Learn to Code with Fantasy Football

v0.5.2

Contents

Prerequisites: Tooling

Python

In this book, we will be working with Python, a free, open source programming language.

The book is heavy on examples, which you should run and explore as you go through the book. To do so, you need the ability to run Python 3 code and install packages. If you can do that and have a setup that works for you, great. If you do not, the easiest way to get one is from Anaconda.

1. Go to: https://www.anaconda.com/products/individual
2. Scroll (way) down and click on the button under Anaconda Installers to download the 3.x version (3.8 at time of this writing) for your operating system.

Figure 0.1: Python 3.x on the Anaconda site

Note: if you're on windows, you probably have a 64-bit system, but if you're not sure here's how to check .

3. Then install it[1]. It might ask whether you want to install it for everyone on your computer or just you. Installing it for just yourself is fine.

[1]One thing about Anaconda is that it takes up a lot of disk space. This shouldn't be a big deal. Most computers have much

4. Once you have Anaconda installed, open up Anaconda Navigator and launch Spyder.
5. Then, in Spyder, go to View -> Window layouts and click on Horizontal split. Make sure pane selected (in blue) on the right side is 'IPython console'

Now you should be ready to code. Your editor is on left, and your Python console is on the right. Let's touch on each of these briefly.

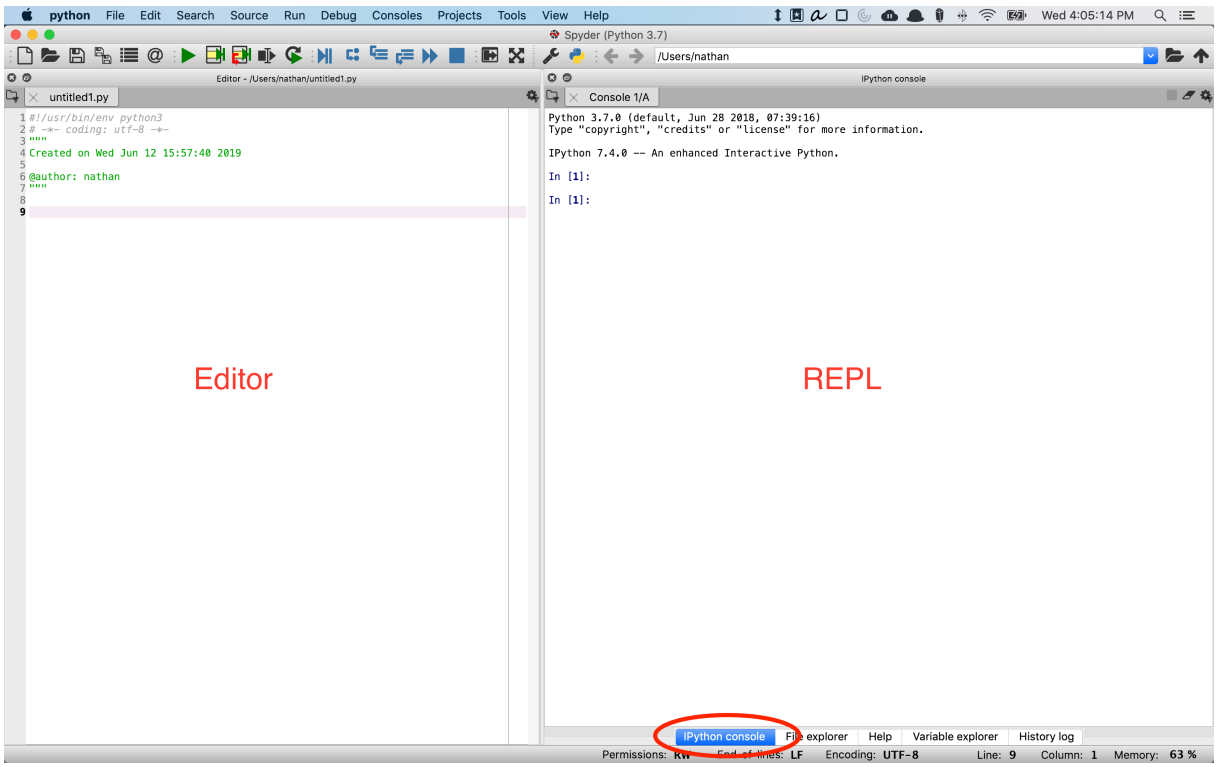


Figure 0.2: Editor and REPL in Spyder

more hard disk space than they need and using it will not slow your computer down. Once you are more familiar with Python, you may want to explore other, more minimalistic ways of installing it.

Editor

This book assumes you have some familiarity working in a spreadsheet program like Excel, but not necessarily any familiarity with code.

What are the differences?

A spreadsheet lets you manipulate a table of data as you look at. You can point, click, resize columns, change cells, etc. The coder term for this style of interaction is "what you see is what you get" (WYSIWYG).

Python code, in contrast, is a set of instructions for working with data. You tell your program what to do, and Python does (executes) it.

It is possible to tell Python what to do one instruction at a time, but usually programmers write multiple instructions out at once. These instructions are called "programs" or "code", and are just plain text files with the extension .py.

When you tell Python to run some program, it will look at the file and run each line, starting at the top.

Your editor is the text editing program you use to write and edit these files. If you wanted, you could write all your Python programs in Notepad, but most people don't. An editor like Spyder will do nice things like highlight special, Python related keywords and alert you if something doesn't look like proper code.

Console (REPL)

Your editor is the place to type code. The place where you actually run code is in what Spyder calls the IPython console. The IPython console is an example of what programmers call a read-eval(uate)-print-loop, or **REPL**.

A REPL does exactly what the name says, takes in ("reads") some code, evaluates it, and prints the result. Then it automatically "loops" back to the beginning and is ready for some new code.

Try typing 1+1 into it. You should see:

```
In [1]: 1 + 1
Out[1]: 2
```

The REPL "reads" `1 + 1`, evaluates it (it equals 2), and prints it. The REPL is then ready for new input.

A REPL keeps track of what you have done previously. For example if you type:

```
In [2]: x = 1
```

And then later:

```
In [3]: x + 1
Out[3]: 2
```

the REPL prints out `2`. But if you quit and restart Spyder and try typing `x + 1` again it will complain that it doesn't know what `x` is.

```
In [1]: x + 1
...
NameError: name 'x' is not defined
```

By Spyder "complaining" I mean that Python gives you an **error**. An error — also sometimes called an **exception** — means something is wrong with your code. In this case, you tried to use `x` without telling Python what `x` was.

Get used to exceptions, because you'll run into them a lot. If you are working interactively in a REPL and do something against the rules of Python it will alert you (in red) that something went wrong, ignore whatever you were trying to do, and loop back to await further instructions like normal.

Try:

```
In [2]: x = 1

In [3]: x = 9/0
...

ZeroDivisionError: division by zero
```

Since dividing by 0 is against the laws of math[2], Python won't let you do it and will throw (raise) an error. No big deal — your computer didn't crash and your data is still there. If you type `x` in the REPL again you will see it's still 1.

We'll mostly be using Python interactively like this, but know Python behaves a bit differently if you have an error in a file you are trying to run all at once. In that case Python will stop executing the file, but because Python executes code from top to bottom everything above the line with your error will have run like normal.

[2] See https://www.math.toronto.edu/mathnet/questionCorner/nineoverzero.html

Using Spyder and Keyboard Shortcuts

When writing programs (or following along with the examples in this book) you will spend a lot of your time in the editor. You will also often want to send (run) code — sometimes the entire file, usually just certain sections — to the REPL. You also should go over to the REPL to examine certain variables or try out certain code.

At a minimum, I recommend getting comfortable with the following keyboard shortcuts in Spyder:

Pressing **F9** in the editor will send whatever code you have highlighted to the REPL. If you don't have anything highlighted, it will send the current line.

F5 will send the entire file to the REPL.

You should get good at navigating back and forth between the editor and the REPL. On Windows:

- **control + shift + e** moves you to the editor (e.g. if you're in the REPL).
- **control + shift + i** moves you to the REPL (e.g. if you're in the editor).

On a Mac, you use command instead of control:

- **command + shift + e** (move to editor).
- **command + shift + i** (move to REPL).

Anki

Remembering What You Learn

A problem with reading technical books is remembering everything you read. To help with that, this book comes with more than 300 flashcards covering the material. These cards are designed for **Anki**, a (mostly) free, open source *spaced repetition* flashcard program.

> "The single biggest change that Anki brings about is that it means memory is no longer a haphazard event, to be left to chance. Rather, it guarantees I will remember something, with minimal effort. That is, Anki makes memory a choice." — Michael Nielsen

With normal flashcards, you have to decide when and how often to review them. When you use Anki, it takes care of this for you.

Anki is definitely optional. Feel free to dive in now and set it up later. But it may be something to consider, particularly if your learning is going to be spread out over a long time or you won't have a chance to use Python on a regular basis.

See Appendix B for more on Anki, installing it, and using the flashcards that come with this book.

1. Introduction

The Purpose of Data Analysis

The purpose of data analysis is to get interesting or useful insights.

- I want to win my fantasy football game, who do I start this week?
- I'm an NFL GM, which QB should I draft?
- I'm a mad scientist, how many yards would Priest Holmes have run behind Dallas's 2017 offensive line?

The process of doing data analysis is one — accurate and consistent — way to get these insights.

Of course, that requires *data*.

What is Data?

At a very high level, data is a *collection of structured information*.

You might have data about anything, but let's take a football game, say Week 12, 2018 Chiefs vs Rams. What would a collection of structured information about it look like?

Let's start with **collection**, or "a bunch of stuff." What is a football game a collection of? How about *plays*? This isn't the only acceptable answer — a collection of players, teams, drives, quarters would fit — but it'll work. A football game is a collection of plays. OK.

Now **information** — what information could we have about a play in this collection? Maybe: down, distance, what team has the ball, whether the play was a run or pass, how many yards it went for, or who made the tackle.

Finally, it's **structured** as a big rectangle with columns and rows. A row is a single item in our collection (a play here). A column is one piece of information (down, distance, etc).

This is an efficient, organized way of presenting information. When we want to know, "who had the first carry in the second half and how many yards did it go for?", we can find the right row and columns, and say "Oh, Kareem Hunt for 3 yards".

```
off  def  qtr      time        type  yards     rusher
 KC   LA    3  14:55:00        pass     27        NaN
 KC   LA    3  14:40:00         run      3     K.Hunt
 KC   LA    3  14:01:00         run      6  P.Mahomes
 KC   LA    3  13:38:00        pass    -16        NaN
 LA   KC    3  13:27:00        pass      0        NaN
 LA   KC    3  13:22:00        pass      0        NaN
```

It's common to refer to rows as **observations** and columns as **variables**, particularly when using the data for more advanced forms of analysis, like modeling. Other names for this rectangle-like format include tabular data or flat file. And it makes sense — all this info about the Chiefs-Rams game is flattened out into one big table.

A spreadsheet program like Microsoft Excel is one way to store data like this, but it's proprietary and may not always be available. Spreadsheets also often contain extra, non-data material, like annotations, highlighting or plots.

A simpler, more common way to store data is in a plain text file, where each row is a line and columns are separated by commas. So you could open up our play by play data in Notepad and see:

```
off,def,qtr,time,type,yards,rusher
KC,LA,3,14:55:00,pass,27,
KC,LA,3,14:40:00,run,3,K.Hunt
KC,LA,3,14:01:00,run,6,P.Mahomes
KC,LA,3,13:38:00,pass,-16,
LA,KC,3,13:27:00,pass,0,
LA,KC,3,13:22:00,pass,0,
```

Data stored like this, with a character (usually a comma, sometimes a tab) in between columns is called *delimited* data. Comma delimited files are called **comma separated values** and usually have a csv extension.

This is just how the data is *stored* on your computer. No one expects you to work with all those commas and open up the plain text file directly. In Excel (or other spreadsheet program) you can open and write csv files and they will be in the familiar spreadsheet format. In fact, that's one of the main benefits of using csvs — most programs can read them.

Example Datasets

This book is heavy on examples, and comes with a few small csv files that we will practice on.

Play-By-Play Data

The first dataset is play-by-play data from two games. The first is what ESPN's Bill Barnwell called the greatest regular season game ever played — the Rams 54-51 victory over the Chiefs in Week 12, 2018. The second is the Patriots 43-40 victory over Kansas City. This game was another classic, including 30 points in the 4th quarter alone.

This play-by-play data includes both pre-snap, situational information about each play (game, quarter, time left, yardline, down, yards to go, team with the ball) and what actually happened: play type (run, pass, field goal, punt), yards_gained, pass_player_name, receiver_player_name, etc.

The dataset includes all regular downed plays — no extra points, kickoffs, or plays nullified by penalties. It's in the file play_data_sample.csv.

Player/Game Data

The second dataset this book comes with is a random sample of player-game totals from 2017. It's in the file player_game_2017_sample.csv. The first few rows (and some of the columns, they don't all fit here) look like this:

player_name	week	carries	...	rec_yards	receptions	targets
T.Brady	1	0.0	...	0.0	0.0	0.0
A.Smith	1	2.0	...	0.0	0.0	0.0
D.Amendola	1	0.0	...	100.0	6.0	7.0
R.Burkhead	1	3.0	...	8.0	1.0	3.0
T.Kelce	1	1.0	...	40.0	5.0	7.0

Remember: *Collection*. *Information*. *Structure*.

This data is a *collection* of player-game combinations. Each row represents one player's statistics for one game (Amendola/Week 1). If we had 100 players, and they each played 16 games, then our dataset would be 100*16=1600 rows long.

The columns (variables) are *information*. In the third row, each column tells us something about how Amendola did Week 1, 2017. The `targets` column shows us he had 7 targets, the `receptions` column indicates he caught 6 of them, and the `rec_yards` column says those catches went for 100 yards.

If we want to look at another player-game (Amendola/Week 2 or Gronk/Week 8), we look at a different row.

Notice how our columns can be different types of information, like text (`player_name`) or numbers (`rec_yards`) or semi-hybrid, technically-numbers but we would never do any math with them (`gameid`).

One thing to keep in mind: just because our data is at some level (player/game), doesn't mean *every* column in the data has to be at that level.

Though you can't see it in the snippet above, in this dataset, there is a column called `season`. It's always `2017`. Does this matter? No. We're just asking: "for this particular player/game result, what season was it?" It just happens that for this dataset the answer is the same for every row.

ADP Data

The third dataset is the 2017 pre-season average draft position (ADP) data from www.fantasyfootballcalculator.com. It's in the file adp_2017.csv. It is at the *player* level and looks like this:

```
           name team    adp  ...  high  low  times_drafted
  David Johnson  ARI    1.3  ...     1    4            310
    LeVeon Bell  PIT    2.3  ...     1    6            303
  Antonio Brown  PIT    3.7  ...     1    7            338
    Julio Jones  ATL    5.7  ...     1   15            131
Ezekiel Elliott  DAL    6.2  ...     1   17            180
```

Later in the book you will learn how to build your own webscraper to get this or data from any other year or scoring system.

I encourage you to open all of these datasets up and explore them in your favorite spreadsheet program.

Now that we know what data is, let's move to the other end of the spectrum and talk about why data is ultimately valuable, which is because it provides insights.

What is Analysis?

How many footballs are in the following picture?

Figure 0.1: Few footballs

Pretty easy question right? What about this one?

Figure 0.2: Many footballs

Researchers have found that humans automatically know how many objects they're seeing, as long as there are no more than three or four. Any more than that, and counting is required.

If you open up the play by play data this book comes with, you'll notice it's 304 rows and 42 columns.

From that, do you think you would be able to open it and immediately tell me who the "best" player was? Worst? Most consistent or explosive? Of course not.

Raw data is the numerical equivalent of a pile of footballs. It's a collection of facts, way more than the human brain can reliably and accurately make sense of and meaningless without some work.

Data analysis is the process of transforming this raw data to something smaller and more useful you can fit in your head.

Types of Data Analysis

Broadly, it is useful to think of two types of analysis, both of which involve reducing a pile of data into a few, more manageable number of insights.

1. The first type of statistical analysis is calculating single number type summary statistics. Examples would include the mean (average), median, mode, quarterback rating (QBR) and wins over replacement player (WORP).
2. The second type of analysis is building *models* to understand relationships between data.

Summary Statistics

Summary statistics can be complex (QBR, WORP) or more basic (yards per catch or games missed due to injury), but all of them involve going from raw data to some more useful number.

The main goal of these single number summary statistics is usually to better understand things that happened previously.

Stats vary in ambition. Some, like QBR, try to be all encompassing, while others get at a particular facet of a player's performance. For example, we might measure "explosiveness" by looking at a player's longest carry, or "steadiness" by looking at the percentage of carries that don't lose yards. These are summary stats.

A key skill in data analysis is knowing how to look at data multiple ways via different summary statistics, keeping in mind their strengths and weaknesses. Knowing how to do this well can give you an edge.

For example, in the 2003 book *Moneyball*, Michael Lewis writes about how the Oakland A's were one of the first baseball teams to realize that batting average — which most teams relied on to evaluate a hitter's ability at the time — did not take into account a player's ability to draw walks.

By using a different statistic — on base percentage — that *did* take walks into account and signing players with high on base percentage relative to their batting average, the A's were able to get good players at a discount, and had a lot of success even though they didn't spend that much money[1].

In practice, calculating summary stats requires creativity, clear thinking and the ability to manipulate data via code. The latter is a big focus in this book.

[1] It didn't take long for other teams to catch on. Now, on base percentage is a standard metric and the discount on players with high OBP relative to BA has largely disappeared.

Modeling

The other type of analysis is *modeling*. A model describes a mathematical *relationship* between variables in your data, specifically the relationship between one or more *input variables* and one *output variable*.

output variable = model(*input variables*)

This is called "modeling *output variable* as a function of *input variables*".

How do we find these relationships and actually "do" modeling in practice?

Remember, when working with flat, rectangular data, *variable* is just another word for *column*. In practice, modeling is making a dataset where the columns are your input variables and one output variable, then passing this data (along with info about which column is the output and which are the inputs) to your Python modeling program.

That's why most data scientists and modelers spend most of their time collecting and manipulating data. Getting all your inputs and output together in a dataset that your modeling program can accept is most of the work.

Later in the book we will get into the technical details and learn how to actually use some of these modeling programs, but for now let's get into the motivation for modeling.

Why Model?

Though all models describe some relationship, *why* you might want to analyze a relationship depends on the situation. Sometimes, it's because models help make sense of things that have already happened and the relationships between them.

For example, modeling rushing yards as a function of the player, offensive line skill and game flow (e.g. down and distance or score), would be one way to figure out — by separating talent from other factors outside of the running back's control — which running back is the most "skilled".

Then as an analyst for an NFL team assigned to scout free agent running backs, I can report back to my GM on who my model thinks is truly the best, the one who will run for the most yards behind our offensive line.

Models that Predict the Future

Often, we want to use a model to predict what will happen in the future, say, next season's fantasy points or this week's daily fantasy ownership percentage.

We know modeling is about relationships, for prediction-type models that relationship is between data we have *now* and events that will happen in the *future*.

But if something hasn't happened yet, how can we relate it to the present? The answer is by starting with the past.

For example, I'm writing this in the summer of 2019, which means I have 2018 data on: ADP; weather; and actual fantasy points scored. And I could build a model:

player's points for some week in 2018 = model(*player's 2018 ADP*, *weather for that week in 2018*)

Training (or *fitting*) this model is the process of using that known/existing/already happened data to find a relationship between the input variables (*ADP*, *weather*) and the output variable (*fantasy points scored*). Once I establish that relationship, I can feed it new inputs — an ADP of 1.5, rainy with 20 mph winds — transform it using my relationship, and get back an output — 18.7 points.

The inputs I feed my model might be from past events that have already happened. This is often done to evaluate model performance. For example, I could put in Gurley's 2018 ADP and the weather for Week 1, 2018, and see what the model says. Hopefully it's close to how he actually did that week[2].

Alternatively, I can feed it data from *right now* in order to predict things that *haven't happened yet*. For example — again, I'm writing this in the summer of 2019 — I can put in Gurley's 2019 ADP, take a guess at the weather and get back projected points for Week 1.

High Level Data Analysis Process

Now that we've covered both the inputs (data) and final outputs (analytical insights), let's take a very high level look at what's in between.

Everything in this book will fall somewhere in one of the following steps:

1. Collecting Data

Whether you scrape a website, connect to a public API, download some spreadsheets, or enter it yourself, you can't do data analysis without data. The first step is getting ahold of some.

This book covers how to scrape a website and get data by connecting to an API. It also suggests a few ready-made datasets.

[2] The actual values for many models are picked so that this difference — called the *residual* — is as small as possible across all observations.

2. Storing Data

Once you have data, you have to put it somewhere. This could be in several spreadsheet or text files in a folder on your desktop, sheets in an Excel file, or a database.

This book covers the basics and benefits of storing data in a SQL database.

3. Loading Data

Once you have your data stored, you need to be able to retrieve the parts you want. This can be easy if it's in a spreadsheet, but if it's in a database then you need to know some SQL — pronounced "sequel" and short for Structured Query Language — to get it out.

This book cover basic SQL and loading data with Python.

4. Manipulating Data

Talk to any data scientist, and they'll tell you they spend the most of their time preparing and manipulating their data. Football data is no exception. Sometimes called munging, this means getting your raw data in the right format for analysis.

There are many tools available for this step. Examples include Excel, R, Python, Stata, SPSS, Tableau, SQL, and Hadoop. In this book you'll learn how to do it in Python, particularly using the library Pandas.

The boundaries between this step and the ones before and after it can be a little fuzzy. For example, though we won't do it in this book, it is possible to do some basic manipulation in SQL. In other words, loading (3) and manipulating (4) data can be done with the same tools. Similarly Pandas — the primary tool we'll use for data manipulation (4) — also includes basic functionality for analysis (5) and input-output capabilities (3).

Don't get too hung up on this. The point isn't to say, "this technology is *always* associated with this part of the analysis process". Instead, it's a way to keep the big picture in mind as you are working through the book and your own analysis.

5. Analyzing Data for Insights

This step is the model, summary stat or plot that takes you from formatted data to insight.

This book covers a few different analysis methods, including summary stats, a few modeling techniques, and data visualization.

We will do these in Python using the scikit-learn, statsmodels, and matplotlib libraries, which cover machine learning, statistical modeling and data visualization respectively.

Connecting the High Level Analysis Process to the Rest of the Book

Again, everything in this book falls into one of the five sections above. Throughout, I will tie back what you are learning to this section so you can keep sight of the big picture.

This is the forest. If you ever find yourself banging your head against a tree, confused or wondering *why* a certain topic is covered, refer back to here and think about where it fits in.

Some sections above may be more applicable to you than others. Perhaps you are comfortable analyzing data in Excel, and just want to learn how to get data via scraping a website or connecting to an API. Feel free to focus on whatever sections are most useful to you.

End of Chapter Exercises

1.1

Name the granularity for the following, hypothetical datasets:

a)

game_id	qtr	posteam	score	yards	passes	turnovers
2018101412	1	KC	3.0	66	9.0	1.0
2018101412	1	NE	10.0	133	9.0	0.0
2018101412	2	KC	9.0	144	14.0	1.0
2018101412	2	NE	24.0	57	7.0	0.0
2018101412	3	KC	19.0	155	8.0	0.0

b)

gameid	player_name	rush_yards	receptions	pass_yards
2017100105	D.Murray	31.0	2.0	0.0
2017090700	T.Brady	0.0	0.0	267.0
2017111201	R.Cobb	8.0	3.0	0.0
2017092408	M.Forte	25.0	0.0	0.0
2017091701	C.Clay	0.0	3.0	0.0

c)

player_name	ngames	rush_yards	receptions	pass_yards	season
A.Collins	15	976.0	23.0	0.0	2017
A.Cooper	14	4.0	47.0	0.0	2017
A.Derby	11	0.0	21.0	0.0	2017
A.Robinson	14	0.0	19.0	0.0	2017
B.Bortles	16	326.0	0.0	3716.0	2017

d)

	pass_tds															
week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
player_name																	
C.Newton	2	0	0	3	3	1	1	1	0	4	0	0	2	1	4	0	1
T.Taylor	2	0	2	1	1	0	1	1	3	0	1	1	0	0	1	1	1
T.Brady	0	3	5	2	1	2	2	1	0	3	3	4	0	1	1	3	2
M.Ryan	1	1	3	1	0	1	1	2	2	2	2	1	0	1	0	1	2
B.Bortles	1	1	4	1	0	1	1	0	1	1	1	0	2	2	3	3	0

e)

position	adp_mean	adp_std	adp_min	adp_25%	adp_50%	adp_75%	adp_max
WR	75.52	47.05	3.7	36.55	70.0	116.450	162.9
RB	78.25	50.45	1.3	33.25	76.6	121.550	160.7
TE	92.90	43.44	18.1	57.60	83.6	132.600	152.7
QB	96.22	39.58	23.4	74.20	100.3	126.500	162.4
DEF	138.92	21.16	105.5	123.90	138.1	161.500	168.2
PK	160.42	9.68	145.0	154.20	161.8	168.275	172.7

1.2

You're an analyst for a team in need of some playmakers. Your GM says he wants someone who is "explosive" with "big play potential". He's tasked you with evaluating the crop of FA WRs.

What are some simple summary stats you might look at?

1.3

I want to build a model that uses weather data (wind speed, temperature at kickoff) to predict a game's combined over-under. What are my:

a) Input variables.

b) Output variable.

c) What level of granularity is this model at?

d) What's the main limitation with this model?

1.4

List where each of the following techniques and technologies fall in the high-level pipeline.

a) getting data to the right level of granularity for your model
b) experimenting with different models
c) dealing with missing data
d) SQL
e) scraping a website
f) plotting your data
g) getting data from an API
h) pandas
i) taking the mean of your data
j) combining multiple data sources

2. Python

Introduction to Python Programming

This section is an introduction to basic Python programming.

In this book, we will use Python for all parts of the data analysis process: collecting; storing; loading; manipulating and analyzing. Much of the functionality for this comes from third party **libraries** (or **packages**), specially designed for specific tasks.

For example: the library Pandas lets us manipulate tabular data. And the library BeautifulSoup is the Python standard for scraping data from websites.

We'll write code that makes heavy use of both packages (and more) later in the book but — even when using packages — you will also be using a core set of Python features and functionality. These features — called the **standard library** — are built-in to Python and don't require installing any external packages.

This section of the book covers the parts of the standard library that are most important. All the Python code we write in this book is built upon the concepts covered in this chapter. Since we'll be using Python for nearly everything, this section touches all parts of the high level, five-step data analysis process.

How to Read This Chapter

This chapter — like the rest of the book — is heavy on examples. All the examples in this chapter are included in the Python file `02_python.py`. Ideally, you would have this file open in your Spyder editor and be running the examples (highlight the line(s) you want and press F9 to send it to the REPL/console) as we go through them in the book.

If you do that, I've included what you'll see in the REPL here in the book. That is:

```
In [1]: 1 + 1
Out[1]: 2
```

Where the line starting with `In[1]` is what you send, and `Out[1]` is what the REPL prints out. These are lines `[1]` for me because this was the first thing I entered in a new REPL session. Don't worry if the numbers you see in `In[ ]` and `Out[ ]` don't match exactly what's in this chapter. In fact, they probably won't, because as you run the examples you should be exploring and experimenting. That's what the REPL is for.

Nor should you worry about messing anything up: if you need a fresh start, you can type `reset` into the REPL and it will clear out everything you've run previously. You can also type `clear` to clear all the printed output.

Sometimes, examples build on each other (remember, the REPL keeps track of what you've run previously), so if somethings not working, it might be relying on something you haven't run yet.

Let's get started.

Important Parts of the Python Standard Library

Comments

As you look at `02_python.py` you might notice a lot of lines beginning with `#`. These are **comments**. When reading (**interpreting**) your code, the computer (**interpretor**) will ignore everything from `#` to the end of the line.

Comments exist in all programming languages. They are a way to explain to anyone reading your code — including your future self — more about what is going on and what you were trying to do when you wrote the code.

The problem with comments is it's easy for them to become out of date — by, for example, changing your code and forgetting to update the comment.

An incorrect or misleading comment is worse than no comment. For that reason, most beginning programmers probably comment too often, especially because Python's **syntax** (the language related rules for writing programs) is usually pretty clear.

For example, this would be an unnecessary comment:

```
# print the result of 1 + 1
print(1 + 1)
```

Because it's not adding anything that isn't obvious by just looking at the code. It's better to use descriptive names, let your code speak for itself, and save comments for particularly tricky portions of code.

Variables

Variables are one of the most fundamental concepts in any programming language.

At their core, variables are just named pieces of information. This information can be anything from a single number to an entire dataset — the point is that they let you store and recall things easily.

The rules for naming variables differ by programming language. In Python, they can be any upper or lowercase letter, number or _ (underscore), but they can't start with a number.

One thing to watch out for: I mentioned previously how — in the language of modeling and tabular data — a variable is another word for column. That's different than what we're talking about here.

A variable in a dataset or model is a column; a variable in your code is named piece of information. You should usually be able to tell by the context which one you're dealing with. Unfortunately, imprecise language comes with the territory when learning new subjects, but I'll do my best to warn you about any similar pitfalls.

While you can name your variables whatever you want (provided it follows the rules), the **convention** in Python for most variables is all lowercase letters, with words separated by underscores.

Conventions are things, while not strictly required, programmers include to make it easier to read each other's code. They vary by language. So, while in Python I might have a variable `pts_per_passing_td`, a JavaScript programmer would write `ptsPerPassingTD` instead.

Assigning data to variables

You **assign** a piece of data to a variable with an equals sign, like this:

```
In [6]: pts_per_passing_td = 4
```

Another, less common, word for assignment is **binding**, as in `pts_per_passing_td` is bound to the number 4.

Now, whenever you use `pts_per_passing_td` in your code, the program automatically substitutes it with 4 instead.

```
In [8]: pts_per_passing_td
Out[8]: 4

In [9]: 3*pts_per_passing_td
Out[9]: 12
```

One of the benefits of developing with a REPL is that you can type in a variable, and the REPL will evaluate (i.e. determine what it is) and print it. That's what the code above is doing. But note while

`pts_per_passing_td` is `4`, the assignment statement itself, `pts_per_passing_td = 4`, doesn't evaluate to anything, so the REPL doesn't print anything out.

You can update and override variables too. Going into the code below, `pts_per_passing_td` has a value of `4` (from the code we just ran above). So the right hand side, `pts_per_passing_td - 3` is evaluated first (4 - 3 = 1), and *then* the result gets (re)assigned to `pts_per_passing_td`, overwriting the `4` it held previously.

```
In [12]: pts_per_passing_td = pts_per_passing_td - 3

In [13]: pts_per_passing_td
Out[13]: 1
```

Types

Like Excel, Python includes concepts for both numbers and text. Technically, Python distinguishes between two types of numbers: integers (whole numbers) and floats (numbers that may have decimal points), but the distinction between them isn't important for us right now.

```
over_under = 48  # int
wind_speed = 22.8  # float
```

Text, called a **string** in Python, is wrapped in either single (`'`) or double (`"`) quotes. I usually just use single quotes, unless the text I want to write has a single quote in it (like in Le'Veon), in which case trying to use `'Le'Veon Bell'` would give an error.

```
starting_qb = 'Tom Brady'
starting_rb = "Le'Veon Bell"
```

You can check the type of any variable with the type function.

```
In [20]: type(starting_qb)
Out[20]: str

In [21]: type(over_under)
Out[21]: int
```

Keep in mind the difference between strings (quotes) and variables (no quotes). Remember, a variable is named of a piece of information. A string (or a number) *is* the information.

One common thing to do with strings is to insert variables inside of them. The easiest way to do that is via **f-strings**.

```
In [4]: team_name = f'{starting_qb}, {starting_rb}, & co.'

In [5]: team_name
Out[5]: "Tom Brady, Le'Veon Bell, & co."
```

Note the `f` immediately preceding the quotation mark. Adding that tells Python you want to use variables inside your string, which you wrap in curly brackets.

f-strings are new as of the latest version of Python (3.8), so if they're not working for you make sure that's the version you're using.

Strings also have useful **methods** you can use to do things to them. You invoke methods with a `.` and parenthesis. For example, to make a string uppercase you can do:

```
In [6]: 'he could go all the way'.upper()
Out[6]: 'HE COULD GO ALL THE WAY'
```

Note the parenthesis. That's because sometimes these take additional data, for example the `replace` method.

```
In [7]: 'Chad Johnson'.replace('Johnson', 'Ochocinco')
Out[7]: 'Chad Ochocinco'
```

There are a bunch of these string methods, most of which you won't use that often. Going through them all right now would bog down progress on more important things. But occasionally you *will* need one of these string methods. How should we handle this?

The problem is we're dealing with a comprehensiveness-clarity trade off. And, since anything short of the Python in a Nutshell: A Desktop Quick Reference (which is 772 pages) is going to necessarily fall short on comprehensiveness, we'll do something better.

Rather than teaching you all 44 of Python's string methods, I am going to teach you how to quickly see which are available, what they do, and how to use them.

Though we're nominally talking about string methods here, this advice applies to any of the programming topics we'll cover in this book.

Interlude: How to Figure Things Out in Python

> "A simple rule I taught my nine year-old today: if you can't figure something out, figure out how to figure it out." — Paul Graham

The first tool you have up your sleeve when trying to figure out options available to you is the REPL. In particular, you have the REPL's **tab completion** functionality. Type in `'tom brady'.` and hit tab.

You'll see all the options available to you (this is only the first page, you'll see more if you keep pressing tab).

```
'tom brady'.
    capitalize()   encode()       format()
    isalpha()      isidentifier() isspace()
    ljust()        casefold()     endswith()
    format_map()   isascii()      islower()
```

Note: tab completion on a string directly like this doesn't always work in Spyder. If it's not working for you, assign `'tom brady'` to a variable and tab complete on that. Like this[1]:

```
In [14]: foo = 'tom brady'
Out[14]: foo.
            capitalize()   encode()       format()
            isalpha()      isidentifier() isspace()
            ljust()        casefold()     endswith()
            format_map()   isascii()      islower()
```

Then, when you find something you're interested in, enter it in the REPL with a question mark after it, like `'tom brady'.capitalize?` (or `foo.capitalize?` if that doesn't work).

You'll see:

```
Signature: str.capitalize(self, /)
Docstring:
Return a capitalized version of the string.

More specifically, make the first character have upper case and
the rest lower case.
```

So, in this case, it sounds like `capitalize` will make the first letter uppercase and the rest of the string lowercase. Let's try it:

```
In [15]: 'tom brady'.capitalize()
Out[15]: 'Tom brady'
```

Great. Many of the items you'll be working with in the REPL have methods, and tab completion is a great way to explore what's available.

The second strategy is more general. Maybe you want to do something that you know is string related but aren't necessarily sure where to begin or what it'd be called.

For example, maybe you've scraped some data that looks like:

[1]The upside of this Spyder autocomplete issue is you can learn about the programming convention "foo". When dealing with a throwaway variable that doesn't matter, many programmers will name it `foo`. Second and third variables that don't matter are `bar` and `baz`. Apparently this dates back to the 1950's.

```
In [8]: '  tom brady'
```

But you want it to be like this, i.e. without the spaces before "tom":

```
In [9]: 'tom brady'
```

Here's what you should do — and I'm not trying to be glib here — Google: "python string get rid of leading white space".

When you do that, you'll see the first result is from stackoverflow and says:

> "The lstrip() method will remove leading whitespaces, newline and tab characters on a string beginning."

A quick test confirms that's what we want.

```
In [99]: '  tom brady'.lstrip()
Out[99]: 'tom brady'
```

Stackoverflow

Python — particularly a lot of the data libraries we'll be using — became popular during the golden age of stackoverflow, a programming question and answer site that specializes in answers to small, self-contained technical problems.

How it works: people ask questions related to programming, and other, more experienced programmers answer. The rest of the community votes, both on questions ("that's a very good question, I was wondering how to do that too") as well as answers ("this solved my problem perfectly"). In that way, common problems and the best solutions rise to the top over time. Add in the power of Google's search algorithm, and you usually have a way to figure out exactly how to do most anything you'll want to do in a few minutes.

You don't have to ask questions yourself or vote or even make a stackoverflow account to get the benefits. In fact, most people probably don't. But enough people do, especially when it comes to Python, that it's a great resource.

If you're used to working like this, this advice may seem obvious. Like I said, I don't mean to be glib. Instead, it's intended for anyone who might mistakenly believe "real" coders don't Google things.

As programmer-blogger Umer Mansoor writes,

> Software developers, especially those who are new to the field, often ask this question… Do experienced programmers use Google frequently?

> The resounding answer is YES, experienced (and good) programmers use Google… a lot. In fact, one might argue they use it more than the beginners. [that] doesn't make them bad programmers or imply that they cannot code without Google. In fact, truth is quite the opposite: Google is an essential part of their software development toolkit and they know when and how to use it.

> A big reason to use Google is that it is hard to remember all those minor details and nuances especially when you are programming in multiple languages… As Einstein said: 'Never memorize something that you can look up.'

Now you know how to figure things out in Python. Back to the basics:

Bools

There are other data types besides strings and numbers. One of the most important ones is **bool** (for boolean). Boolean's — which exist in every language — are for binary, yes or no, true or false data. While a string can have almost an unlimited number of different values, and an integer can be any whole number, bools in Python only have two possible values: `True` or `False`.

Similar to variable names, bool values lack quotes. So `"True"` is a string, not a bool. For that reason, it's also a good idea to avoid using True or False as variable names.

A Python expression (any number, text or bool) is a bool when it's yes or no type data. For example:

```
# some numbers to use in our examples
In [23]: team1_pts = 110

In [24]: team2_pts = 120

# these are all bools:
In [25]: team1_won = team1_pts > team2_pts

In [26]: team2_won = team1_pts < team2_pts

In [27]: teams_tied = team1_pts == team2_pts

In [28]: teams_did_not_tie = team1_pts != team2_pts

In [29]: type(team1_won)
Out[29]: bool

In [30]: teams_did_not_tie
Out[30]: True
```

Notice the == by `teams_tied`. That tests for equality. It's the double equals sign because — as we learned above — Python uses the single = to assign to a variable. This would give an error:

```
In [31]: teams_tied = (team1_pts = team2_pts)
  File IPython Input, line 1
    teams_tied = (team1_pts = team2_pts)
                            ^
SyntaxError: invalid syntax
```

So `team1_pts == team2_pts` will be `True` if those numbers are the same, `False` if not. Similarly, `!=` means "not equal" and `team1_pts != team2_pts` will be `False` if the numbers *are* the same, `True` otherwise.

You can manipulate bools — i.e. chain them together or negate them — using the keywords `and`, `or`, `not` and parenthesis.

```
In [32]: shootout = (team1_pts > 150) and (team2_pts > 150)

In [33]: at_least_one_good_team = (team1_pts > 150) or (team2_pts > 150)

In [34]: you_guys_are_bad = not ((team1_pts > 100) or (team2_pts > 100))

In [35]: meh = not (shootout or at_least_one_good_team or you_guys_are_bad
    )
```

if statements

Bools are used frequently; one place is with if statements. The following code assigns a string to a variable `message` depending on what happened.

```
In [17]: if team1_won:
    ...:   message = "Nice job team 1!"
    ...: elif team2_won:
    ...:   message = "Way to go team 2!!"
    ...: else:
    ...:   message = "must have tied!"

In [18]: message
Out[18]: 'Way to go team 2!!'
```

Notice how in the code I'm saying **`if`** `team1_won`, not **`if`** `team1_won == True`. While the latter would technically work, it's a good way to show anyone looking at your code that you don't really understand bools. `team1_won` is `True`, it's a bool. `team1_won == True` is also `True`, and it's still a bool. Similarly, don't write `team1_won == False`, write `not team1_won`.

Container Types

Strings, integers, floats, and bools are called **primitives**; they're the basic building block types.

There are other **container** types that can hold other values. Two important container types are **lists** and **dicts**. Sometimes containers are also called **collections**.

Lists

Lists are built with square brackets and are basically a simple way to hold other, ordered pieces of data.

```
In [43]: my_roster_list = ['tom brady', 'adrian peterson', 'antonio brown'
    ]
```

Every spot in the list has a number associated with it, and the first spot is `0`. You can get sections (called slices) of your list by separating numbers with a colon. These are called inside `[]` (brackets). A single integer inside a bracket returns one element of your list, while a slice returns a smaller list. Note a slice returns *up to* the end of it, so `[0:2]` returns the `0` and `1` items, not `2`.

```
In [44]: my_roster_list[0]
Out[44]: 'tom brady'

In [45]: my_roster_list[0:2]
Out[45]: ['tom brady', 'adrian peterson']
```

Passing a slice a negative number gives you the end of the list. To get the last two items you could do:

```
In [45]: my_roster_list[-2:]
Out[45]: ['adrian peterson', 'antonio brown']
```

Also note how when you leave off the number after the colon the slice will automatically use the end of the list.

Lists can hold anything, including other lists. Lists that hold other lists are often called *nested* lists.

Dicts

A dict is short for dictionary. You can think about it like an actual dictionary if you want. Real dictionaries have words and definitions, Python dicts have **keys** and **values**.

Dicts are basically a way to hold data and give each piece a name. Dicts are written with curly brackets, like this:

```
In [46]: my_roster_dict = {'qb': 'tom brady',
    ...:                   'rb1': 'adrian peterson',
    ...:                   'wr1': 'antonio brown'}
```

You can access items in a dict like this:

```
In [47]: my_roster_dict['qb']
Out[47]: 'tom brady'
```

And add new things to dicts like this:

```
In [48]: my_roster_dict['k'] = 'mason crosby'
```

Notice how keys are strings (they're surrounded in quotes). They can also be numbers or even bools. They cannot be a variable that has not already been created. You could do this:

```
In [49]: pos = 'qb'

In [50]: my_roster_dict[pos]
Out[50]: 'tom brady'
```

Because when you run it Python is just replacing `pos` with `'qb'`.

But you will get an error if `pos` is undefined. You also get an error if you try to use a key that's not present in the dict (note: assigning something to a key that isn't there yet — like we did with '`mason crosby`' above — is OK).

Dictionary keys are usually strings, but they can also be numbers or bools. Dictionary values can be anything, including lists or other dicts.

Unpacking

Now that we've seen an example of container types, we can mention **unpacking**. Unpacking is a way to assign multiple variables at once, like this:

```
In [51]: qb, rb = ['tom brady', 'todd gurley']
```

That does the exact same thing as assigning these separately on their own line.

```
In [52]: qb = 'tom brady'

In [53]: rb = 'todd gurley'
```

One pitfall when unpacking values is that the number of whatever you're assigning to has to match the number of values available in your container. This would give you an error:

```
In [54]: qb, rb = ['tom brady', 'todd gurley', 'julio jones']
ValueError: too many values to unpack (expected 2)
```

Unpacking isn't used that frequently. Shorter code isn't always necessarily better, and it's probably clearer to someone reading your code if you assign `qb` and `rb` on separate lines.

However, some built-in parts of Python (including material below) use unpacking, so I wanted to cover it here.

Loops

Loops are a way to "do something" for every item in a collection.

For example, maybe I have a list of lowercase player names and I want to go through them and change them all to proper name formatting using the `title` string method, which capitalizes the first letter of every word in a string.

One way to do that is with a **`for`** loop:

```
1 my_roster_list = ['tom brady', 'adrian peterson', 'antonio brown']
2
3 my_roster_list_upper = ['', '', '']
4 i = 0
5 for player in my_roster_list:
6     my_roster_list_upper[i] = player.title()
7     i = i + 1
```

What's happening here is lines 6-7 are run multiple times, once for every item in the list. The first time `player` has the value `'tom brady'`, the second `'adrian peterson'`, etc. We're also using a variable `i` to keep track of our position in our list. The last line in the body of each loop is to increment `i` by one, so that we'll be working with the correct spot the next time we go through it.

```
In [9]: my_roster_list_upper
Out[9]: ['Tom Brady', 'Adrian Peterson', 'Antonio Brown']
```

The programming term for "going over each element in some collection" is **iterating**. Collections that allow you to iterate over them are called **iterables**.

Dicts are also iterables. The default behavior when iterating over them is you get access to the keys only. So:

```
In [58]: for x in my_roster_dict:
    ...:     print(f"position: {x}")

position: wr1
position: qb
position: rb1
```

But what if we want access to the values too? One thing we could do is write `my_roster_dict[x]`, like this:

```
In [62]: for x in my_roster_dict:
    ...:     print(f"position: {x}")
    ...:     print(f"player: {my_roster_dict[x]}")

position: wr1
player: antonio brown
position: qb
player: tom brady
position: rb1
player: adrian peterson
position: k
player: mason crosby
```

But Python has a shortcut that makes things easier: we can add `.items()` to our dict to get access to the value.

```
In [65]: for x, y in my_roster_dict.items():
    ...:     print(f"position: {x}")
    ...:     print(f"player: {y}")

position: wr1
player: antonio brown
position: qb
player: tom brady
position: rb1
player: adrian peterson
```

Notice the `for x, y...` part of the loop. Adding `.items()` unpacks the key and value into our two loop variables (we choose `x` and `y`).

Loops are occasionally useful. And they're definitely better than copying and pasting a bunch of code over and over and making some minor change.

But in many instances, there's a better option: **comprehensions**.

Comprehensions

Comprehensions are a way to modify lists or dicts with not a lot of code. They're like loops condensed onto one line.

List Comprehensions

When you want to go from one list to another different list you should be thinking comprehension. Our first `for` loop example, where we wanted to take our list of lowercase players and make a list where they're all properly formatted, is a great candidate.

The list comprehension way of doing that would be:

```
In [66]: my_roster_list
Out[66]: ['tom brady', 'adrian peterson', 'antonio brown']

In [67]: my_roster_list_proper = [x.title() for x in my_roster_list]

In [68]: my_roster_list_proper
Out[68]: ['Tom Brady', 'Adrian Peterson', 'Antonio Brown']
```

All list comprehensions take the form [*a* for *b* in *c*] where *c* is the list you're iterating over, and *b* is the variable you're using in *a* to specify exactly what you want to do to each item.

In the above example *a* is `x.title()`, *b* is `x`, and *c* is `my_roster_list`.

Note, it's common to use `x` for your comprehension variable, but — like loops — you can use whatever you want. So this:

```
In [69]: my_roster_list_proper_alt = [y.title() for y in my_roster_list]
```

does exactly the same thing as the version using `x` did.

