

---

# **Learn to Code with Basketball**

v0.0.3

# Contents

<b>Prerequisites: Tooling</b>	<b>1</b>
Files Included with this Book . . . . .	1
Python . . . . .	2
Editor . . . . .	5
Console (REPL) . . . . .	5
Using Spyder and Keyboard Shortcuts . . . . .	7
Anki . . . . .	7
Remembering What You Learn . . . . .	7
<b>1. Introduction</b>	<b>8</b>
The Purpose of Data Analysis . . . . .	8
What is Data? . . . . .	8
Example Datasets . . . . .	10
Shot Data . . . . .	10
Player/Game Data . . . . .	11
Player Data . . . . .	11
Game Data . . . . .	12
What is Analysis? . . . . .	13
Types of Data Analysis . . . . .	14
Summary Statistics . . . . .	14
Modeling . . . . .	16
High Level Data Analysis Process . . . . .	18
1. Collecting Data . . . . .	18
2. Storing Data . . . . .	18
3. Loading Data . . . . .	18
4. Manipulating Data . . . . .	18
5. Analyzing Data for Insights . . . . .	19
Connecting the High Level Analysis Process to the Rest of the Book . . . . .	19
End of Chapter Exercises . . . . .	20

<b>2. Python</b>	<b>23</b>
Introduction to Python Programming . . . . .	23
How to Read This Chapter . . . . .	23
Important Parts of the Python Standard Library . . . . .	24
Comments . . . . .	24
Variables . . . . .	24
Types . . . . .	26
Interlude: How to Figure Things Out in Python . . . . .	27
Bools . . . . .	30
if statements . . . . .	31
Container Types . . . . .	31
Unpacking . . . . .	33
Loops . . . . .	34
Comprehensions . . . . .	35
Functions . . . . .	39
Libraries are Functions and Types . . . . .	45
os Library and path . . . . .	45
End of Chapter Exercises . . . . .	47
<b>3. Pandas</b>	<b>50</b>
Introduction to Pandas . . . . .	50
Types and Functions . . . . .	50
Things You Can Do with DataFrames . . . . .	51
How to Read This Chapter . . . . .	51
Part 1. DataFrame Basics . . . . .	52
Importing Pandas . . . . .	52
Loading Data . . . . .	53
DataFrame Methods and Attributes . . . . .	53
Working with Subsets of Columns . . . . .	54
Indexing . . . . .	56
Outputting Data . . . . .	59
Exercises . . . . .	60
Part 2. Things You Can Do With DataFrames . . . . .	61
Introduction . . . . .	61
1. Modify or Create New Columns of Data . . . . .	62
Creating or Modifying Columns - Same Thing . . . . .	62
Math and Number Columns . . . . .	63
String Columns . . . . .	64

Boolean Columns . . . . .	65
Applying Functions to Columns . . . . .	66
Dropping Columns . . . . .	67
Renaming Columns . . . . .	67
Missing Data in Columns . . . . .	67
Changing Column Types . . . . .	69
Review . . . . .	70
Exercises . . . . .	72
2. Use Built-In Pandas Functions That Work on DataFrames . . . . .	73
Summary Statistic Functions . . . . .	73
Axis . . . . .	74
Summary Functions on Boolean Columns . . . . .	75
Other Misc Built-in Summary Functions . . . . .	78
Review . . . . .	79
Exercises . . . . .	80
3. Filter Observations . . . . .	81
loc . . . . .	81
Combining Filtering with Changing Columns . . . . .	85
The query Method is an Alternative Way to Filter . . . . .	86
Review . . . . .	88
Exercises . . . . .	89
4. Change Granularity . . . . .	90
Ways of Changing Granularity . . . . .	90
Grouping . . . . .	90
A Note on Multilevel Indexing . . . . .	93
Stacking and Unstacking Data . . . . .	94
Review . . . . .	95
5. Combining Two or More DataFrames . . . . .	97
Merging . . . . .	97
Merge Question 1. What columns are you joining on? . . . . .	97
Merging is Precise . . . . .	98
Merge Question 2. Are you doing a 1:1, 1:many (or many:1), or many:many join? . . . . .	99
Merge Question 3. What are you doing with unmatched observations? . . . . .	101
More on pd.merge . . . . .	103
pd.merge() Resets the Index . . . . .	104
pd.concat() . . . . .	105
Combining DataFrames Vertically . . . . .	106
Review . . . . .	108

<b>4. SQL</b>	<b>110</b>
Introduction to SQL . . . . .	110
How to Read This Chapter . . . . .	110
Databases . . . . .	110
SQL Databases . . . . .	112
A Note on NoSQL . . . . .	112
SQL . . . . .	113
Pandas . . . . .	113
Creating Data . . . . .	113
Queries . . . . .	114
Filtering . . . . .	116
Joining, or Selecting From Multiple Tables . . . . .	118
Misc SQL . . . . .	123
SQL Example — LEFT JOIN, UNION, Subqueries . . . . .	124
End of Chapter Exercises . . . . .	128
<b>5. Web Scraping and APIs</b>	<b>129</b>
Introduction to Web Scraping and APIs . . . . .	129
Web Scraping . . . . .	129
HTML and CSS . . . . .	129
BeautifulSoup . . . . .	132
Simple vs Nested Tags . . . . .	135
Remember the ABA - Web Scraping Example . . . . .	139
APIs . . . . .	147
Two Types of APIs . . . . .	147
Web APIs . . . . .	147
HTTP . . . . .	148
JSON . . . . .	148
Benefits of APIs . . . . .	149
Working with APIs - General Process . . . . .	150
NBA API . . . . .	151
nba_api Wrapper . . . . .	161
<b>6. Data Analysis and Visualization</b>	<b>175</b>
Introduction . . . . .	175
Distributions . . . . .	176
Summary Stats . . . . .	179
Density Plots in Python . . . . .	183

Relationships Between Variables . . . . .	199
Scatter Plots with Python . . . . .	199
Contour Plots . . . . .	202
Correlation . . . . .	204
Line Plots with Python . . . . .	208
Plot Options . . . . .	211
Shot Charts . . . . .	217
Shot Charts As Seaborn Scatter Plots . . . . .	217
kwargs . . . . .	221
Contour Plots . . . . .	224
End of Chapter Exercises . . . . .	227
<b>7. Modeling</b>	<b>228</b>
Introduction to Modeling . . . . .	228
The Simplest Model . . . . .	228
Linear regression . . . . .	229
Statistical Significance . . . . .	234
Regressions hold things constant . . . . .	237
Fixed Effects . . . . .	241
Squaring Variables . . . . .	245
Logging Variables . . . . .	246
Interactions . . . . .	247
Logistic Regression . . . . .	249
Random Forest . . . . .	251
Classification and Regression Trees . . . . .	251
Random Forests are a Bunch of Trees . . . . .	252
Using a Trained Random Forest to Generate Predictions . . . . .	252
Random Forest Example in Scikit-Learn . . . . .	253
Random Forest Regressions . . . . .	258
End of Chapter Exercises . . . . .	259
<b>8. Intermediate Coding and Next Steps: High Level Strategies</b>	<b>261</b>
Gall's Law . . . . .	261
Get Quick Feedback . . . . .	262
Use Functions . . . . .	262
DRY: Don't Repeat Yourself . . . . .	263
Functions Help You Think Less . . . . .	263
Attitude . . . . .	263

Review . . . . .	265
<b>9. Conclusion</b>	<b>266</b>
<b>Appendix A: Places to Get Data</b>	<b>267</b>
nba_api Python wrapper . . . . .	267
Other Options . . . . .	267
Josh Gonzales's List of NBA Data Sources . . . . .	267
Kaggle.com . . . . .	267
Google Dataset Search . . . . .	267
<b>Appendix B: Anki</b>	<b>268</b>
Remembering What You Learn . . . . .	268
Installing Anki . . . . .	269
Using Anki with this Book . . . . .	270
<b>Appendix C: Answers to End of Chapter Exercises</b>	<b>272</b>
1. Introduction . . . . .	272
2. Python . . . . .	274
3.0 Pandas Basics . . . . .	278
3.1 Columns . . . . .	280
3.2 Built-in Functions . . . . .	284
3.3 Filtering . . . . .	286
3.4 Granularity . . . . .	288
3.5 Combining DataFrames . . . . .	295
4. SQL . . . . .	297
6. Summary and Data Visualization . . . . .	298
7. Modeling . . . . .	305

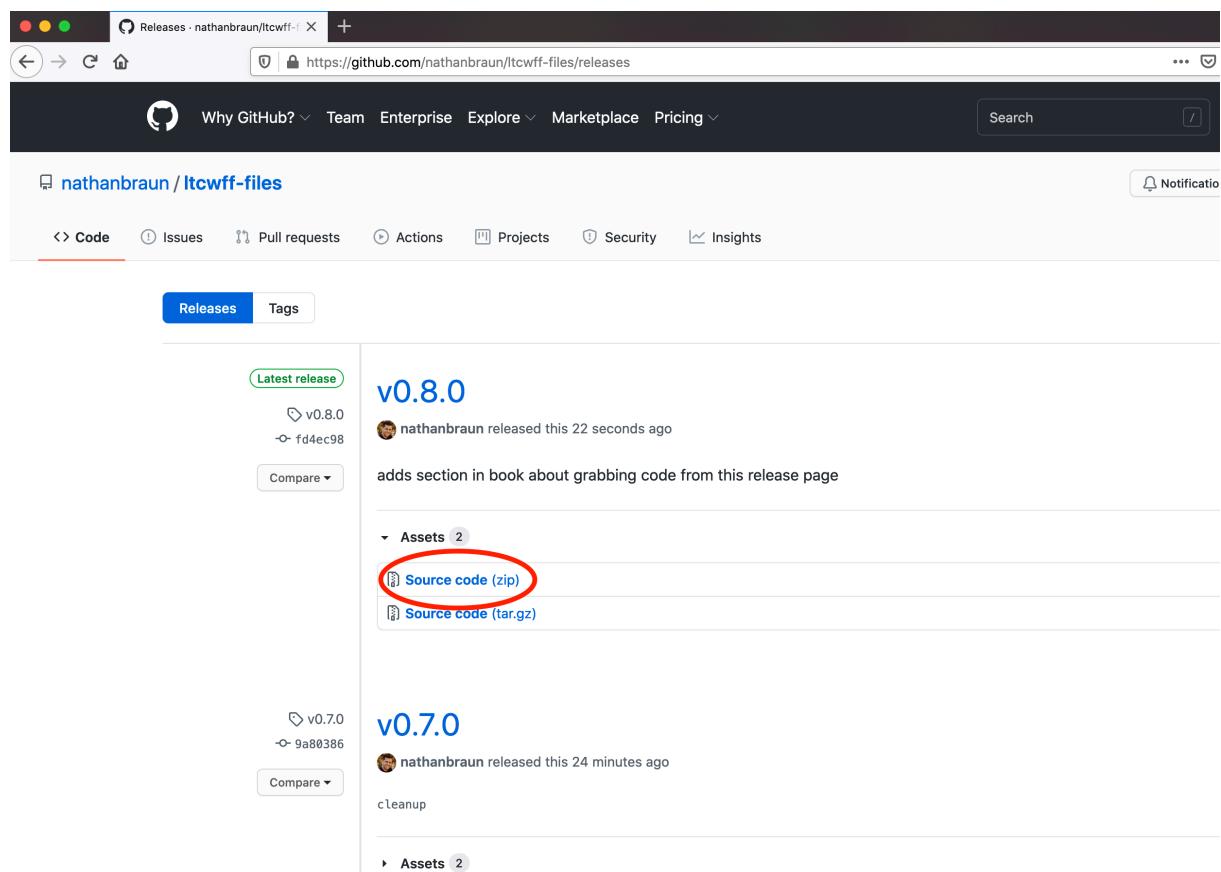
# Prerequisites: Tooling

## Files Included with this Book

This book is heavy on examples, most of which use small, “toy” datasets. You should be running and exploring the examples as you work through the book.

