a script file to save our code. We will use these for the early part of the book. Later, we will switch to using Jupyter Notebooks, which allow code and text to be embedded together. In the end, typing in the terminal is not effective or easy. We can use powerful code-editing tools called *Integrated Development Environments* (IDEs). Much like football fans fight over who is the best quarterback of all time, programmers often argue over which

IDEs are best (well, not exactly "much like"). We use Jupyter Lab because it is easy to install from conda and is simple enough to not have too many features to overwhelm new programmers.

Although powerful, IDEs can have downsides. Some IDEs are complex, which can be great for expert users, but overwhelming for beginners and casual users. For example, the emacs text editor has been jokingly described as an operating system with a good text editor or two built into it. Likewise, some professional programs feel that the shortcuts built into some IDEs limit or constrain understanding of languages because they do not require the programmer to have as deep of understanding of a language. However, for most users, especially casual users, the benefits of IDEs far outweigh the downsides.

We use the JupyterLab editor for this book because it works with both Python and R. Jupyter Lab grew out of Jupyter Notebooks. Jupyter Notebooks allow people to include code directly with text describing the code and the code's output, much like a lab notebook from science class. Fernando Pérez and Brian Granger spun Jupyter Notebooks off of the Interactive Python (IPython Project (*https://ipython.org/)*) to work with more languages. In fact, *Jupyter* stands for *Julia*, *Python*, and *R*. These were the three languages that Jupyter was originally created to work with. Jupyter now works with many other languages.

Many useRs (slang for users of R) like the RStudio IDE] *https://www.rstudio.com/)*, and, if you decide to use R, we encourage you to check out this program. A lot of different Python IDEs exist for Pythonistas (slang for users of Python). We personally just use Jupyter Lab, but common popular choices include *Integrated Development and Learning Environment* (IDEL (*https://docs.python.org/3/library/idle.html);* that comes with Python), Visual Studio (*https://visualstudio.microsoft.com/;* Microsoft's IDE that works with both R and Python), and PyCharm (*https://www.jetbrains.com/pycharm/*). If you already use another IDE for a different language at work or elsewhere, that IDE also likely works with Python and possibly R as well.

> **NOTE**
>
> People who use Python are commonly called Pythonistas. People who use R are commonly called useRs.

We have included screenshots of three key features of Jupyter Lab. When you first open Jupyter Lab, you will see a launcher such as in Figure 1-3. The launcher allows you to start (or *launch*) a Jupyter Notebook running Python or R, open a console (or terminal) for Python or R, and open other types of programs include an operating system-specific terminal, a plain text file, a Markdown file, a Python script file, an R script file, and programs' build-in help files.

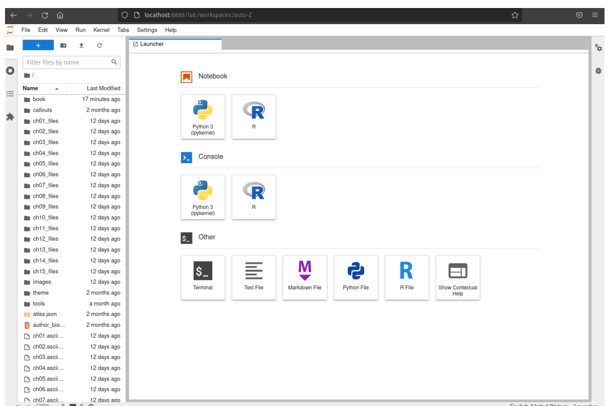

Figure 1-3. Launcher with Jupyter Notebook.

Opening a Python terminal, such as in Figure 1-4 gives you many options. However, this is mainly like the command-line terminal you started earlier. To run code, type it in the box at the bottom and and then type **`shift`** + **`enter`** to run the code.

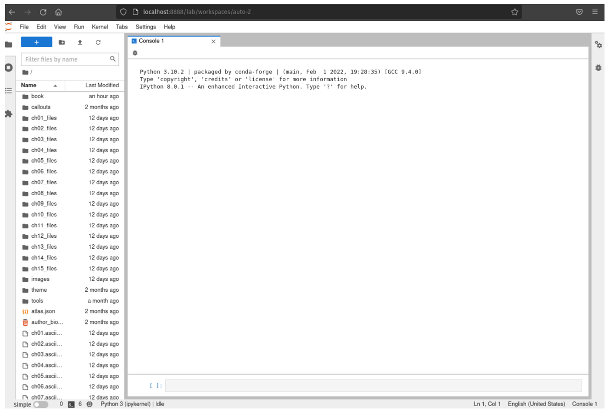

Figure 1-4. R script, launching terminal with Jupyter Notebook.

> **TIP**
>
> Running code such as R or Python from inside Jupyter lab requires you to type **`shift`** + **`enter`**. This is true for both the console and script files.

Finally, from the launcher, you can open a Python or R script file, such as the R script shown in Figure 1-5. From this script file, you can right-click on the top and launch a console for the script file, from the drop-down menu shown in Figure 1-5. This allows you to interactively run a script file, line-by-line to see what happens.

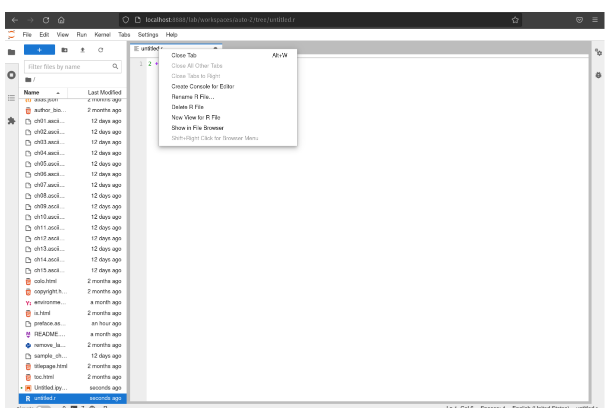

Figure 1-5. Python terminal with Jupyter Notebook.

During the course of this book, we will be using Python and R interactively. However, some people also run these languages as batch files. A *batch file* simply tells the computer to run and entire file and then spits out the outputs from the file. An example batch file might calculate summary statistics that get run weekly during the NFL season by ProFootball Focus and then placed into client reports.

Overview of Datasets

Any and all good data endeavors require datasets from which to work. In this book, we're going to work on a few of the cornerstone public datasets in the football analytics space. In 2017 Max Horowitz, Sam Ventura, and Ron Yurko built an R package called `nflscrapR`, which parsed publicly-available NFL play-by-play data, and supplemented it with expected points added (EPA) and win probability (WP) information on each play. Later, Ben Baldwin and Sebastian Carl updated the work in the form of the R package `nflfastR`, which is now the most commonly-used public data set in the football analytics space.

While the `nflfastR` data is very clean, thanks to Ben and Sebastian, not every situation is going to give rise to clean data. Most of the time we spend as data scientists - at least during the initial phase of work after data is collected - is spent cleaning and formatting data. In the spirit of this reality, Chapter 3 will have you scraping and cleaning datasets from Pro Football Reference (*https://www.pro-football-reference.com/*), the best source for raw American football (and other sports) data. For this book, readers will scrape and analyze NFL Draft and NFL Combine data.

Suggested Reading

If you get really interested in analytics, here are some suggestions for further reading:

- Lewis, Michael. *Moneyball: The art of winning an unfair game*. WW Norton & Company, 2004.

Lewis describes the rise of analytics in baseball and shows how the stage was set for other sports. The book helps us think about how modeling and data can help guide sports. A movie was also made of this book as well.

- Silver, Nate. *The Signal and the Noise: Why So Many Predictions Fail, but Some Don't*. Penguin, 2012.

Silver describes why models work in some instances and fail in others. He draws upon his experience with poker, baseball analytics, and running the political prediction website fivethirtyeight.com. The book does a good job of showing how to think quantitatively for big picture problems without getting bogged down into details.

Chapter 2. Exploratory Data Analysis

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

There are relatively few people who can live entirely in the abstract. We like to see the data that we're working with. We like to touch the data that we're working with. Metaphorically, of course. We like to understand data. We like to verify the quality of data so that we don't do more harm than we do good. Plotting data serves as a first step of the *Exploratory Data Analysis* process (or EDA for short). We focus on plotting for our first foray into EDA. The term EDA was coined by the American statistician John Tukey as he prompted people to thoroughly understand their data before formal statistical analysis. This is very important in football and sports in general, as analytically-oriented analysts are often viewed as outsiders, and hence failing to take into account nuances in the data can undermine the efforts of your or your team's analyses. EDA includes tools such as plotting data and summarizing data to see what is going on.

NOTE

John Tukey also coined other terms you may know or will hopefully know by the end of this book including *boxplot* (a type of graph), *Analysis of Variance* (ANOVA for short; a type of statistical test), *software* (computer programs), and *bit* (the smallest unit of computer data, usually represented as 0/1; you're probably more familiar with larger units such as the byte, which is eight bits). Tukey also helped the Princeton University football team implement data analysis. To read more about Tukey's life and contributions, checkout the obituary written by David Brillinger that appeared in the *Annals of Statistics* (*https://www.jstor.org/stable/1558729*).

We use EDA throughout football analysis and as an iterative process. First, we formulate our objectives, such as predicting the winner of a game or who will cover the spread. Second, we acquire data to answer our questions. Third, we read data in a program like R or Python and then we explore the data's structure. This helps us understand the data's form and spot check the data's quality for major problems, such as missing data or corrupted data. Fourth, we plot the data. This allows us to visualize data and start to understand its shape. For example, if somebody recorded minutes rather than seconds we likely could see this wrong entry in the data. Fifth, we summarize data. This allows us to quantify what is going on with the data. Sixth, we use statistical models to estimate patterns in the data.

Finally, we go back and use plots and summaries to help explain our models and the stories we seek to tell with our data. If we are using deductive reasoning, we will start with ideas or hypotheses we want to test from our data. If we are using inductive reasoning, we will let the data guide our conclusions and hypotheses from the data. We use both approaches on a regular basis. For example, if we want to understand why some players are better than others, we can test the hypothesis that quarterbacks who are drafted earlier in the NFL draft are more productive than those taken later. Conversely, if we want predict fantasy football performance during a game or a season, we might build the model, test the model, and then put the model into an easier-to-use format.

We start with the technical skills of plotting. Plotting will help you *see* the data and gain understanding. We often plot data first, use other tools such a models and statistical summaries to explore data, and end with creating summary plots of the data to drive home our point. Plus, we find visualizing to be one of the most fun parts of telling data stories.

[Link to Come] and [Link to Come] cover data acquisition and wrangling as well as more advanced data importing. We included these materials after this current chapter because we find the topics to be easier once you have gained some experience with programming. Acquiring data and then wrangling it into a usable format is important, but can be more tedious. Think of data skills as analogous the weight-training or cross-training of football analytics. Casual players such as recreation leagues or intramural players may not need these train for these skills. However, competitive players need cross-train. Furthermore, some people only focus on cross-training exercises, such as as competitive weight-lifters or sprinters. Likewise, some people focus on working with data, and are often called *data engineers*. A primary job for a data engineer is to focus on data workflows.

After learning how to work with data, we transition to more ways to use data to inform football. Chapter 5 provides an introduction to statistics and modeling. Chapter 6, [Link to Come], and [Link to Come] cover different modeling approaches and build upon each other. [Link to Come] and [Link to Come] tie together EDA into telling stories. The later portions of our book touch on advanced topics with [Link to Come] describing advanced modeling topics and [Link to Come] covering advanced tools we use on daily basis.

Motivating Problem: How Do We "See" or Explore Passing Data?

Likely, you learn best by doing. We seek to teach by example in this book, using football data. For this chapter, we will use yards from passing play data from the 2020 week 2 game between the Green Bay Packers and

Detroit Lions. We also include passing play data from the 2020 game between the Detroit Lions and Houston Texas for you explore on your own as `det_hou_2020_pass.csv`. We obtained both of these datasets using the *nflfastR* package. We describe this package in Chapter 3 so you can start to get your hands on data to answer your specific questions.

Perhaps we are interested in the Green Bay Packers and Detroit Lions passing game. We may seek to answer specific questions. For example:

- Does either team have better passing based upon the side of the field?
- How far do the teams take the ball after successful passes?
- Does scrambling change where the ball goes?

First, we need to read in the data. In Python, we use the `pandas` package, which we load with the `import` command. You can then read the data into the computer. We need to give the data a name in the computers, which can be tricky because we want something long enough to be descriptive but short enough to be easy to type. The name, `gb_det_20202_pass` tells us the teams (`gb` for Green Bay and `det` for Detroit, with the home team first and away team second), year (`2020`), and type of data (`pass` for passing). You could use any valid name you wanted including silly names such as `fred` or low information names such as `dat`. We also assume the data is in a sub-folder, `data`.

> **TIP**
>
> Naming objects can actually be hard when programming. Try to balance simple names that are easier to type with longer, more informative names. This can be especially important if you start writing scripts with longer names. The most important part of naming to create names that you, and hopefully others, will understand when you read the code later.

```python
import pandas as pd
gb_det_2020_pass = pd.read_csv("./data/gb_det_2020_pass.csv")
```

Similarly, we can read the data into R use base R's `read.csv()` and name the data the same name:

```
gb_det_2020_pass <- read.csv("./data/gb_det_2020_pass.csv")
```

Before we dive into the data, can examine the top or *head* of the data. For both Python and R, we also use the `print()` function around the heads of the data. In Python, we use `.head()` after the data object. Then, we wrap `print()` around the head of the data frame.

> **NOTE**
>
> `print()` is not required for most functions because the languages have a default command for printing. However, explicitly calling the command ensure we know exactly what will occur.

```
print(gb_det_2020_pass.head())
```

```
  posteam  yards_after_catch  air_yards pass_location  qb_scramble
0     DET                0.0          5        middle            0
1     DET               16.0         13          left            0
2     DET                3.0          3          left            0
3      GB               11.0          4        middle            0
4      GB                4.0          0         right            0
```

In contrast with R, we first wrap `head(...)` around the data frame and then wrap `print(...)` around the head function.

```
print(head(gb_det_2020_pass))
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1     DET                 0         5        middle           0
2     DET                16        13          left           0
3     DET                 3         3          left           0
4      GB                11         4        middle           0
5      GB                 4         0         right           0
6      GB                NA         0         right           0
```

Notice how Python starts numbering with `0` whereas R starts numbering with `1`. Python uses standard convention for computer science whereas R uses standard convention for mathematics and statistics. This reflects the history of the languages' authors. Also, as a smaller point, notice R's `head()` prints the first 6 (1, 2, 3, 4, 5, and 6) rows while Python's `.head()` prints the first 5 (0, 1, 2, 3, and 4) rows.

> **WARNING**
>
> Python starts numbering a 0. R starts numbering at 1. Many an aspiring data scientist has been tripped up if using both languages.

Next, we'd like to know about our data. Specifically, the data about data or *meta-data*. For the columns, `posteam` is the team in possession of the ball at the start of the play. `yards_after_catch` is the number of yards gained after the reception of the ball. `air_yards` is how far the ball was passed in the air (whether completed or not). `pass_location` is which side of the field the quarterback passed the ball to. `qb_scramble` is a binary response (that is, `0` for no or `1` for yes) for if the quarterback had to scramble.

> **WARNING**
>
> With any data, mane sure you understand the meta-data. For example, what does 0 and 1 mean? Or, do the authors use 1 and 2 for the levels? We have heard about studies being retracted becaues the data analystic and scientists mis-understood the meta-data and the uses of 1 and 2 versus the standard 0 and 1. For example, a 2021 article in *Significance* describes an occurance of this mistake (*https://www.doi.org/10.1111/1740-9713.01522*).

Applying EDA

We will demonstrate how we use EDA by examining the pass data. First, we will start with a broad examination of the data. You will examine if there are fundamental differences between the two teams or if any patterns emerge in the data. Second, we will focus in on specific questions with the data. For example, *Is there a relationship between between air yards and yards after the catch?* or *Does either team do better on on aspect of offense than another?*

Uncovering broad trends can help you understand data and refine your questions. For example, with the passing data, which side of the field does a team throw to more often. Does a team defend more poorly? Or, if picking fantasy players, which side do you hope your wide receiver plays? Lastly, EDA method also allows you to check data for any outliers of possible mistakes. Data points that stand out might be mis-entered, or worth investigating more. We will teach you tools for removing outliers in Chapter 4. However, these points might also belong. For these data points, we may want to dig in deeper to figure out the story behind them.

We view and use EDA and center and key component of our storytelling process. First, we use simple plots to visualize the data. These helps us to both get a feel for the data as well as check and develop our intuition. Next, we probe the data by expanding the plots to include more details. If we do not understand our data, we dig in and figure out what is going on. Perhaps we need to make sure we understand the data source or that the data source is error free. Lastly, we generate future questions. These questions often motivate us to find additional data to repeat the process.

We view EDA as an iterative process. We plot the data (this chapter). Then, we summarize and model the data (something we Chapter 5 starts to cover). Next, we use plots to summarize our models. While doing this, we prepare the data to tell a story and then communicate with our stakeholders. Lastly, we repeat plotting, modeling, and communication as necessary.

Histograms

We start by examining the yards after catch. To do, this, we use a histogram. Histograms summarize data by placing counts of the data into bins. Different programs have different defaults bin width. For example, Figure 2-1 from Python has a total of 7 bins by default. In contrast, [Link to Come] from R has 30 bins by default. These different bins illuminate different parts for the data. For example, the R plot shows that some pass yards are negative, where this may not be as obvious from the Python plot. However, the R plot also looks very fragmented. The number of *idea* bins for your story therefore is somewhere between 7 and 30.

> **WARNING**
>
> Intentionally using the wrong number of bins to hide important attributes of your data is considered fraud by the larger statistically community. Be thoughtful and intentional when you select the number of bins for a histogram.

In Python, we import the `seaborn` package and then use the `displot()` function with our data. All of the function arguments (inputs) have intuitive names: `data`, the input `DataFrame`; `bins`, the number of bins; and the `x` variable to plot, which is a column in the `DataFrame`. If we wanted to add more to plot, we would probably need to call additional functions. This type of plotting is *pen-and-paper* because it as analogous to drawing a plot with a pen on paper and adding items one-at-a-time. `matplotlib`, which `seaborn` is built upon this philosophical approach:

```
import seaborn as sns
sns.displot(data = gb_det_2020_pass, bins = 7, x = "yards_after_catch")
```

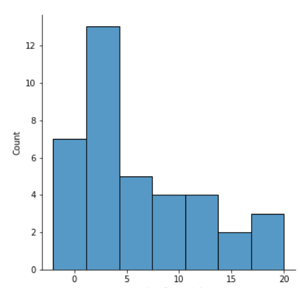

Figure 2-1. Example histogram plot with `seaborn` in Python.

Plotting with `ggplot2` uses a different type of syntax compared to `seaborn`. `ggplot2` is based upon a coherent syntax, the *grammar of graphics* mentioned earlier in Chapter 1. First, we load the `tidyverse` that contains `ggplot2`. Then, we use the `ggplot()` function. We specify `data`, like with seaborn. However, ggplot() has *aesthetics* (aes) as a function for the plot. For this simple plot, the only aesthetics is the x-axis. We then add a geometry to the plot, specifically a histogram using geom_histogram(). This has an obeject called bins, which like the argument for displot:

```
library(tidyverse)
ggplot(data = gb_det_2020_pass, aes(x = yards_after_catch)) +
    geom_histogram(bins = 30)
```

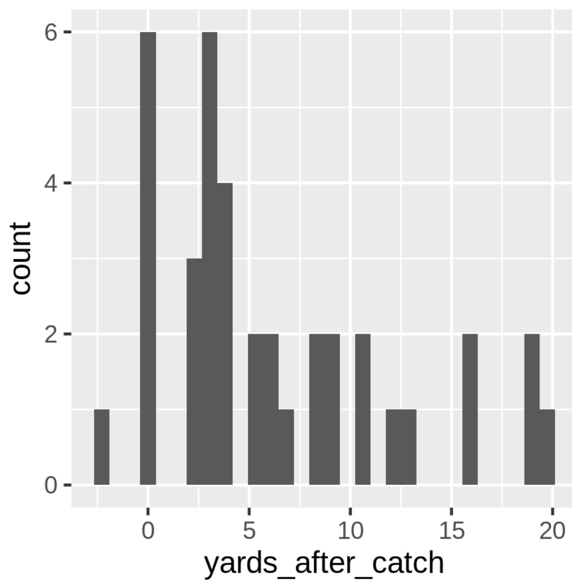

Figure 2-2. Example histogram plot with ggplot2 in R.

Comparing seaborn to ggplot2, we find both syntax helpful at times and frustrating at others. Both are highly customizeable and do share some similarities in terms because parts of seaborn are modeled after ggplot2. If you have not yet picked one language, we encourage you to work through this chapter and see which plotting language jumps out to. Plotting is an important part of football analytics.

> **TIP**
>
> Work through both Python and R in this chapter if you have not yet picked a language. See which type of plotting comes most naturally to you.

From both figures, all of the data seems reasonable. No outliers appear and the values seem reasonable. That data do not follow a normal or bell curve, as a player is much more likely to have big (in absolute values) yards after the catch in the positive direction than in the negative direction. We will talk about this more in future chapters. But, we are missing something very important. We have two teams in the game. Quite likely, the teams had different passing distributions. In fact, we are seeking to examine if these differences exist.

We can plot both teams distribution as their histogram. We will spit, or *facet*, each plot into columns. In `seaborn`, we add a `col` argument and plot facet columns by the team in possession of the ball `"posteam"`. Notice like the `x` argument, `"yards_after_catch"`, this input is in quotes as well:

```
sns.displot(
    data = gb_det_2020_pass, x="yards_after_catch", col="posteam",
    binwidth=3, height=3,
)
```

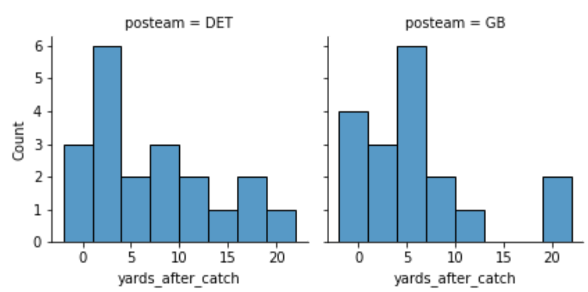

Figure 2-3. Example histogram plot with each possession team being faceted by column in `seaborn` in Python.

Notice here that in this particular game, the Detroit Lions had a more evenly-distributed set of yards after the catch, while the Packers' yards after the catch data was distributed more closely to zero, with a small set of bigger plays (~20 yards).

TIP

Line breaks and white-space are important for coding. These breaks help make our code easier to read. Python and R also handle line breaks different, but, sometimes, both languages treat line breaks as special commands. In both languages, we often split function inputs in script files to create shorter lines that are easier to read. For example, we space a function like

```
my_plot(data = big_name_data_frame,
        x = "long_x_name",
        y = "long_y_name")
```

to break up line names and make our code easier to read. In R, we need to make sure the comma stays on a previous line. In Python, we may need to use a \ for line breaks. For example, we would need to use:

```
x = \
    2 + 4
```

In R, we add a new command, `facet_grid(...)` to the old plot. We use an tilde, ~, to with the inputs row ~ column. This may be read to a human as row faceted by column. To either only facet by rows or only facet by columns, use a period, `.`, for the non-used entry. For example, to facet by possession team, we add `facet_grid(. ~ posteam)`. Notice that R does not use quotes around the plotting parameters (in contrast to Python):

```r
ggplot(data = gb_det_2020_pass, aes(x = yards_after_catch)) +
    geom_histogram(binwidth = 3) +
    facet_grid(. ~ posteam)
```

TIP

The ~ sybmol is to the left of the one key on a standard US keyboard and requires the `shift`-key to access.

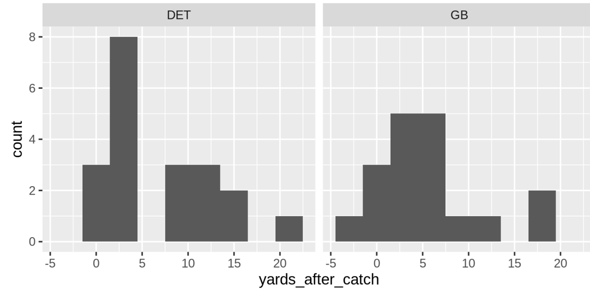

Figure 2-4. Example histogram plot with each possession team being faceted by column in `ggplot2` in R.

Boxplots

Histograms allow use to *see* the distribution of data points. However, they can be cumbersome, especially if we have many different predictors variables we seek to explores. Boxplots are a common plotting methods for summarizing data. Boxplots get their name because they have a box that contains the middle 50% of the data. That is to say, 50% of the data occurs within the box. The line in the middle of the box is the median, or the line where half of the data falls above the line and half of the data falls under the the line. Boxplots are sometimes also known as *stem-and-whisker* plots because lines extend above and under the box. These contain the remainder of the data other than outliers. Outliers are points that are more than 1.5 times the interquartile range from either the first or third quartile. These outliers are plotted with dots. We show you the default definition for outliers in the next paragraphs.

Table 2-1. Parts of a boxplot.

Part name	Range of data
Top dots	Outliers above the data
Top whisker	100% to 75% of data, excluding outliers
Top portion of box	75% to 50% of data

Line in middle of box	50% of data
Bottom portion of box	50% to 25% of data
Bottom whisker	25% to 0% of data, excluding outliers
Bottom dots	Outliers under the data

To help you see a boxplot, we contract the parts over a histogram in Figure 2-5. First, we take the yards traveled in the air by passes (whether completed or not) and create a faceted histogram. Second, we plot a blue line at the median. Third, we add red lines for the 25th and 75th quantiles of data (that is to say, 50% of data lies between these two lines). The red lines also denote the *interquartile range* or IQR for short. Fourth, we add gold lines for where Python or R consider the cutoff for outliers to be. To calculate the This cutoff for upper outliers is the 75th quantile + 1.5 × the IQR. This cutoff for the lower outliers is the 25th quantile - 1.5 × the IQR.

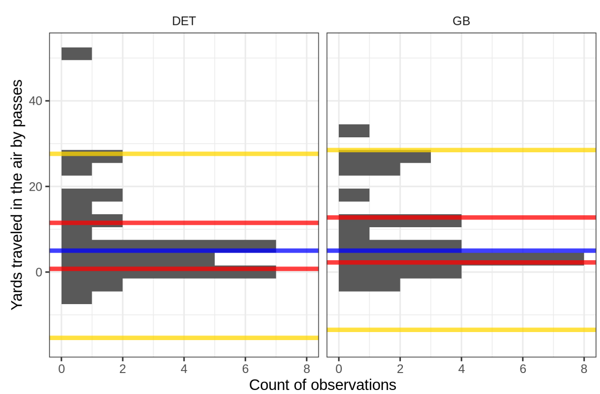

Figure 2-5. Histograms of plot of yards traveled in the air by passes with facet columns by the possession team. The blue line is the median, the red lines are the upper and lower limits of the interquartile range, and the gold lines are the cutoff values for outliers.

Next, we take the same colors and plot them over a boxplot. We jitter the points so that they are non-overlapping.

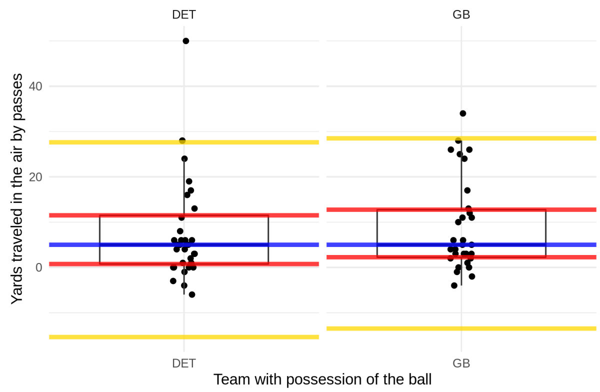

Figure 2-6. Boxplot of plot of yards traveled in the air by passes with facet columns by the possession team. The blue line is the median, the red lines are the upper and lower limits of the interquartile range, and the gold lines are the cutoff values for outliers.