Comprehensions can be tricky at first, but they're not that bad once you get the hang of them. They're useful and we'll see them again though, so if the explanation above is fuzzy, read it again and look at the example above it until it makes sense.

A List Comprehension is a List

A comprehension evaluates to a regular Python list. That's a fancy way of saying the result of a comprehension *is* a list.

```
In [70]: type([x.title() for x in my_roster_list])
Out[70]: list
```

And we can slice it and do everything else we could do to a normal list:

```
In [71]: [x.title() for x in my_roster_list][:2]
Out[71]: ['Tom Brady', 'Adrian Peterson']
```

More Comprehensions

Let's do another, more complicated, comprehension:

```
In [72]: my_roster_last_names = [full_name.split(' ')[1]
             for full_name in my_roster_list]

In [73]: my_roster_last_names
Out[73]: ['brady', 'peterson', 'brown']
```

In the example above, I'm taking a name separated by a space, and getting the last name from it. I do that by calling `.split('')` on it (a string method that returns a list of substrings), then picking out the last name from the list. Remember, list indexes start at 0, which is why the 1 in `full_name.split('')[1]` returns the last name.

The programming term for "doing something" to each item in a collection is **mapping**. As in, I *mapped* `title` to each element of `my_roster_list`.

You can also optionally **filter** a comprehension to include only certain items by adding an `if` *some criteria that evaluates to a boolean* at the end.

```
In [72]: my_roster_a_only = [
    ...:     x.title() for x in my_roster_list if x.startswith('a')]

In [73]: my_roster_a_only
Out[73]: ['Adrian Peterson', 'Antonio Brown']
```

Again, `startswith` is another string method. It takes a string to match and returns a bool indicating whether the original string starts with it or not.

Dict Comprehensions

Dict comprehensions work similarly to list comprehensions. Except now, the whole thing is wrapped in `{}` instead of `[]`.

And — like with our `for` loop over a dict, we can use `.items()` to get access to the key and value.

```
In [74]: pts_per_player = {'tom brady': 20.7, 'adrian peterson': 10.1, '
    antonio brown': 18.5}

In [75]: pts_x2_per_upper_player = {
             name.upper(): pts*2 for name, pts in pts_per_player.items()}

In [76]: pts_x2_per_upper_player
Out[76]: {'ADRIAN PETERSON': 20.2, 'ANTONIO BROWN': 37.0, 'TOM BRADY':
    41.4}
```

Comprehensions make it easy to go from a list to a dict or vice versa. For example, say we want to total up all the points in our dict `pts_per_player`.

Well, one way to add up a list of numbers in Python is to pass it to the `sum()` function.

```
In [77]: sum([1, 2, 3])
Out[77]: 6
```

If we want to get the total number points in our `pts_per_player` dict, we make a list of just the points using a list comprehension, then pass it to `sum` like:

```
In [78]: sum([pts for _, pts in pts_per_player.items()])
Out[78]: 49.3
```

This is still a list comprehension even though we're starting with a dict (`pts_per_player`). When in doubt, check the surrounding punctuation. It's brackets here, which means list.

Also note the `for _, pts in ...` part of the code. The only way to get access to a value of a dict (i.e., the points here) is to use `.items()`, which also gives us access to the key (the player name in this case). But since we don't actually need the key for summing points, the Python convention is to name that variable `_`. This lets people reading our code know we're not using it.

Functions

In the last section we saw `sum()`, which is a Python built-in that takes in a list of numbers and totals them up.

`sum()` is an example of a **function**. Functions are code that take inputs (the function's **arguments**) and returns outputs. Python includes several built-in functions. Another common one is `len`, which finds the length of a list.

```
In [80]: len(['tom brady', 'adrian peterson', 'antonio brown'])
Out[80]: 3
```

Using the function — i.e. giving it some inputs and having it return its output — is also known as **calling** or **applying** the function.

Once we've called a function, we can use it just like any other value. There's no difference between `len(['tom brady', 'adrian peterson', 'antonio brown'])` and `3`. We could define variables with it:

```
In [81]: pts_per_fg = len(['tom brady', 'adrian peterson', 'antonio brown'
    ])

In [82]: pts_per_fg
Out[82]: 3
```

Or use it in math.

```
In [83]: 4 + len(['tom brady', 'adrian peterson', 'antonio brown'])
Out[83]: 7
```

Or whatever. Once it's called, it's the value the function returned, that's it.

Defining Your Own Functions

It is very common to define your own functions.

```
def over_100_total_yds(rush_yds, rec_yds):
    """
    multi line strings in python are between three double quotes

    it's not required, but the convention is to put what the fn does in
       one of these multi line strings (called "docstring") right away in
       function

    when you type over_100_total_yds? in the REPL, it shows this docstring

    this function takes rushing, receiving yards, adds them, and returns a
        bool indicating whether they're more than 100 or not
    """
    return rush_yds + rec_yds > 100
```

Note the arguments `rush_yds` and `rec_yds`. These work just like normal variables, except they are only available inside your function (the function's **body**). So, even after defining this function you try to type:

```
In [81]: print(rush_yds)
NameError: name 'rush_yds' is not defined
```

You'll get an error. `rush_yds` only exists inside `over_100_total_yds`. You can put the `print` statement inside the function:

```
def noisy_over_100_total_yds(rush_yds, rec_yds):
    """
    this function takes rushing, receiving yards, adds them, and returns a
        bool
    indicating whether they're more than 100 or not

    it also prints rush_yds
    """
    print(rush_yds)
    return rush_yds + rec_yds > 100
```

The programming term for where you have access to a variable (inside the function for arguments) is **scope**.

```
In [87]: over_100_total_yds(60, 39)
Out[87]: False

In [88]: noisy_over_100_total_yds(84, 32)
84
Out[88]: True
```

Notice that — along with returning a bool — `noisy_over_100_total_yds` also prints out the value of `rush_yds`. This is a **side effect** of calling the function. A side effect is anything your function does besides returning a value. Printing is fine, but apart from that you generally you should avoid side effects in your functions.

Default Values in Functions

Here's a question: what happens if we leave out any of the arguments when calling our function? Let's try it:

```
In [18]: over_100_total_yds(92)
TypeError: over_100_total_yds() missing 1 required positional
argument: 'rec_yds'
```

We can avoid this error by writing our functions to include default values.

```
def over_100_total_yds_wdefault(rush_yds=0, rec_yds=0):
    """
    this function takes rushing, receiving yards, adds them, and returns a
        bool
    indicating whether they're more than 100 or not

    if a value for rushing or receiving yards is not entered, it'll
       default to 0
    """
    return rush_yds + rec_yds > 100
```

Adding default values lets you pick and choose which arguments you want give the function, but that also means you need to let the function know *which* arguments you're passing. You do that with **keyword** arguments, where you input `keyword=value` like this:

```
In [17]: over_100_total_yds_wdefault(rush_yds=92)
Out[17]: False
```

Keyword arguments can get cumbersome, and you can leave them off, in which case Python will assign arguments to keywords based on the order you enter them. So if we do:

```
In [18]: over_100_total_yds_wdefault(92, 9)
Out[18]: True
```

The value of `rush_yds` is 92 and `rec_yds` is 9. These are **positional** arguments. If you leave some of the arguments empty — e.g. if we leave out a second argument for `rec_yds` — the arguments you do pass will get assigned to the keywords left to right:

So in:

```
In [19]: over_100_total_yds_wdefault(92)
Out[19]: False
```

`rush_yds` gets 92 and `rec_yds` defaults to 0. What if you want to pass a value for `rec_yds` and use the default for `rush_yds`? Your only option is to use keyword arguments.

```
In [20]: over_100_total_yds_wdefault(rec_yds=44)
Out[20]: False
```

That's why it's a good idea to put your most "important" arguments first, and leave your optional arguments at the end.

For example, later we'll learn about the `read_csv` function in Pandas, whose job is to load your csv data into Python. The first argument to `read_csv` is a string with the file path to your csv, and that's the only one you'll use 95% of the time. But is also has more than 40 optional keyword arguments, everything from `skip_blank_lines` (defaults to `True`) to `parse_dates` (defaults to `False`).

What it means is usually you can just use the function like:

```
data = read_csv('my_data_file.csv')
```

And on the rare occasions when we want to tweak some option, do:

```
data = read_csv('my_data_file.csv', skip_blank_lines=False,
                parse_dates=True)
```

Note the above is mixing positional (`my_data_file.csv`) and keyword arguments (`skip_blank_lines=False` and `parse_dates=True`). That's standard. The only rule is any keyword arguments have to be *after* any positional arguments.

Functions That Take Other Functions

A cool feature of Python is that functions can take other functions.

```
def do_to_list(working_list, working_fn, desc):
    """
    this function takes a list, a function that works on a list, and a
    description

    it applies the function to the list, then returns the result along
       with
    description as a string
    """

    value = working_fn(working_list)

    return f'{desc} {value}'
```

Now let's also make a function to use this on.

```
def last_elem_in_list(working_list):
    """
    returns the last element of a list.
    """
    return working_list[-1]
```

And try it out:

```
In [102]: positions = ['QB', 'RB', 'WR', 'TE', 'K', 'DST']

In [103]: do_to_list(positions, last_elem_in_list, "last element in your
    list:")
Out[103]: 'last element in your list: DST'

In [104]: do_to_list([1, 2, 4, 8], last_elem_in_list, "last element in
    your list:")
Out[104]: 'last element in your list: 8'
```

The function `do_to_list` can work on built in functions too.

```
In [105]: do_to_list(positions, len, "length of your list:")
Out[105]: 'length of your list: 6'
```

You can also create functions on the fly without names, usually for purposes of passing to other, flexible functions.

```
In [106]: do_to_list([2, 3, 7, 1.3, 5], lambda x: 3*x[0],
     ...:     "first element in your list times 3 is:")
Out[106]: 'first element in your list times 3 is: 6'
```

These are called **anonymous** or **lambda** functions.

Libraries are Functions and Types

There is much more to basic Python than this, but this is enough of a foundation to learn the other **libraries** we'll be using.

Libraries are just a collection of functions and types that other people have written using Python [2] and other libraries. That's why it's critical to understand the concepts in this section. Libraries *are* Python, with lists, dicts, bools, functions and all the rest.

os Library and path

Some libraries come built-in to Python. One example we'll use is the `os` (for operating system) library. To use it, we have to import it, like this:

```
In [107]: import os
```

[2] Technically sometimes they use other programming languages too. Parts of the data analysis library Pandas, for example, are written in C for performance reasons. But we don't have to worry about that.

That lets us use all the functions written in the library `os`. For example, we can call `cpu_count` to see the number of computer cores we currently have available.

```
In [108]: os.cpu_count()
Out[108]: 12
```

Libraries like `os` can contain other sub libraries too. The sub library we'll use from `os` is the `path` library, which is useful for working with filenames. One of the main function is `join`, which takes a directory (or multiple directories) and a filename and puts them together in a string. Like this:

```
In [108]: from os import path

In [109]: DATA_DIR = '/Users/nathan/ltcwff-files/data'

In [109]: path.join(DATA_DIR, 'adp_2017.csv')
Out[109]: '/Users/nathan/ltcwff-files/data/adp_2017.csv'
```

With `join`, you don't have to worry about trailing slashes or operating system differences or anything like that. You can just replace `DATA_DIR` with the directory that holds the csv files that came with this book and you'll be set.

End of Chapter Exercises

2.1

Which of the following are valid Python variable names?

a) `_throwaway_data`
b) `pts_per_passing_td`
c) `2nd_rb_name`
d) `puntReturnsForTds`
e) `rb1_name`
f) `flex spot name`
g) `@home_or_away`
h) `'pts_per_rec_yd'`

2.2

What is the value of weekly_points at the end of the following code?

```
weekly_points = 100
weekly_points = weekly_points + 28
weekly_points = weekly_points + 5
```

2.3

Write a function named `for_the_td` that takes in the name of two players (e.g. `'Dak'`, `'Zeke'`) and returns a string of the form: `'Dak to Zeke for the td!'`

2.4

Without looking it up, what do you think the string method `islower` does? What type of value does it return? Write some code to test your guess.

2.5

Write a function `is_leveon` that takes in a player name and returns a `bool` indicating whether the player's name is "Le'Veon Bell" — regardless of case or whether the user included the `'`.

2.6

Write a function `commentary` that takes in a number (e.g. 120 or 90) and returns a string `'120 is a good score'` if the number is >= 100 or `"90's not that good"` otherwise.

2.7

Say we have a list:

```
giants_roster = ['Daniel Jones', 'Saquon Barkley', 'Evan Engram', 'OBJ']
```

List at least three ways you can to print the list without `'OBJ'`. Use at least one list comprehension.

2.8

Say we have a dict:

```
league_settings = {'number_of_teams': 12, 'ppr': True}
```

a) How would you change `'number_of_teams'` to 10?

b) Write a function `toggle_ppr` that takes a dict like `league_settings`, turns `'ppr'` to the opposite of whatever it is, and returns the updated settings dict.

2.9

Assuming we've defined our same dict:

```
league_settings = {'number_of_teams': 12, 'ppr': True}
```

Go through each line and say whether it'll work without error:

a) `league_settings['has_a_flex']`
b) `league_settings[number_of_teams]`
c) `league_settings['year founded'] = 2002`

2.10

Say we're working with the list:

```
my_roster_list = ['tom brady', 'adrian peterson', 'antonio brown']
```

a) Write a loop that goes through and prints the last name of every player in `my_roster_list`.
b) Write a comprehension that uses `my_roster_list` to make a dict where the keys are the player names and the values are the length's of the strings.

2.11

Say we're working with the dict:

```
my_roster_dict = {
    'qb': 'tom brady', 'rb1': 'adrian peterson', 'wr1': 'davante adams', '
       wr2':
    'john brown'}
```

a) Write a comprehension that turns 'my_roster_dict1 into a list of just the positions.
b) Write a comprehension that turns `my_roster_dict` into a list of just players who's last names start with `'a'` or `'b'`.

2.12

a) Write a function `mapper` that takes a list and a function, applies the function to every item in the list and returns it.
b) Assuming 0.1 points per rushing yard, use `mapper` with an anonymous function to get a list of pts from rushing yds

```
list_of_rushing_yds = [1, 110, 60, 4, 0, 0, 0]
```

3. Pandas

Introduction to Pandas

In the last chapter we talked about basic, built-in Python.

In this chapter we'll talk about Pandas, which is the most important Python library for working with data. It's an external library, but don't underestimate it. It's really the only game in town for what it does.

And what it does is important. In the first chapter, we talked about the five steps to doing data analysis — (1) collecting, (2) storing, (3) loading, (4) manipulating, and (5) analyzing data. Pandas is the primary tool for (4), which is where data scientists spend most of their time.

But we'll use it in other sections too. It has input-output capabilities for (2) storing and (3) loading data, and works well with key (5) analysis libraries.

Types and Functions

In chapter one, we how learned data is a collection of structured information; each row is an observation and each column some attribute.

Pandas gives you types and functions for working with this tabular data. The most important is a **DataFrame**, which is a container type like a list or dict and holds a tabular data table. One column of a DataFrame is its own type, called a **Series**, which you'll also sometimes use.

At a very high level, you can think about Pandas as a Python library that gives you access to these two types and functions that operate on them.

This sounds simple, but Pandas is powerful, and there are many ways to "operate" on tabular data. As a result, this is the longest and most information-dense chapter in the book. Don't let that scare you, it's all learnable. To make it easier, let's map out what we'll cover and the approach we'll take.

First, we'll learn how to load data from a csv file into a DataFrame. We'll learn basics like how to access specific columns and print out the first five rows. We'll also cover **indexing** — which is a handy, powerful feature of DataFrames — and how to save data from DataFrames to csv files.

With those basic covered, we'll learn about *things you can do with DataFrames*. You can do a lot with DataFrames, and — rapid fire one after the other — it might get overwhelming, but generally I'd say everything falls into one of five specific categories:

1. Modifying or creating new columns of data.
2. Using built-in Pandas functions that operate on DataFrames (or Series) and provide you with ready-made statistics or other useful information.
3. *Filtering* observations, i.e. selecting only certain rows from your data.
4. Changing the *granularity* of the data within a DataFrame.
5. Combining two or more DataFrames via Pandas's `merge` or `concat` functions.

That's it. Most of your time spent as a Python fantasy data analyst will be working with Pandas, most of your time in Pandas will be working with DataFrames, and most of your time working with DataFrames will fall into one of these five categories.

How to Read This Chapter

This chapter — like the rest of the book — is heavy on examples. All the examples in this chapter are included in a series of Python files. Ideally, you would have the file open in your Spyder editor and be running the examples (highlight the line(s) you want and press F9 to send it to the REPL/console) as we go through them in the book.

Let's get started.

Part 1. DataFrame Basics

Loading Data

This first section of examples can be found in the file `03_00_basics.py`.

The first step to working with DataFrames is loading data. We'll talk about other ways (like SQL and scraping) to load data later in the book, but for now, let's start by importing our libraries and setting our DATA_DIR to the location of the csv files this book came with.

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'

adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))
```

A few things:

First, note the lines at very top: `import pandas as pd`. Because Pandas is an external library, we have to import it. The convention is to import it under the name `pd`. We're also importing the `path` library. It's customary to import all the libraries you'll need at the top of a file.

In the last chapter we learned about Python types and functions and how to write our own. Remember, libraries are just a collection of functions and types that were themselves written using Python and other libraries.

We're importing Pandas as `pd`, so we can use any Pandas function by calling `pd.` (i.e. pd dot — type the period) and the name of our function.

Usually you have install third party libraries — using a tool like **pip** — before you can import them, but if you're using the Anaconda Python bundle, it comes with these already installed.

One of the functions Pandas comes with is `read_csv`, which takes as its argument a string with the path to the csv file you want to load. It returns a DataFrame of data.

After you've imported your libraries and change `DATA_DIR` to the location of your files that came with the book you can run:

```
In [1]: adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))

In [2]: type(adp)
Out[2]: pandas.core.frame.DataFrame
```

DataFrame Methods and Attributes

Like other Python types, DataFrames have methods you can call. For example, the function `head` prints the first five rows your data.

```
In [3]: adp.head()
Out[3]:
   adp  adp_formatted  bye  ...  stdev  team times_drafted
0  1.3           1.01   12  ...    0.6   ARI           310
1  2.3           1.02    7  ...    0.8   PIT           303
2  3.7           1.04    7  ...    1.0   PIT           338
3  5.7           1.06    9  ...    3.2   ATL           131
4  6.2           1.06    8  ...    2.8   DAL           180

[5 rows x 11 columns]
```

Note `head` hides some columns here (indicated by the `...`) because they don't fit on the screen.

We'll use `head` frequently in this chapter to quickly glance at DataFrames in our examples and show the results of what we're doing. This isn't just for the sake of the book; I use `head` all the time to

examine results and look at DataFrames when I'm coding.

Methods vs Attributes

`head` is a *method* because you can pass it the number of rows to print (the default is 5). But DataFrames also have fixed attributes that you can access without passing any data in parenthesis.

For example, the columns are available in the attribute `columns`.

```
In [4]: adp.columns
Out[4]:
Index(['adp', 'adp_formatted', 'bye', 'high', 'low', 'name', 'player_id',
       'position', 'stdev', 'team', 'times_drafted'],
      dtype='object')
```

And the number of rows and columns are in `shape`.

```
In [6]: adp.shape
Out[6]: (184, 11)
```

Working with Subsets of Columns

A Single Column

Referring to a single column in a DataFrame is similar to returning a value from a dictionary, you put the name of the column (usually a string) in brackets.

```
In [7]: adp['name'].head()
Out[7]:
0      David Johnson
1        LeVeon Bell
2      Antonio Brown
3        Julio Jones
4    Ezekiel Elliott
```

Technically, a single column is a Series, not a DataFrame.

```
In [8]: type(adp['name'])
Out[8]: pandas.core.series.Series
```

The distinction isn't important right now, but eventually you'll run across functions that operate on Series instead of a DataFrame or vis versa. If you need to go from a Series to a DataFrame, you can turn it into a one-column DataFrame with `to_frame`.

```
In [9]: adp['name'].to_frame().head()
Out[9]:
              name
0    David Johnson
1      LeVeon Bell
2    Antonio Brown
3      Julio Jones
4  Ezekiel Elliott

In [10]: type(adp['name'].to_frame())
Out[10]: pandas.core.frame.DataFrame
```

Multiple Columns

To refer to multiple columns in a DataFrame, you pass it a list. The result — unlike the single column case — is another DataFrame.

```
In [11]: adp[['name', 'position', 'adp']].head()
Out[11]:
              name position  adp
0    David Johnson       RB  1.3
1      LeVeon Bell       RB  2.3
2    Antonio Brown       WR  3.7
3      Julio Jones       WR  5.7
4  Ezekiel Elliott       RB  6.2

In [12]: type(adp[['name', 'position', 'adp']])
Out[12]: pandas.core.frame.DataFrame
```

Notice the difference between `adp['name']` and `adp[['name', 'position', 'adp']]`. In the former we have `'name'`, and it's completely replaced by `['name', 'position', 'adp']` in the latter. That is — since you're putting a list with your column names *inside* another pair brackets — there are two sets of brackets in all.

I guarantee at some point you will forget about this and accidentally do something like:

```
In [13]: adp['name', 'position', 'adp'].head()
...
KeyError: ('name', 'position', 'adp')
```

which will throw an error. No big deal, just remember: inside the brackets is a single, dict-like string to return one column, a list to return multiple columns.

Indexing

A key feature of Pandas is that every DataFrame (and Series) has an **index**.

You can think of the index as a built-in column of row IDs. You can specify which column to use as the index when loading your data. If you don't, the default is a series of numbers starting from 0 and going up to the number of rows.

The index is on the very left hand side of the screen when you look at output from the `head` method. We didn't specify any column to use as the index when calling `read_csv` above, so you can see it defaults to 0, 1, 2, …

```
In [14]: adp[['name', 'position', 'adp']].head()
Out[14]:
              name position  adp
0    David Johnson       RB  1.3
1      LeVeon Bell       RB  2.3
2    Antonio Brown       WR  3.7
3      Julio Jones       WR  5.7
4  Ezekiel Elliott       RB  6.2
```

Indexes don't have to be numbers; they can be strings or dates, whatever. A lot of times they're more useful when they're meaningful to you.

```
In [15]: adp.set_index('player_id').head()
Out[15]:
            adp  adp_formatted  ...  team  times_drafted
player_id                       ...
2297        1.3           1.01  ...   ARI            310
1983        2.3           1.02  ...   PIT            303
1886        3.7           1.04  ...   PIT            338
1796        5.7           1.06  ...   ATL            131
2343        6.2           1.06  ...   DAL            180
```

Copies and the Inplace Argument

Now that we've run `set_index`, our new index is the `player_id` column. Or is it? Try running `head` again:

```
In [16]: adp.head()
Out[16]:
   adp  adp_formatted  bye  ...  stdev  team times_drafted
0  1.3           1.01   12  ...    0.6   ARI           310
1  2.3           1.02    7  ...    0.8   PIT           303
2  3.7           1.04    7  ...    1.0   PIT           338
3  5.7           1.06    9  ...    3.2   ATL           131
4  6.2           1.06    8  ...    2.8   DAL           180
```

Our index is still 0, 1 … 4 — what happened? The answer is that `set_index` returns a new, *copy* of the `adp` DataFrame with the index we want. When we called `adp.set_index('player_id')` above,

we just displayed that newly index `adp` in the REPL. We didn't actually do anything to our original, old `adp`.

To make it permanent, we can either set the `inplace` argument to `True`:

```
In [17]: adp.set_index('player_id', inplace=True)

In [18]: adp.head()  # now player_id is index
Out[18]:
            adp  adp_formatted  ...  team  times_drafted
player_id                       ...
2297        1.3           1.01  ...   ARI            310
1983        2.3           1.02  ...   PIT            303
1886        3.7           1.04  ...   PIT            338
1796        5.7           1.06  ...   ATL            131
2343        6.2           1.06  ...   DAL            180
```

Or we can overwrite `adp` with our copy with the updated index:

```
In [19]: adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))

In [20]: adp = adp.set_index('player_id')
```

Most DataFrame methods (including non-index related methods) behave like this — returning copies unless you include `inplace=True` — so if you're calling a method and it's behaving unexpectedly, this is one thing to watch out for.

The opposite of `set_index` is `reset_index`. It sets the index to 0, 1, 2, … and turns the old index into a regular column.

```
In [21]: adp.reset_index().head()
Out[21]:
   player_id  adp  ...  team  times_drafted
0       2297  1.3  ...   ARI            310
1       1983  2.3  ...   PIT            303
2       1886  3.7  ...   PIT            338
3       1796  5.7  ...   ATL            131
4       2343  6.2  ...   DAL            180
```

Indexes Keep Things Aligned

The main benefits of indexes in Pandas is automatic alignment.

To illustrate this, let's make a mini, subset of our DataFrame with just the names and teams of the RBs. Don't worry about the `loc` syntax for now, just know that we're creating a smaller subset of our data that is just the running backs.

```
In [22]: adp_rbs = adp.loc[adp['position'] == 'RB',
                           ['name', 'position', 'team']]

In [23]: adp_rbs.head()
Out[23]:
                    name position team
player_id
2297         David Johnson       RB  ARI
1983           LeVeon Bell       RB  PIT
2343       Ezekiel Elliott       RB  DAL
2144       Devonta Freeman       RB  ATL
1645          LeSean McCoy       RB  BUF
```

Yep, all running backs. Now let's use another DataFrame method to sort them by name.

```
In [24]: adp_rbs.sort_values('name', inplace=True)

In [25]: adp_rbs.head()
Out[25]:
                    name position team
player_id
925        Adrian Peterson       RB  ARI
2439          Alvin Kamara       RB   NO
2293        Ameer Abdullah       RB  DET
1867          Bilal Powell       RB  NYJ
2071           CJ Anderson       RB  DEN
```

Now, what if we want to go back and add in the ADP column from our original, all positions DataFrame?

Adding a column works similarly to variable assignment in regular Python.

```
In [26]: adp_rbs['adp'] = adp['adp']

In [27]: adp_rbs.head()
Out[27]:
                    name position team    adp
player_id
925        Adrian Peterson       RB  ARI   69.9
2439          Alvin Kamara       RB   NO  140.0
2293        Ameer Abdullah       RB  DET   53.0
1867          Bilal Powell       RB  NYJ   48.6
2071           CJ Anderson       RB  DEN   51.4
```

Viola. Even though we have a separate, smaller dataset with a different number of rows (only the RBs) in a different order (sorted alphabetically vs by ADP) compared to the main data, we were able to easily add in adp back in from our old DataFrame.

We're able to do that because `adp`, `adp_rbs`, and the Series `adp['adp']` all have the same index

values for the rows they have in common.

In a spreadsheet program you have to be aware about how your data was sorted and the number of rows before copying and pasting columns around. The benefit of indexes in Pandas is you can just what you want and not have to worry about it.

Outputting Data

The opposite of loading data is outputting it, and Pandas does that too.

While the input methods are in the top level Pandas namespace — i.e. you read csvs by calling `pd.read_csv` — the output methods are called on the DataFrame itself.

For example, if we want to save our RB DataFrame:

```
In [28]: adp_rbs.to_csv(path.join(DATA_DIR, 'adp_rb.csv'))
```

By default, Pandas will include the index in the csv file. This is useful when the index is meaningful (like it is here), but if the index is just the default range of numbers you might not want to write it. In that case you would set `index=False`.

```
In [29]: adp_rbs.to_csv(path.join(DATA_DIR, 'adp_rb_no_index.csv'), index=
    False)
```

Exercises

3.0.1

Load the ADP data into a DataFrame named `adp`. You'll use it for the rest of the problems in this section.

3.0.2

Use the `adp` DataFrame to create another DataFrame, `adp50`, that is the top 50 (by ADP) players.

3.0.3

Sort `adp` by name in descending order (so Zay Jones is on the first line). On another line, look at `adp` in the REPL and make sure it worked.

3.0.4

What is the type of `adp.sort_values('adp')`?

3.0.5

a) Make a new DataFrame, `adp_simple`, with just the columns `'name'`, `'position'`, `'adp'` in that order.

b) Rearrange `adp_simple` so the order is `'pos'`, `'name'`, `'adp'`.

c) Using the original `adp` DataFrame, add the `'team'` column to `adp_simple`.

d) Write a copy of `adp_simple` to your computer, 'adp_simple.txt' that is `'|'` (pipe) delimited instead of `','` (comma) delimited.

Part 2. Things you can do with DataFrames

Introduction

Now that we understand DataFrames (Python container types for tabular data with indexes), and how to load and save them, let's get into the things you can do with them.

To review, the things you can do with DataFrames fall into the following categories.

1. Modifying or creating new columns of data.
2. Using built-in Pandas functions that operate on DataFrames (or Series) and provide you with ready-made statistics or other useful information.
3. *Filtering* observations, i.e. selecting only certain rows from your data.
4. Changing the *granularity* of the data within a DataFrame.
5. Combining two or more DataFrames via Pandas's `merge` or `concat` functions.

Let's dive in.

1. Modify or create new columns of data

The first thing we'll learn is how to work with *columns* in DataFrames. Note, you should follow along with this these examples in the file `03_01_columns.py`.

This section covers both modifying and creating new columns, because they're really just variations on the same thing. For example, we've already seen how creating a new column works similarly to variable assignment in regular Python:

```
In [2]: pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'
    ))

In [3]: pg['pts_pr_pass_td'] = 4

In [4]: pg[['gameid', 'player_id', 'pts_pr_pass_td']].head()
Out[4]:
       gameid   player_id  pts_pr_pass_td
0  2017090700  00-0019596               4
1  2017090700  00-0023436               4
2  2017090700  00-0026035               4
3  2017090700  00-0030288               4
4  2017090700  00-0030506               4
```

What if we want to modify `df['pts_pr_pass_td']` after we created it? It's no different than creating it originally.

```
In [5]: pg['pts_pr_pass_td'] = 6

In [6]: pg[['gameid', 'player_id', 'pts_pr_pass_td']].head()
Out[6]:
       gameid   player_id  pts_pr_pass_td
0  2017090700  00-0019596               6
1  2017090700  00-0023436               6
2  2017090700  00-0026035               6
3  2017090700  00-0030288               6
4  2017090700  00-0030506               6
```

So the distinction between modifying and creating columns is minor. Really this section is more about working with columns in general.

To start, let's go over three of the main column types — number, string, and boolean — and how you might work with them.

Math and number columns

Doing math operations on columns is intuitive and probably works how you would expect:

```
In [7]:
pg['rushing_pts'] = (
    pg['rush_yards']*0.1 + pg['rush_tds']*6 + pg['rush_fumbles']*-3)

In [8]: pg[['player_name', 'gameid', 'rushing_pts']].head()
Out[8]:
  player_name      gameid  rushing_pts
0     T.Brady  2017090700          0.0
1     A.Smith  2017090700          0.9
2  D.Amendola  2017090700          0.0
3  R.Burkhead  2017090700          1.5
4     T.Kelce  2017090700          0.4
```

This adds a new column `rushing_pts` to our `pg` DataFrame[1]. We can see Tom Brady and Alex Smith combined for less than 1 rushing fantasy point for the first week of the 2017 season. Sounds about right.

Other math operations work too, though for some functions we have to load new libraries. Numpy is a more raw, math oriented Python library that Pandas is built on. It's commonly imported as `np`.

Here we're talking the absolute value and natural log[2] of rushing yards:

```
In [9]: import numpy as np

In [10]: pg['distance_traveled'] = np.abs(pg['rush_yards'])

In [11]: pg['ln_rush_yds'] = np.log(pg['rush_yards'])
```

You can also assign scalar (single number) columns. In that case the value will be constant throughout the DataFrame.

[1] If we wanted, we could also create a separate column (Series), unattached to `pg`, but with the same index like this `rushing_pts = (pg['rush_yards']*0.1 + pg['rush_tds']*6 + pg['rush_fumbles']*-3)`.

[2] If you're not familiar with what the natural log means this is a good link

Aside: I want to keep printing out the results with `head`, but looking at the first five rows all time time is sort of boring. Instead, let's pick 5 random rows using the `sample` method. If you're following along in Spyder, you'll see five different rows because `sample` returns a random sample every time.

```
In [12]: pg['points_per_fg'] = 3

In [13]: pg[['player_name', 'gameid', 'points_per_fg']].sample(5)
Out[13]:
     player_name      gameid  points_per_fg
149      T.Kelce  2017120308              3
336     L.Murray  2017120300              3
638     M.Breida  2017100109              3
90     A.Jeffery  2017091704              3
1014     T.Smith  2017102906              3
```

String columns

Data analysis work almost always involves columns of numbers, but it's common to have to work with string columns too.

Pandas let's you manipulate string columns by calling `str` on the relevant column.

```
In [14]: pg['player_name'].str.upper().sample(5)
Out[14]:
53             R.BURKHEAD
511                K.COLE
264     B.ROETHLISBERGER
1134             K.ALLEN
366               G.TATE

In [15]: pg['player_name'].str.replace('.', ' ').sample(5)
Out[15]:
643     A Jeffery
1227    D Jackson
517        K Cole
74         K Hunt
1068     A Kamara
```

The + sign concatenates (sticks together) string columns.

```
In [16]: (pg['player_name'] + ', ' + pg['pos'] + ' - ' + pg['team']).
    sample(5)
Out[16]:
489          M.Lee, WR - JAX
1255         C.Sims, RB - TB
350     M.Stafford, QB - DET
129     R.Anderson, WR - NYJ
494          M.Lee, WR - JAX
```

If you want to chain these together (i.e. call multiple string functions in a row) you can, but you'll have to call `str` multiple times.

```
In [17]: pg['player_name'].str.replace('.', ' ').str.lower().sample(5)
Out[17]:
499          m lee
937     t williams
1249        c sims
53      r burkhead
1150      b fowler
Name: player_name, dtype: object
```

Bool Columns

It's common to work with columns of booleans.

The following creates a column that is `True` if the row in question is a running back.

```
In [18]: pg['is_a_rb'] = (pg['pos'] == 'RB')

In [18]: pg[['player_name', 'is_a_rb']].sample(5)
Out[18]:
       player_name  is_a_rb
332   L.Fitzgerald    False
137         M.Ryan    False
681         F.Gore     True
548       D.Murray     True
1142      B.Fowler    False
```

We can combine logic operations too. Note the parenthesis, and `|` and `&` for `or` and `and` respectively.

```
In [19]: pg['is_a_rb_or_wr'] = (pg['pos'] == 'RB') | (pg['pos'] == 'WR')

In [20]: pg['good_rb_game'] = (pg['pos'] == 'RB') & (pg['rush_yards'] >=
    100)
```

You can also negate (change `True` to `False` and vis versa) booleans using the tilda character (~).

```
In [97]: pg['is_not_a_rb_or_wr'] = ~((pg['pos'] == 'RB') | (pg['pos'] == '
    WR'))
```

Pandas also lets you work with multiple columns at once. Sometimes this is useful when working with boolean columns. For example, to check whether a player went for over 100 rushing or receiving yards you could do:

```
In [26]: (pg[['rush_yards', 'rec_yards']] > 100).sample(5)
Out[26]:
      rush_yards  rec_yards
1105       False      False
296         True      False
627        False      False
455        False      False
1165       False      False
```

This returns a DataFrame of all boolean values.

Applying Functions to Columns

Pandas has a lot of built in functions that work on columns, but sometimes you need to come up with your own.

For example, maybe you want to go through and flag (note: when you hear "flag" think: make a column of bools) skill position players. Rather than doing a very long boolean expression with many | values, we might do something like:

```
In [27]:
def is_skill(pos):
  """
  Takes some string named pos ('QB', 'K', 'RT' etc) and checks
  whether it's a skill position (RB, WR, TE).
  """
  return pos in ['RB', 'WR', 'TE']

In [28]: pg['is_skill'] = pg['pos'].apply(is_skill)

In [29]: pg[['player_name', 'is_skill']].sample(5)
Out[29]:
     player_name  is_skill
1112  C.Anderson      True
289     M.Bryant      True
637      T.Smith      True
730    G.Everett      True
107     T.Taylor     False
```

This takes our function and **applies** it to every `pos` in our column of positions, one at a time.

Our function `is_skill` is pretty simple, it just takes one argument and does a quick check to see if it's in a list. This is where an unnamed, anonymous (or lambda) function would be useful.

```
In [30]: pg['is_skill_alternate'] = pg['pos'].apply(
                lambda x: x in ['RB', 'WR', 'TE'])
```

Dropping Columns

Dropping a column works like:

```
In [32]: pg.drop('is_skill_alternate', axis=1, inplace=True)
```

Note the `inplace` and `axis` arguments. The `axis=1` is necessary because the default behavior of `drop` is to operate on rows. I.e., you pass it an index value, and it drops that row from the DataFrame. In my experience this is hardly ever what you want. It's much more common to have to pass `axis=1` so that it'll drop the name of the column you provide instead.

Renaming Columns

Technically, one way to modify columns is by renaming them, so let's talk about that here.

There are two ways to rename columns. First, you can assign new data to the `columns` attribute of your DataFrame.

Let's rename all of our columns in `pg` to be uppercase (note the list comprehension):

```
In [33]: pg.columns = [x.upper() for x in pg.columns]

In [34]: pg.head()
Out[34]:
       GAMEID   PLAYER_ID  ...  GOOD_RB_GAME  IS_NOT_A_RB_OR_WR
0  2017090700  00-0019596  ...         False               True
1  2017090700  00-0023436  ...         False               True
2  2017090700  00-0026035  ...         False              False
3  2017090700  00-0030288  ...         False              False
4  2017090700  00-0030506  ...         False               True
```

Uppercase isn't the Pandas convention so let's change it back.

```
In [35]: pg.columns = [x.lower() for x in pg.columns]
```

Another way to rename columns is by calling the `rename` method and passing in a dictionary. Maybe we want to rename `interceptions` to just `ints`.

```
In [36]: pg.rename(columns={'interceptions': 'ints'}, inplace=True)
```

Missing Data in Columns

Missing values are common when working with data. Sometimes data is missing because we just don't have it. For instance, maybe we've set up an automatic web scraper to collect and store daily injury

reports, but the website was temporarily down and we missed a day. That day might be represented as missing in our dataset.

Other times data we might want to intentionally treat data as missing. For example, in our play by play data we have a yards after the catch column. What should the value of this column be for running plays?

It could be 0, but then if we tried to calculate average yards after catch, it'd be misleading low. Missing is better.

Missing values in Pandas have the value of `np.nan`. Remember, `np` is the numpy library that Pandas is built upon.

Pandas comes with functions that work with missing values, including `isnull` and `notnull`, which return a column of bools indicating whether the column is or is not missing respectively.

```
In [37]: pbp = pd.read_csv(path.join(DATA_DIR, 'play_data_sample.csv'))

In [38]: pbp['yards_after_catch'].isnull().head()
Out[38]:
0    False
1     True
2     True
3     True
4     True

In [39]: pbp['yards_after_catch'].notnull().head()
Out[39]:
0     True
1    False
2    False
3    False
4    False
```

You can also use `fillna` to replace all missing values with a value of your choosing.

```
In [40]: pbp['yards_after_catch'].fillna(-99).head()
Out[40]:
0     3.0
1   -99.0
2   -99.0
3   -99.0
4   -99.0
```

Changing column types

Another common way to modify columns is to change between data types, go from a column of strings to a column of numbers, or vis versa.

For example, maybe we want to add a "month" column to our player-game data and notice we can get it from the `gameid` column.

```
In [41]: pg['gameid'].head()
Out[41]:
0    2017090700
1    2017091705
2    2017092407
3    2017100107
4    2017100500
```

In normal Python, if we wanted to get the month out of the of a string like `'2017090700'` we would just do:

```
In [42]: gameid = '2017090700'

In [43]: year = gameid[0:4]
In [44]: month = gameid[4:6]
In [45]: day = gameid[6:8]

In [46]: year
Out[46]: '2017'

In [47]: month
Out[47]: '09'

In [48]: day
Out[48]: '07'
```

So let's try some of our string methods on it.

```
In [49]: pg['month'] = pg['gameid'].str[4:6]

AttributeError: Can only use .str accessor with string values,
which use np.object_ dtype in pandas
```

It looks like our `gameid` is stored as a number, which means we can't use `str` methods on it. No problem, we can convert it to a string using the `astype` method.

```
In [49]: pg['month'] = pg['gameid'].astype(str).str[4:6]

In [50]: pg[['month', 'gameid']].head()
Out[50]:
  month      gameid
0    09  2017090700
1    09  2017091705
2    09  2017092407
3    10  2017100107
4    10  2017100500
```

But now `month` is a string now too (you can tell by the leading 0). We can convert it back to an integer with another call to `astype`.

```
In [51]: pg['month'].astype(int).head()
Out[51]:
0     9
1     9
2     9
3    10
4    10
```

The DataFrame attribute `dtypes` tells us what all of our columns are.

```
In [52]: pg.dtypes.head()
Out[52]:
gameid            int64
player_id        object
carries         float64
rush_yards      float64
rush_fumbles    float64
```

Don't worry about the 64 after int and float; it's beyond the scope of this book. Also note for some reason string columns are denoted `object` (rather than `str`) in the `dtypes` output.

Review

This section was all about creating and manipulating columns. In reality, these are basically the same thing; the only difference is whether we make a new column (create) or overwrite and replace a column we already had (manipulate).

We learned about number, string, and boolean columns and how to convert between them. We also learned how to apply our own functions to columns and work with missing data. Finally, we learned how to drop and rename columns.

In the next section of things you can do with DataFrames we'll learn about some built in Pandas functions that provide some basic insight and analysis.

Exercises

3.1.1 Load the player game data into a DataFrame named `pg`. You'll use it for the rest of the problems in this section.

3.1.2 Add a column to `pg`, `'rec_pts_ppr'` that gives total receiving fantasy points, where the scoring system is 1 point for every 10 yards receiving, 6 points per receiving TD and 1 point per reception.

3.1.3 Add a column `'player_desc'` to pg that takes the form, ' is the ', e.g. `'T.Brady is the NE QB'` for tom brady

3.1.4 Add a boolean column to pg `'is_possesion_rec'` indicating whether a players air yards were greater than his total yards after the catch.

3.1.5 Add a column `'len_last_name'` that gives the length of the player's last name.

3.1.6 `'gameid'` is a numeric (int) column, but it's not really meant for doing math, change it into a string column.

3.1.7

a) Let's make the columns in `pg` more readable. Replace all the `'_'` with `' '` in all the columns.

b) This actually isn't good practice. Change it back.

3.1.8

a) Make a new column `'rush_td_percentage'` indicating the percentage of a players carries that went for touchdowns.

b) There are missing values in this column, why? Replace all the missing values with `-99`.

3.1.9 Drop the column `'rush_td_percentage'`. In another line, confirm that it worked.

2. Use built-in Pandas functions that work on DataFrames

Recall how analysis is the process of going from raw data down to some stat. Well, Pandas includes a lot of functions that operate on DataFrames and calculate these stats for you. In this section we'll learn about some of these functions and how to apply them to columns (the default) or rows. Note the examples for this section are in the file `03_02_functions.py`.

For example to take the average (mean) of every numeric column in your DataFrame you can do:

```
In [2]: adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))

In [3]: adp.mean()
Out[3]:
adp                88.582609
adp_formatted       7.897391
bye                 8.907609
high               67.108696
low               107.250000
player_id        1850.195652
stdev               8.338587
times_drafted     164.614130
```

We can also do max.

```
In [4]: adp.max()
Out[4]:
adp                  172.7
adp_formatted        15.05
bye                     12
high                   162
low                    180
name             Zay Jones
player_id             2523
position                WR
stdev                 21.3
team                   WAS
times_drafted          338
```

This returns the highest value of every column. Note unlike mean, it operates on string columns too (it treats "max" as latest in the alphabet, which is why we got Zay Jones).

Other functions include `std`, `count`, `sum`, and `min`.

Axis

All of the mean, max, sum etc functions take an axis argument which lets you specify whether you want to calculate the stat on the *columns* (the default, axis=0) or the *rows* (axis=1).

Calculating the stats on the columns is usually what you want. Like this (note the `axis=0` below is unnecessary since it's the default, but I'm including it for illustrative purposes):

```
In [5]: adp[['adp', 'low', 'high', 'stdev']].mean(axis=0)
Out[5]:
adp       88.582609
low      107.250000
high      67.108696
stdev      8.338587
dtype: float64
```

Calling the function by rows (with `axis=1`) on the other hand, would give us some nonsensical output. Remember, our ADP data is by *player*, so calling mean with `axis=1` would give us the average of each player's adp, lowest pick, highest pick, and standard deviation.

```
In [6]: adp[['adp', 'low', 'high', 'stdev']].mean(axis=1).head()
Out[6]:
player_id
2297    1.725
1983    2.525
1886    3.175
1796    6.225
2343    6.750
dtype: float64
```

This isn't useful here, but sometimes data is structured differently. For example, you might have a DataFrame where the columns are: name, pts1, pts2, pts3 …, pts17 where row represents a player and there are 17 columns, one for every week.

Then `axis=0` would give the average score (across all players) each week, which could be interesting, and `axis=1` would give you each player's average for the whole season, which could also be interesting.

Summary Functions on Boolean Columns

When you use the built-in summary stats on boolean columns, Pandas will treat them as 0 for `False`, 1 for `True`.

How often did RBs go over 100 yards in our player-game sample?

```
In [5]: pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'
    ))

In [6]: pg['good_rb_game'] = (pg['pos'] == 'RB') & (pg['rush_yards'] >=
    100)

In [7]: pg['good_rb_game'].mean()
Out[7]: 0.027952480782669462

In [8]: pg['good_rb_game'].sum()
Out[8]: 40
```

Two boolean specific summary type columns are `.all` — which returns `True` if all values in the column are `True`, and `any`, which returns `True` if any values in the column are `True`.

For example, did anyone in here pass for over 400 yards?

```
In [9]: (pg['pass_yards'] > 400).any()
Out[9]: True
```

Yes. Did everyone finish their games with non-negative rushing yards?

```
In [10]: (pg['rush_yards'] >= 0).all()
Out[10]: False
```

No.

Like the other summary statistic columns, `any` and `all` take an `axis` argument.

For example, to look by row and see if a *player* went for over 100 yards rushing or receiving in a game we could do.