The first step is grabbing these files. They’re available at:

<https://github.com/nathanbraun/code-basketball-files/releases>



**Figure 0.1:** LTCWFF Files on GitHub

If you're not familiar with Git or GitHub, no problem. Just click the [Source code](#) link under the latest release to download a file called `code-basketball-files-vX.X.X.zip`, where X.X.X is the latest version number (v0.8.0 in the screenshot above).

When you unzip these (note in the book I've dropped the version number and renamed the directory just `code-basketball-files`, which you can do too) you'll see four sub-directories: `code`, `data`, `anki`, `solutions-to-exercises`.

You don't have to do anything with these right now except know where you put them. For example, on my mac, I have them in my home directory:

`/Users/nathanbraun/code-basketball-files`

If I were using Windows, it might look like this:

`C:\Users\nathanbraun\code-basketball-files`

Set these aside for now and we'll pick them up in chapter 2.

## Python

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

This book is hands on, and you'll 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 0.2:** Python 3.x on the Anaconda site

3. Then install it<sup>1</sup>. It might ask whether you want to install it for everyone on your computer or just you. Installing it for just yourself is fine.
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 on the right side is 'IPython console'.

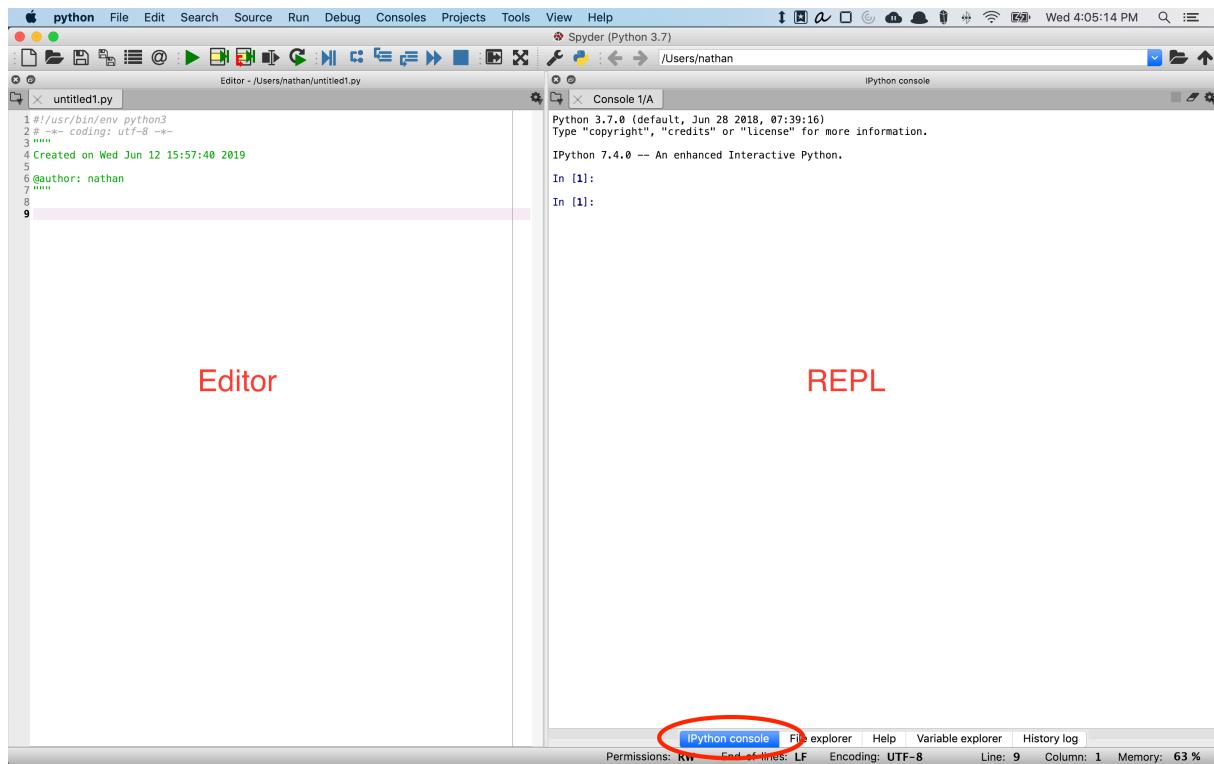
Now you should be ready to code. Your editor is on left, and your Python console is on the right. Let's touch on each of these briefly.

---

<sup>1</sup>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 more hard disk space than they need and using it will not slow down your computer. Once you are more familiar with Python, you may want to explore other, more minimalistic ways of installing it.

## Learn to Code with Basketball

---



**Figure 0.3:** Editor and REPL in Spyder

## 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).

In contrast, Python code is a set of instructions for working with data. You tell your program what to do, and Python does (aka *executes* or *runs*) 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 (for Python, each language has its own file extension) 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 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<sup>2</sup>, Python won't let you do it and will throw (or *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 and quit, but — because Python executes code from top to bottom — everything above the line with your error will have run like normal.

---

<sup>2</sup>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, it's 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 decides 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'm a sports bettor, will the Orlando Magic win more or less than 22.5 games this season?
- I'm an NBA GM, who should I draft with my first pick?
- I'm a mad scientist, how many more championships would Michael Jordan have had if he had not retired to play baseball?

Data analysis is one (hopefully) 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 basketball game, say Lakers vs Clippers, October 22, 2019. What would a collection of structured information about it look like?

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

Now **information** — what information might we have about each shot in this collection? Maybe: the player shooting, where it is on the court, how much time is left, what the score was, whether it went in, or whether it was a dunk.

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

This is an efficient, organized way of presenting information. When we want to know, "who took the first shot in the second quarter, how far was it and did it go in?", we can find the right row and columns, and say "Oh, Patrick Beverley from 25 feet, and no."

shot	name	period	min_left	sec_left	dist	made	value
31	L. Williams	1	2	26	16	True	2
32	Q. Cook	1	2	10	1	True	2
33	A. Davis	1	1	49	1	True	2
34	M. Harrell	1	1	35	10	False	2
35	K. Leonard	1	0	59	8	True	2
36	Q. Cook	1	0	44	18	False	2
37	J. Green	1	0	20	24	True	3
38	P. Beverley	2	11	43	25	False	3
39	K. Leonard	2	11	25	15	True	2
40	K. Caldwell-Pope	2	11	13	26	False	3
41	K. Leonard	2	11	2	23	True	3
42	L. James	2	10	48	29	True	3
43	J. Dudley	2	10	19	24	True	3
44	T. Daniels	2	10	1	26	False	3
45	M. Harkless	2	9	50	1	True	2

The **granularity** of a dataset is another word for the level the collection is at. Here, each row is a shot, and so the granularity of our data is at the shot level. It's very important to always know the granularity of your data.

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 (because all this info about Lakers-Clippers is *flattened* out into one big table).

A spreadsheet program like Microsoft Excel is one way to store data, but it's proprietary and not available on every computer. Also, spreadsheets often contain extra, non-data material like annotations, highlighting or plots.

A simpler, more common way (in the data world) 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 the shots data in a basic text editor like Notepad and see:

```
shot,name,period,min_left,sec_left,dist,made,value
31,L. Williams,1,2,26,16,True,2
32,Q. Cook,1,2,10,1,True,2
33,A. Davis,1,1,49,1,True,2
34,M. Harrell,1,1,35,10,False,2
35,K. Leonard,1,0,59,8,True,2
36,Q. Cook,1,0,44,18,False,2
37,J. Green,1,0,20,24,True,3
38,P. Beverley,2,11,43,25,False,3
39,K. Leonard,2,11,25,15,True,2
40,K. Caldwell-Pope,2,11,13,26,False,3
41,K. Leonard,2,11,2,23,True,3
42,L. James,2,10,48,29,True,3
43,J. Dudley,2,10,19,24,True,3
44,T. Daniels,2,10,1,26,False,3
45,M. Harkless,2,9,50,1,True,2
```

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 file extension.

This is just how the data is stored on your computer. No one expects you to open these files in Notepad and work with all those commas directly. In Excel (or other spreadsheet program) you can open and write csv files and they will be in the familiar spreadsheet format. 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 csv files that we will practice on. Instructions for getting these files are in the prerequisites section.

All these files are from 100 (random) 2019-2020 NBA games, sliced a few different ways:

### Shot Data

The first file is the shot data we were looking at above, where each row is a shot.

It includes: the player shooting, time left in game, and shot location (coordinates on the floor in x, y form) and distance, as well as information on the type of shot (2 or 3, whether it went in, whether it was a dunk, layup etc.

This data is in the file [./data/shot.csv](#).

## Player/Game Data

The second dataset this book comes with is at the player-game level. It's in the file `player_game.csv`.

The first few rows (and some of the columns, they don't all fit here) look like this:

game_id	name	fgm	fga	reb	ast	min	plus_minus	start
21900002	L. James	7	19	10	8	36.00	-8	True
21900002	D. Howard	1	3	6	1	19.03	-1	False
21900002	L. Williams	8	14	5	7	36.72	13	False
21900002	J. Dudley	2	2	0	0	13.35	-20	False
21900002	J. McGee	2	3	2	0	17.35	0	True

Remember: *Collection. Information. Structure.*

This data is a *collection* of player-game combinations. Each row represents one player's statistics for one game (Lebron James/vs Clippers). If we had 100 players, and they each played 82 games, then our dataset would be  $100 \times 82 = 8200$  rows long.

The columns (variables) are *information*. In the fourth row, each column tells us something about how Jared Dudley did in this game on 10/23/2019 vs the Clippers. The `fga` column shows us he took 2 shots, the `fgm` column indicates both went in, and the `plus_minus` column says the Lakers were down -20 points when he was on the floor (yikes).

If we want to look at another player-game (Lebron>this same game or Dudley/some other game), we look at a different row.

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

One thing to keep in mind: just because our data is at some level (player/game in this case), 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`. In this data, it's always 2019–20. Does this matter? No. We're just asking: "for this particular player/game, what season was it?" It just happens that for this data the answer is the same for every row.

## Player Data

We can also organize our data 2019-2020 season sample at the *player* level. This dataset is in `players.csv` and looks like this:

	name	team	pos	height	weight	jersey		school
V.	Carter	ATL	Guard	6-6	220.0	15.0	North	Carolina
J.	Crawford	BKN	Guard	6-5	200.0	1.0	Michigan	
L.	James	LAL	Forward	6-9	250.0	23.0	St. Vincent HS (OH)	
C.	Anthony	POR	Forward	6-7	238.0	0.0	Syracuse	
K.	Korver	MIL	Guard	6-7	212.0	26.0	Creighton	

## Game Data

Same thing at the game level. The file `games.csv` has a row for every game (not just our 100 game sample) in the 2019-2020 season. If a game (like LAC-LAL on 10/23) is in our detailed, random sample the `sample` column is `True`, otherwise it's `False`.

The `bubble` column indicates whether the game was played in the `bubble` (recall the final 8 games + playoffs of the 2019-2020 season were played in Orlando with heavy Covid precautions).

game_id	home	away	date	home_pts	away_pts	...	sample	bubble
21900001	TOR	NOP	2019-10-22	130	122	...	False	False
21900002	LAC	LAL	2019-10-22	112	102	...	True	False
21900003	CHA	CHI	2019-10-23	126	125	...	False	False
21900004	IND	DET	2019-10-23	110	119	...	False	False
...								
21901315	HOU	PHI	2020-08-14	96	134	...	False	True
21901316	IND	MIA	2020-08-14	109	92	...	False	True
21901317	LAC	OKC	2020-08-14	107	103	...	True	True
21901318	TOR	DEN	2020-08-14	117	109	...	True	True

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 basketballs are in the following picture?