Creating boxplots is easy to do in both `seaborn` and `ggplot2`. In `seaborn`, we use the `boxplot` function. We specify `data` to be `gb_det_2020_pass`, `x` to be `"posteam"`, and `y` to be `"air_yards"`, the distance the ball travels in the air from the line of scrimmage during a pass. In addition to creating the boxplot, we also save the boxplot to be an object in Python, `pass_boxplot` This allows us to start customizing the figure. Specifically, we set the x-label by using the function `.pass_boxplot.set_xlabel(...)` on the saved object, `pass_boxplot`. Likewise, we repeat for the y-label with `.pass_boxplot.set_ylabel(...)`. This creates Figure 2-7:

```
pass_boxplot = sns.boxplot(data = gb_det_2020_pass,
                           x = "posteam", y = "air_yards")
pass_boxplot.set_xlabel("Team with possession of the ball")
pass_boxplot.set_ylabel("Yards traveled in the air by passes")
```

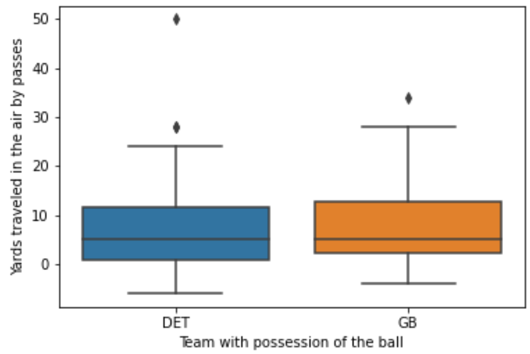

Figure 2-7. Boxplots of plot of yards traveled in the air by passes with facet columns by the possession team. The example is from `seaborn` in Python.

Notice that, after we saw that Detroit had a better yards after the catch distribution than the Packers, the Packers had the better air yards, which is sometimes a negatively-correlated metric with yards after catch, as shorter passes are intended to generate a run after the catch.

We can create a similar figure in R using the boxplot `geom`, `geom_boxplot()` (Figure 2-8). We can also add the x-label using `xlab(...)` and the y-label using `ylab(...)`. Lastly, we change the theme using `theme_bw()` to remove gray background from the plot. This last choice is largely one of personal preference:

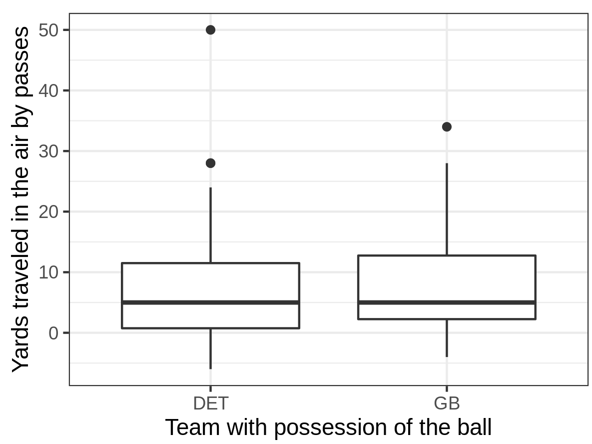

Figure 2-8. Boxplots of plot of yards traveled in the air by passes with facet columns by the possession team. The example is from `ggplot2` in R.

Scatterplots

Boxplots and histograms allow us to see one variable. Often, we want to see two variables. Scatter plots show two variables plotted against each other. Sometimes these are called *x*-*y* plot because the horizontal (left-right) axis is the *x*-axis and the vertical (up-down) axis is the *y*-axis. These plots let us see how two variables interact with each other and their relation. We often find these plots to a workhorse of showing data.

With `seaborn`, we use the `scatterplot()` function. We tell `scatterplot()` to use the `gb_det_2020_pass` data and plot `"air_yards` on the x-axis and `yards_after_catch"` on the y-axis. By convention, the feature on the x-axis usually thought to *predict* the feature on the y-axis, if a casual relation exists. In this case, one football game, there is no strong reason to expect a casual relation between air yards and yards gained after the catch (assuming a successful reception, as notice notice that incomplete passes have a `yards_after_catch` reading of `NA`), even if a relationship is expected to exist over a larger timeframe (e.g. a season).

```
sns.scatterplot(data = gb_det_2020_pass,
                x = "air_yards",
                y = "yards_after_catch")
```

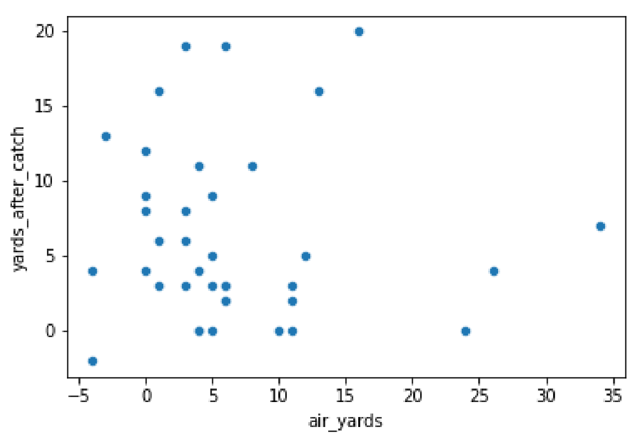

Figure 2-9. Scatterplot of passing yards gained in the air (x-axis) versus yards gained after the catch on the ground (y-axis). The example is from `seaborn` *in Python.*

Scatter plots may also be created with R. We set the `x` and `y` aesthetics to be the columns we want to plot. We use `geom_point()` to add points to the plots. This creates Figure 1-2.

```
ggplot(data = gb_det_2020_pass, aes(x = air_yards, y = yards_after_catch)) +
    geom_point()
```

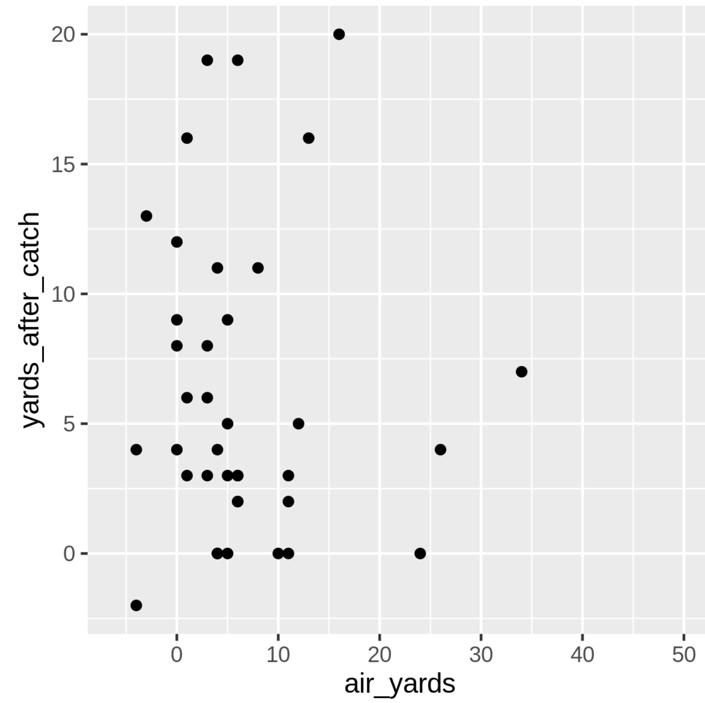

Figure 2-10. Scatterplot of passing yards gained in the air (x-axis) versus yards gained after the catch on the ground (y-axis). The example is from `ggplot2` in R.

We can also facet the scatter plots, just like the boxplots Figure 2-11. In `seaborn`, this is slightly different syntax. We use `FacetGrid` to create an empty plot and then use the `map` function to apply (or, in Python-speak) *map* to `scatterplot` function to the facet grid object. We also tell the mapping to use "`air_yards` and `yards_after_catch`":

```
yards_plot = sns.FacetGrid(data = gb_det_2020_pass, col="posteam")
yards_plot.map(sns.scatterplot, "air_yards", "yards_after_catch")
```

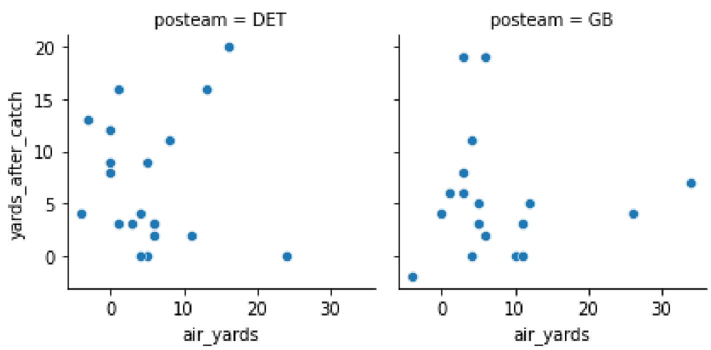

Figure 2-11. Example faceted scatterplot from `seaborn` in Python.

With `ggplot2`, we can also created a faceted plot. For this, we simply add `facet_grid(.~ posteam)` to the previous scatter plot. In general, we think `ggplot2` has a more predictable and internally consistent language than `seaborn` and that is one reason we often use `ggplot2` on a daily basis.

```
ggplot(data = gb_det_2020_pass, aes(x = air_yards, y = yards_after_catch)) +
    geom_point() +
    facet_grid(.~ posteam)
```

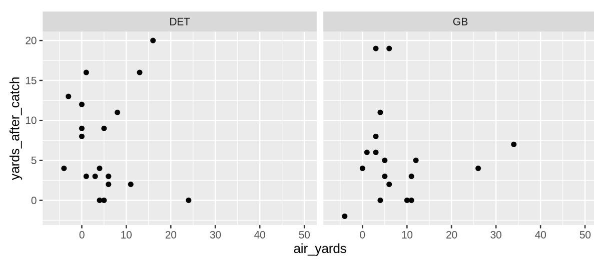

Figure 2-12. Example faceted scatter plot from `ggplot2` in R.

Colors and shapes

Many different methods exist for changing figures and allowing us to see more groupings, Figure 2-13. You can change the shape of point in scatter plots. In Python, this is the called the style. For example, rather than faceting by `posteam`, we can change the point's shape:

```
sns.scatterplot(data = gb_det_2020_pass,
                x = "air_yards",
                y = "yards_after_catch",
                style = "posteam")
```

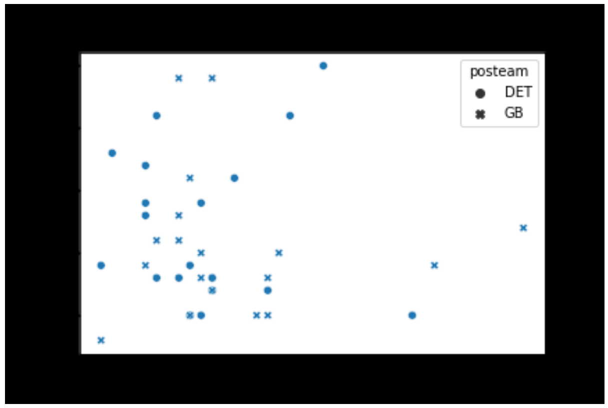

Figure 2-13. Example changing shapes plot from `seaborn`.

We can also change the shape in R Figure 2-14. You do this by changing the `shape` aesthetic:

```
ggplot(data = gb_det_2020_pass,
       aes(x = air_yards, y = yards_after_catch, shape = posteam)) +
    geom_point()
```

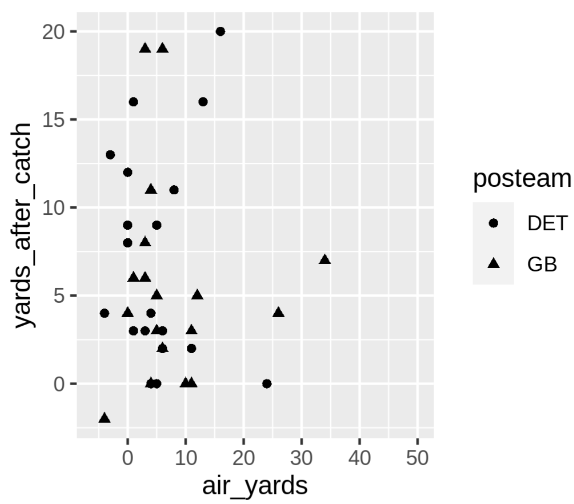

Figure 2-14. Example changing shapes plot from `ggplot2`.

Similar to changing shape, we may also change point types by color. You can also change shape and color to allow for more variables to be plotted, although we do not do this in our examples here. In `seaborn`, color is called `hue`:

```
sns.scatterplot(data = gb_det_2020_pass,
                x = "air_yards",
```

```
y = "yards_after_catch",
hue = "posteam")
```

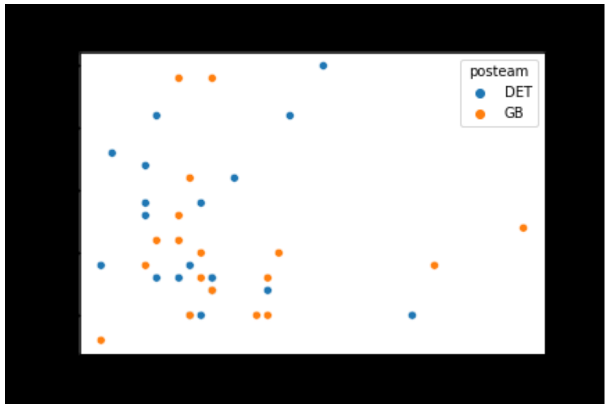

Figure 2-15. Example changing colors plot from `seaborn`.

In R, we change the color using the `color` aesthetic. We also change the colors using `scale_color_manual(...)` because the default colors are hard for people (including one of the authors) with color deficiencies (commonly known as color blindness, such as red-green colorblindness) to see:

```
ggplot(data = gb_det_2020_pass,
       aes(x = air_yards, y = yards_after_catch, color = posteam)) +
    geom_point(binwidth = 3) +
    scale_color_manual(values = c("red", "blue"))
```

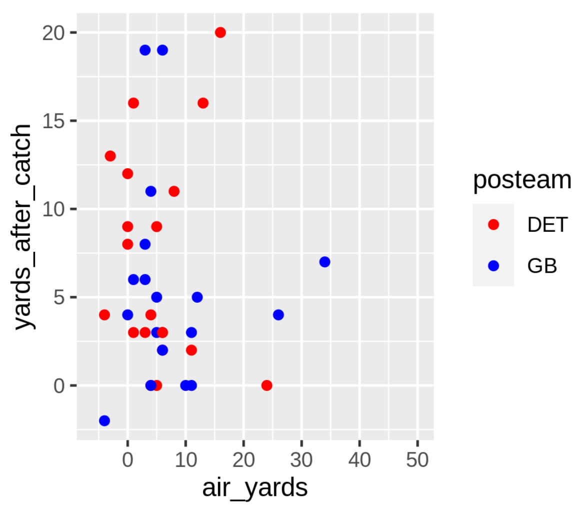

Figure 2-16. Example changing colors plot from `ggplot2`.

Plots may be customized to include many different features. For example, you might want to plot points on the plot as text. For example, you could pass location on the field (right, center, or left) as text. With `seaborn`, this requires writing a custom function, such as this StackOverflow question: *https://stackoverflow.com/q/46027653*. With `ggplot2`, we simply add `geom_text(aes(label = pass_location))`. For example, you can includes texts with this R code:

```
ggplot(data = gb_det_2020_pass,
       aes(x = air_yards, y = yards_after_catch, color = posteam)) +
    scale_color_manual(values = c("red", "blue")) +
    geom_text(aes(label = pass_location))
```

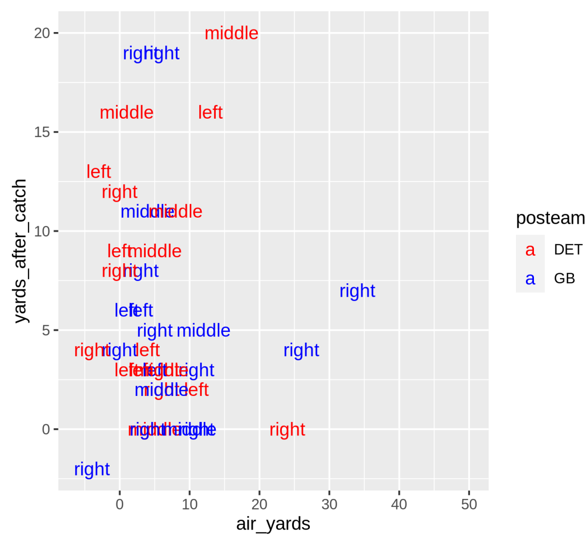

Figure 2-17. Example of text plot with `ggplot2`*.*

Application of plotting options

We can combine multiple plotting options to help tell our story. The order we apply these combinations can be important and help us tell our story. For example, if we are are interested in comparing team, pass location, and the yards gained via the pass, there are different orders for plotting. `air_yards` makes an obvious choice for the y-axis because this is the response observation. However, we could plot `pass_location` on the x-axis and facet by `posteam`. Or, we could do the opposite. We will start by plotting with `pass_location` on the x-axis.

> **NOTE**
>
> For the remainder of the book, we will often switch between only plotting R or Python, but not both unless we are teaching a new plotting tool. We encourage you to plot with your chosen language as you follow along.

With R, here is how we could create that plot. We also change the theme to be black and white with `theme_bw()` and remove the gray background from the facet grid by changing the `theme(...)`. If the `theme()` function seems seems tricky, you're not alone. Learning `strip.background = element_blank()` took Richard over a decade of using `ggplot2` to learn. We also change the x and y labels. Note that R uses the alphabetical order for ordering plots. We discuss how to change factor orders in Chapter 4.

```
ggplot(data = gb_det_2020_pass, aes(x = pass_location, y = air_yards)) +
    geom_boxplot() +
    facet_grid( ~ posteam) +
    theme_bw() +
    theme(strip.background = element_blank()) +
    xlab("Team with possession of ball") +
    ylab("Yards traveled through the air by passes")
```

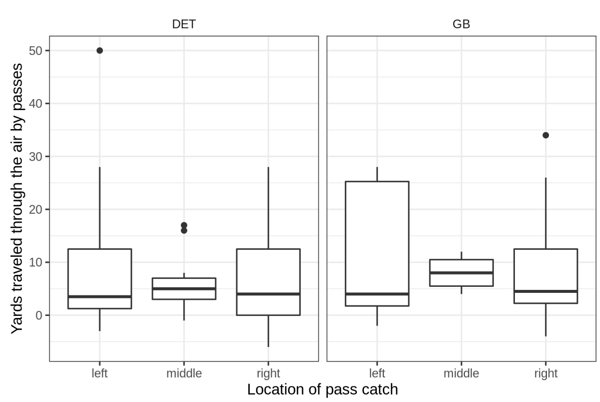

Figure 2-18. Plot of air yards with pass location on the x-axis and team as the facet.

Figure 2-18 highlights how teams air pass yards vary by location within team. However, we might also be interested in how pass yards vary between teams rather than within team. In this case, we switch the facet and a-axis for Figure 2-19. We demonstrate this plot using `seaborn`. For this plot, we also add `order` and `col_order` options to specify which order to plot variables in. If we do not include these options, `seaborn` given us a warning message.

```
yards_plot_team_location = sns.FacetGrid(data = gb_det_2020_pass,
                                         col="pass_location",
                                         col_order = ["left", "middle",
"right"])
yards_plot_team_location.map(sns.boxplot, "posteam", "air_yards", order =
["DET", "GB"])
yards_plot_team_location.set_axis_labels("Team with possession of the ball.",
                                        "Yards traveled through the air by
passes")
```

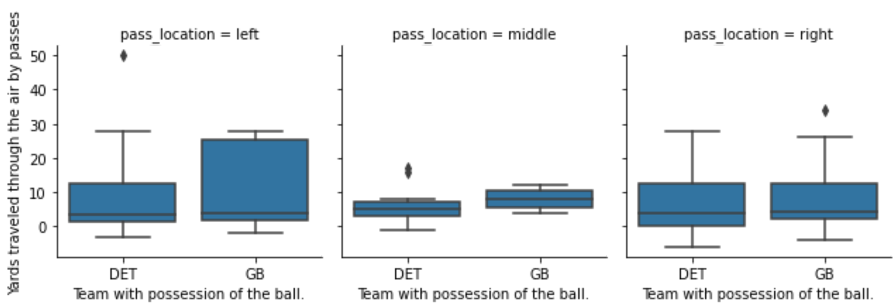

Figure 2-19. Plot of air yards with team on the x-axis and pass location as the facet.

Looking at both of these two figures. Which story jumps out to you? Which one would you use to describe the data? Neither type of plot is correct and either one might be the best choice depending upon the story you are trying to tell. Understanding and picking plots is part art and part science. The only way to get better is to do practice and get feedback.

From a football perspective, it looks pretty clear that in this game, the Packers attacked downfield in all three directions more than Detroit did, but the distributions (and likely sample sizes) were different depending on the direction of the pass. For example, it looks like both teams used the middle of the field for short passes, and the sides of the field for the down-the-field passes. For reference, Green Bay, after trailing 14-3, eventually won the game 42-21.

Advanced plotting and customization

In this chapter, we have only scratched the surface of plotting possibilities. We often start with simple boxplots or scatter plots and then start adding details. We might add in colors, shapes, facet rows, and facet columns to *see* what jumps out of plots. We may change the order multiple times and discuss with co-workers and friends to see what tells the story best. Browsing other peoples' plots is a good way to improve your own plotting.

For example, Tufte's *The Visual Display of Quantitative Information* is a classic book on how to plot. Sometimes we will browse and read this book when we get stuck with plots. Other people like other authors. Read blogs such as Fivethirtyeight.com and other quantitative blogs. Think about their figures and what works. Constructively, think about what could be improved. However, be careful with the *thinking about improvement step*. Any figure can be criticized by *arm-chair quarterbacks*, but it is much harder to actually make good figures, consistently.

Exercises with your data

1. Explore the number of bins with the example air plots. What size bins hide important parts of the data?

2. Repeat the histograms with the `air_yards` column of data.

3. Repeat the boxplots with the yards_after_catch column of data.

4. Repeat all of the plots with the Houston-Detroit data. Any differences between this game and the Detroit-Green Bay game?

5. Plot a boxplot of `` yards_after_catch`on the y-axis and `pass_location `` on the x-axis. Facet by `posteam`. What does this figure show you?

6. Repeat exercise 5, but change the facet and x-axis. You just plotted the same data. However, how does your interpretation of the data change?
7. Describe these results to a friend and explain what the plots mean.
8. Repeat all of the exercises with `det_hou_2020_pass.csv` data.

Suggested Reading

If you want to learn more about plotting, here are some resources that we found helpful:

- *The Visual Display of Quantitative Information* by Edward Tufte. *https://www.edwardtufte.com/tufte/books_vdqi*

This books is classic on how to think about data. The books does not contain code, but instead shows how to see information for data. The guidance in the book is priceless.

- `ggplot2` package documentation at *https://ggplot2.tidyverse.org/*

For our readers using R, this is the place to start to learn more about `ggplot2`. The page includes beginner resources and links to advanced resources. The page also includes examples that are great to browse.

- `seaborn` package documentation at *https://seaborn.pydata.org/*

For our readers using Python, this is the place to tart for learning more about `seaborn`. The page includes beginner resources and links to advanced resources. The page also includes examples that are great to browse. The gallery on this page is especially helpful when trying to think about how to visualize data.

- *ggplot2: elegant graphics for data analysis, Third Edition* by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen. *https://ggplot2-book.org/*

The third edition is currently under development. This book explains how `ggplot2` works in great detail but also provides a good method for thinking about plotting data using words. The easiest way to become an expert in `ggplot2` is to read this book. But, this is not necessarily an easy route.

Chapter 3. Acquiring and reading in data

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Obtaining useful data may be hardest part of data science and analytics. We saved this part until after we learned some of the basic of visualizing data for two reasons. Firstly, hopefully seeing some end products will provide inspiration to learn about working with data. Secondly, we use these tools to help check how our data looks and that we correctly obtained and read in the data.

In this chapter, we first cover in the basics of reading in data. Next, we cover how we check if data is looks okay to spot check our inputs. Lastly, we cover obtaining data because it builds upon the other skills.

Reading data into a computer

Ideally, your data comes pre-cleaned like the examples we provide in the book. Companies like Pro Football Reference (PFF) (Eric’s employer) and

Football Outsiders (a competitor) sell clean data to clients like NFL teams, betting groups, fantasy football fans, and media members. We cover obtaining unclean data from the web at the end of this chapter. Regardless of the source of your clean data, their ideal format is will be a plain text file.

> **TIP**
>
> Although commercial companies sell football data, many datasets are freely available if you know the right tools. We cover tools for this at the end of this chapter. Furthermore, you can calculate your own summary statistics (similar to what the companies sell) using tools in this book. We encourage you to work with free data. Once you reach the limits of free data, you will have the skills and knowledge to evaluate if data subscriptions are worth the money for your needs.

Plain text files mean you can open the file with a text editor, such as Notepad, and the contents are editable, readable, and make sense (specifically, the text is not gibberish symbols or strings of nonsensical characters). We often create data files using Microsoft Excel or similar spreadsheets programs and saving the outputs comma separated value (`.csv`) files. We tread carefully when editing existing data files with Excel because the program may change values in unexpected ways. For example, the scientific field of genomics has DNA data changed by Excel and impact scientists results (as described in a 2021 *Nature News* article, *https://www.nature.com/articles/d41586-021-02211-4)*.

Mechanically, you need to tell the computer where to read in files. There are two easy methods for doing this. First, you can set you working directory (the folder Python or R currently operates in) to the same folder as your data. By default, the Jupyter Lab editor sets the working directory to be the same folder as your code file. To check the default working directory in Python, use the `getcwd()` (get current working directory) function from the `os` package:

```
## Python
import os
os.getcwd("")
```

to get an output like

```
'C://Users//bob//Documents//football-analytics-with-python-and-r//book'
```

For R, we can use a similar command, `getwd()` (get working directory) that comes with base R:

```
## R
getwd("")
```

and produces a similar results as Python:

```
"C:/Users/bob/Documents/football-analytics-with-python-and-r/book"
```

You will have a different working directory than us because your computer is different.

> **WARNING**
>
> Incorrectly telling Python or R the location of your data is one of the most common problems we see learners have during our inperson programming courses. We also commonly make this mistake on a regular basis. However, when we see the error message, we know the solution. Not finding the data is like tuning into the wrong TV station on game day. Frustrating, but not the end of the world. Simply go to the correct place and everything will be okay.

Second, you can change the file path (computer direction to the data). For example, during our jobs where we have many files for projects keep data in one or more folders and code in a second folder. Likewise, you may want to access multiple data files from different folders. For this, we just add different files paths to data. Both Python and R use Linux and macOS style path names with forward slashes, such as `C:/User/me/Documents/`, rather than Windows style names with backslashes such as `C:\User\bob\Documents`.

WARNING

macOS and Linux both care about the case of paths, files, and other names. For example, `myFolder` is different from `myfolder` Windows usually, but not always does not care about the case of files. Like macOS and Linux, Python and R both care about the case of names. A common typo is using the wrong case for a name in a file or folder path.

To illustrate both example, we will describe a file path on a typical computer. Let's consider a computer with the main hard drive named `C:` (as is standard on Windows computers). There is a folder `Users` that contains all of the users' data. Unless there are multiple accounts (e.g., spouse's account, children's account) on your machine or you have a work machine managed by your employer's IT, you probably are the only user on your computer. This `User` directory is also probably your user name. For our example, we will use Bob's account name, `bob`. Bob has standard Windows folder under his user account such as `Documents`, `Desktop`, and `Downloads`. Insides Bob's documents, he probably has several folders. Perhaps he has a `Pets` folder, a `Fishing` folder, and his `football` folder. Inside his `football` folder, Bob puts the his materials for this book as `learn_code`. The file path for this folder would be `C:\User\bob\Documents\football\learn_code` and you can find this file path in Window's File Explorer (or similar program Finder on macOS). Bob avoids spaces in his name and uses the `_` instead because computers dislike spaces in file names (although computer often tolerate them). Bob like to organize files so he puts data into a `data` folder and is code into a `code` folder.

TIP

Think about how you want to organize your files for this book now. Although boring, well organize files help you find your files easier later with Python or R. Think of this like footwork drills for a wide receiver. A sometimes tedious fundamental so you do not stumble during a game.

To open a data file with a script file, you would take the following steps:

1. Make sure Miniconda is installed following the directions in Chapter 1.
2. Open the Miniconda terminal
3. Type `jupyter lab` into the terminal.

Figure 3-1. Typing `jupyter lab` into the terminal to launch the program.

4. Click on the folder icons on the left hand side until you get to the folder with your data and where you want to save your scripts.