```
In [9]: (pg[['rush_yards', 'rec_yards']] > 100).any(axis=1)
Out[9]:
0       False
1       False
2       False
3       False
4       False
        ...
1426    False
1427    False
1428    False
1429    False
1430    False
```

We can see passing `axis=1` makes it return column of booleans indicating whether the player went for over 100 rushing OR receiving yards.

If we want, we can then call another function on *this* column to see how often it happened.

```
In [10]: (pg[['rush_yards', 'rec_yards']] > 100).any(axis=1).sum()
Out[10]: 100
```

How often did someone go for *both* 100 rushing AND receiving yards in the same game?

```
In [11]: (pg[['rush_yards', 'rec_yards']] > 100).all(axis=1).sum()
Out[11]: 0
```

Never, at least in our sample. Maybe we can lower the bar?

```
In [12]: (pg[['rush_yards', 'rec_yards']] > 75).all(axis=1).sum()
Out[12]: 4
```

Other Misc Built-in Summary Functions

Not all built-in, useful, raw data to stat Pandas functions just return one number. Another useful function is `value_counts`, which summarizes the frequency of individual values.

```
In [13]: adp['position'].value_counts()
Out[13]:
WR     63
RB     62
QB     21
TE     17
DEF    13
PK      8
```

Also useful is `crosstab`, which shows the cross frequency for all the combinations of *two* columns.

```
In [14]: pd.crosstab(adp['team'], adp['position']).head()
Out[14]:
position  DEF  PK  QB  RB  TE  WR
team
ARI         1   0   1   2   0   2
ATL         1   1   1   2   1   3
BAL         1   1   0   2   0   2
BUF         0   0   1   1   0   3
CAR         1   0   1   2   1   0
```

Just like we did with `str` methods in the last chapter, you should set aside some time to explore functions and types available in Pandas using the REPL, tab completion, and typing the name of a function followed by a question mark.

There are three areas you should explore. The first is high-level Pandas functions, which you can see by typing `pd.` into the REPL and tab completing. These include functions for reading various file formats, as well as the `DataFrame`, `Series`, and `Index` Python types.

You should also look at Series specific methods that operate on single columns. You can explore this by typing `pd.Series.` into the REPL and tab completing.

Finally, we have DataFrame specific methods — `head`, `mean` or `max` are all examples — which you can use on any DataFrame (we called them on `adp` above). You can view all of these by typing in `pd.DataFrame.` into the REPL and tab completing.

Review

In this section we learned about summary functions — `mean`, `max`, etc — that operate on DataFrames, including two — `any` and `all` — that operate on boolean data specifically. We learned how they can apply to columns (the default) or across rows (by setting `axis=1`).

We also learned about two other useful functions for viewing frequencies and combinations of values, `value_counts` and `pd.crosstab`.

Exercises

3.2.1 Load the player game data into a DataFrame named `pg`. You'll use it for the rest of the problems in this section.

3.2.2 Add a column to pg that gives the total rushing, receiving and passing yards. for each player-game. Do it two ways, one with basic arithmetic operators and another way using a built-in pandas function. Call them '`total_yards1`' and '`total_yards2`'. Prove that they're the same.

3.2.3

a) What were the average values for rushing and receiving yards?

b) How many times in our data did someone throw for over at least 300 and at least 3 tds?

c) What % of qb performances was that?

d) How many rushing tds were there total in our sample?

e) What week is most represented in our sample? Least?

3. Filter Observations

The third thing we can do with DataFrames is **filter** them, which means picking out a subset of data at the row level.

We'll learn how to filter based on criteria we set, as well as how to filter by dropping duplicates.

The examples for this section are in `03_03_filter.py`.

loc

One way to filter observations is to pass the *index value* you want to `loc[]` (note the brackets as opposed to parenthesis):

For example:

```
In [2]: adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))

In [3]: adp.set_index('player_id', inplace=True)

In [4]: tom_brady_id = 119

In [5]: adp.loc[tom_brady_id]
Out[5]:
adp                  29.6
adp_formatted        3.06
bye                    10
high                   14
low                    43
name            Tom Brady
position               QB
stdev                 5.8
team                   NE
times_drafted         193
```

Similar to how you select multiple columns, you can pass multiple values via a list:

```
In [6]: my_player_ids = [119, 1886, 925]

In [7]: adp.loc[my_player_ids]
Out[7]:
             adp  adp_formatted  bye  ...  stdev  team times_drafted
player_id                             ...
119         29.6           3.06   10  ...    5.8    NE           193
1886         3.7           1.04    7  ...    1.0   PIT           338
925         69.9           6.10   12  ...    7.4   ARI           187
```

While not technically filtering, you can also pass `loc` a second argument to limit which columns you return. This returns the `name`, `adp`, `stddev` columns for the ids we specified.

```
In [8]: adp.loc[my_player_ids, ['name', 'adp', 'stdev']]
Out[8]:
                     name   adp  stdev
player_id
119              Tom Brady  29.6    5.8
1886         Antonio Brown   3.7    1.0
925        Adrian Peterson  69.9    7.4
```

Like in other places, you can also pass the column argument of `loc` a single, non-list value and it'll return just the one column.

```
In [9]: adp.loc[my_player_ids, 'name']
Out[9]:
player_id
119            Tom Brady
1886       Antonio Brown
925      Adrian Peterson
```

Boolean indexing

Though `loc` can take specific index values (a list of player ids in this case), this isn't done that often. More common is **boolean indexing**, where you pass it a column of bool values and it returns only values where the column is `True`.

So say we're just interested in RBs:

```
In [10]: is_a_rb = adp['position'] == 'RB'

In [11]: is_a_rb.head()
Out[11]:
player_id
2297     True
1983     True
1886    False
1796    False
2343     True

In [12]: adp_rbs = adp.loc[is_a_rb]

In [13]: adp_rbs[['name', 'adp', 'position']].head()
Out[13]:
                     name  adp position
player_id
2297         David Johnson  1.3       RB
1983           LeVeon Bell  2.3       RB
2343       Ezekiel Elliott  6.2       RB
2144       Devonta Freeman  7.0       RB
1645          LeSean McCoy  7.8       RB
```

Boolean indexing requires that the column of booleans you're passing has the same index as the DataFrame we're calling `loc` on. In this case we already know `is_a_rb` has the same index as `adp` because that's how Pandas works.

Also, we broke it into two separate steps above, but that's not necessary:

```
In [14]: adp_df_wrs = adp.loc[adp['position'] == 'WR']

In [15]: adp_df_wrs[['name', 'adp', 'position']].head()
Out[15]:
                      name   adp position
player_id
1886          Antonio Brown   3.7       WR
1796            Julio Jones   5.7       WR
2113       Odell Beckham Jr   6.4       WR
2111             Mike Evans   7.9       WR
1795             A.J. Green  10.0       WR
```

Having to refer to the name of your DataFrame (`adp` here) multiple times may seem cumbersome at first. It can get a bit verbose, but this is a very common thing to do so get used to it.

Any bool column or operation on it works.

```
In [16]: is_a_te = adp['position'] == 'TE'

In [16]: adp_not_te = adp.loc[~is_a_te]

In [17]: adp_not_te[['name', 'adp', 'position']].head()
Out[17]:
                     name  adp position
player_id
2297         David Johnson  1.3       RB
1983           LeVeon Bell  2.3       RB
1886         Antonio Brown  3.7       WR
1796           Julio Jones  5.7       WR
2343       Ezekiel Elliott  6.2       RB
```

Duplicates

A common way to filter data is by removing duplicates. Pandas has built-in functions for this.

```
In [18]: adp.shape
Out[18]: (184, 10)

In [19]: adp.drop_duplicates(inplace=True)
```

Though in this case it didn't do anything since we had no duplicates.

```
In [20]: adp.shape
Out[20]: (184, 10)
```

We can see we have the same number of rows as before.

The `drop_duplicates` method drops duplicates across *all* columns, if you are interested in dropping only a subset of variables you can specify them:

```
In [21]: adp.drop_duplicates('position')[['name', 'adp', 'position']]
Out[21]:
                    name    adp position
player_id
2297         David Johnson    1.3       RB
1886         Antonio Brown    3.7       WR
1740       Rob Gronkowski   18.1       TE
1004         Aaron Rodgers   23.4       QB
1309       Denver Defense  105.5      DEF
1962         Justin Tucker  145.0       PK
```

Note, although it though it might look strange to have `[['name', 'adp', 'position']]` immediately after `.drop_duplicates('position')`, it works because the result of `adp.`

`drop_duplicates(...)` is a DataFrame. We're just immediately using the multiple column bracket syntax immediately after we call it to pick out and print certain columns.

Alternatively, to identify — but not drop — duplicates you can do:

```
In [22]: adp.duplicated().head()
Out[22]:
player_id
2297    False
1983    False
1886    False
1796    False
2343    False
```

The `duplicated` method returns a bool column indicating whether the row is a duplicate (none of these are). Note how it has the same index as our original DataFrame.

Like `drop_duplicates`, you can pass it a subset of variables and it will only look at those, or you can just call it only on the columns you want to check.

```
In [23]: adp['position'].duplicated().head()
Out[23]:
player_id
2297    False
1983     True
1886    False
1796     True
2343     True
```

By default, `duplicated` *only* identifies the duplicate observation — not the original — so if you have two "T. Smith"'s in your data `duplicated` will indicate `True` for the second one. You can tell it to identify *both* duplicates by passing `keep=False`.

Combining filtering with changing columns

Often you'll want to combine the previous sections and update columns *only* for certain rows.

For example, say we want a column `'primary_yards'` with passing yards for QBs, rushing yards for RBs, and receiving yards for WRs.

We would do that using something like this:

```
In [24]: pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv
    '))

In [25]: pg['primary_yards'] = np.nan

In [26]: pg.loc[pg['pos'] == 'QB', 'primary_yards'] = pg['pass_yards']

In [27]: pg.loc[pg['pos'] == 'RB', 'primary_yards'] = pg['rush_yards']

In [28]: pg.loc[pg['pos'] == 'WR', 'primary_yards'] = pg['rec_yards']
```

We start by creating an empty column, `primary_yards`, filled with missing values (remember, missing values have a value of `np.nan` in Pandas, which is a numpy datatype called "not a number"). Then we go through and use `loc` to pick out only the rows (where `pos` equals what we want) and the one column (`primary_yards`) and assign the correct value to it.

```
In [29]:
pg[['player_name', 'pos', 'pass_yards', 'rush_yards', 'rec_yards',
    'primary_yards']].sample(5)
Out[29]:
     player_name pos  pass_yards  rush_yards  rec_yards  primary_yards
0         T.Brady  QB       267.0         0.0        0.0          267.0
798     T.Lockett  WR         0.0         0.0       25.0           25.0
1093     D.Thomas  WR         0.0         0.0       67.0           67.0
84         K.Hunt  RB         0.0       155.0       51.0          155.0
766      J.Graham  TE         0.0         0.0       45.0            NaN
```

Query

The `loc` method is flexible, powerful and can do everything you need — including letting you update columns only for rows with certain values (the last section). But, if you're just interested in filtering, there's alternative to `loc` that isn't as wordy: `query`.

To use query you pass it a string. Inside the string you can refer to variable names and do normal Python operations.

For example, to return a DataFrame of only RBs:

```
In [30]: pg.query("pos == 'RB'").head()
Out[30]:
       gameid   player_id  carries  ...  player_name  week  primary_yards
3  2017090700  00-0030288      3.0  ...   R.Burkhead     1           15.0
5  2017090700  00-0033923     17.0  ...       K.Hunt     1          148.0
7  2017091705  00-0030288      2.0  ...   R.Burkhead     2            3.0
8  2017091705  00-0025394      8.0  ...   A.Peterson     2           26.0
9  2017091705  00-0027966      8.0  ...     M.Ingram     2           52.0
```

Note — inside the string — we're referring to position without quotes, like a variable name. We can refer to any column name like that. We can also call boolean columns directly:

```
In [31]: pg['is_a_rb'] = pg['pos'] == 'RB'

In [32]: pg.query("is_a_rb").head()
Out[32]:
       gameid   player_id  carries  ...  week  primary_yards  is_a_rb
3  2017090700  00-0030288      3.0  ...     1           15.0     True
5  2017090700  00-0033923     17.0  ...     1          148.0     True
7  2017091705  00-0030288      2.0  ...     2            3.0     True
8  2017091705  00-0025394      8.0  ...     2           26.0     True
9  2017091705  00-0027966      8.0  ...     2           52.0     True
```

Another thing we can do is use basic Pandas functions. For example, to filter based on whether `pg['raw_yac']` is missing:

```
In [33]: pg.query("raw_yac.notnull()")[['gameid', 'player_id',
         'raw_yac']].head()
Out[33]:
       gameid   player_id  raw_yac
0  2017090700  00-0019596      0.0
1  2017090700  00-0023436      0.0
2  2017090700  00-0026035     49.0
3  2017090700  00-0030288      7.0
4  2017090700  00-0030506     23.0
```

Note: if you're getting an error in this, try passing `engine='python'` to query. That's required on some systems.

Again, `query` doesn't add anything beyond what you can do with `loc` (I wasn't even aware of `query` until I started writing this book), and there are certain things (like updating columns based on values in the row) that you can *only* do with `loc`. So I'd focus on that first. But, once you have `loc` down, `query` can let you filter data a bit more concisely.

Review

In this section we learned about how to limit our data to fewer *rows*, i.e. filter it. We learned how to filter by passing both specific index values and a column of booleans to `loc`. We also learned about identifying and dropping duplicates. Finally, we learned about `query`, which is a shorthand way to filter your data.

Exercises

3.3.1 Load the ADP data into a DataFrame named `adp`. You'll use it for the rest of the problems in this section.

3.3.2 Make smaller DataFrame with just Dallas Cowboy players and only the columns: `'name'`, `'position'`, `'adp'`. Do it two ways: (1) using the `loc` syntax, and (b) another time using the `query` syntax. Call them `adp_cb1` and `adp_cb2`.

3.3.3 Make a DataFrame `adp_no_cb` that is everyone EXCEPT Dallas players, add the `'team'` column to it.

3.3.4

a) Are there any duplicates by last name AND position in our `adp` DataFrame? How many?

b) Divide `adp` into two separate DataFrames `adp_dups` and `adp_nodups`, one with dups (by last name and pos) one without.

3.3.5 Add a new column to `adp` called `'adp_description'` with the values:

- `'stud'` for adp's < 40
- `'scrub'` for adp's > 120
- missing otherwise

3.3.6 Make a new DataFrame with only observations for which '`adp_description`' is missing. Do this with both the (a) loc and (b) query syntax. Call them `adp_no_desc1` and `adp_no_desc2`.

4. Change Granularity

The fourth thing to do with DataFrames is change the granularity. The examples for this section are in the file `03_04_granularity.py`.

Remember: data is a collection of structured information. Granularity is the level your collection is at. Our ADP data is at the player level; each row in `adp` is one player. In `pg` each row represents a single player-game.

Ways of changing granularity

It's common to want to change the level of granularity in your data. There are two ways to do this:

1. **Grouping** involves going from fine grained data (e.g. play) to a more course grained (e.g. game). It necessarily involves a loss of information; once my data is at the game level I have no way of knowing what happened any particular play.
2. **Stacking** or **unstacking** — sometimes called reshaping — is less common. It involves no loss of information, essentially by cramming data that was formerly unique row into separate columns. So say I have a dataset of weekly fantasy points at the player-game level. I could move things around so my data is one line for every player, but now with 17 separate columns (week 1's fantasy points, week 2's, etc).

Grouping

Grouping fine grained data to less fine grained data is very common. Examples would include going from play to game data or from player-game data to player-season data.

Grouping necessarily involves some function that says HOW your data should go to this more to less granular state.

So if we have have play level data with yards and wanted to go to the game level we could:

- sum for total yards for the game
- average yards per rush for ypc
- take the max, min for some level of floor/ceiling
- take the count for total number of rushes

Or we could also do something more sophisticated, maybe the percentage of runs that went for more than 5 yards[3].

Let's use our play-by-play sample (every play from the 2018 KC-NE and KC-LA games in 2018, loaded earlier) to look at some examples.

```
In [2]: pbp = pd.read_csv(path.join(DATA_DIR, 'play_data_sample.csv'))
```

Pandas handles grouping via the `groupby` function.

[3] In practice, this would involve multiple steps. First we'd create a bool column indicating whether a rushing play went for more than 5 yards. Then we'd take the average of that column to get a percentage between 0-1.

```
In [3]: pbp.groupby('game_id').sum()
Out[3]:
            play_id  posteam_score  ...         wp       wpa
game_id                             ...
2018101412   287794         2269.0  ...  72.384102  1.374290
2018111900   472385         3745.0  ...  76.677250  0.823359
```

This gives us a DataFrame where every column is summed over `game_id`. Note how `game_id` is the index of our newly grouped-by data. This is the default behavior; you can turn it off either by calling `reset_index` right away or passing `as_index=False` to `groupby`.

Also note calling `sum` on our groupby gives us the sum of every column. Usually though we'd only be interested in a subset of variables, e.g. maybe yards or attempts for sum.

```
In [4]: sum_cols = ['yards_gained', 'rush_attempt',
                    'pass_attempt', 'shotgun']

In [5]: pbp.groupby('game_id').sum()[sum_cols]
Out[5]:
            yards_gained  rush_attempt  pass_attempt  shotgun
game_id
2018101412           946          55.0          73.0       85
2018111900          1001          41.0         103.0      101
```

Or we might want to take the sum of yards, and use a different function for other columns. We can do that using the `agg` function, which takes a dictionary.

```
In [6]:
pbp.groupby('game_id').agg({'yards_gained': 'sum', 'play_id':
    'count', 'interception': 'sum', 'touchdown': 'sum'})
Out[6]:
            yards_gained  play_id  interception  touchdown
game_id
2018101412           946      144           2.0        8.0
2018111900          1001      160           3.0       14.0
```

Note how the names of the resulting grouped by columns are the same as whatever the original column was. But after a groupby that name may no longer be relevant. For instance,`play_id` isn't the best name for the total number of plays run, which is what this column gives us post grouping.

As of this writing, the newest version of Pandas (0.25) let's you pass new variable names to `agg` as keywords, with values being variable, function tuple pairs (for our purposes, a tuple is like a list but with parenthesis — a tuple pair is a tuple with two items in it). The following code is the same as the above, but renames `play_id` to `nplays`.

```
In [6]:
pbp.groupby('game_id').agg(
    yards_gained = ('yards_gained', 'sum'),
    nplays = ('play_id', 'count'),
    interception = ('interception', 'sum'),
    touchdown = ('touchdown', 'sum'))

Out[6]:
            yards_gained  nplays  interception  touchdown
game_id
2018101412           946     144           2.0        8.0
2018111900          1001     160           3.0       14.0
```

Again, this won't work unless you have the newest version of Pandas installed.

The `agg` function also lets you calculate multiple statistics per column. Maybe we want to get total and average yards.

```
In [7]: pbp.groupby('game_id').agg({'yards_gained': ['sum', 'mean']})
Out[7]:
           yards_gained
                    sum      mean
game_id
2018101412          946  6.569444
2018111900         1001  6.256250
```

You can also group by more than one thing — for instance, game AND team — by passing `groupby` a list.

```
In [8]: yards_per_team_game = pbp.groupby(
         ['game_id', 'posteam']).agg({'yards_gained': ['sum', 'mean']})

In [9]: yards_per_team_game
Out[9]:
                   yards_gained
                            sum      mean
game_id    posteam
2018101412 KC               446  7.689655
           NE               500  6.250000
2018111900 KC               546  7.479452
           LA               455  5.617284
```

A note on multilevel indexing

Grouping by two or more variables shows that it's possible in Pandas to have a *multilevel* index, where your data by two or more variables. In the example above, our index is a combination of`game_id` and `posteam`.

You can still use the `loc` method with multi-level indexed DataFrames, but you need to pass it a tuple (another collection type that's like a list except with parenthesis instead of brackets):

```
In [10]: yards_per_team_game.loc[[(2018101412, 'NE'), (2018111900, 'LA')]]
Out[10]:
                    yards_gained
                             sum  mean
game_id    posteam
2018101412 NE                500  6.250000
2018111900 LA                455  5.617284
```

I personally find multilevel indexes unwieldy and don't use them that often, however there are situations where they are the only way to do what you want, like in stacking and unstacking.

Stacking and unstacking data

I would say stacking and unstacking doesn't come up *that* often, so if you're having trouble with this section or feeling overwhelmed, feel free to make mental note of what it is broadly and come back to it later.

Stacking and unstacking data technically involves changing the granularity of our data, but instead of applying some function (sum, mean) to aggregate, we're just moving data from rows to columns or vis versa.

For example, let's look total passing yards by week in our player game data.

```
In [11]: pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv
    '))

In [12]: qbs = pg.loc[pg['pos'] == 'QB', ['player_name', 'week', 'pass_tds
    ']]

In [13]: qbs.sample(5)
     player_name  week  pass_tds
214     T.Taylor    10       0.0
339       M.Ryan    14       1.0
220     T.Taylor    11       1.0
70       T.Brady    15       1.0
1192    C.Newton     1       2.0
```

But say instead of being at the player, game level and having one column for passing touchdowns, we want it to be at the player level and have up to 17 separate columns of points. Then we'd do:

```
In [14]: qbs_reshaped = qbs.set_index(['player_name', 'week']).unstack()

In [15]: qbs_reshaped.head()
Out[15]:
                pass_tds       ...
week                   1    2  ...    12   13   14   15   16   17
player_name                    ...
A.Smith              4.0  1.0  ...   1.0  4.0  1.0  2.0  1.0  NaN
B.Bortles            1.0  1.0  ...   0.0  2.0  2.0  3.0  3.0  0.0
B.Hundley            NaN  NaN  ...   3.0  0.0  3.0  NaN  0.0  1.0
B.Roethlisberger     2.0  2.0  ...   5.0  2.0  2.0  3.0  2.0  NaN
C.Newton             2.0  0.0  ...   0.0  2.0  1.0  4.0  0.0  1.0
```

This move doesn't cost us any information. Initially we find passing touchdowns for a particular QB and week by finding the right row. After we unstack (or reshape it) it we can find total passing tds for a particular player and week by finding the player's row, then finding the column for the right week.

This lets us do things like calculate season totals:

```
In [16]: total_tds = qbs_reshaped.sum(axis=1).head()

In [17]: total_tds
Out[17]:
player_name
A.Smith             27.0
B.Bortles           22.0
B.Hundley           10.0
B.Roethlisberger    32.0
C.Newton            23.0
```

If we want to undo this operation and to stack it back up we could do:

```
In [18]: qbs_reshaped.stack().head()
Out[18]:
                 pass_tds
player_name week
A.Smith     1         4.0
            2         1.0
            3         2.0
            4         1.0
            5         3.0
```

Review

In this section we learned about how to change the granularity of our data. We can do that two ways, by grouping — where the granularity of our data goes from fine to less-fine grained — and by stacking and unstacking, where we shift data from columns to row or vis versa.

Exercises

3.4.1 How does shifting granularities affect the amount of information in your data? Explain.

3.4.2

a) Load the play by data.

b) Figure out who the leading rusher in each game was and how many rushing yards they had.

c) Figure out everyone's ypc.

d) What portion of everyone's runs got 0 or negative yardage?

3.4.3 Using the play by play data, run `pbp.groupby('game_id').count()`.

Based on the results, explain what you think it's doing. How does `count` differ from `sum`? When would `count` and `sum` give you the same result?

3.4.4 In `pbp`, which team/game had the highest percentage of their plays go for first downs? Turnovers?

3.4.5 How does stacking or unstacking affect the amount of information in your data? Explain.

5. Combining two or more DataFrames

The last thing we need to know how to do with DataFrames is combine them. The examples for this section are in `03_05_combine.py`.

We've already combined DataFrames with single-columned Series that have the same index. In this section we'll learn how to combine multiple-column DataFrames together, whether they have the same index or not.

We'll also learn about sticking data on top of each other.

There are three main factors to be aware of when combining DataFrames. They are:

1. The columns you're joining on.
2. Whether you're doing a 1:1, 1:many (or many:1), or many:many join.
3. What you're doing with unmatched observations.

Let's go over each of these.

1. The columns you're joining on.

Say we want to analyze whether QBs throw for more yards when they're playing at home. But we currently don't have home or away info in our player-game DataFrame. Instead, it's in a separate games file.

```
In [2]: pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'
    ))

In [3]: games = pd.read_csv(path.join(DATA_DIR, 'game_2017_sample.csv'))
```

Recall the basics of tabular data. Each row is some item in a collection, and each column some piece of information.

When merging data, we want to extend our rows out wider and get *more* information. To be able to do that we need to have some piece of information in common that we can link on.

In this example, it'd be `gameid`.

The `pg` data has a `gameid` field which tells us which game that row is from. The `games` data has this same `gameid` field, along with other information about the game, including home and away.

So we can do:

```
In [4]: pd.merge(pg, games[['gameid', 'home', 'away']]).head()
Out[4]:
       gameid   player_id  carries  ...  week  home  away
0  2017090700  00-0019596      0.0  ...     1    NE    KC
1  2017090700  00-0023436      2.0  ...     1    NE    KC
2  2017090700  00-0026035      0.0  ...     1    NE    KC
3  2017090700  00-0030288      3.0  ...     1    NE    KC
4  2017090700  00-0030506      1.0  ...     1    NE    KC
```

If we didn't have the column `gameid` in either of the tables, we wouldn't be able to join these. Our program has no way of knowing which `pg` row is connected to which `game` row and can't do the merge. There'd be no overlapping columns between tables.

Even if you have the overlapping columns, keep in mind your program is only going to be able to link them when values of each column are *exactly* the same.

If your name column in one DataFrame has "Odell Beckham Jr." and the other has "Odell Beckham" or just "O.Beckham" or "OBJ", they won't be merged. It's your job to modify one or both of them (using the column manipulation functions we talked about in this chapter) to make them the same so you can combine them properly. If you're new to working with data might be surprised at how much of your time is spent doing things like this.

Another issue to watch out for is inadvertent duplicates. For example, if you're combining two DataFrames on a `name` column and you have two "J.Jones" (J.J. and Julio), that will lead to unexpected behavior that could cause problems. That's why it's usually best to merge on a unique id variable if you can.

So far we've been talking about merging on a single column, but merging on more than one column is perfectly OK too. For example, say we have separate rushing and receiving DataFrames:

```
In [5]: rush_df = pg[['gameid', 'player_id', 'rush_yards', 'rush_tds']]

In [6]: rec_df = pg[['gameid', 'player_id', 'rec_yards', 'rec_tds']]
```

And want to combine them:

```
In [7]: combined = pd.merge(rush_df, rec_df, on=['player_id', 'gameid'])

In [8]: combined.head()
Out[8]:
       gameid   player_id  rush_yards  rush_tds  rec_yards  rec_tds
0  2017090700  00-0019596         0.0       0.0        0.0      0.0
1  2017090700  00-0023436         9.0       0.0        0.0      0.0
2  2017090700  00-0026035         0.0       0.0      100.0      0.0
3  2017090700  00-0030288        15.0       0.0        8.0      0.0
4  2017090700  00-0030506         4.0       0.0       40.0      0.0
```

The `merge` function in Pandas takes an `on` argument where you can specify the column to match on. If you don't explicitly specify `on`, Pandas will attempt to merge whatever columns the two DataFrames have in common.

2. Whether you're doing a one-to-one, one-to-many, or many-to-many merge

In the preceding example, we merged two DataFrames together on `player_id` and `gameid`. Neither DataFrame had duplicates; e.g. they each had *one* row with a gameid of `2017090700` and player_id of `'00-0019596'`, so that was a one-to-one merge.

One-to-one merges are straight forward; all DataFrames involved (the two we're merging, plus the final product) are at the same level of granularity.

This is not the case with one-to-many (or many-to-one, which is the same thing — order doesn't matter) merges.

Say we've done our analysis on our combined rushing and receiving yard data from above, and now we want to add back in some information so we can see who we're talking about. The player table includes info on name, team and position.

```
In [9]: player = pd.read_csv(path.join(DATA_DIR, 'player_2017_sample.csv')
   )

In [10]: player['player_id'].duplicated().any()
Out[10]: False
```

The same cannot be said for the player-game data.

```
In [11]: pg['player_id'].duplicated().any()
Out[11]: True
```

These DataFrames are *not* at the same level; each row in the first one denotes a player-game combination, while each row in the second is one player.

They both have `player_id` in common, so it's no problem to them together. It's just that every one `player_id` in the `player` DataFrame is being matched to *many* `player_id`s in the player-game table. This is a one-to-many join.

```
In [12]: pd.merge(combined, player).head()
Out[12]:
       gameid   player_id  rush_yards  ...  team  pos  player_name
0  2017090700  00-0019596         0.0  ...    NE   QB      T.Brady
1  2017091705  00-0019596         9.0  ...    NE   QB      T.Brady
2  2017092407  00-0019596         6.0  ...    NE   QB      T.Brady
3  2017100107  00-0019596         2.0  ...    NE   QB      T.Brady
4  2017100500  00-0019596         5.0  ...    NE   QB      T.Brady
```

One-to-many joins are common, especially when data is efficiently stored. We don't really need to store name, position and team for every single line in our player-game table when we can easily merge it back in with a one-to-many join on a player table.

The result of a one-to-many join is a table at the same level of granularity at the most granular input table. So if we combine a player table with a player-game table, the resulting data will be at the player-game level.

Finally, although it's technically possible to many-to-many joins, in my experience this is almost always done unintentionally, usually when merging on columns that you didn't realize you had duplicates in. There are situations where something like this might be useful that we'll touch on more in the SQL section.

Note how the Pandas merge commands for 1:1 and 1:m merges look exactly the same. Pandas will take care of combining these automatically, depending on what your data looks like.

If you want, you can pass the type of merge you're expecting in the `validate` keyword. If you do that, Pandas will throw an error if the merge isn't what you say it should be.

Let's try that last example doing that. We know this isn't really a 1:1 merge, so we should get an error.

```
In [177]: pd.merge(combined, player, validate='1:1')
...
MergeError: Merge keys are not unique in left dataset; not a one-to-one
    merge
```

Perfect.

Something you *do* have control over is how you handle unmatched observations. Let's get to that next.

3. What you do with unmatched observations

Logically, there's no requirement that two tables have to include information about the exact same observations. For example, maybe we have rushing and receiving data that we'd like to combine. Let's keep positive values only.

```
In [14]: rush_df = pg.loc[pg['rush_yards'] > 0,
           ['gameid', 'player_id', 'rush_yards', 'rush_tds']]

In [15]: rec_df = pg.loc[pg['rec_yards'] > 0,
           ['gameid', 'player_id', 'rec_yards', 'rec_tds']]

In [16]: rush_df.shape
Out[16]: (555, 4)

In [17]: rec_df.shape
Out[17]: (1168, 4)
```

Many players in the rushing table (not all) will also have some receiving stats. Many players in the receiving table (i.e. the TEs) won't be in the rushing table.

When you merge these, Pandas defaults to keeping observations that are only in *both* tables.

```
In [18]: comb_inner = pd.merge(rush_df, rec_df)

In [19]: comb_inner.shape
Out[19]: (355, 6)
```

Alternatively, we can keep everything in the left (`rush_df`) or right (`rec_df`) table by passing `'left'` or `'right'` to the `how` argument.

```
In [20]: comb_left = pd.merge(rush_df, rec_df, how='left')

In [21]: comb_left.shape
Out[21]: (555, 6)
```

Where "left" and "right" just denote the order we passed them into `merge` (first one is left, second right). Note that the result of our left merge has exactly the same number of rows as our left DataFrame pre-join. This is what we'd expect.

Finally, we can also do an "outer" merge, where we keep observations that match, as well as unmatched data from both tables. One thing I find helpful when doing non-inner joins is to include `indicator=True`, which adds a column `_merge` indicating whether the observation was in the left DataFrame, right DataFrame, or both.

```
In [22]: comb_outer = pd.merge(rush_df, rec_df, how='outer', indicator=
    True)

In [23]: comb_outer.shape
Out[23]: (1368, 7)

In [24]: comb_outer['_merge'].value_counts()
Out[24]:
right_only    813
both          355
left_only     200
```

This tells us that — out of the 1368 player-game observations in our sample — 813 didn't have any rushing yards, 200 didn't have any receiving yards, and 355 had both.

More on pd.merge

All of our examples so far have been neat merges on similarly named id columns, but that won't always be the case. It's common that the columns you're joining on have two different names.

```
In [25]: rush_df = pg.loc[pg['rush_yards'] > 0,
     ['gameid', 'player_id', 'rush_yards ', 'rush_tds']]

In [26]: rush_df.columns = ['gameid', 'rb_id', 'rush_yards', 'rush_tds']
```

(Note: assigning the `columns` attribute of a DataFrame a new list is one way to rename columns.)

But now if we want to combine these the column is `rb_id` in `rush_df` and `player_id` in `rec_df`. What to do? Simple, just use `left_on` and `right_on` instead of `on`.

```
In [27]:
pd.merge(rush_df, rec_df, left_on=['gameid', 'rb_id'],
    right_on=['gameid', 'player_id']).head()
--
Out[27]:
       gameid       rb_id  rush_yards  ...   player_id rec_yards  rec_tds
0  2017090700  00-0030288         NaN  ...  00-0030288       8.0      0.0
1  2017090700  00-0030506         NaN  ...  00-0030506      40.0      0.0
2  2017090700  00-0033923         NaN  ...  00-0033923      98.0      2.0
3  2017091705  00-0030288         NaN  ...  00-0030288      41.0      1.0
4  2017091705  00-0027966         NaN  ...  00-0027966      18.0      0.0
```

Sometimes you might want attach one of the DataFrames you're merging by its index. That's also no problem.

```
In [28]: max_rush_df = rush_df.groupby('playerid')[['rush_yards', '
    rush_tds']].max()

In [29]: max_rush_df.head()
Out[29]:
            rush_yards  rush_tds
playerid
00-0019596        14.0       1.0
00-0022924        25.0       0.0
00-0023436        70.0       1.0
00-0023500       130.0       1.0
00-0024389         1.0       1.0

In [30]: max_rush_df.columns = [f'max_{x}' for x in max_rush_df.columns]

In [31]: max_rush_df.columns
Out[31]: Index(['max_rush_yards', 'max_rush_tds'], dtype='object')
```

Above, we grouped our rushing DataFrame to get max rush yards and touchdowns by player. We then renamed the columns by sticking `max_` on the front of each. If you're confused about that line look back at list comprehensions and f-strings.

Because we grouped on `player_id`, that's the level `max_rush_df` is at, and `player_id` is the index.

What if we want to add this back into our original DataFrame? They both have `player_id`, but one is as an index, one a regular column. No problem.

```
In [32]: pd.merge(rush_df, max_rush_df, left_on='rb_id',
          right_index=True).head()
Out[32]:
          gameid       rb_id  ...  max_rush_yards  max_rush_tds
1     2017090700  00-0023436  ...             NaN           1.0
86    2017091704  00-0023436  ...             NaN           1.0
93    2017092412  00-0023436  ...             NaN           1.0
99    2017100200  00-0023436  ...             NaN           1.0
102   2017100811  00-0023436  ...             NaN           1.0
```

One thing to be aware of when using `merge` is that the resulting data has a new, reset index. If you think about it, this makes sense — presumably you're using `merge` because you want to combine two, non-identically index DataFrames. If that's the case, how would Pandas know which of their indexes to use for the resulting, final DataFrame? It doesn't, and so resets it to a 0, 1, … sequence.

But if you're relying on Pandas indexing to automatically align everything for you, and doing some merging in between, this is something you'll want to watch out for, because it can lead to some unexpected results.

pd.concat()

If you're combining two DataFrames with the same index, you can concatenate them with pd.concat() instead of merging them.

That works like this:

```
In [33]:
rush_df = (pg.loc[pg['rush_yards'] > 0,
                  ['gameid', 'player_id', 'rush_yards', 'rush_tds']]
           .set_index(['gameid', 'player_id']))

In [34]:
rec_df = (pg.loc[pg['rec_yards'] > 0,
                 ['gameid', 'player_id', 'rec_yards', 'rec_tds']]
          .set_index(['gameid', 'player_id']))

In [35]: pd.concat([rush_df, rec_df], axis=1).head()
Out[35]:
                       rush_yards  rush_tds  rec_yards  rec_tds
gameid     player_id
2017090700 00-0023436         9.0       0.0        NaN      NaN
           00-0026035         NaN       NaN      100.0      0.0
           00-0030288        15.0       0.0        8.0      0.0
           00-0030506         4.0       0.0       40.0      0.0
           00-0033923       148.0       1.0       98.0      2.0
```

Note we're not passing the DataFrames in one at a time. Instead we're passing in a list of DataFrames — all with the same index — and sticking them together. What that means is combine three or more DataFrames together using concat, whereas merge only lets you do two.

For example, we could add in a passing DataFrame here:

```
In [36]:
pass_df = (pg.loc[pg['pass_yards'] > 0,
                  ['gameid', 'player_id', 'pass_yards', 'pass_tds']]
           .set_index(['gameid', 'player_id']))

In [37]: pd.concat([rush_df, rec_df, pass_df], axis=1).head()
Out[37]:
                       rush_yards  rush_tds  ...  pass_yards  pass_tds
gameid     player_id                        ...
2017090700 00-0019596         NaN       NaN  ...       267.0       0.0
           00-0023436         9.0       0.0  ...       368.0       4.0
           00-0026035         NaN       NaN  ...         NaN       NaN
           00-0030288        15.0       0.0  ...         NaN       NaN
           00-0030506         4.0       0.0  ...         NaN       NaN
```

When you pass `axis=1`, concat sticks the DataFrames together side by side horizontally (similar to

`merge`). That isn't the default behavior (`axis=0`), which is to stick the DataFrames on top of each other.

Apart from axis the only other keyword is `how`, which can be `'inner'` or `'outer'` and specifies how you want to handle unmatched observations.

Both `merge` and `concat` provide some of the same functionality. I usually stick with `merge` for straightforward, two DataFrame joins since it's more powerful. But `concat` comes up often too, particularly when I need to combine more than two DataFrames.

It's also the only way to stick two DataFrames on top of each other:

```
In [38]: adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))

In [39]: qbs = adp.loc[adp['position'] == 'QB']

In [40]: rbs = adp.loc[adp['position'] == 'RB']

In [41]: qbs.shape
Out[41]: (21, 11)

In [42]: rbs.shape
Out[42]: (62, 11)

In [43]: pd.concat([qbs, rbs]).shape
Out[43]: (83, 11)
```

In this case, we know `qbs` and `rbs` don't have any index values in common (because we set the index to be `player_id` earlier in the chapter), but often that's not the case. If it isn't, we can pass, `ignore_index=True` and `concat` will stick both on top of each other and reset the index to 0, 1, …

Review

In this section we learned about combining DataFrames. We first covered `pd.merge` for joining two DataFrames with one or more columns in common. We talked about specifying those columns in the `on`, `left_on` and `right_on` arguments, and how to control what we do with unmatched observations by setting how equal to `'inner'`, `'outer'`, `'left'` or `''right'`.

We also talked about `pd.concat` and how the default behavior is to stick two DataFrames on top of each other (optionally setting `ignore_index=True`). We also talked about, separately, `pd.concat` can let you combine two DataFrames with the same index left to right via the `axis=1` keyword.

Exercises

3.5.1

a) Load the three datasets in in `./data/problems/combine1/`.

They contain touch (carries and receptions), yardage (rushing and receiving), and touchdown (rushing and receiving) data.

Combine them: b) using `pd.merge`. c) using `pd.concat`.

Note players are only in the touchdown data if they got at least one rushing or receiving touchdowns. Make sure your final combined data includes all players, even if they didn't score any touchdowns. If a player didn't score a touchdown, make sure number of rushing and receiving touchdowns are set to 0.

d) Which do you think is better here, `pd.merge` or `pd.concat`?

3.5.2

a) Load the four datasets in in `./data/problems/combine2/`. It contains the same data, but split "vertically" by position instead of horizontally.

b) Combine them.

3.5.3.

a) Load the ADP data.

b) Write a two line for loop to save subsets of the ADP data frame for each position to the `DATA_DIR`.

c) Then using `pd.concat` and list comprehensions, write one line of Python that loads these saved subsets and combines them.

4. SQL

Introduction to SQL

This section is on databases and SQL, which cover the second (storing data) and third (loading data) sections of our high level data analysis process respectively.

They're in one chapter together because they go hand in hand (that's why they're sometimes called "SQL databases"): once you have data in a database, SQL is how you get it out. Similarly, you can't use SQL unless you have a database to use it on.

This chapter might seem redundant given we've been storing and loading data already: the book came with some csv files, which we've already loaded in Pandas. What advantages do databases and SQL give us over that?

We'll start there.

How to Read This Chapter

This chapter — like the rest of the book — is heavy on examples. All the examples in this chapter are included in the Python file `04_sql.py`. Ideally, you would have this file open in your Spyder editor and be running the examples (highlight the line(s) you want and press F9 to send it to the REPL/console) as we go through them in the book.

Databases

Why do we need databases — can't we just store all our data in one giant spreadsheet?

The main issue with storing everything in one giant spreadsheet is that things can get unwieldy very quickly. For example, say we're building a model to project weekly fantasy points. This data is at the player and week level, but we might want to include less granular data — about the *player* (position, years in league, height) or game (home/away, weather).

When it comes time to actually run the model, we'll want all those values filled in for every row, but there's no reason we need to *store* the data like that.

Think about it: a player's position (usually) and number of years in the league stay the same every week, and whether they're home or away is the same for every player playing in a given game. It would be more efficient to store it as:

- One player table with just name, team, position, height, weight, year they came in the league.
- One game table with week, home and away team.
- Another player-game table with ONLY the things that vary at this level: points, yards, number of touchdowns, etc.

In each one we would want an id column — i.e. the player table would have a "player id", the game table a "game id", and the player-game table both player and game id — so we could link them back together when it's time to run our model.

This process of storing the minimal amount data is called **normalizing** your data. There are at least two benefits:

First, storing data in one place makes it easier to update it. For example, what if our initial dataset was incorrect had the home and away teams wrong for the Chicago-Green Bay Week 1. If that information is in a single `games` table, we can fix it there, rerun our code, and have it propagate through our analysis. That's preferable to having it stored on multiple rows in multiple tables, all of which would need to be fixed.

This applies to code as well as data. There's a programming saying, "don't repeat yourself" (abbreviated DRY). It's easier to change things in one spot than all over, which is why we use variable constants and reusable functions in our code.

The other advantage data is a smaller storage footprint. Storage space isn't as big of a deal as it has been in the past, but data can get unwieldy. Take our play by play data; right now we have mostly play-specific information in there (the down, time left, how many yards it went for, pass or run). But imagine if we had to store every single thing we might care about — player height, weight, where they went to college, etc — on every single line.

It's much better to keep what varies by play in the play data, then link it up to other data when we need it.

OK, so everything in one giant table isn't ideal, but what about just storing each table (play, game, player-game etc) in its own csv file. Do things really need to be in a SQL database?

Multiple csv files is better than one giant one, and honestly I don't care that much if you want to do that (I do it myself for quick, smaller projects), but it means loading your data in Pandas, and THEN

doing your merging there. That's fine, but joining tables is what SQL is good at. It's why they're called "relational databases", they keep track of relationships.

The other thing is SQL is good at letting you pick out individual columns and just the data you need. In Pandas that would be another extra step.

Finally, it doesn't matter that much for the size of the datasets we'll be working with, but it can be easier using SQL (or SQL-like tools) for very large datasets.

SQL Databases

SQL database options include Postgres, SQLite, Microsoft SQL Server, and MySQL, among others. Postgres is open source and powerful, and you might want to check it out if you're interested in going beyond this chapter.

Many databases can be complicated to set up and deal with. All of the above require installing a database *server* on your computer (or on another computer you can connect to), which runs in the background, interprets the SQL code you send it, and returns the results.

The exception is SQLite, which requires no server and just sits as a file on disk. There whatever analysis program you're using can access it directly. Because it's more straight forward, that's what we'll use here.

A Note on NoSQL

Sometimes you'll hear about NoSQL databases, the most common of which is MongoDB. We won't use them here, but for your reference, NoSQL databases are databases that store data as (nested) "documents" similar to Python's dictionaries. This dictionary-like data format is also called JSON (JavaScript object notation).

NoSQL databases are flexible and can be good for storing information suited to a tree-like structure. They also can be more performant in certain situations. But in modeling we're more interested in structured data in table shapes, so SQL databases are much more common.

SQL

Pandas

While pretty much everything you can do in SQL you can also do in Pandas, there are a few things I like leaving to SQL. Mainly: initial loading; joining multiple tables; and selecting columns from raw

data.

In contrast, though SQL does offer some basic column manipulation and grouping functionality, I seldom use it. Generally, Pandas is so powerful and flexible when it comes to manipulating columns and grouping data that I usually just load my data into it and do it there[1]. Also, though SQL has some syntax for updating and creating data tables, I also usually handle writing to databases inside Pandas.

But SQL is good for loading (not necessarily modifying) and joining your raw, initial, normalized tables. Its also OK for filtering, e.g. if you want to only load RB data or just want to look at a certain week.

There are a few benefits to limiting SQL to the basics like this. One is that SQL dialects and commands can change depending on the database you're using (Postgres, SQLite, MS SQL, etc), but most of the basics we'll cover will be the same.

Another benefit: when you stick to just basics, learning SQL is pretty easy and intuitive.

Creating Data

Remember, SQL and databases go hand-in-hand, so to be able to write SQL we need a database to practice on. Let's do it using SQLite, which is the simplest to get started.

The following code (1) creates an empty SQLite database, (2) loads the csv files that came with this book, and (3) puts them inside our database.

It relies on the sqlite3 library, which is included in Anaconda.

[1] One exception: SQL can be more memory efficient if your data is too large to load into Pandas, but that usually isn't a problem with the medium sized football datasets we'll be using.