**Figure 0.1:** Few basketballs

Pretty easy question right? What about this one?



**Figure 0.2:** Many basketballs

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 player-game data this book comes with, you'll notice it's 2114 rows and 31 columns.

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

Raw data is the numerical equivalent of a pile of basketballs. 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. Single number type summary statistics.
2. Models that help us understand relationships between data.

Let's talk about these more.

### Summary Statistics

Summary statistics can be complex (Defensive Win Share, Player Efficiency Rating) or more basic (field goal percentage or games missed due to injury), but all of them involve going from raw data to some more useful number.

These types of statistics are common in basketball. Some (field goals, assists, rebounds) have been around forever, others (PER or DRtg) are newer.

But stats don't necessarily need fancy acronyms to be useful. Take our player data:

	name	pos	height	weight
V.	Carter	Guard	6-6	220.0
J.	Crawford	Guard	6-5	200.0
L.	James	Forward	6-9	250.0
C.	Anthony	Forward	6-7	238.0
K.	Korver	Guard	6-7	212.0
D.	Howard	Center	6-10	265.0
A.	Iguodala	Guard	6-6	215.0
J.	Smith	Guard	6-6	220.0
T.	Ariza	Forward	6-8	215.0
M.	Williams	Forward	6-8	237.0
C.	Paul	Guard	6-0	175.0
...				
I.	Mahinmi	Center	6-11	262.0
C.	Miles	Guard	6-6	220.0
E.	Ilyasova	Forward	6-9	235.0
L.	Williams	Guard	6-1	175.0
L.	Aldridge	Center	6-11	250.0
R.	Gay	Forward	6-8	250.0
J.	Redick	Guard	6-3	200.0
T.	Sefolosha	Forward	6-6	215.0
R.	Rondo	Guard	6-1	180.0

What “statistic” might we use to understand the physical characteristics of NBA players?

How about, the average?

height	6.52	ft
weight	217.26	lbs

(Or the median, or mode, or a series of percentiles). Those are statistics and this is analysis.

The main goal of these single number summary statistics is usually to *summarize and make sense of some past performance*, e.g. when deciding who should be the MVP or arguing online about who contributed the most to the collapse of the 76ers.

Stats also vary in scope. Some, like plus-minus or offensive and defensive rating, are more all encompassing, while others get at a particular facet of a player’s performance. For example, we might measure “aggressiveness” by looking at the number of times a player gets to the line, or “clutchness” by looking at the difference between playoff and regular season free throw percentage.

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. As a result, they had a lot of success for the amount of money they spent<sup>1</sup>.

In practice, calculating summary statistics requires creativity, clear thinking and the ability to manipulate data via code.

## 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?

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 (with information about which columns are which) to your modeling program.

In practice, modeling is making a dataset where the columns are your input variables and one output variable, then passing this data (with information about which columns are which) to your 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 details and learn how to actually use these programs, but for now let’s get into motivation.

## 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.

---

<sup>1</sup>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.

For example, modeling number of assists as a function of the player, quality of other shooters and number of possessions, would be one way to figure out — by separating talent from other factors outside of a players control — which which point guard is the most “skilled”.

Then as an analyst for an NBA team assigned to scout free agent point guards, I can report back to my GM on who my model thinks is truly the best, the one who will have the most assists given our current team as it is.

## Models that Predict the Future

Often, we want to use a model to predict what will happen in the future.

For example, say I’m writing this on the eve of the regular season in October, 2022. I have everyone’s 2022 preseason stats, and I want to use them to predict points scored in 2022.

Modeling is about relationships. In this case the relationship is between data I have *now* (preseason stats for this year) and events that will happen in the *future* (regular season stats).

But if something is in the future, how can we relate it to the present?

By starting with the past.

If I’m writing this in October 2022, I have data on 2021. And I could build a model:

```
player 2021 points scored = model(player 2021 pre-season points scored)
```

*Training* (or *fitting*) this model is the process of using that known/existing/already happened data to find a relationship between the input variables (2021 preseason points) and the output variable (2021 regular season points).

Once I establish that relationship, I can feed it new inputs — 19 preseason points scored — and transform it using my relationship to get back a prediction for regular season points.

The inputs I feed my model might be from past events that have already happened. Often this done to evaluate model performance. For example, I could put in Ja Morant’s 2021 preseason stats to see what the model would have predicted for 2021, even though I already know how he did (hopefully the model gives me something close to how he did <sup>2</sup>).

Alternatively, I can feed it data from *right now* in order to predict things that *haven’t happened yet*. For example — again, say I’m writing this on the eve of the season opener in 2022 — I can put in Anthony Edward’s pre-season points scored and get back a projection for the regular season.

---

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

## 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. 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 covers basic SQL and loading data with Python.

### 4. Manipulating Data

Talk to any data scientist, and they'll tell you they spend most of their time preparing and manipulating their data. Basketball 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 — either confused or wondering *why* we're talking about something — refer back 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	quarter		name	nshots	nmade	ave_shot_distance
21901297	2	D. Hall		2	2	5.00
21900286	1	A. Wiggins		4	2	17.25
21901256	1	E. Bledsoe		3	0	13.00
21901232	2	A. Davis		2	1	2.50
21901235	2	D. Booker		6	2	12.33
21901293	3	T. Hardaway Jr.		4	2	18.50
21900308	3	M. Bridges		3	1	17.67
21900546	3	B. Forbes		5	1	22.00
21901313	1	R. Rubio		1	0	0.00
21900308	2	E. Fournier		4	2	10.75

b)

team_id	team	conference	division
1610612737	ATL	East	Southeast
1610612751	BKN	East	Atlantic
1610612738	BOS	East	Atlantic
1610612766	CHA	East	Southeast
1610612741	CHI	East	Central
1610612739	CLE	East	Central
1610612742	DAL	West	Southwest
1610612743	DEN	West	Northwest
1610612765	DET	East	Central
1610612744	GSW	West	Pacific

c)

	team	opp	date	wl	pts	reb	ast	stl	blk
261	MIA	UTA	2019-12-23	W	107	56	16	8	3
209	PHI	CHA	2019-11-10	W	114	49	29	9	5
1385	UTA	DET	2020-03-07	W	111	40	17	10	4
465	OKC	POR	2020-01-18	W	119	48	25	5	4
1554	SAC	DAL	2020-01-15	L	123	42	28	5	0
53	HOU	ATL	2019-11-30	W	158	52	30	9	2
491	OKC	POR	2019-11-27	L	119	38	19	4	2
1072	LAL	MEM	2019-11-23	W	109	40	19	12	7
529	TOR	IND	2020-02-05	W	119	37	27	12	0
1137	SAS	HOU	2019-12-03	W	135	56	35	10	8

d)

	shot_type	nshots	nmade	ave_shot_distance	make_pct
0	dunk	931	851	0.83	0.91
1	fadeaway	569	223	12.08	0.39
2	<b>float</b>	1215	502	8.33	0.41
3	hook	500	228	5.46	0.46
4	jump	6252	2304	23.05	0.37
5	layup	4702	2579	1.77	0.55
6	pullup	2005	787	19.50	0.39
7	step	702	286	20.89	0.41

e)

	jersey_number	n_players	max_height	min_height	ave_weight
0	19	6-8	5-10	208.00	
1	19	6-9	6-10	216.58	
2	12	6-9	6-1	213.33	
3	20	6-7	5-9	199.85	
4	15	6-7	5-11	202.40	
5	21	7-0	5-10	207.00	
6	6	7-3	6-1	214.50	
7	15	6-9	6-0	217.53	
8	13	7-0	6-0	215.15	
9	17	6-9	6-1	218.00	

## 1.2

I want to build a model that uses some team data (combined shooting percentage over last 5 games, total number of shots over last 5 games) to predict how many points a team will score. 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?

## 1.3

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.

Much of the functionality in Python comes from third party **libraries** (or **packages**), specially designed for specific tasks.

For example: the **Pandas** library 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 later in the book. But, even when using third party 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.

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. 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 something isn't working, it might be relying on code 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 your code, the computer 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's going on and what you were trying to do when you wrote it.

The problem with comments is it's easy for them to become out of date. This often happens when you change your code and forget 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 a fundamental concept in any programming language.

At their core, variables<sup>1</sup> 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.

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 that, 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 `assists_per_game`, a JavaScript programmer would write `assistsPerGame` instead.

## Assigning data to variables

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

```
In [1]: three_pt_made = 4
```

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

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

```
In [2]: three_pt_made  
Out[2]: 4  
  
In [3]: 3*three_pt_made  
Out[3]: 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 `three_pt_made` is 4, the assignment statement itself, `three_pt_made = 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, `three_pt_made` has a value of 4 (from the code we just ran above). So the right hand side, `three_pt_made + 1` is evaluated

---

<sup>1</sup>Note: previously we talked about how, in the language of modeling and tabular data, *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.

first ( $4 + 1 = 5$ ), and *then* the result gets (re)assigned to `three_pt_made`, overwriting the `4` it held previously.

```
In [4]: three_pt_made = three_pt_made + 1  
In [5]: three_pt_made  
Out[5]: 5
```

## 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 difference isn't important for us right now.

```
In [6]: over_under = 216 # int  
In [7]: fg_percentage = 0.48 # 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 D'Angelo), in which case trying to use '`D'Angelo Russel`' would give an error.

```
In [8]: starting_c = 'Karl-Anthony Towns'  
In [9]: starting_pg = "D'Angelo Russel"
```

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

```
In [10]: type(starting_c)  
Out[10]: str  
  
In [11]: type(over_under)  
Out[11]: int
```

Keep in mind the difference between strings (quotes) and variables (no quotes). A variable is a 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 them. The easiest way to do that is via **f-strings**.

```
In [12]: starters = f'{starting_c}, {starting_pg}, etc.'  
In [13]: starters  
Out[13]: "Karl-Anthony Towns, D'Angelo Russel, etc."
```

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 Python 3.8, so if they're not working for you make sure that's at least 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 [14]: 'from downtown!'.upper()  
Out[14]: 'FROM DOWNTOWN!'
```

Note the parenthesis. That's because sometimes these take additional data. For example the `replace` method takes two strings: the one you want to replace, and what you want to replace it with:

```
In [15]: 'Ron Artest'.replace('Artest', 'World Peace')  
Out[15]: 'Ron World Peace'
```

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 [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 can use to figure out your options is the REPL. In particular, the REPL's **tab completion** functionality. Type in a string like `'lebron james'` then . 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).

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

## Learn to Code with Basketball

---

Note: tab completion on a string directly like this doesn't always work in Spyder. If it's not working for you, assign '`lebron james`' to a variable and tab complete on that. Like this<sup>2</sup>:

```
In [16]: foo = 'lebron james'  
Out[16]: 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 '`lebron james`.`capitalize?`' (or `foo.capitalize?` if you're doing it that way).

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 [17]: 'lebron james'.capitalize()  
Out[17]: 'Lebron james'
```

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:

```
In [18]: ' lebron james'
```

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

```
In [19]: 'lebron james'
```

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

<sup>2</sup>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.

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 [20]: ' lebron james'.lstrip()  
Out[20]: 'lebron james'
```

## Stackoverflow

Python — particularly the data libraries we'll be using — became popular during the golden age of [stackoverflow.com](#), 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 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.