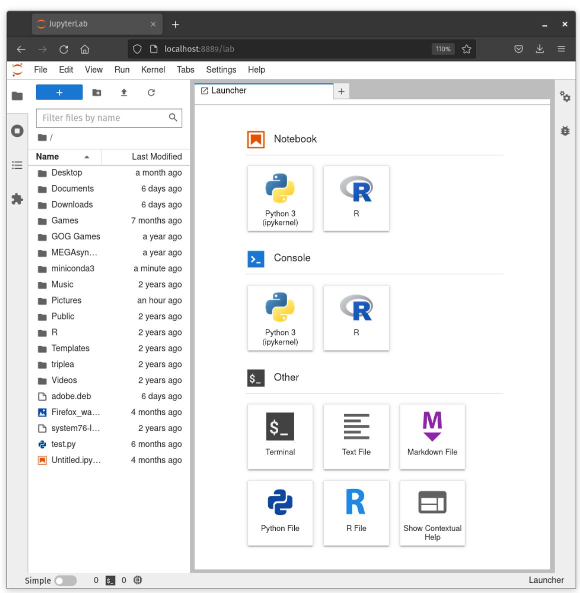

Figure 3-2. Jupyter Lab screen shot showing folders on the left.

5. Create a new Python or R script (depending upon your language choice) using the plus sign the bottom of the launcher.

6. Right click on the files name at the top of the file and click `Rename Python file` or `Rename R file` to a name you find helpful such as `learn_read_data.py` for a Python file or `learn_read_data.R` for an R file.

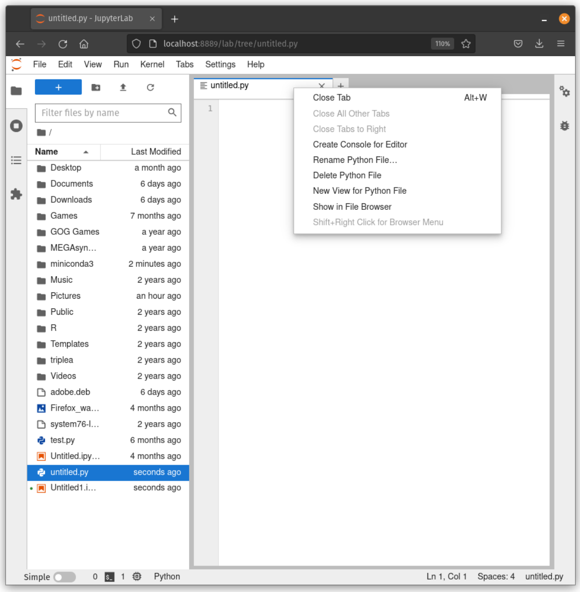

Figure 3-3. Right clicking on a file's tab.

7. Right click a second time of the file name. This time, select `Create Console for Editor`. Select the appropriate *Kernel* (that is, set of software packages) to use your script. You will want the Python option if you are using Python or the R option if you are using R. If you select the wrong kernel, you may change the kernel using the kernel drop down menu.

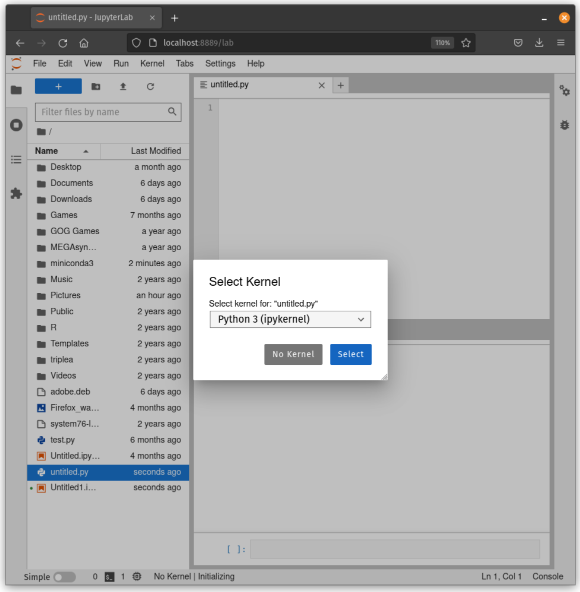

Figure 3-4. Example selecting kernel for Python script.

8. To check that everything is working, type `2 + 2` in the script file. While you are still on the line (shown by the flashing symbol |), press the Press *shift* + *enter* to run the code. Your output should be `4`. To run multiple lines at once, highlight all of the lines you want to run and press *shift* + *enter*.

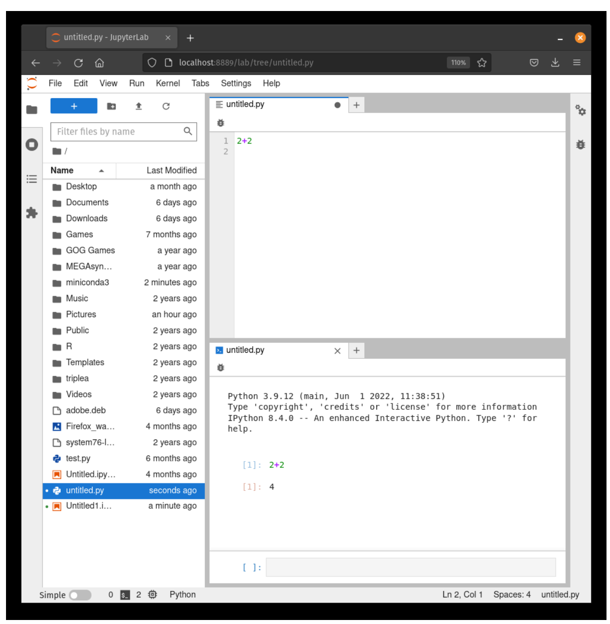

Figure 3-5. Example of running `2 + 2` from Python script.

> **TIP**
>
> Read the Jupyter Lab Overview (*https://jupyterlab.readthedocs.io/en/stable/*) to better understand this powerful IDE.

Much like footwork drills, these steps will become second nature if you do them on a regular basis. We encourage you to jot down the steps to help you remember them. Regardless, we are now ready to read in data. Chapter 2 listed these steps, but we will repeat here because the steps are important. If your file is in the same folder, load `pandas` and then read the `gb_det_2020_pass.csv` file with

```
## Python
import pandas as pd
gb_det_2020_pass = pd.read_csv("./gb_det_2020_pass.csv")
```

If you are using R, read in the file with

```
## R
gb_det_2020_pass <- read.csv("./gb_det_2020_pass.csv")
```

The file path, is the same between languages because both are based upon common Linux shell languages ([Link to Come] provides an overview of these tools in greater detail). With this name, `./` explicitly tells the computer to use the current working directory that the code file is in. This is followed by the file's name, `./gb_det_2020_pass.csv`. Also, notice both languages requires the path to be inside of quotes (`"path_to_file"`).

We can also use different paths. For example, if we have the data in a path *football_2020*, we would use the path `"./football_2020/gb_det_2020_pass.csv"`. The `./` may not always be necessary, but tells the computer to start in the current working directory. Next,the text `football_2020` tells the computer to use this folder. Last, we include the file name, `gb_det_2020_pass.csv`.

Although we could use the absolute path, such as `C:\User\bob\Documents\football\learn_code`, using relative paths from the working directory are better for multiple reasons.1 First, if you share the file or change computers, the absolute path will usually not work. Second, using a relative path with allow others to reuse your code or allows you to reuse your code in remote settings such as cloud servers. Third, absolute paths include your user name, which is a small security concern.

Relative paths start in the current working directory. For example, if we are working in the folder `learn_code` with the absolute path `C:\User\bob\Documents\football\learn_code`, `learn_code` is our file path. The current working directory is `\"./\"`, although this explicit command is often not needed. For example, `\"./path_to_csv/my_file.csv\"` usually works the same as `\"path_to_csv/my_file.csv\"`. For example, if we have folder `data` under learn code, `\"./data/gb_det_2020.csv\"` takes use to our file. To use the default home directory on your computer, start with the path `\"~/path_to_file/\"`. Usually, the default home directory on a Windows computer is the user name, for example `C:\User\bob\`. Thus, ~ would work with files and folders located in the `\"bob\"` folder. To start up a level from your current working directory, use `\"../\"`. This may be combined to move up multiple levels: `\"../../\"` goes up two folders and `\"../../../\"` goes up three levels. For example, if we were working in `C:\User\bob\Documents\football\learn_code`, `../` would take us to `football` and `../../` would take us to `Documents`.

Table 3-1. Common path settings.

Symbol	Location
./	Current working directory
	Up one level

`../`	
`~/`	Default computer home directory

WARNING

Python or R *must* know where your data lives. If you cannot load data, a common mistake is that you have not set the correct working directory. In Python load the `os` package by typing `import os` and then `os.getcwd()` will show you your current working directory and `os.listdir("./")` will show you the files in a directory. Then you may use `os.chdir("./new_file_path/folder")` to change your working directory to be inside the `new_file_path` folder and then the `folder` inside this directory. In R, `getwd()` will show you your current working directory and `list.files("./")` will show you the files in a directory and `setwd("./new_file_path/folder/")` to change your working directory like the Python example. The input options for all of these functions may be changed to a file path of your choice.

With Python, we use the `read_csv()` function from `pandas`. With R, we use the `read.csv()` function that is included as part of base R.

TIP

If `read.csv()` is too slow in R, look up the `tidyverse` function `read_csv()`. If you need faster performance, checkout the `data.table` package's function `fread()`. Both functions are much quicker than base R's `read.csv()`. We find `data.table` to be less intuitive, but much quicker. `read_csv()` will create a special type of a `data.frame` called a `tibble` and `fread()` will create a special type of `data.frame` called a `data.table`. Both of these behave like `data.frames`, but have extra features and performance benefits. For `pandas` users in Python, investigate the `Dask` package, which supports a parallel computing read option.

Besides *CSV* files, data comes in many different types of plain text files. Both `read_csv()` function in Python's `pandas` and `read.csv()` function in base R are special cases of more general read table functions. In Pandas, this is `pd.read_table()` and in R, this is `read.table()`. Both read table functions have a `sep` option for setting the deliminator between objects. For example, `sep = ","` would contain comma separated variables, `sep = "\t"` would contain tab separated files, and `sep = "\s"` would contain space separated variables. Consistent, but weird to human file structures often occur with machine or instrument generated data. For example, weather station data might be downloaded in a non-comma separated format. These formats are common in science, but more rare with football data.

> **TIP**
>
> Both Python and R have built in help functions. For example, typing `help("+")` in Python or R shows the help file for the addition operator, `+`. R also has a help shortcut with the question mark: `?"+"`. More broadly, we usually search for functions because their documentation appears online and this is easier to than the build in help files. However, sometimes in pinch (or if you are in location such as an airplane or remote cabin without internet) these basic help tools will give you the answer more readily than an search engine.

Sometimes you may want or need to read in Excel files. With the `pandas` package for Python, use `pd.read_excel()`. You may need to specify the spreadsheet you open using `sheet_name` to open a sheet other than the first spreadsheet. In R, we need to load the `readxl` library with `library(readxl)` to access the `readxl_excel()` function. The `readxl` package in R also contains an `excel_sheet()` function for accessing specific spreadsheets.

Verifying data is correct

After we read in data, we want to make sure the data is correct. Several different functions exist we use to check files. These functions differ slightly by language so, we will walk through the tools in each language. Starting with Python, we first load the Green Bay-Detroit pass data from their first game against each other in 2020. Chapter 2 also used this data.

First, we load in the data. In this case, we have have our data in a folder *data*. You may need to change the path depending upon where you have your data file located.

```
## Python
import pandas as pd
gb_det_2020_pass = pd.read_csv(".//data//gb_det_2020_pass.csv")
```

We could view all of the data by typing `print(gb_det_2020_pass)`. However, this would print all of the data and fill our screen with text. Instead, we may look at the top of the data using the `.head()` function or the bottom using `.tail()`. Remember that in Python, we call the data frame's head or tail using the functions from the object with a period. We also use an explicit `print(...)` around the head.

```
## Python
print(gb_det_2020_pass.head())
```

```
  posteam  yards_after_catch  air_yards pass_location  qb_scramble
0     DET                0.0          5        middle            0
1     DET               16.0         13          left            0
2     DET                3.0          3          left            0
3      GB               11.0          4        middle            0
4      GB                4.0          0         right            0
```

Notice how the tail data is similar to the head, but shows the bottom of the file. We like to look at the top and bottom of the data to make sure we understand it. For long files, printing the head of the data lets us peak and see if things make sense for the first few entries. Likewise, the next most common area for problems to emerge is in the bottom of the file. For example, some people include summaries in files, especially if they they created the file in Excel or similar program.

```
## Python
print(gb_det_2020_pass.tail())
```

```
   posteam  yards_after_catch  air_yards pass_location  qb_scramble
57     DET                NaN          0        middle            0
58      GB               -2.0         -4         right            0
```

```
59      DET                  20.0          16        middle             0
60      DET                   NaN          17        middle             0
61      DET                   NaN          50          left             0
```

Pandas also lets us examine the *information* about a file using `.info()`.

```
## Python
print(gb_det_2020_pass.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 62 entries, 0 to 61
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---


-
 0   posteam            62 non-null     object
 1   yards_after_catch  38 non-null     float64
 2   air_yards          62 non-null     int64
 3   pass_location      62 non-null     object
 4   qb_scramble        62 non-null     int64
dtypes: float64(1), int64(2), object(2)
memory usage: 2.5+ KB
None
```

This function prints the type of object at the type (in this case, a `pandas.core.frame.DataFrame`, the number of entries and their range, and details about the columns. If the `RangeIndex` did not make sense, the Pandas data frame has likely been edited by a function in Python and may not behave as expected. The function `.reset_index()` allows this to be reset. The first entry about columns are the column number, abbreviated using the `#` symbol. Next, is the column name, followed by the `Non-Null Count`. *Null* entries in Pandas are missing values. for example, not all plays had yards after the catch and these are null entries. Hence there are only 38 plays with catches compared to a total of 62 passing plays. Last, is the data type (or `Dtype` for short). In this case, we have *object* columns that contain characters, 64-bit floating-point numbers, and 64-bit integers. Data type is important in all data contexts, but especially football, where something like coverage type could be an integer at first blush (cover 1, cover 2) but actually be part of a group of factors, categories or characters. Integers are a special type of number that are only whole such as 0, 1, 2, or 3. After summarizing the data types, `.info()` shows the memory used by the data frame.

We can also *describe* the data in a data frame using the `.describe()` function. This shows, the number of observations as the `count` and summary statistics. Chapter 5 will cover these statistics more detail.

```
## Python
print(gb_det_2020_pass.describe())
```

```
       yards_after_catch  air_yards  qb_scramble
count          38.000000  62.000000         62.0
mean            6.263158   8.612903          0.0
std             5.912352  10.938509          0.0
min            -2.000000  -6.000000          0.0
25%             2.250000   1.250000          0.0
50%             4.000000   5.000000          0.0
75%             9.000000  12.750000          0.0
max            20.000000  50.000000          0.0
```

We can also get the dimension or *shape* of the data. This is the number of rows and columns of the data. Notice that is an attribute of the data frame and not a function. Hence, we simply add `.shape` to the end of the data frame rather than a function with parentheses.

```
## Python
## Notice this is NOT a function
gb_det_2020_pass.shape
```

```
(62, 5)
```

Likewise, we can directly view the data types by adding `.dtypes`.

```
## Python
## Notice this is NOT a function
gb_det_2020_pass.dtypes
```

```
posteam               object
yards_after_catch    float64
air_yards              int64
pass_location         object
qb_scramble            int64
dtype: object
```

The above entries methods all presented similar outputs with different themes and variations. We find those tools be useful in different situations. For example, sometimes we may only want to know the dimensions of a data frame rather than all of the information.

Next, we will show an example of how to summarize data to see the unique entries. Let's say we want to see if both possession teams passed to all three locations on the filed. We can create a list of the these two columns, `['posteam', 'pass_location']` and then use this list to only call those two columns from the `gb_det_2020_pass` data frame using square brackets. Last, we apply the function `.drop_duplicates()` to see what distinct combinations exist for the two columns. We save this to be `team_pass_loc` and then print the object to the screen to read.

```
## Python
team_pass_loc = gb_det_2020_pass[['posteam',
'pass_location']].drop_duplicates()
print(team_pass_loc)
```

```
   posteam pass_location
0      DET        middle
1      DET          left
3       GB        middle
4       GB         right
6       GB          left
10     DET         right
```

If we look at the `.info()` for this new object, the has 6 entries that go from 0 to 10 rather than 0 to 5 (remember, Python counting starts at zero). This is because the when Python dropped the duplicate values, it kept the first entry for each observation. For example, the first pass play of the game was by Detroit to the middle of the field, but they did not pass to the right side of the field until the 10th passing play of the game. If you use `.reset_index()` on this object, the index would reset to be 0 to 6.

```
print(team_pass_loc.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6 entries, 0 to 10
Data columns (total 2 columns):
 #   Column         Non-Null Count  Dtype
---


-
 0   posteam        6 non-null      object
 1   pass_location  6 non-null      object
dtypes: object(2)
memory usage: 144.0+ bytes
None
```

R has similar function for exploring data, but are different enough not all functions have direct equivalents. We start by reading in the data to R.

```
## R
gb_det_2020_pass <- read.csv("./data/gb_det_2020_pass.csv")
```

Next, we print the *head* of the data using `head()`. Notice how R uses the function on the outside of the object.

```
## R
print(head(gb_det_2020_pass))
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1     DET                 0         5        middle           0
2     DET                16        13          left           0
3     DET                 3         3          left           0
4      GB                11         4        middle           0
5      GB                 4         0         right           0
6      GB                NA         0         right           0
```

Likewise, we can use `tail()` to look at the bottom of the data.

```
## R
print(tail(gb_det_2020_pass))
```

```
   posteam yards_after_catch air_yards pass_location qb_scramble
57      GB                 6         3          left           0
58     DET                NA         0        middle           0
59      GB                -2        -4         right           0
60     DET                20        16        middle           0
61     DET                NA        17        middle           0
62     DET                NA        50          left           0
```

In R, we can view the *structure* of the data using `str()`. This tells use the type of object that `gb_det_2020_pass` is. In this case, `gb_det_2020_pass` is a `data.frame` with 62 observations and 5 variables. R indicates columns names with a `$` sign. This is because we could use this function to access the column values. For example, typing `gb_det_2020_pass$posteam` will show you all of the values for `posteam`. We also see the columns types. We have `chr` for character, that is, non-numeric values and integers, that is whole numbers. Notice how R treats the yards as integers whereas Python treats the yards as numbers.

```
## R
str(gb_det_2020_pass)
```

```
'data.frame':   62 obs. of  5 variables:
 $ posteam          : chr  "DET" "DET" "DET" "GB" ...
 $ yards_after_catch: int  0 16 3 11 4 NA NA 0 NA NA ...
 $ air_yards        : int  5 13 3 4 0 0 26 10 -2 25 ...
 $ pass_location    : chr  "middle" "left" "left" "middle" ...
 $ qb_scramble      : int  0 0 0 0 0 0 0 0 0 0 ...
```

We can get the dimension of the data using `dim()`. Although not shown, we could also use `ncol()` to get the number of columns or `nrow()` to get the number of rows.

```
## R
dim(gb_det_2020_pass)
```

```
[1] 62  5
```

R also provides a summary of the data using the `summary()` function. This provides summary statistics for integer and numerical columns. In contrast to Python, R calls missing values `NA`.

```
## R
summary(gb_det_2020_pass)
```

```
   posteam          yards_after_catch   air_yards       pass_location     
 Length:62          Min.   :-2.000    Min.   :-6.000   Length:62         
 Class :character   1st Qu.: 2.250    1st Qu.: 1.250   Class :character  
 Mode  :character   Median : 4.000    Median : 5.000   Mode  :character  
                    Mean   : 6.263    Mean   : 8.613                     
                    3rd Qu.: 9.000    3rd Qu.:12.750                     
                    Max.   :20.000    Max.   :50.000                     
                    NA's   :24                                           
  qb_scramble
 Min.   :0   
 1st Qu.:0   
 Median :0   
 Mean   :0   
 3rd Qu.:0   
 Max.   :0   
```

If we load the `tidyverse`, we may *glimpse* at the data using the `glimpse()` function. This provides a view of the head of the data as well as column information. Sometimes this format may be convenient that base R's data inspection tools.

```
library(tidyverse)
## R
glimpse(gb_det_2020_pass)
```

```
Rows: 62
Columns: 5
$ posteam           <chr> "DET", "DET", "DET", "GB", "GB", "GB", "GB", "GB",
"…
$ yards_after_catch <int> 0, 16, 3, 11, 4, NA, NA, 0, NA, NA, 3, 2, NA, 9,
NA,…
$ air_yards         <int> 5, 13, 3, 4, 0, 0, 26, 10, -2, 25, 6, 6, -1, 5, 3,
4…
$ pass_location     <chr> "middle", "left", "left", "middle", "right",
"right"…
```

```
$ qb_scramble       <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0…
```

The Tidyverse also contains a function to let us see the distinct or unique values by parameter combinations. We first use the `select()` function with the `gb_det_2020_pass` data to select the `postteam` and `pass_location` columns and then use `distinct()` function to view the unique values.

```
## R
distinct(select(.data = gb_det_2020_pass, posteam, pass_location))
```

```
  posteam pass_location
1     DET        middle
2     DET          left
3      GB        middle
4      GB         right
5      GB          left
6     DET         right
```

Common problems

In our experience, reading in data often has problems. Here, we describe some common problems and solutions.

Different separator: Sometimes people save files with confusing names. For example, somebody or something (such as an instrument) might save a space separated file with a `.csv` ending rather than a `.txt` ending. Trying to read this in with Python would give you confusing inputs because everything gets lumped into one column.

```
## Python
print(pd.read_csv("./data/space.csv"))
```

```
  col1 col2 col3
0          a 1 2
1          b 1 2
2          c 3 4
```

If we instead use the `.read_table()` function with the space plus separator option (`sep = " "`) we get three columns as we would expect.

```
## Python
print(pd.read_table("./data/space.csv", sep = " "))
```

```
  col1  col2  col3
0    a     1     2
1    b     1     2
2    c     3     4
```

In R, we also need to specify a header option as `TRUE`, otherwise R treats the first row as a row rather than column names.

```
## R
read.table("./data/space.csv", sep = " ", header = TRUE)
```

```
  col1 col2 col3
1    a    1    2
2    b    1    2
3    c    3    4
```

Multiple lines of column names/headers: Sometimes people include meta-data, that is data about data, in the header of their files. We need to tell the computer to skip these extra rows. R will give us values (as shown) whereas Python gives us a error message (not shown).

```
## R
read.csv("./data/multihead.csv")
```

```
                    Example.file.with.un.needed.header.data
Column 1 is letters
Column 2 is numbers
col1                                                   col2
A                                                         1
B                                                         2
```

Instead, we need to tell the computer how many lines to skip. In R, we do this with the `skip` option.

```
## R
read.csv("./data/multihead.csv", skip = 3)
```

```
  col1 col2
1    A    1
2    B    2
```

In contrast, we tell Python using the `skiprows` option.

```
## Python
print(pd.read_csv("./data/multihead.csv", skiprows = 3))
```

```
  col1  col2
0    A     1
1    B     2
```

Missing separator: Sometimes files will have missing separators. For example, `a 1` rather than `a,1`. If these mis-entries occur rarely due to human errors entering data, the simplest solution is probably to edit the files by hand. Conversely, if these mis-entries occur due to a computer problem such as another program mis-formatting data, then you probably need to write a custom function to clean up your data, which beyond the scope of this book. In Python, this error has an `a 2` and a missing value, `NaN` for one column.

```
## Python
print(pd.read_csv("./data/mis_sep.csv"))
```

```
  col1  col2
0  a 2   NaN
1    b   3.0
```

```
## R
print(read.csv("./data/mis_sep.csv"))
```

```
  col1 col2
1  a 2   NA
2    b    3
```

Non-numbers in number columns: Sometimes people enter non-numbers into number columns. Perhaps they entered something like `<10` (less than 10) or `1 to 2` or a typo like `1O` (one-oh) rather than `10` (one-zero). These error may be found by taking a `glimpes()` at the data in R or looking at the `.info()` of the data in Python. For example, perhaps we have a spreadsheet with three columns. Column 1 is letter, but columns 2 and 3 *should* be numbers. However, there is a typo in column 2.

For a simple, small table you might be able to see the typo. Often this is not the case, espeically for larger data frames and, if you are like us, may not notice the error until you try to use the data but get an error message. in Python, we could read in the data, save the data to an object, and then look at the information, specifically looking at the `Dtype` column:

```
## Python
wrong_number = pd.read_csv("./data/wrong_number.csv")
wrong_number.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---


-
 0   col1    2 non-null      object
 1   col2    2 non-null      object
 2   col3    2 non-null      int64
dtypes: int64(1), object(2)
memory usage: 176.0+ bytes
```

We can take similar steps in R:

```
## R
wrong_number <- read.csv("./data/wrong_number.csv")
glimpse(wrong_number)
```

```
Rows: 2
Columns: 3
$ col1 <chr> "a", "b"
```

```
$ col2 <chr> "11", "10"
$ col3 <int> 2, 44
```

Notice how the `col3` is a `int64` in Python and an `int` in R, but columns 1 and 2 the type `object` in Python and `chr` in R.

> **WARNING**
>
> Changing data *by hand* (such as using a spreadsheet) is generally considered bad practice for data science. Hand editing data can introduce errors, leaves no log or history of changes made, and is not reproducible. However, we view this like a cook tasting the cookie batter with their finger. The hand editing (and sticking the finger in the batter) may not be ideal, but is acceptable for home use, but should not be done in a production setting (or commercial kitchen) where the product is to be consumed by others.

Lastly, we have some other closing tips. We (again) encourage you to not use spreadsheet programs like Excel can change data with unpredictable results. If you have having problems editing plain text files (like `.txt` or `.csv` files), we encourage you use a plain text editor such as Notepad on Windows (or, a cross platform editor like NotePad++, available free from *https://notepad-plus-plus.org/downloads/)*. Also, we cannot cover all possible data errors. Working through data errors is can be frustrating. We have found our search engine *ninja* skills have improved through time with programming. For example, we have had our interns take a day to find a solution we find in 5 minutes simply because we have more experience searching for error terms. Remember, anything, be it football or programming, requires practice to get better.

> **TIP**
>
> One reason we suggested you use the same language as your friends in Chapter 1 is that they can lend you an extra set of eyes and hopefully help you when you run into problems such as mis-formatted data.

Obtaining data from the web

Often times there are situations where you need to scrape data off of the web. While it is beyond the scope of this book to teach you all of webscraping in Python and R, there are some pretty easy commands that can get you a significant amount of data to analyze.

Here, we are going to scrape NFL Draft and NFL Scouting Combine data from Pro Football Reference (*https://www.pro-football-reference.com/*), which, as we've mentioned before, is a wonderful resource out of Philadelphia, PA that provides free data for every sport imaginable. The NFL Draft is a yearly event held in various cities around the country. In the draft, teams select from a pool of players that have completed at least three post-high school years. While there used to be more rounds, the NFL draft currently consists of seven rounds. The draft order in each round is determined by how well each team played the year before. Weaker teams pick higher in the draft than the stronger teams. Teams can trade draft picks for other draft picks or players.

The NFL Scouting Combine is a yearly event held each year in Indianapolis, IN. In the combine, a pool of athletes eligible for the NFL Draft meet with evaluators from NFL teams to perform various physical and psychological tests. Additionally, this is generally thought of as the NFL's yearly convention.

The combination of these two data sets are a great resource for beginners in the football analytics space for a couple of reasons. Firstly, the data is collected over a small set of days once a year and is does not change thereafter. Although some players may re-test physically at a later date, and players can often leave the team that drafted them for a number of reasons, the draft teams by cannot change. Thus, once you obtain the data, it's generally good to use for almost an entire calendar year, after which you can simply add the new data when it's obtained the following year. We will scrape all one year of NFL draft data from 2022.

TIP

Web scrapping is a lot of trial and error, especially when you get started. In generally, we find an example that works and then change one piece at a time until we get something that works for us.

for loops

We are going to cover a fundamental programming skill before we start web scarping, `for loops`. Often when programming, we want to repeat code. We can use tools such as `for loops` for this task. The simplest `for loops` in an lanugage print the index of the loop. The loop takes an index, commonly the character `i`, and goes over a range of values.