```
import pandas as pd
from os import path
import sqlite3

# handle directories
DATA_DIR = '/Users/nathan/ltcwff-files/data'
DB_DIR = '/Users/nathan/ltcwff-files/data'

# create connection
conn = sqlite3.connect(path.join(DB_DIR, 'fantasy.sqlite'))

# load csv data
player_game = pd.read_csv(path.join(DATA_DIR, 'game_data_sample.csv'))
player = pd.read_csv(path.join(DATA_DIR, 'game_data_player_sample.csv'))
game = pd.read_csv(path.join(DATA_DIR, 'game_2017_sample.csv'))
team = pd.read_csv(path.join(DATA_DIR, 'teams.csv'))

# and write it to sql
player_game.to_sql('player_game', conn, index=False, if_exists='replace')
player.to_sql('player', conn, index=False, if_exists='replace')
game.to_sql('game', conn, index=False, if_exists='replace')
team.to_sql('team', conn, index=False, if_exists='replace')
```

You only have to do this once. Now we have a SQLite database with data in it.

Queries

SQL is written in **queries**, which are just instructions for getting data out of your database.

Every query has at least this:

```
SELECT <...>
FROM <...>
```

where in `SELECT` you specify the names of columns you want (`*` means all of them), and in `FROM` you're specifying the names of the tables.

So if you want all the data from your teams table, you'd do:

```
SELECT *
FROM teams
```

Though not required, a loose SQL convention is to put keywords like SELECT and FROM in uppercase, as opposed to particular column names or the name of our table (`teams`) which are lowercase.

Because the job of SQL is to get the data into Pandas so we can work with it, we'll be writing all our SQL within Pandas, i.e.:

```
In [1]:
df = pd.read_sql(
    """
    SELECT *
    FROM player
    """, conn)
```

The SQL query is written inside a Python string and passed to Pandas' `read_sql` method. I like writing my queries inside of multi-line strings, which start and end with three double quotation marks. In between you can put whatever you want and Python treats it like one giant string. In this, `read_sql` requires the string be valid SQL code.

The first argument to `read_sql` is this query string; the second is the connection to your SQLite database. You create this connection by passing the location of your database to the`sqlite3.connect` method.

Calling `read_sql` returns a DataFrame with the data you asked for.

```
In [2]: df.head()
Out[2]:
    player_id  season team pos player_name
0  00-0031359    2017  BUF  WR  K.Benjamin
1  00-0028116    2017   SF  WR  A.Robinson
2  00-0031390    2017   TB  RB      C.Sims
3  00-0031228    2017  ATL  WR   T.Gabriel
4  00-0031382    2017  MIA  WR    J.Landry
```

In the example file, you'll notice this is how we run all of the example queries in this chapter (in Python). And you should stick to that when running your own queries and as you run through the example file.

However, because the `pd.read_sql( ..., conn)` is the same for every query, I'm going to leave it (as well as the subsequent call to `head` showing what it returns) off for the rest of examples in this chapter. Hopefully that makes it easier to focus on the SQL code.

Just remember, to actually run these yourself, you have to pass these queries to sqlite via Python. To actually view what you get back, you need to call `head`.

What if we want to modify the query above to only return a few columns?

```
In [3]:
```

```
SELECT player_id, player_name AS name, pos
FROM player
```

```
Out [3]:
```

```
    player_id        name pos
0  00-0031359  K.Benjamin  WR
1  00-0028116  A.Robinson  WR
2  00-0031390      C.Sims  RB
3  00-0031228   T.Gabriel  WR
4  00-0031382    J.Landry  WR
```

You just list the columns you want and separate them by commas. Notice the `player_name AS name` part of the `SELECT` statement. Though column is stored in the database as `player_name`, we're renaming it to `name` on the fly, which can be useful.

Filtering

What if we want to *filter* our rows, say — only get back Miami players? We need to add another clause, a `WHERE`:

`In [4]:`

```
SELECT player_id, player_name AS name, pos
FROM player
WHERE team = 'MIA'
```

`Out [4]:`

```
    player_id      name pos
0  00-0031382  J.Landry  WR
1  00-0033118   K.Drake  RB
2  00-0031609   A.Derby  TE
3  00-0031547  D.Parker  WR
```

A few things to notice here. First, note the single equals sign. Unlike Python, where = is assignment and == is testing for equality, in SQL just the one = tests for equality. Even though we're writing this query inside a Python string, we still have to follow SQL's rules.

Also note the single quotes around 'MIA'. Double quotes won't work.

Finally, notice we're filtering on `team` (i.e. we're choosing which rows to return depending on the value they have for `team`), even though it's not in our `SELECT` statement. That's fine. We could include it if we wanted (in which case we'd have a column `team` with `'MIA'` for every row), but it's not necessary.

We can use logic operators like `OR` and `AND` in our `WHERE` clause too.

`In [5]:`

```
SELECT player_id, player_name AS name, pos
FROM player
WHERE team = 'MIA' AND pos == 'WR'
```

Out [5]:

```
    player_id      name pos
0  00-0031382  J.Landry  WR
1  00-0031547  D.Parker  WR
```

In [6]:

```
SELECT player_id, player_name AS name, pos, team
FROM player
WHERE team = 'MIA' OR team == 'NE'
```

Out [6]:

```
    player_id      name pos team
0  00-0031382  J.Landry  WR  MIA
1  00-0033118   K.Drake  RB  MIA
2  00-0031609   A.Derby  TE  MIA
3  00-0028087   D.Lewis  RB   NE
4  00-0019596   T.Brady  QB   NE
```

That last query could also be rewritten as:

```
SELECT player_id, player_name AS name, pos, team
FROM player
WHERE team IN ('MIA', 'NE')
```

SQL also allows negation:

In [7]:

```
SELECT player_id, player_name AS name, pos, team
FROM player
WHERE team NOT IN ('MIA', 'NE')
```

Out [7]:

```
    player_id        name pos team
0  00-0031359  K.Benjamin  WR  BUF
1  00-0028116  A.Robinson  WR   SF
2  00-0031390      C.Sims  RB   TB
3  00-0031228   T.Gabriel  WR  ATL
4  00-0032209    T.Yeldon  RB  JAX
```

Joining, or Selecting From Multiple Tables

SQL is also good at combining multiple tables. Say we want to see a list of players (in the `player` table) and the division they play in (in the `team` table).

We might try adding a new table to our `FROM` clause like this:

`In [8]:`

```
SELECT
    player.player_name as name,
    player.pos,
    player.team,
    team.conference,
    team.division
FROM player, team
```

Note we now pick out the columns we want from each table using the `table.column_name` syntax.

But there's something weird going on here. Look at the first 10 rows of the table:

`Out [8]:`

	name	pos	team	conference	division
0	K.Benjamin	WR	BUF	NFC	West
1	K.Benjamin	WR	BUF	NFC	South
2	K.Benjamin	WR	BUF	AFC	North
3	K.Benjamin	WR	BUF	AFC	East
4	K.Benjamin	WR	BUF	NFC	South
5	K.Benjamin	WR	BUF	NFC	North
6	K.Benjamin	WR	BUF	AFC	North
7	K.Benjamin	WR	BUF	AFC	North
8	K.Benjamin	WR	BUF	NFC	East
9	K.Benjamin	WR	BUF	AFC	West

It's all Kelvin Benjamin, and he's in every division.

The problem is we haven't told SQL how the `player` and `team` tables are related.

When you don't include that info, SQL doesn't try to figure it out or complain and give you an error. Instead it returns a crossjoin, i.e. EVERY row in the `player` table gets matched up with EVERY row in the `team` table.

In this case we have two tables: (1) player, and (2) team. So we have our first row (K.Benjamin, WR, BUF) matched up with the first division in the team table (NFC West); then the second (NFC South), and so on.

To make it even clearer, let's add in the `team` column from the team table too.

`In [9]:`

```
SELECT
    player.player_name as name,
    player.pos,
    player.team as player_team,
    team.team as team_team,
    team.conference,
    team.division
FROM player, team
```

Out [9]:

	name	pos	player_team	team_team	conference	division
0	K.Benjamin	WR	BUF	ARI	NFC	West
1	K.Benjamin	WR	BUF	ATL	NFC	South
2	K.Benjamin	WR	BUF	BAL	AFC	North
3	K.Benjamin	WR	BUF	BUF	AFC	East
4	K.Benjamin	WR	BUF	CAR	NFC	South
5	K.Benjamin	WR	BUF	CHI	NFC	North
6	K.Benjamin	WR	BUF	CIN	AFC	North
7	K.Benjamin	WR	BUF	CLE	AFC	North
8	K.Benjamin	WR	BUF	DAL	NFC	East
9	K.Benjamin	WR	BUF	DEN	AFC	West

This makes it clear it's a crossjoin — every line for Kelvin Benjamin (and also every other player once Benjamin is done) is getting linked up with every team in the team table.

Also, since we have a 100 row player table and 32 row team table, that means we should get back 100*32 = 3200 rows and sure enough:

```
In [10]: df.shape
Out[10]: (3200, 6)
```

What if we added a third table, say a `conference` table with two rows: one for NFC, one for AFC. In that case, each of these 3,200 rows in the table above gets matched yet again with *each* of the two rows in the conference table. In other words, our table is 100*32*2 = 6400 rows.

This is almost never what we want[2] (in fact an inadvertent crossjoin is something to watch out for if a query is taking way longer to run than it should) but it's useful to think of the FROM part of a multi-table SQL query as doing cross joins initially.

But we're not interested in a full crossjoin and getting back a row where Kelvin Benjamin plays in the NFC North, so we have specify a `WHERE` clause to filter and keep only the rows that make sense.

In [11]:

[2] Can you think of a time when it would be what you want? I can think of one: if you had a table of teams, and another of weeks 1-17 and wanted to generate a schedule so that every team had a line for every week. That's it though.

```
SELECT
    player.player_name as name,
    player.pos,
    player.team,
    team.conference,
    team.division
FROM player, team
WHERE player.team = team.team
```

Out [11]:

```
          name pos team conference division
0  K.Benjamin  WR  BUF        AFC     East
1  A.Robinson  WR   SF        NFC     West
2      C.Sims  RB   TB        NFC    South
3   T.Gabriel  WR  ATL        NFC    South
4    J.Landry  WR  MIA        AFC     East
```

Let's walk through it:

First, SQL is doing the full cross join (with 3200 rows).

Then we have a `WHERE`, so then we're saying give us *only* the rows where the column `team` from the `players` table matches the column `team` from the `team` table. We go from having 32 separate rows for Kelvin Benjamin, to only the one row — where his team in the `player` table (`BUF`) equals `BUF` in the team table.

That gives us a table of 100 rows — the same number of players we originally started with — and includes the conference and division info for each.

Again, adding in the `team` column from table team makes it more clear:

In [12]:

```
SELECT
    player.player_name as name,
    player.pos,
    player.team as player_team,
    team.team as team_team,
    team.conference,
    team.division
FROM player, team
WHERE player.team = team.team
```

Out [12]:

```
          name pos player_team team_team conference division
0  K.Benjamin  WR         BUF       BUF        AFC     East
1  A.Robinson  WR          SF        SF        NFC     West
2      C.Sims  RB          TB        TB        NFC    South
3   T.Gabriel  WR         ATL       ATL        NFC    South
4    J.Landry  WR         MIA       MIA        AFC     East
```

I first learned about this crossjoin-then `WHERE` framework of conceptualizing SQL queries from the book, The Essence of SQL by David Rozenshtein, which is a great book, but out of print and as of this writing going for $125 (as a 119 page paperback) on Amazon. It covers more than just crossjoin-WHERE, but we can use Pandas for most of the other stuff. If you want you can think about this section as The Essence of The Essence of SQL.

What if we want to add a third table? We just need to add it to `FROM` and refine our `WHERE` clause.

In [13]:

```
SELECT
    player.player_name as name,
    player.pos,
    team.team,
    team.conference,
    team.division,
    player_game.*
FROM player, team, player_game
WHERE
    player.team = team.team AND
    player_game.player_id = player.player_id
```

Out [13]:

```
          name  pos  team conference division      gameid   player_id ...
0  K.Benjamin   WR   BUF        AFC     East  2017091011  00-0031359 ...
1  K.Benjamin   WR   BUF        AFC     East  2017091701  00-0031359 ...
2  K.Benjamin   WR   BUF        AFC     East  2017092402  00-0031359 ...
3  K.Benjamin   WR   BUF        AFC     East  2017100107  00-0031359 ...
4  K.Benjamin   WR   BUF        AFC     East  2017100802  00-0031359 ...
```

This is doing the same as above (player info + each players team) but also combining it with info from the `player_games`. Note, that if we had left off our `WHERE` clause, SQL would have done a full, three table player⋆team⋆player_game crossjoin. Kelvin Benjamin would be matched with each division, then each of *these* rows would be matched with every row from the player-game table. That'd give us a 50⋆32⋆716=1,145,600 row result.

Also note the `player_games.*` syntax, that gives us all the columns from that table.

Sometimes table names can get long and unwieldy — especially when working with multiple tables

— so we could also write above as:

```
SELECT
    p.player_name as name,
    p.pos,
    t.team,
    t.conference,
    t.division,
    pg.*
FROM player AS p, team AS t, player_game AS pg
WHERE
    p.team = t.team AND
    pg.player_id = p.player_id
```

We just specify the full names once (in `FROM`), then add an alias with `AS`. Then in the `SELECT` and `WHERE` clauses we can use the alias instead.

Combing Joining and Other Filters

We can also add in other filters, e.g. maybe we want this same query but only the RBs:

```
SELECT
    p.player_name as name,
    p.pos,
    t.team,
    t.conference,
    t.division,
    pg.*
FROM player AS p, team AS t, player_game AS pg
WHERE
    p.team = t.team AND
    pg.player_id = p.player_id AND
    p.pos == 'RB'
```

The basics of `SELECT`, `FROM` and `WHERE` plus the crossjoin-then filter way of conceptualizing joins + the fact you're leaving the other parts of SQL to Pandas should make learning SQL straightforward.

But there are a few additional minor features that can be useful and sometimes come up.

Misc SQL

LIMIT/TOP

Sometimes want to make sure a query works and see what columns you get back before you just run the whole thing.

In that case you can do:

```
SELECT *
FROM player
LIMIT 5
```

Which will return the first five rows. Annoyingly, the syntax for this is something that changes depending on the database you're using, for Microsoft SQL Server it'd be:

```
SELECT TOP 5 *
FROM player
```

DISTINCT

Including `DISTINCT` right after `SELECT` drops duplicate observations.

For example, maybe we want to see a list of how many different calendar days NFL games were on.

`In [14]:`

```
SELECT DISTINCT season, week, date
FROM game
```

`Out [14]`

```
   season  week        date
0    2017     1  2017-09-07
1    2017     1  2017-09-10
2    2017     1  2017-09-11
3    2017     2  2017-09-14
4    2017     2  2017-09-17
```

UNION

`UNION` lets you stick data on top of each other to form on table. It's similar to `concat` in Pandas.

Above and below `UNION` are two separate queries, and both queries *have* to return the same columns in their `SELECT` statements.

So maybe I want to do an analysis over the past two years and I'm inheriting data is in separate tables. I might do:

```
SELECT *
FROM player_data_2018
UNION
SELECT *
FROM player_data_2019
```

Subqueries

Previously, we've seen how we can do `SELECT ... FROM table_name AS` abbreviation. In a subquery, we replace `table_name` with another, inner `SELECT ... FROM` query and wrap it in parenthesis.

LEFT, RIGHT, OUTER JOINS

You may have noticed our mental crossjoin-then-`WHERE` framework can only handle inner joins. That is, we'll only keep observations that are in *both* tables. But this isn't always what we want.

For example, maybe we want 16 rows (one for each game) for every player, regardless of whether they played. In that case we'd have to do a **left join**, where our left table is a full 16 rows for every player, and our right table is games they actually played. The syntax is:

```
SELECT *
FROM <left_table>
LEFT JOIN <right_table>
ON <left_table>.<common_column> = <right_table>.<common_column>
```

SQL Example — LEFT JOIN, UNION, Subqueries

I find left and right joins in SQL less intuitive than the crossjoin then filter with `WHERE` framework, and do most of my non-inner joins in Pandas. But writing this full, one row-for-every-game and player query using the tables we have does require some of the previous concepts (unions and subqueries), so it might be useful to go through this as a final example.

Feel free to skip this if you don't envision yourself using SQL that much and are satisfied doing this in Pandas.

This query gives us rushing and receiving yards and TDs for every player and game in our database, whether the player actually suited up or not. We'll walk through it below:

```
1  SELECT a.*, b.rec_yards, b.rush_yards, b.rec_tds, b.rush_tds
2  FROM
3      (SELECT
4          game.season, week, gameid, home as team, player_id,
5          player_name
6      FROM game, player
7      WHERE game.home = player.team
8      UNION
9      SELECT
10         game.season, week, gameid, away as team, player_id,
11         player_name
12     FROM game, player
13     WHERE game.away = player.team) AS a
14 LEFT JOIN player_game AS b
15 ON a.gameid = b.gameid AND a.player_id = b.player_id
```

Let's go through it. First, we need a full set of 16 rows for every player. We do that in a subquery (lines 3-13) and call the resulting table `a`.

This subquery involves a `UNION`, let's look at the top part (lines 3-7).

```
SELECT game.season, week, gameid, home as team, player_id,
       player_name
FROM game, player
WHERE game.home = player.team
```

Remember, the first thing SQL is doing when we query FROM game and player is a full crossjoin, i.e. we get back a line for *every* player in *every* game. So, after the game, player crossjoin here not only is there a line for Aaron Rodgers, week 1, Green Bay vs Seattle, but there's a line for Aaron Rogers, week 1, Kansas City at New England too.

This is way more than we need. In the case of Aaron Rodgers, we want to filter the rows to ones where one of the teams is Green Bay. We do that in our `WHERE` clause. This is all review from above.

The problem is our `game` table is by week, not team and week. If it were the latter, GB-SEA week 1 would have *two* lines, one for Green Bay, one for Seattle. Instead it's just the one line, with Green Bay in the `home` field, Seattle in `away`.

What this means is, to match up Aaron Rodgers to only the Green Bay games, we're going to have to run this part of the query twice: once when Green Bay is home, another time when Green Bay is away, then stick them together with a `UNION` clause

That's what we're doing here:

```
SELECT game.season, week, gameid, home as team, player_id, player_name
FROM game, player
WHERE game.home = player.team
UNION
SELECT game.season, week, gameid, away as team, player_id, player_name
FROM game, player
WHERE game.away = player.team
```

Above the `UNION` gives us a line for every player's home games (so 8 per player), below a line for every away game. Stick them on top of each other and we have what we want.

It's all in a subquery, which we alias as `a`. So once you understand that, you can ignore it and mentally replace everything in parenthesis with `full_player_by_week_table` if you want.

In fact, let's do that, giving us:

```
    SELECT a.*, b.rec_yards, b.rush_yards, b.rec_tds, b.rush_tds
    FROM
        full_player_by_week_table AS a
    LEFT JOIN
        player_game AS b ON
            a.gameid = b.gameid AND
            a.player_id = b.player_id
```

From there it's the standard left join syntax: `LEFT JOIN table_name ON ...`

And the result

```
In [15]: df.loc[df['player_name'] == 'M.Sanu']
Out[15]:
     season  week player_name  rec_yards  rush_yards  rec_tds  rush_tds
7      2017     1      M.Sanu       47.0         0.0      0.0       0.0
89     2017     2      M.Sanu       85.0         0.0      0.0       0.0
110    2017     3      M.Sanu       27.0         0.0      1.0       0.0
152    2017     4      M.Sanu        3.0         4.0      0.0       0.0
243    2017     6      M.Sanu        NaN         NaN      NaN       NaN
324    2017     7      M.Sanu       65.0         0.0      0.0       0.0
350    2017     8      M.Sanu       74.0         0.0      1.0       0.0
373    2017     9      M.Sanu       23.0         0.0      1.0       0.0
443    2017    10      M.Sanu       29.0         0.0      0.0       0.0
495    2017    11      M.Sanu       34.0         3.0      1.0       0.0
503    2017    12      M.Sanu       64.0         0.0      0.0       0.0
552    2017    13      M.Sanu       54.0         0.0      0.0       0.0
600    2017    14      M.Sanu       83.0         0.0      1.0       0.0
721    2017    16      M.Sanu       31.0         0.0      0.0       0.0
747    2017    17      M.Sanu       83.0         0.0      1.0       0.0
```

Mohamed Sanu missed game 6 in 2017, yet we still have a row for him. So this is returning what we'd expect.

End of Chapter Exercises

4.1

a) Using the same sqlite database we were working with in this chapter, use SQL to make a DataFrame that's only AFC QBS with the following columns:

`season`,week', `player_name`, `team`, `attempts`, `completions`, `pass_yards`, `pass_tds`, `interceptions`

Since we know were only dealing with QBs, rename `pass_yards` and `pass_tds` to just `yards` and `tds`.

b) Now pretend your data is normalized and `player_game` doesn't have the `team`, `pos` or `player_name` columns. How do you have to modify your query to get the same result as in (a)? Use abbreviations too (`t` for the team table, `pg` for `player_game`, etc).

4.2

Using the same sqlite database, make a DataFrame that's the `game` table, with two new columns: `'home_mascot'` and `'away_mascot'`.

Hint: you can include the same table multiple separate times in your `FROM` statement as long as you use a different alias for it each time.

5. Web Scraping and APIs

Introduction to Web Scraping and APIs

This chapter is all about the first section of the high level pipeline: collecting data. Sometimes you'll find structured, ready-to-consume tabular data directly available in a spreadsheet or a database, but often you won't, or — even if you do — you'll want to supplement it with other data.

You'll learn here how to build programs that can grab this data for you. In general, there are two situations where these automatic, data-collecting programs are particularly useful.

First, if you're dealing with some dynamic (changing over time) data that you want to take snapshots of at regular intervals. For example, a program that gets the current weather at every NFL stadium that you run every week.

The other case would be when you want to grab a lot of similarly structured data. Scrapers scale really well. You want annual fantasy point leaders by position for the past 10 years? Write a Python that function that's flexible enough to get data given any arbitrary position and year. Then run it 60 times (10 years X six positions — QB, RB, WR, TE, K, DST) and you'll have the data. Trying to manually copy and paste data from that many times from a website would be very tedious and error prone.

These two use cases — collecting snapshots of data over time and grabbing a bunch of similarly structured data — aren't the only time you'll ever want to get data from the internet. There might be situations you find a single one off table online that you'll just need one time. In that case you might be better off just copying and pasting it into a csv instead of taking the time to write code to get it automatically.

We'll cover two ways to get data in this section: web scraping and getting data from an API.

Web Scraping

Most websites — including websites with data — are designed for human eyeballs. A web scraper is a program built to interact with and collect data from these websites. Once they're up and running, you usually can run and collect data without actually having to visit the site in person.

HTML

Building a webscraper involves understanding some basic HTML + CSS, which are two of the main building blocks used to build websites. We'll focus on the minimum required for webscraping. So while you won't necessarily be able to build your own website after this, it will make getting data from websites a lot easier.

HTML is a *markup* language, which means it includes both the content you see on the screen (i.e. the text) along with built in instructions (the markup) for how the browser should show it.

These instructions come in **tags**, which are wrapped in arrow brackets (<>) and look like this:

```
<p>
<b>
<div>
```

Most tags come in pairs, with the start one like `<p>` and the end like this `</p>`. We say any text in between is *wrapped* in the tags. For example, the p tag stands for paragraph and so:

```
<p>This text is a paragraph.</p>
```

Tags themselves aren't visible to regular users, though the text they wrap is. You can see them by right clicking on website and selecting 'view source'.

Tags can be nested. The i tag stands for italics, and so:

```
<p><i>This text is an italic paragraph.</i></p>
```

Tags can also have one or more *attributes*, which are just optional data and are also invisible to the user. Two common attributes are `id` and **`class`**:

```
<p id="intro">This is my intro paragraph.</p>
<p id="2" class="main-body">This is is my second paragraph.</p>
```

The id's and classes are there so web designers can specify — separately in what's called a CSS file — more rules about how things should look. So maybe the intro paragraph should look one way, or paragraphs with the class "main-body" should look another way, etc.

As scrapers, we don't care how things look, and we don't care about CSS itself. But these tags, ids, and classes are a good way to tell our program what we want it to get, so it's good to be aware of what it is.

Common HTML tags include:

p paragraph

div this doesn't really do anything directly, but they're a way for web designer's to divide up their HTML however they want so they can assign classes and ids to particular sections

table tag that specifies the start of a table

th header in a table

tr denotes table row

td table data

a link, always includes the attribute href, which specifies where the browser should go when you click on it

HTML Tables

As data analysts, we're often interested in tables, so it's worth exploring how HTML tables work in more depth.

The `table` tag is a good example of how HTML tags can be nested. Everything in the table (all the data, the headers, etc) is between the `<table>` `</table>` tags.

Inside those, the tags `<tr>` (for table row), `<td>` (table data), and `<th>` (table header) let you specify more structure.

The first pair of tags in any table is `<tr>` and `</tr>`, which denotes when a row starts and ends. Everything between them is one row.

Inside a row, we can specify the start and end of particular columns with `<td>` and `</td>` or — if we're inside the header (column name) row — `<th>` and `</th>`. Usually you'll see header tags in the first row.

So if we had a table giving weekly fantasy points, it might look something like this:

```
<table>
  <tr>
   <th>Name</th>
   <th>Pos</th>
   <th>Week</th>
   <th>Pts</th>
  </tr>
  <tr>
   <td>Todd Gurley</td>
   <td>RB</td>
   <td>1</td>
   <td>22.6</td>
  </tr>
  <tr>
   <td>Christian McCaffrey</td>
   <td>RB</td>
   <td>1</td>
   <td>14.8</td>
  </tr>
</table>
```

BeautifulSoup

The library BeautifulSoup (abbreviated BS4) is the Python standard for working with HTML data like this. It lets you manipulate, search and work with HTML and put it into normal python data structures (like lists and dicts), which we can then put into Pandas.

The key Python type in BeautifulSoup is called *tag*. What DataFrame is to Pandas, a tag is to BeautifulSoup.

Like lists and dicts, BeautifulSoup tags are containers (they hold things). One thing tags *can* hold is other tags, though they don't have to.

If we're working with an HTML table, we could have a BS4 tag that represents the whole table. We could also have a tag that represents just the one `<td>Todd Gurley</td>` element. They're both tags. In fact, the entire web page (all the html) is just one zoomed out html tag.

Let's mentally divide tags into types[1].

The first type is a tag with just text inside it, not another tag. We'll call this a "simple" tag. Examples of HTML elements that would be simple tags:

```
<td>Todd Gurley</td>
<td>1</td>
<th>Name</th>
```

[1] BeautifulSoup doesn't really distinguish between these, but I think it makes things easier to understand.

This are three separate simple tags, and doing `.name` on them would give td, td, th respectively. But they don't hold any more tags inside of it, just the text.

On simple tags, the key method is `string`. That gives you access to the data inside of it. So running `string` on these three tags would give 'Todd Gurley', '1', and 'Name' respectively. These are BeautifulSoup strings, which carry around a bunch of extra data, so it's always good to do convert them to regular Python strings with `str(mytag.string)`.

The second type of tags are tags that contain other tags. Call these "nested" tags. An example would be the `tr` or `table` tags above.

The most important method for nested tags is `find_all(tag_name)`, where `tag_name` is the name of an HTML tag like 'tr', 'p' or 'td'. The method searches your nested tag for `tag_name` tags and returns them all in a list.

Our div tag above is a nested tag.

So, in the example above `mytable.find_all('tr')` would give back a list like [tr, tr, tr], where each tr is a BeautifulSoup tr tag.

Since each of these tr has a bunch (3) of td tags, they themselves are nested tags. That means we can call `find_all()` on them to get back a list of the td tags in that row. These td tags are simple tags, so we can call `.string` on each of them to (finally) get the data out.

The constant, nested inside of nested tags aspect of everything can make it hard to wrap your mind around, but it's not too bad with code.

Fantasy Football Calculator ADP - Web Scraping Example

In this section, we'll build a scraper to get all the ADP data off fantasyfootballcalculator.com and put it in a DataFrame. The code for this is in `05_01_scraping.py`.

We can import what we need from it like this (see the file bs4_demo.py):

```
In [1]: from bs4 import BeautifulSoup as Soup

In [2]: import requests

In [3]: from pandas import DataFrame
```

Besides BeautifulSoup, we're also importing the requests library to visit the page and store the raw HTML we get back. We're also going to want to put our final result in a DataFrame, so we've imported that too.

The first step using using s requests visit the page we want to scrape and store the raw HTML we get back.

```
In [3]: ffc_response = requests.get('https://fantasyfootballcalculator.com
    /adp/ppr/12-team/all/2017')
```

Let's look at (part of) this HTML.

```
In [4] print(ffc_response.text)
Out[4]:

      <tr class='PK'>
        <td align='right'>178</td>
        <td align='right'>14.07</td>
        <td class="adp-player-name"><a style="color: rgb(30, 57, 72); font
           -weight: 300;" href="/players/mason-crosby">Mason Crosby</a></
           td>
        <td>PK</td>
        <td>GB</td>


          <td class="d-none d-sm-table-cell" align='right'>162.6</td>
          <td class="d-none d-sm-table-cell" align='right'>5.9</td>
          <td class="d-none d-sm-table-cell" align='right'>13.06</td>
          <td class="d-none d-sm-table-cell" align='right'>15.10</td>
          <td class="d-none d-sm-table-cell" align='right'>44</td>
          <td align='center'>

        </td>
        </tr>
```

Calling text on `ffc_response` turns it into a string. This is just a small snippet of the HTML you see back — there are almost 250k lines of HTML here — but you get the picture.

Now let's *parse* it, i.e. turn it into BeautifulSoup data.

```
In [5]: adp_soup = Soup(ffc_response.text)
```

Remember, this top level `adp_soup` object is a giant nested tag, which means we can run `find_all()` on it.

We can never be 100% sure when dealing with sites that we didn't create, but looking at the fantasy-footballcalculator.com, it's probably safe to assume the data we want is on an HTML table.

```
In [6]: tables = adp_soup.find_all('table')
```

Remember `find_all()` always returns a list, even if there's just one table. Let's see how many tables we got back.

```
In [7]: len(tables)
Out[7]: 1
```

It's just the one here, which is easy, but it's not at all uncommon for sites to have multiple tables, in which case we'd have to pick out the one we wanted from the list. Technically we still have to pick this one out, but since there's just one it's easy.

```
In [8]: adp_table = tables[0]
```

Looking at it in the REPL, we can see it has the same `<tr>`, `<th>` and `<td>` structure we talked about above, though — being a real website — the tags have other attributes (class, align, style etc).

`adp_table` is still a nested tag, so let's run another `find_all()`.

```
In [9]: rows = adp_table.find_all('tr')
```

Giving us a list of all the tr tags inside our table. Let's look at the first one.

```
In [10]: rows[0]
Out[10]:
<tr>
<th>#</th>
<th>Pick</th>
<th>Name</th>
<th>Pos</th>
<th>Team</th>
<th class="d-none d-sm-table-cell">Overall</th>
<th class="d-none d-sm-table-cell">Std.<br/>Dev</th>
<th class="d-none d-sm-table-cell">High</th>
<th class="d-none d-sm-table-cell">Low</th>
<th class="d-none d-sm-table-cell">Times<br/>Drafted</th>
<th></th>
</tr>
```

It's the header row, good. That'll be useful later for knowing what columns are what.

Now how about some data.

```
In [11]: first_data_row = rows[1]

In [12]: first_data_row
Out[12]:
<tr class="RB">
<td align="right">1</td>
<td align="right">1.01</td>
<td class="adp-player-name"><a href="/players/david-johnson" style="color:
    rgb(30, 57, 72); font-weight: 300;">David Johnson</a></td>
<td>RB</td>
<td>ARI</td>
<td align="right" class="d-none d-sm-table-cell">1.3</td>
<td align="right" class="d-none d-sm-table-cell">0.6</td>
<td align="right" class="d-none d-sm-table-cell">1.01</td>
<td align="right" class="d-none d-sm-table-cell">1.04</td>
<td align="right" class="d-none d-sm-table-cell">310</td>
<td align="center">
</td>
</tr>
```

It's the first, lowest-ADP pick — David Johnson for the of summer 2017 — nice. Note this is *still* a nested tag, so we will be using `find_all()`, but there are a bunch of td columns in it so end is in site.

```
In [13]: first_data_row.find_all('td')
Out[13]:
[<td align="right">1</td>,
 <td align="right">1.01</td>,
 <td class="adp-player-name"><a href="/players/david-johnson" style="color
    : rgb(30, 57, 72); font-weight: 300;">David Johnson</a></td>,
 <td>RB</td>,
 <td>ARI</td>,
 <td align="right" class="d-none d-sm-table-cell">1.3</td>,
 <td align="right" class="d-none d-sm-table-cell">0.6</td>,
 <td align="right" class="d-none d-sm-table-cell">1.01</td>,
 <td align="right" class="d-none d-sm-table-cell">1.04</td>,
 <td align="right" class="d-none d-sm-table-cell">310</td>,
 <td align="center">
 </td>]
```

This returns a list of simple `td` tags, finally. Now we can call str(.string) on them to get the data out.

```
In [14]: [str(x.string) for x in first_data_row.find_all('td')]
Out[14]:
['1',
 '1.01',
 'David Johnson',
 'RB',
 'ARI',
 '1.3',
 '0.6',
 '1.01',
 '1.04',
 '310',
 '\n']
```

Note the list comprehensions. Scraping is an area comprehensions really become valuable.

Now that that's done, we have to do it to every row. Having to do something a bunch of times should make you think about putting it in a function, so let's make a function that'll parse some row (a BeautifulSoup `tr` tag).

```
def parse_row(row):
    """
    Take in a tr tag and get the data out of it in the form of a list of
    strings.
    """
    return [str(x.string) for x in row.find_all('td')]
```

Now we have to apply `parse_row` to every row and put the results in a DataFrame. To make a new DataFrame we do `df = DataFrame(data)`, where data can be various data structures.

Looking at the pandas documentation, one type of data we can use is a "structured or record array". In other words, a list of lists (rows) — as long as they're all the same shape — should work.

```
In [15]: list_of_parsed_rows = [parse_row(row) for row in rows[1:]]
```

Remember the first row (`row[0]` is the header, so `rows[1:]` will give us our data, 1 to the end). Then we have to put it in a DataFrame:

```
In [16]: df = DataFrame(list_of_parsed_rows)

In [17]: df.head()
Out[17]:
   0     1                2   3    4    5  ...     8    9  10
0  1  1.01    David Johnson  RB  ARI  1.3  ...  1.04  310  \n
1  2  1.02      LeVeon Bell  RB  PIT  2.3  ...  1.06  303  \n
2  3  1.04    Antonio Brown  WR  PIT  3.7  ...  1.07  338  \n
3  4  1.06      Julio Jones  WR  ATL  5.7  ...  2.03  131  \n
4  5  1.06  Ezekiel Elliott  RB  DAL  6.2  ...  2.05  180  \n
```

Great, it just doesn't have column names. Let's fix that. We could parse them, but it's easiest to just name them what we want.

```
In [18]: df.columns = ['ovr', 'pick', 'name', 'pos', 'team', 'adp',
                       'std_dev', 'high', 'low', 'drafted', 'graph']
```

We're almost there, our one remaining issue is that all of the data is in string form at the moment, just like numbers stored as text in Excel. This is because we explicitly told Python to do this in our parse_row function, but via converting types, it's an easy fix.

```
In [19]: float_cols =['adp', 'std_dev']

In [20]: int_cols =['ovr', 'drafted']

In [21]: df[float_cols] = df[float_cols].astype(float)

In [22]: df[int_cols] = df[int_cols].astype(int)
```

It's debatable whether pick, high, low should be numbers or string. Technically they're numbers, but I kept them strings because the decimal in the picks (e.g. 1.12) doesn't mean what it normally means and wouldn't behave the way we'd expect if we tried to do math on it.

Also let's get rid of the graph column, which we can see on the website is a UI checkbox on fantasy-footballcalculator and not real data.

```
In [23]: df.drop('graph', axis=1, inplace=True)
```

And we're done:

```
In [24]: df.head()
Out[24]:
   ovr  pick             name pos team  adp  std_dev  high   low  drafted
0    1  1.01    David Johnson  RB  ARI  1.3      0.6  1.01  1.04      310
1    2  1.02      LeVeon Bell  RB  PIT  2.3      0.8  1.01  1.06      303
2    3  1.04    Antonio Brown  WR  PIT  3.7      1.0  1.01  1.07      338
3    4  1.06      Julio Jones  WR  ATL  5.7      3.2  1.01  2.03      131
4    5  1.06  Ezekiel Elliott  RB  DAL  6.2      2.8  1.01  2.05      180
```

There we go, we've built our first webscraper!

Exercises

5.1.1

Put everything we did in the Fantasy Football ADP Calculator example inside a function `scrape_ffc` that takes in `scoring` (a string that's one of: 'ppr', 'half', 'std'), `nteams` (number of teams) and `year` and returns a DataFrame with the scraped data.

Your function should work on all types of scoring systems leagues, 8, 10, 12 or 14 team leagues, between 2010-2019.

Caveats:

- Fantasy Football Calculator only has half-ppr data going back to 2018, so you only need to be able to get std and ppr before that.
- All of Fantasy Football Calculators archived data is for 12 teams. URLs with other numbers of teams will work (return data), but if you look closely the data is always the same.

5.1.2

On the Fantasy Football Calculator page we're scraping, you'll notice that each player's name is a clickable link. Modify `scrape_ffc` to store that as well, adding a column named 'link' for it in your DataFrame.

Hint: bs4 tags include the `get` method, which takes the name of an html attribute and returns the value.

5.1.3

Write a function `ffc_player_info` takes one of the urls we stored in 4.1, scrapes it, and returns the player's team, height, weight, birthday, and draft info (team and pick) as a dict.

APIs

Above, we learned how to scrape a website — i.e. use BeautifulSoup to interact with HTML and get data into a format Pandas could deal with. While HTML basically tells the browser what to render onscreen for the benefit of human eyeballs — and therefore necessarily contains the contents itself, which we grab using our program — an API works different.

Two Types of APIs

Code specifications

Before we get into this it's important to note that, in practice, people mean at least two things by the term API. API stands for Application Programming Interface, and really the two ways people talk about them depend on what you mean by "application". For example, the application might be the Pandas library. Then the API is just the exact rules, the functions, what they take and return, etc of Pandas. So part of the Pandas API is the `pd.merge()` function. `merge` specifies that it requires two DataFrames, has optional arguments (with defaults) for how, on, left_on, right_on, left_index, right_index, sort, suffixes, copy, and indicator and returns a DataFrame.

Note that API used in this context isn't the same thing as just the `pd.merge` documentation. Ideally, an API is accurately documented (and Pandas is), but it's not required. Nor is an API the pd.merge function itself. Instead it's how the function interacts with the outside world and the programmer, basically what it takes and returns.

Being familiar with the Pandas API basically means just knowing how to use Pandas.

Web APIs

The other, probably more common way the term API is used is as a web API. In that case the website is the "application" and we interact with it via it's URL. Everyone is familiar with the most basic urls, e.g. www.fantasymath.com, but urls can have extra stuff too, e.g. api.fantasymath.com/v2/players-wdis/?wdis=aaron-rodgers&wdis=drew-brees

At a very basic level: a web API lets you specifically manipulate the URL (e.g. maybe you want to put `kurt-cousins` in instead of `aaron-rodgers`) and get data back. It does this via the same mechanisms (HTTP) that regular, non-API websites when you go to some given URL.

It's important to understand web APIs are specifically designed, built and maintained by website owners with the purpose of providing data in an easy to digest, predictable way. You can't just tack on "api" to the front of any URL, play around with parameters, and start getting back useful data. Any API you'll be using should give you some instructions on what to do and what everything means.

Not every API is documented and meant for anybody to use. More commonly, they're built for a website's own, internal purposes. The Fantasy Math API, for example, is only called by the www.fantasymath.com site. You can go to fantasymath.com, pick two players from the dropdown (Aaron Rodgers and Drew Brees), and when you click submit the site (behind the scenes) accesses the API via that URL behind the scenes, then collects and displays the results. Decoupling the website

(front-end) from the part that does the calculations (the back-end) makes things simpler and easier to program.

But some web APIs are public, which means anyone can access them to get back some data. A good example is the Fantasy Football Calculator API.

Fantasy Football Calculator ADP - API Example

Let's get the same data we scraped up above via Fantasy Football Calculator's public API. This example is in `05_02_api.py`. Just a heads up — it's going to be a lot shorter than the scraping example. We'll run through it quick and then do a more in depth explanation after.

First we need to import requests and DataFrame.

```
In [1]: import requests

In [2]: from pandas import DataFrame
```

Then we need to call the URL. We can see via Fantasy Football Calculator's documentation's that for 12 team, PPR scoring we want to call.

```
In [3]: fc_url = 'https://fantasyfootballcalculator.com/api/v1/adp/ppr?
   teams=12&year=2017'
```

Then we access it with the `get` method from requests. The data it returns gets assigned to 'resp.

```
In [4]: resp = requests.get(fc_url)
```

Finally, we call the `json` method on `resp`, pass that to Pandas and we're good to go.

```
In [5]: df = DataFrame(resp.json()['players'])

In [6]: df.head()
Out[6]:
              name  adp  times_drafted team  ...  position bye  stdev
0    David Johnson  1.3            310  ARI  ...        RB  12    0.6
1      LeVeon Bell  2.3            303  PIT  ...        RB   7    0.8
2    Antonio Brown  3.7            338  PIT  ...        WR   7    1.0
3      Julio Jones  5.7            131  ATL  ...        WR   9    3.2
4  Ezekiel Elliott  6.2            180  DAL  ...        RB   8    2.8
```

That was easy! Let's wrap it in a function to make calling this API a little more flexible.

```
def fc_adp(scoring='ppr', teams=12, year=2019):
    ffc_com = 'https://fantasyfootballcalculator.com'
    resp = requests.get(
        f'{ffc_com}/api/v1/adp/{scoring}?teams={teams}&year={year}'
    )
    df = DataFrame(resp.json()['players'])

    # data doesn't come with teams, year columns, let's add them
    df['year'] = year
    df['teams'] = teams
    return df
```

Now we can get any arbitrary ADP — 10 team standard scoring for 2019 say — in one line.

```
In [8]: df_10_std = fc_adp('standard', 10, 2019)

In [9]: df_10_std.head()
Out[9]:
                name  adp  times_drafted  ... stdev  year  teams
0     Adrian Peterson  1.3            529  ...   0.6  2009     10
1  Maurice Jones-Drew  2.6            292  ...   1.0  2009     10
2      Michael Turner  2.9            218  ...   1.1  2009     10
3          Matt Forte  3.8            512  ...   1.2  2009     10
4    Larry Fitzgerald  6.1            274  ...   1.9  2009     10
```

Or maybe we want to get 10 years worth of ADP results in a few lines of code:

```
In [10]: df_history = pd.concat(
             [fc_adp('standard', 12, year=y) for y in range(2009, 2020)],
             ignore_index=True)

In [11]: df_history.head()
Out[11]:
                name  adp  times_drafted  ... stdev  year  teams
0     Adrian Peterson  1.3            529  ...   0.6  2009     12
1  Maurice Jones-Drew  2.6            292  ...   1.0  2009     12
2      Michael Turner  2.9            218  ...   1.1  2009     12
3          Matt Forte  3.8            512  ...   1.2  2009     12
4    Larry Fitzgerald  6.1            274  ...   1.9  2009     12
```

HTTP

This isn't a web programming book, but it will be useful to know a tiny bit about how HTTP works.

Anytime you visit a website, your browser is making an HTTP *request* to some web server. For our purposes, we can think about a request as consisting of URL URL we want to visit, plus some optional data. The web server handles the request, then — based on what is in it — sends an HTTP *response* back.

There are different types of requests. The one you'll use most often when dealing with public APIs is a *GET* request, which just indicates you want to read ("get") some data. Compare that with a *POST* request, where you're sending data along with your request, and want to do the server to do something with it. For example, if someone signs up for fantasymath.com, I might send a POST request containing their user data to my API to add them to the database.

There are other types of requests, but *GET* is likely the only one you'll really need to use.

JSON

When you visit (request) a normal site, the response consists of HTML (which your browser displays on the screen), but with an API you usually get data instead of HTML.

There are different data formats, but the most common is JSON, which stands for java script object notation. Technically, JSON is a string of characters, (i.e. it's wrapped in ""), which means we can't do anything to it until you convert (*parse*) it to a more useful format.

Luckily that's really easy, because inside that format is just a (likely nested) combination of the python equivalent to dict, list, string's, and numbers.