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 [21]: team1_pts = 110
In [22]: team2_pts = 120

# these are all bools:
In [23]: team1_won = team1_pts > team2_pts

In [24]: team2_won = team1_pts < team2_pts

In [25]: teams_tied = team1_pts == team2_pts

In [26]: teams_did_not_tie = team1_pts != team2_pts

In [27]: type(team1_won)
Out[27]: bool

In [28]: teams_did_not_tie
Out[28]: 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 [29]: 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.

The reverse is `!=`, which means *not equal*. The expression `team1_pts != team2_pts` is `True` if the values are different, `False` if they're the same.

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

```
In [30]: shootout = (team1_pts > 130) and (team2_pts > 130)

In [31]: at_least_one_good_team = (team1_pts > 120) or (team2_pts > 120)

In [32]: you_guys_are_bad = not ((team1_pts > 100) or (team2_pts > 100))

In [33]: 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 [34]
if team1_won:
    message = "Nice job team 1!"
elif team2_won:
    message = "Way to go team 2!!"
else:
    message = "must have tied!"

In [35]: message
Out[35]: '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 [36]: roster_list = ['kevin durant', 'kyrie irving', 'james harden']
```

Every spot in a list has a number associated with it. The first spot is 0. You can get sections (called **slices**) of your list by separating numbers with a colon. Both single numbers and slices are called inside square brackets, i.e. [].

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 last number, so [0:2] returns the 0 and 1 items, but not item 2.

```
In [37]: roster_list[0]
Out[37]: 'kevin durant'

In [38]: roster_list[0:2]
Out[38]: ['kevin durant', 'kyrie irving']
```

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

```
In [39]: roster_list[-2:]
Out[39]: ['kyrie irving', 'james harden']
```

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. They're written with curly brackets, like this:

```
In [40]:
roster_dict = {'PF': 'kevin durant',
               'SG': 'kyrie irving',
               'PG': 'james harden'}
```

You can access items in a dict like this:

```
In [41]: roster_dict['PF']
Out[41]: 'kevin durant'
```

And add new things to dicts like this:

```
In [42]: roster_dict['C'] = 'deandre jordan'

In [43]: roster_dict
Out[43]:
{'PF': 'kevin durant',
 'SG': 'kyrie irving',
 'PG': 'james harden',
 'C': 'deandre jordan'}
```

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 [44]: pos = 'PF'

In [45]: roster_dict[pos]
Out[45]: 'kevin durant'
```

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

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 `'deandre jordan'` above — is OK).

While dictionary *keys* are usually strings, 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 [46]: sg, pg = ['kyrie irving', 'james harden']
```

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

```
In [47]: sg = 'kyrie irving'

In [48]: pg = 'james harden'
```

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 [49]: sg, pg = ['kevin durant', 'kyrie irving', 'james harden']
...
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 `sg` and `pg` on separate lines.

However, some built-in parts of Python (including material below) use unpacking, so we needed to touch on it briefly.

## 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 roster_list = ['kevin durant', 'james harden', 'kyrie irving']
2
3 roster_list_upper = ['', '', '']
4 i = 0
5 for player in roster_list:
6     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 '`kevin durant`', the second '`james harden`', 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 [50]: roster_list_upper
Out[50]: ['Kevin Durant', 'James Harden', 'Kyrie Irving']
```

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 dicts is you get access to the keys only. So:

```
In [51]:
for x in roster_dict:
    print(f"position: {x}")
--
position: PF
position: SG
position: PG
position: C
```

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

```
In [52]:  
for x in roster_dict:  
    print(f"position: {x}")  
    print(f"player: {roster_dict[x]}")  
  
--  
position: PF  
player: kevin durant  
position: SG  
player: kyrie irving  
position: PG  
player: james harden  
position: C  
player: deandre jordan
```

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

```
In [53]:  
for x, y in roster_dict.items():  
    print(f"position: {x}")  
    print(f"player: {y}")  
  
position: PF  
player: kevin durant  
position: SG  
player: kyrie irving  
position: PG  
player: james harden  
position: C  
player: deandre jordan
```

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.

Mark Pilgrim, author of [Dive into Python](#), says that every programming language has some complicated, powerful concept that it makes intentionally simple and easy to do. Not every language can make everything easy, because all language decisions involve tradeoffs. Pilgrim says comprehensions are that feature for Python.

## List Comprehensions

When you want to go from one list to another, different list you should be thinking comprehension.

Earlier we took this list:

```
In [54]: roster_list
Out[54]: ['kevin durant', 'james harden', 'kyrie irving']
```

And changed it into `roster_list_proper` using a `for` loop.

The list comprehension way of doing that would be:

```
In [55]: roster_list_proper = [x.title() for x in roster_list]

In [56]: roster_list_proper
Out[56]: ['Kevin Durant', 'James Harden', 'Kyrie Irving']
```

All list comprehensions take the form `[a for b in c]` where `c` is the list you're starting with, 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 `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 [57]: roster_list_proper_alt = [y.title() for y in 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 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 [58]: type([x.title() for x in roster_list])
Out[58]: list
```

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

```
In [59]: [x.title() for x in roster_list][:2]
Out[59]: ['Kevin Durant', 'James Harden']
```

There is literally no difference.

## More Comprehensions

Let's do another, more complicated, comprehension:

```
In [60]: roster_last_names = [full_name.split(' ')[1]
                             for full_name in roster_list]

In [61]: roster_last_names
Out[61]: ['durant', 'harden', 'irving']
```

Remember, all list comprehensions take the form `[a for b in c]`. The last two are easy: `c` is just `roster_list` and `b` is `full_name`.

That leaves `a`, which is `full_name.split(' ')[1]`.

Sometimes its helpful to prototype this part in the REPL with an actual item from your list.

```
In [62]: full_name = 'kevin durant'

In [63]: full_name.split(' ')
Out[63]: ['kevin', 'durant']

In [64]: full_name.split(' ')[1]
Out[64]: 'durant'
```

We can see `split` is a string method that returns a list of substrings. After calling it we can pick out each player's last name in spot 1 of our new list.

The programming term for how we've been using comprehensions so far — “doing something” to each item in a collection — is **mapping**. As in, I *mapped title* to each element of `roster_list`.

We can also use comprehensions to **filter** a collection to include only certain items. To do this we add **if some criteria that evaluates to a boolean** at the end.

```
In [65]: roster_k_only = [  
    x for x in roster_list if x.startswith('k')]  
  
In [66]: roster_k_only  
Out[66]: ['kevin durant', 'kyrie irving']
```

Updating our notation, a comprehension technically has the form  $[a \text{ for } b \text{ in } c \text{ if } d]$ , where  $if \, d$  is optional.

Above,  $d$  is `x.startswith('k')`. The `startswith` string method takes a string and returns a bool indicating whether the original string starts with it or not. Again, it's helpful to test it out with actual items from our list.

```
In [67]: 'kevin durant'.startswith('k')  
Out[67]: True  
  
In [68]: 'james harden'.startswith('k')  
Out[68]: False  
  
In [69]: 'kyrie irving'.startswith('k')  
Out[69]: True
```

Interestingly, in this comprehension the  $a$  in our  $[a \text{ for } b \text{ in } c \text{ if } d]$  notation is just `x`. That means we're doing *nothing* to the value itself (we're taking `x` and returning `x`); the whole purpose of this comprehension is to filter `roster_list` to only include items that start with '`k`'.

You can easily extend this to map and filter in the same comprehension:

```
In [70]: roster_k_only_title = [  
    x.title() for x in roster_list if x.startswith('k')]  
--  
  
In [71]: roster_k_only_title  
Out[71]: ['Kevin Durant', 'Kyrie Irving']
```

## 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 [72]:  
salary_per_player = {  
    'kevin durant': 39058950, 'kyrie irving': 33460350,  
    'james harden': 41254920}  
  
In [73]:  
salary_m_per_upper_player = {  
    name.upper(): salary/1000000 for name, salary in salary_per_player.  
    items()}  
  
In [74]: salary_m_per_upper_player  
Out[74]: {'KEVIN DURANT': 39.05895, 'KYRIE IRVING': 33.46035,  
         'JAMES HARDEN': 41.25492}
```

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 money in our dict `salary_per_player`.

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

```
In [75]: sum([1, 2, 3])  
Out[75]: 6
```

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

```
In [76]: sum([salary for _, salary in salary_per_player.items()])  
Out[76]: 113774220
```

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

Also note the `for _, salary in ...` part of the code. The only way to get access to a value of a dict (i.e., the salary 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 salary, 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 return outputs. Python includes several built-in functions. Another common one is `len`, which finds the length of a list.

```
In [77]: len(['kevin durant', 'james harden', 'kyrie irving'])
Out[77]: 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(['kevin durant', 'james harden', 'kyrie irving'])` and 3. We could define variables with it:

```
In [78]: pts_per_3 = len(['kevin durant', 'james harden', 'kyrie irving'])

In [79]: pts_per_3
Out[79]: 3
```

Or use it in math.

```
In [80]: 4 + len(['kevin durant', 'james harden', 'kyrie irving'])
Out[80]: 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 in all programming languages to define your own functions.

```
def pts(fg2, fg3, ft):
    """
        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
        this function takes number of 2 point fgs, 3 point fgs, and free
        throws and
        returns total points scored
    """
    return fg2*2 + fg3*3 + ft*1
```

After defining a function (making sure to highlight it and send it to the REPL) you can call it like this:

```
In [81]: pts(8, 4, 5)
Out[81]: 33
```

Note the arguments `fg2`, `fg3` and `ft`. These work just like normal variables, except they are only available inside your function (the function's **body**).

So, even after defining and running this function it you try to type:

```
In [82]: print(fg2)
...
NameError: name 'fg2' is not defined
```

You'll get an error. `fg2` only exists inside the `pts` function.

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

You could put the `print` statement *inside* the function:

```
def pts_noisy(fg2, fg3, ft):
    """
    this function takes number of 2 point fgs, 3 point fgs, and free
    throws and
    returns total points scored

    it also prints out fg2
    """
    print(fg2) # works here since we're inside fn
    return fg2*2 + fg3*3 + ft*1
```

And then when we call it:

```
In [83]: pts_noisy(8, 4, 5)
8
Out[83]: 33
```

Note the 8 in the REPL. Along with returning a bool, `pts_noisy` prints the value of `pts2`. This is a **side effect** of calling the function. A side effect is anything your function does besides returning a value.

Printing variable values isn't a big deal (it can be helpful if your function isn't working like you expect), but apart from that 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 [84]: pts(8, 4)
...
TypeError: pts() missing 1 required positional argument: 'ft'
```

We get an error. We gave it 8 and 4, which got assigned to `fg2` and `fg3` but `ft` didn't get a value.

We can avoid this error by including **default** values. Let's make `ft` default to 0.

```
def pts_w_default(fg2, fg3, ft=0):
    """
    this function takes number of 2 point fgs, 3 point fgs, and free
    throws and
    returns total points scored
    """
    return fg2*2 + fg3*3 + ft*1
```

Now `ft` is **optional** because we gave it a default value. Note `fg2` and `fg3` are still **required** because they don't have default values. Also note this mix of required and optional arguments — this is fine. Python's only rule is any optional arguments have to come after required arguments.

Now the function call works:

```
In [85]: pts_w_default(8, 4)
Out[85]: 28
```

**Positional vs Keyword Arguments** Up to this point we've just passed the arguments in order, or by *position*.

So when we call:

```
In [87]: pts(8, 4, 5)
Out[87]: 33
```

The function assigns 8 to `fg2`, 4 to `fg3` and 5 to `ft`. It's in that order (`fg2`, `fg3`, `ft`) because that's the order we wrote them when we defined `pts` above.

These are called **positional arguments**.

We wrote this function, so we know the order the arguments go, but often we'll use third party code with functions we *didn't* write.

In that case we'll want to know the function's **Signature** — the arguments it takes, the order they go, and what's required vs optional.