In Python, this would be

```
##Python
for i in range(10):
    print(i)
```

In R, this would be

```
##R
for (i in seq(1, 10)){
    print(i)
    }
```

In this case, notice how Python's loop does not need squiggly brackets (`{}`) around the code. Instead, Python uses white space (the spaces) to show the loop. Design choices like this are one reason many people consider Python to be a more elegant language than R. For example, we could write `for (i in seq(1, 10)){ print(i) }` on one line. However, this one line of code is harder to read.

Web scrapping with Python

The following code allows us to scrape with Python. We save the Uniform Resource Locator (more commonly known as URL or web address) to an object, `url`. In this case, the URL is simply the URL for the 2022 NFL draft. Next, we use `read_html` from the `pandas` package to simply read in tables from the given URL. Remember that Python starts counting with 0. Thus, the zeroth element of the data frame, `df` from `read_html()` is simply the first table on the webpage.

```
## Python
url = "https://www.pro-football-reference.com/years/2022/draft.htm"
df = pd.read_html(url)[0]
```

We can peak at the data using `print()`.

```
print(df)
```

```
    Unnamed: 0_level_0 Unnamed: 1_level_0 Unnamed: 2_level_0  \
                   Rnd               Pick                 Tm   
0                    1                  1                JAX   
1                    1                  2                DET   
2                    1                  3                HOU   
3                    1                  4                NYJ   
4                    1                  5                NYG   
..                 ...                ...                ...   
263                  7                258                GNB   
264                  7                259                KAN   
265                  7                260                LAC   
266                  7                261                LAR   
267                  7                262                SFO   

    Unnamed: 3_level_0 Unnamed: 4_level_0 Unnamed: 5_level_0  \
                Player                Pos                Age   
0        Travon Walker                 DE                 21   
1      Aidan Hutchinson                DE                 22   
2        Derek Stingley                CB                 23   
3        Ahmad Gardner                 CB                 22   
4     Kayvon Thibodeaux                DE                 21   
..                 ...                ...                ...   
263        Samori Toure                WR                 24   
264      Nazeeh Johnson               SAF                 24   
265      Zander Horvath                RB                 23   
266           AJ Arcuri                OT                 25   
267         Brock Purdy                QB                NaN   
```

```
    Unnamed: 6_level_0 Misc    Unnamed: 9_level_0  ... Rushing      Receiving  \
                    To  AP1 PB                 St  ...     Yds   TD       Rec   
0                  NaN    0  0                  0  ...     NaN  NaN       NaN   
1                  NaN    0  0                  0  ...     NaN  NaN       NaN   
2                  NaN    0  0                  0  ...     NaN  NaN       NaN   
3                  NaN    0  0                  0  ...     NaN  NaN       NaN   
4                  NaN    0  0                  0  ...     NaN  NaN       NaN   
..                 ...  ... ..                ...  ...     ...  ...       ...   
263                NaN    0  0                  0  ...     NaN  NaN       NaN   
264                NaN    0  0                  0  ...     NaN  NaN       NaN   
265                NaN    0  0                  0  ...     NaN  NaN       NaN   
266                NaN    0  0                  0  ...     NaN  NaN       NaN   
267                NaN    0  0                  0  ...     NaN  NaN       NaN   

              Unnamed: 24_level_0 Unnamed: 25_level_0 Unnamed: 26_level_0  \
     Yds   TD                Solo                 Int                  Sk   
0    NaN  NaN                 NaN                 NaN                 NaN   
1    NaN  NaN                 NaN                 NaN                 NaN   
2    NaN  NaN                 NaN                 NaN                 NaN   
3    NaN  NaN                 NaN                 NaN                 NaN   
4    NaN  NaN                 NaN                 NaN                 NaN   
..   ...  ...                 ...                 ...                 ...   
263  NaN  NaN                 NaN                 NaN                 NaN   
264  NaN  NaN                 NaN                 NaN                 NaN   
265  NaN  NaN                 NaN                 NaN                 NaN   
266  NaN  NaN                 NaN                 NaN                 NaN   
267  NaN  NaN                 NaN                 NaN                 NaN   

    Unnamed: 27_level_0 Unnamed: 28_level_0  
           College/Univ Unnamed: 28_level_1  
0               Georgia       College Stats  
1              Michigan       College Stats  
2                   LSU       College Stats  
3            Cincinnati       College Stats  
4                Oregon       College Stats  
..                  ...                 ...  
263            Nebraska       College Stats  
264            Marshall       College Stats  
265              Purdue       College Stats  
266        Michigan St.       College Stats  
267            Iowa St.       College Stats  

[268 rows x 29 columns]
```

Although kind of ugly, this is workable! To scrape multiple years, for example 2000 to 2022, you can do a simple for loop - which is often necessary due to changes in the structure of the data - experimentation is key.

Also, when creating our own `for loops`, we often start with a simple index value (for example, set `i = 1`) and then make our code work. After making our code work, we add in the the `for ...` line to run the code over many different values.

WARNING

When setting the index value to one while building `for loops`, make sure you remove the place holder index is 1 (such as `i = 1`) from your code. Otherwise, you loop will simply run over the same functions or data multiple times.

```
## Python
df = pd.DataFrame()
for i in range(2000, 2023):
    url = 'https://www.pro-football-reference.com/years/' + str(i) +
'/draft.htm'
    temp = pd.read_html(url)[0]
    temp["Season"] = i
    df = pd.concat([df, temp])
seasons = df["Season"] #keeping season around
df.to_csv("nfl_draft_data_py.csv", index = false)
```

The tables at Pro Football Reference are a little weird in that they repeat the column names frequently throughout the table. Additionally, there is a table header that python initially interprets as the column names. The simplest way to undo this is to save the table as a csv and re-read it in. To do this, run this code:

```
##Python
df = pd.read_csv("draft_data.csv")
df["Season"] = seasons
print(df.head())
```

```
##Python
print(df.head())
```

```
  Unnamed: 0_level_0 Unnamed: 1_level_0 Unnamed: 2_level_0 Unnamed: 3_level_0  
\
0                Rnd               Pick                 Tm             Player
1                  1                  1                CLE     Courtney Brown
2                  1                  2                WAS     LaVar Arrington
3                  1                  3                WAS      Chris Samuels
4                  1                  4                CIN      Peter Warrick

  Unnamed: 4_level_0 Unnamed: 5_level_0 Unnamed: 6_level_0 Misc Misc.1  \
0                Pos                Age                 To  AP1     PB
1                 DE                 22               2005    0      0
2                 LB                 22               2006    0      3
3                  T                 23               2009    0      6
4                 WR                 23               2005    0      0

  Unnamed: 9_level_0  ... Rushing.2 Receiving Receiving.1 Receiving.2  \
0                 St  ...        TD       Rec         Yds          TD
1                  4  ...         0         0           0           0
2                  5  ...         0         0           0           0
3                  9  ...         0         0           0           0
4                  4  ...         2       275        2991          18

  Unnamed: 24_level_0 Unnamed: 25_level_0 Unnamed: 26_level_0  \
0                Solo                 Int                  Sk
1                 156                 NaN                19.0
2                 338                   3                23.5
3                 NaN                 NaN                 NaN
4                   3                 NaN                 NaN

  Unnamed: 27_level_0  Unnamed: 28_level_0  Season
0        College/Univ  Unnamed: 28_level_1     NaN
1            Penn St.        College Stats  2000.0
2            Penn St.        College Stats  2000.0
3             Alabama        College Stats  2000.0
4         Florida St.        College Stats  2000.0

[5 rows x 30 columns]
```

Now inspecting the data frame, you column names.
The current column names are not helpful. However, we may obtain the column name we want by taking the first row of the data frame and saving

this as an object `names`. We use the `iloc[0]` command to take the first (row).

```
names = df.iloc[0]
```

Now, we also remove unwanted rows such as:

```
df = df[(df["Approx Val"] != "CarAV")]
```

Lastly, we can set the column names to be the first row values we saved as `names`:

```
df.columns = names
```

Let’s look at the other columns available to us.

```
df.columns
```

```
Index([                'Rnd',                'Pick',                  'Tm',
                    'Player',                 'Pos',                 'Age',
                        'To',                 'AP1',                  'PB',
                        'St',                 'wAV',                'DrAV',
                         'G',                 'Cmp',                 'Att',
                       'Yds',                  'TD',                 'Int',
                       'Att',                 'Yds',                  'TD',
                       'Rec',                 'Yds',                  'TD',
                      'Solo',                 'Int',                  'Sk',
              'College/Univ', 'Unnamed: 28_level_1',                   nan],
      dtype='object', name=0)
```

We can also look at the head of the data.

```
print(df.head())
```

```
0  Rnd  Pick   Tm          Player  Pos  Age    To  AP1  PB  St  ...  TD  Rec
\
0  Rnd  Pick   Tm          Player  Pos  Age    To  AP1  PB  St  ...  TD  Rec
1    1     1  CLE   Courtney Brown   DE   22  2005    0   0   4  ...   0    0
2    1     2  WAS  LaVar Arrington   LB   22  2006    0   3   5  ...   0    0
3    1     3  WAS    Chris Samuels    T   23  2009    0   6   9  ...   0    0
4    1     4  CIN    Peter Warrick   WR   23  2005    0   0   4  ...   2  275
```

```
0    Yds  TD  Solo  Int    Sk  College/Univ  Unnamed: 28_level_1     NaN
0    Yds  TD  Solo  Int    Sk  College/Univ  Unnamed: 28_level_1     NaN
1      0   0   156  NaN  19.0      Penn St.        College Stats  2000.0
2      0   0   338    3  23.5      Penn St.        College Stats  2000.0
3      0   0   NaN  NaN   NaN       Alabama        College Stats  2000.0
4   2991  18     3  NaN   NaN   Florida St.        College Stats  2000.0

[5 rows x 30 columns]
```

We still have some work to do to clean the data. For example, we need to remove or `drop` the first row. We do this with the `.drop()` function. `labels = 0` tells Python to drop the first entry. `axis = 0` tells Python to drop the first row. Conversely, using `axis = 1` would tell Python to drop the first column.

> **TIP**
>
> With R and Python, we usually need to tell the computer to save our updates. Hence, we often save objects over the same same, such as `df = df.drop(labels = 0, axis = 0)`.

We next save the updated data frame, and look at the head of the data. The data now looks better!

```
df = df.drop(labels = 0, axis = 0)
print(df.head())
```

```
0 Rnd Pick   Tm           Player Pos Age    To AP1 PB St  ...  TD  Rec   Yds
\
1   1    1  CLE   Courtney Brown  DE  22  2005   0  0  4  ...   0    0     0
2   1    2  WAS  LaVar Arrington  LB  22  2006   0  3  5  ...   0    0     0
3   1    3  WAS    Chris Samuels   T  23  2009   0  6  9  ...   0    0     0
4   1    4  CIN    Peter Warrick  WR  23  2005   0  0  4  ...   2  275  2991
5   1    5  BAL      Jamal Lewis  RB  21  2009   1  1  9  ...  58  221  1879

0   TD Solo  Int    Sk College/Univ Unnamed: 28_level_1     NaN
1    0  156  NaN  19.0     Penn St.       College Stats  2000.0
2    0  338    3  23.5     Penn St.       College Stats  2000.0
3    0  NaN  NaN   NaN      Alabama       College Stats  2000.0
```

```
4  18     3  NaN    NaN  Florida St.        College Stats  2000.0
5   4   NaN  NaN    NaN    Tennessee        College Stats  2000.0

[5 rows x 30 columns]
```

We also need to change the last column name to be `season` rather than `nan`. We can use the `fillna()` function to help with this.

```
df.columns = df.columns.fillna('Season')
```

We can duplicate the first column it and give it a name to represent the data:

```
df["Selection"] = df.iloc[:, 0]
```

Lastly, lets take the data that we care about for the purposes of this analysis:

- the season in which the player was drafted (`Season`),
- which selection number they were taken at (`Selection`),
- the player's name (`Player`)
- the player's position (`Pos`)
- the player's whole career approximate value (`wAV`)
- The player's approximate value for drafting team (`DrAV`)

Now, we can see how we have cleaned up the data.

```
print(df.head())
```

```
0 Rnd Pick   Tm           Player Pos Age    To AP1 PB St  ...  Rec   Yds  TD
\
1   1    1  CLE   Courtney Brown  DE  22  2005   0  0  4  ...    0     0   0
2   1    2  WAS  LaVar Arrington  LB  22  2006   0  3  5  ...    0     0   0
3   1    3  WAS    Chris Samuels   T  23  2009   0  6  9  ...    0     0   0
4   1    4  CIN    Peter Warrick  WR  23  2005   0  0  4  ...  275  2991  18
5   1    5  BAL      Jamal Lewis  RB  21  2009   1  1  9  ...  221  1879   4

0 Solo  Int    Sk College/Univ Unnamed: 28_level_1  Season Selection
1  156  NaN  19.0     Penn St.       College Stats  2000.0         1
```

```
2  338    3  23.5      Penn St.       College Stats  2000.0          1
3  NaN  NaN    NaN       Alabama      College Stats  2000.0          1
4    3  NaN    NaN  Florida St.       College Stats  2000.0          1
5  NaN  NaN    NaN     Tennessee      College Stats  2000.0          1

[5 rows x 31 columns]
```

Lastly, we might want to re-order and select on certain columns. For example, we might only want 6 columns and also to change their order:

```
print(df[["Season", "Selection", "Player", "Pos", "wAV", "DrAV"]])
```

```
0      Season Selection          Player  Pos  wAV DrAV
1      2000.0         1   Courtney Brown   DE   27   21
2      2000.0         1  LaVar Arrington   LB   46   45
3      2000.0         1    Chris Samuels    T   63   63
4      2000.0         1    Peter Warrick   WR   27   25
5      2000.0         1      Jamal Lewis   RB   69   53
...       ...       ...              ...  ...  ...  ...
6005   2022.0         7     Samori Toure   WR  NaN  NaN
6006   2022.0         7   Nazeeh Johnson  SAF  NaN  NaN
6007   2022.0         7   Zander Horvath   RB  NaN  NaN
6008   2022.0         7        AJ Arcuri   OT  NaN  NaN
6009   2022.0         7      Brock Purdy   QB  NaN  NaN

[6009 rows x 6 columns]
```

Web-scrapping in R

We can use the `rvest` package to do a similar loop in R. You will need to install this package. To do so, type

```
## R
install.packages("rvest", repo = "https://cloud.r-project.org")
```

After the package install, we need to load the package and create an empty data frame.

```
## R
library(rvest)
df <- data.frame()
```

Then, we can loop over the years 2000 to 2023. Ranges can be specified using a colon, such as `2000:2022`. However, explicitly using using the `seq()` command because it is more robust. A key difference of the R code is that is the `html_nodes` command is called with the pipe.

```
## R
for (i in seq(from = 2000, to = 2022)) {
    url <- paste0("https://www.pro-football-reference.com/years/",
                  i,
                  "/draft.htm")
    temp <-
        read_html(url) |>
        html_nodes(xpath = '//*[@id="drafts"]') |>
        html_table()

    temp <- temp[[1]]
    colnames(temp) <- temp[1, ]
    temp$season <- i
    temp <- subset(temp, Tm != "Tm")
    temp <-
        temp |>
        as.data.frame()

    df <- rbind(df, temp)
}
write.csv(df, "nfl_draft_data_r.csv", row.names = FALSE)
```

> **NOTE**
>
> Compare the two web scrapping methods. Python functions tend to be more self-contained and call functions that belong to the object. In contract, R tends to use multiple functions on the same object. This is a design trait of the languages. Python is a more object-orientated language whereas R is a more functional language.

Notice how we save the outputs at the end. This is good practice for multiple reasons. First, it allows us to avoid re-downloading data. Second, it locks down the version of the data we use in case the website changes or crashes when we want to re-run our code.

Like the Python web-scrapping, some data cleaning is needed. We can also load our previously saved code. Lastly, we can examine the outputs using tools we learned about in Chapter 2 . For example, *Approximate Value* (AV) is PFF's way of assigning value to players. PFF, where Eric works, uses another metric, which you can see in the reference.

```
## R
library(tidyverse)
df <- read.csv("nfl_draft_data_r.csv")
df <-
    df %>%
    mutate(DrAV = ifelse(is.na(DrAV), 0, DrAV))
```

One way to evaluate a team's drafting prowess is to see how much AV they have acquired with their picks. Additionally, *Draft AV* (DrAV) is the AV earned by a player with the team that drafted him. To see how this varies with respect to draft position, we have to turn `NA` values (recall that Python uses `nan` whereas R uses `NA`) into 0, meaning that such players did not earn any value with the team that drafted them. We may plot this relationship and include a spline curve to help us see the trend:

```
## R
ggplot(df, aes(Pick, DrAV)) +
    geom_point() +
    geom_smooth() +
    theme_bw()
```

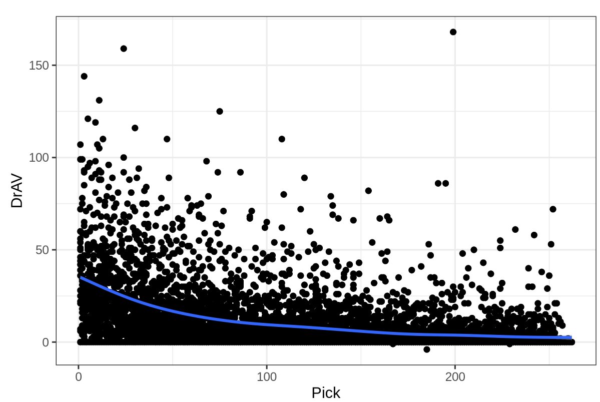

Figure 3-6. Draft pick plotted against draft approximate value (DrAV) for each player. The blue line is a spline that shows a general trend. Specifically, that lower draft picks, on average, contribute less to the DrAV.

This makes sense, as players drafted early are expected to have the highest value, but there is a lot of noise in Figure 3-6. One of the best player in the history of the NFL, Tom Brady, was taken with the 199th selection in his draft, after all.

Closing remarks on web scrapping

Now, you've seen the basics of web scrapping. What you do with this data is largely up to you! Like almost anything, the more you web scrape, the better you will become!

One tip for finding URLs is to use your web browser's (such as Chrome, Edge, or Firefox) inspection tool. This shows the html code for the webpage you are visiting. You can use this to help find which path for the table that you want. "Suggested reading" provides additional resources on web scarping.

Suggested reading

Loading data is covered in books such as

- *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data* by Garrett Grolemund and Hadley Wickham (O'Reilly), also updated at the book's homepage *https://r4ds.had.co.nz/* and
- *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (2nd edition, 3rd edition coming soon) by Wes McKinney

Grolemund and Wickham provide a through introduction to data science with R (both helped to write the Tidyverse) and McKinney created the Pandas package for Python.

Many different books and other resources exist for web scraping. Besides the package documentation for `rvest` in R and `read_html` in Pandas, two books include

- R Web Scraping Quick Start Guide by Olgun Aydin (Packet Publishing) and
- Web Scraping with Python, 2nd Edition by Ryan Mitchell (O'Reilly Media).

Exercises

1. Change the web-scraping examples to different ranges of years for the NFL draft.
2. Find you own data on the web and scrape it. An example data you can use is NFL Combine data which can be found with the URL *https://www.pro-football-reference.com/draft/YEAR-combine.htm*. Explore the relationship between variables like the 40-yard dash time and the broad jump for different position groups.
3. Import the data you found into your language and clean up your data.

Chapter 4. Data wrangling

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Sports analytics generally, and football analytics specifically, are still in their early stages of development. As such, datasets may not always be the cleanest, or *tidy*. *Tidy* datasets are usually in a table form that computers can easily read and humans understand. Furthermore, breaking ground in any field (and football analytics is no different), often requires us to adapt datasets that were created for different purposes. This is where data wrangling can come in handy. Some synonyms for *data wrangling* include include data cleanup, manipulation, mutating, shaping, tidying, and munging and describes the process of using a scripting language such as Python or R to tidy datasets to meet our needs. During the course of our careers, we have found that this task takes the most time for our projects. For example, one of our bosses once pinged us on Google chat because he was having trouble fitting a new model. His problem turned out to not be the model, but rather data formatting. Figuring out how to format the data to work with the model took about 30 minutes. However, running the new model only took about 30 seconds in R after we figured out the data.

Scripting tools like Python or R are our most-effective tools to change our data into the form in which we can be most effective. This allows us to keep track of our changes and see what we did and if we introduced any errors into our data. Many people like to use spreadsheet programs such as Microsoft Excel or Google Sheets for data manipulation. Unfortunately, these programs do not keep track of changes easily. Likewise, hand editing data does not scale, so as the size of the problem becomes too large, such as when you are working with player tracking data, you will not be able to quickly and efficiently build a workflow that works. Thus, editing one or two files by hand is easy to do with Excel, but editing one or two thousand files by hand is not easy to do. Conversely, programming languages, such as Python or R, readily scale. For example, if you have to format data after each weeks' games, Python or R could easily be used as part of a data pipeline, but spreadsheets would be difficult to automate into a data pipeline.

That being said, we understand many people like to use tools they are familiar with. If you are switching over to Python or R from using programs like Excel, we encourage you to switch one step at a time. As an analogy, think about a cook licking the batter spoon to taste the dish. When cooking at home for your family, many people do this. But, the chef at a restaurant would hopefully be fired for licking the spoon. Likewise, recreational data analysis can reasonably use program like Excel to edit data. But, professional data analysis requires the us of code to wrangle data.

> **TIP**
>
> We encourage you to start doing one step at a time in Python or R if you already use a program like Excel. For example, let's say you currently format your football data in Excel, plot the data in Excel, and then fit a linear regression model in Excel. Start by plotting your data in Python or R the next time you work with your data. Once you get the hang of that, start fitting your model in Python or R. Finally, switch to formatting data in Python or R.

Logical operators

Filtering or querying data is a fundamental skills and the basic part of filtering data is logical statement. We do this on a regular basis. For example, perhaps we want to sort a play-by-play data frame for a player or for whom we're doing an analysis. Filtering or querying data can also be hard to get the hang of. Richard remembers spending a half-a-day in grad school stuck in the computer lab trying to filter out example air quality data with R. Now, this task takes him about 30 seconds.

Both Python and R have many different methods for filtering data. We focus on the tools we use, but you will see other people use different tools if you start to read other people's code either by working with them or on the web such as tutorials or blogs. `pandas` data frames have a `.query()` function. Likewise, the `tidyverse` in R has a `filter()` function. We use these functions with logical operators.

> **NOTE**
>
> Logic operators simply refer to computer code that compares a statement and provides a binary response. In Python, logical results are either `True` or `False`. In R, logical results are either `TRUE` or `FALSE`.

Fortunately, these operators are the same across most languages, including Python and R. We will explore these operators by creating a vector in R:

```
## R
x <- c(1, 2, 3, 4)
y <- c("a", "b", "c", "a")
```

or an arrays with `numpy` in Python:

```
## Python
import numpy as np
x = np.array([1, 2, 3, 4])
y = np.array(["a", "b", "c", "a"])
```

First, we can use basic operators. Some are easy to figure out like `>` for greater than or `<`. For example, we can see which elements are greater than 2 in Python:

```
## Python
x > 2.0
```

```
array([False, False,  True,  True])
```

Likewise, we can see which elements are less than 3 in R:

```
## R
x < 3
```

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

Less than or equal to or greater than or equal to use the equals sign plus the operator. `>=` is greater than or equal to and `<=` is less than or equal to. For example, compare the next code example to the previous code example:

```
x <= 3
```

```
array([ True,  True,  True, False])
```

Other operators are less obvious. Because we already use `=` to define objects, `==` is used for equals. For example, we can find elements of `y` that are equal to `a`. Make sure you put `a` in quotes like **`"a"`**. Otherwise, the computer thinks you are trying to use an object named `a`.

```
## Python
y == "a"
```

```
array([ True, False, False,  True])
```

We can find multiple items in the same list by comparing to a list. For example, let's say we want to find which elements contain `b` or `c`. In `numpy`, we do this the `.isin()` function:

```
np.isin(y, ["b", "c"])
```

```
array([False,  True,  True, False])
```

`pandas` has a similar function for data frames we demonstrate later in this chapter.

This is really useful when there are a number of ways to chart a player playing a similar position. For example, "DE", "OLB" and "ED" mean similar things in football, and subsetting a data set for when a player is designated as any one of those labels is often something one does in analysis. R has a slightly different operator, an `%in%` function.

```
## R
y %in% c("b", "c")
```

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

When using `%in%`, be careful with the order. For example, compare `y %in% c("a", "b")` from the last example with `c("a", "b") %in% y`.

```
## R
c("b", "c") %in% y
```

```
[1] TRUE TRUE
```

> **TIP**
>
> Using `in` operators can be hard. We will often grab a test subset of our data to make sure our code works as expected. More broadly, do not trust your code until you have convinced yourself that your code works as expected!

We can also string together operators using the and operator (`&`) or the or operator (`|`). For example, we can see what entries are greater than or equal to 2 for `x` and have a `y` values of `"a"`. When working with the `numpy` arrays, we need to use the `where()` function, but this logic will be the same and use similar notation with Pandas later in this chapter. The results tells us which entry meets the criteria.

```
## Python
np.where((x >= 2) & (y == "a"))
```

```
(array([3]),)
```

We can also use an or operator for a similar comparison to see what values of `x` are greater than 2 *or* what values of `y` are equal to `"a"`.

```
## R
x > 2 | y == "a"
```

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

We can string together multiple conditions parentheses. For example, we can see what has `x` values greater than 3 *and* `y` equal to “a” *or* `x` equal to `2`.

```
## Python
np.where((x > 3) & (y == "a") | (x == 2))
```

```
(array([1, 3]),)
```

Likewise, similar notation may be used in R.

```
## R
(x > 3 & y == "a") | (x == 2)
```

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

Table 4-1. Common logical operators.

Symbol	Example	Name	Question

`==`	`x == 2`	equals	Is x equal to 2?
`>`	`x > 2`	greater than	Is x greater than 2?
`<`	`x < 2`	less than	Is x less than 2?
`>=`	`x >= 2`	greater than or equal to	Is x greater than or equal to 2?
`<=`	`x <= 2`	less than or equal to	Is x less than or equal to 2?
`|`	`(x > 2) | (y =="a")`	or	Is x less than 2 or y equal to a?
`&`	`(x > 2) & (y =="a")`	and	Is x less than 2 and y equal to a?

Filtering and sorting data

In the previous section, you learned about logical operators. These functions serve as the foundation of filtering data. In fact, when we get stuck with filtering, we often build small test cases like the ones above to make sure we understand our data and how our filters work (or, as is sometimes the case, do not work).

> **TIP**
>
> Filtering can hard. Start small and build complexity into your filtering commands. Keep adding details until you are able to solver your problem. Sometimes, you might need to do two or more smaller filters rather than one grand filter operation. This is okay. Get your code working before worrying about optimization.

We will again be working with the Green Bay-Detroit data from the second week of the 2020 season. First, we will read in the data and do a simple filter to look at plays that had yards after catch greater than 15 yards to get an idea of where some big plays were generated. In R, load the tidyverse, then our data. Next, we use the `filter()` function. The first argument into filter is data. The second argument is the filter criteria.

```
## R
library(tidyverse)
gb_det_2020_pass <- read.csv("./data/gb_det_2020_pass.csv")
filter(gb_det_2020_pass, yards_after_catch > 15)
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1     DET                16        13          left           0
2      GB                19         3         right           0
3      GB                19         6         right           0
4     DET                16         1        middle           0
5     DET                20        16        middle           0
```

> **TIP**
>
> With R and Python, you do not always need to use argument names. Instead, the languages match arguments with their predefined order. This order is listed in the help files. For example, we could type `read.csv(file = "our_file.csv")` or `read.csv("our_file.csv")`. We usually define argument names for more complex function or when we want to be clear. It is better to err on the side of being explicit and using the argument names because doing this makes your code easier to read.

Notice in this example that plays that generated a lot of yards after the catch come in many shapes and sizes, including short throws with one yard in the air, and longer throws with 16 yards in the air. We can also filter with multiple arguments using the and operator, `&`. For example, we can filter by yards after catch being greater than 15 and Detroit on offense.

```
##R
filter(gb_det_2020_pass, yards_after_catch > 15 & posteam == "DET")
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1     DET                16        13          left           0
2     DET                16         1        middle           0
3     DET                20        16        middle           0
```

However, what if we want to look at plays with yards after catch being greater than 15 yards or air yards being greater than 20 years and Detroit the offensive team? If we try `yards_after_catch > 15 | air_yards > 20 & posteam == "DET"` in the filter, we get results with both Green Bay and Detroit rather than only Detroit.