```
"""
{
    "players": ["aaron-rodgers", "aaron-jones"],
    "positions": ["qb", "rb"],
    "season": 2018
}
"""
```

To convert our JSON string to something useful to python we just do `resp.json()`. But note, this won't work on just anything we throw at it. For example, if we had:

```
"""
{
    "players": ["aaron-rodgers", "aaron-jones",
    "positions": ["qb", "rb"],
    "season": 2018
}
"""
```

we'd be missing the closing `]` on `players` and get an error when we try to parse it.

Once we do parse the JSON, we're back in Python, just dealing with manipulating data structures. Assuming we're working with tabular data, our goal should be to get it in Pandas ASAP.

In the FCC data this data happened to be in a format Pandas could read easily, i.e. a - list of dicts, where dict has the same fields. It's easy to convert to a DataFrame, but not all data is like that.

Often it's more complicated (e.g. the public API on myfantasyleague.com) and you have to use Python skills to get data into something (like a list of dicts) that you can put in a DataFrame.

This is where it's very helpful and powerful to have a good understanding of lists and dicts, and also comprehensions. It might be a good time to go back and review the intro to python section.

APIs have some benefits over scraping HTML. Most importantly, because they're specifically designed to convey data, they're usually much easier to use and require much less code. We saw that when we rewrote our webscraper to use Fantasy Football Calculator's API.

Also, sometimes data is available via an API that you can't get otherwise. In the past, website contact was primarily HTML + CSS, which meant you could generally get data out of it. In the past few years, however, much of what used to be done in HTML and CSS is now loaded — and hidden to scrapers — in JavaScript. If you try right clicking and viewing source in your gmail inbox for example, what you'll see is mainly nonsensical, minified JavaScript code. As more and more websites are built like this, sometimes getting data via an API is the only option.

That's not to say APIs don't have their disadvantages though. For one, they aren't as common. A website builder has to explicitly design, build and expose it to the public for it to be useful. Also when a public API exists, it's not always clear on how to go about using it — documentation for APIs can be out of date and sometimes doesn't even exist.

Exercises

5.2.1

The league hosting site myfantasyleague.com has an api available for (limited) use.

The endpoints for ADP and player info are:

https://api.myfantasyleague.com/2019/export?TYPE=adp&JSON=1

https://api.myfantasyleague.com/2019/export?TYPE=players&JSON=1

respectively.

Connect to them, grab the data, and use it to make an ADP table of top 200, non-IDP players.

6. Summary Stats and Data Visualization

Introduction to Summary Stats

In the first section of the book we talked about analysis as deriving useful insights from data. In this section we'll go over some of the main, non-modeling (which we'll do modeling in the next chapter) types of analysis. We'll learn about simple summary statistics and data visualization and about how these are often different ways of conveying the same information.

I think this information roughly falls into one of three categories:

1. Information about the *distribution* of a single variable.
2. Information about the *relationship* between two or more variables.
3. Information about a bunch of variables via one, composite score.

Distributions

A distribution is a way to convey the *frequency* of outcomes for some variable.

For example, take fantasy points. Here are 25 random RB point performances from 2017:

```
player_name team  week  pts
  D.Freeman  ATL     2   24
   M.Ingram   NO     9    9
   A.Ekeler  LAC    13    6
     C.Sims   TB     2    0
   M.Ingram   NO    14   13
   J.Conner  PIT    13    1
   L.Murray  MIN    16    8
     C.Sims   TB     7    4
 C.Anderson  DEN    15   16
     M.Mack  IND     1    9
  T.Riddick  DET    14   28
   L.Murray  MIN    17   25
L.Fournette  JAX     6   22
   M.Breida   SF     2    5
     F.Gore  IND    12   16
   A.Ekeler  LAC    11   14
    L.McCoy  BUF    10    6
.Cunningham  CHI    12    2
   A.Kamara   NO     7   16
    J.Ajayi  PHI     2   15
  J.Charles  DEN    14    4
  D.Freeman  ATL    16    7
     F.Gore  IND    17   14
      C.Ham  MIN    15    2
     L.Bell  PIT    11   27
```

Let's arrange these smallest to largest, marking each with an x and stacking x's when a score shows up multiple times. Make sense? Like this:

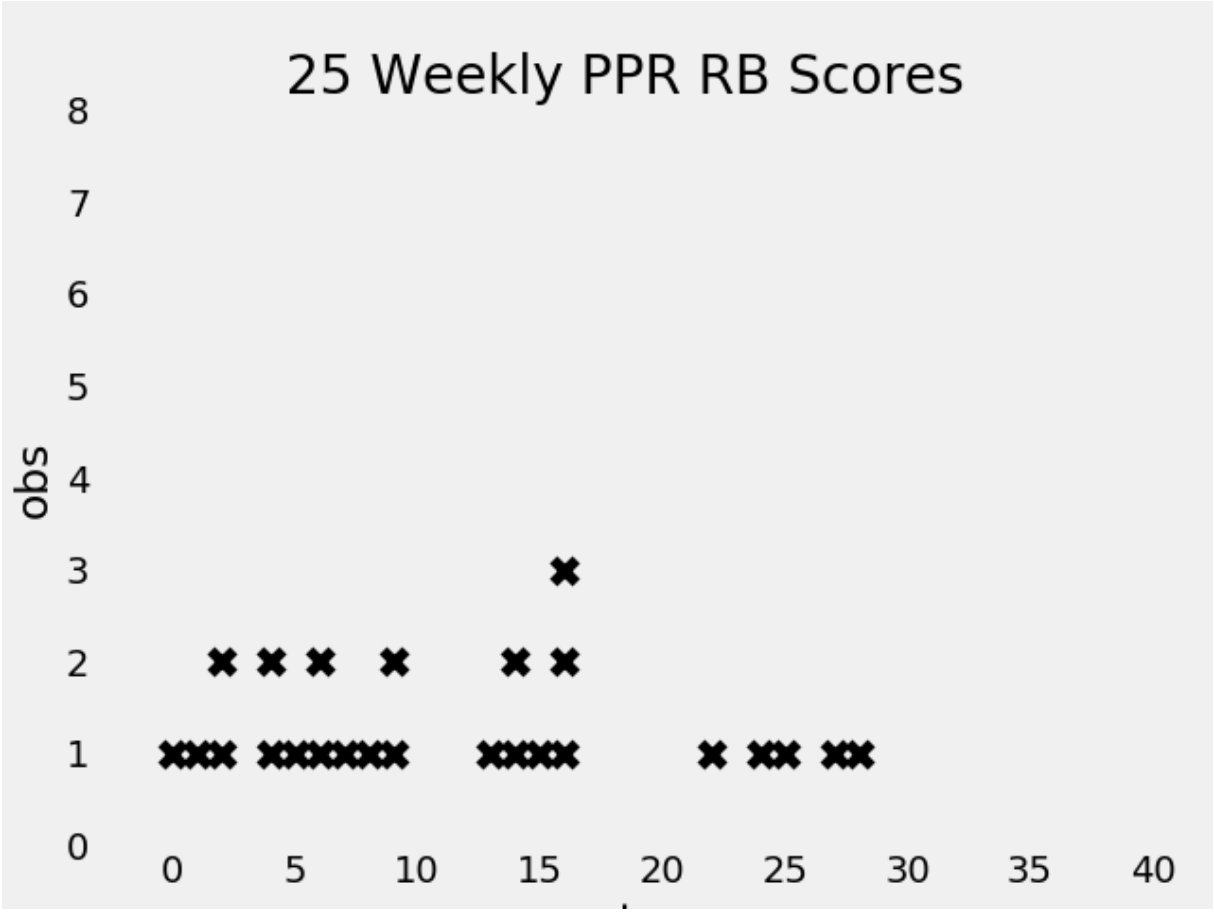


Figure 0.1: 25 RB Performances from 2017

Interesting, but why should we limit ourselves to just 25 player-game combinations? Let's do all of them:

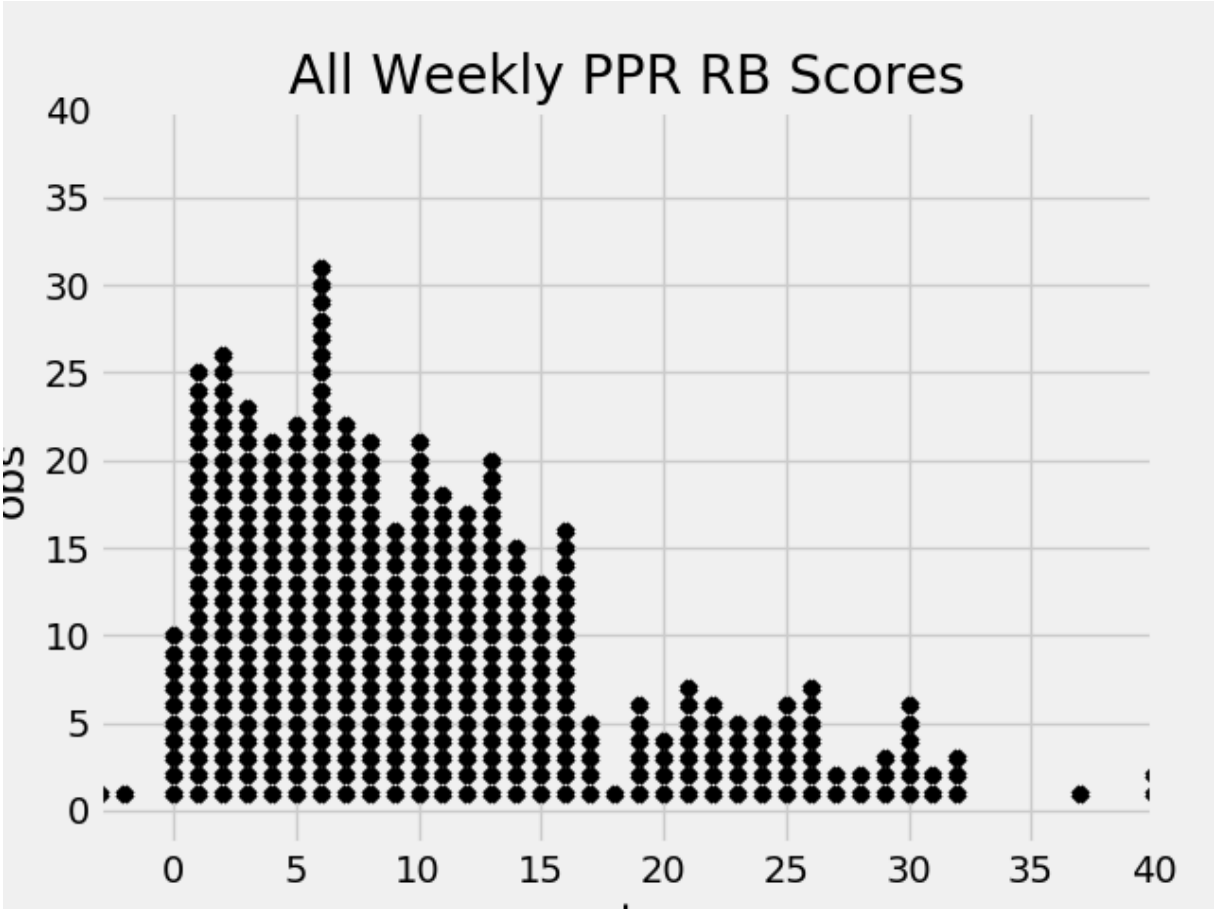


Figure 0.2: All RB Performances from 2017

These stacked x's show us the *distribution* of running back points in 2017. Understanding distributions is important if you want to take a statistical approach to fantasy football. When you start a RB any given week, you're picking one of these little x's out at random. Each *x* is equally likely to get picked. Each *score* is not. There are lot more x's between 0-10 points than there are between 20 and 30.

In this case, I rounded points to the nearest whole number, and stacked x's on top of each, but it may make more sense to treat points as a *continuous* variable, one that can take on any value. In that case, we'd move from stacked x's to area under a curve, like this:

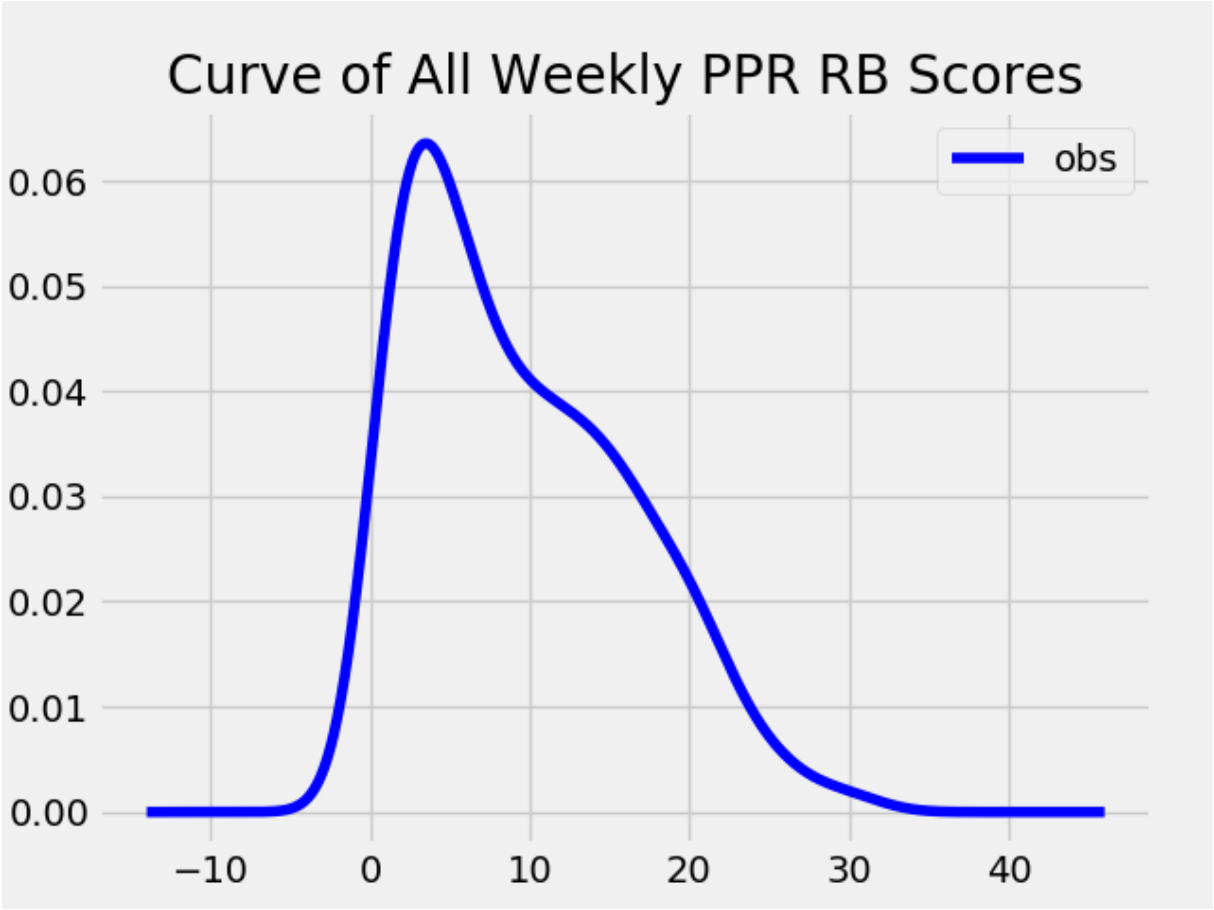


Figure 0.3: Kernel Density - All RB Performances from 2017

I'm not going to get into the math on this but the concepts are similar. From here on out we'll usually use curves. For our purposes — which will mainly be plotting and visualizing these curves — we can mentally treat them as smoothed out stacks of x's that are drawn so the area under the curve equals 1. These curves are called **kernel densities**[1].

Summary Stats

Summary stats convey information about a variable's distribution in just a few numbers. Sometimes they're called **point estimates**, which makes sense because they reduce the entire two dimensional distribution to a single number.

One example of a summary statistic is the **median**, which is the number where half area under the

[1]Kernel densities are more flexible than traditional probability density functions like the normal or gamma densities. They can follow the "bumpiness" of our stacked x's — but they're more complicated mathematically.

curve (or x's) is above above, the other half below.

A median is just a named example (the 50th) of a more general **percentile**. Pandas can calculate any percentile x, which is just the number where x% of the area is below and 100-x% is above.

You can get any percentile in Pandas with the quantile function. For example, what's the 90% percentile for rushing yards among RBs in our data?

```
In [45]: df = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv
    '))

In [51]: df.loc[df['pos'] == 'RB', 'rush_yards'].quantile(.9)
Out[51]: 95.0
```

The code above — and in the rest of this chapter — is in `06_summary.py`.

You can also calculate multiple statistics — including several percentiles — at once in Pandas with `describe`.

```
In [52]: df[['rush_yards', 'rec_yards']].describe()
Out[52]:
        rush_yards    rec_yards
count  1431.000000  1431.000000
mean     14.909154    32.761705
std      29.182716    33.848901
min      -9.000000    -4.000000
25%       0.000000     6.500000
50%       0.000000    24.000000
75%      16.000000    48.500000
max     203.000000   224.000000
```

The `describe` results also includes the mean (average). When it comes to probability distributions you can also interpret the mean as the **expected value** — the probability weighted sum of all the outcomes.

So if 2% of RBs get 1 pt, 3% get 2 pts … etc, the expected value (and also the mean) would be: .02*1 + .03+2 …

This isn't a coincidence, it's just math. When you manipulate the algebra, multiplying every term by the probability it occurs and adding them all up is another way of saying, "add up all the x values and divide by the total number of x's", i.e. "take the average".

Other summary stats summarize dispersion — how close or far apart the observations are. The standard deviation, for instance, is (basically) the average distance to the mean. You (1) figure out the average of your observations, (2) figure out how far each observation is from the average, and (3) take the average of that [2] It's smaller when values are tightly clustered around their average, larger when

[2] Technically, there are some slight modifications, but it's the general idea.

values are more dispersed around the mean.

There are times when using actual numbers in place of distributions is useful, mainly because distributions can be difficult to manipulate mathematically.

For example, if you want to project your team's projected points for the week — and somehow had access to each players projected point distributions — it'd be easier to just add up the mean of each distribution vs adding up the distributions, then trying to take the mean of that.

But overall, I think working with distributions is underrated.

First, understanding them will make you a better fantasy player. Recognizing that a player's weekly score is a random draw from some distribution helps you make better decisions in the long run. It makes it clear that your job in building a fantasy team is to send out a team with curves as far to the right as possible.

Does that mean you won't ever start the "wrong" guy? No, point distributions — especially for tight "who do I start decisions" — are very close, with a lot of overlap. It's normal that occasionally you'll get a bad draw from your starter, a good draw from your backup, or both. In fact, it'd be weird if it *didn't* ever happen. When it does, understanding distributions leads to less point chasing and second guessing, and hopefully more optimal decisions int he future.

But we're not just interested in becoming better fantasy players, we're interested in learning how to get fantasy insight from data. And plotting probability densities are one of the best ways to do this, because of how much information they can convey at a glance.

In the next section we'll learn how.

Density Plots with Python

Unlike data manipulation, where Pandas is the only game in town, data visualization in Python is a bit more fragmented.

The most common tool is matplotlib, which is very powerful, but is also trickier to learn[3]. One of the main problems is that there are multiple ways of doing the same thing, which can get confusing.

So instead we'll be using another library, seaborn, which is built on top of matplotlib. Though it's not as widely used, seaborn is still very popular and it comes bundled in with Anaconda. We'll go over some specific parts that I think provide the best mix of functionality and ease of use. Again, the code below is in `06_summary.py`.

First we have to import seaborn; the convention is to do so as `sns`.

[3]Note: "trickier" is relative, if after reading this book you wanted to sit down with the matplotlib documentation and master matplotlib, you'd be able to do it.

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

Then let's load our player-game data and use the raw stat categories to calculate points using a few common scoring systems

```
df['std'] = (0.1*(df['rush_yards'] + df['rec_yards']) +
             0.04*df['pass_yards'] +
             -3*(df['rush_fumbles'] +df['rec_fumbles'] +
                 df['interceptions']) +
             6*(df['rush_tds'] + df['rec_tds']) + 4*df['pass_tds'])
df['ppr'] = df['std'] + 1*df['receptions']
df['half_ppr'] = df['std'] + 0.5*df['receptions']
```

When making density plots in seaborn you basically have three big "levers" to work with. All three are specific columns in your data. Two of them are optional.

Let's start with just the first, the one required columns, which is the name of the variable we want to view the distribution of.

Say we're we want to plot the distribution of `df['std']`. This is how we'd do it:

```
g = sns.FacetGrid(df).map(sns.kdeplot, 'std', shade=True)
```

What we're doing is instantiating a FacetGrid (a powerful, seaborn plotting construct). Then, we're "mapping" the kdeplot function (for kernel density plot) to it, with shading option on (optional, but I think it looks better shaded).

I usually write it on two lines (you can put line breaks wherever you want in Python code by wrapping it with parenthesis) like this because I think it's a little easier to understand:

```
g = (sns.FacetGrid(df)
     .map(sns.kdeplot, 'std', shade=True))
```

Calling a bunch of methods in a row like this is called **method chaining**. See Figure 7 below for the result.

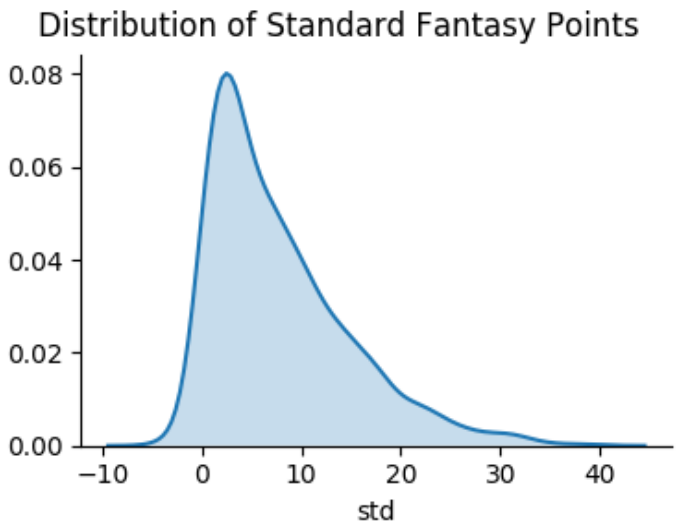


Figure 0.4: Distribution of Standard Fantasy Pts

Note, I've added a title and changed some height and width options to make things clearer. Yours won't show that yet. We'll cover these options when we go over other formatting options later.

There are quicker ways to make this specific plot, but I'd recommend sticking with the FacetGrid-then-map approach because it's easy to extend.

For example, say we want separate plots for position. Now we can introduce our second lever, the **hue** keyword, which gets passed to `sns.FacetGrid`.

```
g = (sns.FacetGrid(df, hue='pos')
     .map(sns.kdeplot, 'std', shade=True)
     .add_legend())
```

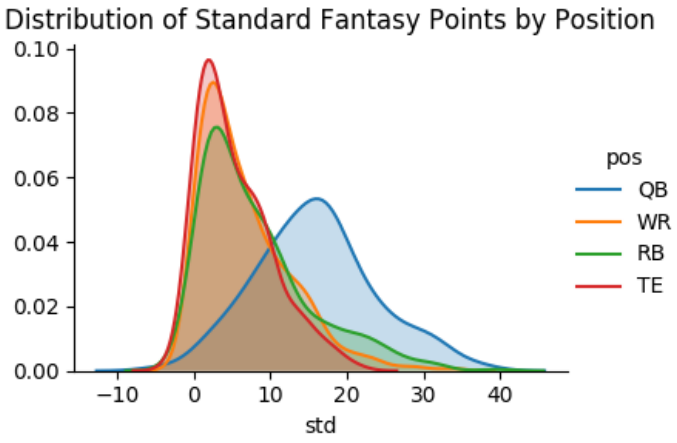


Figure 0.5: Distribution of Standard Fantasy Pts by Position

Again: hue always takes a column of data. By passing it the `df['pos']` column, I've told it to make different plots of `std` (with different colors, or hues) for each value of pos. `So now we can see we have one density of`std`when`pos='RB', `another for`pos='QB'' etc.

Now we're effectively showing the distribution of fantasy points over different positions. What if we wanted to add in another dimension, say distribution of points by position and week? That brings us to our third level — the col keyword.

```
g = (sns.FacetGrid(df, hue='pos', col='week', col_wrap=4, height=2)
     .map(sns.kdeplot, 'std', shade=True)
     .add_legend())
```

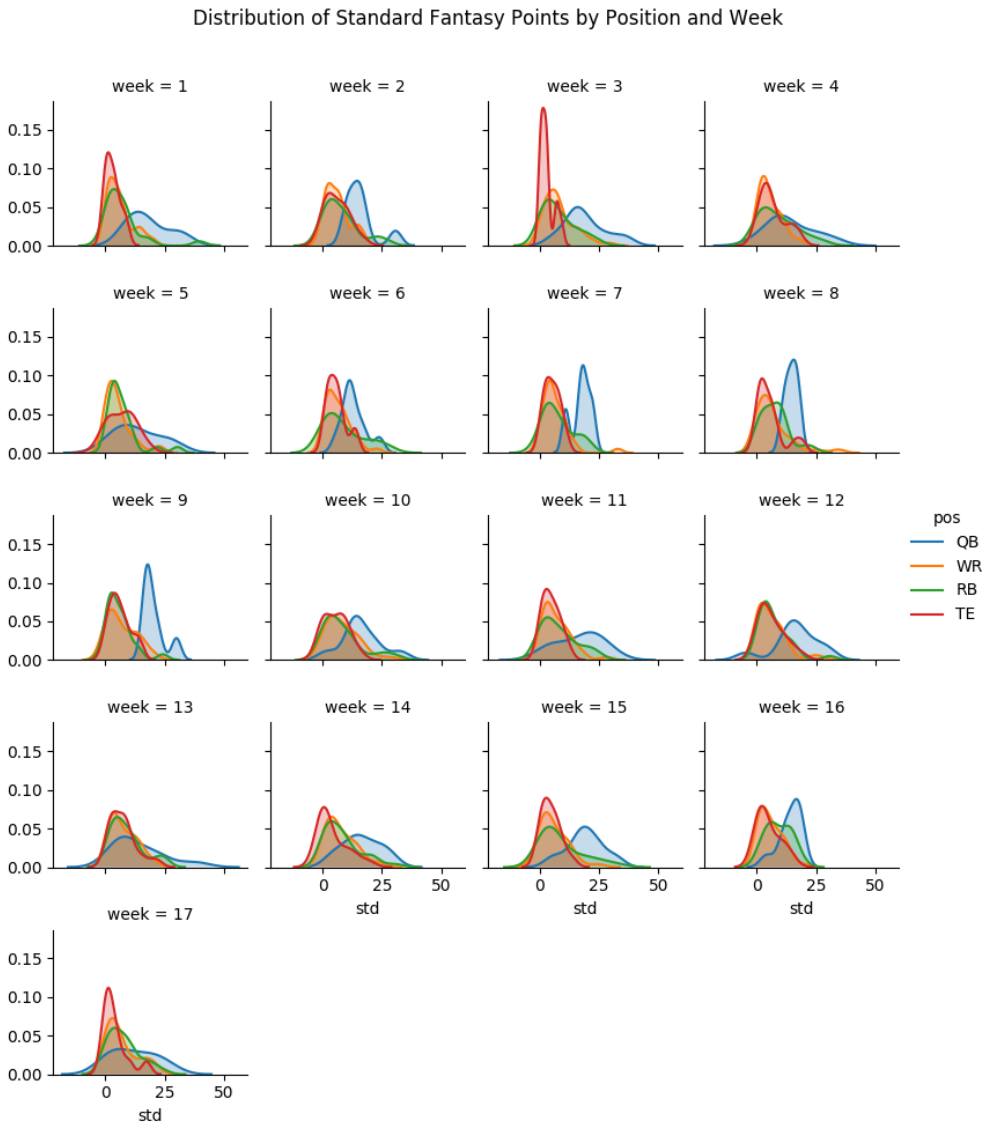


Figure 0.6: Distribution of Standard Fantasy Pts by Position and Week

Don't worry about the `col_wrap` and `height` options for right now, they're aesthetic and we'll cover them later.

This draws separate plots (one in every column) for every value of whatever you pass to col. So we have one column for `wee=1`, another for `week=2` etc.

Within each of these, the `hue=pos` draws separate curves for each position.

Note all three of these variables — std, pos, and week are columns in our data. Seaborn *needs* the data in this format to make these types of plots. It's not guaranteed that your data will automatically come like this.

For example, what if — instead of week — we want to look at scoring system. That's no problem conceptually, except our points data for our three scoring systems (standard, ppr, and half-ppr) are in three separate columns.

We currently have this:

```
In [74]: df[['pos', 'std', 'ppr', 'half_ppr']].head()
Out[74]:
  pos    std    ppr  half_ppr
0  QB  10.68  10.68     10.68
1  QB  31.62  31.62     31.62
2  WR   7.00  13.00     10.00
3  RB   2.30   3.30      2.80
4  TE   4.40   9.40      6.90
```

But we actually need something more like this:

```
  pos   scoring    pts
0  QB       std  10.68
1  QB       std  31.62
2  WR       std   7.00
3  RB       std   2.30
4  TE       std   4.40
0  QB       ppr  10.68
1  QB       ppr  31.62
2  WR       ppr  13.00
3  RB       ppr   3.30
4  TE       ppr   9.40
0  QB  half_ppr  10.68
1  QB  half_ppr  31.62
2  WR  half_ppr  10.00
3  RB  half_ppr   2.80
4  TE  half_ppr   6.90
```

They contain the same information, we've just changed the granularity (from player-game to player-game-scoring system) and shifted data from columns to rows.

If you've read the Python and Pandas section, you should know everything you need to do this, but let's walk through it for review.

First let's build a function that — given our data (`df`) and a scoring system (`std`) — moves that score (e.g. df['std']) to a points column, then adds in another column indicating what type of points they are. So this:

```
def score_type_df(_df, scoring):
    _df = _df[['pos', scoring]]
    _df.columns = ['pos', 'pts']
    _df['scoring'] = scoring
    return _df
```

And to use it with `std`:

```
In [82]: score_type_df(df, 'std').head()
Out[82]:
  pos    pts scoring
0  QB  10.68     std
1  QB  31.62     std
2  WR   7.00     std
3  RB   2.30     std
4  TE   4.40     std
```

That's what we want, we just need to call it on all three of our scoring systems, then stick the resulting DataFrames on top each other (like a snowman). Recall vertical stacking (concatenation) is done with the `concat` function.

Unlike `merge`, `concat` takes a list (so it can take more than one) of DataFrames and combines them all.

So we need a list of DataFrames: one with standard scoring, another with ppr, and another with half-ppr, and then we need to pass them to `concat`. Let's do it using a list comprehension.

```
In [84]:
df_pts_long = pd.concat(
    [score_type_df(df, scoring) for scoring in ['std', 'ppr', 'half_ppr'
       ]],
    ignore_index=True)
```

Now we have what we want: the same position, points, and scoring information we had before, but now in three separate columns. Now we can passing it to seaborn:

```
g = (sns.FacetGrid(df_pts_long, col='pos', hue='scoring', col_wrap=2,
                   aspect=1.3)
     .map(sns.kdeplot, 'pts', shade=True)
     .add_legend())
```

The call to FacetGrid in seaborn is easy once we get our data in the right format. This is a good example of how data manipulation is most of the work (both in time and lines of code) compared to analysis.

The final result:

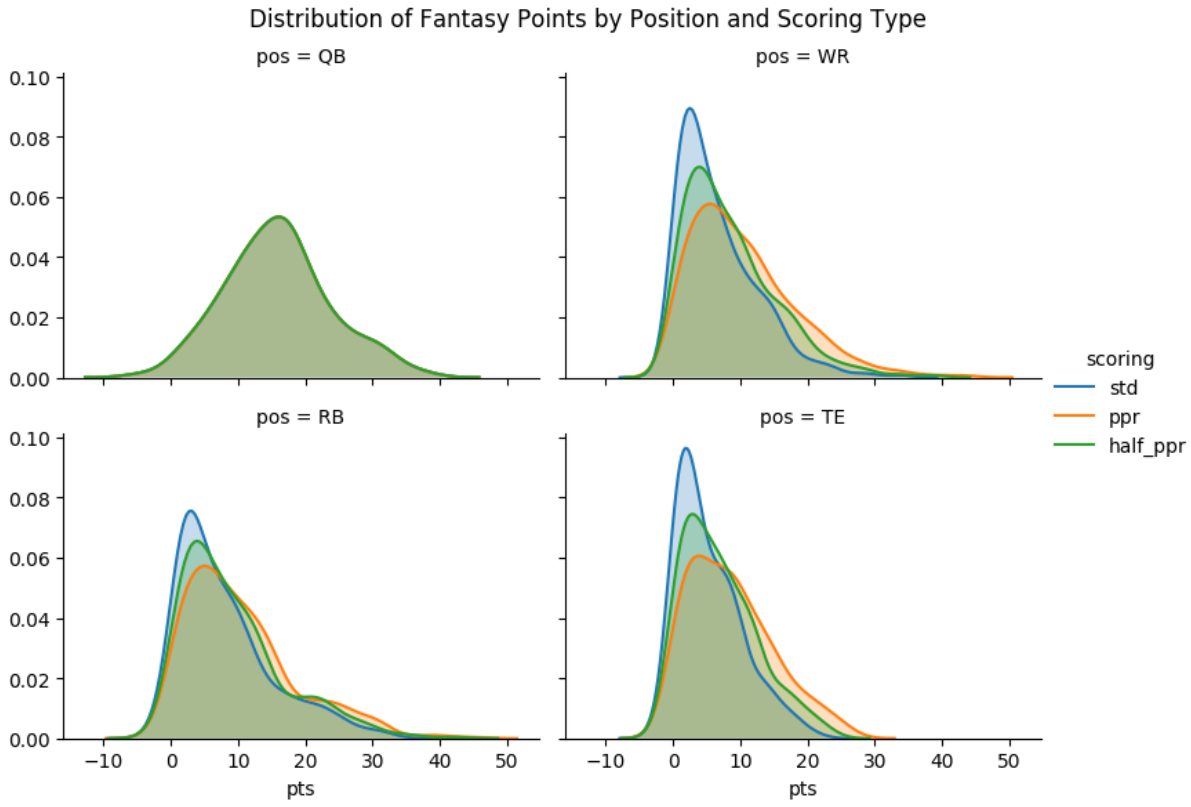


Figure 0.7: Distribution of Fantasy Points by Position and Scoring System

Relationships Between Variables

Leaving aside the relationships described by models, which we'll cover in the next section. Most non-modeling relationship insights deal with the relationship between two variables, and shed light on how two variables move together.

For example, in our player-game data we can break out a players receiving yards into two parts: air yards (how long the pass was in the air before the player caught it), and yards after the catch (YAC).

We might be wondering about the relationship between air yards and YAC: do players who have games with more air yards tend to have less yards after the catch and vis versa? Are there such things as "possession receivers"?

Let's check it out.

Scatter Plots with Python

The tool I use the most when looking at relationships between two variables is a scatter plot. Scatter plots are just plots of points on a standard coordinate system. One of your variables is your x axis, the other your y axis. Each observation gets placed on the graph. The result is a sort of "cloud" of points. The more the cloud generally moves from lower left to upper right, the stronger the relationship between the two variables.

In terms of actually making the plot in seaborn, scatter plots are a lot like the density plots we've already covered. Like density plots, you can pick out columns for hue and col and seaborn makes it easy to plot different slices of your data. If anything, making scatter plots in seaborn is actually easier than making a density plot because seaborn has a built in function to do it. This function — `relplot` (for *relationship* plot) — combines the FacetGrid and map steps into one.

So say we want to look at the relationship between air yards (the number of yards a pass is in the air) and receiving yards. Again, don't worry about the `subplots_adjust` and `suptitle` methods — they're just adding a title, and we'll cover those later.

```
g = sns.relplot(x='caught_airyards', y='raw_yac', data=df)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Airyards vs YAC')
```

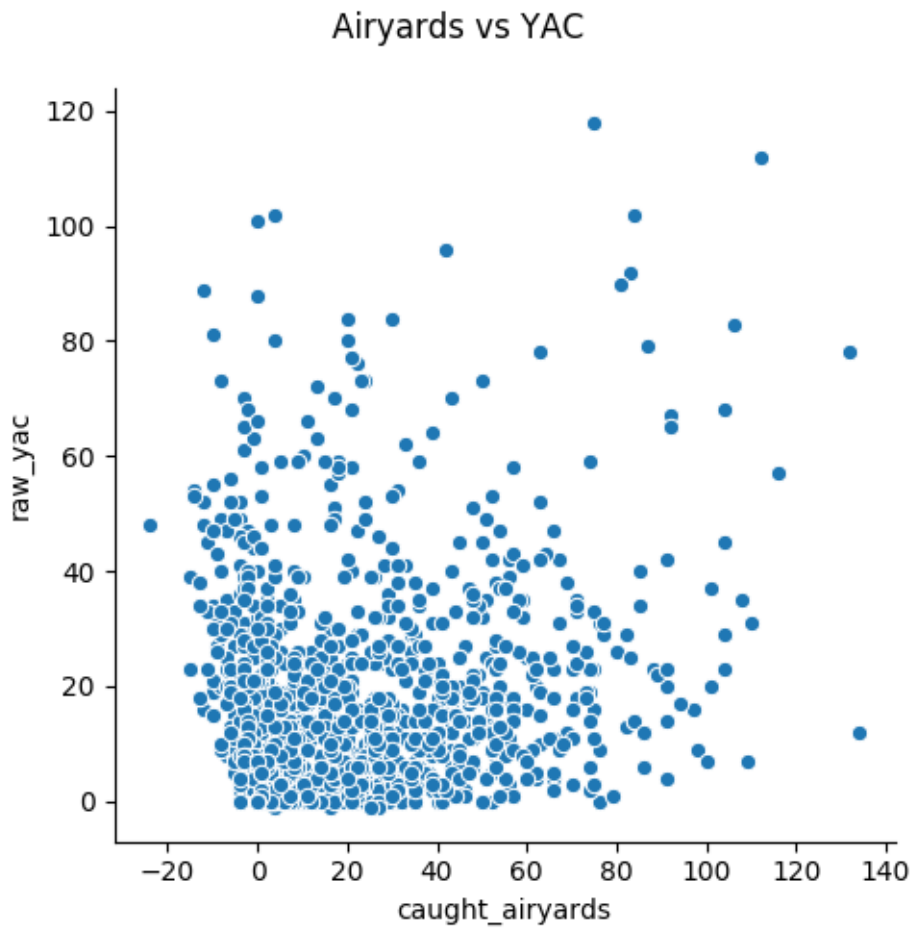


Figure 0.8: Air Yards vs YAC

Is this cloud of points moving up and to the right? It's hard for me to tell. If it is, the relationship isn't *that* strong. Later, we'll look at other, numeric options for measuring this but for now, let's color our plots by position. Just like the density plots, we do this using the hue keyword.

```
g = sns.relplot(x='caught_airyards', y='raw_yac', hue='pos', data=df)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Airyards vs YAC - by Position')
```

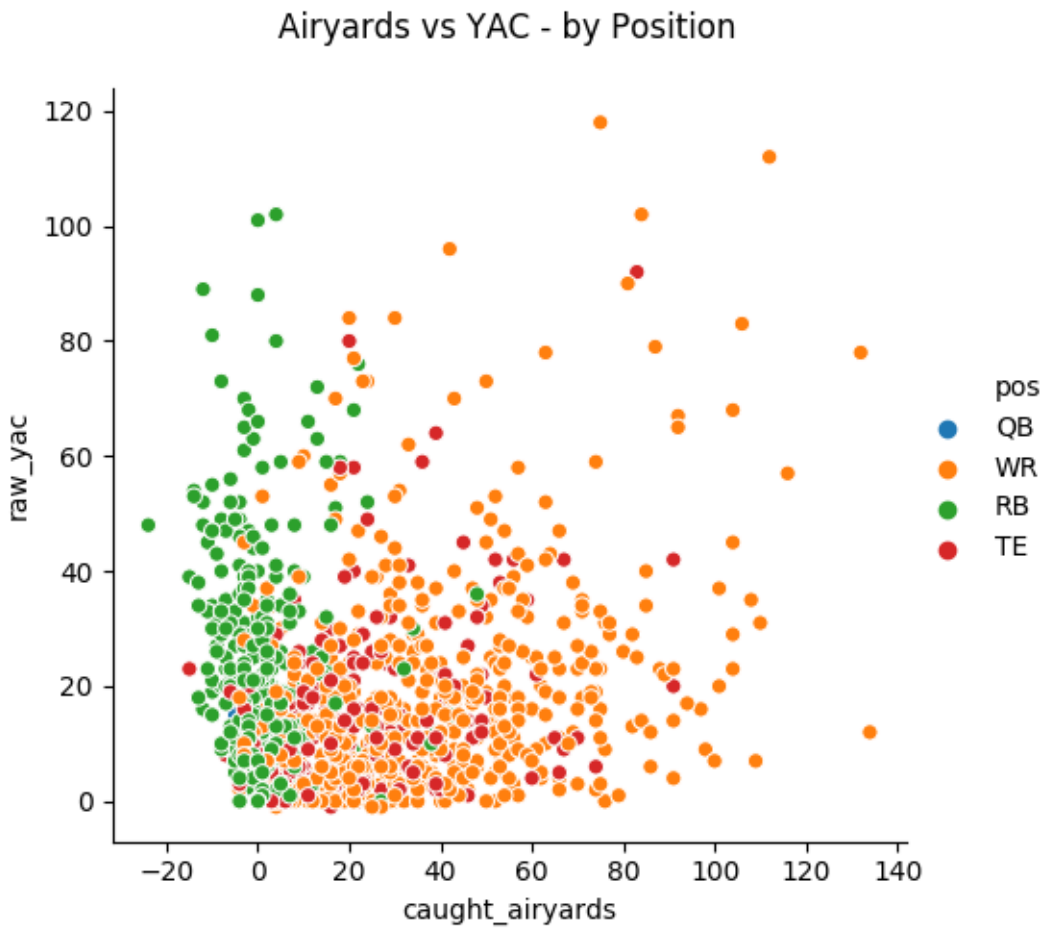


Figure 0.9: Air Yards vs YAC - by Position

This graph makes it obvious how running backs get most of their receiving yards from yards after the catch on short dump offs vs WRs and TEs. Note these are player-game *totals*, not plays, which is why some of the totals are more than 100.

Like density plots, scatter plots also take an optional `col` argument.

Correlation

Just like a median or mean gets at certain aspects of a variable's distribution, similar summary statics get at the relationship between variables.

The most basic is correlation. The usual way of calculating correlation (Pearson's) summarizes the tendency of variables to move together by giving you a number between -1 and 1.

Variables with a -1 correlation move perfectly in opposite directions; variables that move in the same direction have a correlation of 1. Two variables that are exactly the same are perfectly correlated, but so are variables that are just linear transformations of other variables. So number of touchdowns (say n) is perfectly correlated with points from touchdowns (6*n).

A correlation of 0 means the variables have no relationship; they're independent. Finding variables with no correlation isn't as easy as you might think. For example, is the total score of the NYG-PHI game in NYC correlated with the score of OAK-NE in Foxborough? You wouldn't think so, but what if there's a snow storm moving through the Northeast that puts a damper on scoring in both games? In that case they'd be positively correlated.

One interesting way to view correlations across multiple variables at once (though still in pairs) is via a correlation matrix. In a correlation matrix, the variables you're interested in the rows and columns. Then to check the correlation between any two variables you're interested in, you find the right row and column and look at the value.

For example, let's look at a correlation matrix for WR air yards (distance a ball traveled in the air for all targets regardless of whether it was caught), targets, rushing carries, receiving TDs, and PPR points.

In Pandas you can easily get a correlation matrix with the `corr` function.

```
In [61]:
df.loc[df['pos'] == 'WR',
       ['rec_raw_airyards', 'targets', 'carries', 'ppr', 'std']].corr()—

Out[61]:
          airyards  targets   carries       ppr       std
airyards  1.000000  0.791787 -0.042175  0.589707  0.553359
targets   0.791787  1.000000 -0.063028  0.723363  0.612425
carries  -0.042175 -0.063028  1.000000  0.008317  0.013609
ppr       0.589707  0.723363  0.008317  1.000000  0.976131
std       0.553359  0.612425  0.013609  0.976131  1.000000
```

Note that the diagonal elements are all 1. That's because every variable is perfectly correlated with itself. Also note the matrix is symmetrical around the diagonal. This makes sense; the correlation is like multiplication, order doesn't matter (the correlation between targets and carries is the same as the correlation between carries and targets).

To pick out any individual correlation, we can look at the row and column we're interested in. So we can see the correlation between air yards and PPR points is 0.5897.

Let's look at that on a scatter plot (note the `query` method on `df`, which we covered at the end of the filtering section on Pandas).

```
g = sns.relplot(x='rec_raw_airyards', y='ppr', data=df.query("pos == 'WR'"
    ))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Air Yards vs PPR Pts')
```

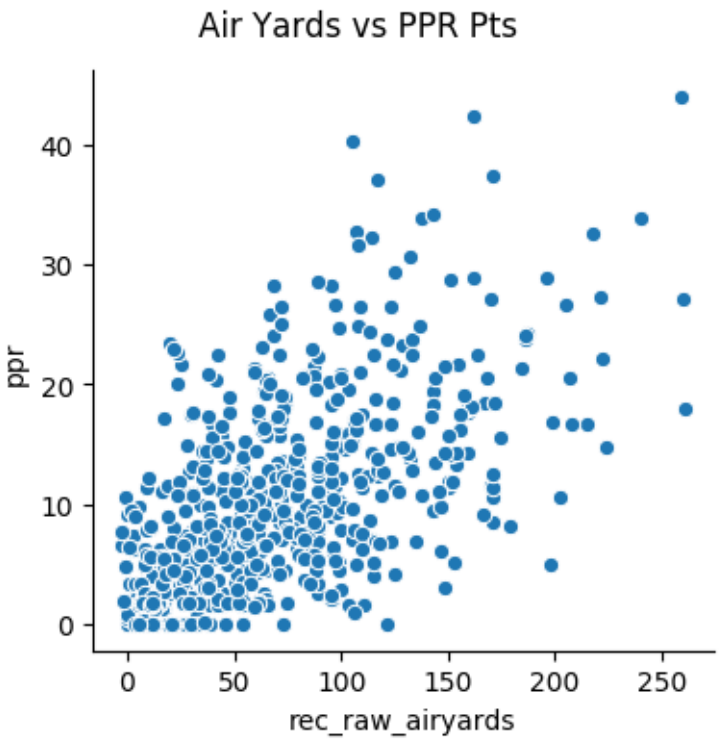


Figure 0.10: Air Yards vs PPR Points

We can see the cloud of points generally goes from bottom left to top right. What about something like standard vs PPR points, which have a 0.98 correlation?

```
sns.relplot(x='std', y='ppr', data=df.query("pos == 'WR'"))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('STD vs PPR Points')
```

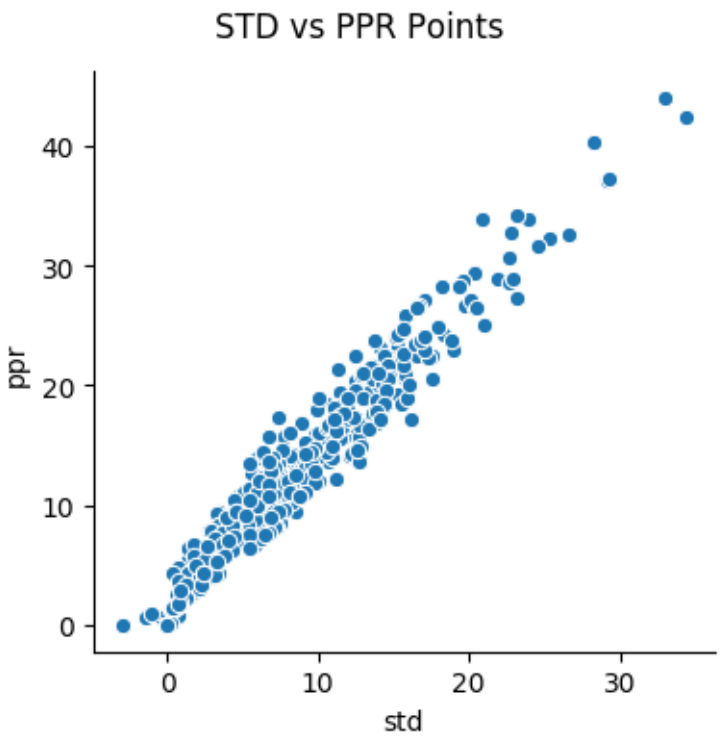


Figure 0.11: Standard vs PPR Pts

We can see those points are much more clustered around an upward trending line.

Line Plots with Python

Scatter plots are good for viewing the relationship between two variables generally, but when one of the variables is some measure of time (e.g. week 1-17) a lineplot is usually more useful.

You make a lineplot by passing the argument `kind='line'` to `relplot`. When you're doing a lineplot, you'll want your time variable to be on the x axis. Because it's still seaborn, we still have control over `hue` and `col` just like our other plots.

Let's try plotting points by position over the 2017 season.