It's easy to check in the REPL, just type the name of the function and a question mark:

```
In [88]: pts?
Signature: pts(fg2, fg3, ft)
...
Type:     function
```

The alternative to passing all the arguments in the correct positions is to use **keyword arguments**, like this:

```
In [89]: pts(fg3=4, fg2=8, ft=5)
Out[89]: 33
```

Keyword arguments are useful because you no longer have to remember the exact argument order. You also often need to use them with default values (think about it, if you're passing some values and not others, how's Python supposed to know which is which?)

You're allowed to mix positional and keyword arguments:

```
In [90]: pts(8, 4, ft=5)
Out[90]: 33
```

But Python's rule is that positional arguments have to come first.

One thing this implies is it's a good idea to put your most "important" arguments first, leaving your optional arguments for the end of the function definition.

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 path to your file, and that's the only argument you'll use 95% of the time. But it also has more than 40 optional arguments, everything from `skip_blank_lines` (defaults to `True`) to `parse_dates` (defaults to `False`).

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

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

And on the rare occasions when you do need to tweak some option, change the specific settings you want using keyword arguments:

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

Python's argument rules are precise, but pretty intuitive when you get used to them. See the end of chapter exercises for more practice.

## Functions That Take Other Functions

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

```
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 [91]: positions = ['C', 'PF', 'SF', 'SG', 'PG']

In [92]: do_to_list(positions, last_elem_in_list,
                    "last element in your list:")
Out[92]: 'last element in your list: PG'

In [93]: do_to_list([1, 2, 4, 8], last_elem_in_list,
                    "last element in your list:")
Out[93]: 'last element in your list: 8'
```

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

```
In [94]: do_to_list(positions, len, "length of your list:")
Out[94]: 'length of your list: 5'
```

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

```
In [95]: do_to_list([2, 3, 7, 1.3, 5], lambda x: 3*x[0],
                    "first element in your list times 3 is:")
Out[95]: '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 user defined functions and types <sup>3</sup> that other people have written using Python <sup>4</sup> 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 [96]: import os
```

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

```
In [97]: os.cpu_count()
Out[97]: 12
```

Libraries like `os` can contain sub-libraries too. The sub-library we'll use from `os` is `path`, 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 [98]: from os import path
In [99]: DATA_DIR = '/Users/nathan/nba-book/data'
In [100]: path.join(DATA_DIR, 'shot.csv')
Out[100]: '/Users/nathan/nba-book/data/shot.csv'
In [101]: os.path.join(DATA_DIR, 'shot.csv') # alt way of calling
Out[101]: '/Users/nathan/nba-book/data/shot.csv'
```

With `join`, you don't have to worry about trailing slashes or operating system differences or anything

<sup>3</sup>While we covered defining your own functions, we did *not* cover defining your own types — sometimes called *classes* — in Python. Working with classes is sometimes called *object-oriented* programming. While object-oriented programming and being able to write your own classes is sometimes helpful, it's definitely not required for everyday data analysis. I hardly ever use it myself.

<sup>4</sup>Technically sometimes they use other programming languages too. Parts of the data analysis library Pandas, for example, are written in the programming language C. But we don't have to worry about that.

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) `n_shots`
- c) `3_pt_percentage`
- d) `numOfBoards`
- e) `flagrant2`
- f) `coach name`
- g) `@home_or_away`
- h) `'ft_attempts'`

### 2.2

What is the value of `total_points` at the end of the following code?

```
total_points = 100
total_points = total_points + 28
total_points = total_points + 5
```

### 2.3

Write a function named `commentary` that takes in the name of a player and a play (e.g. '`Lebron`', '`dunk`') and returns a string of the form: '`Lebron with the dunk!`'.

### 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_fox` that takes in a player name and returns a `bool` indicating whether the player's name is "De'Aaron Fox" — regardless of case or whether the user included the '`'`.

## 2.6

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

## 2.7

Say we have a list:

```
roster = ['kevin durant', 'kyrie irving', 'james harden']
```

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

## 2.8

Say we have a dict:

```
shot_info = {'shooter': 'Steph Curry', 'is_3pt': True, 'went_in': False}
```

- How would you change '`shooter`' to 'Devon Booker'?
- Write a function `toggle3` that takes a dict like `shot_info`, turns '`is_3pt`' to the opposite of whatever it is, and returns the updated dict.

## 2.9

Assuming we've defined our same dict:

```
shot_info = {'shooter': 'Steph Curry', 'is_3pt': True, 'went_in': False}
```

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

- `shot_info['is_ft']`
- `shot_info[shooter]`
- `shot_info['distance'] = 23`

## 2.10

Say we're working with the list:

```
roster = ['kevin durant', 'kyrie irving', 'james harden']
```

- a) Write a loop that goes through and prints the last name of every player in `roster`.
- b) Write a comprehension that uses `roster` 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:

```
roster_dict = {'PF': 'kevin durant',
               'SG': 'kyrie irving',
               'PG': 'james harden',
               'C': 'deandre jordan'}
```

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

## 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) Use `mapper` with an anonymous function to turn this into a list of *points* from threes made (i.e. multiply each of these by three):

```
list_of_n_3pt_made = [5, 6, 1, 0, 4, 4]
```

# 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. Remember 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 learned data is a collection of structured information; each row is an observation and each column some attribute.

In chapter two, we learned about Python *types* and *functions* and talked about how third-party, user written *libraries* are just collections of types and functions people have written.

Now we can tie the two together. **Pandas** is a third-party Python library that gives you types and functions for working with tabular data. The most important is a **DataFrame**, which is a container type like a list or dict and holds a single 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 the DataFrames and Series types and functions that operate on them.*

This sounds simple, but Pandas is powerful, and there are many ways to "operate" on 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 go over a fundamental feature of DataFrames called indexes, and wrap up with outputting DataFrames as csv files.

With those basics covered, we'll learn about *things you can do with DataFrames*. This list is long and — rapid fire one after the other — might get overwhelming. But everything you do with DataFrames falls into one of the following categories:

### Things You Can Do with DataFrames

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 basketball data analyst will be working with Pandas. Most of your time in Pandas will be working with DataFrames. 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

### Importing Pandas

Note the examples for this section are in the file `03_00_basics.py`. The book picks up from the top of the file.

The first step to working with Pandas is importing it. Open up `03_00_basics.py` in your editor, and lets take a look at a few things.

First, note the lines at very top:

```
from os import path
import pandas as pd

DATA_DIR = '/Users/nathanbraun/nba-book/data'
```

These import the *libraries* (collections of functions and types that other people have written) we'll be using in this section.

It's customary to import all the libraries you'll need at the top of a file. We covered `path` in chapter 2. Though `path` is part of the standard library (i.e. no third party installation necessary), we still have to import it in order to use it.

Pandas is an external, third party library. Normally you have to 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 Pandas installed.

After you've changed `DATA_DIR` to the location where you've stored the files that came with the book, you can run these in your REPL:

```
In [1]: from os import path

In [2]: import pandas as pd

In [3]: DATA_DIR = './data'
```

Like all code — you have to send this to the REPL before you can use `pd` or `DATA_DIR` later in your programs. To reduce clutter, I'll usually leave this part out on future examples. But if you ever get an error like:

```
...
NameError: name 'pd' is not defined
```

Remember you have to run `import pandas as pd` in the REPL before doing anything else.

## Loading Data

When importing Pandas, the convention is to import it under the name `pd`. This lets us use any Pandas function by calling `pd.` (i.e. `pd` dot — type the period) and the name of our function.

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.

Let's try it:

```
In [4]: shots = pd.read_csv(path.join(DATA_DIR, 'shots.csv'))  
In [5]: type(shots)  
Out[5]: pandas.core.frame.DataFrame
```

Congratulations, you've loaded your first DataFrame!

## DataFrame Methods and Attributes

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

```
In [6]: shots.head()  
Out[6]:  
      name  dist  value  made  ...  period  min_left  sec_left  
0    L. James     2      2   True  ...        1       11      47  
1    L. Shamat   26      3  False  ...        1       11      40  
2    D. Green    25      3   True  ...        1       11      23  
3  P. Beverley   26      3  False  ...        1       11       0  
4    A. Davis    18      2  False  ...        1       10      47  
  
[5 rows x 37 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 when I'm coding with Pandas in real life.

## 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 [7]: shots.columns
Out[7]:
Index(['name', 'dist', 'value', 'made', 'desc', 'team', 'opp', 'x', 'y',
       'player_id', 'game_id', 'event_id', 'shot_id', 'running', 'jump',
       'hook', 'layup', 'driving', 'dunk', 'alley', 'reverse', 'turnaround',
       '',
       'fadeaway', 'bank', 'finger', 'putback', 'float', 'pullup', 'step',
       'cutting', 'tip', 'zone', 'area', 'date', 'period', 'min_left',
       'sec_left'],
      dtype='object')
```

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

```
In [8]: shots.shape
Out[8]: (16876, 37)
```

## 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 [9]: shots['name'].head()
Out[9]:
0      L. James
1      L. Shamet
2      D. Green
3      P. Beverley
4      A. Davis
```

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

```
In [10]: type(shots['name'])
Out[10]: 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 DataFrames or vis versa. Calling the `to_frame` method will turn any Series into a one-column DataFrame.

```
In [11]: shots['name'].to_frame().head()
Out[11]:
      name
0    L. James
1    L. Shamat
2    D. Green
3  P. Beverley
4    A. Davis

In [12]: type(shots['name'].to_frame().head())
Out[12]: 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 [13]: shots[['name', 'dist', 'value', 'made']].head()
Out[13]:
      name  dist  value  made
0    L. James    2      2   True
1    L. Shamat   26      3  False
2    D. Green    25      3   True
3  P. Beverley   26      3  False
4    A. Davis    18      2  False

In [14]: type(shots[['name', 'dist', 'value', 'made']].head())
Out[14]: pandas.core.frame.DataFrame
```

Notice the difference between the two:

- `shots['name']`
- `shots[['name', 'dist', 'value', 'made']]`

In the former we have `'name'`.

It's completely replaced by `['name', 'dist', 'value', 'made']` in the latter.

That is — since you're putting a list with your column names *inside* another pair of brackets — there are two sets of brackets when you're selecting multiple columns.

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

```
In [15], in <cell line: 1>()
...
KeyError: ('name', 'dist', 'value', 'made')
```

which will throw an error. No big deal, just remember — inside the brackets it's: one string (like a dict) to return a column, and a *list* of strings 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. Pandas lets you 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 `head`. 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 [16]: shots[['name', 'dist', 'value', 'made']].head()
Out[16]:
      name  dist  value  made
0    L. James     2      2   True
1    L. Shamat    26      3  False
2    D. Green     25      3   True
3  P. Beverley    26      3  False
4    A. Davis     18      2  False
```

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.

Let's make our index the `shot_id` column.

```
In [17]: shots.set_index('shot_id').head()
Out[17]:
           name  dist  value  ... period  min_left  sec_left
shot_id
0021900002-007    L. James     2      2  ...        1       11      47
0021900002-009    L. Shamat    26      3  ...        1       11      40
0021900002-011    D. Green     25      3  ...        1       11      23
0021900002-013  P. Beverley    26      3  ...        1       11       0
0021900002-015    A. Davis     18      2  ...        1       10      47
```

## Copies and the Inplace Argument

Now that we've run `set_index('shot_id')`, our new index is the `shot_id` column. Or is it?

Try running `head` again:

```
In [18]: shots.head()
Out[18]:
      name  dist  value  made  ...  period  min_left  sec_left
0    L. James     2      2  True  ...        1       11      47
1    L. Shamet   26      3 False  ...        1       11      40
2    D. Green    25      3  True  ...        1       11      23
3  P. Beverley   26      3 False  ...        1       11       0
4    A. Davis    18      2 False  ...        1       10      47
```

Our index is still 0, 1 ... 4 — what happened? The answer is that `set_index` returns a new, *copy* of the `shots` DataFrame with the index we want.