Furthermore, sometimes with R or Python, our code gets too long to fit on one line. In this case, R lets us simply do a line break either within functions or between operators. In contrast, Python, as shown in the next section, requires a special character for line breaks in code. Also, with this R code, notice how we include white space to the arguments all lineup after the `filter(`, we do this to help make our code easier to read:

```
##R
filter(gb_det_2020_pass,
       yards_after_catch > 15 | air_yards > 20 &
       posteam == "DET")
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1     DET                16        13          left           0
2      GB                19         3         right           0
3     DET                NA        28          left           0
4     DET                NA        28         right           0
5      GB                19         6         right           0
6     DET                16         1        middle           0
7     DET                 0        24         right           0
8     DET                20        16        middle           0
9     DET                NA        50          left           0
```

Instead, we get all plays with yards after catching being greater than 15 or all plays with yards greater than 20 and Detroit starting with possession of the ball. Instead, we need to add a set of parentheses to the filter: `(yards_after_catch > 15 | air_yards > 20) & posteam == "DET"`. The use of parentheses in both coding and mathematics align, so the order of operations start with the inner most set of parentheses and then move outward.

> **TIP**
>
> The *order of operations* refers to how we do math. For example, 1 + 2 * 3 = 1 + 6 = 7 and is different from (1 + 2) * 3 = 3 * 3 = 9.

```
##R
filter(gb_det_2020_pass,
       (yards_after_catch > 15 | air_yards > 20) &
       posteam == "DET")
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1     DET                16        13          left           0
2     DET                NA        28          left           0
3     DET                NA        28         right           0
4     DET                16         1        middle           0
5     DET                 0        24         right           0
6     DET                20        16        middle           0
7     DET                NA        50          left           0
```

We can also change the filter to only look at possession teams that are not Detroit using the not equal to operator, `!=`. In this case, the not equal operator gives us Green Bay's admissible offensive plays, but this would not always be the case. For example, if we were working with season long data with all teams, the not equal operator would give us data for the 31 other NFL teams.

```
##R
filter(gb_det_2020_pass,
       (yards_after_catch > 15 | air_yards > 20) &
       posteam != "DET")
```

```
  posteam yards_after_catch air_yards pass_location qb_scramble
1      GB                NA        26          left           0
2      GB                NA        25          left           0
3      GB                19         3         right           0
4      GB                NA        24         right           0
5      GB                 4        26         right           0
6      GB                NA        28          left           0
7      GB                19         6         right           0
8      GB                 7        34         right           0
```

In Python with `pandas`, filtering is done with similar logical structure as with the `tidyverse` in R, but with different syntax. First, Python uses a `.query()` function. Second, the logical operator is inside of quotes.

```
## Python
gb_det_2020_pass = pd.read_csv("./data/gb_det_2020_pass.csv")
print(gb_det_2020_pass.query("yards_after_catch > 15"))
```

```
   posteam  yards_after_catch  air_yards pass_location  qb_scramble
1      DET               16.0         13          left            0
16      GB               19.0          3         right            0
46      GB               19.0          6         right            0
52     DET               16.0          1        middle            0
59     DET               20.0         16        middle            0
```

However, the or operator, `|` works the same with both languages.

```
## Python
print(gb_det_2020_pass.query("yards_after_catch > 15 | air_yards > 20"))
```

```
   posteam  yards_after_catch  air_yards pass_location  qb_scramble
1      DET               16.0         13          left            0
6       GB                NaN         26          left            0
9       GB                NaN         25          left            0
16      GB               19.0          3         right            0
21     DET                NaN         28          left            0
22     DET                NaN         28         right            0
29      GB                NaN         24         right            0
38      GB                4.0         26         right            0
40      GB                NaN         28          left            0
46      GB               19.0          6         right            0
52     DET               16.0          1        middle            0
54     DET                0.0         24         right            0
55      GB                7.0         34         right            0
59     DET               20.0         16        middle            0
61     DET                NaN         50          left            0
```

> **WARNING**
>
> In R or Python, we can open or close with single quotes (`'`) or double quotes (`"`). When using functions such as `.query()` in Python, we see why the languages contain two different methods for quoting. We could use `"posteam == 'DET'"` or `'posteam == "DET"'`. But, we need to be consistent within the same function call.

In Python, when our code gets too long to easily read on a line we need a backslash, \ for Python to understand the line break. This is because Python treats white space as a special type of code, where as R usually treats *white space*, such as spaces, indentations, or line breaks, simply as aesthetic. To a novice, this part of Python can be frustrating, but the use of white space is actually a beautiful part of the language once one gains experience to appreciate it.

Next, we look at the use of parentheses with the or operator and and operator, just like R:

```
print(gb_det_2020_pass.query("(yards_after_catch > 15 | \
                              air_yards > 20) & \
                              posteam == 'DET'"))
```

```
   posteam  yards_after_catch  air_yards pass_location  qb_scramble
1      DET               16.0         13          left            0
21     DET                NaN         28          left            0
22     DET                NaN         28         right            0
52     DET               16.0          1        middle            0
54     DET                0.0         24         right            0
59     DET               20.0         16        middle            0
61     DET                NaN         50          left            0
```

Cleaning

Having accurate data is important for sports analytics, as the edges in sports like football can be as little as one or two percentage points over your opponents, the sportsbook, or other players in a fantasy football tournament. Cleaning data by hand using programs such as Excel can be tedious and also leaves no log of what values where changed. Also, fixing one or two systematic errors by hand can easily be done with Excel. However, fixing or reformatting thousands of cell in Excel would be difficult and time consuming. Luckily, we can use scripting to help us clean data.

> **NOTE**
>
> When estimating which team will win a game, the *edge* is the ability to predict which team will win with better odds than predicted by your opponent. For example, if you have the edge over the house in betting odds, you think an event is more likely to occur than the odds suggest. Prior to the internet, these pieces of knowledge were easier to find before sportsbooks or fantasy players updated their odds. As a concrete example, imagine the Green Bay Packers had 3-to-1 odds over the Minnesota Viking. The *odds* mean that the Packers would be expected to win 3 games for every 1 that they played against the Vikings under similar circumstances and if you bet one dollar on the the Vikings winning and they won, you would win three dollars. Conversely, if you bet three dollars on the Packers and they won, you would only win one dollar. However, if you learned that Aaron Rogers was injured before the sportsbooks could update their odds, you would have an *edge*.

We will revisit the example datasets from Chapter 3. First, we will read in the data with Python and the look at the data using the print to screen command. Notice how `col2` has a `1O` (one oh) rather than a `10` (ten).

```
wrong_number = pd.read_csv("./data/wrong_number.csv")
print(wrong_number)
```

```
  col1 col2  col3
0    a   11     2
1    b   1O    44
```

Next, we use the locate function, `.loc()` to locate the wrong cell. We also select the columns, `col2`. Finally, we replace with with a `10` (ten).

NOTE

Both R and Python allow you to access data frames using a coordinate like system with rows as the first entry and columns as the second entry. Think of this like a game of Battleship or Bingo when people call out cells like *A4* or *B2*. R still allows people to use commands like df[1, 2] to access the cells. However, it is better to use filters or explicit names. This way, if your data changes, you call the correct cell. Also, this way future you and other people will also know why you are trying to access specific cells.

```
wrong_number.loc[wrong_number.col2 == "10", "col2"] = 10
```

However, looking at the data frames information, we see that `col2` is still an object rather than an number or integer.

```
wrong_number.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---


-
 0   col1    2 non-null      object
 1   col2    2 non-null      object
 2   col3    2 non-null      int64
dtypes: int64(1), object(2)
memory usage: 176.0+ bytes
```

WARNING

Both R and Python usually require users to save data files as outputs after editing. Otherwise, the computer will not save your changes. Failure to this can cost you hours of debugging code, as we have learned from our own experiences.

We can change this by using the `to_numeric()` function from `pandas` and then look at the information for the data frame. Notice how we need to save the results to `col2` and re-write the old data. *If we skip this step, the computer will not save our edits!*

```
wrong_number["col2"] = pd.to_numeric(wrong_number["col2"])
wrong_number.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---


-
 0   col1    2 non-null      object
 1   col2    2 non-null      int64
 2   col3    2 non-null      int64
dtypes: int64(2), object(1)
memory usage: 176.0+ bytes
```

Now notice the column has been changed to an integer.

If we want to save these changes for later, we can the `to_csv()` function to save the outputs. Generally, you will want to use a new file name that makes sense to you now, other, as well as your future self. Because our data frame does not have a meaningful row names or index, we tell `pandas` to not save this information using `index = False`.

```
wrong_number.to_csv("wrong_number_corrected.csv", index = False)
```

WARNING

Wrong data types often cause problems with modeling. When debugging code we often realized we have the wrong type of data. For example, if we are building a regression model in Chapter 6 and we think a coverage scheme (for example, cover 1 or cover 2) is an actual numerical variable, then our model is going to be wrong. We've all made this mistake before.

R uses slightly different syntax. First, we use the `mutate()` function to change the column. Next, we tell R to change `col2` using `col2 = ...`. We then use the `ifelse()` function to tell R to change `col2` if it is equal to `1O` (one oh) to be `10` (one-zero or ten), else use the current value in `col2`.

```
## R
wrong_number <- read.csv("./data/wrong_number.csv")
wrong_number <- mutate(wrong_number, col2 = ifelse(col2 == "1O", 10, col2))
```

Next, just like in Python, we need to change `col2` to be numeric. In R, we use the `as.numeric()` function. Then we can look at the data frames structure using `str()`.

```
## R
wrong_number <- mutate(wrong_number, col2 = as.numeric(col2))
str(wrong_number)
```

```
'data.frame':	2 obs. of  3 variables:
 $ col1: chr  "a" "b"
 $ col2: num  11 10
 $ col3: int  2 44
```

Finally, just like in Python, we can save the file using a name that makes sense both the current us and future us. Hopefully this name names sense to other people. Creating names can be one of the most difficult parts of programming. With R, we use the `write.csv()` function. We also need to tell R to not save the row names. We do this using the `row.names = FALSE`

```
## R
write.csv("wrong_numbers_corrected.csv", row.names = FALSE)
```

> **WARNING**
>
> Python uses `False` for the logical results false and `True` for true. R uses `FALSE` for false and `TRUE` for true. If you are switching between the languages, be careful with these terms.

With programming, sometimes we want to pass outputs from one function to another without needing to save the intermediate outputs. In mathematics this is called composition, and while teaching college math classes, Eric observed this to be one of the more misunderstood procedures due to the confusing notational. In computer programming, this is called *piping* because outputs are piped from one function to another.

Luckily, R has allowed composition through the piping operators through the `tidyverse` with a pipe function, `%>%`. As of R version 4.1 released in 2021, base R alo now include a `|>` pipe operator. We use the base R pipe operator, but you may see both in *in the wild* when looking at other peoples code or websites.

NOTE

The `tidyverse` pipe allows piping to any function's input option using a `.`. This period is optional and `tidyverse`, will be default, use the first function input with piping. For example if we could do `read.csv("my_file.scv") %>% func(x = col1, data = .)` or `read.csv("my_file.scv") %>% function(col1)`. With `|>`, we can only pass to the first input, thus, we would need to define all inputs prior to the one we are piping (in this case, `data`. Thus, our code would be written as `read.csv("my_file.scv") |> func(x = col1, data = .)`

WARNING

Any reference material can become dated, especially online tutorials. The piping example demonstrates how any tutorial created before R 4.1 would not include the new piping notation. Thus, when using a tutorial, examine when the material was written and ensure you can recreate a tutorial before applying it to your problem. Lastly, when using sites such as Stack Overflow, we look at several of the top answers to make sure the accepted answer has not become outdated as languages change.

We introduce piping here for two reasons. First, you will likely see it when you start to look at other people's code as you teach yourself. Second, piping allows you to be more efficient with coding once you get the hand of it. For example, we could repeat the previous example only saving the output once:

```
## R
wrong_number <-
    read.csv("./data/wrong_number.csv") |>
    mutate(col2 = ifelse(col2 == "10", 10, col2)) |>
    mutate(col2 = as.numeric(col2))
str(wrong_number)
```

```
'data.frame':	2 obs. of  3 variables:
 $ col1: chr  "a" "b"
 $ col2: num  11 10
 $ col3: int  2 44
```

Checking and cleaning data for outliers

Data often contains errors. Perhaps people collecting or entering the data made a mistake. Or, maybe an instrument like a weather station malfunctioned. Sometimes, computer systems corrupt or otherwise change files. In football, quite often there will be errors in things like number of air yards generated, yards after the catch earned, or even the player targeted. Resolving these errors quickly, and often through data wrangling, is a required process of learning more about the game. Chapter 2 presented tools to help you catch these errors.

We have included a file with an outlier entered in to it. We'll go through how to find and remove this outlier with both languages.

In R, we first read in the data and then look at the summary of the data.

```
## R
pass_outlier <-
    read.csv("./data/gb_det_2020_pass_outlier.csv")
pass_outlier |>
    summary(.)
```

```
   posteam          yards_after_catch   air_yards       pass_location
 Length:62          Min.   :-2.000    Min.   :  -6.00   Length:62
 Class :character   1st Qu.: 2.250    1st Qu.:   1.25   Class :character
 Mode  :character   Median : 4.000    Median :   5.00   Mode  :character
                    Mean   : 6.263    Mean   :  88.45
                    3rd Qu.: 9.000    3rd Qu.:  12.75
                    Max.   :20.000    Max.   :5000.00
```

```
                    NA's   :24
 qb_scramble
Min.   :0
1st Qu.:0
Median :0
Mean   :0
3rd Qu.:0
Max.   :0
```

We can get similar results with Python using `.describe()`.

```
## Python
pass_outlier = \
    pd.read_csv("./data/gb_det_2020_pass_outlier.csv")
pass_outlier.describe()
```

	yards_after_catch	air_yards	qb_scramble
count	38.000000	62.000000	62.0
mean	6.263158	88.451613	0.0
std	5.912352	634.064812	0.0
min	-2.000000	-6.000000	0.0
25%	2.250000	1.250000	0.0
50%	4.000000	5.000000	0.0
75%	9.000000	12.750000	0.0
max	20.000000	5000.000000	0.0

Looking at the summaries, the maximum value for one column seems a bit off for `air_yards`. We can also see this with a histogram. If you need help with the syntax for the histogram, Chapter 2 provides directions. We include an example histogram from R, but the Python histogram would should similar results.

```
## R
pass_outlier |>
    ggplot(data = ., aes(x = air_yards)) +
    geom_histogram()
```

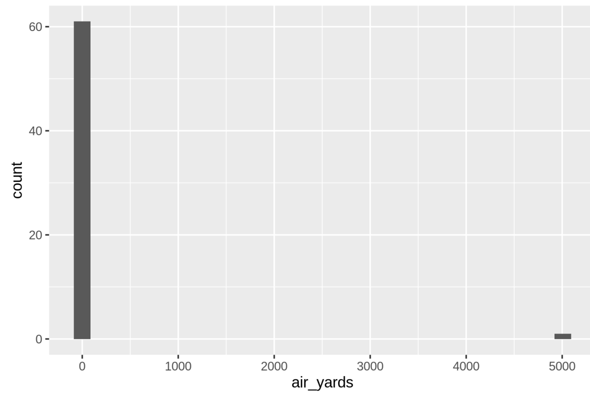

Figure 4-1. Histogram of air yards showing an outiler.

The air yards value of 5,000 yards does not seem correct. In fact, this would be impossible for a single play. What should we do? We have two reasonable choices. First, we can remove the value because it is obviously wrong. With R, we can filter the data and create a second data frame. We filter by 109 yards because this is the theoretical maximum from throwing the ball from the one yard line to the back of the opposing end zone. Looking at the summary, we see this value is now gone.

```
## R
no_pass_outlier_1 <-
    pass_outlier |>
    filter(air_yards < 109)
no_pass_outlier_1 |>
    summary(.)
```

```
   posteam          yards_after_catch   air_yards       pass_location
 Length:61          Min.   :-2.000    Min.   :-6.000   Length:61
 Class :character   1st Qu.: 2.250    1st Qu.: 1.000   Class :character
 Mode  :character   Median : 4.000    Median : 5.000   Mode  :character
                    Mean   : 6.263    Mean   : 7.934
                    3rd Qu.: 9.000    3rd Qu.:12.000
                    Max.   :20.000    Max.   :34.000
                    NA's   :23
  qb_scramble
 Min.   :0
```

```
 1st Qu.:0
 Median :0
 Mean   :0
 3rd Qu.:0
 Max.   :0
```

Likewise, we may do a `query()` with Pandas.

```
pass_outlier = pd.read_csv("./data/gb_det_2020_pass_outlier.csv")
no_pass_outlier_1 = pass_outlier.query("air_yards < 109")
print(no_pass_outlier_1.describe())
```

```
       yards_after_catch  air_yards  qb_scramble
count          38.000000  61.000000         61.0
mean            6.263158   7.934426          0.0
std             5.912352   9.624394          0.0
min            -2.000000  -6.000000          0.0
25%             2.250000   1.000000          0.0
50%             4.000000   5.000000          0.0
75%             9.000000  12.000000          0.0
max            20.000000  34.000000          0.0
```

A second option would be to replace the value. Perhaps we think 5,000 just has two extra zeros and should be 50. With R, we can use `mutate()` with `ifelse()` to change this single value.

```
## R
no_pass_outlier_2 <-
    pass_outlier |>
    mutate(air_yards = ifelse(air_yards == 5000, 50, air_yards))
no_pass_outlier_2 |>
    summary(.)
```

```
   posteam          yards_after_catch   air_yards      pass_location
 Length:62          Min.   :-2.000    Min.   :-6.000   Length:62
 Class :character   1st Qu.: 2.250    1st Qu.: 1.250   Class :character
 Mode  :character   Median : 4.000    Median : 5.000   Mode  :character
                    Mean   : 6.263    Mean   : 8.613
                    3rd Qu.: 9.000    3rd Qu.:12.750
                    Max.   :20.000    Max.   :50.000
                    NA's   :24
  qb_scramble
 Min.   :0
 1st Qu.:0
```

```
Median :0
Mean   :0
3rd Qu.:0
Max.   :0
```

With Python, we first copy the original data to avoid changing it. Then, we use the `.loc[]` function to find the wrong value and change it to be 50. Notice, the results now match R.

```
no_pass_outlier_2 = pass_outlier.copy()
no_pass_outlier_2.loc[no_pass_outlier_2.air_yards == 5000.0, "air_yards"] =
50.0
print(no_pass_outlier_2.describe())
```

```
       yards_after_catch  air_yards  qb_scramble
count          38.000000  62.000000         62.0
mean            6.263158   8.612903          0.0
std             5.912352  10.938509          0.0
min            -2.000000  -6.000000          0.0
25%             2.250000   1.250000          0.0
50%             4.000000   5.000000          0.0
75%             9.000000  12.750000          0.0
max            20.000000  50.000000          0.0
```

Merging multiple datasets

Sometimes we need combine datasets. For example, often you want to adjust the results of a play - say the number of passing yards - by the weather in which the game was played. Both `pandas` and the `tidyverse` readily allow merging datasets. For example, perhaps we have team and and game data we want to merge. Or, maybe weather data to each game.

For this example, we will create two data frames and then merge them together. Once data frame will be city information that contains the teams' names and city. The other will be a schedule. We create a small example for multiple reasons. First, a small toy dataset is easier to handle and see compared to a large dataset. Second, we often create toy datasets to make sure our merges work.

TIP

When learning some new, start with a small example you understand. The small example will be easier to debug and fail faster and easier than a large example or actual dataset.

You might be wondering, why merge these data frames? We often have to do merges like this when summarizing data because we want or need a prettier name. Likewise, we often need to change names for plots. Next, you might be wondering, why not hand type these values into a spreadsheet? Hand typing can be tedious and error prone. Plus, doing tens, hundereds, or even thousands of games would take a long type to hand type.

As you create the data frames in R, remember that each column you create is a vector.

```
## R
library(tidyverse)
city_data <- data.frame(city = c("DET", "GB", "HOU"),
                        team = c("Lions", "Packers", "Texans"))
schedule <- data.frame(home = c("GB", "DET"),
                       away = c("DET", "HOU"))
```

As you create the data frames in Python, remember that the `DataFrame()` uses a dictionary to create columns and elements in the columns.

```
## Python
import pandas as pd
city_data = pd.DataFrame({"city" : ["DET", "GB", "HOU"],
                          "team" : ["Lions", "Packers", "Texans"]})
schedule = pd.DataFrame({"home" : ["GB", "DET"],
                         "away" : ["DET", "HOU"]})
```

Now, that we have the data sets, we can use them to explore different merges. Both `pandas` and the `tidyverse` base their merge functions upon SQL. The joins requires common, shared key or keys between the two data frames. In the `tidyverse`, this argument is called *by*, for example joining city and schedule data frames by *team name* and *home* team columns. In `pandas`, this argument is called *on*, for example joining city and schedule data frames on *team name* and *home* team columns.

There are four main joins we use on a regular basis and these are included with the `tidyverse` and `pandas`. `pandas` has both a `merge()` and `join()` function. `merge()` contains almost everything as `join()` plus some more so we will only include `merge()` here. With both Python and R, there are two datasets, a left data and a right datasets. The left dataset is the one on the left (or the first datasets) and the right dataset is the one on the right (or the second dataset).

For our example, we want to create a new dataframe that includes both schedule and the teams' names. We will use this to explore the different types of joins. Think of this example like the fairy tale of Goldilocks and the four joins (rather than three bears). Rather than a girl trying bears, beds and food, we'll be exploring data joins. This problem actually has two steps. The first step is to add in the home team's name. The second step is to add in the away team's name. At the end, will show you the complete workflow because it also involves renaming columns.

TIP

Football analytics, like the broader field of data science, usually involves breaking big jobs down into smaller jobs. As you become more experienced, you will become better at seeing the small steps and knowing where and how to re-use them. When faced with intimidating problems, we break them down into smaller steps that we can readily solve.

First, we will examine a *full* or *outer join*. This merges both data frames based upon all values in both data frames' keys. If one or both keys contain values not found in the other dataset, these are replaced by missing values (`NA` in R, `NaN` in Python). For both languages, `schedule` will be our left data frame and `city_data` will be our right data frame. Because both data frames do not have the key, we need to tell the computer how to pair up the keys.

In R, we use the `full_join()` function. We, put `schedule` in first, followed by `city_data`. We tell R to *join* the data frames using `home` as the left key matching up with `city` as the right key.

```
## R
print(full_join(schedule, city_data, by = c("home" = "city")))
```

```
  home away    team
1   GB  DET Packers
2  DET  HOU   Lions
3  HOU <NA>  Texans
```

Notice how we get three entries because the `city_data` has three rows. The missing value is replaced by `NA`. Notice how R dropped the duplicate column and only has three columns.

In Python, we use the `.merge()` function on the `schedule` data frame. Notice that `schedule` is on the left. The fist argument is `city_data`. We tell Pandas to how to merge, specifically and `outer` merge. We then tell Pandas to use `home` as the left key and `city` as the right key.

```
print(schedule.merge(city_data, how = "outer",
                     left_on = "home", right_on = "city"))
```

```
  home away city     team
0   GB  DET   GB  Packers
1  DET  HOU  DET    Lions
2  NaN  NaN  HOU   Texans
```

Notice Pandas kept all four columns. Also, notice how both `home` and `away` are `NaN` for the new data frame.

> **NOTE**
>
> This example demonstrates how Python tends to be an object-orientated language and R tends to be functional language. Python uses `.merge()` as an object contained by the data frame `schedule`. R uses a `full_join()` as a function on two different objects, `schedule` and `city_data`. Although R and Python both contain object-orientated and functional features, this example nicely demonstrates the underlying philosophy of the two languages.
>
> Think of this distinction of language types similar to how some football teams are build for a run offense and others as a pass offense. Under certain circumstances one language can be better than the other, but usually both contain the tools for given job. Advanced data scientists recognize these trait off between languages and will switch languages to fit their needs.

Next, we will do an *inner join*. This only joins the shared key values. Whereas an outer join may possibly grow data frames, an inner join shrinks data frames. The R syntax is very similar to the previous example, only the function name changes. However, notice how the output only has three values.

```
## R
print(inner_join(schedule, city_data, by = c("home" = "city")))
```

```
  home away    team
1   GB  DET Packers
2  DET  HOU   Lions
```

Like R, the Python code is similar. In Python, we use the same function, but a different `how` argument.

```
## Python
print(schedule.merge(city_data, how = "inner",
                     left_on = "home", right_on = "city"))
```

```
  home away city     team
0   GB  DET   GB  Packers
1  DET  HOU  DET    Lions
```

Next, we will do a *right join*. The right join keeps all of the values from the right data frame. For this specific case, the outputs are the same as the outer join. This is an artifact of our example and may not always be the case. With R, we just change the function name to be `right_join()`.

```
## R
print(right_join(schedule, city_data, by = c("home" = "city")))
```

```
  home away    team
1   GB  DET Packers
2  DET  HOU   Lions
3  HOU <NA>  Texans
```

With Python, we change the `how` to be `right`.

```
## Python
print(schedule.merge(city_data, how = "right",
                     left_on = "home", right_on = "city"))
```

```
  home away city     team
0  DET  HOU  DET    Lions
1   GB  DET   GB  Packers
2  NaN  NaN  HOU   Texans
```

A *left join* is the opposite of a right join. This keeps all of the values from the left data frame. In fact, rather than switching the function, one could switch the order of inputs. Consider merging data frames `A` and `B` in Python that share a common column, `key`.

```
A.merge(B, how = "left", on = "key")
```

This could also be written in reverse.

```
B.merge(A, how = "right", on = "key")
```

Here is what the R code and output looks like.

```
## R
print(left_join(schedule, city_data, by = c("home" = "city")))
```

```
  home away    team
1   GB  DET Packers
2  DET  HOU   Lions
```

The Python code also looks similar to the right join. For both of the outputs, the left join was the same as the inner join. This is an artifact our example choice and will not always be the case. Here, the left data frame had fewer rows than the right data frame. Hence, this occurred in the example.

```
print(schedule.merge(city_data, how = "left",
                     left_on = "home", right_on = "city"))
```

```
  home away city     team
0   GB  DET   GB  Packers
1  DET  HOU  DET    Lions
```

Table 4-2. Common joint types in R and Pyt

h
o
n
.

Name	Brief description	Tidyverse function	Pandas merge how
Full/outer join	Merges based upon all key values	`full_join(left_data, right_data)`	`left_data.merge(right_data, how = "outer"`
Inner join	Only merges based upon shared key values	`inner_join(left_data, right_data)`	`left_data.merge(right_data, how = "inner"`
Left join	Only merges based upon *left* data's key values	`left_join(left_data, right_data)`	`left_data.merge(right_data, how = "left"`
Right join	Only merges based upon *right* data's key values	`right_join(left_data, right_data)`	`left_data.merge(right_data, how = "right"`

Returning to our initial problem *How do we merge the data frame to include the team names for both the home and away teams?*

Multiple solutions exist, as is often the case with programming. We use multiple left joins because we think about adding data to schedule and putting this data frame on the left. However, you might think about the problem differently, which is okay. In fact, you might be think about a better way to do this that is either quicker, easier to read, or uses less code!

> **NOTE**
>
> Unlike high school math, both statistics and coding often have no single best or right way to do something. Instead, many unique solutions exist. Some people play a game called "golf coding" where they try to solve a problem using the fewest lines of code. But, the fewest lines of code is usually not the best answer in real life. Instead, focus on writing code you and other people can read later.

So, we will use a series of left joins (although, we could also do everything in reverse using right joins). Here is our step-by-step solutions:

1. Merge in for home team
2. Rename column in R, rename and delete column in Python
3. Merge in for away team. Needed for clarity and avoid duplicate names.
4. Rename column in R, rename and delete column in Python.
5. Make sure output is saved to new data frame, `schedule_name`.

Some notes about how and why we use these specific steps. Whether we merged by the away or home order is not important and we arbitrarily selected order. We needed to rename columns to avoid duplicate names later and also keep column names clear. The importance of this will become important when you have to cleanup your own mess or somebody else's messy code! Lastly, we encourage you to start with one line of code and keep adding more code until you understand the big picture. That's how we constructed this example.