```
g = sns.relplot(x='week', y='std', kind='line', hue='pos', data=df)
```

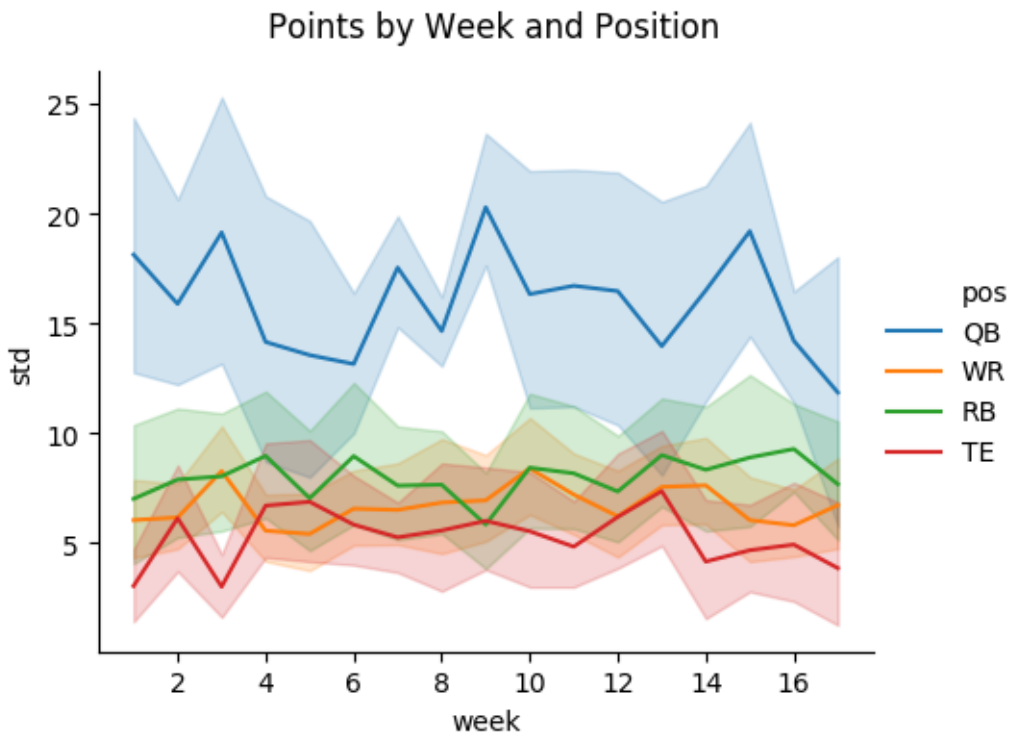


Figure 0.12: Points by Week, Position

Woah. What's happening here? We told seaborn we want week on the x axis and standard points on the y axis and that we wanted different plots (hues) for each position.

But remember, our data is at the *player-game* level, so for any given week and position — say week 1 QBs — there are multiple observations.

```
In [24]:
df.loc[(df['pos'] == 'QB') & (df['week'] == 1),
       ['player_name', 'week', 'std']].head()—

Out[24]:
          player_name  week    std
0             T.Brady     1  10.68
1             A.Smith     1  31.62
174          T.Taylor     1  18.30
287            M.Ryan     1  18.14
512  B.Roethlisberger     1  15.72
```

Instead of plotting separate lines for each, seaborn automatically calculates the mean (the line) and 95% confidence intervals (the shaded part), and plots that.

If we pass seaborn data with just one observation for any week and position — say the maximum score

by position each week, it'll plot just the single lines.

```
In [25]:
max_pts_by_pos_week = (df.groupby(['pos', 'week'], as_index=False)
                         ['std'].max())

g = sns.relplot(x='week', y='std', kind='line', hue='pos', style='pos',
                data=max_pts_by_pos_week, height=4, aspect=1.2)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle("Max Points by Week and Position")
```

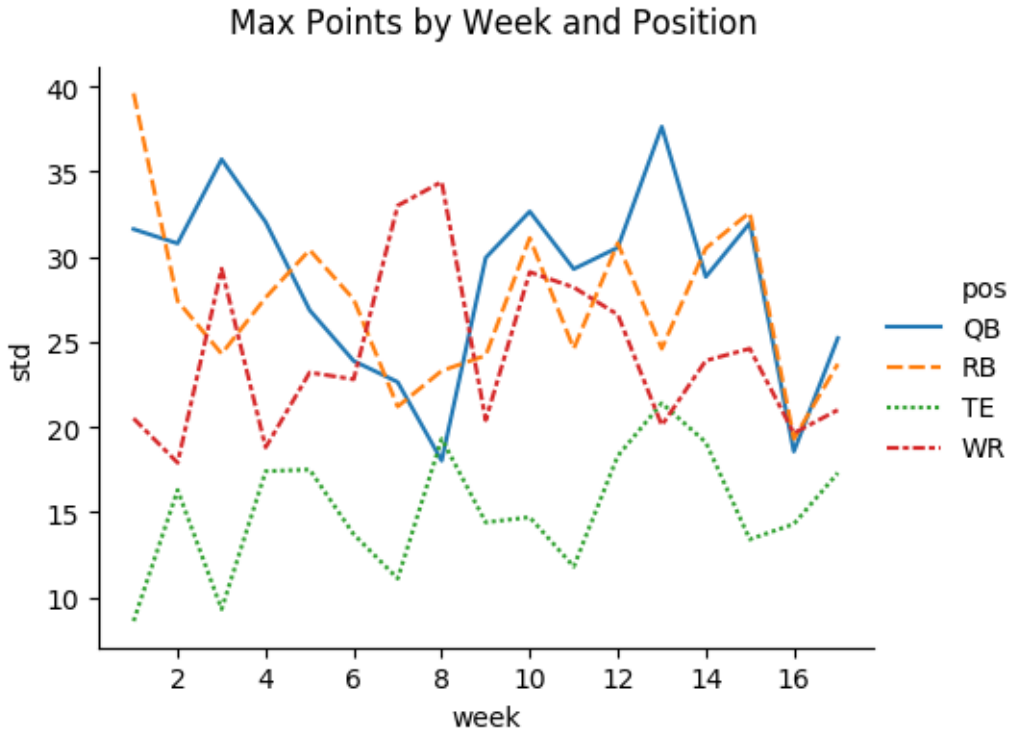


Figure 0.13: Max Points by Week, Position

Note in the previous plot we also included an additional fourth lever available for line plots — style — that controls how the lines look (e.g. dotted, dashed, solid etc). Like the density examples, relplots include options for separating plots by columns.

```
g = sns.relplot(x='week', y='std', kind='line', hue='player_name',
                col='player_name', height=2, aspect=1.2, col_wrap=3,
                legend=False, data=df.query("pos == 'QB'"))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle("Points by Week = QBs")
```

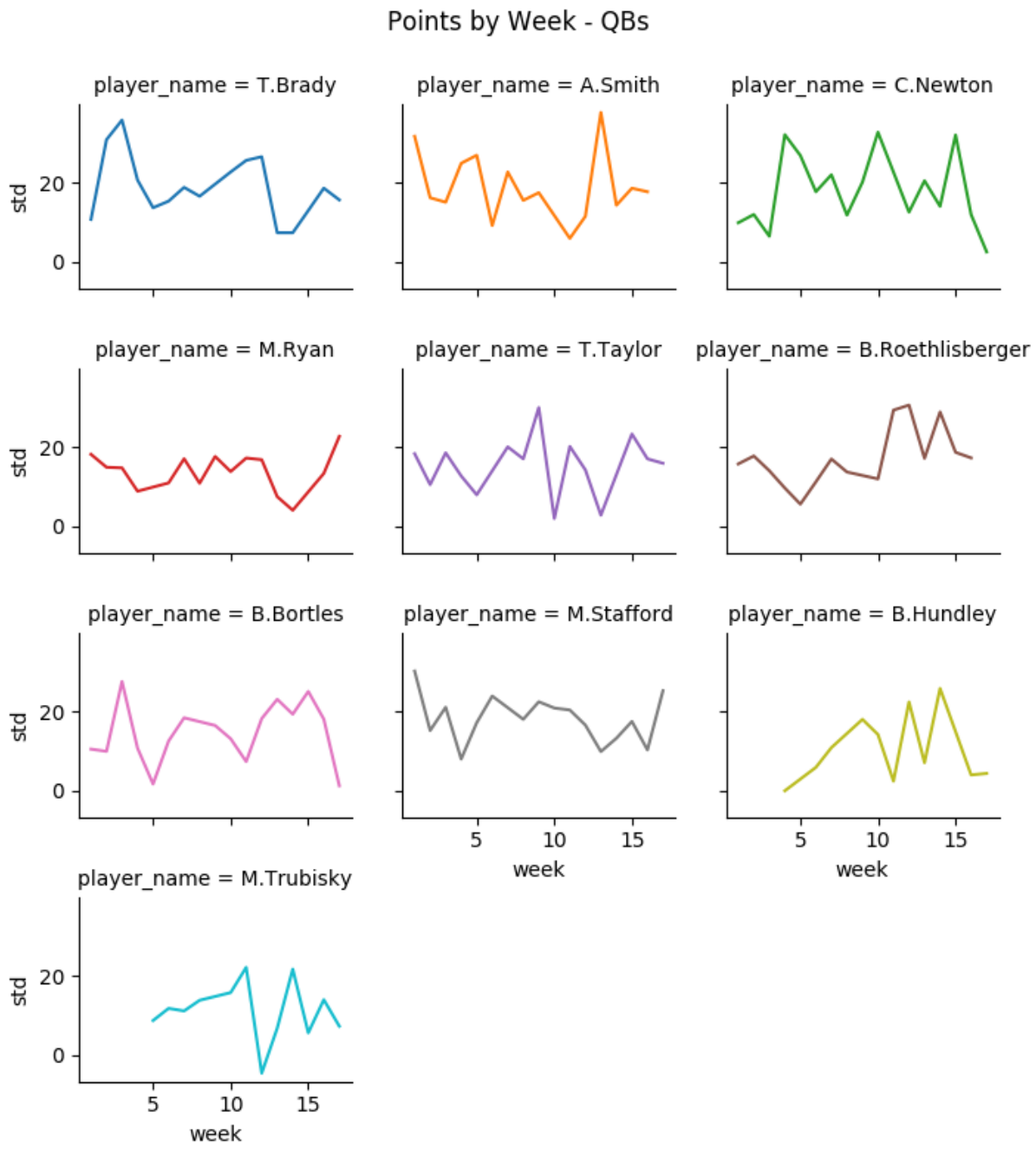


Figure 0.14: Points by Week - QBs

Again, these plots includes some other options — height, aspect, col_wrap, etc — we'll cover below.

Composite Scores

Another type of non-modeled data insight you sometimes see is a *composite score* like passer rating — the 0 to 158.3 scale for rating QBs — or the SPARQ score (speed, power, agility, reaction and quickness) — which you hear about sometimes during the NFL draft.

Most of these composite scores make use of two statistical concepts: weighting and rescaling.

Weighting is fairly common in statistical work. The idea is you take the average of a bunch of variables, but — rather than treating everything as equal — give more weight to certain columns. For example, maybe you're aggregating expert rankings and want to weight certain experts more than others. If it's helpful, you can view regular averages a special case of weighted averages where everything is weighted the same.

The other concept that comes up when combining several columns of data is the fact they aren't always on the same scale. For example, the SPARQ score is a combination of 40 yard dash, 20 yard shuttle, and vertical jump, among others. It'd make no sense to just take the average of these, weighted or otherwise. The solution is to make them on the same scale, usually by converting to a percentile. So if a 6'5", 260 pound outside rusher runs a 4.6 40 yard dash, that might be in the 98% percentile (he'd run faster than 98% of NFL players with a similar height and weight), and that would get combined with his other percentiles.

Personally, I don't do much with composite scores. I don't like that it's not obvious how they're calculated, and would usually rather just work with (potentially several columns of) real data instead.

Plot Options

The most important parts of the FacetGrid, then map approach for density plots and relplot (which just combines the FacetGrid and mapping into one command) are the data you're plotting, along with the optional hue and col keywords.

That gives you a powerful, flexible framework that — when combined with the ability to manipulate data in Pandas — should let you get your point across efficiently and effectively.

But there are a few other cosmetic options it'd be useful to know too. For the most part, these are the same for every type of plot we've covered, so I thought it'd be easier to cover them all at once.

Let's use our scoring system distribution plot from earlier:

```
g = sns.FacetGrid(df_pts_long, col='pos', hue='scoring')
g = g.map(sns.kdeplot, 'pts', shade=True)
```

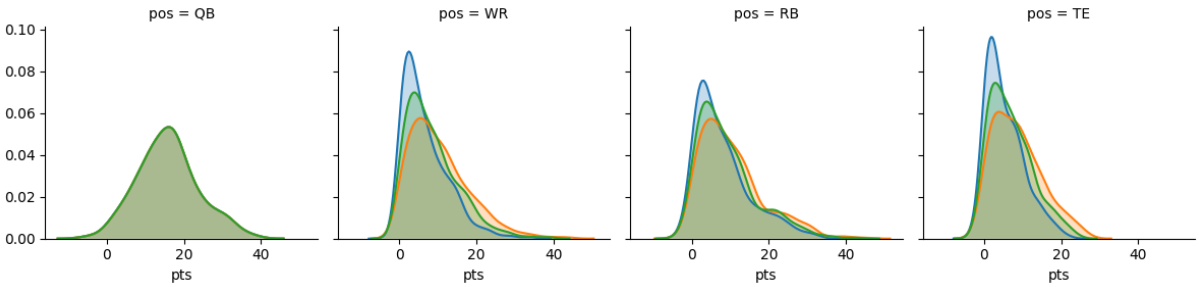


Figure 0.15: Basic Density Plot

Wrapping columns

By default, seaborn will spread all our columns out horizontally. We can change that with the `col_wrap` keyword, which will make seaborn start a new row after n number of columns. For our density plots, it gets passed to FacetGrid, while for scatter and line plots it gets passed to the `relplot` method itself.

```
g = sns.FacetGrid(df_pts_long, col='pos', hue='scoring', col_wrap=2)
g = g.map(sns.kdeplot, 'pts', shade=True)
```

Adding a title

Adding a title is a two step process. First you have to make room, then you have to add the title itself. The method is suptitle (for super title) because title is reserved for individual plots.

```
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Fantasy Points by Position, Scoring System')
```

This is something that seems like it should be easier, but it's a small price to pay for the overall flexibility of this approach. I'd recommend just memorizing it and moving on.

Modifying the axes

Though by default seaborn will try to show you whatever data you have — you can decide how much of the x and y axis you want to show.

You do that via the `set` method.

```
g.set(xlim=(-10, 40), ylim=(0, 0.1))
```

`set` is for anything you want to change on every plot. There are a bunch of options for it, but — apart from xlim and ylim — the ones I could see being useful include: yscale and xscale (can set to 'log') and xticks and yticks.

To change the x and y labels you can use the special `set_xlabels` and `set_ylabels` methods.

```
g.set_xlabels('points')
g.set_ylabels('density')
```

Legend

Seaborn will automatically add a legend to a relplot when you use the hue keyword. If you don't want it you can pass it `legend=False`.

For density plots, you need to add the legend yourself. You can do that easily with:

```
g.add_legend()
```

Plot size

The size of plots in seaborn is controlled by two keywords: height and aspect. Height is the height of each of the individual, smaller plots (denoted by col). Width is controlled indirectly, and is given by aspect*height. I'm not positive why seaborn does it this way, but it seems to work OK.

Whether you want your aspect to be greater, less than or equal to one (the default) depends on the type of data you're plotting.

I also usually make plots smaller when making many little plots.

Saving

To save your image you just call the `savefig` method on it, which takes the file path to where you want to save it. There are a few options for saving, but I usually use `png`.

```
g.savefig('points_by_type_pos.png')
```

There are many more options you can set when working with seaborn visualizations, especially because it's built on top of the extremely customizable matplotlib. But this covers most of what I need.

If you do find yourself needing to do something — say modify the legend say — you should be able to find it in the seaborn and matplotlib documentation (and stackoverflow) fairly easily.

Here's our final plot after adjusting all those options.

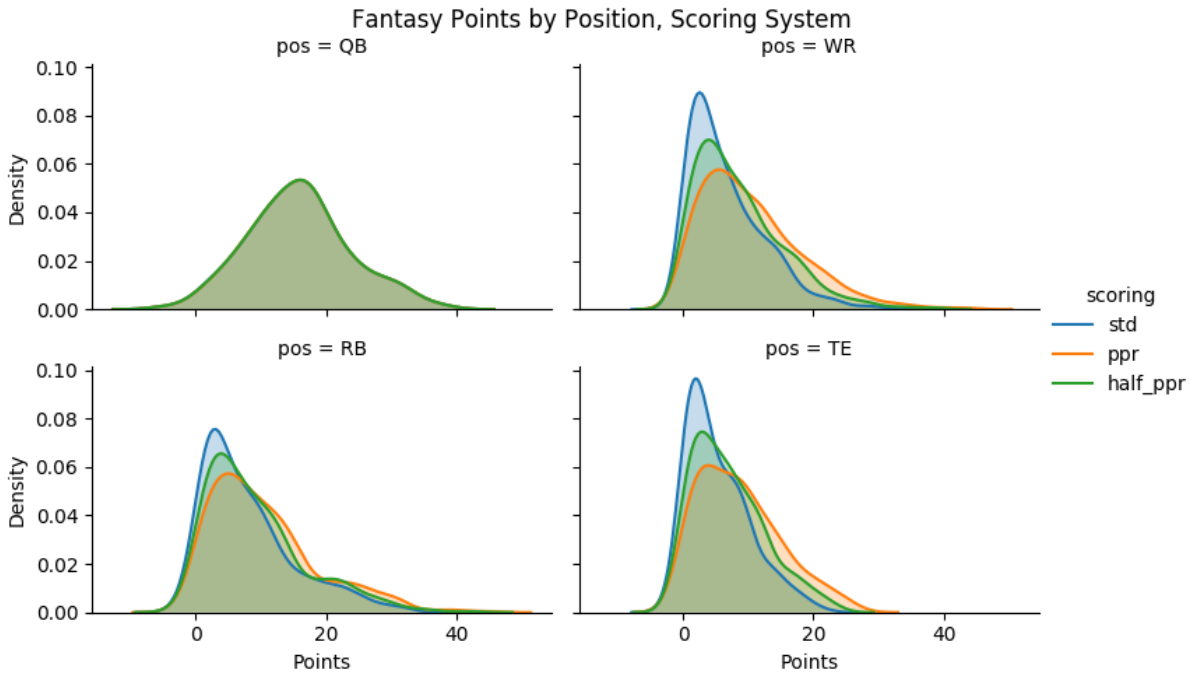


Figure 0.16: Formatted Basic Density Plot

End of Chapter Exercises

6.1

a) Using the play by play data, plot the distribution of yards gained. Make sure to give your plot a title.

Now modify your plot to show the distribution of yards gained by down. Restrict your analysis to downs 1-3. Do it (b) with down as separate colors on the same plot (make sure you add a legend), and (c) also with downs as separate plots.

(d) Using your down as columns plot from (c) add `row='posteam'` keyword. We didn't cover this in the chapter, but what's is it doing?

(e) Sometimes it's effective to use the multiple keywords to display redundant information, experiment with this.

6.2

Load the player-game data.

(a) Plot the relationship between carries and rushing yards by position. Again, make sure your plot has a title.

Based on this plot, which position do you think averages the most yards per carry?

(b) Verify this in Pandas.

7. Modeling

Introduction to Modeling

In the first section of the book we talked about how a model is the details of a relationship between an output variable and one or more input variables. Here we'll be talking about what our model actually is and how to find it in Python.

The Simplest Model

Let's say we want a model that takes in distance to the end zone in yards and predicts whether a play from there will be a touchdown or not. So we might have something like:

```
touchdown or not = model(yards to endzone)
```

Terminology

First some terminology: the variable "touchdown or not" is our **output variable**. There's always exactly one output variable. Other terms this variable include: **left hand side variable** (it's to the left of the equal sign); **dependent** variable (its value *depends* on the value of yards to endzone), or **y** variable (traditionally output variables are denoted with y, inputs with x's).

The variable "yards to endzone" is our **input variable**. In this case we just have one, but we could have as many as we want. For example:

```
touchdown or not = model(yards to endzone, time left in game)
```

Other words for input variables include: **right hand side**, **independent explanatory**, or **x** variables.

OK. Back to:

```
touchdown or not = model(yards to endzone)
```

Here's a question: what is the simplest implementation for model we might come up with here? How about:

```
model(...)= No
```

So give it any yards to go, our model spits out: "no, it will not be a touchdown". Since the vast majority of plays are not touchdowns, this model will be very accurate. But since it never says anything besides no, it's not that interesting or useful.

What about:

```
prob td = 1 + -0.01*yards to go + -0.0000001*yards to go ^ 2
```

So for 1 yard we'd get prob of 0.99, 3 yards 0.97, 10 yards 90% and 99 yards 0.0002%. This is more interesting. It isn't a *good* model (for 50 yards it gives about a 0.50 probability of a play being a TD, which is way too high) — probably because I just made these numbers up — but it is an example of a model that transforms our input data to our output variable using some mathematical function

Linear regression

This type of model format:

```
data we care about =
    some number + another number*data + yet another number*other data
```

is called **linear regression**. It's *linear* because when you have one piece of data, the equation is a line on a set of x, y coordinates, like this:

```
y = m*x + b
```

If you recall math class, `m` is the slope, `b` the **intercept**, and `x` and `y` the horizontal and vertical axes.

Notice instead of saying "some number", "another number" and "input and output data" we use b, m, x and y. This shortens things and gives you an easier way to refer back to specific parts of the equation. The particular letters we use for these don't matter (though the convention is to use certain values to make it easier for others to understand what you're talking about). The point is to provide an abstract way of thinking about and referring to our model.

A linear equation can have more than one data term in it, which is why statisticians use b0 and b1 instead of b and m. So we can have:

```
y = b0 + b1*x1 + b2*x2 + ... + ... bn*xn
```

Up to any number n you can think of. As long as it's a bunch of x*b terms added together it's a linear equation. Don't get tripped up by the notation: b0, b1, and b2 are different numbers, and x1 and x2 are different columns of data. The notation just ensures you can always include as many variables in your model (just add another number) as you need to.

In our probability-of-touchdown model that I made up, x1 was yards to go, and x2 was yards to go squared. We had:

```
prob td = b0 + b1*(yards to go)+ b2*(yards to go ^ 2)
```

Let's try running this model in Python and see if we can get better values for b0, b1, and b2.

Remember: the first step in modeling is making a dataset where the columns are your input variables (yards to go, yards to go squared in this case) and one output variable (touchdown or not). Let's do it. The code for this section is in `07_01_ols.py`.

```
1  import pandas as pd
2  import statsmodels.formula.api as smf
3  from os import path
4
5  DATA_DIR = '/Users/nathan/ltcwff-files/data'
6
7  # load
8  df = pd.read_csv(path.join(DATA_DIR, 'play_data_sample.csv'))
9
10 # process
11 df = df.loc[(df['play_type'] == 'run') | (df['play_type'] == 'pass')]
12 df['offensive_td'] = (
13     (df['touchdown'] == 1) & (df['yards_gained'] > 0))
14 df['offensive_td'] = df['offensive_td'].astype(int)
15
16 df['yardline_100_sq'] = df['yardline_100'] ** 2
```

Besides loading our libraries and the data, this first section of the code also does some minor processing:

First, we've limited our analysis to regular offensive plays (line 11) — no punts or field goals.

Next — although there is a `touchdown` field in our data — it includes both offensive and defensive touchdowns (e.g. an interception returned for a touchdown). Lines 12-13 create a new version that's offensive touchdowns only.

This initial offensive touchdown variable is a column of booleans (`True` if the play resulted in an offensive TD, `False` otherwise). That's fine except our model can only operate on actual numbers, which means we need to convert this boolean column into its numeric equivalent.

The way to do this while making everything easy to internet is by transforming `df['offensive_td']` into a **dummy variable**. Like a column of booleans, a dummy variable only has two values. Instead of `True` and `False` it's just 1 and 0. Calling `astype(int)` on a boolean column will automatically do that conversion (line 13) [1].

In fact, including two perfectly correlated variables like this in your RHS variables breaks the model, and most statistical programs will drop the unnecessary variable automatically.

[1] Notice even though we have two outcomes — touchdown or not — we just have the one column. There'd be no benefit to including an extra column `'not_a_td'` because it'd be the complete opposite of td; it doesn't add any information.

Finally, our play by play data doesn't actually have yards to go squared built in, so we have to create it (line 16).

Now we have our data. In many ways, getting everything to this point is the whole reason we've learned Pandas, SQL, scraping data and everything else so far in this book. All for this:

```
In [1]:
df[['offensive_td', 'yardline_100', 'yardline_100_sq']].head()

Out[1]:
   offensive_td  yardline_100  yardline_100_sq
0                0          75.0           5625.0
1                0          63.0           3969.0
2                0          58.0           3364.0
3                0          47.0           2209.0
4                0          47.0           2209.0
```

Now we just need to step and pass it to our modeling function, which we get from the third party library `statsmodels`. We're using the `ols` function. OLS stands for Ordinary Least Squares, and is another term for basic, standard linear regression.

We have to tell `smf.ols` which output variable and which are the inputs, then run it. We do that in two steps like this:

```
In [3]: model = smf.ols(
    formula='offensive_td ~ yardline_100 + yardline_100_sq', data=df)

In [4]: results = model.fit()
```

Once we've done that, we can look at the results:

```
In [5]: results.summary2()
Out[5]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.083
Dependent Variable: touchdown        AIC:                29.3048
Date:               2019-07-15 15:34 BIC:                40.3351
No. Observations:   292              Log-Likelihood:     -11.652
Df Model:           2                F-statistic:        14.25
Df Residuals:       289              Prob (F-statistic): 1.26e-06
R-squared:          0.090            Scale:              0.064073
-----------------------------------------------------------------
                  Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-----------------------------------------------------------------
Intercept          0.2950   0.0438  6.7421 0.0000  0.2089  0.3811
yardline_100      -0.0110   0.0023 -4.7287 0.0000 -0.0156 -0.0064
yardline_100_sq    0.0001   0.0000  3.9885 0.0001  0.0001  0.0002
-----------------------------------------------------------------
Omnibus:              185.464      Durbin-Watson:         2.140
Prob(Omnibus):        0.000        Jarque-Bera (JB):      1008.120
Skew:                 2.797        Prob(JB):              0.000
Kurtosis:             10.180       Condition No.:         11073
=================================================================
* The condition number is large (1e+04). This might indicate
strong multicollinearity or other numerical problems.
"""
```

We get back a lot of info from running this regression. The part we're interested in — the values for `b0`, `b1`, `b2` — are under Coef (for coefficients). They're also available in `results.params`.

Remember the intercept is another word for `b0`. It's the value of `y` when all the data is `0`. In this case, we can interpret as the probability of scoring a touchdown when you're right on the edge — the 0th yard — of the goalline. The other coefficients are next to `yardline_100` and `yardline_100_sq`.

So instead of my made up formula from earlier, the formula that best fits this data is:

`0.2950 + -0.011*yards to go + 0.0001*(yards to go ^ 2)`.

Let's test it out with some values:

```
In [6]:
def prob_of_td(yds):
    b0, b1, b2 = results.params
    return (b0 + b1*yds + b2*(yds**2))

In [7]: prob_of_td(75)
Out[7]: 0.017520991820126786

In [8]: prob_of_td(25)
Out[8]: 0.10310069303712123

In [9]: prob_of_td(5)
Out[9]: 0.2695666142114951
```

Seems reasonable. Let's predict it for every value of our data.

```
In [10]: df['offensive_td_hat'] = results.predict(df)

In [11]: df[['offensive_td', 'offensive_td_hat']].head()
Out[11]:
   offensive_td  offensive_td_hat
0             0          0.017521
1             0         -0.005010
2             0         -0.006370
3             0          0.007263
4             0          0.007263
```

We can see the first five plays were not touchdowns, and our predictions were all pretty close to 0.

It's common in linear regression to predict a newly trained model on your input data. The convention is to write this variable with a ^ over it, which is why it's often suffixed with "hat".

The difference between the predicted/hat values and what actually happened is called the **residual**. The math of linear regression is beyond the scope of this book, but basically the computer is picking out b0, b1, b2's to make the **residuals**[2] as small as possible.

The proportion of variation in the output variable that your model "explains" — the rest of variation is in the residuals — is called R^2. It's always between 0-1. An R2 of 0 means your model explains nothing; an R2 of 1 means your model is perfect and your yhat always equals y — i.e. the residuals are 0.

Your model will almost always do better on data you used to make the model then other, external data. For example, we built this model on data from two high scoring games with good teams. Because of that, it very likely overestimates the probability for the typical games.

[2]Technically, OLS regression finds the coefficients that make the total sum of each squared residual (in part so they're all positive) as small as possible. In fact, "squares as small as possible" is partly the name, ordinary "least squares" regression.

Statistical Significance

If we look at the results from our regression, we can see we get back a bunch of columns in addition to the coefficient. All of these are getting at the same thing, the **significance** of a coefficient.

Statistical significance is bit tricky to explain, but it basically gets at: is this effect real? Or is just luck of the draw?

To wrap our heads around this we need to distinguish between two things: the "true", real relationship between variables (coefficient) — which we usually can't observe, and the observed/measured relationship, which we *can* see.

For example, consider flipping a coin (if this coin flipping example isn't football related enough related pretend it's a pre-game coin toss).

What if we wanted to run a regression:

prob of winning toss = model(whether you call heads)

Now, we *know* in this case (because of the way the world, fair coins, and probability work), whether you call heads or tails has no impact on your odds of winning the toss. That's the *true* relationship.

But data is noisy, and — if we actually flip a few coins — we probably will observe *some* relationship in our data.

Measures of statistical significance get at trying to help you tell whether the result you observe is "true" or just a result of random noise.

They do this by saying: (1) assume the *true* effect of this variable is that there is none, i.e. it doesn't effect the outcome. Then, assuming that true relationship is 0, (2) how often would we observe what we're seeing in the data?

Make sense? Let's actually run a regression on some fake data that does this.

First let's make the fake data. We'll "flip a coin" coin using Python's built-in random library. Then we'll guess the result (via another call to random) and make a dummy indicating whether we got it right or not. We'll do that 100 times.

This example is in `07_02_coinflip.py`.

```
import random
from pandas import DataFrame
import statsmodels.formula.api as smf

coin = ['H', 'T']

# make empty DataFrame
df = DataFrame(index=range(100))

# now fill it with a "guess" and a "flip"
df['guess'] = [random.choice(coin) for _ in range(100)]
df['result'] = [random.choice(coin) for _ in range(100)]

# did we get it right or not?
df['right'] = (df['guess'] == df['result']).astype(int)
```

Now let's run a regression on it:

```
model = smf.ols(formula='right ~ C(guess)', data=df)
results = model.fit()
results.summary2()
"""
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.006
Dependent Variable: right            AIC:                146.5307
Date:               2019-07-22 14:09 BIC:                151.7411
No. Observations:   100              Log-Likelihood:     -71.265
Df Model:           1                F-statistic:        1.603
Df Residuals:       98               Prob (F-statistic): 0.208
R-squared:          0.016            Scale:              0.24849
-----------------------------------------------------------------
                Coef.   Std.Err.     t     P>|t|    [0.025  0.975]
-----------------------------------------------------------------
Intercept       0.6170    0.0727   8.4859  0.0000   0.4727  0.7613
C(guess)[T.T]  -0.1265    0.0999  -1.2661  0.2085  -0.3247  0.0717
-----------------------------------------------------------------
Omnibus:              915.008       Durbin-Watson:          2.174
Prob(Omnibus):        0.000         Jarque-Bera (JB):       15.613
Skew:                 -0.196        Prob(JB):               0.000
Kurtosis:             1.104         Condition No.:          3
=================================================================

"""
```

Our results: according to this regression (since we're working with randomly generated data you'll get different results), guessing tails lowers your probability of correctly calling the flip by almost 0.13.

This is huge if true!

But, let's look at significance columns. The one to really pay attention to is P>|t|. It says: (1) start by assuming no true relationship between what you guess and probability of calling correctly, (2) *if* that were the case, we'd expect to see a relationship as "strong" as the result we got about 21% of the time.

So looks like we had a semi-unusual draw, i.e. 80th percentile, but nothing *that* crazy.

If you want to get a feel for how often we should expect to see a result like this, try running `random.randint(1, 10)` a couple times in the REPL and make note of how often you get higher than an 8.

Traditionally, the rule has been for statisticians and social scientists to call a result **significant** if the P value is less than .05, i.e. — if there were no true relationship — you'd only see those type of results 1/20 times.

But in recent years, P values have come under fire a bit.

Usually, there's a large incentive for people running regressions to want to see an interesting, significant result. Couple with the fact that there are many, many researchers and it's a problem. If we have 100 researchers running a regression on relationships that don't actually exist, and you'll get an average of five "significant" results (1/20) just by chance. Then those five analysts get published and paid attention to, even though the result they're describing is actually just noise.

The real situation is even worse that that though, because often even one person can run enough variations of a regression — adding in certain variables here, making different data assumptions there — to get an interesting and significant result.

But if you keep running regressions till you find something you like, the traditional interpretation of p value goes out the window. Your "statistically significant" effect may be a function of you running many models. Then, when someone comes along trying to replicate your study with new data, they find the relationship and result doesn't actually exist (i.e., it's not significant). This appears to have happened in quite a few scientific disciplines, and is what as known as the "replicability crisis".

There are a few ways to handle this. The best option would be to write out your regression before you run it, so you have to stick to it no matter what the results are. Some scientific journals are encouraging this by committing to publish based only on a "pre-registration" of the regression.

It also is a good idea — particularly if you're playing around with different regressions — to have much stricter standards than just 5% for what's significant or not.

Finally, it's also good to mentally come to grips with the fact that no effect or a statistically insignificant effect still might be an interesting result.

So, back to yardline and probability of scoring a touchdown. Yardline clearly has an effect. Looking at P>|t| it says:

1. Start by assuming no true relationship between yards to goal and probability of scoring a TD.
2. *If* that were the case, we'd see our observed results — where teams *do* seem to score more as they get closer to the goal — less than 1 in 100,000 times.

So either this was a major major fluke, or how close you are to goal line actually does effect probability you score.

Regressions hold things constant

One neat thing about the interpretation of any particular coefficient is it allows you to check the relationship between some input variable and your output holding everything else constant.

Let's go through another example: our play-by-play data comes with a variable called `wpa`, which stands for win probability added. It's a measure for how much each play added or subtracted to a teams probability of winning.

Now, win probability itself is modeled — but for now let's take it as given and assume it's accurate. It might be fun to run a regression on `wpa` to see how different kinds of plays impact a team's probability of winning.

Let's start with something simple. This example is in `07_03_ols2.py`.

```
df = pd.read_csv(path.join(DATA_DIR, 'play_data_sample.csv'))

model = smf.ols(formula=
        """
        wpa ~ offensive_td + turnover + first_down
        """, data=df)
results = model.fit()

print(results.summary2())
```

That gives us:

```
                  Results: Ordinary least squares
==============================================================================
Model:               OLS               Adj. R-squared:      0.612
Dependent Variable:  wpa               AIC:                 -942.2190
Date:                2019-08-11 20:23  BIC:                 -927.8402
No. Observations:    269               Log-Likelihood:      475.11
Df Model:            3                 F-statistic:         141.7
Df Residuals:        265               Prob (F-statistic):  8.74e-55
R-squared:           0.616             Scale:               0.0017375
------------------------------------------------------------------------------
                      Coef.  Std.Err.    t     P>|t|   [0.025  0.975]
------------------------------------------------------------------------------
Intercept            -0.0136   0.0033  -4.1272 0.0000 -0.0201 -0.0071
offensive_td[T.True]  0.1323   0.0101  13.0786 0.0000  0.1124  0.1522
turnover[T.True]     -0.1599   0.0136 -11.7704 0.0000 -0.1867 -0.1332
first_down[T.True]    0.0573   0.0057  10.0369 0.0000  0.0460  0.0685
------------------------------------------------------------------------------
Omnibus:               117.758       Durbin-Watson:          2.198
Prob(Omnibus):         0.000         Jarque-Bera (JB):       1492.616
Skew:                  1.383         Prob(JB):               0.000
Kurtosis:              14.203        Condition No.:          6
==============================================================================
```

Let's look at a few coefficients to practice reading them. According to this, the *benefit* to a team from scoring a touchdown (0.1300) is smaller in magnitude than the *loss* in win probability from turning the ball over (-0.1592). Both are statistically significant (by a lot).

Then look at the coefficient on a first down. It's fairly large — almost a 0.06 increase in probability.

If you think about it, there are two reasons plays gaining a first down are helpful: (1) the team gets a new set of downs, (2) plays that get first downs gain more yards than usual.

To verify the second claim we can use our groupby in Pandas:

```
In [1]: df.groupby('first_down')['yards_gained'].mean()
Out[1]:
first_down
False     3.670270
True     15.130952
```

First down plays go for an average of 15 yards, vs non-first down plays, which average under four. But what if we want to quantity *just* the impact of just a new set of downs?

The neat thing about regression is the interpretation of a coefficient — the effect of that variable — assumes all the other variables in the model are held constant.

So, in this first version we're saying, *controlling* (holding constant) for whether the play was touchdown or turnover, how much do plays that result in a first down add to WPA? In this version of the

model, we know first down plays go for more yards on average, but we're not explicitly controlling for it.

To do so, we can add yards gained to the model. Let's run it:

```
                  Results: Ordinary least squares
==================================================================
Model:                OLS              Adj. R-squared:     0.751
Dependent Variable:   wpa              AIC:                -1061.1756
Date:                 2019-08-11 20:23 BIC:                -1043.2021
No. Observations:     269              Log-Likelihood:     535.59
Df Model:             4                F-statistic:        203.4
Df Residuals:         264              Prob (F-statistic): 2.31e-79
R-squared:            0.755            Scale:              0.0011125
------------------------------------------------------------------
                     Coef.  Std.Err.    t     P>|t|   [0.025  0.975]
------------------------------------------------------------------
Intercept           -0.0205   0.0027  -7.6111 0.0000 -0.0258 -0.0152
offensive_td[T.True]  0.0825   0.0091   9.1079 0.0000  0.0647  0.1003
turnover[T.True]    -0.1400   0.0110 -12.7333 0.0000 -0.1616 -0.1183
first_down[T.True]    0.0224   0.0054   4.1577 0.0000  0.0118  0.0330
yards_gained          0.0028   0.0002  12.2430 0.0000  0.0023  0.0032
------------------------------------------------------------------
Omnibus:              71.033        Durbin-Watson:           2.217
Prob(Omnibus):        0.000         Jarque-Bera (JB):        2199.522
Skew:                 -0.006        Prob(JB):                0.000
Kurtosis:             17.009        Condition No.:           73
==================================================================
```

Now we're saying, controlling for whether the play was a TD, turnover, AND how many yards it gained, what is the effect of gaining a first down on win probability?

Now that yards is explicitly accounted for somewhere else, we know that the coefficient on `first_down` measures just the effect of new set of downs.

We can see the effect drops by more than half. It's still positive — getting a new set of downs adds a little over 0.02 to a teams win probability on average — but it does bring it down, which is what we expected.

Other examples of holding things constant

This idea of holding variables constant is useful and one of the major reasons people run regressions. It might be useful to run through some other hypothetical examples of where this would come up and be helpful.

For an example of the latter, say we wanted to do analyze whether the fantasy community properly evaluates players returning from IR injuries. In that case we might do something like:

```
ave points = b0 + b1*adp + b2*ended_last_season_on_ir
```

Where `ended_last_season_on_ir` is 1 if a player ended the previous season on IR, 0 otherwise.

If b2 is 0 or not significantly different from 0, that suggests — controlling for ADP — there's no systemic gain/loss from drafting a player who ended last season on IR.

If b2 *was* < 0, it's suggest players on IR do worse than their ADP would suggest - > 0, better. Depending on the results we could adjust our strategy accordingly.

You could do this analysis with any other data you had and wanted to test whether the crowd controlled for — rookies, players changing teams, etc.

I don't know this for sure, but I would guess any coefficient wouldn't be statistically significant from 0. That is, I would guess the crowd doesn't systematically under or over value guys in predictable ways.

Holding things equal is also useful for comparing players' performance in different situations. For example, you could:

- Measure RB performance holding the quality of offensive line constant.
- Measure WR performance holding the quality of the quarterback constant.
- Measure a fantasy players points scored holding the amount of garbage time they get constant.

Fixed Effects

We've seen how dummy variables works well for binary, true or false variables, but what about something with more than two categories?

Not just `ended_season_on_ir` or `offensive_td`, but say, position (QB, RB, WR, TE, K, DST) or down (1, 2, 3, 4).

These are called categorical variables or "fixed effects" and the way we handle them is by putting our one categorical variable (position) variable into a series of dummies that give us the same info, e.g.:

`is_qb`, `is_rb`, `is_wr`, `is_te`, `is_k`, `is_dst`

Where `is_qb` is 1 if player is a qb, 0 otherwise, etc. Except, again, we don't need ALL of these. There are only 6 fantasy positions, and a player has to be one — if a player isn't a QB, RB, WR, TE, or K, then we know it must be a DST. That means we can (by can I mean have to so the math will work) leave out one of the categories.

Fixed effects are very common in right hand side variables, and Pandas has built in functions to make them for you:

```
In [2]: pd.get_dummies(df['down']).head()
Out[2]:
   1.0  2.0  3.0  4.0
0    1    0    0    0
1    1    0    0    0
2    0    1    0    0
3    1    0    0    0
4    0    1    0    0
```

Again, ALL of these variables would be redundant — there are only four downs, so you can pass the `drop_first=True` argument to `get_dummies` to have it return only three columns.

In practice, programs like statsmodels can automatically convert categorical data to a set of fixed effects behind the scenes, which is why it's useful to know what it's doing.

Squaring Variables

When we run a linear regression, we're assuming certain things about how the world works, namely that change in one our x variables always means the *same* change in y.

Say for example we're looking at:

```
ave points = b0 + b1*adp
```

The fact we're modeling this as a linear relationship means we're assuming — whenever we change ADP by one unit — it has the same affect on final points. So this effect (b1) is the same whether we're going from pick 1 overall to pick 2, or whether pick 171 to 172.

Is that that a good assumption? In this case probably not, we'd expect the difference in ADP early to matter a lot more for final points than ADP later on.

So does this mean we have to abandon linear regression?

No, because there are tweaks we can make — while keeping things linear — that help make our model more realistic. All of these tweaks basically involve keeping the linear framework (y = b0 + x1*b1* + *x2*b2 … xn*bn), but doing some new things to our data (the x's) that allow us to deal with things like this.

In this case — where we think the relationship between ADP and ave points might change depending on exactly where we are in ADP — we could square ADP and include it in the model.

```
ave points = b0 + b1*ADP + b2*ADP^2
```

This allows the relationship to change depending where we're at on ADP. For early values of ADP, ADP^2 is relatively small, and the b2 term doesn't come into play as much — later it does.

Including squared (sometimes called quadratic) variables is common when you think the relationship between an input and output might depend where you are on the input.

Logging Variables

Another common transformation is to take the natural log of either our y variable, any of our x variables, or both. This let's you move from absolute to relative changes and interpret coefficients as percent changes.

So if we had a regression like:

```
rookie qb passing yards = b0 + b1*college passing yards
```

We'd interpret b1 as the number of yards passing yards associated with 1 more yard in college. If we did:

```
ln(passing yards)= b0 + b1*ln(college passing yards)
```

We'd interpret b1 as how a *percent change* in passing yards change as *percent change* in college yards. It works for dummy variables too, for example if you're doing:

```
ln(passing yards)= b0 + b1*ln(college passing yards)+ b2*is_under_6_ft
```

Then `b2` is the effect — as a percentage of passing yards — of a quarterback being under 6 ft.

Interactions

Again, in a normal linear regression, we're assuming the relationship between some x variable and our y variable is always the same for every type of observation.

So above, we ran a regression on win probability and found that a turnover reduced a teams probability of winning by about 0.16.

But that assumes the impact of a turnover is the same whenever it happens. Is that true? I could see turnovers being more costly in certain situations, e.g. in a close game or maybe towards the end of a game.

To test this, we can add in an **interaction** — which allow the effect of a variable to vary depending on the value of another variable.

In practice, it means our regression goes from this:

wpa = b0 + b1*offensive_td* + *b2*turnover + …

To this:

wpa = b0 + b1*offensive_td* + *b2*turnover + b3*(is_4th_q*turnover) + …

Then b2 is the impact of a normal, non-4th quarter turnover and b2 + b3 is the effect of a 4th quarter turnover.

Let's run that regression:

```
df['is_4'] = (df['qtr'] == 4)
df['turnover'] = df['turnover'].astype(int)

model = smf.ols(formula=
        """
        wpa ~ offensive_td + turnover + turnover:is_4 + yards_gained +
           first_down
        """, data=df)
results = model.fit()
results.summary2()
```

And the results:

```
                 Results: Ordinary least squares
=================================================================
Model:               OLS              Adj. R-squared:     0.762
Dependent Variable:  wpa              AIC:                -1072.4398
Date:                2019-08-11 20:33 BIC:                -1050.8715
No. Observations:    269              Log-Likelihood:     542.22
Df Model:            5                F-statistic:        173.0
Df Residuals:        263              Prob (F-statistic): 5.50e-81
R-squared:           0.767            Scale:              0.0010630
-----------------------------------------------------------------
                     Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-----------------------------------------------------------------
Intercept           -0.0205   0.0026 -7.7938 0.0000 -0.0257 -0.0154
offensive_td[T.True]  0.0824   0.0089  9.3014 0.0000  0.0649  0.0998
first_down[T.True]    0.0223   0.0053  4.2343 0.0000  0.0119  0.0326
turnover            -0.1153   0.0127 -9.0811 0.0000 -0.1403 -0.0903
turnover:is_4       -0.0820   0.0225 -3.6460 0.0003 -0.1263 -0.0377
yards_gained          0.0028   0.0002 12.5600 0.0000  0.0023  0.0032
-----------------------------------------------------------------
Omnibus:               78.617       Durbin-Watson:          2.139
Prob(Omnibus):         0.000        Jarque-Bera (JB):       1103.675
Skew:                  0.712        Prob(JB):               0.000
Kurtosis:              12.820       Condition No.:          162
=================================================================
```

So we can see a turnover in 4th quarter is indeed worse than a turnover in quarters 1-3. The latter lowers win probability by -0.12, the former by -0.12 + -0.08 = -0.20.

Logistic Regression

Earlier we went through an example where we built a model to calculate the probability of a touchdown given some value for yards (and yards squared).

That model worked OK — especially since it was built on data from only two games — but for some yardlines around the middle of the field it returned negative probability.

We can avoid this by running a **logistic** regression instead of a linear regression.

You can use all the same tricks (squared variables, interactions, dummy variables, fixed effects, and logged variables) on the right hand side, we're just working with a different, non-linear model.

$1/(1 + \exp(-(b0 + b1x1 + \ldots + bnxn)))$

It's just a one line change in statsmodels.