When we called `shots.set_index('shot_id')` above, we just displayed that newly index `shots` DataFrame in the REPL. We didn't actually do anything to our original, old `shots` DataFrame.

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

```
In [19]: shots.set_index('shot_id', inplace=True)

In [20]: shots.head() # now shot_id is index
Out[20]:
      name  dist  value  ...  period  min_left  sec_left
shot_id
0021900002-007    L. James     2      2  ...        1       11      47
0021900002-009    L. Shamet   26      3  ...        1       11      40
0021900002-011    D. Green    25      3  ...        1       11      23
0021900002-013  P. Beverley   26      3  ...        1       11       0
0021900002-015    A. Davis    18      2  ...        1       10      47
```

Or we can overwrite `shots` with our new, updated DataFrame:

```
# reload shot with default 0, 1, ... index
In [21]: shots = pd.read_csv(path.join(DATA_DIR, 'shots.csv'))

In [22]: shots = shots.set_index('shot_id')
```

Most DataFrame methods (including non-index related methods) behave like this, returning copies unless you explicitly 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 [23]: shots.reset_index().head()
Out[23]:
   shot_id      name  dist  ...  period  min_left  sec_left
0  0021900002-007    L. James    2  ...      1       11      47
1  0021900002-009    L. Shamet   26  ...      1       11      40
2  0021900002-011    D. Green   25  ...      1       11      23
3  0021900002-013  P. Beverley  26  ...      1       11       0
4  0021900002-015    A. Davis   18  ...      1       10      47
```

## Indexes Keep Things Aligned

The main benefit of indexes in Pandas is automatic alignment.

To illustrate this, let's make a mini subset of our DataFrame with just the name, distance and values of any overtime shots. Don't worry about the `loc` syntax for now, just know that we're creating a smaller subset of our data with just shots in overtime.

```
In [24]: shots_ot = shots.loc[shots['period'] > 4,
                           ['name', 'dist', 'value']]
In [25]: shots_ot.head()
Out[25]:
   name  dist  value
shot_id
0021900644-666  B. Adebayo    1      2
0021900644-668  D. Fox     22      3
0021900644-670  D. Robinson  25      3
0021900644-672  D. Fox      7      2
0021900644-674  G. Dragic    11      2
```

OK, so those are our overtime shots. Now let's use another DataFrame method to sort them by player name.

```
In [26]: shots_ot.sort_values('name', inplace=True)
In [27]: shots_ot.head()
Out[27]:
   name  dist  value
shot_id
0021901317-671  A. Coffey    0      2
0021901317-693  A. Coffey   25      3
0021900808-683  A. Gordon   14      2
0021900644-666  B. Adebayo   1      2
0021900644-748  B. Bogdanovic 26      3
```

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

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

```
In [28]: shots_ot['made'] = shots['made']

In [29]: shots_ot.head()
Out[29]:
   name  dist  value  made
shot_id
0021901317-671    A. Coffey    0     2  True
0021901317-693    A. Coffey   25     3 False
0021900808-683    A. Gordon   14     2 False
0021900644-666    B. Adebayo   1     2  True
0021900644-748    B. Bogdanovic 26     3 False
```

Viola. Even though we have a separate, smaller dataset with a different number of rows (only the overtime shots) in a different order (sorted alphabetically by player name instead of by time in the game), we were able to easily add in the correct `made` values from our old DataFrame.

We're able to do that because `shots`, `shots_ot`, and the Series `shots['made']` all have the same index 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 and moving columns around. The benefit of indexes in Pandas is you can just modify what you want without having 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 load csv files by calling `pd.read_csv` — the output methods are called on the DataFrame itself.

For example, to save our overtime DataFrame:

```
In [30]: shots_ot.to_csv(path.join(DATA_DIR, 'shots_ot.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 [31]: shots_ot.to_csv(path.join(DATA_DIR, 'shots_ot_no_index.csv'),
                       index=False)
```

## Exercises

### 3.0.1

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

### 3.0.2

Use the `game` DataFrame to create another DataFrame, `game50`, that is the first 50 games in the season (e.g. earliest by date).

### 3.0.3

Sort `game` by `home_pts` in descending order. On another line, look at `games` in the REPL and make sure it worked.

### 3.0.4

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

### 3.0.5

- a) Make a new DataFrame, `game_simple`, with just the columns '`date`', '`home`', '`away`', '`home_pts`' and '`away_pts`' in that order.
- b) Rearrange `game_simple` so the order is '`home`', '`away`', '`date`', '`home_pts`', '`away_pts`'.
- c) Using the original `game` DataFrame, add the '`game_id`' column to `game_simple`.
- d) Write a copy of `game_simple` to your computer, `game_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 what you can do with them.

In general, here's everything you can do with DataFrames:

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

Let's dive in.

## 1. Modify or Create New Columns of Data

WHERE WE ARE: *There are five main things you can do with Pandas DataFrames. This section is about the first, which is creating and modifying columns of data.*

*Note the examples for this section are in the file `03_01_columns.py`. The top of the file is importing libraries, setting `DATA_DIR`, and loading the player game data into a DataFrame named `pg`. The rest of this section picks up from there.*

The first thing we'll learn is how to work with *columns* in DataFrames. We'll cover both modifying and creating new columns, because they're really variations on the same thing.

### Creating or Modifying Columns - Same Thing

We've already seen how creating a new column works similarly to variable assignment in regular Python.

```
In [1]: pg['pts_per_shot'] = 2

In [2]: pg[['game_id', 'player_id', 'pts_per_shot']].head()
Out[2]:
   game_id  player_id  pts_per_shot
0  21900002        2544            2
1  21900002        2730            2
2  21900002       101150            2
3  21900002       201162            2
4  21900002       201580            2
```

What if we want to modify `pg['pts_per_shot']` after creating it?

It's no different than creating it originally.

```
In [3]: pg['pts_per_shot'] = 3

In [4]: pg[['game_id', 'player_id', 'pts_per_shot']].head()
Out[4]:
   game_id  player_id  pts_per_shot
0  21900002        2544            3
1  21900002        2730            3
2  21900002       101150            3
3  21900002       201162            3
4  21900002       201580            3
```

So the distinction between modifying and creating columns is minor. Really this section is 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 [5]: pg['pts_from_fgs'] = (pg['fg3m']*3 + (pg['fgm'] - pg['fg3m'])*2)

In [6]: pg[['name', 'game_id', 'pts_from_fgs']].head()
Out[6]:
      name    game_id  pts_from_fgs
0   L. James  21900002          15
1   D. Howard  21900002           2
2  L. Williams  21900002          17
3   J. Dudley  21900002           6
4    J. McGee  21900002           4
```

This adds a new column `pts_from_fgs` (number of threes \* 3 + number of twos \* 2) to our `pg` DataFrame<sup>1</sup>.

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 taking the absolute value and natural log<sup>2</sup> of plus minus:

```
In [7]: import numpy as np

In [8]: pg['biggest_impact'] = np.abs(pg['plus_minus'])

In [9]: pg['ln_pts'] = np.log(pg['pts'])
```

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

```
In [10]: pg['court_length'] = 94
```

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 (i.e. 5 random player-game combinations) 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.

<sup>1</sup>If we wanted, we could also create a separate column (Series), unattached to `pg`, but with the same index like this  
`pts_from_fgs = (pg['fg3m']*3 + (pg['fgm'] - pg['fg3m'])*2)`.

<sup>2</sup>If you're not familiar with what the natural log means this is a good link: <https://betterexplained.com/articles/demystifying-the-natural-logarithm-eln/>

We can see `court_length` is the same for every row:

```
In [11]: pg[['name', 'game_id', 'court_length']].sample(5)
Out[11]:
      name    game_id  court_length
1034    M. Oni    21900921        94
949     B. Fernando    21900867        94
1059    D. Favors    21901231        94
1762    A. Simons    21901289        94
941   M. Carter-Williams    21900867        94
```

## String Columns

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

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

```
In [1]: pg['name'].str.upper().sample(5)
Out[1]:
1553    K. PORZINGIS
1773    R. MCGRUDER
1725    D. AUGUSTIN
449     A. TOLLIVER
1051    M. ROBINSON

In [2]: pg['name'].str.replace('.', ' ').sample(5)
Out[2]:
904     D Mitchell
927     J Okafor
1170    H Whiteside
1079     J Noah
370     P George
```

The plus sign (+) concatenates (sticks together) string columns.

```
In [3]: (pg['name'] + ', ' + pg['team']).sample(5)
Out[3]:
502    B. Marjanovic, DAL
1131    J. Grant, WAS
1770    K. Leonard, LAC
581     P. Beverley, LAC
2059    C. Metu, SAS
```

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 [4]: pg['name'].str.replace('.',' ').str.lower().sample(5)
Out[4]:
114      n  vucevic
330      d  lillard
1134     t  brown jr
999      a  wiggins
1330     k  kuzma
```

## Boolean Columns

It's also common to work with columns of booleans.

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

```
In [1]: pg['is_a_guard'] = (pg['pos'] == 'Guard')

In [2]: pg[['name', 'is_a_guard']].sample(5)
Out[2]:
      name  is_a_guard
2054  J. Poeltl    False
225   M. Fultz     True
1711  D. Murray    True
751   B. McLemore  True
504   A. Nader    False
```

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

```
In [3]: pg['is_a_forward_or_center'] = ((pg['pos'] == 'Forward') |
                                         (pg['pos'] == 'Center'))

In [4]: pg['good_guard_game'] = (pg['pos'] == 'Guard') & (pg['pts'] >= 30)
```

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

```
In [5]: pg['not_gt_10_pts_or_assists'] = ~((pg['pts'] > 10) |
                                         (pg['ast'] > 10))
```

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 got either 10 points *or* 10 assists you could do:

```
In [6]: (pg[['pts', 'ast']] > 10).sample(5)
Out[6]:
      pts      ast
2028  False  False
1154   True  False
617   False  False
445   False  False
1201   True  False
```

This returns a DataFrame of all boolean values.

## Applying Functions to Columns

Pandas has a lot of built in functions that modify 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 booleans*) whether a team is in the Pacific Division. Rather than writing a long boolean expression with many | values, we might do something like:

```
In [1]:
def is_w_pac(team):
    """
    Takes some string named team ('MIL', 'LAL', 'CHI' etc) and checks
    whether it's in the Pacific Division
    """
    return team in ['LAC', 'LAL', 'PHX', 'SAC', 'GSW']

In [2]: pg['is_w_pac'] = pg['team'].apply(is_w_pac)

In [3]: pg[['name', 'team', 'is_w_pac']].sample(5)
Out[3]:
      name  team  is_w_pac
1070  T. Bradley  UTA    False
9      K. Leonard  LAC     True
722    M. Kleber  DAL    False
253    D. Eubanks  SAS    False
60     D. Brooks  MEM    False
```

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

Our function `is_pac` 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 [4]: paste -q pg['is_w_pac_alternate'] = pg['team'].apply(
    lambda x: x in ['LAC', 'LAL', 'PHX', 'SAC', 'GSW'])
```

## Dropping Columns

Dropping a column works like this:

```
In [5]: pg.drop('is_w_pac_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, renaming a column is one way to modify it, so let's talk about that here.

There are two ways to rename columns in Pandas. 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 [1]: pg.columns = [x.upper() for x in pg.columns]
```

```
In [2]: pg.head()
```

```
Out[2]:
```

	NAME	FGM	FGA	...	NOT_GT_10 PTS OR ASSISTS	IS_W_PAC
0	L. James	7	19	...	False	True
1	D. Howard	1	3	...	True	True
2	L. Williams	8	14	...	False	True
3	J. Dudley	2	2	...	True	True
4	J. McGee	2	3	...	True	True

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

```
In [3]: 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 `fgm` to `field_goals_made`.