With the R example, we use piping to avoid re-writing objects like we did for the Python example. First, we take the `schedule` data frame and then left join to the `city_data`. We tell R to join by (or match) the `home` column to the `city` column. We then rename the `team` column to be the `home_team` column. This helps us keep the team columns straight in the final data frame. We then repeat these steps and join the away team data.

```
%%R
## R
schedule_name <-
    schedule |>
    left_join(city_data, by = c("home" = "city"))  |>
    rename(home_team = team)  |>
    left_join(city_data, by = c("away" = "city"))  |>
    rename(away_team = team)
print(schedule_name)
```

```
  home away home_team away_team
1   GB  DET   Packers     Lions
2  DET  HOU     Lions    Texans
```

With Python we create temporary objects rather than piping. This is because Pandas's piping is not as intuitive to us and requires writing custom functions. Furthermore, some people like writing out code to see all of the steps and we want to show you a second method for this example. With Python, we first do a left merge. We tell Python we use `home` for the left merge on and `city` for the right merge on. We then need to rename the `team` column to be `home_team`. The Pandas rename function requires a dictionary as a input. Then, we tell Pandas to remove (or `.drop()`) the city column to avoid confusion later. We then repeat these steps for the away team.

```
## Python
step_1 = schedule.merge(city_data, how = "left",
                        left_on = "home", right_on = "city")
step_2 = step_1.rename(columns =
                       {"team": "home_team"}).drop(columns = "city")
step_3 = step_2.merge(city_data, how = "left",
                      left_on = "away", right_on = "city")
schedule_name = step_3.rename(columns =
                              {"team": "home_team"}).drop(columns = "city")
print(schedule_name)
```

```
  home away home_team home_team
0   GB  DET   Packers     Lions
1  DET  HOU     Lions    Texans
```

Exercises

1. Examine short plays by sorting yards after catch to be less than 10 and air yards to be less than 5.
2. Repeat the previous filter, but also group by each team.
3. Use your skills from Chapter 2 to plot the results from the previous step.

4. Convince yourself that right and left joins are the same, but in reverse.
5. Find a different way to join the schedules without using a left join.

Suggested reading

To become an expert on these topics, use them on a regular basis and find new methods to get started. Additionally, we found these resources to be helpful:

- *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse* by Chester Ismay and Albert Y. Kim (CRC Press), also updated at the book's homepage *https://moderndive.com/*

The book contains sections for complete beginners to learn the Tidyverse and is a great place to start learning the tidyverse.

- *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data* by Garrett Grolemund and Hadley Wickham (O'Reilly), also updated at the book's homepage *https://r4ds.had.co.nz/*

Grolemund and Wickham provide an in depth explanation of many different methods for data wrangling with chapters expanding upon topics we briefly describe in this chapter. This book is deeper, but less accessible than the Ismay and Kim book previously mentioned. Also, Wickham created the Tidvyerse, starting with ggplot2 as part of his doctoral thesis at Iowa State University.

- *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (2nd edition, 3rd edition coming soon) by Wes McKinney

McKinney is the author of the Pandas package and provides an in depth and accessible explanation for data wrangling in Python.

- *Advancing into Analytics: From Excel to Python and R* by George Mount (O'Reilly Media).

Mount helps current Excel users learn how to use Python and R as well as some advanced features of Excel. For current Excel users who want to learn more programming in Python, R, or both, we suggest they checkout this book.

The package's homepages also provide excelled documentation on many additional features of the functions we use.

- For the `tidyverse`, visit *https://tidyverse.org/*
- For `pandas`, visit *https://pandas.pydata.org/docs/* and checkout the getting starting guide

Chapter 5. Summary Statistics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

The word *statistics* means different things to different people. During our day jobs, we see three uses of the word. Commonly, people use the word to refer to data. For example, we might talk about the numbers from a game or a player’s performance as the game’s statistics or player’s stats. More formally, *statistics* can refer to the systematic collection and analysis of data as well as the corresponding field of study. For example, you might have taken a statistics course in high school or somebody works as a statistician. Lastly, a *statistic* can be something that is estimated, like expected points added per play, or completion percentage above expected by a quarterback or offense.

This chapter focuses on the last definition. We describe how to summarize data using statistics. For example, rather than needing to read the play-by-play report for a game, we can get an understanding of what occurred by looking at the summary statistics from the game. We use summary statistics on a daily basis to help us understand data and also help other to see the story held within the data. We also use these summary statistics to lay a foundation for modeling methods that we cover in future chapters.

Basic statistics

Averages

Perhaps the simplest statistic is the *average*, or what goes on goes on in typical observation from a dataset. In fact, some mathematically minded people call the average the *expectation* for a dataset because it is what they expect to see in the data. Commonly, when we talk about the *average* for a dataset, we are talking about a the central tendency of the data, or, where is the “middle” of the data. More formally, three common methods exist for estimating averages. Typically, when people say average, they are using the definition for a *mean*. In fact, we use the word *average* in this book, we are talking about the mean unless we clearly state otherwise. However, average can also refer to *median* and *model*. We show how to calculate these by hand in the next section.

We intentionally do not include code for this section. Furthermore, we hope you do this hand, just this one time to better learn and understand the ways to calculate these statistics. And yes, computers are much, much better at calculating than people and computers also make fewer mistakes than people. We will show you how use Python and R to calculate these later in the chapter. However, doing the calculations once by hand will help you learn the concepts better.

First, let’s calculate a *mean* by hand. We will use the air yards from passes to the middle of the field by Detroit from their first game against Green Bay in 2020. This is the same dataset we previously used. The air yards are: $5, -1, 5, 8, 5, 6, 1, 0, 16, \text{and }\ 17$. To calculate the mean, we first add up all of the numbers (a mathematical operation also called taking the sum):

$$5 + -1 + 5 + 8 + 5 + 6 + 1 + 0 + 16 + 17 = 68.$$

Next, we divide by the total number plays with air yards:

$$68/11 = 6.18.$$

This allows us to estimate the mean air yards to the middle of the field to be 6.18 yards for Detroit during their first game against Green Bay in 2020. Also, we rounded the output to be 6.18. We rounded because the resulting mean does not end and we only truly know the first two digits, but include the last digit to capture uncertainty. More formally, this is known as the number of significant digits or figures.

> **TIP**
>
> Significant digits are important when reporting results. Although formal rules exist, a rule of thumb that works most of the time is to simply report the number of digits you have confidence in the result.

Another way to estimate an average or typical outcome is to examine the *median*. The median is simply the middle number, or the value of the average individual (rather than the average value). One way to think about the median is that it's the value earned by the average individual (whereas the mean is the average value earned).
The calculate the median, we write the numbers in order for smallest to largest and then find the middle number:

$$-1,\ 0,\ 1,\ 5,\ 5,\ \mathbf{5},\ 6,\ 6,\ 8,\ 16,\ 17$$

.

Because we have 11 numbers, 5 is the middle number. If we have a tie when we have an even number of numbers, then we take the mean of the two middle numbers. For example, if we have 4 numbers

$$-1,\ 0,\ 1,\ 5$$

then $\frac{(0+1)}{2} = 0.5$ is the median.

The last method to estimate an average number is to examine the *mode*. The mode is the most common outcome. To calculate the mode, we need to create a table with counts and air yards.

With this example, 5 is the mode because there were 3 observations with 5. Data can be multi-modal, that is to say, have multiple modes. For example, if two outcomes have the same number of occurrences, then the a bimodal outcome occurred. Modes also allow us to estimate the *average* for categories. For example, we could count the number of passing plays to either the middle, left, or right sides of the field to calculate the model.

Lastly, we want to note that different types of means exist. Other than this references, we do not include them elsewhere in the book. However, you may run into them in the future if you keep learning as a football analyst. Examples of different means include arithmetic, geometric, harmonic, and power means. We only us arithmetic mean, but if you take the dive into advanced statistics books you may see these terms. For example, we use geometric means at work when dealing with environmental chemistry data. This is more robust to outliers, but also harder to explain and work with. Hence we stick to the arithmetic mean.

To see the three different types of averages, let's examine a for all pass locations for both teams from the 2020 Green Bay and Detroit's first game. This sub-set of the data is more interesting to examine, but would have been harder to use *by hand*. First, notice the blue line that is the mean. The mean is to the right of the median, which means the data is skewed or has outliers to the right. Second, the median is the same as the model.

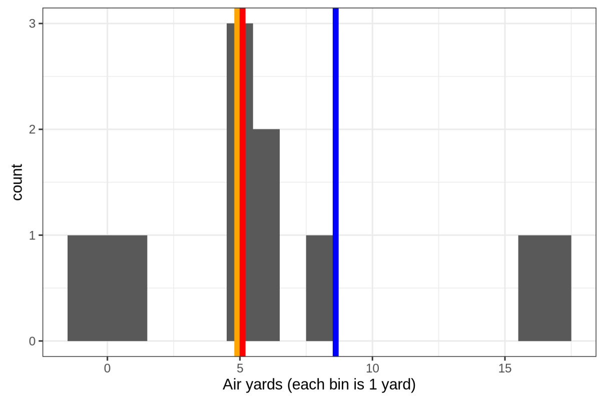

Figure 5-1. Histogram of yards after catch. The blue line shows the mean, the red line shows the median, and the orange lines shows both modes. Notice each bin is 1 yard in width. The median and mode are offset to allow each to be seen.

So, what does this tell us about Detroit's passing game to the middle of the field? First, the difference between the median and mean shows us that most plays are short, but one really long play differs from the rest and hence skews our perception if we only look at the mean. If we were analyzing the game, we would want to see if what impact this play had on the game. Second, the mode and median show us that most plays to the center of the field did not have many yards for Detroit. The histogram shows these observations nicely. In fact, the histogram would probably tell as better story than the summary statistics for this simple case. However, when dealing with more data, not always able to include plots or need numbers to quickly capture what the figure shows. Lastly, we will use these concepts to introduce more complex ideas as well.

Range and distribution

The previous section shows how to examine the middle or central tendency of the data. However, we can also be interested in how much variability exists in the data. This is the distribution of the data. One of the simplest methods for examining a distribution may be the range.

Using the previously constructed table (or histogram), we can calculate the *range*. The range is simply difference between the minimum and maximum value. The minimum (commonly abbreviated *min*) or lowest value is -1 and the maximum (commonly abbreviated *max*) or highest value is 17. Range goes from -1 to 17 yards and range width is 18 yards. Recall that subtracting a negative number is the same as addition, so $17 - -1 = 17 + 1 = 18$.

Another methods to examine the range and distribution of a dataset is examine the *quantiles*. These focus on a specific parts of the distribution. For example, the 50th quantile is the median. Chapter 2 covered quantiles in the section of this book on boxplots. At the core, boxplots allow us to easily see the distribution of data.

Recall that boxplots show us where the middle 50% of the data occurs. Sometimes, other types of quantiles may be used as well. The benefit of quantile are that they allow to estimate end points other than the central tenancy. For example, the mean or median allows us to examine how well average players do, but a quantile allows us to examine how well the best player do (for example, what does a player need to be better than 95% of other players?). [Link to Come] covers methods for modeling quantiles.

Another methods for examining the distribution is to look at the *variance* and *standard deviation*. Variance examines how far apart each observation is from the mean. Because some values will be negative, the variance is squared and then summed to be calculated. Using the Detroit middle of field example air yards example, we can do this calculation by hand. We include a column mean in case you are doing this calculations in a spreadsheet such as Excel and to also help you see where the numbers come from.

After we create this table, we then take the sum of the difference squared column, which is 337.6. We divide 10 because this is the number of observations minus 1. We subtract one to reflect the degrees of freedom, or number of unused data points we have. Because we are using one data point to calculate the mean, are degrees of freedom drops by one.

Calculating $\frac{337.6}{(11-1)}$, we get the variance, 33.7. The units for variance with this example would be yards $\times$ yards or yards^2. This unit does not help us very much, so we can take the square root to get the standard deviation: 5.81. The unit for this example standard deviations are now yards.

Both the variance and standard deviation allow us to easily see how much variability exists in the data. People more commonly use the standard deviation because the units are easier to understand. Also, the standard deviation can be more helpful when comparing many different variables. Lastly, we can use use the standard deviation to understand uncertainty around the estimates.

Uncertainty around estimates

When people give us predictions or summaries, how much certainty exists around the data? We can show uncertainty around the mean using the *standard error of the mean*, often abbreviated as SEM or simply SE for *standard error* because other estimated values may have SEs as well. More informatively, we can estimate confidence intervals, abbreviated CI. The most commonly CI is 95%. The CI will contain the true or correct estimate 95% of the time if we repeat our observation process. If we accept this probability view of the world, we know our CIs will include the mean 95% of the time, but we just won't know which 95% of the time. Both CIs and SEMs required to make assumptions about the data's distribution that [Link to Come] goes over.

Continuing with the previous example, we can calculate the standard error by dividing the standard deviation by the square root of the number of observations: $5.81/\sqrt{11} = 1.75$. Thus, we can write the mean as $6.18 \pm 1.75\ (\mathrm{SE})$. We could also calculate the 95% CI, which would be $\pm 1.75 \times 1.96$. For now, trust us on the 1.96. [Link to Come] shows where this number comes from.

> **TIP**
>
> As an approximate rule of thumb when working with statistically normal data, 99% of data falls with 3 standard deviation from, 95% of data falls within 2 standard deviations (1.96 rounded up) from the mean, and 68% of data falls within 1 standard deviation from the mean.

Thus, we could write the means as 6.18 ± 3.43 $(95\%\ \mathrm{CI})$, or more informatively, we can write the mean as 6.18 $(2.38\ \ \text{to}\ \ 9.24;\ \ 95\%\ \mathrm{CI})$

WARNING

Always include uncertainty such as a confidence interval around estimates such as mean when presenting to a technical audience. Presenting a *naked* mean is considered bad form because it does not allow the reader to see how much uncertainty exists around an estimate.

Based upon statistical convention, we can compare 95% CIs to examine if estimates differ. For example, 6.18 $(2.38\ \ \text{to}\ \ 9.24\ 95\%;\mathrm{CI})$ differs from 0, so we can say the air yards differs from zero based upon statistical uncertainty. If we were comparing two estimated means, we could compare both 95% CIs. If the CIs did not overlap, then we can say the means are different.

NOTE

People use 0.05 as the probability for being wrong because of historical convention. There is not a great reason other than people have always been doing this. Wasserstein, Schirm, & Lazar discuss this in a 2019 editorial in the *The American Statistician*, which is freely available online at *https://www.doi.org/10.1080/00031305.2019.1583913*. Their editorial present many different perspectives on alterive methods for statistcial inference.

Chapter 6 also covers more about statistical inferences. In Chapter 6, we will also cover about different methods for estimating variances and confidence intervals. [Link to Come] will also cover more about basic statistics when we cover about probability distributions. Now, enough about theory and hand calculations, let’s see how to estimate these values in Python and R!

Calculating summary statistics with Python and R

To calculate summary statistics with Python and R, first we read in the data. Remember to change your path to point to where your data is located. We also load our required R and Python packages.

```
## R
library(tidyverse)
gb_det_2020_pass <- read.csv("./data/gb_det_2020_pass.csv")
```

```
## Python
import pandas as pd
gb_det_2020_pass = pd.read_csv("./data/gb_det_2020_pass.csv")
```

Next we look at summary of data frame, like in Chapter 4. Hopefully you now understand where these numbers come from and what they tell us. The `1st Qu.` and `3rd Qu.` are the first and third quantiles in R. Thus, 50% of the data falls between these data points. They help us get a sense for the middle of the data.

```
## R
summary(gb_det_2020_pass)
```

```
   posteam          yards_after_catch   air_yards      pass_location     
 Length:62          Min.   :-2.000    Min.   :-6.000   Length:62         
 Class :character   1st Qu.: 2.250    1st Qu.: 1.250   Class :character  
 Mode  :character   Median : 4.000    Median : 5.000   Mode  :character  
                    Mean   : 6.263    Mean   : 8.613                     
                    3rd Qu.: 9.000    3rd Qu.:12.750                     
                    Max.   :20.000    Max.   :50.000                     
                    NA's   :24                                           
  qb_scramble
 Min.   :0   
 1st Qu.:0   
 Median :0   
 Mean   :0   
 3rd Qu.:0   
 Max.   :0   
```

In R, we can also `describe()` the data see similar summaries that also include the median, count, and maximum value. One benefit of using `summary()` is that shows the missing or `NA` values in R. This can help you see possible problems in the data.

```
print(gb_det_2020_pass.describe())
```

```
       yards_after_catch  air_yards  qb_scramble
count          38.000000  62.000000         62.0
mean            6.263158   8.612903          0.0
std             5.912352  10.938509          0.0
min            -2.000000  -6.000000          0.0
25%             2.250000   1.250000          0.0
50%             4.000000   5.000000          0.0
75%             9.000000  12.750000          0.0
max            20.000000  50.000000          0.0
```

We can also summarize the data by hand in R. We pipe the data using `|>` to the `summarize()` function. We then tell R to what functions to use on which columns. We use `min()` for the minimum, `max()` for the maximum, `mean()` for the mean, `median()` for the median, `sd()` for standard deviation, `var()` for the variance, and `n()` for the count. We also need to tell R what to call the output columns. You can see our naming for output columns here. We chose these names because they are short are relatively easy to both type and understand what they are from.

```
## R
gb_det_2020_pass |>
summarize(min_yac = min(yards_after_catch),
          max_yac = max(yards_after_catch),
          mean_yac = mean(yards_after_catch),
          median_yac = median(yards_after_catch),
          sd_yac = sd(yards_after_catch),
          var_yac = var(yards_after_catch),
          n_yac = n())
```

```
  min_yac max_yac mean_yac median_yac sd_yac var_yac n_yac
1      NA      NA       NA         NA     NA      NA    62
```

R only give us `NA` values. What is going on? Recall that these columns have missing data, so we need to tell R to ignore them using the `na.rm = TRUE` option in the functions.

```
## R
gb_det_2020_pass |>
summarize(min_yac = min(yards_after_catch, na.rm = TRUE),
          max_yac = max(yards_after_catch, na.rm = TRUE),
          mean_yac = mean(yards_after_catch, na.rm = TRUE),
          median_yac = median(yards_after_catch, na.rm = TRUE),
          sd_yac = sd(yards_after_catch, na.rm = TRUE),
          var_yac = var(yards_after_catch, na.rm = TRUE),
          n_yac = n())
```

```
  min_yac max_yac mean_yac median_yac   sd_yac var_yac n_yac
1      -2      20 6.263158          4 5.912352 34.9559    62
```

A reasonable question would be, why did we just do all that coding for almost the same output as `describe()`? First, we can customize what outputs appear. Second, we can now group or aggregate by other predictors. For example, we can now easily estimate these values by the team with possession by using `group_by()` function with `posteam` as an input. Notice how we keep piping outputs along to the next function. We also demonstrate how this may be done for a second variable, `air_yards` as well. We drop variance and medians to allow the results to more easily be displayed.

```
## R
gb_det_2020_pass |>
group_by(posteam) |>
summarize(min_yac = min(yards_after_catch, na.rm = TRUE),
          max_yac = max(yards_after_catch, na.rm = TRUE),
          mean_yac = mean(yards_after_catch, na.rm = TRUE),
          sd_yac = sd(yards_after_catch, na.rm = TRUE),
          min_ay = min(air_yards, na.rm = TRUE),
          max_ay = max(air_yards, na.rm = TRUE),
          mean_ay = mean(air_yards, na.rm = TRUE),
          sd_ay = sd(air_yards, na.rm = TRUE),
          n = n())
```

```
# A tibble: 2 × 10
  posteam min_yac max_yac mean_yac sd_yac min_ay max_ay mean_ay sd_ay     n
  <chr>     <int>   <int>    <dbl>  <dbl>  <int>  <int>   <dbl> <dbl> <int>
1 DET           0      20     6.9    6.05     -6     50    8.03  11.6    32
2 GB           -2      19     5.56   5.84     -4     34    9.23  10.3    30
```

We can also do similar summarizes with Python. For Python, we use the `.agg()` function to aggregate the data frame. We use a dictionary insides of Python to tell Pandas which column to aggregate and what functions to use. Recall that Python defines dictionaries using `{"key" : [values]}` notation. In this case, the dictionary uses the column `"yards_after_catch"` as the key and the aggregating functions as the list values.

```
print(gb_det_2020_pass.agg(
    {
        "yards_after_catch": ["min", "max", "mean", "median",
                              "std", "var", "count"]
    }
))
```

```
        yards_after_catch
min             -2.000000
max             20.000000
mean             6.263158
median           4.000000
std              5.912352
var             34.955903
count           38.000000
```

Python also has a grouping function, `.groupby()`, that can take `"posteam"`. Notice that Python does not use piping. Instead, we string together function one after each other. This is due to the object orientated nature of Python compared to the procedural nature of R. Both approaches have trade-offs and largely boil down to personal preference.

```
print(gb_det_2020_pass.groupby("posteam").agg(
    {
        "yards_after_catch": ["min", "max", "mean",
                              "median", "std", "var", "count"]
```

```
    }
))
```

```
        yards_after_catch
                      min   max      mean median       std        var count
posteam
DET                   0.0  20.0  6.900000    4.0  6.051533  36.621053    20
GB                   -2.0  19.0  5.555556    4.5  5.843269  34.143791    18
```

With Python, we can include a second variable by including a second entry in the dictionary. Also, Pandas, unlike the Tidyverse, allows us to calculate different summaries for each variable by changing the dictionary values.

```
print(gb_det_2020_pass.groupby("posteam").agg(
    {
        "yards_after_catch": ["min", "max", "mean",
                              "median", "std", "var", "count"],
        "air_yards": ["min", "max", "mean",
                      "median", "std", "var", "count"]
    }
))
```

```
        yards_after_catch                                                   \
                      min   max      mean median       std        var count
posteam
DET                   0.0  20.0  6.900000    4.0  6.051533  36.621053    20
GB                   -2.0  19.0  5.555556    4.5  5.843269  34.143791    18

        air_yards
              min max      mean median        std         var count
posteam
DET            -6  50  8.031250    5.0  11.607796  134.740927    32
GB             -4  34  9.233333    5.0  10.338023  106.874713    30
```

Presenting summary statistics

The key for presenting summary statistics are to make sure you use the information available to you to effectively tell your story. Firstly, know your target audience is extremely important. For example, if you're talking to Cris Collinsworth about his next Sunday Night Football broadcast (something Eric does on a regular basis) or to your buddies at the bar during a game, you're going to present the information differently.

Furthermore, if you're presenting your work to the Director of Research and Strategy for an NFL team, you're probably going to have to supply different, specifically more, information than in the aforementioned two examples. Likewise, when talking to the Director of Research and Strategy, you will likely need to justify both your numbers and your method choices. Unless you're having beers with Eric and Richard (or other quants), you probably will not be discussing model choices over beers!

The "why" is key and you'll have to dig into data and truly understand it well, so that you can speak it in a number of different languages. For example, is the dynamic you're seeing due to coverage differences, the wide receivers, or changes in the quarterback's form?

Second, use numbers to support your story, but do not use numbers as your story. For example, say *"Green Bay has slightly less yards after the catch compared to Detroit, with Green Bay having an average of 5.5 yards and Detroit having 6.9 yards"* rather than saying *"Detroit passed an average of 6.9 years. Green Bay passed an average of 5.5 years. Green Bay scored 2 more points on average in the middle of the field…"*. Adding context to numbers is something that we, as authors, have noticed helps the best quantitative people stand out compared to many quantitative people. In fact, communication skills about numbers helped both us get our current jobs.

Third, while a picture may be worth a thousands words, walk your reader through the picture. A graph with no context is likely worse than no graph at all.
For a non-technical audience, you may include a figure and mention the "averages" in your words. Thus, the raw summary statistics may not even be shown in your writing. For more technical audiences, include the details and uncertainty either in text for one or two number or in a table or supplemental materials for more summary statistics.

Finally, we have found there are two good ways to improve our presenting of summary statistics. First, present early and present often to people who will give you constructive feedback. Make sure they can understand your message, and if they cannot, ask them what is unclear and figure out how to more clearly make your point. For example, the authors like to give lectures and seminar to students because we will ask our students how they might explain a figure and then they help us to more clearly think about data. Also, if we cannot explain concepts to high school and college students, we do not clearly understand the ideas well.

Second, look at other people's work. Read blogs, read other books, read articles. Other people's examples will help you see what is clear and what is not. Besides casual reading, read critically. What works? What does not work? Why did the authors make a choice? If you have a chance, ask the authors if you see them or interact with them on social media such re-tweeting. A diplomatic tweet, will likely start a conversation. For example, replying to a tweeting *I liked your model and the insight it gave me to Friday's game. Why did you use X rather than Y?*. Conversely, replying to a tweet with *your model sucked, you should use my favorite model.* will likely be ignored or possibly start a pointless flamewar and decrease not only the original poster's view of you, but also other people who read the tweet.

Exercises

1. Using the NFL Draft data scraped in Chapter 3, find the mean, median and standard deviation of `DrAV` for all position groups. Is the NFL better at finding talent at some positions in the draft rather than others?

2. Using the NFL Draft data alluded to in question 1, find the maximum `DrAV` earned for every pick in the draft. We know that Tom Brady has been the league's most valuable pick 199, but how can we quantify this?

3. Using the NFL Scouting data scraped in the exercises of Chapter 3, find the mean, median and standard deviation of the 40-yard dash times among different positions. It's obvious that some positions are faster than others, but are there any surprises in this analysis?

4. Using the NFL Scouting data alluded to in Chapter 3, find the minimum forty-yard dash time for all position groups. Are there any surprises in this list? What does this say about the 40-yard dash and how it translates to NFL success?

Future readings

Many different books exist describing introductory statistics. If you want to learn more about statistics, we suggest reading the first 1-2 chapters of several books until you find one that *speaks* to you. Some books you may wish to consider include:

- *Advancing into Analytics: From Excel to Python and R* by George Mount (O'Reilly Media).

This book assumes a reader knows Excel well, but then helps the reader to transition to either R or Python. The book covers the basis of statistics.

- *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse* by Chester Ismay and Albert Y. Kim (CRC Press), also updated at the book's homepage *https://moderndive.com/*

This book provides a robust introduction to statistical inferences for people who also want to learn R.

- Practical Statistics for Data Scientists by Peter Bruce, Andrew Bruce (O'Reilly Media).

This book provides an introduction to statistics for people who already know some R.

Chapter 6. Linear models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Linear models, while being the simplest of inference and prediction tools, are also one of the most powerful. Linear models leverage the two mathematical operations that we most often encounter in our everyday lives, even if we don't necessarily think of it that way. Through scaling and addition we put together combinations of predictor variables (also called "features") in attempt to explain or predict an outcome variable (also called a "response"). Subject matter expertise helps us determine which predictor variables to use, while the data determines the scaling factors, or coefficients, through the regression process. Examining this coefficients allow us to understand the past through statistical inference. Using these coefficients with new data allows us to make predictions about new situations such as the future or new locations.

For example, let's say we want to make bratwurst (brats for short) for our tailgate party. The average brat is 1 sausage link and 1 bun halves. We could write this as an equation:

$$\text{brat} = \text{sausagelink} + 2 \times \text{bunhalves}$$

With this delicious, but silly example, brat would be the response variable and sausage link and bun halves would be the predictor variables. Predictor variables are often called x such as x_1 and x_2 and the response variable y due to historic convention. When plotting a regression where there is one predictor variable, the predictor variable usually goes on the left-to-right or horizontal axis (x-axis) and the response variable goes on the up-and-down or vertical (y-axis).

The above example, while illustrative, is not exactly how linear models work in situations where we are trying to learn something new. In the real world we don't know before hand how many sausage links or bun halves go into a bratwurst, the same way we don't know how many times to count a defensive lineman's pressures when trying to determine how many wins he's worth on the football field for his team.

Mathematically, we have described one brat in terms of its ingredients We start with simple linear models because we use them on a daily basis in our jobs and think you will find them to be a helpful tool as well. Linear models also help us to better understand more complex models ranging from statistical methods we cover in future chapters such as logistic regression as well as many machine learning tools. In fact, we would go as far as saying that linear models are the workhorse for much of statistics!

Linear models have a long history in the field of statistics and have been used by people to understand and predict the world since the early 1800s. As computers have become more powerful, we can now fit linear models to larger and larger datasets as well as more complex models. If you have had an introductory statistics course, you almost certainly learned about some special cases of linear models. The broad term, linear models captures several other types of models.

This includes regressions such as linear regression and multiple regression as well as ordinary least-squared regression (OLS) when certain algorthims are used to fit the regression model. Other statistical methods are also related to linear models. The analysis of variance (ANOVA) model is a special case of a linear model and a t-test is a special case of an ANOVA.

An ANOVA focuses on how a model explains variation (e.g., what predictors affect the amount of uncertainty or variability in the data) whereas a linear model focuses on how we predict the *average* outcome for input parameters.