```
smf.logit(formula='touchdown ~ yardline_100 + yardline_100_sq', data=df)
results = model.fit()

In [16]: results.summary2()
Out[16]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                         Results: Logit
=================================================================
Model:              Logit            Pseudo R-squared: 0.196
Dependent Variable: offensive_td     AIC:              116.3669
Date:               2019-07-23 21:50 BIC:              127.1511
No. Observations:   269              Log-Likelihood:   -55.183
Df Model:           2                LL-Null:          -68.668
Df Residuals:       266              LLR p-value:      1.3926e-06
Converged:          1.0000           Scale:            1.0000
No. Iterations:     8.0000
-----------------------------------------------------------------
                  Coef.  Std.Err.    z    P>|z|   [0.025  0.975]
-----------------------------------------------------------------
Intercept         -0.2686   0.4678 -0.5743 0.5658 -1.1855  0.6482
yardline_100      -0.1212   0.0381 -3.1824 0.0015 -0.1958 -0.0465
yardline_100_sq    0.0010   0.0005  2.0822 0.0373  0.0001  0.0019
=================================================================
"""
```

Now to get the probability for any yardage, we need to do something similar to before, but then run it through the logistic function.

```
In [19]:
def prob_of_td_logit(yds):
    b0, b1, b2 = logit_results.params
    value = (b0 + b1*yds + b2*(yds**2))
    return 1/(1 + math.exp(-value))

In [20]: prob_of_td_logit(75)
Out[20]: 0.02008026676665306

In [21]: prob_of_td_logit(25)
Out[21]: 0.063537736346553

In [22]: prob_of_td_logit(5)
Out[22]: 0.2993805493398434
```

You should always use logit when you want a linear model and you're modeling some yes or no type outcome.

Random Forest

Both linear and logistic regression are examples of statistical models that are useful for analyzing the relationships between data (by looking at and saying things about the coefficients on data) as well as making predictions about new data (by getting values for a model, then running new data through it to see what it predicts for y).

The Random Forest algorithm, by contrast, is more of a black box. Its flexibility and looser assumptions about the structure of your data can mean for much more accurate predictions, but it's really not as useful for understanding relationships between your data. There are no coefficients or their equivalent to analyze, for example.

But it is flexible, accurate (there's a famous article — the unreasonable effectiveness of random forests) and pretty easy to run. And unlike linear or logistic regression, where your y variable has to be continuous or 0/1, Random Forests work well for classification problems.

Let's run through a classification problem: say we're trying to build a model to predict position based on everything else in our player-game data.

When we're done, we'll be able to input any players stats, height, weight, 40 time etc, and our model will make a guess at position.

Classification and Regression Trees

The foundation of Random Forest models is the classification and regression tree (CART). A CART is a single tree made up of a series of splits on some numeric variables. So, in our position example, maybe the first split in the tree is on "targets", where if a player averages less than 1 a game they go one way, more another. Players averaging less than one target a game might then go onto the "passing yards" — if it's above the split point they get classified as QBs, below Ks.

Details on how split points are selected exactly are beyond the scope of this book, but essentially the computer goes through all the variables/possible split points along each variable and picks the one that separates the data the "best". The final result is a bunch of if-then decision rules.

You can tell your program when to stop doing splits a few ways: (1) tell it to keep going until all observations in a branch are "pure" (i.e. all classified the same thing, (2) tell it to split only a certain number of times, (3) split until a branch reaches a certain number of samples.

Python seems to have sensible defaults for this, so I don't find myself changing them too often.

Once you stop, the end result is a tree where the endpoints (the leaves) are one of your output classifications, or — for a regression — the mean of your output variable for all the observations in the group.

Regardless you have a tree, and you can follow it through till the end and get some prediction.

Random Forests are a Bunch of Trees

That's one CART tree. The Random Forest algorithm consists of multiple CARTs combined together for a sort of wisdom-of-crowds approach. The default in Python is to take 100 trees, but this is a parameter you have control over.

Each CART is trained on a subset of your data. This subsetting happens in two ways: using a random sample of observations (rows), but also by limiting each CART to a random subset of *columns*. This helps makes sure the trees are different from each other, and provides the best results overall.

So the final model is stored as some number of binary trees, each trained on a different, random sample of your data with a random selection of available features.

Getting a prediction depends on whether your output variable is categorical (classification) or continuous (regression).

When it's a classification problem, you just run it through each of the trees (say 100), and see what the most common outcome is.

So you might might have QB = 80% of the time, RB = 15%, K 5% or something.

For a regression, you run it through the 100 trees, then take the average of what each of them says.

In general, a bunch of if … then tree rules make this model way more flexible than something like a linear regression, which imposes the linear structure. This is nice for accuracy, but it also means random forests are much more susceptible to things like overfitting, and it's a good idea to set aside some data to evaluate how well your model does.

Random Forest Example in Scikit-Learn

Let's go through an example. Here, we'll use our player-game data and try to predict position given all of our other numeric data. That is, we'll model position as a function of:

```
xvars = ['carries', 'rush_yards', 'rush_fumbles', 'rush_tds', 'targets',
         'receptions', 'rec_yards', 'raw_yac', 'rec_fumbles', 'rec_tds',
         'ac_tds', 'rec_raw_airyards', 'caught_airyards', 'attempts',
         'completions', 'pass_yards', 'pass_raw_airyards', 'comp_airyards'
            ,
         'timeshit', 'interceptions', 'pass_tds', 'air_tds']
yvar = 'pos'
```

This example is in `07_05_random_forest.py`. First let's import our libraries and load our player-game data.

```
import pandas as pd
from pandas import DataFrame
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data/output'

df = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'))

train, test = train_test_split(df, test_size=0.20)
```

We can see the random forest tools are in the SciKit Learn library.

Because tree based models like Random Forest are so flexible, it's basically meaningless to evaluate them on the same data you built the model — they'll perform too well. Instead, it's good practice to take a **holdout** set, i.e. some portion of the data where you KNOW the outcome you're trying to predict (position) that you set aside to run your model on after it's built and see how it does. Scikit-learn's `train_test_split` method automatically does that. Here we have it using a random 80% of our data to build the model; the remaining 20% is the test data.

Running the model takes place on two lines. Note the `n_estimators` option. That's the number of

different trees the algorithm will run. Scikit learn defaults to 10, but that really isn't very many and they're changing that to 100 in an upcoming release. We'll just use 100 now.

```
model = RandomForestClassifier(n_estimators=100)
model.fit(train[xvars], train[yvar])
```

Note how the `fit` function takes your input and output variable as separate arguments. Unlike statsmodels, scikit-learn doesn't give us any fancy, pre-packaged results string to look at. But we can check to see how this model does on our holdout, test dataset.

```
In [2]: test['pos_hat'] = model.predict(test[xvars])

In [3]: test['correct'] = (test['pos_hat'] == test['pos'])

In [4]: test['correct'].mean()
Out[4]: 0.7630662020905923
```

Above 75%, not bad. Note, when you run this yourself, you'll get something different. Remember, a Random Forest model contains randomness (that's why it's named what it is), so you'll get a different answer every time you run it.

Another thing it's interesting to look at how confident the model is about each prediction. Remember, this model ran 100 different trees. Each classified every observation into one of: QB, RB, WR, TE. If the model assigned some observation RB for 51/100 trees and WR for the other 49/100, we can interpret it as relatively unsure in it's prediction.

We can interpret these frequencies as rough probabilities and calculate them with:

```
In [5]: model.predict_proba(test[xvars])
Out[5]:
array([[0.  , 0.  , 0.54, 0.46],
       [0.  , 0.  , 0.63, 0.37],
       [0.  , 0.03, 0.36, 0.61],
       ...,
       [0.  , 1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  , 0.  ],
       [0.  , 0.67, 0.22, 0.11]])
```

But this is just a raw, unformatted matrix. Let's put it into a DataFrame with the same index as `test`:

```
In [6]:
probs = DataFrame(model.predict_proba(test[xvars]),
                  index=test.index,
                  columns=model.classes_)
--

In [7]: probs.head()
Out[7]:
       QB    RB    TE    WR
936   0.0  0.00  0.54  0.46
440   0.0  0.00  0.63  0.37
1100  0.0  0.03  0.36  0.61
1210  0.0  0.00  0.04  0.96
902   1.0  0.00  0.00  0.00
```

So we can see the model says the first observation has a 54% of being a TE, and a 46% of being a WR.

Let's add these into the actual results from our test dataset.

```
In [8]:
results = pd.concat([
    test[['player_id', 'player_name', 'pos', 'pos_hat', 'correct']],
    probs], axis=1)
```

We can look at this to see how our model did for different positions.

```
In [9]: results.groupby('pos')[['correct', 'QB', 'RB', 'WR', 'TE']].mean()
Out[9]:
      correct        QB        RB        WR        TE
pos
QB   1.000000  0.987647  0.011765  0.000588  0.000000
RB   0.947368  0.000263  0.911711  0.052667  0.035360
TE   0.189655  0.000000  0.056724  0.619003  0.324273
WR   0.857143  0.000756  0.058319  0.735185  0.205739
```

We can see the model performs worst on TEs, getting them correct only 19% of the time. As we might expect, it usually misclassifies them as a WR. The good news is that our probability of being a TE was the highest for TEs (0.32) but it's still difficult to identify them.

Working with a holdout dataset let's us do interesting things like check how exactly the model performed on different positions and is easy conceptually to understand, but — when it comes simply evaluating how well a model performs, a technique called **cross validation** is probably more common.

To do cross validation, you divide your data into some number of groups, say 10. Then, you run your model 10 separate times, each time using 1 of the groups as the test data, and the other 9 to train it. That gives you 10 different accuracy numbers, which you can average to get a better look at overall

performance. It has the advantage of (1) being less noisy (because you're running multiple separate models instead of just one training/test set) and (2) getting more out of your data.

To run it, you create a new model as before, but this time — instead of calling fit on it — you pass it to the sklearn function `cross_val_score`.

```
model = RandomForestClassifier(n_estimators=100)
scores = cross_val_score(model, df[xvars], df[yvar], cv=10)
```

With `cv=10`, we're telling scikit learn to do divide our data into 10 groups. Also note how we're using the entire dataset now (`df`) instead of just our 80% training set. This gives back 10 separate scores, which we can look at and take the mean over.

```
In [4]: scores
Out[4]:
array([0.8137931 , 0.84137931, 0.8137931 , 0.85416667, 0.77777778,
       0.8041958 , 0.74647887, 0.77304965, 0.73049645, 0.78014184])

In [5]: scores.mean()
Out[5]: 0.7935272582383476
```

Again, your results will vary a bit, both due to the randomness of the Random Forest models and the randomness in how scikit learn split the data when the cross validation procedure.

Finally, it might be interesting to see how important our x variables were in classifying position. We can do that with:

```
In [13]: Series(model.feature_importances_, xvars).sort_values(ascending=
    False)
Out[13]:
rec_raw_airyards     0.170337
carries              0.158026
caught_airyards      0.126099
rush_yards           0.106618
rec_yards            0.071190
raw_yac              0.061879
targets              0.053837
receptions           0.037801
completions          0.037200
comp_airyards        0.036965
attempts             0.035756
pass_yards           0.033988
pass_raw_airyards    0.019725
rec_tds              0.012279
pass_tds             0.010109
rush_tds             0.009317
ac_tds               0.005330
air_tds              0.004494
rec_fumbles          0.004319
timeshit             0.002719
interceptions        0.001030
rush_fumbles         0.000981
```

There you go, you've run a Random Forest model. Regression in Random Forest works similarly, you just import RandomForestRegressor instead of RandomForestClassifier and you're all set.

While RandomForestClassifier is the only thing we've covered that will work for modeling an output variable with more than two categories, if you're modeling a continuous valued variable (like fantasy points) you might be wondering when to use RandomForestRegressor vs the OLS Linear Regression we covered first.

Generally, if you're interested in the coefficients and understanding and interpreting your model, you should lean towards the classic linear regression. If you just want the model to be as accurate as possible and don't care about understanding how it works, try Random Forest[3].

[3] Of course, there are other, more advanced models than Random Forest available in scikit-learn too. See the documentation for more.

End of Chapter Exercises

7.1

This problem builds off `07_01_ols.py` and assumes you have it open and run in the REPL.

(a) Using `prob_of_td` and the apply function, create a new column in the data `offensive_td_hat_alt` — how does it compare to `results.predict(df)`?

(b) Try adding distance to first down to the model, is it significant at the 5% level?

In our statistical significance coin flipping example, we had a variable 'guess' that was either 'H' or 'T'. We put it in our coin flipping regression with C(guess).

c) Add C(down) to the probability of touchdown model, and look the results, what does C() do? Controlling for everything else in our regression, which down is most likely to lead to a touchdown?

d) Run the same model without the C(down) syntax, creating dummy variables for down manually instead, do you get the same thing?

7.2

This problem builds off `07_02_coinflip.py` and assumes you have it open and run in the REPL.

(a) Build a function `run_sim_get_pvalue` that flips a coin n times (default to 100), runs a regression on it, and returns the p value of your guess.

Hint: the p values are available in `results.pvalues`.

b) Run your function at least 1k times and put the results in a `Series`, what's the average p value? About what do you think it'd be if you ran it a million times?

(c) The function below will run your `run_sim_get_pvalue` simulation from

(d) until it gets a significant result, then return the number of simulations it took.

```
def runs_till_threshold(i, p=0.05):
    pvalue = run_sim_get_pvalue()
    if pvalue < p:
        return i
    else:
        return runs_till_threshold(i+1, p)
```

You run it a single time like this: `runs_till_threshold(1)`.

Run it 100 or so times and put the results in a Series.

(d) The probability distribution for what we're simulating ("how many times will it take until an event with probability p happens?") is a called the Geometric distribution, look up the median and mean of it and compare it to your results.

7.3

This problem builds off `07_03_ols2.py` and assumes you have it open and run in the REPL.

(a) Using the play by play data, run the regression:

```
wpa ~ offensive_td + interception + yards_gained + fumble
```

What's worse for team's win probability, fumbling or throwing a pick? Why do you think this is?

(b) Replace `fumble` with `fumble_lost` instead. How do you think the coefficients on `fumble_lost` and `interception` will compare now?

7.4

Explain why your prediction about the sign and significance of `b2` might depend on your opinion on the aggregate smarts and skills of your fellow fantasy players:

```
ave points = b0 + b1*adp + b2*is_rookie
```

7.5

This problem builds off of `07_05_random_forest.py` and assumes you have it open and run in the REPL.

Right now, in `07_05_random_forest.py` we're predicting position for each player-game combination. But really, we're interesting in position at the *player* level.

Try grouping the player-game data to the player-season level rerunning the random forest model. Use cross-validation to see how the models perform.

(a) First try using the season average of each stat for every player.

(b) Then try other (or multiple) aggregations. Use cross validation to figure out which model performs the best.

8. Intermediate Coding and Next Steps: High Level Strategies

If you've made it this far (or really even just through the Pandas chapter) you should have all the technical skills you need to start working on your own projects.

That's not to say you won't continue to learn (the opposite!), just that moving beyond the basics and into intermediate programming is less "things you can do with DataFrames #6-10" and more mindset and high level strategies.

That's why, to wrap up, I wanted to cover a few, mostly non-technical strategies that I've found useful.

These concepts are both high (e.g. Gall's Law) and low (get your code working at the top level then put it in a function) level, but all of them should help as you move beyond the self-contained examples in this book and into your own projects.

Gall's Law

> *"A complex system that works is invariably found to have evolved from a simple system that worked."* - John Gall

Perhaps the most important idea to keep in as you move beyond small, self-contained examples while learning and into doing your own larger projects is *Gall's Law*.

Applied to programming, is says: any complicated, working program or piece of code (and most programs that do real work are complicated) evolved from some simpler, working code.

You may look at the final source code of analysis projects or even some of the extend examples in the book and think "there's so way I could ever do that."

But if I just sat down and tried to write these in their final form off the top of my head, I couldn't either.

The key is building up to it, starting with simple things that work (even if they're not exactly what you want), and going from there.

I sometimes imagine writing a program as tunnelling through a giant wall of rock. When starting, your job is to get a tiny hole through to the other side, even if just lets in a small glimmer of light. Once it's there, you can enlarge and expand it

Also: Gall's law says that complex systems evolve from simpler ones, but what's "simple" and "complex" might change depending on where you're at as a programmer.

If you're just starting out, writing some code to concatenate or merge two DataFrame together might be complex enough that you'll want to examine your your outputs when you're done to sure everything's working as expected.

As you get more experience and practice (a big goal of this book!) everything will seem easier, your intuition and first attempts will gradually get better, and your initial working "simple" systems will get more complicated.

Get Quick Feedback

Another related idea that will help you move faster: get quick feedback.

When writing code, you want to do it in small pieces that you run and test as soon as you can.

That's why I recommend coding in Spyder with your editor on the left and your REPL (read eval print loop) on the right, as well as getting comfortable with the short cut keys to quickly move between them.

This is important because you'll inevitably (and often) screw up, mistyping variable names, passing incorrect function arguments, etc. Running code as your write it helps you spot and fix these as they come up.

Here's a question — say you need to write some small, self contained piece of code; something you've done before that is definitely in your wheelhouse — what are the chances it does what you want without any errors the first time you try it?

For me, if it's anything over three lines, it's maybe 50-50 at *best*. Less if it's something I haven't done in a while.

Coding is precise, and it's really easy to mess things up in some minor way. If you're not constantly testing and running what you're writing, it's going to be really painful to figure out what's going on when you actually do run it.

Use Functions

In practice, for me the advice above (start simple + get quick feedback) usually means starting out by writing simple, working code in the "top level" (the main, regular Python file — vs inside a function).

Then — after I've examined the outputs in the REPL and am confident some particular piece is working — I'll usually put it inside some function.

Using functions has two benefits: (1) DRY and (2) letting you set aside and abstract parts of your code so you don't have to worry about as much stuff.

DRY: Don't Repeat Yourself

One of the maxims and sayings among programmers is "DRY" for Don't Repeat Yourself [1].

For example: maybe you need to write some similar code (code that summarizes points by position, say) a bunch of times (across RB, WR, TE etc).

The really naive approach might be to just get it working for one position, then copy and paste it a bunch of times and make the necessary tweaks for the others.

But what happens if you need to modify it, either because you change something or find a mistake?

Well if it's a bunch of copy and pasted code, you need to change it everywhere, which is tedious (best case), error-prone (worse) or both (worst).

Instead, put the similar code in a function, with arguments to allow for slight differences among use cases, and you only have worry about fixing it in one spot when you need to make inevitable changes.

Functions Help You Think Less

The other benefit of functions is they let you group related concepts and ideas together. This is nice because it gives you fewer things to think about.

For example, say we're working with some function called `win_prob` that takes information about the game — the score, who has the ball, how much time is left — and uses that to calculate each team's probability of winning.

Putting that logic in a function like `win_prob` means we no longer have to think about how we're calculating win probability every time we want to do it. We can just use our function instead.

[1] DRY comes from a famous (but old) book called The Pragmatic Programmer, which is good, but first came out in 1999 and is a bit out of date technically. It also takes a bit of a different (object oriented) approach that we do here

The flip side is also true. Once it's in a function, we no longer have to mentally process a bunch of Pandas code (what's that doing… multiplying time left by a number … adding it to the difference between team scores … … oh, that's right — win prob!) when reading through our program.

This is another reason it's usually better to tend towards using small functions that have one-ish job vs large functions that do everything and are harder to think about.

Attitude

As you move into larger projects and "real" work, coding will go much better for you adopt a certain mindset.

First, it's helpful to take a sort of "pride" (pride isn't exactly the right word but it's close) in your code.

You should appreciate and care about well designed, functioning code and strive to write it yourself. The guys who coined DRY talk about this as a sense of *craftsmanship*.

Of course, your standards for whats well-designed will change over time, but that's OK.

Second, you should be continuously growing and improving. You'll be able to do more faster if you deliberately try to get better as a programmer — experimenting, learning and pushing yourself to write better code — especially early on.

One good sign you're doing this well is if you can go back to code you've written in the past and are able to tell approximately when you wrote it.

For example, say we want to modify this dictionary:

```
roster_dict = {'qb': 'Tom Brady', 'rb': 'Dalvin Cook', 'wr': 'DJ Chark'}
```

So that all the player names are in all uppercase.

When we're first starting out, maybe we do something like:

```
In [5]: roster_dict1 = {}
In [6]: for pos in roster_dict:
            roster_dict1[pos] = roster_dict[pos].upper()

In [7]: roster_dict1
Out[7]: {'qb': 'TOM BRADY', 'rb': 'DALVIN COOK', 'wr': 'DJ CHARK'}
```

Then we learn about comprehensions and realize we could just do this on one line.

```
In [8]: roster_dict2 = {pos: roster_dict[pos].upper() for pos in
   roster_dict}

In [9]: roster_dict2
Out[9]: {'qb': 'TOM BRADY', 'rb': 'DALVIN COOK', 'wr': 'DJ CHARK'}
```

Then later we learn about `.items()` in dictionary comprehensions and realize we can write the same thing:

```
In [10]: roster_dict3 = {pos: name.upper() for pos, name in roster_dict.
   items()}

In [11]: roster_dict3
Out[11]: {'qb': 'TOM BRADY', 'rb': 'DALVIN COOK', 'wr': 'DJ CHARK'}
```

This example illustrates a few points:

First, if you go back and look at some code with `roster_dict2`, you can roughly remember, "oh I must have written this after getting the hang of comprehensions but before I started using `.items`." You definitely don't need to memorize your complete coding journey and remember when exactly when you started doing what, but noticing things like this once in a while can be a sign you're learning and getting better, and is good.

Second, does adding `.items` matter much for the functionality of the code in this case? Probably not.

But preferring the `.items` in v3 to the regular comprehension in v2 is an example of what I mean about taking pride in your code and wanting it to be cleaned and well designed.

Over time these things will accumulate and eventually you'll be able to do things that people who don't care what their code looks like and "just want it to work" can't.

Finally, taking pride in your code doesn't always mean you have to use the fanciest techniques. Maybe you think the code is easier to understand and reason about without `.items`. That's fine.

The point is you should be consciously thinking about these types of decisions and be deliberate (i.e. have reasons for what you do). And what you do should be changing over time as you learn and get better.

I think most programmers have this mindset to some degree, and if you're reading this, you probably do too. I encourage you to embrace it.

Review

Combined with the fundamentals, this high and medium level advice:

- Start with a working simple program then make more complex.
- Get quick feedback about what you're coding by running it.
- Care about the design your code and keep trying to get better at.
- Don't repeat yourself and think more clearly by putting common code in functions.

Will get you a long way.

9. Conclusion

Congratulations! If you've made it this far you are well on your way to doing data analysis. We've covered a lot of material, and there many ways you could go from here.

I'd recommend starting on your own analysis ASAP (see the appendix for a few ideas on places to look for data). Especially if you're self-taught, diving in and working on your own projects is by far the fastest and most fun way to learn.

When I started building the website that became www.fantasymath.com I knew nothing about Python, Pandas, SQL, web scraping, or machine learning. I learned all of it because I wanted to beat my friends in fantasy football.

The goal of this book has been to make things easier for people who feel the same way. Judging by the response so far, there are a lot of you. I hope I've been successful, but if you have any questions, errata, or other feedback, don't hesitate to get in touch — nate@nathanbraun.com

Appendix A: Places to Get Data

I thought it'd be helpful to list a few places to get fantasy related data.

Ready-made Datasets

RScraper

The play-by-play and player-game data we use in this book is a modified subset of the data now available via:

https://www.nflfastr.com

`nflfastR` is an R package written by Sebastian Carl and Ben Baldwin that scrapes play by play data.

Note on R

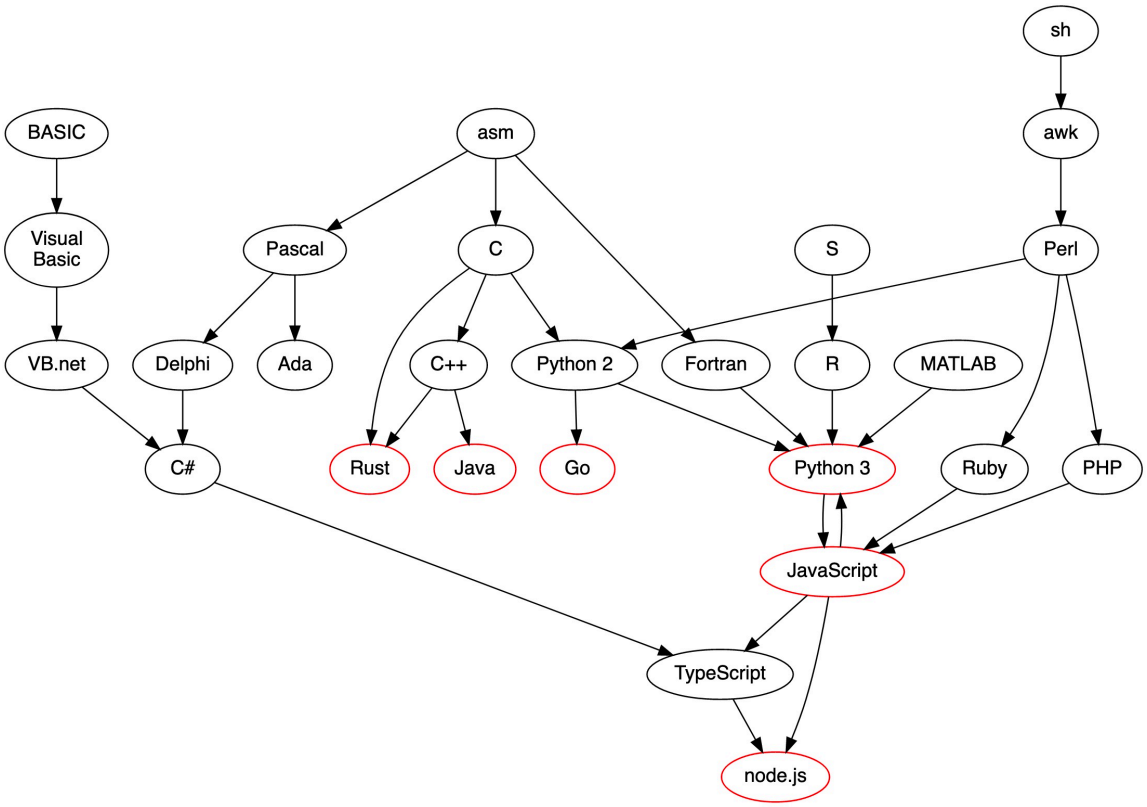
R is a programming language and an alternative to Python. You can basically think of it as Pandas + a few other statistical libraries. It's been around longer and has a solid NFL analytics following, as evidence by the guys from `nflfastR`, among others.

If this gives you a case of FOLTWPR (fear of learning the wrong programming language), relax. Until about 5 years ago the R/Python split was about 50/50, but since then Python is the much more popular option, as a quick search for "R vs Python market share" shows.

Also interesting is this study of programmer migration patterns by programmer blogger apenwarr. He found that R users are much more likely to jump to learning Python next than the other way around.

As apenwarr explains, nodes in red below are "currently the most common 'terminal nodes' — where people stop because they can't find anything better".

Python 3 (what we've been learning) is a terminal node. It's not only R users that tend to flow to it, but MATLAB and Fortran (and in my case Stata and SAS) users too.



Programmer migration patterns (apenwarr)

Figure 0.1: Programmer Migration Pattersn

Anyway, just because we've decided to learn Python instead of R doesn't mean we can't recognize good work and useful tools when we see them.

In this case it's even easier (we don't even have to do anything in R) because the authors have helpfully put up all their historical data (as zipped csv files, among others) here:

https://github.com/guga31bb/nflfastR-data

See the `data`, `legacy-data` and `roster-data` directories.

Google Dataset Search

Google has a dataset search engine with some interesting datasets:

https://toolbox.google.com/datasetsearch

Kaggle.com

Kaggle.com is best known for its modeling and machine learning competitions, but it also has a dataset search engine with some football related datasets.

Data Available via Public APIs

myfantasyleague.com

My Fantasy League is one of the oldest league hosting websites, and has always been very developer friendly. They have a public API available:

http://home.myfantasyleague.com/features/developers-api/

The nice thing about MFL is the amount detail they have available. A few years ago, for example, I used the API to get detailed draft data for more than 6,000 leagues to build a draft pick trade value chart.

fantasyfootballcalculator.com

We scraped and connected to in chapter 5, Fantasy Football Calculator, is one of the most popular places for mock drafts. It might be one of the only sites where people are doing mock drafts in March. As a result, their ADP is generally pretty good and up to date when it comes to late breaking, preseason news.

https://fantasyfootballcalculator.com/resources/average-draft-position-api/

sportsdata.io

It isn't free, but sportsdata.io, the company that provides data to some of the big name sites, has a nice hobbyist focused API with a ton of data available.

Appendix B: Anki

Remembering What You Learn

A problem with reading technical books is remembering everything you read. To help with that, this book comes with more than 300 flashcards covering the material. These cards are designed for **Anki**, a (mostly) free, open source *spaced repetition* flashcard program.

> "The single biggest change that Anki brings about is that it means memory is no longer a haphazard event, to be left to chance. Rather, it guarantees I will remember something, with minimal effort. That is, Anki makes memory a choice." — Michael Nielsen

With normal flashcards, you have to decide when and how often to review them. When you use Anki, it takes care of this for you.

Take a card that comes with this book, "What does REPL stand for?" Initially, you'll see it often — daily, or even more frequently. Each time you do, you tell Anki whether or not you remembered the answer. If you got it right ("Read Eval Print Loop") Anki will wait longer before showing it to you again — 3, 5, 15 days, then weeks, then months, years etc. If you get a question wrong, Anki will show it to you sooner.

By gradually working towards longer and longer intervals, Anki makes it straightforward to remember things long term. I'm at the point where I'll go for a year or longer in between seeing some Anki cards. To me, a few moments every few months or years is a reasonable trade off in return for the ability to remember something indefinitely.

Remembering things with Anki is not costless — the process of learning and processing information and turning that into your own Anki cards takes time (though you don't have to worry about making cards on this material since I've created them for you) — and so does actually going through Anki cards for a few minutes every day.

Also, Anki is a tool for *remembering*, not for learning. Trying to "Ankify" something you don't understand is a waste of time. Therefore, I strongly recommend you read the book and go through the code *first*, then start using Anki after that. To make the process easier, I've divided the Anki cards into "decks" corresponding with the major sections of this book. Once you read and understand the material in a chapter, you can add the deck to Anki to make sure you'll remember it.

Anki is optional — all the material in the cards is also in the book — but I strongly recommend at least trying it out.

If you're on the fence, here's a good essay by YCombinator's Michael Nielson for inspiration:

http://augmentingcognition.com/ltm.html

Like Nielsen, I personally have hundreds of Anki cards covering anything I want to remember long term — programming languages (including some on Python and Pandas), machine learning concepts, book notes, optimal blackjack strategy, etc.

Anki should be useful to everyone reading this book — after all, you bought this book because you want to remember it — but it'll be particularly helpful for readers who don't have the opportunity to program in Python or Pandas regularly. When I learned how to code, I found it didn't necessarily "stick" until I was able to do it often — first as part of a sports related side project, then at my day job. I still think working on your own project is a great way to learn, but not everyone is able to do this immediately. Anki will help.

Now let's get into installation.

Installing Anki

Anki is available as desktop and mobile software. I almost always use the desktop software for making cards, and the mobile client for reviewing them.

You can download the desktop client here:

https://apps.ankiweb.net/

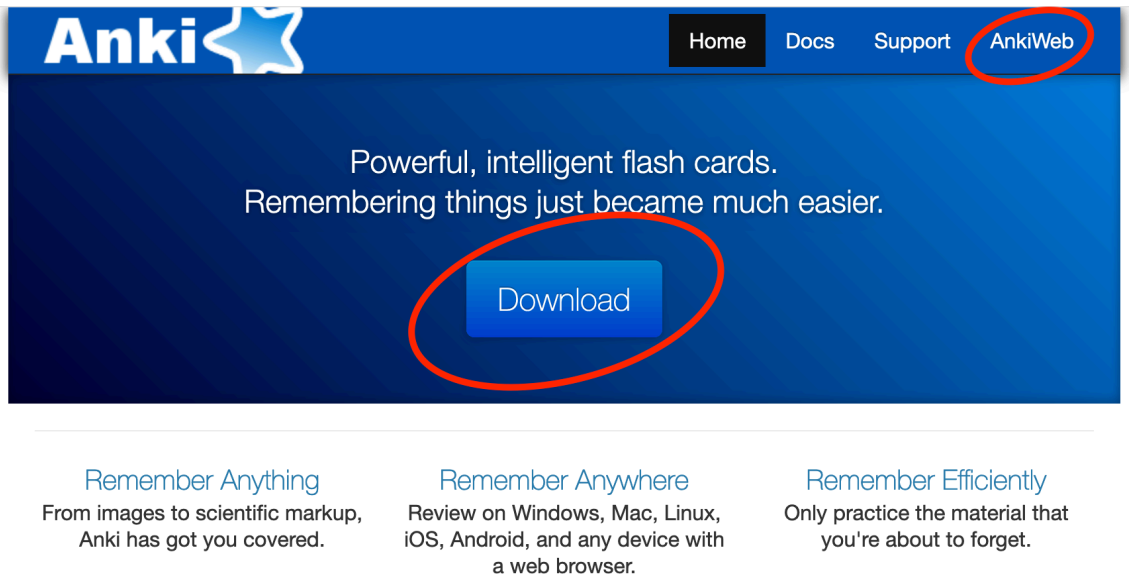


Figure 0.1: Anki Website

You should also make a free AnkiWeb account (in the upper right hand corner) and login with it on the desktop version so that you can save your progress and sync it with the mobile app.

Then install the mobile app. I use AnkiDroid, which is free and works well, but is only for Android.

The official IPhone version costs $25. It would be well worth it to me personally and goes towards supporting the creator of Anki, but if you don't want to pay for it you can either use the computer version or go to https://ankiweb.net on your phone's browser and review your flash cards there. I've also included text versions of the cards if you want to use another flashcard program.

Once you have the mobile app installed, go to settings and set it up to sync with your AnkiWeb account.

Using Anki with this Book

Anki has a ton of settings, which can be a bit overwhelming. You can ignore nearly all of them to start. By default, Anki makes one giant deck (called Default), which you can just add everything to. This is how I use it and it's worked well for me.

Once you've read and understand a section of the book and want to add the Anki cards for it, open up the desktop version of Anki, go to File -> Import… and then find the name of the apkg file (included with the book) you're importing.

For instance, you're nearly done with this Tooling section of the book, so you can import 00_tooling.apkg.

Importing automatically add them to your Default deck. Once they're added, you'll see some cards under New. If you click on the name of your deck 'Default' and then the 'Study Now' button, you can get started.

You'll see a question come up — "What does REPL stand for?" — and mentally recite, "Read Eval Print Loop".

Then click 'Show Answer'. If you got the question right click 'Good', if not click 'Again'. If it was really easy and you don't want to see it for a few days, press 'Easy'. How soon you'll see this question again depends on whether you got it right. Like I said, I usually review cards on my phone, but you can do it wherever works for you.

As you progress through sections of this book, you can keep adding more cards. By default, they'll get added to this main deck.

If you like Anki and want to apply it elsewhere, you can add other cards too. If you want to edit or make changes on the fly to any of the cards included here, that's encouraged too.

Appendix C: Answers to End of Chapter Exercises

All of the 2-7 chapter solutions are also available as (working) Python files in the `./solutions-to-exercises` directory of the files that came with this book.

1. Introduction

1.1

a) team-quarter (and game)
b) player-game
c) player-season
d) player-season with games as columns
e) position

1.2

This is somewhat subjective, but:

- 40 time
- longest (touchdown) reception
- number of touchdowns
- total or average air yards

might work

1.3

a) Wind speed and temperature at kickoff.
b) Total combined points for both teams.
c) This model is at the game level.

d) The main limitation has to be it includes no information about the teams playing (e.g. how good their offenses are, how many points they typically score or let up). The over under for KC-NO is going to be very different from MIA-CHI.

1.4

a) manipulating data
b) analyzing data
c) manipulating data
d) loading data
e) collecting data
f) analyzing data
g) collecting data
h) usually manipulating your data, though sometimes loading or analyzing too
i) analyzing data
j) loading or manipulating data

2. Python

2.1

a) Valid. Python programmers often start variables with _ if they're throwaway or temporary, short term variables.
b) Valid.
c) Not valid. Can't start with a number.
d) Valid, though convention is to split words with _, not camelCase.
e) `rb1_name`. Valid. Numbers OK as long as they're not in the first spot
f) `flex spot name`. Not valid. No spaces
g) `@home_or_away`. Not valid. Only non alphanumeric character allowed is _
h) `'pts_per_rec_yd'`. Not valid. A string (wrapped in quotes), not a variable name. Again, only non alphanumeric character allowed is _

2.2

```
In [3]:
weekly_points = 100
weekly_points = weekly_points + 28
weekly_points = weekly_points + 5

In [4]: weekly_points # 133
Out[4]: 133
```

2.3

```
In [5]:
def for_the_td(player1, player2):
    return f'{player1} to {player2} for the td!'

In [6]: for_the_td('Dak', 'Zeke')
Out[6]: 'Dak to Zeke for the td!'
```

2.4

It's a string method, so what might `islower()` in the context of a string? How about whether or not the string is lowercase.

A function "is *something*" usually returns a yes or no answer (is it something or not), which would mean it returns a boolean.

We can test it like:

```
In [6]: 'tom brady'.islower()  # should return True
Out[5]: True

In [7]: 'Tom Brady'.islower()  # should return False
Out[7]: False
```

2.5

```
In [7]
def is_leveon(player):
    return player.replace("'", '').lower() == 'leveon bell'

In [8]: is_leveon('tom brady')
Out[8]: False

In [9]: is_leveon("Le'Veon Bell")
Out[9]: True

In [10]: is_leveon("LEVEON BELL")
Out[10]: True
```

2.6

```
In [11]
def commentary(score):
    if score >= 100:
        return f'{score} is a good score'
    else:
        return f"{score}'s not that good"

In [14]: commentary(90)
Out[14]: "90's not that good"

In [15]: commentary(200)
Out[15]: '200 is a good score'
```

2.7

Here's what I came up with. The last two use list comprehensions.

```
giants_roster[0:3]
giants_roster[:3]
giants_roster[:-1]
[x for x in giants_roster if x != 'OBJ']
[x for x in giants_roster if x in ['Daniel Jones', 'Saquon Barkley',
                                   'Evan Engram']]
```

2.8a

```
In [26]: league_settings['number_of_teams'] = 10

In [27]: league_settings
Out[27]: {'number_of_teams': 10, 'ppr': True}
```

2.8b

```
In [29]:
def toggle_ppr(settings):
    settings['ppr'] = not settings['ppr']
    return settings

In [30]: league_settings
Out[30]: {'number_of_teams': 10, 'ppr': False}

In [31]: toggle_ppr(league_settings)
Out[31]: {'number_of_teams': 10, 'ppr': True}
```

2.9

a) No. `'has_a_flex'` hasn't been defined.

b) No, `number_of_teams` is a variable that hasn't been defined, the key is `'number_of_teams'`.

c) Yes.

2.10a

```
In [34]:
for x in my_roster_list:
  print(x.split(' ')[-1])
--
brady
peterson
brown
```

2.10b

```
In [35]: {player: len(player) for player in my_roster_list}
Out[35]: {'tom brady': 9, 'adrian peterson': 15, 'antonio brown': 13}
```

2.11a

```
In [37]: [pos for pos in my_roster_dict]
Out[37]: ['qb', 'rb1', 'wr1', 'wr2']
```

2.11b

```
In [38]:
[player for _, player in my_roster_dict.items()
    if player.split(' ')[-1][0] in ['a', 'b']]
--
Out[38]: ['tom brady', 'davante adams', 'john brown']
```

2.12a

```
def mapper(my_list, my_function):
  return [my_function(x) for x in my_list]
```

2.12b

```
In [41]: mapper(list_of_rushing_yds, lambda x: x*0.1)
Out[41]: [0.1, 11.0, 6.0, 0.4, 0.0, 0.0, 0.0]
```

3.1 Pandas Basics

3.1.1

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))
```

3.1.2

```
# works because data is sorted by adp already
In [2]: adp50 = adp.head(50)

# this is better if don't want to assume data is sorted
In [3]: adp50 = adp.sort_values('adp').head(50)
```

3.1.3

```
In [4]: adp.sort_values('name', ascending=False, inplace=True)

In [5]: adp.head()
Out[5]:
       adp  ...                name  ...  stdev team  times_drafted
125  122.0  ...           Zay Jones  ...   10.4  BUF             94
72    72.8  ...           Zach Ertz  ...    7.8  PHI            148
86    83.8  ...       Willie Snead   ...    9.2   NO            122
173  161.0  ...            Wil Lutz  ...    6.7   NO             42
151  146.1  ...  Wendell Smallwood   ...   14.4  PHI            157
```

Note: if this didn't work when you printed it on a new line in the REPL you probably forgot the `inplace=True` argument.

3.1.4

```
In [8]: type(adp.sort_values('adp'))  # it's a DataFrame
Out[8]: pandas.core.frame.DataFrame
```

3.1.5a

```
In [10]: adp_simple = adp[['name', 'position', 'adp']]
```

3.1.5b

```
In [11]: adp_simple = adp_simple[['position', 'name', 'adp']]
```

3.1.5c

```
In [12]: adp_simple['team'] = adp['team']
```

3.1.5d

```
In [13]: adp.to_csv(path.join(DATA_DIR, 'adp.txt'), sep='|')
```

3.2 Columns

3.2.1

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'))
```

3.2.2

```
In [15]: pg['rec_pts_ppr'] = (0.1*pg['rec_yards'] + 6*pg['rec_tds']
             + pg['receptions' ])

In [16]: pg['rec_pts_ppr'].head()
Out[16]:
0     0.0
1     0.0
2    16.0
3     1.8
4     9.0
```

3.2.3

```
In [17]: (pg['player_desc'] = pg['player_name'] + ' is the ' + pg['team']
    +
             ' ' + pg[ 'pos'])

In [18]: pg['player_desc'].head()
Out[18]:
0       T.Brady is the NE QB
1       A.Smith is the KC QB
2    D.Amendola is the NE WR
3    R.Burkhead is the NE RB
4       T.Kelce is the KC TE
```

3.2.4

```
In [25]: pg['is_possession_rec'] = pg['caught_airyards'] > pg['raw_yac']

In [26]: pg['is_possession_rec'].head()
Out[26]:
0    False
1    False
2     True
3    False
4    False
```

3.2.5

```
In [27]:
pg['len_last_name'] = (pg['player_name']
                       .apply(lambda x: len(x.split('.')[-1])))
--

In [28]: pg['len_last_name'].head()
Out[28]:
0    5
1    5
2    8
3    8
4    5
```

3.2.6

```
In [29]: pg['gameid'] = pg['gameid'].astype(str)
```

3.2.7a

```
In [31]: pg.columns = [x.replace('_', ' ') for x in pg.columns]

In [32]: pg.head()
Out[32]:
  player name  ...              player desc is possession rec  len last
      name
0     T.Brady  ...     T.Brady is the NE QB            False
    5
1     A.Smith  ...     A.Smith is the KC QB            False
    5
2  D.Amendola  ...  D.Amendola is the NE WR             True
    8
3  R.Burkhead  ...  R.Burkhead is the NE RB            False
    8
4     T.Kelce  ...     T.Kelce is the KC TE            False
    5
```

3.2.7b

```
In [33]: pg.columns = [x.replace(' ', '_') for x in pg.columns]

In [34]: pg.head()
Out[34]:
  player_name  ...              player_desc is_possession_rec
      len_last_name
0     T.Brady  ...     T.Brady is the NE QB            False
    5
1     A.Smith  ...     A.Smith is the KC QB            False
    5
2  D.Amendola  ...  D.Amendola is the NE WR             True
    8
3  R.Burkhead  ...  R.Burkhead is the NE RB            False
    8
4     T.Kelce  ...     T.Kelce is the KC TE            False
    5
```

3.2.8a

```
In [35]: pg['rush_td_percentage'] = pg['rush_tds']/pg['carries']
```

3.2.8b

`'rush_td_percentage'` is rushing tds divided by carries. Since you can't divide by 0, rush td percentage is missing whenever carries are missing.

To replace all the missing values with `-99`:

```
In [37]: pg['rush_td_percentage'].fillna(-99, inplace=True)

In [38]: pg['rush_td_percentage'].head()
Out[38]:
0   -99.0
1     0.0
2   -99.0
3     0.0
4     0.0
```

3.2.9

```
In [39]: pg.drop('rush_td_percentage', axis=1, inplace=True)
```

If you forget the `axis=1` Pandas will try to drop the *row* with the index value `'rush_td_percentage'`. Since that doesn't exist, it'll throw an error.

Without the `inplace=True`, Pandas just returns a new copy of `pg` without the `'rush_td_percentage'` column. Nothing happens to the original `pg`, though we could reassign it if we wanted like this:

```
pg = pg.drop('rush_td_percentage', axis=1)  # alt to inplace
```

3.3 Built-in Functions

3.3.1

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'))
```

3.3.2

```
In [42]:
pg['total_yards1'] = pg['rush_yards'] + pg['rec_yards'] + pg['pass_yards']

pg['total_yards2'] = pg[['rush_yards', 'rec_yards', 'pass_yards']].sum(
    axis=1)

(pg['total_yards1'] == pg['total_yards2']).all()
--
Out[42]: True
```

3.3.3a

```
In [43]: pg[['rush_yards', 'rec_yards']].mean()
Out[43]:
rush_yards    14.909154
rec_yards     32.761705
```

3.3.3b

```
In [44]: ((pg['pass_yards'] >= 300) & (pg['pass_tds'] >= 3)).sum()
Out[44]: 15
```

3.3.3c

```
[ins] In [48]:
(((pg['pass_yards'] >= 300) & (pg['pass_tds'] >= 3)).sum()
 / (pg['pos'] == 'QB').sum())
--
Out[48]: 0.10204081632653061
```

3.3.3d

```
In [49]: pg['rush_tds'].sum()
Out[49]: 141.0
```

3.3.3e

Most: 14, least: 8

```
In [50]: pg['week'].value_counts()
Out[50]:
14    92
4     91
13    89
12    89
3     89
2     86
7     86
10    85
11    84
5     84
15    82
16    82
17    79
9     79
6     79
1     79
8     76
```

3.4 Filtering

3.4.1

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))
```

3.4.2a

```
In [52]: adp_cb1 = adp.loc[adp['team'] == 'DAL', ['name', 'position', 'adp
    ']]

In [53]: adp_cb1.head()
Out[53]:
                name position    adp
4    Ezekiel Elliott       RB    6.2
20        Dez Bryant       WR   20.5
76   Darren McFadden       RB   75.3
115     Dak Prescott       QB  112.8
142     Cole Beasley       WR  136.2
```

3.4.2b

```
In [54]: adp_cb2 = adp.query("team == 'DAL'")[['name', 'position', 'adp']]
```

3.4.3

```
In [56]: adp_nocb = adp.loc[adp['team'] != 'DAL',
            ['name', 'position', 'adp', 'team']]

In [57]: adp_nocb.head()
Out[57]:
               name position  adp team
0     David Johnson       RB  1.3  ARI
1       LeVeon Bell       RB  2.3  PIT
2     Antonio Brown       WR  3.7  PIT
3       Julio Jones       WR  5.7  ATL
5  Odell Beckham Jr       WR  6.4  NYG
```

3.4.4a

Yes.