```
In [4]: pg.rename(columns={'fgm': 'field_goals_made'}, 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 data.

Other times we might want to intentionally treat data as missing. For example, in our player-game data we could make a column *free throw percentage* (free throws made/free throw attempts).

```
In [1]: pg['ft_pct'] = pg['ftm']/pg['fta']
```

But what should it's value be for players with no free throw attempts?

Remember dividing by 0 is against the laws of math. We could use zero, but then there'd be no way to distinguish between guys who missed all their free throws and guys who didn't take any. Missing is better.

```
In [2]: pg[['name', 'team', 'ftm', 'fta', 'ft_pct']].head()
Out[2]:
   name  team  ftm  fta  ft_pct
0  L. James  LAL    3    4    0.75
1  D. Howard  LAL    1    1    1.00
2  L. Williams  LAC    4    4    1.00
3  J. Dudley  LAL    0    0     NaN
4  J. McGee  LAL    0    0     NaN
```

Missing values in Pandas have the value `np.NaN`. Remember, `np` is the numpy library that much of Pandas is built upon; `NaN` stands for “not a number”.

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

```
In [3]: pg['ft_pct'].isnull().head()
Out[3]:
0    False
1    False
2    False
3    True
4    True

In [4]: pg['ft_pct'].notnull().head()
Out[4]:
0    True
1    True
2    True
3    False
4    False
```

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

```
In [5]: pg['ft_pct'].fillna(-99).head()
Out[5]:
0      0.75
1      1.00
2      1.00
3    -99.00
4    -99.00
```

## Changing Column Types

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

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

```
In [1]: pg['date'].sample(5)
Out[1]:
221    20191129
40     20191023
578    20200104
146    20191117
294    20191202
```

In normal Python, if we wanted to get the year, month and day out of a string like `'20191119'` we would just do:

```
In [2]: date = '20191119'

In [3]: year = date[0:4]
In [4]: month = date[4:6]
In [5]: day = date[6:8]

In [6]: year
Out[6]: '2019'

In [7]: month
Out[7]: '11'

In [8]: day
Out[8]: '19'
```

So let's try some of our string methods on the `date` column.

```
In [9]: pg['month'] = pg['date'].str[4:6]
...
AttributeError: Can only use .str accessor with string values,
which use np.object_ dtype in pandas
```

It looks like `date` is stored as a number, which means `str` methods are not allowed.

No problem, we can convert it to a string using the `astype` method.

```
In [10]: pg['month'] = pg['date'].astype(str).str[4:6]

In [11]: pg[['name', 'team', 'month', 'date']].sample(5)
Out[11]:
   name  team  month      date
223    P. Siakam  TOR     11  20191129
1847   E. Sumner  IND     08  20200810
1669  D. McDermott  IND     08  20200808
1422    G. Niang  UTA     08  20200805
2105   C. Boucher  TOR     08  20200814
```

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 [12]: pg['month'].astype(int).sample(5)
Out[12]:
2106     8
1248     8
724      1
36      10
2008     8
```

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

```
In [13]: pg.dtypes.head()
Out[13]:
name          object
field_goals_made    int64
fga            int64
fg_pct         float64
pts            int64
dtype: object
```

Don't worry about the 64 after `int` and `float`; it's beyond the scope of this book. Also note Pandas refers to string columns as `object` (instead of `str`) in the `dtypes` output. This is normal.

## Review

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

## Learn to Code with Basketball

---

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.

## 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`, `'net_takeaways'` that is steals minus turnovers.

**3.1.3** Add a column `'player_desc'` to `pg` that takes the form, 'is the', e.g. `'L. James is the LAL Forward'` for Lebron.

**3.1.4** Add a boolean column to `pg` `'bad_game'` indicating whether took over 20 shots and had less than 15 points.

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

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

### 3.1.7

- Let's make the columns in `pg` more readable. Replace all the `'_'` with `' '` in all the columns.
- This actually isn't good practice. Change it back.

### 3.1.8

- Make a new column `'oreb_percentage'` indicating the percentage of a players rebounds that were offensive.
- There are missing values in this column, why? Replace all the missing values with -99.

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

## 2. Use Built-In Pandas Functions That Work on DataFrames

Note the examples for this section are in the file `03_02_functions.py`. We'll pick up right after you've loaded the DataFrame `pg` into your REPL.

Recall how analysis is the process of going from raw data to some statistic. Well, Pandas includes functions that operate on DataFrames and calculate certain statistics for you. In this section we'll learn about some of these and how to apply them to columns (the default) or rows.

### Summary Statistic Functions

Pandas includes a variety of functions to calculate summary statistics. For example, to take the *average* (or *mean*) of every numeric column in your DataFrame you can do:

```
In [1]: pg.mean()
Out[1]:
fgm           3.853832
fga           8.403500
fg_pct        0.430826
pts           10.659413
fg3m          1.176916
fg3a          3.220908
fg3_pct       0.271643
ftm           1.774834
fta           2.241722
ft_pct        0.436743
oreb          0.945128
dreb          3.297540
reb            4.242668
ast            2.308420
tov            1.317408
stl            0.715705
blk            0.437559
blk_a          0.437559
pf             2.021287
pdf            2.021287
min           22.824030
player_id      inf
date           inf
game_id         inf
plus_minus     0.000000
dd2            0.078524
start          0.473037
```

We can also do `max`.

```
In [2]: pg.max()
Out[2]:
name          Z. Williamson
fgm           17
fga           29
fg_pct        1.0
pts            51
fg3m           11
fg3a           18
fg3_pct        1.0
ftm            26
fta            27
ft_pct         1.0
oreb            8
dreb            18
reb             22
ast             17
tov             10
stl              7
blk              8
blka             5
pf               6
pdf              15
min            51.266667
season        2019-20
player_id      2772
pos             Guard
team            WAS
date        20200814
game_id       21901318
plus_minus      38
dd2              1
start           True
dtype: object
```

This returns the highest value of every column. Note unlike `mean`, `max` operates on string columns too (it treats “max” as latest in the alphabet, which is why we get ‘Z. Williamson’ for `name` in our player-game data).

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

## Axis

All of these functions take an `axis` argument which lets you specify whether you want to calculate the statistic 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:

```
In [3]: pg[['pts', 'ast', 'stl', 'reb']].mean(axis=0)
Out[3]:
pts    10.659413
ast    2.308420
stl    0.715705
reb    4.242668
```

(Note explicitly passing `axis=0` was unnecessary since it's the default, but I included it for illustrative purposes.)

Calling the function by rows, with `axis=1`, would make no sense. Remember, our `pg` data is by *player-game*, so calling `mean` with `axis=1` would give us the average of each player's number of: points, assists, steals and rebounds.

```
In [4]: pg[['pts', 'ast', 'stl', 'reb']].mean(axis=1).head()
Out[4]:
0    9.25
1    2.50
2    8.50
3    1.50
4    1.50
```

That number is meaningless here, but sometimes data is structured differently. For example, you might have a DataFrame where the columns are: `name, pts1, pts2, pts3 ... , pts82` — where each row represents a player and there are 82 columns, one for each game.

Then `axis=0` would give the average score across *all* players for *each game* (e.g. what was the average number of points scored by player for the first game of the season), which could be interesting, and `axis=1` would give you *each player's* average weekly score 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`.

What portion of our player-game sample are players who took more than 10 shots but scored less than 5 points?

```
In [1]: pg['cold_shooting'] = (pg['fga'] > 10) & (pg['pts'] < 5)

In [2]: pg['cold_shooting'].mean()
Out[2]: 0.0033112582781456954

In [3]: pg['cold_shooting'].sum()
Out[3]: 7
```

Two boolean specific summary functions 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 here attempt more than 20 three pointers?

```
In [4]: (pg['fg3a'] > 20).any()
Out[4]: False
```

Did everyone in this dataset play at least one minute?

```
In [5]: (pg['min'] > 0).all()
Out[5]: True
```

Yes. (Note this tells us something about our \*data — that it includes only player/game combinations where guys played at least a minute — rather than the NBA. Obviously sometimes players play 0 minutes in real life).

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

For example, to look by row and check if each player got 10 or more points, assists, rebounds, steals or blocks (a single double?) we could do:

```
In [6]: (pg[['pts', 'ast', 'reb', 'stl', 'blk']] >= 10).any(axis=1)
Out[6]:
0      True
1     False
2      True
3     False
4     False
...
2109    True
2110   False
2111   False
2112   False
2113    True
```

Note we can also modify this to calculate the number of triple doubles.

```
In [7]:  
pg['triple_double'] = ((pg[['pts', 'ast', 'reb', 'stl', 'blk']] >= 10)  
                        .sum(axis=1) >= 3)
```

Let's look at this. First, we're creating a DataFrame of booleans that indicates whether points, assists, rebounds, steals or blocks were 10 or more:

```
In [8]: (pg[['pts', 'ast', 'reb', 'stl', 'blk']] >= 10).head()  
Out[8]:  
      pts    ast    reb    stl    blk  
0  True  False  True  False  False  
1 False  False  False  False  False  
2  True  False  False  False  False  
3 False  False  False  False  False  
4 False  False  False  False  False
```

Then we're adding all these up at the player-game (row) level. So we need to do `axis=1`:

```
In [9]: (pg[['pts', 'ast', 'reb', 'stl', 'blk']] >= 10).sum(axis=1).head()  
Out[9]:  
0    2  
1    0  
2    1  
3    0  
4    0
```

Finally, we're flagging when this number is 3 or more:

```
In [10]: ((pg[['pts', 'ast', 'reb', 'stl', 'blk']] >= 10).sum(axis=1) >=  
            3).head()  
Out[10]:  
0    False  
1    False  
2    False  
3    False  
4    False
```

Let's see how many triple doubles we had in our sample:

```
In [11]: pg['triple_double'].sum()  
Out[11]: 6
```

## Other Misc Built-in Summary Functions

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

```
In [1]: pg['pos'].value_counts()
Out[1]:
Guard      1013
Forward     824
Center      277
```

We can **normalize** these frequencies — dividing each by the total so that they add up to 1 and represent proportions — by passing the `normalize=True` argument.

```
In [2]: pg['pos'].value_counts(normalize=True)
Out[2]:
Guard      0.479186
Forward    0.389782
Center     0.131031
```

So 47.92% (1013/2114) of the positions in our player-game data are guards, 38.97% (824/2114) are forwards, etc.

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

```
In [3]: pd.crosstab(pg['team'], pg['pos']).head()
Out[3]:
          pos   Center  Forward  Guard
team
ATL        4       12      17
BKN        7       18      39
BOS        8       24      24
CHA        1        4       4
CHI        0       4       6
```

Crosstab also takes an optional `normalize` argument.

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 number of shots (field goals and free throws) 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_shots1`' and '`total_shots2`'. Prove that they're the same.

### 3.2.3

- a) What were the average values for points, field goal attempts, and rebounds?
- b) How many times in our data did someone score at least 40 points and get at least 10 rebounds?
- c) What % of 40 point performances was that?
- d) How many three point attempts were there total in our sample?
- e) What team is most represented in our sample? Least?

### 3. Filter Observations

WHERE WE ARE: *There are five main things you can do with Pandas DataFrames. The third thing we can do with DataFrames is filter them.*

*Note the examples for this section are in the file `03_03_filter.py`. We'll pick up right after you've loaded the DataFrame `dfp` into your REPL.*

The third thing we can do with DataFrames is **filter** them, which means picking out a subset rows. In this section we'll learn how to filter based on criteria we set, as well as how to filter by dropping duplicates.

#### 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, 2544 is Lebron's `player_id`.