> **NOTE**
>
> The term *Linear model* covers many types of statistcial models including linear regression, multiple regression, analysis of variance (ANOVA), and the **t**-test.

Linear models also extend to other methods, some of which we cover in this book. Linear models assume continuous response variables (e.g., player height or weight). A generalized linear models (GLMs) allows other types of response variables such as binary (e.g., heads or tails, win or lose) or counts with many zeros and integers (e.g., sacks per game). Non-linear predictor variables can be modeled using generalized additive models (GAMs). For example, the optimal temperature for a football game is neither too cold nor too hot. A GAM model could capture this. We can also model nested data such a players through the seasons using linear mixed-effect models (LMMs) and generalized linear mixed-effect models.

Total passing yards

We will examine the total passing yards from plays, `passing_yards` for short with the computer, from the first Green Bay and Detroit game of 2020. However, this variable does not exist in our dataset. Thus, we will need to create it. We find we often need to manipulate data to create the variables that we want.

Creating a new variable, `passing_yards` takes multiple steps.

1. We need to read in data.
2. We need to calculate whether a pass was complete or not.

3. We need to calculate the `passing_yards` by adding together the `yards_after_catch` and `air_yards` to completed passes.

With R, we load the `tidyverse` package. Then, we read in the data and filter. To make a column for complete or non-complete, we create a column called `complete`, which is `1` if the value of `yards_after_catch` is not `NA` and `0` otherwise.
In R, we can `mutate()` our data with the `tidyverse` to create the new columns `complete` and `passing_yards`, with the latter being the total yards generated on a completed pass.

```
## R
library(tidyverse)
gb_det_2020_pass <-
    read.csv("./data/gb_det_2020_pass.csv") |>
    mutate(complete = ifelse(!is.na(yards_after_catch), 1, 0)) |>
    mutate(passing_yards = ifelse(complete == 1,
                                  yards_after_catch + air_yards, 0))
```

`pandas` uses parallel steps to R. First, we import the `pandas` package as `pd` and the `numpy` package as `np`. Then we read in the data. Next, we use the functions `np.where` and `np.isnan` to create the `complete` column by sorting through the `yards_after_catch` rows that have `NaN` values. Finally, we use the same `np.where` function to add up the total yards in the event of a completion (and 0 in the event of an incompletion).

```
import pandas as pd
import numpy as np
gb_det_2020_pass = pd.read_csv("./data/gb_det_2020_pass.csv")
gb_det_2020_pass["complete"] = \
    np.where(np.isnan(gb_det_2020_pass["yards_after_catch"]), 0, 1)

gb_det_2020_pass["passing_yards"] = \
    np.where(gb_det_2020_pass["complete"] == 1,
             gb_det_2020_pass["yards_after_catch"] +
             gb_det_2020_pass["air_yards"], 0)
```

> **TIP**
>
> Complex code is simply small code commands built together. To understand complex code, look at the little parts. Conversely, to solve problems, build up small steps to solve your problem.

Intercept only models

Global intercept

Linear models have predictor coefficients. Often, people call these coefficients *slopes* if the predictor variable is continuous and *intercepts* or *contrasts* if the predictor variable is a category. Some of the simplest linear models only have intercepts. In fact, the simplest model only has one intercept! This intercept is the global mean. We will work with the `passing_yards` column we just created. You may see this in Figure 6-1. Revisit Chapter 2 if you need help creating this figure on your own.

> **NOTE**
>
> Formulas with `statsmodels` are usually similar or identical to R. This is because computer languages often borrow from other computer languages. Python's `statsmodels` borrowed formulas from R, similar to `panda` borrowing data frames from R. R also borrows ideas, and R is infact a recreation of the S language. As another example in R, the `tidyverse` borrows syntax and ideas for cleaning data from `SQL`-type languages.

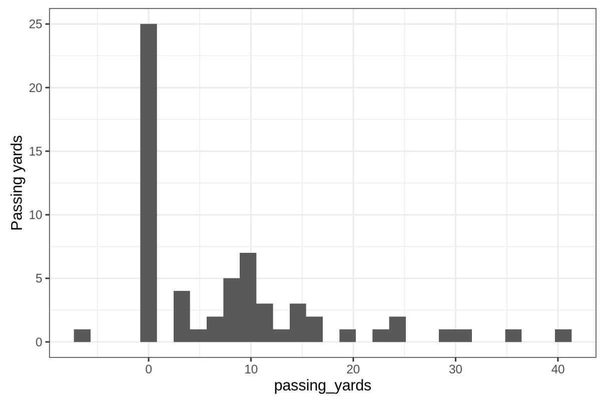

Figure 6-1. Histogram of passing yards.

Before we build our first model, let's summarize the data using R. We will calculate the mean, number of observations, and standard deviation. We will then use these to calculate the *standard error of the mean* (SEM). These can then be compared to a linear model:

```
## R
gb_det_2020_pass |>
    summarize(ave_passing = mean(passing_yards),
              n = n(),
              sd = sd(passing_yards), .groups = "drop") |>
    mutate(sem = sd / sqrt(n))
```

```
  ave_passing  n       sd      sem
1    7.806452 62 9.826092 1.247915
```

A formula in R has a *left-hand side* (LHS for short) and *right-hand side* (RHS for short). The LHS is predicted by the RHS. The tilde symbol (~) tells the computer what is predicted, and we often read the ~ symbol as *predicted by*. The simplest model only has an intercept and no predictor variables. For example, we could use `passing_yards ~ 1` to tell the computer that `passing_yards` is predicted by a global intercept. We use the linear model function, `lm()` and also specify the data. We save the model as `lm_out` and then print the output:

```
## R
lm_out <- lm(formula = passing_yards ~ 1,
             data = gb_det_2020_pass)
print(lm_out)
```

```
Call:
lm(formula = passing_yards ~ 1, data = gb_det_2020_pass)

Coefficients:
(Intercept)
      7.806
```

The output tells us the input setting for the formula or (computer `call`) as well as the coefficients (or, in this case coefficient named `(Intercept)`). Notice, the intercept from this output is the same as the average passing yards from before. In this simple case, the linear model is only a fancy method for calculating the mean. We can look at the summary of our model, which we saved as `lm_out`, using the `summary()` function:

```
## R
summary(lm_out)
```

```
Call:
lm(formula = passing_yards ~ 1, data = gb_det_2020_pass)

Residuals:
    Min      1Q  Median      3Q     Max
-13.806  -7.806  -2.306   3.194  33.194

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    7.806      1.248   6.256 4.33e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.826 on 61 degrees of freedom
```

The summary include the `Call`, like `print()`. Next, the summary includes the residuals, which help us understand how the model fits and are described later in this chapter. The summary then shows the model's estimated coefficients. In addition to the estimated value, a standard error for the model coefficient (`Std. Error`), test statistics (specifically a $t-$value), and $p-$value are included. Notice how the `Std. Error` was the same as the `SEM` we calculated by hand.

The $p-$value provides the probability of obtaining the observed $t-$value assuming the null hypothesis of the coefficient being zero is true. The $p-$value ties into null hypothesis significance testing (NHST), something that most introductory statistics courses cover, but is increasing falling out of use by practicing statisticians. Next, the summary provides us with a graphical summary of the p-value as well well as the code:

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Note that this code shows ranges. For example, `'***'` corresponds to P-values between 0 and 0.001. Lastly, the summary provides residual standard error, that is to save the variability not captured by the model and the degrees for freedom. Degrees of freedom are *extra* data points compared to the number of coefficients estimates. For example, we had `38` observations and `1` coefficient estimated. Thus, we have `38 - 1 = 37` degrees for freedom. Models sometimes pool data for estimates, thus degrees of freedom may not always be integers such as 1, 2, or 34, but also real numbers such as 3.4 or 55.7.

TIP

Checking degrees of freedom may seem strange to people starting out modeling. However, this can be a great check for your data to make sure the model is using all of your inputs correctly and values are not being lost. Likewise, checking degrees of freedom can be a great check for your model to make sure your model is using the data correctly. We have a friend who spends most of a semester teaching her graduate students in statistics how to compare degrees of freedom across different models. *Do not underestimate the utility of understanding degrees of freedom!*

The purpose of this coding exercise was to show you how a linear model calculates a mean and standard deviation. For more complex models, things change and are not as simple, as we will see in the future. Before we look at more complex examples, let’s repeat with Python.

NOTE

R was created for teaching statistics, based upon the S language. Given this history and the state of statistics in the early 1990s, R has linear models well integrated into the language. In contrast, Python has a clone of R for linear models for statistcial inference, specifically the `statsmodels` package. The main package for models in Python, scikit-learn (`sklearn`) focuses on machine learning rather than statistcial inference. Understanding the history of R and Python can provide insight into *why* the language exist as they do. We would also argue that if all one needs and wants to do is fit regression models for statistical inference, R would be the better software choice.

With Python, we import `statsmodels.formula.api` as `smf`. We then need to build a model using the ordinary least squares (`.ols()`) function. Notice this syntax is *almost* identical to R, but uses quotes around the formula. After building the model, we have to explicitly tell Python to fit the model using the `.fit()` command. Then, we can print the `.summary()`:

```
## Python
import statsmodels.formula.api as smf
lm_out_build = smf.ols(formula = "passing_yards ~ 1",
                       data = gb_det_2020_pass)
lm_out = lm_out_build.fit()

print(lm_out.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:          passing_yards   R-squared:                       0.000
Model:                            OLS   Adj. R-squared:                  0.000
Method:                 Least Squares   F-statistic:                       nan
Date:                Sat, 09 Jul 2022   Prob (F-statistic):                nan
Time:                        16:14:49   Log-Likelihood:                -229.14
No. Observations:                  62   AIC:                             460.3
Df Residuals:                      61   BIC:                             462.4
Df Model:                           0
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------
--
Intercept      7.8065      1.248      6.256      0.000       5.311      10.302
==============================================================================
```

```
Omnibus:                       21.351   Durbin-Watson:                   1.962
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               28.412
Skew:                           1.411   Prob(JB):                     6.77e-07
Kurtosis:                       4.744   Cond. No.                         1.00
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly
 > specified.
```

The Python printout for `OLS Regression Results` is similar to R's `summary()`, but contains more details. First, notice the details about the model that include the dependent (response) variable (`Dep. Variable`), the model's name (`Model`), the numerical methods used to fit the model (`Method`), the `Date` and `Time` the model was fit, the total number of observations (`No. Observations`), the number of degrees of freedom for the residuals (`Df Residuals`) and model (`Df model`), and the method used for the model's covariance. The model output also includes the R^2 and adjusted R^2 values that provide insight into how well the model fits. An R^2 of 1.0 is a prefect fit and 0.0 is no fit. Hence, this model does a poor job of predicting the data because the $R^2 = 0$. The adjusted R^2 accounts for the number of parameters. The `F-statistic` and corresponding probability allow for model comparison. The `Log-Likelihood` comes from how well the model fits the data. The Akaike information criterion (`AIC`) and Bayesian information criterion (`BIC`) allow for model selection and comparison of different models. [Link to Come] covers information criterion in greater detail as part of material on model selection.

Next, the model summary includes the coefficient estimates that are in a similar format to R, but also include the 95% confidence interval (CI). Given statistical theory, we can expect the 95% CI to contain the *correct* or *true* value 95% of the time if we were to repeat our observation process or experiment a very large number of times. Although these definition may (and hopefully does) seem strange, the definition highlights a major constraint of NHSTs and fitting statistical models. Philosophically, most modern statistical method assume data reflects long-term averages if the observation process or experiment is repeated an infinite number of times. Hence, we need to be aware our models' estimates will be wrong. Practically, we can compare the 95% CI to to other values. If the 95% CI does not include a value, we can say the estimate differs from that value. Usually, people care if coefficients differ from zero. Hence, both Python and R compare coefficients to a null model of a coefficient equaling zero.

With both Python and R we often times want to extract coefficients. With R, the `broom` package allows us to extract model coefficients across almost all models in R to a standard, *tidy* format using the `tidy()` function. We can also tell R to include CIs by setting `conf.int = TRUE`. The default setting is 95% CI:

```
## R
library(broom)
tidy(lm_out, conf.int = TRUE)
```

```
# A tibble: 1 × 7
  term        estimate std.error statistic      p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>        <dbl>    <dbl>     <dbl>
1 (Intercept)     7.81      1.25      6.26 0.0000000433     5.31      10.3
```

Python does not have as well of developed option, so we show you how to extract the outputs directly from the fit model. The suffix `.params` shows a model's parameter estimates. Notice that `.params` is *not* a function and does not include parentheses. Instead, this shows a model's estimated parameters, which are an attribute (also sometimes called an attribute) of the fitted model object.

> **NOTE**
>
> In object-oriented programming two broad methods exist for viewing properties of an object. First, many objects have species functions to view properties. For example, `lm_out.summary()` is a function to view outputs formatted nicely. Using a car analogy, this is like viewing how much fuel is in the gas tank using the gas gauge of your car. The *summary* might also tells you other useful outputs based upon how much gas is in your car such as your millage or range. Second, the raw parts of an object may often be viewed by directly typing the part's name. For example, `lm.params` directly shows us the parameter values. Returning to the car analogy, this is like viewing the amount of gas by opening the gas lid and looking in. Sometimes, opening might be the only way, but it can be messier and more accident prone. With code, directly looking at objects can be slightly dangerous if you accidentally change the value for the part of an object you look at.

```
## Python
print(lm_out.params)
```

```
Intercept    7.806452
dtype: float64
```

The Python function `.conf_int()` displays the confidence intervals for a model:

```
## Python
print(lm_out.conf_int())
```

```
                  0          1
Intercept  5.311091  10.301812
```

Multiple intercepts

We often want to predict or compare one or more groups. For example, we might want to compare the passing yards by each team Figure 6-2. We will repeat the summarizing, but this time group by the team of possession of the ball.

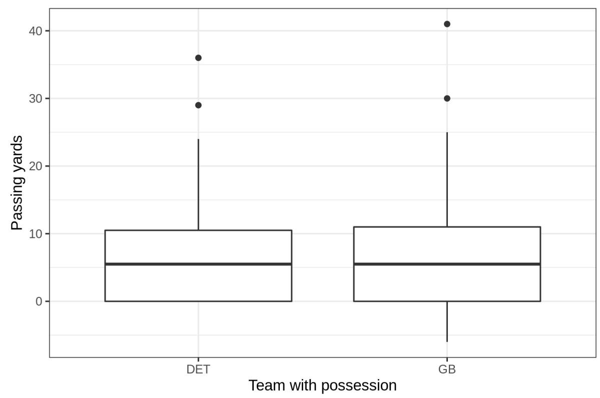

Figure 6-2. Boxplot of passing yards by team.

```
## R
gb_det_2020_pass |>
    group_by(posteam) |>
    summarize(ave_passing = mean(passing_yards),
              n = n(),
              sd = sd(passing_yards),
              .groups = "drop") |>
    mutate(sem = sd / sqrt(n))
```

```
# A tibble: 2 × 5
  posteam ave_passing     n    sd   sem
  <chr>         <dbl> <int> <dbl> <dbl>
1 DET            7.62    32  9.23  1.63
2 GB             8       30 10.6   1.93
```

We can include the team of possession of the football (`posteam`) by adding a term to the formula in either R or Python:

```
1 + posteam
```

This output calculates an intercept for the first team alphabetically (`DET`) and then the *contrast* or difference with the other team (`GB`). Both languages tells us this is from the `posteam` variable, R with `posteamGB` and Python `posteam[T.GB]`. We simply print the outputs because we do not want or need to save the models for later:

```
## R
lm(passing_yards ~ 1 + posteam, gb_det_2020_pass)
```

```
Call:
lm(formula = passing_yards ~ 1 + posteam, data = gb_det_2020_pass)

Coefficients:
(Intercept)     posteamGB
      7.625         0.375
```

```
## Python
print(smf.ols(formula = 'passing_yards ~ 1 + posteam',
              data = gb_det_2020_pass).fit().params)
```

```
Intercept          7.625
posteam[T.GB]      0.375
dtype: float64
```

We do not need to include the `1 +` to estimate an intercept. In fact, we do not usually include the `1 +` in models' formula unless we want to be explicit, usually when teaching. Notice the outputs are the same as above:

```
## R
lm(passing_yards ~ posteam, gb_det_2020_pass)
```

```
Call:
lm(formula = passing_yards ~ posteam, data = gb_det_2020_pass)

Coefficients:
(Intercept)     posteamGB
      7.625         0.375
```

```
## Python
print(smf.ols(formula = 'passing_yards ~ posteam',
              data = gb_det_2020_pass).fit().params)
```

```
Intercept          7.625
posteam[T.GB]      0.375
dtype: float64
```

However, what if we want to estimate the mean for each team? We can use the formula `posteam - 1`. The `- 1` tells the formula to estimate an intercept for each team rather than one intercept and a contrast. In this case, we save the outputs, because we want to look at them:

```
## R
lm_out_posteam <- lm(passing_yards ~ posteam - 1,
                     gb_det_2020_pass)
```

We can then use `summary()` to look at the output using R. Previously, we calculated the mean and sem *by hand*. The means are the same as those we previously calculated. Notice that now the `Std. Error` now differs from the SEM. That's the "magic sauce" of linear models. Specifically, how models capture variability make them different from simply estimating means. In fact, a sub-discipline of statistics deals with the analysis of variance, or ANOVA, for short! We only include the R example, but the Python example produces the same results.

```
## R
summary(lm_out_posteam)

Call:
lm(formula = passing_yards ~ posteam - 1, data = gb_det_2020_pass)

Residuals:
    Min      1Q  Median      3Q     Max 
-14.000  -7.625  -2.125   3.000  33.000 

Coefficients:
           Estimate Std. Error t value Pr(>|t|)    
posteamDET    7.625      1.751   4.354 5.29e-05 ***
posteamGB     8.000      1.809   4.423 4.16e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.906 on 60 degrees of freedom
Multiple R-squared:  0.391,	Adjusted R-squared:  0.3707 
F-statistic: 19.26 on 2 and 60 DF,  p-value: 3.45e-07
```

Design matrix

The formulas in R and Python create something called a design matrix. The design matrix tells the computer what parameters to use for the model. This matrix is then solved to the observed data to estimate the model coefficients. Usually, the linear algebra used results in ordinary least-square (OLS methods). Different numerical methods exist, but are beyond the scope of this book. However, these numerical methods can be important for professional data scientist and statisticians

R uses the function `model.matrix()` to create a design matrix. Let's look at the first example for the formula for predicted by team of possession: `~ posteam`. We only look at the top (or `head()`) of the data to keep the output easy to read:

```
## R
model.matrix( ~ posteam, gb_det_2020_pass) |>
head()
```

```
  (Intercept) posteamGB
1           1         0
2           1         0
3           1         0
4           1         1
5           1         1
6           1         1
```

Notice, that an intercept is estimated for all data points, but a `0` and `1` are used to indicate if Green Bay was in possession of the ball. In contrast, compare to predicting where each team has their own intercept: `posteam - 1`:

```
## R
model.matrix( ~ posteam - 1, gb_det_2020_pass) |>
head()
```

```
  posteamDET posteamGB
1          1         0
2          1         0
3          1         0
4          0         1
5          0         1
6          0         1
```

In this case, column 1 is Detroit's possession and column 2 is Green Bay's possession.

With Python, we need to use the `patsy` package to access a design matrix function, `dmatrix()`. We also tell Python to only look at the head of the data when we read the data into the function, in contrast to R where we use the `head()` function on the output.

```
## Python
import patsy
patsy.dmatrix("posteam - 1", gb_det_2020_pass.head())
```

```
DesignMatrix with shape (5, 2)
  posteam[DET]  posteam[GB]
             1            0
             1            0
             1            0
             0            1
             0            1
  Terms:
    'posteam' (columns 0:2)
```

> **TIP**
>
> We included designed matrices primarily as a learning tools. However, we use design matrices when debugging or understanding our models in our day jobs, especially when we want to understand models we are using and re-using for important predictions. Furthermore, Richard also uses them when formatting data for custom, advanced models he builds using the program *Stan*. So, tuck the design matrix tool in the back of your toolbox, because you may someday use it!

Slopes and intercepts

Linear models can also be expanded to include continuous predictor variables, which are commonly called slopes. Hence, we now have models with slopes and intercepts. These models are commonly called linear regression. The simplest linear regressions only have one slope and one intercept. [Link to Come] covers more complex regressions.

With a simple linear regression, several different methods exist for describing the model with math. The predictor variable is usually x and the response variable is y. Some people write the model with the slope m and intercept b: Another way people write a simple linear regression is with the intercept a and slope b: Confused yet? We found these notations confusing as well when learning. If you see somebody using a linear regression and are confused about their equations, don't be afraid to ask. If their math is confusing to you, they probably have not explained it well!

Personally, we like to use an equation with the Greek letter β for the regression terms. Specifically, we use β_0 (pronounced "beta-naught") for the intercept and β_1 for the slope. We also use a ~ tilde in the equation:

We read this equation as y is predicted by beta-naught plus beta-one times x . We use these Greek letters because this notation allows us to extend the model in [Link to Come].

With a simple linear regression, the *intercept* is where the model's predicted value crosses or intercepts the *y*-axis. The *slope* is how the response variable changes given one unit change in x. As a more concrete example, we will build a model to examine the change in Green Bay's points scored during games from 2018 to 2020. We can use this question to ask if Green Bay started scoring more points after 2018, less points per game, or statistically the same number of points per game.

To start, we load in the data file, `score_GB.csv`, using R or Python. We need to reformat the data slightly before we can use it with these steps:

1. Break the `game_id` column into multiple column.
2. Reformat the new `Year` and `Week` columns to be numbers rather than text.

TIP

Manipulating data is hard. We understand if our approaches seem strange or counter-intuitive to you. We have found the only way to get better at it is to do it on a regular basis. Hopefully you will find that you become better at teaching yourself Python or R skills as you learn the language more. Our data manipulation skills come from three sources:

1. Experience programming with Python and R.
2. Looking up methods our knowledge gaps.
3. Experience effectively looking up information to fill our knowledge gaps.

This last step is the ability to know which book to look in or know how to effectively search the web for information as well as the experience to know when a book is better than the internet.

In R, we load the Tidyverse. Then, we first read in the file. We use the `separate()` function on `game_id` to split this column into `Year`, `Week`, `Home` (team), and `Away` (team) while deleting the value with `NA`. Next, we `mutate()` the `Year` and `Week` column into numeric values.

```
## R
library(tidyverse)
gb_scores <-
    read.csv("./data/score_GB.csv") |>
    separate(game_id, c("Year", "Week", "Home", "Away", NA)) |>
    mutate(Year = as.numeric(Year),
           Week = as.numeric(Week))
```

In Python, we first make sure Pandas is loaded. Then, we read in the data file. Next, we take the text *string* and split the text using `.str.split()`. We need to tell Python that the text is separated by the _ symbol and to `expand` the output. Lastly, we use the `.astype()` function with a dictionary to change `Year` and `Week` into integers.

```
## Python
import pandas as pd
gb_scores = pd.read_csv("./data/score_GB.csv")
gb_scores[["Year", "Week", "Home", "Away"]] = (
```

```
    gb_scores["game_id"].str.split("_",expand = True)
)
gb_scores = gb_scores.astype({"Year": "int32",
                              "Week": "int32"})
```

To see what our data looks like, we can plot `Year` against points scored by `GB` Figure 6-3. If you need help creating this figure, please revisit Chapter 2. Notice that the COVID19 year of 2020 had higher scoring on average for the Packers (and league wide). Also, Packers' quarterback Aaron Rodgers won league MVP in 2020, which contributed to this boost in scoring.

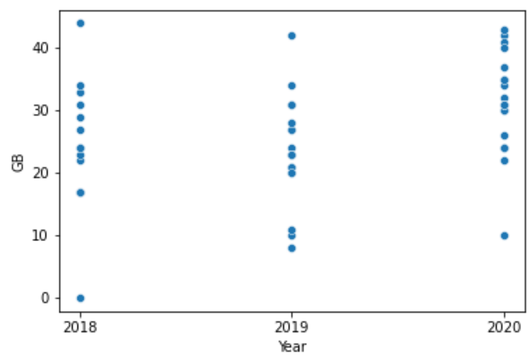

Figure 6-3. Plot of points scored by Green Bay in 2018, 2019, and 2020.

With R, we use the `lm()` function like before. We (optionally) use a `1` for the intercept in the formula and the continuous variable `Year` in the formula as well, we can also print the model outputs to see the coefficient.

```
## R
lm_gb_score_year <- lm(GB ~ 1 + Year, gb_scores)
print(lm_gb_score_year)

Call:
lm(formula = GB ~ 1 + Year, data = gb_scores)

Coefficients:
(Intercept)         Year
   -8212.40         4.08
```

Python uses the same formula as R. You'll need to make sure you have `statsmodels.formula.api` package imported before you use the `.ols` function. We can also build and `.fit()` the model in one step because we are not planning on re-using the model. Lastly, you can view the parameter estimates using `.params`.

```
## Python
import statsmodels.formula.api as smf
lm_gb_score_year = smf.ols(formula = "GB ~ 1 + Year",
                           data = gb_scores).fit()
lm_gb_score_year.params
```

```
Intercept   -8212.395692
Year            4.080499
dtype: float64
```

Both models produce the same outputs accounting for rounding on the screen. These outputs could be written as

$ = -8212.40 + 4.08 . $

Thus, for each additional year, Green Bay scored about 4 extra points per game per year during that three-year stretch of play. Likewise, at `Year = 0`, Green Bay would have scored negative 8,000 points.

This slope estimate hopefully seems reasonable. **But**, the intercept should not. Neither the Green Bay Packers (despite some cheesheads assertions) nor football existed around 0 A.D!

These observations highlight two major limitations with modeling. First, extrapolating beyond your dataset (in our case, any year other than 2018, 2019, or 2020) may lead to wrong conclusions.

> **NOTE**
>
> Models diverging from reality is a problem with any type of modeling and can happen to professional data scientists and even big tech companies. For example, Zillow misused their models or misunderstood limitations of their models when their house-flipping business when broke and caused them to lay-off around a quarter of their workforce around January 2022. Likewise, Google stopped their Google Flu because they realized their statistical models were diverging from reality in 2015 after working well for 7 years.

Second, having our first year be `2018` may not be the best choice. We may want to transform year to start with year 0 as the first year of data (2018) rather than year 0 being 0 AD. "Transformations" covers this topic in greater detail.

Looking at the summary from this model, we see the same formatted output as before for both R and Python. Looking at the outputs, notice how the slope estimate for `Year` differs from zero. Thus, scores increased through time. However, both models have low R^2 values and thus do a poor job of predicting score. In this case, our model does a good job of observing a trend, which may help us understand data, but a poor job of predicting the future.

```
## R
summary(lm_gb_score_year)

Call:
lm(formula = GB ~ 1 + Year, data = gb_scores)

Residuals:
    Min      1Q  Median      3Q     Max 
-22.051  -5.051   0.788   4.889  21.949 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)  
(Intercept) -8212.396   3124.448  -2.628   0.0114 *
Year            4.080      1.547   2.637   0.0111 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.013 on 50 degrees of freedom
Multiple R-squared:  0.1221,	Adjusted R-squared:  0.1045 
F-statistic: 6.953 on 1 and 50 DF,  p-value: 0.01112
```

With the Python output, notice the second `Note`:

```
The condition number is large, 5.05e+06. This might indicate that there are
strong multicollinearity or other numerical problems.
```

This message suggests a problem with the model. The `multicollinearity` warning would indicate we have two or more predictor variables that are highly correlated. But, we only have one predictor variable. Hence, we have `other numerical problems`. Give our personal experiences with data analysis and statistics, we would see this and think maybe the input predictor needs to be scaled. One clue suggesting this is that the estimated intercept is three orders of magnitude larger than the slope estimate. Before moving on to scaling data in "Transformations", let's look at how well the data fits the model.

```
## Python
print(lm_gb_score_year.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     GB   R-squared:                       0.122
Model:                            OLS   Adj. R-squared:                  0.105
Method:                 Least Squares   F-statistic:                     6.953
Date:                Sat, 09 Jul 2022   Prob (F-statistic):             0.0111
Time:                        16:14:50   Log-Likelihood:                -187.10
No. Observations:                  52   AIC:                             378.2
Df Residuals:                      50   BIC:                             382.1
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------
--
Intercept  -8212.3957   3124.448     -2.628      0.011   -1.45e+04   -1936.757
Year           4.0805      1.547      2.637      0.011       0.972       7.189
==============================================================================
Omnibus:                        1.568   Durbin-Watson:                   2.423
Prob(Omnibus):                  0.456   Jarque-Bera (JB):                0.836
Skew:                          -0.255   Prob(JB):                        0.658
Kurtosis:                       3.353   Cond. No.                     5.05e+06
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly
 > specified.
[2] The condition number is large, 5.05e+06. This might indicate that there
are
strong multicollinearity or other numerical problems.
```

Residuals

A very good question for any model *How well does the model fit the data?* Residuals are the difference between the predicted values from a model and the observed values. Figure 6-4 shows the residuals for our model. With our example, most data points are away from the regression line and therefore have large residuals.