```
In [58]: adp['last_name'] = adp['name'].apply(lambda x: x.split(' ')[1])

In [59]: adp[['last_name', 'position']].duplicated().any()
Out[59]: True
```

3.4.4b

```
In [60]: dups = adp[['last_name', 'position']].duplicated(keep=False)

In [61]: adp_dups = adp.loc[dups]

In [62]: adp_no_dups = adp.loc[~dups]
```

3.4.5

```
In [78]:
import numpy as np

adp['adp_description'] = np.nan
adp.loc[adp['adp'] < 40, 'adp_description'] = 'stud'
adp.loc[adp['adp'] > 120, 'adp_description'] = 'scrub'

In [79]: adp[['adp', 'adp_description']].sample(5)
Out[79]:
       adp adp_description
43    42.5             NaN
5      6.4            stud
16    17.1            stud
77    76.5             NaN
139  134.5           scrub
```

3.4.6a

```
In [80]: adp_no_desc1 = adp.loc[adp['adp_description'].isnull()]
```

3.4.6b

```
In [81]: adp_no_desc2 = adp.query("adp_description.isnull()")
```

3.5 Granularity

3.5.1

Usually you can only shift your data from more (play by play) to less (game) granular, which necessarily results in a loss of information. If I go from knowing what Kareem Hunt rushed for on every particular play to just knowing how many yards he rushed for *total*, that's a loss of information.

3.5.2a

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
pbp = pd.read_csv(path.join(DATA_DIR, 'play_data_sample.csv'))
```

3.5.2b

Looks like it was Sony Michel with 106 yards in game `2018101412` and Kareem Hunt with 70 yards in game `2018111900`.

```
In [83]:
(pbp
.query("play_type == 'run'")
.groupby(['game_id', 'rusher_player_name'])['yards_gained'].sum())
--
Out[83]:
game_id     rusher_player_name
2018101412  C.Patterson             3
            Dam. Williams           1
            J.Edelman               7
            J.White                39
            K.Barner               16
            K.Hunt                 80
            P.Mahomes               9
            S.Michel              106
            S.Ware                  5
            S.Watkins              -1
            T.Brady                 3
            T.Hill                  0
2018111900  B.Cooks                 0
            J.Goff                  8
            K.Hunt                 70
            M.Brown                15
            P.Mahomes              28
            T.Gurley               55
```

3.5.2c

Just change `sum` of `'yards_gained'` to `mean`:

```
In [86]:
(pbp
.query("play_type == 'run'")
.groupby(['game_id', 'rusher_player_name'])['yards_gained'].mean())
--
Out[86]:
game_id     rusher_player_name
2018101412  C.Patterson           3.000000
            Dam. Williams         1.000000
            J.Edelman             7.000000
            J.White               6.500000
            K.Barner              5.333333
            K.Hunt                8.000000
            P.Mahomes             4.500000
            S.Michel              4.416667
            S.Ware                2.500000
            S.Watkins            -1.000000
            T.Brady               1.500000
            T.Hill                0.000000
2018111900  B.Cooks               0.000000
            J.Goff                4.000000
            K.Hunt                5.000000
            M.Brown               3.750000
            P.Mahomes             4.666667
            T.Gurley              4.583333
```

3.5.2d

```
In [87]:
pbp['lte_0_yards'] = pbp['yards_gained'] <= 0

(pbp
.query("play_type == 'run'")
.groupby(['game_id', 'rusher_player_name'])['lte_0_yards'].mean())
--
Out[87]:
game_id     rusher_player_name
2018101412  C.Patterson           0.000000
            Dam. Williams         0.000000
            J.Edelman             0.000000
            J.White               0.166667
            K.Barner              0.000000
            K.Hunt                0.100000
            P.Mahomes             0.000000
            S.Michel              0.125000
            S.Ware                0.000000
            S.Watkins             1.000000
            T.Brady               0.500000
            T.Hill                1.000000
2018111900  B.Cooks               1.000000
            J.Goff                0.000000
            K.Hunt                0.071429
            M.Brown               0.000000
            P.Mahomes             0.000000
            T.Gurley              0.250000
```

3.5.3

Count counts the number of non missing (non `np.nan`) values. This is different than `sum` which adds up the values in all of the columns. The only time `count` and `sum` would return the same thing is if you had a column filled with 1s without any missing values.

3.5.4

Pats (31.25%) and Chiefs vs LA (6.85%).

```
In [89]: pbp.groupby(['posteam', 'game_id'])['turnover', 'first_down'].
    mean()
Out[89]:
                     turnover  first_down
posteam game_id
KC       2018101412  0.034483    0.241379
         2018111900  0.068493    0.287671
LA       2018111900  0.024691    0.246914
NE       2018101412  0.012500    0.312500
```

3.5.5

Stacking is when you change the granularity in your data, but shift information from rows to columns (or vis versa) so it doesn't result in any loss on information.

An example would be going from the player-game level to the player level. If we stacked it, we'd go from rows being:

```
     player_name  week  pass_tds
214     T.Taylor    10       0.0
339       M.Ryan    14       1.0
220     T.Taylor    11       1.0
70       T.Brady    15       1.0
1192    C.Newton     1       2.0
```

To:

```
                 pass_tds        ...
week                    1    2  ...    12   13   14   15   16   17
player_name                     ...
A.Smith               4.0  1.0  ...   1.0  4.0  1.0  2.0  1.0  NaN
B.Bortles             1.0  1.0  ...   0.0  2.0  2.0  3.0  3.0  0.0
B.Hundley             NaN  NaN  ...   3.0  0.0  3.0  NaN  0.0  1.0
B.Roethlisberger      2.0  2.0  ...   5.0  2.0  2.0  3.0  2.0  NaN
C.Newton              2.0  0.0  ...   0.0  2.0  1.0  4.0  0.0  1.0
```

3.6 Combining DataFrames

3.6.1a

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
df_touch = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'touch.csv
    '))
df_yard = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'yard.csv')
    )
df_td = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'td.csv'))
```

3.6.1b

```
In [4]:
df_comb1 = pd.merge(df_touch, df_yard)
df_comb1 = pd.merge(df_comb1, df_td, how='left')

df_comb1[['rush_tds', 'rec_tds']] = df_comb1[['rush_tds', 'rec_tds']].
    fillna(0)
```

3.6.1c

```
In [5]:
df_comb2 = pd.concat([df_touch.set_index('id'), df_yard.set_index('id'),
                      df_td.set_index('id')], axis=1)
df_comb2[['rush_tds', 'rec_tds']] = df_comb2[['rush_tds', 'rec_tds']].
    fillna(0)
```

3.6.1d

Which is better is somewhat subjective, but I generally prefer `concat` when combining three or more DataFrames because you can do it all in one step.

Note `merge` gives a little more fine grained control over how you merge (left, or outer) vs `concat`, which just gives you inner vs outer.

3.6.2a

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
qb = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'qb.csv'))
rb = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'rb.csv'))
wr = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'wr.csv'))
te = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'te.csv'))
```

3.6.2b

```
In [7]: df = pd.concat([qb, rb, wr, te])
```

3.6.3a

```
import pandas as pd
from os import path

DATA_DIR = '/Users/nathan/ltcwff-files/data'
adp = pd.read_csv(path.join(DATA_DIR, 'adp_2017.csv'))
```

3.6.3b

```
In [12]:
for pos in ['QB', 'RB', 'WR', 'TE', 'PK', 'DEF']:
    (adp
     .query(f"position == '{pos}'")
     .to_csv(path.join(DATA_DIR, f'adp_{pos}.csv'), index=False))
```

Write a two line for loop to save subsets of the ADP data frame for each position.

3.6.3c

```
In [13]:
df = pd.concat([pd.read_csv(path.join(DATA_DIR, f'adp_{pos}.csv'))
    for pos in ['QB', 'RB', 'WR', 'TE', 'PK', 'DEF']], ignore_index=True)
```

4. SQL

Note: like the book, I'm just showing the SQL, be sure to call it inside `pd.read_sql` and pass it your sqlite connection to try these. See `04_sql.py` file for more.

4.1a

```
SELECT
    game.season, week, player_name, player_game.team, attempts,
    completions, pass_yards AS yards, pass_tds AS tds, interceptions
FROM player_game, team, game
WHERE
    player_game.team = team.team AND
    game.gameid = player_game.gameid AND
    team.conference = 'AFC' AND
    player_game.pos = 'QB'
```

4.1b

```
SELECT
    g.season, week, p.player_name, t.team, attempts, completions,
       pass_yards AS
    yards, pass_tds AS tds, interceptions
FROM player_game AS pg, team AS t, game AS g, player AS p
WHERE
    pg.player_id = p.player_id AND
    g.gameid = pg.gameid AND
    t.team = p.team AND
    t.conference = 'AFC' AND
    p.pos = 'QB'
```

4.2

```
SELECT g.*, th.mascot as home_mascot, ta.mascot as away_mascot
FROM game as g, team as th, team as ta
WHERE
    g.home = th.team AND
    g.away = ta.team
```

5.1 Scraping

5.1.1

```
from bs4 import BeautifulSoup as Soup
import requests
from pandas import DataFrame

ffc_base_url = 'https://fantasyfootballcalculator.com'

def scrape_ffc(scoring, nteams, year):
    # build URL based on arguments
    ffc_url = (ffc_base_url + '/adp' + _scoring_helper(scoring) +
               f'/{nteams}-team/all/{year}')
    ffc_response = requests.get(ffc_url)

    # all same as 05_01_scraping.py file
    adp_soup = Soup(ffc_response.text)
    tables = adp_soup.find_all('table')
    adp_table = tables[0]
    rows = adp_table.find_all('tr')

    # move parse_row to own helper function
    list_of_parsed_rows = [_parse_row(row) for row in rows[1:]]

    # put it in a dataframe
    df = DataFrame(list_of_parsed_rows)

    # clean up formatting
    df.columns = ['ovr', 'pick', 'name', 'pos', 'team', 'adp', 'std_dev',
                  'high', 'low', 'drafted', 'graph']

    float_cols =['adp', 'std_dev']
    int_cols =['ovr', 'drafted']

    df[float_cols] = df[float_cols].astype(float)
    df[int_cols] = df[int_cols].astype(int)

    df.drop('graph', axis=1, inplace=True)

    return df

# helper functions - just moving some logic to own section
def _scoring_helper(scoring):
    """
    Take a scoring system (either 'ppr', 'half', 'std') and return the
       correct
    FFC URL fragment.

    Note: helper functions are often prefixed with _, but it's not
       required.
    """
    if scoring == 'ppr':
        return '/ppr'
    elif scoring == 'half':
        return '/half-ppr'
    elif scoring == 'std':
        return '/standard'

def _parse_row(row):
```

5.1.2

The solution for this is the same as the previous, with two changes. First, `_parse_row` becomes:

```
def _parse_row_with_link(row):
    """
    Take in a tr tag and get the data out of it in the form of a list of
    strings, also get link.
    """
    # parse the row like before and save it into a list
    parsed_row = [str(x.string) for x in row.find_all('td')]

    # now get the link, which is the href attribute of the first 'a' tag
    link = ffc_base_url + str(row.find_all('a')[0].get('href'))

    # add link onto the end of the list and return
    return parsed_row + [link]
```

And then inside `scrape_ffc` we need to add `'link'` when renaming our columns.

```
df.columns = ['ovr', 'pick', 'name', 'pos', 'team', 'adp', 'std_dev', '
    high',
                  'low', 'drafted', 'graph', 'link']
```

5.1.3

```
def ffc_player_info(url):
    ffc_response = requests.get(url)
    player_soup = Soup(ffc_response.text)

    # info is in multiple tables, but can get all rows with shortcut
    rows = player_soup.find_all('tr')

    list_of_parsed_rows = [_parse_player_row(row) for row in rows]

    # this is a list of two item lists [[key1, value1], [key2, value2],
       ...],
    # so we're unpacking each key, value pair with for key, value in ...
    dict_of_parsed_rows = {key: value for key, value in
       list_of_parsed_rows}

    # now modify slightly to return what we want, which (per problem
    # instructions) is team, height, weight, birthday, and draft info
    return_dict = {}
    return_dict['team'] = dict_of_parsed_rows['Team:']
    return_dict['height'] = dict_of_parsed_rows['Ht / Wt:'].split('/')[0]
    return_dict['weight'] = dict_of_parsed_rows['Ht / Wt:'].split('/')[1]
    return_dict['birthday'] = dict_of_parsed_rows['Born:']
    return_dict['drafted'] = dict_of_parsed_rows['Drafted:']
    return_dict['draft_team'] = dict_of_parsed_rows['Draft Team:']

    return return_dict

def _parse_player_row(row):
    return [str(x.string) for x in row.find_all('td')]
```

5.2 APIs

5.2.1

```
import requests
from pandas import DataFrame
import pandas as pd

# given urls
url_adp = 'https://api.myfantasyleague.com/2019/export?TYPE=adp&JSON=1'
url_player = 'https://api.myfantasyleague.com/2019/export?TYPE=players&
    JSON=1'

# call w/ requests
resp_adp = requests.get(url_adp)
resp_player = requests.get(url_player)

# need to inspect the .json() results in the REPL to see what we want/can
    turn
# into a DataFrame - usually list of dicts
df_adp = DataFrame(resp_adp.json()['adp']['player'])
df_player = DataFrame(resp_player.json()['players']['player'])

# now combine
df = pd.merge(df_adp, df_player)

# get rid of IDP players, take top 200
df = df.query("position in ('WR', 'RB', 'TE', 'QB', 'Def', 'PK')")
df = df.head(200)
```

6. Summary and Data Visualization

Assuming you've loaded the play-by-play data into a DataFrame named pbp and imported seaborn as sns.

6.1a

```
g = (sns.FacetGrid(pbp)
     .map(sns.kdeplot, 'yards_gained', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Yards Gained Per Play, LTCWFF Sample')
```

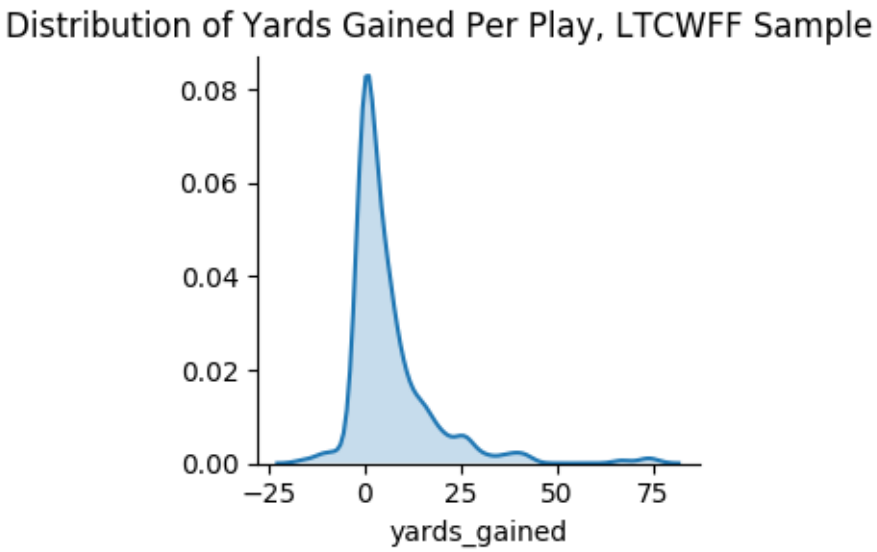


Figure 0.1: Solution 6-1a

6.1b

```
g = (sns.FacetGrid(pbp.query("down <= 3"), hue='down')
     .map(sns.kdeplot, 'yards_gained', shade=True))
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Yards Gained Per Play by Down, LTCWFF
    Sample')
```

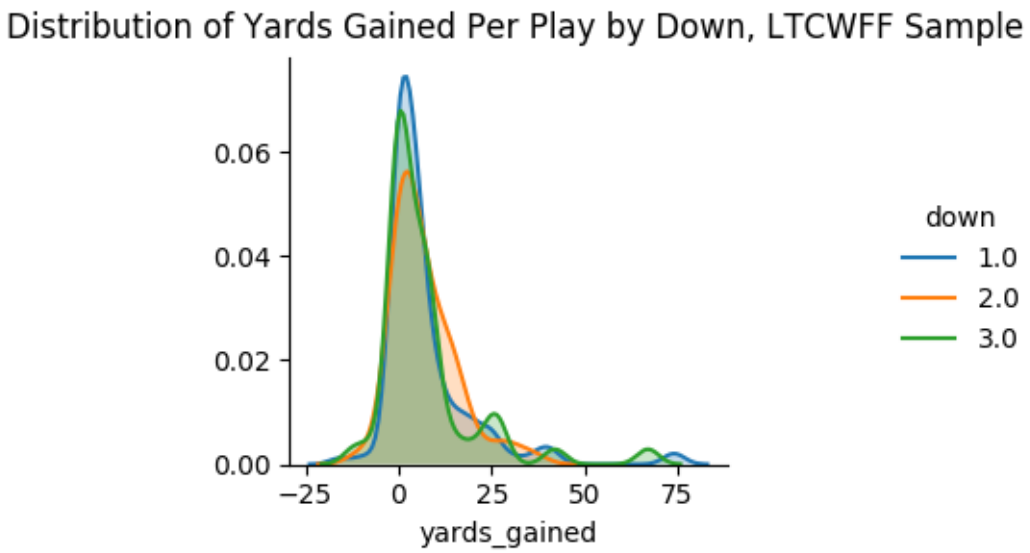


Figure 0.2: Solution 6-1b

6.1c

```
g = (sns.FacetGrid(pbp.query("down <= 3"), col='down')
     .map(sns.kdeplot, 'yards_gained', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Yards Gained Per Play by Down, LTCWFF
    Sample')
```

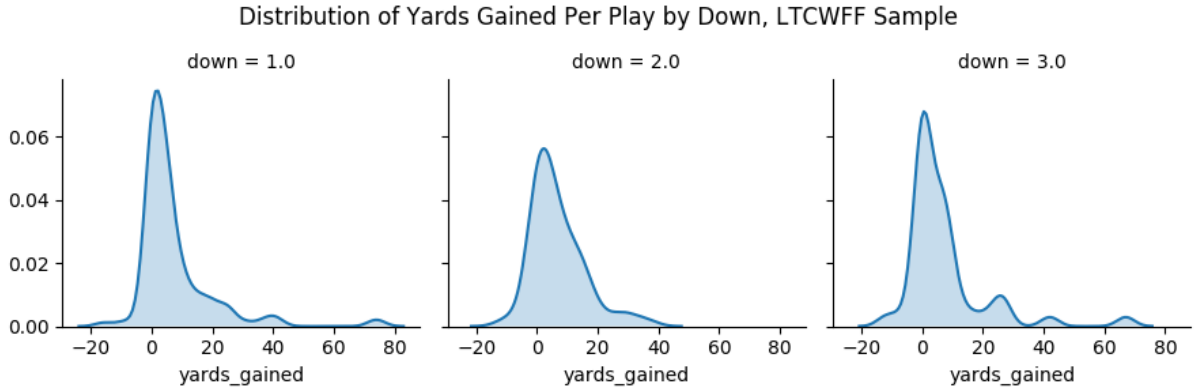


Figure 0.3: Solution 6-1c

6.1d

```
g = (sns.FacetGrid(pbp.query("down <= 3"), col='down', row='posteam')
     .map(sns.kdeplot, 'yards_gained', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Yards Gained Per Play by Down, Team,
    LTCWFF Sample')
```

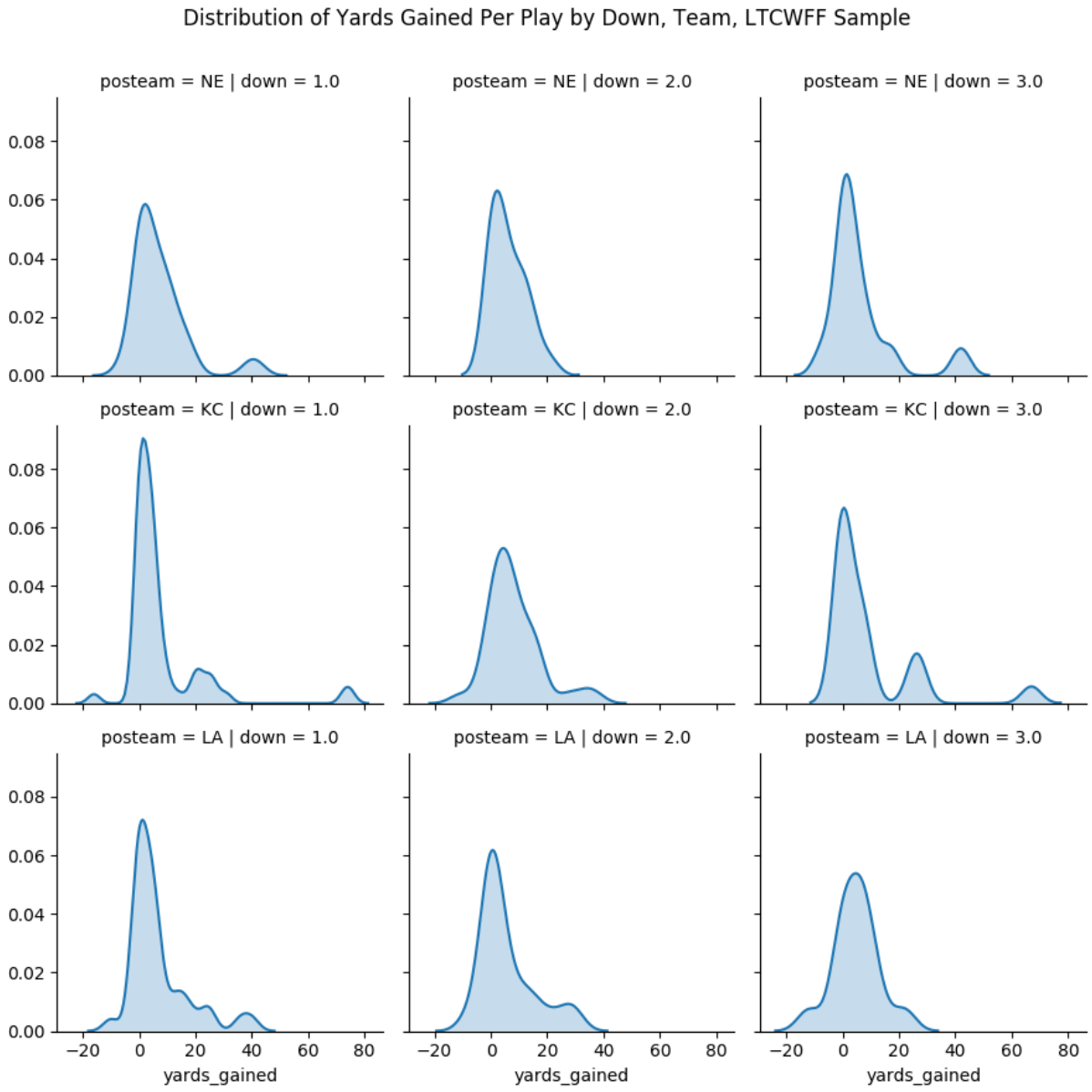


Figure 0.4: Solution 6-1d

6.1e

```
g = (sns.FacetGrid(pbp.query("down <= 3"), col='down', row='posteam',
                   hue='posteam')
     .map(sns.kdeplot, 'yards_gained', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Yards Gained Per Play by Down, Team,
    LTCWFF Sample')
```

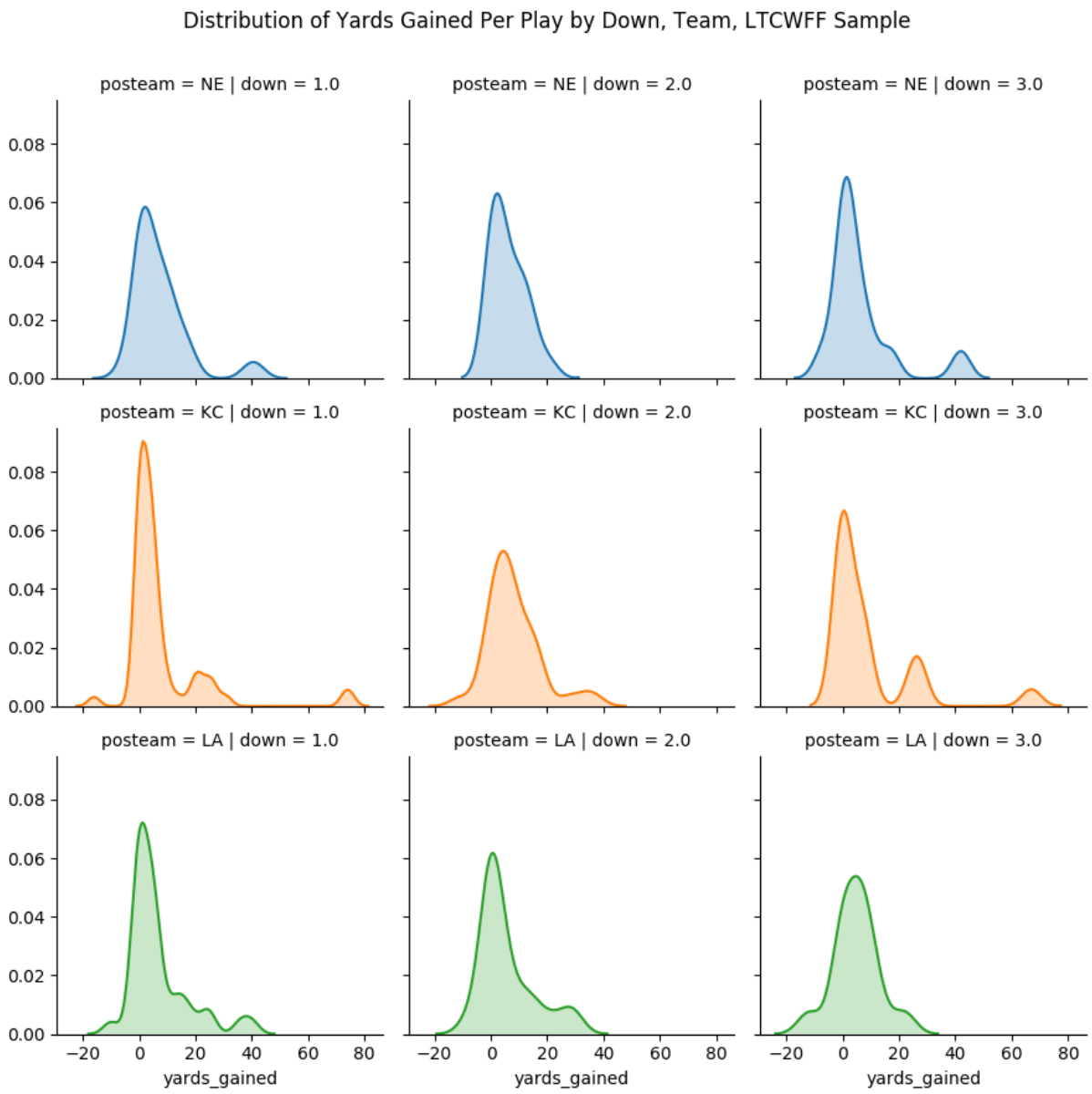


Figure 0.5: Solution 6-1e

6.2a

```
pg = pd.read_csv(path.join(DATA_DIR, 'player_game_2017_sample.csv'))
g = sns.relplot(x='carries', y='rush_yards', hue='pos', data=pg)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Carries vs Rush Yards by Position, LTCWFF Sample')
```

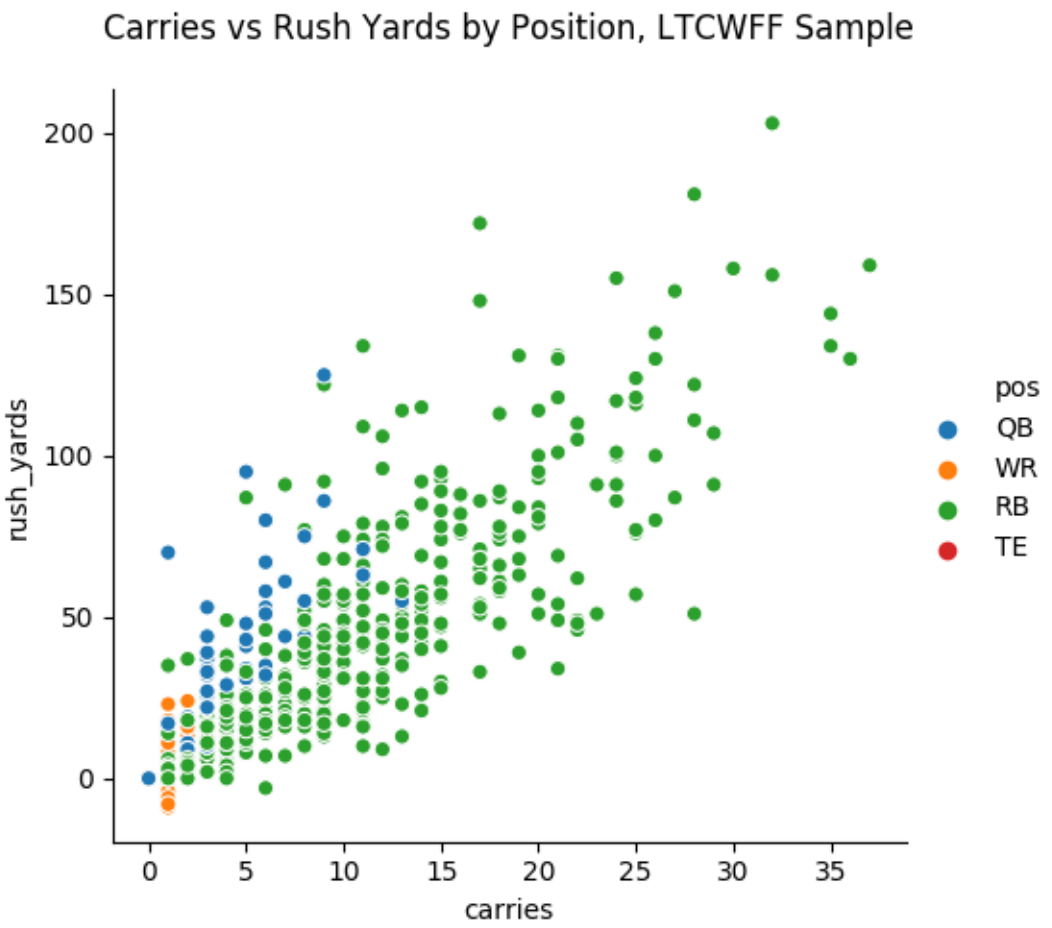


Figure 0.6: Solution 6-2

6.2b

To me, it looks like QBs have the highest yards per carry, but it's easy to verify:

```
pg['ypc'] = pg['rush_yards']/pg['carries']

# easy way to check
pg.groupby('pos')['ypc'].mean()

# more advanced (not seen before), but Pandas often is intuitive
pg.groupby('pos')['ypc'].describe()
```

7. Modeling

7.1a

To apply `prob_of_td` to our yardage data.

```
In [2]:
def prob_of_td(yds):
    b0, b1, b2 = results.params
    return (b0 + b1*yds + b2*(yds**2))

df['offensive_td_hat_alt'] = df['yardline_100'].apply(prob_of_td)
```

The two should be the same. With `'offensive_td_hat_alt'` we're just doing manually what `results.predict(df)` is doing behind the scenes. To verify:

```
In [3]: df[['offensive_td_hat', 'offensive_td_hat_alt']].head()
Out[3]:
   offensive_td_hat  offensive_td_hat_alt
0          0.017521              0.017521
1         -0.005010             -0.005010
2         -0.006370             -0.006370
3          0.007263              0.007263
4          0.007263              0.007263

In [4]: (results.predict(df) == df['offensive_td_hat_alt']).all()
Out[4]: True
```

7.1b

```
In [5]:
model_b = smf.ols(
    formula='offensive_td ~ yardline_100 + yardline_100_sq +ydstogo', data
       =df)
results_b = model_b.fit()
results_b.summary2()
--
Out[5]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.118
Dependent Variable: offensive_td     AIC:                2.1081
Date:               2020-03-05 11:39 BIC:                16.4869
No. Observations:   269              Log-Likelihood:     2.9460
Df Model:           3                F-statistic:        12.89
Df Residuals:       265              Prob (F-statistic): 6.88e-08
R-squared:          0.127            Scale:              0.058146
-----------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-----------------------------------------------------------------
Intercept          0.3273   0.0485  6.7430 0.0000  0.2317  0.4229
yardline_100      -0.0110   0.0024 -4.6646 0.0000 -0.0157 -0.0064
yardline_100_sq    0.0001   0.0000  3.6585 0.0003  0.0000  0.0001
ydstogo           -0.0009   0.0040 -0.2144 0.8304 -0.0088  0.0071
-----------------------------------------------------------------
Omnibus:              170.817      Durbin-Watson:         2.086
Prob(Omnibus):        0.000        Jarque-Bera (JB):      963.480
Skew:                 2.729        Prob(JB):              0.000
Kurtosis:             10.495       Condition No.:         12455
=================================================================
"""
```

Looking at the results in `results_b.summary2()` we can see the p value on `ydstogo` is 0.8304, which means the coefficient is not statistically significant.

7.1c

Wrapping a variable in `C()` turns it into a categorical variable (that' what the C stands for) and puts it in the model via fixed effects. Remember, including dummy variables for all four downs would be redundant, so statsmodels automatically drops first down.

```
In [6]:
model_c = smf.ols(formula='offensive_td ~ yardline_100 + yardline_100_sq +
    C(down)', data=df)
results_c = model_c.fit()
results_c.summary2()
--
Out[6]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.134
Dependent Variable: offensive_td     AIC:                -0.9816
Date:               2020-03-05 11:39 BIC:                20.5866
No. Observations:   269              Log-Likelihood:     6.4908
Df Model:           5                F-statistic:        9.288
Df Residuals:       263              Prob (F-statistic): 3.68e-08
R-squared:          0.150            Scale:              0.057064
-----------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-----------------------------------------------------------------
Intercept          0.3082   0.0475  6.4858 0.0000  0.2146  0.4018
C(down)[T.2.0]    -0.0321   0.0332 -0.9680 0.3339 -0.0975  0.0332
C(down)[T.3.0]     0.0830   0.0408  2.0330 0.0431  0.0026  0.1633
C(down)[T.4.0]    -0.0137   0.1221 -0.1123 0.9107 -0.2542  0.2267
yardline_100      -0.0106   0.0023 -4.5806 0.0000 -0.0151 -0.0060
yardline_100_sq    0.0001   0.0000  3.5085 0.0005  0.0000  0.0001
-----------------------------------------------------------------
Omnibus:              166.240      Durbin-Watson:         2.076
Prob(Omnibus):        0.000        Jarque-Bera (JB):      909.736
Skew:                 2.647        Prob(JB):              0.000
Kurtosis:             10.289       Condition No.:         31623
=================================================================
"""
```

The coefficient on third down (C(down)[T.3.0]) is highest at 0.0830 (since first down was dropped, we can interpret the coefficient on it as 0).

7.1d

Yes, the coefficients are the same. To see:

```
In [7]:
df['is2'] = df['down'] == 2
df['is3'] = df['down'] == 3
df['is4'] = df['down'] == 4

model_d = smf.ols(formula='offensive_td ~ yardline_100 + yardline_100_sq +
    is2 + is3 + is4', data=df)
results_d = model_d.fit()
results_d.summary2()

--
Out[7]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.134
Dependent Variable: offensive_td     AIC:                -0.9816
Date:               2020-03-05 11:42 BIC:                20.5866
No. Observations:   269              Log-Likelihood:     6.4908
Df Model:           5                F-statistic:        9.288
Df Residuals:       263              Prob (F-statistic): 3.68e-08
R-squared:          0.150            Scale:              0.057064
-----------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-----------------------------------------------------------------
Intercept          0.3082   0.0475  6.4858 0.0000  0.2146  0.4018
is2[T.True]       -0.0321   0.0332 -0.9680 0.3339 -0.0975  0.0332
is3[T.True]        0.0830   0.0408  2.0330 0.0431  0.0026  0.1633
is4[T.True]       -0.0137   0.1221 -0.1123 0.9107 -0.2542  0.2267
yardline_100      -0.0106   0.0023 -4.5806 0.0000 -0.0151 -0.0060
yardline_100_sq    0.0001   0.0000  3.5085 0.0005  0.0000  0.0001
-----------------------------------------------------------------
Omnibus:              166.240      Durbin-Watson:         2.076
Prob(Omnibus):        0.000        Jarque-Bera (JB):      909.736
Skew:                 2.647        Prob(JB):              0.000
Kurtosis:             10.289       Condition No.:         31623
=================================================================
* The condition number is large (3e+04). This might indicate
strong multicollinearity or other numerical problems.
"
```

7.2a

```
def run_sim_get_pvalue():
    coin = ['H', 'T']

    # make empty DataFrame
    df = DataFrame(index=range(100))

    # now fill it with a "guess"
    df['guess'] = [random.choice(coin) for _ in range(100)]

    # and flip
    df['result'] = [random.choice(coin) for _ in range(100)]

    # did we get it right or not?
    df['right'] = (df['guess'] == df['result']).astype(int)

    model = smf.ols(formula='right ~ C(guess)', data=df)
    results = model.fit()

    return results.pvalues['C(guess)[T.T]']
```

7.2b

When I ran it, I got an average p value of 0.4935 (it's random, so you're numbers will be different). The more you run it, the closer it will get to 0.50. In the language of calculus, the p value *approaches* 0.5 as the number of simulations approaches infinity.

```
In [13]:
sims_1k = Series([run_sim_get_pvalue() for _ in range(1000)])
sims_1k.mean()
--
Out[13]: 0.4934848731037103
```

7.2c

```
def runs_till_threshold(i, p=0.05):
    pvalue = run_sim_get_pvalue()
    if pvalue < p:
        return i
    else:
        return runs_till_threshold(i+1, p)

sim_time_till_sig_100 = Series([runs_till_threshold(1) for _ in range(100)
    ])
```

7.2d

According to Wikipedia, the mean and median of the Geometric distribution are 1/p and -1/log_2(1-p). Since we're working with a p of 0.05, that'd give us:

```
[ins] In [19]:
from math import log
p = 0.05
g_mean = 1/p
g_median = -1/log(1-p, 2)

g_mean, g_median
Out[19]: (20.0, 13.513407333964873)
```

After simulating 100 times and looking at the summary stats, I got 19.3 and 15 (again, your numbers will be different since we're dealing with random numbers), which are close.

```
In [20]: sim_time_till_sig_100.mean()
Out[20]: 19.3

In [21]: sim_time_till_sig_100.median()
Out[21]: 15.0
```

7.3a

```
[ins] In [33]:
model_a = smf.ols(formula=
    """
    wpa ~ offensive_td + interception + yards_gained + fumble
    """, data=df)
results_a = model_a.fit()
results_a.summary2()
--
Out[33]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                  Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.733
Dependent Variable: wpa              AIC:                -1041.8797
Date:               2020-03-05 11:54 BIC:                -1023.9062
No. Observations:   269              Log-Likelihood:     525.94
Df Model:           4                F-statistic:        184.8
Df Residuals:       264              Prob (F-statistic): 2.93e-75
R-squared:          0.737            Scale:              0.0011952
-----------------------------------------------------------------
                Coef.   Std.Err.     t      P>|t|    [0.025   0.975]
-----------------------------------------------------------------
Intercept      -0.0178    0.0026   -6.8153  0.0000  -0.0229  -0.0127
offensive_td    0.0663    0.0087    7.6368  0.0000   0.0492   0.0834
interception   -0.1850    0.0157  -11.7993  0.0000  -0.2159  -0.1541
yards_gained    0.0034    0.0002   17.2176  0.0000   0.0030   0.0038
fumble         -0.0560    0.0135   -4.1536  0.0000  -0.0826  -0.0295
-----------------------------------------------------------------
Omnibus:              71.070       Durbin-Watson:          2.161
Prob(Omnibus):        0.000        Jarque-Bera (JB):       1185.810
Skew:                 0.505        Prob(JB):               0.000
Kurtosis:             13.236       Condition No.:          101
=================================================================
"
```

An interception (-0.1850) is worse for win probability than a fumble (-0.0560). This is almost certainly because fumbles can be recovered by the offense, while interceptions are always turnovers.

7.3b

I'd expect the coefficient on `fumble_lost` to be a lot closer to the coefficient on `interception` than just `fumble` since now both are turnovers. In theory, it's hard for me to think why they wouldn't the same.

If you run the regression:

```
In [34]:
model_b = smf.ols(formula=
    """
    wpa ~ offensive_td + interception + yards_gained + fumble_lost
    """, data=df)
results_b = model_b.fit()
results_b.summary2()
--
Out[34]:
<class 'statsmodels.iolib.summary2.Summary'>
"""
                  Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.752
Dependent Variable: wpa              AIC:                -1061.9489
Date:               2020-03-05 11:56 BIC:                -1043.9753
No. Observations:   269              Log-Likelihood:     535.97
Df Model:           4                F-statistic:        204.2
Df Residuals:       264              Prob (F-statistic): 1.59e-79
R-squared:          0.756            Scale:              0.0011093
-----------------------------------------------------------------
                Coef.   Std.Err.     t      P>|t|    [0.025   0.975]
-----------------------------------------------------------------
Intercept      -0.0169    0.0025   -6.7305  0.0000  -0.0218  -0.0119
offensive_td    0.0672    0.0084    8.0317  0.0000   0.0507   0.0837
interception   -0.1859    0.0151  -12.3101  0.0000  -0.2157  -0.1562
yards_gained    0.0033    0.0002   17.3536  0.0000   0.0030   0.0037
fumble_lost    -0.0960    0.0154   -6.2480  0.0000  -0.1263  -0.0657
-----------------------------------------------------------------
Omnibus:              75.209       Durbin-Watson:          2.175
Prob(Omnibus):        0.000        Jarque-Bera (JB):       1457.539
Skew:                 0.520        Prob(JB):               0.000
Kurtosis:             14.356       Condition No.:          104
=================================================================

"
```

and compare the results you'll see that — while `fumble_lost` is worse for win probability than just `fumble` — the coefficient on interception is still larger. I would guess this is due to small sample sizes (we're only dealing with play-by-play data from two games). I'd be interested in seeing the regression on more games.

7.4

Because ADP is also in the regression, b2 can be interpreted as "the impact of being a rookie on average fantasy points, *controlling for draft position*."

If you think ADP is generally efficient, and players go where they're supposed to, you might think the b2 is close to 0 and statistically insignificant. On the other hand, if you're more dubious about the crowd, you might expect them to under (b2 > 0) or over value (b2 < 0) rookies.

7.5a

Aggregating down to the player level and taking the mean of all the x variables:

```
In [37]:
df_mean = df.groupby('player_id')[xvars].mean()
df_mean['pos'] = df.groupby('player_id')[yvar].first()

model_a = RandomForestClassifier(n_estimators=100)
scores_a = cross_val_score(model_a, df_mean[xvars], df_mean[yvar], cv=10)

scores_a.mean()  # 0.8807, which is better than player-game level model
--
Out[37]: 0.8525252525252526
```

7.5b

Aggregating down to the player level and taking the mean AND median, min and max, chucking them all into the model, and seeing how it does.

```
In [38]:
df_med = df.groupby('player_id')[xvars].median()
df_max = df.groupby('player_id')[xvars].max()
df_min = df.groupby('player_id')[xvars].min()
df_mean = df.groupby('player_id')[xvars].mean()

df_med.columns = [f'{x}_med' for x in df_med.columns]
df_max.columns = [f'{x}_max' for x in df_max.columns]
df_min.columns = [f'{x}_min' for x in df_min.columns]
df_mean.columns = [f'{x}_mean' for x in df_mean.columns]

df_mult = pd.concat([df_med, df_mean, df_min, df_max], axis=1)
xvars_mult = list(df_mult.columns)

df_mult['pos'] = df.groupby('player_id')[yvar].first()

model_b = RandomForestClassifier(n_estimators=100)
scores_b = cross_val_score(model_b, df_mult[xvars_mult], df_mult[yvar], cv
    =10)

scores_b.mean()

--
Out[38]: 0.869949494949495
```