Many times in sports analytics we want to separate the expected from the observed. Many models that are created in this space are "above expected", meaning there's some expected value like "expected yards", "expected completion", etc., which are derived using a model like a regression. Then, what is actually observed is compared with this value. That's a residual.

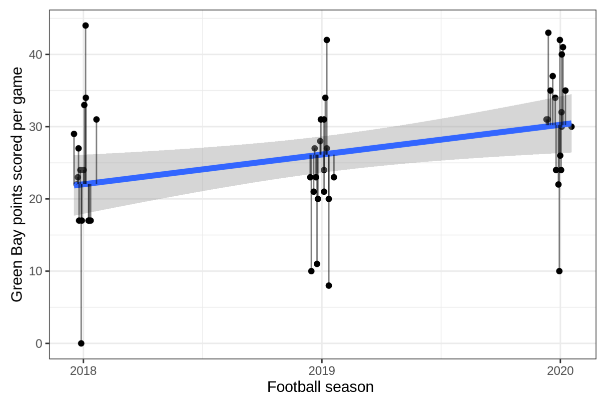

Figure 6-4. Plot of points scored by Green Bay in 2018, 2019, and 2020. The points jittered slightly on the x-axis to avoid overlap. The blue line is the regression line and the shaded region around the line is the model's 95% confidence interval. The lighter vertical lines show the residuals between the observed and predicted values.

Examining residuals also provide insight into a key assumption of linear models: That residuals are assumed to follow a normal distribution. The normal distribution may be familiar to you because it is also called the *bell curve*, aptly named after the bell-shaped line, for example see Figure 6-5. Although formal statistical tests exist, we find visualizing data to be a better approach to check for normality compare to the formal test for two reasons. First, the formal tests often fail to catch a lack of normality when dealing with smaller datasets. Second, the formal tests often declare large datasets to be non-normal simply due to many data points decreasing the $p-$value for the tests.

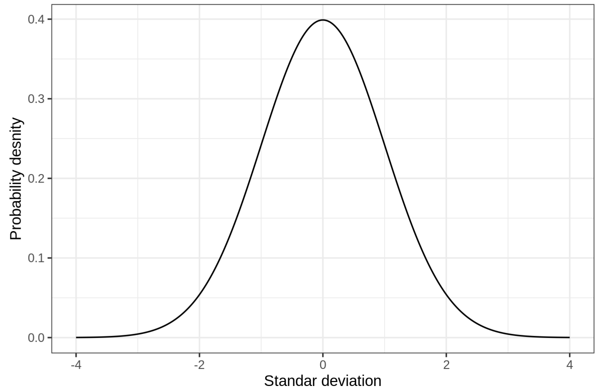

Figure 6-5. Example bell curve with a mean of zero and standard deviation of 1.

Both R and Python have tools for extracting the residuals. After extracting there resisduals, we can plot them and examine if the data look normal *enough*. With R, we use the `residuals()` function. Then, we create a `data.frame` before plotting a histogram with `ggplot2`.

```
## Python
df_res = pd.DataFrame({"residuals": lm_gb_score_year.resid})
sns.histplot(df_res, x = "residuals")
```

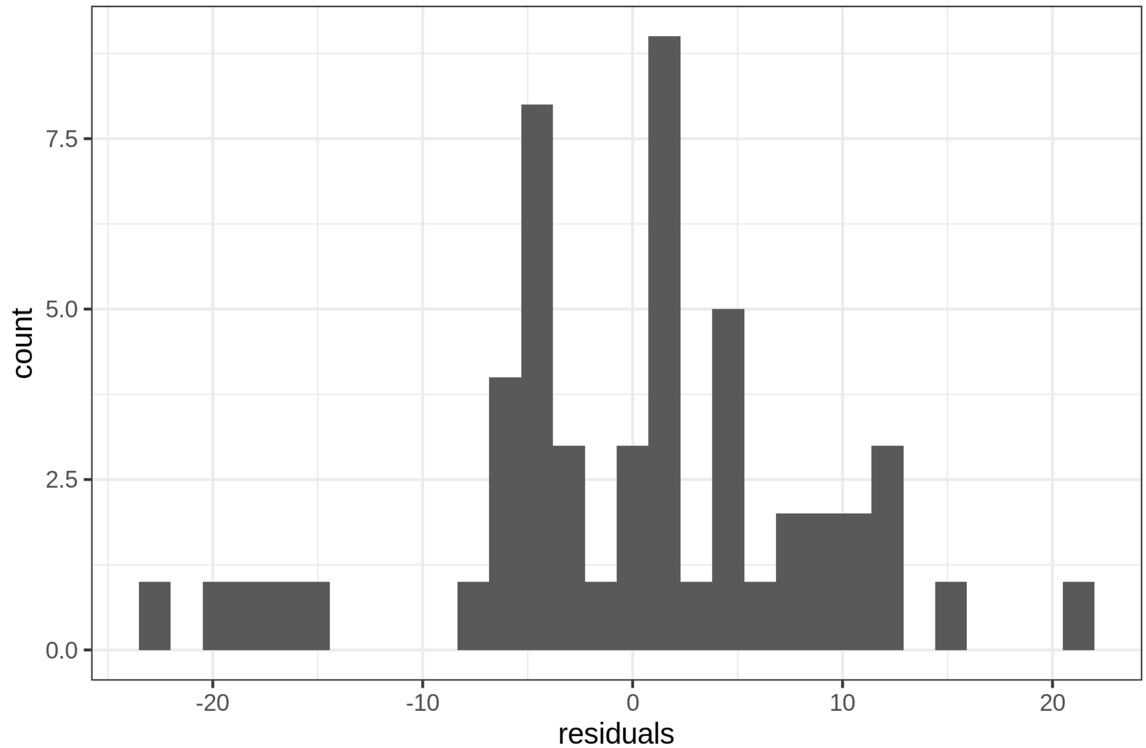

Figure 6-6. Histograms of residuals plotted in R (left) and Python (right).

The residuals plots shown in Figure 6-6 look close enough to a bell curve for our purposes. Not all data in football are this way. For example, earlier in this chapter we looked at passing yards. Incomplete passes take up about 30 to 40 percent of these passes, and so there would be a huge mass of plays centered at 0 passing yards, which would cause residuals to not be normal.

The residuals above are slightly *overdisperesed*, that is to say the far left and right values are farther away from 0 than would be expected with a normal distribution. This is due to the fact that there is theoretically no upper bound on the number of points that can be scored in a game, but there is a lower bound of zero.

Transformations

We sometimes have trouble fitting model. For example, our data (or, more specifically, the data's residuals) might not be *normal enough* for linear methods. So, what do we do? We have multiple options. In this section, we'll talk about transforming data. In [Link to Come], we will talk about other models.

We start of with two warnings about transformations. First, transformations can impact the importance of predictors, especially with multiple regression, which we talk about in [Link to Come]. For example, in the next example, we will see how changing the start year from 2018 to 0 changed the value of the slope coefficient and our interpretation of the coefficient. Second, transformations can make our model outputs harder to understand. Ben Bolker's book warns about this in his book, *Ecological Models and Data in R* (Princeton University Press, 2008) using a hypothetical ecology field study where a scientist counts the number of seeds. When analyzing the data, the scientist transformed the data so much that they are left asking: *What is the probability of observing at this much variability among the arcsine-square-root-transformed counts of seeds in different treatment?* Instead of creating statistical gibberish, we encourage you to think about how you will explain your model before transforming your data. Likewise, with modern tools, you may not even need to transform your data if you build your model around your data rather than forcing your data into the model!

Rescaling is one type of transformation, Revisiting the example from "Slopes and intercepts", what does the year zero mean and how do we define and understand it? Or, more directly what is special about year 0? Should 0 be 0 A.D. or the start of your observations? Or, should you use the middle? Maybe you should use the start of an era (for example, a new coach or quarterback). With our example, use can rescale to use the start of observations.

We need to format our data before we build a second model. In R, we can `mutate()` our data. Notice how the intercept is now much closer to slope:

```
## R
gb_scores <-
    gb_scores |>
    mutate(Year_0 = Year - min(Year))

gb_score_year_0 <-
```

```r
    gb_scores |>
    lm(formula = GB ~ Year_0)
summary(gb_score_year_0)
```

```

Call:
lm(formula = GB ~ Year_0, data = gb_scores)

Residuals:
    Min      1Q  Median      3Q     Max 
-22.051  -5.051   0.788   4.889  21.949 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   22.051      2.036  10.831 1.02e-14 ***
Year_0         4.080      1.547   2.637   0.0111 *  
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.013 on 50 degrees of freedom
Multiple R-squared:  0.1221,	Adjusted R-squared:  0.1045 
F-statistic: 6.953 on 1 and 50 DF,  p-value: 0.01112
```

Likewise, we can wrangle our data in Python. With the Python outputs, also notice how the second note goes away compared to "Slopes and intercepts":

```python
## Python
gb_scores["Year_0"] = gb_scores["Year"] - gb_scores["Year"].min()
lm_gb_score_year_0 = smf.ols(formula = "GB ~ Year_0",
                           data = gb_scores).fit()
print(lm_gb_score_year_0.summary())
```

```
                            OLS Regression Results                            
==============================================================================
Dep. Variable:                     GB   R-squared:                       0.122
Model:                            OLS   Adj. R-squared:                  0.105
Method:                 Least Squares   F-statistic:                     6.953
Date:                Sat, 09 Jul 2022   Prob (F-statistic):             0.0111
Time:                        16:14:52   Log-Likelihood:                -187.10
No. Observations:                  52   AIC:                             378.2
Df Residuals:                      50   BIC:                             382.1
Df Model:                           1                                         
Covariance Type:            nonrobust                                         
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
```

```
--
Intercept     22.0510      2.036     10.831      0.000     17.962     26.140
Year_0         4.0805      1.547      2.637      0.011      0.972      7.189
==============================================================================
Omnibus:                        1.568   Durbin-Watson:                   2.423
Prob(Omnibus):                  0.456   Jarque-Bera (JB):                0.836
Skew:                          -0.255   Prob(JB):                        0.658
Kurtosis:                       3.353   Cond. No.                         3.05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly
 > specified.
```

In addition to these two transformations, we could also subtract by middle (median value) or other important number (for example start of a coach's tenure or other *era* in football).

Another type of transformation can be to *scale* (also known as standardizing or normalizing) the data. This changes the data from the raw scale to have a mean of 0 and standard deviation of 1. We can use the `scale()` function in R, or do calculation by hand in Python. To scale the data, the computer subtracts mean and then divides by the standard deviation.

```
## R
gb_scores <-
    gb_scores |>
    mutate(GB_normal = scale(GB))
gb_scores |> pull(GB_normal) |> head()
```

```
           [,1]
[1,] -0.2402671
[2,]  0.2846862
[3,] -0.4502484
[4,]  0.7046489
[5,]  0.4946676
[6,] -0.9752017
```

Notice how Python does not have a scale function in the core statistics packages so we must do our own transformation.

```
## Python
gb_scores["GB_normal"] = gb_scores["GB"]
gb_scores["GB_normal"] -= gb_scores["GB_normal"].mean()
gb_scores["GB_normal"] /= gb_scores["GB_normal"].std()
print(gb_scores["GB_normal"].head())
```

```
0   -0.240267
1    0.284686
2   -0.450248
3    0.704649
4    0.494668
Name: GB_normal, dtype: float64
```

When we compare the raw data to the transformed data, both look fairly normal, as seen in Figure 6-7. Hence, a transformation would not be needed in this case, and, in fact, make our example harder to follow.

```
images = [Image.open(x) for x in
          ["./images/hist_raw_bg_score.png",
           "./images/hist_raw_bg_normal.png"]]
widths, heights = zip(*(i.size for i in images))

total_width = sum(widths)
max_height = max(heights)

new_im = Image.new('RGB', (total_width, max_height))

x_offset = 0
for im in images:
  new_im.paste(im, (x_offset,0))
  x_offset += im.size[0]

new_im.save("./images/hist_raw_and_normal_gb_score.png")
```

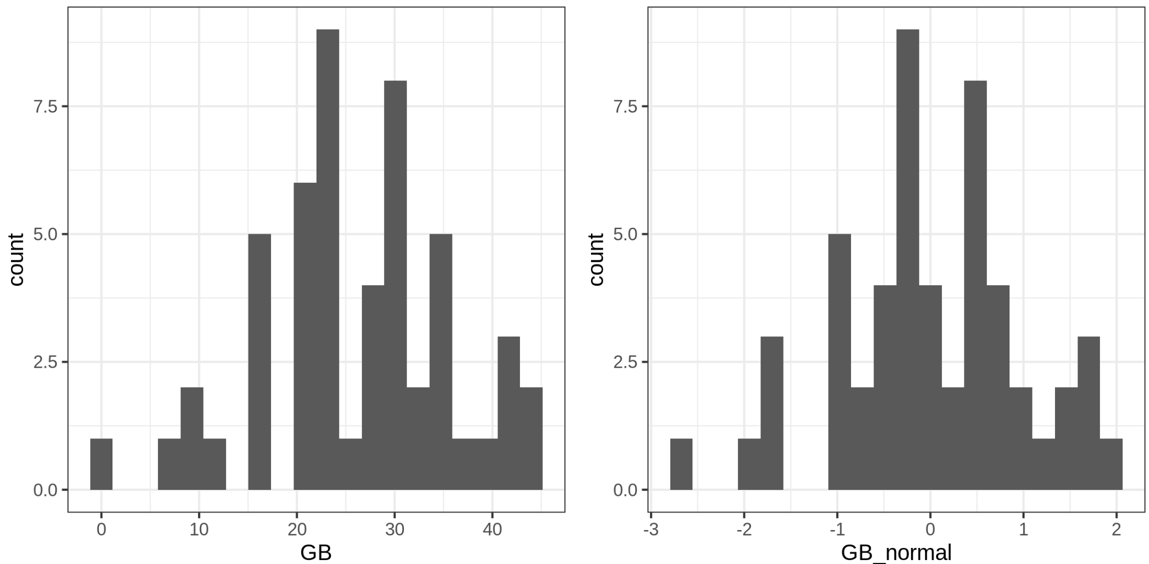

Figure 6-7. Histograms of raw and normalized score for Green Bay from their first game against Detroit during 2022.

```
## R
gb_score_year_0_normal <-
    gb_scores |>
    lm(formula = GB_normal ~ Year_0)
summary(gb_score_year_0_normal)


Call:
lm(formula = GB_normal ~ Year_0, data = gb_scores)

Residuals:
     Min       1Q   Median       3Q      Max
-2.31515 -0.53031  0.08273  0.51326  2.30444

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.4449     0.2137  -2.081   0.0425 *
Year_0        0.4284     0.1625   2.637   0.0111 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.9463 on 50 degrees of freedom
Multiple R-squared:  0.1221,    Adjusted R-squared:  0.1045
F-statistic: 6.953 on 1 and 50 DF,  p-value: 0.01112


## Python
lm_gb_score_year_0_normal = smf.ols(formula = "GB_normal ~ Year_0",
```

```
                                    data = gb_scores).fit()
print(lm_gb_score_year_0_normal.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:              GB_normal   R-squared:                       0.122
Model:                            OLS   Adj. R-squared:                  0.105
Method:                 Least Squares   F-statistic:                     6.953
Date:                Sat, 09 Jul 2022   Prob (F-statistic):             0.0111
Time:                        16:14:52   Log-Likelihood:                -69.895
No. Observations:                  52   AIC:                             143.8
Df Residuals:                      50   BIC:                             147.7
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------
--
Intercept     -0.4449      0.214     -2.081      0.043      -0.874      -0.016
Year_0         0.4284      0.162      2.637      0.011       0.102       0.755
==============================================================================
Omnibus:                        1.568   Durbin-Watson:                   2.423
Prob(Omnibus):                  0.456   Jarque-Bera (JB):                0.836
Skew:                          -0.255   Prob(JB):                        0.658
Kurtosis:                       3.353   Cond. No.                         3.05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly
 > specified.
```

Other transformations exist as well. For example the natural log and log-based 10 transformation may be used for data that have skew to the right. Other people like to use the square-root transformation. We use log transformation in our chemistry and population ecology research, but do not include them here because the outputs can be hard to explain.

Case Study: Score Through Years

Now that we have some tools, let’s put them together to talk about football. For example, maybe we and our sports buddies think the Green Bay

Packers have been scoring more points recently compared to a couple of years ago. Sure, we could argue about it, but we can also use statistical models to estimate if a trend is occurring. Using the tools we learned in Chapter 3, we obtain some data. Then, we use the tools from Chapter 4 to clean up the data. Now, we're ready to plot the data using tools from Chapter 2 and we create figures like Figure 6-8.

To start off, let's read in the data:

```
gb_scores_lm_plot <-
    ggplot(gb_scores, aes(x = Year_0, y = GB)) +
    geom_point() +
    stat_smooth(method = "lm", formula = y ~ x) +
    ylab("Green Bay score") +
    scale_x_continuous("Year (staring in 2018)", breaks = seq(1, 3, by = 1)) +
    theme_bw()
```

Next, we fit a model and format the outputs to create a forest plot. The R code might look like this (we use the word *might*, because there many different ways to read this code; the *best* one is the one you use and understand!):

```
## R
gb_score_year_0_coef <-
    tidy(gb_score_year_0, conf.int = TRUE) |>
    select(-p.value, - std.error, -statistic) |>
    pivot_longer(-c(term, conf.low, conf.high))
print(gb_score_year_0_coef)

gb_score_year_0_coef_plot <-
    ggplot(gb_score_year_0_coef,
           aes(x = term, y = value,
               ymin = conf.low, ymax = conf.high)) +
    geom_hline(yintercept = 0, size = 2, color = "red") +
    geom_point() +
    geom_linerange() +
    coord_flip() +
    theme_bw() +
    xlab("Parameter") +
    ylab("Estimate")
```

```
# A tibble: 2 × 5
  term        conf.low conf.high name     value
  <chr>          <dbl>     <dbl> <chr>    <dbl>
1 (Intercept)   18.0       26.1  estimate 22.1
2 Year_0         0.972      7.19 estimate  4.08
```

Likewise, we could create similar figures using Python and models using Python using this code. The plots of the regression coefficients (Figure 6-9) are sometimes called a forest plot (see Wikipedia article for an extended discussion). These plots quickly allow people to see differences for parameters from the model.

```
import seaborn as sns
import matplotlib.pyplot as plt
gb_score_plot = sns.regplot(x = "Year_0", y = "GB", data=gb_scores)
gb_score_plot.set_xlabel("Year (staring in 2018)")
gb_score_plot.set_ylabel("Green Bay score")
gb_score_plot.set_xticks([0, 1, 2])


images = [Image.open(x) for x in
          ["./images/gb_scores_lm_plot.png",
           "./images/python_regplot.png"]]
widths, heights = zip(*(i.size for i in images))

total_width = sum(widths)
max_height = max(heights)

new_im = Image.new('RGB', (total_width, max_height))

x_offset = 0
for im in images:
  new_im.paste(im, (x_offset,0))
  x_offset += im.size[0]

new_im.save("./images/gb_score_lm_plot_both.png")
```

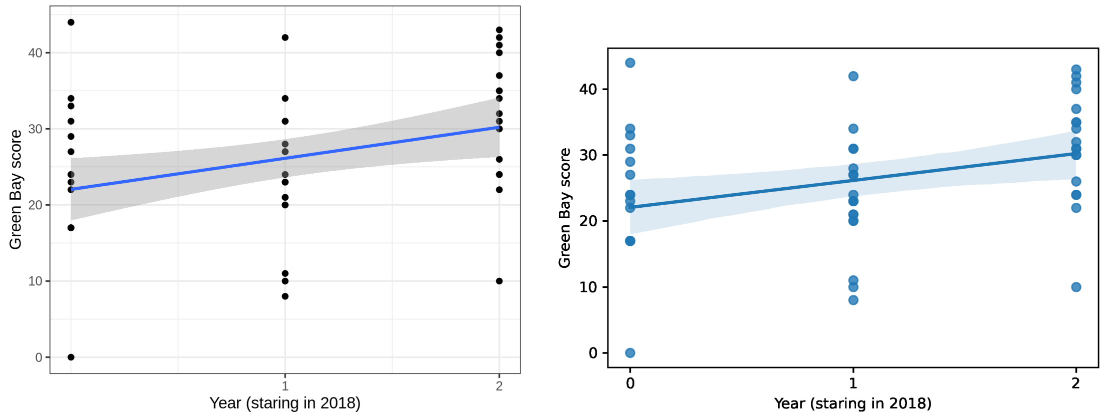

Figure 6-8. Regression plots with `ggplot2` *in R (left) and* `seaborn` *in Python (right).*

```
lm_gb_score_year_0_conf_int = lm_gb_score_year_0.conf_int(alpha=0.05)
lm_gb_score_year_0_parms = lm_gb_score_year_0.params


lm_all_coef = pd.concat([lm_gb_score_year_0_parms,
                         lm_gb_score_year_0_conf_int], axis=1)
lm_all_coef.columns = ["Estimate", "Lower", "Upper"]
lm_all_coef = lm_all_coef.rename_axis("Coef").reset_index()
print(lm_all_coef)
```

```
        Coef   Estimate      Lower      Upper
0  Intercept  22.051020  17.961872  26.140169
1     Year_0   4.080499   0.972268   7.188730
```

```
## Python
import matplotlib.pyplot as plt
coef_int = [lm_all_coef["Estimate"] - np.array(lm_all_coef["Lower"]),
            np.array(lm_all_coef["Upper"] - lm_all_coef["Estimate"])]

plt.errorbar(lm_all_coef["Estimate"],
             lm_all_coef["Coef"],
             xerr = coef_int,
             fmt = "o")
```

```
<ErrorbarContainer object of 3 artists>
![png](output_155_1.png)
```

```
images = [Image.open(x) for x in
          ["./images/gb_score_year_0_coef_plot.png",
           "./images/py_coef_plot.png"]]
widths, heights = zip(*(i.size for i in images))
```

```
total_width = sum(widths)
max_height = max(heights)

new_im = Image.new('RGB', (total_width, max_height))

x_offset = 0
for im in images:
  new_im.paste(im, (x_offset,0))
  x_offset += im.size[0]

new_im.save("./images/gb_score_year_0_coef_plot_both.png")
```

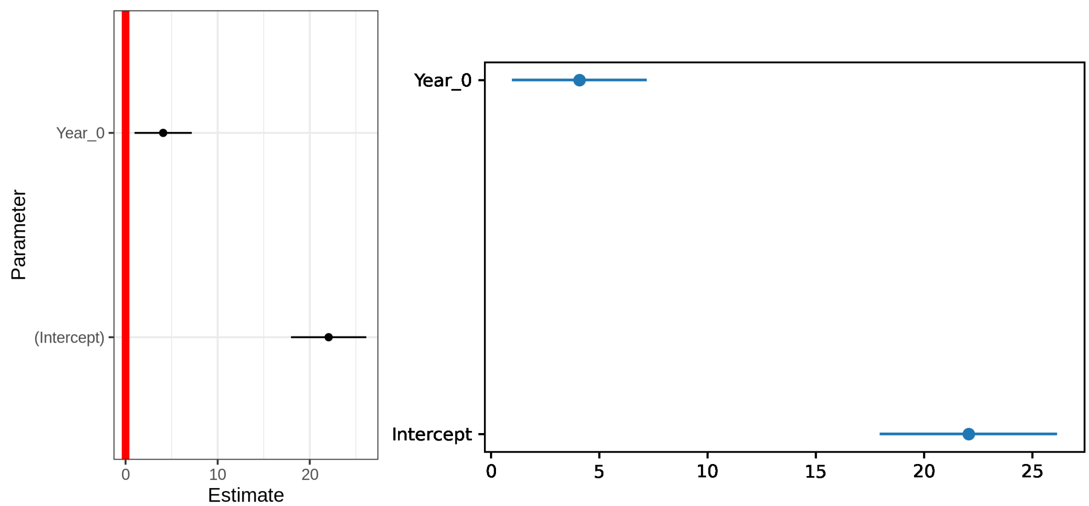

Figure 6-9. Regression coefficient plots with `ggplot2` in R (left) and `seaborn` in Python (right)

One thing to take into consideration, as we discussed above, is the fact that 2020 was played largely without fans in the stands, and as such road teams actually fared pretty well offensively relative to historical standards. Thus, when modeling this problem, one has to make sure that other factors are accounted for before jumping to causal conclusions about the game of football.

Exercises

1. Fit a linear model for `yards_after_the_catch` as the response, with `air_yards` as the feature, for completed passes. What do you find?

2. For the Draft Data scraped in Chapter 3, fit a linear model for draft position and `DrAV`. What are some issues that can arise when trying to approach this problem that way?
3. Transform `DrAV` in such a way so that a linear model with draft position as the feature fits the assumptions laid out in this chapter.
4. Merge the Draft data and the Scouting Combine data together. For wide receivers (`WR`) fit a linear model for 40-yard dash time and career receiving yards. Is there a positive relationship? What does this say about the efficiency of the scouting combine and finding good players at the wide receiver position? What if you do the same thing for vertical jump and sacks for defensive ends (`DE`)?
5. With the merging of the Draft data and Scouting Combine data in 4), fit a transformed linear model for 40-yard dash time and draft position for wide receivers. Compare this to your answer to 4). Do the same thing for vertical jump for defensive ends, and compare.

Further reading

Many books exist on regression.

Andrew Gelman, Jennifer Hill, and Aki Vehtari. Regression and Other Stories (2020; Cabridge University Press in 2020).

This book shows how to apply regression analysis to real world problems. For people looking for more *worked* case studies, we recommend this book to help you learn how to think about applying regression.

Frank Harell's Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis (2015, 2nd edition; Springer)

This book helped one of the authors think through the world of regression modeling. The books is advanced, but provides a good oversight into regression analysis. The book is written at an advanced undergraduate/introductory graduate-level. Although hard, working through this book provides mastery of regression analysis.

About the Authors

Eric A Eager is the Head of Research, Development and Innovation at Pro Football Focus (PFF), where he uses his training as an applied mathematician to produce solutions to quantitative problems for 32 National Football League clients, over 105 NCAA Football clients and numerous media clients and contacts. He also co-hosts the PFF Forecast Podcast, which can be found on PodcastOne and iTunes and is the most popular football analytics podcast in the world since 2018. Additionally, Eager supplies odds used by Steve Kornacki on Football Night in America, the Today Show, and other programs since 2020.

He studied applied mathematics and mathematical biology at the University of Nebraska, where he wrote his PhD thesis on how stochasticity and nonlinear processes affect population dynamics. Eager spent his first six years thereafter as a professor at the University of Wisconsin - La Crosse, before transitioning to PFF full-time in 2018. He has since taught statistics and mathematics to over 10,000 students through college-level courses, the Wharton Sports Analytics and Business Initiative's Moneyball Academy, as well as an online course, "Linear Algebra for Data Science in R" with DataCamp.

Eager has been interviewed by nfl.com's Ian Rappoport about Cowboys in-game decision making and The Washington Post for commentary about sports analytics. He joined the legendary Peter King's podcast about fourth-down decisions and is a frequent guest on Cris Collinsworth's podcast.

Richard A Erickson helps people use mathematics and statistics to understand our world as well as make decisions with this data. He is a lifelong Green Bay Packer fan, and, like thousands of other cheeseheads, a team owner. He has taught over 25,000 students statistics through graduate-level courses, workshops, and his DataCamp courses on Generalized Linear Models in R and Hierarchical Models in R. He also uses Python on a regular basis to model scientific problems.

Erickson received his PhD in Environmental Toxicology with an applied math minor from Texas Tech where he wrote his dissertation on modeling population-level effects of pesticides. He has modeled and analyzed diverse datasets including topics such as soil productivity for the USDA, impacts of climate change on disease dynamics, and improving rural healthcare. Erickson currently works as a research scientist and has over 70 peer-reviewed publications. Besides teaching Eric about R and Python, he also taught Eric to like cheese curds.