```
In [1]: lebron_id = 2544

In [2]: dfp.loc[lebron_id]
Out[2]:
first                  LeBron
last                   James
name                  L. James
birthdate             19841230
school                St. Vincent-St. Mary HS (OH)
country                USA
last_affiliation      St. Vincent-St. Mary HS (OH)/USA
height                 6-9
weight                 250.0
season_exp              17
jersey                  23.0
pos                     Forward
rosterstatus            Active
from_year               2003.0
dleague_flag              N
draft_year               2003
draft_round                1.0
draft_number                1.0
team_id                 1610612747
team_id2                  0
team                     LAL
team2                    NaN
```

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

```
In [3]: laker_ids = ([2544, 203076, 201566])  
  
In [4]: dfp.loc[laker_ids]  
Out[4]:
```

player_id	first	last	name	...	team_id2	team	team2
2544	LeBron	James	L. James	...	0	LAL	NaN
203076	Anthony	Davis	A. Davis	...	0	LAL	NaN
201566	Russell	Westbrook	R. Westbrook	...	0	HOU	NaN

While not technically filtering by rows, you can also pass `loc` a second argument to limit which columns you return. This returns the `name`, `school`, `height`, and `weight` columns for the ids we specified.

```
In [5]: dfp.loc[laker_ids, ['name', 'school', 'height', 'weight']]  
Out[5]:
```

player_id	name	school	height	weight
2544	L. James	St. Vincent-St. Mary HS (OH)	6-9	250.0
203076	A. Davis	Kentucky	6-10	253.0
201566	R. Westbrook	UCLA	6-3	200.0

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 [6]: dfp.loc[laker_ids, 'name']  
Out[6]:
```

player_id	name
2544	L. James
203076	A. Davis
201566	R. Westbrook

## 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 `loc` a column of bool values and it returns only values where the column is `True`.

So say we're just interested in North Carolina-inians. Let's create our column of booleans that indicate whether a player went to school in North Carolina.

```
In [7]: school_in_nc = dfp['school'] == 'North Carolina'

In [8]: school_in_nc.head()
Out[8]:
player_id
1713      True
2037     False
2544     False
2546     False
2594     False
```

And now pass this to `loc[]`. Again, note the brackets; calling `loc` is more like retrieving a value from a dictionary than calling a method.

```
In [9]: players_nc = dfp.loc[school_in_nc]

In [10]: players_nc[['name', 'school', 'height', 'weight']].head()
Out[10]:
          name        school  height  weight
player_id
1713      V. Carter  North Carolina    6-6   220.0
101107    M. Williams  North Carolina    6-8   237.0
201961    W. Ellington  North Carolina    6-4   207.0
201980      D. Green  North Carolina    6-6   215.0
202334      E. Davis  North Carolina    6-9   218.0
```

Boolean indexing requires that the column of booleans you're passing has the same index as the DataFrame you're calling `loc` on.

In this case we already know `school_in_nc` has the same index as `dfp` because we created it with `dfp['school'] == 'North Carolina'` and that's how Pandas works.

We broke the process into two separate steps above, first creating `school_in_nc` and then passing it to `loc`, but that's not necessary. This does the same thing without leaving `school_in_duke` lying around:

```
In [11]: players_duke = dfp.loc[dfp['school'] == 'Duke']

In [12]: players_duke[['name', 'school', 'height', 'weight']].head()
Out[12]:
          name  school  height  weight
player_id
200755    J. Redick    Duke    6-3   200.0
202498    L. Thomas    Duke    6-8   225.0
203085    A. Rivers    Duke    6-4   200.0
203486    M. Plumlee   Duke   6-11   254.0
203552    S. Curry     Duke    6-2   185.0
```

Having to refer to the name of your DataFrame (`dfp.loc[dfp['school'] == 'Duke']` here) multiple times in one line may seem cumbersome at first, but it's a very common thing to do so get used to it.

Any boolean column or boolean operation works.

```
In [12]: from_usa = dfp['country'] == 'USA'

In [13]: players_not_usa = dfp.loc[~from_usa]

In [14]: players_not_usa[['name', 'country', 'height', 'weight']].head()
Out[14]:
```

		name	country	height	weight
player_id					
101133		I. Mahinmi	France	6-11	262.0
101141		E. Ilyasova	Turkey	6-9	235.0
200757		T. Sefolosha	Switzerland	6-6	215.0
200826		J. Barea	Puerto Rico	5-10	180.0
201143		A. Horford	Dominican Republic	6-9	240.0

## Duplicates

A common way to filter data is by removing **duplicates**, or rows that have identical values for all columns. Pandas has built-in functions for this.

```
In [1]: dfp.drop_duplicates(inplace=True)
```

In this case it didn't do anything since we had no duplicates, but it would have if we did. 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 [2]: dfp.drop_duplicates('pos')[['name', 'pos', 'height', 'weight']]
Out[2]:
```

		name	pos	height	weight
player_id					
1713		V. Carter	Guard	6-6	220.0
2544		L. James	Forward	6-9	250.0
2730		D. Howard	Center	6-10	265.0

Note, although it though it might look strange to have

`[['name', 'pos', 'height', 'weight']]`

immediately after `.drop_duplicates('pos')`, it works because the result of `dfp.drop_duplicates(...)` is a DataFrame. We're just immediately using the multiple column bracket syntax after calling `drop_duplicates` to pick out and print certain columns.

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

```
In [3]: dfp.duplicated().head()
Out[3]:
player_id
1713    False
2037    False
2544    False
2546    False
2594    False
```

The `duplicated` method returns a boolean 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. Alternatively, you can just call it on the columns you want to check.

```
In [4]: dfp['pos'].duplicated().head()
Out[4]:
player_id
1713    False
2037    True
2544    False
2546    True
2594    True
```

By default, `duplicated` only identifies the duplicate observation — not the original — so if you have two “D. Green”’s in your data (Danny and Draymond), `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 filtering with modifying columns and update columns *only* for certain rows.

For example, say we want a column “draft description” that summarizes where a player was drafted (“top 5”, “lottery”, “first round”, “second round”).

We would do that using something like this:

```
In [1]: dfp['draft_desc'] = np.nan

In [2]: dfp.loc[dfp['draft_round'] == 1, 'draft_desc'] = 'first round'

In [3]: dfp.loc[(dfp['draft_round'] == 1) & (dfp['draft_number'] <= 14),
           'draft_desc'] = 'lottery'

In [4]: dfp.loc[(dfp['draft_round'] == 1) & (dfp['draft_number'] <= 5),
           'draft_desc'] = 'top 5'

In [5]: dfp.loc[dfp['draft_round'] == 2, 'draft_desc'] = 'second round'

In [6]: dfp.loc[dfp['draft_round'].isnull(), 'draft_desc'] = 'undrafted'
```

We start by creating an empty column, `draft_desc`, filled with missing values (remember, missing values have a value of `np.nan` in Pandas).

Then we go through and use `loc` to pick out only the rows (where `draft_round` and `draft_number` equals what we want) and the one column (`draft_desc`) and assign the correct value to it.

Note Pandas indexing ability is what allows us to assign, e.g., `'second round'` only to observations where `draft_round` equals 2 without worrying that the other observations will be affected.

```
In [7]: dfp[['name', 'school', 'draft_round', 'draft_number',
           'draft_desc']].sample(5)
Out[7]:
          name   ...   draft_desc
player_id
1626156      D. Russell   ...       top 5
1627750      J. Murray   ...       lottery
203521       M. Dellavedova   ...     undrafted
201188       M. Gasol   ...    second round
202340       A. Bradley   ...    first round
```

## The `query` Method is an Alternative Way to Filter

The `loc` method is flexible, powerful and can do everything you need — including letting you update columns only for rows with certain values. But, if you're *only* interested in filtering, there's a less verbose alternative: `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 filter `dfp` so it only includes North Carolina players:

## Learn to Code with Basketball

---

```
In [1]: dfp.query("school == 'North Carolina'").head()
Out[1]:
   first      last       name  ...   team  team2    draft_desc
player_id
1713      Vince     Carter   V. Carter  ...   ATL   NaN    top 5
101107    Marvin  Williams  M. Williams  ...   MIL   NaN    top 5
201961     Wayne Ellington  W. Ellington  ...   NYK   NaN  first round
201980     Danny    Green    D. Green  ...   LAL   NaN second round
202334      Ed      Davis    E. Davis  ...   UTA   NaN    lottery
```

Notice how inside the string we're referring to `school` without quotes, like a variable name. We can refer to any column name like that. String values inside `query` strings (e.g. `'North Carolina'`) still need quotes. If you normally tend to use single quotes for strings, it's good practice to wrap all your `query` strings in double quotes (e.g. `"school == 'North Carolina'"`) so that they work well together.

We can also call boolean columns directly:

```
In [2]: dfp['school_in_kentucky'] = dfp['school'] == 'Kentucky'

In [3]: dfp.query("school_in_kentucky").head()
Out[3]:
   first      last  ...  draft_desc  school_in_kentucky
player_id
200765    Rajon     Rondo  ...  first round                True
202335    Patrick  Patterson  ...    lottery                True
202339     Eric     Bledsoe  ...  first round                True
202688    Brandon    Knight  ...    lottery                True
203076    Anthony    Davis  ...      top 5                True
```

Another thing we can do is use basic Pandas functions. For example, to filter on whether `draft_round` is missing:

```
In [4]: dfp.query("draft_round.isnull()")[['name', 'school',
                                             'draft_round', 'draft_number']].head()
Out[4]:
          name        school  draft_round  draft_number
player_id
200826    J. Barea  Northeastern        NaN        NaN
201229    A. Tolliver    Creighton        NaN        NaN
202066    G. Temple Louisiana State        NaN        NaN
202083    W. Matthews    Marquette        NaN        NaN
202397    I. Smith     Wake Forest        NaN        NaN
```

Note: if you're getting an error here, 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 how to filter our data, i.e. take a subset of rows. 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 team game data into a DataFrame named `dftg`. You'll use it for the rest of the problems in this section.

**3.3.2** Make smaller DataFrame with just Chicago Bulls games and only the columns: `'team'`, `'date'`, `'pts'`, `'fgm'`, `'fga'`. Do it two ways: (1) using the `loc` syntax, and (b) another time using the `query` syntax. Call them `dftg_chi1` and `dftg_chi2`.

**3.3.3** Make a DataFrame `dftg_no_chi` with the same columns that is everyone EXCEPT the Bulls.

### 3.3.4

- Are there performances where teams had the exact same shooting performance (ignoring free throws)? How many?
- Divide `dftg` into two separate DataFrames `dftg_fg_dup` and `dftg_fg_no_dup`, one with duplicates (by field goals + threes made and attempted) one without.

**3.3.5** Add a new column to `dftg` called `'three_pt_desc'` with the values:

- `'great'` for teams shooting more than 50% from three
- `'brutal'` for teams shooting 25% or less from three
- missing otherwise

**3.3.6** Make a new DataFrame with only observations for which `'three_pt_desc'` is missing. Do this with both the (a) loc and (b) query syntax. Call them `dftg_no_desc1` and `dftg_no_desc2`.

## 4. Change Granularity

WHERE WE ARE: *There are five main things you can do with Pandas DataFrames. This is number four, granularity.*

*Note the examples for this section are in the file `03_04_granularity.py`. We'll pick up right after you've loaded the DataFrame `shots` into your REPL and done some light processing.*

The fourth thing to do with DataFrames is change the granularity.

Remember: data is a collection of structured information. Granularity is the level your collection is at. Our player data is at the player level; each row in is one player. In the `shots` data we've loaded each row represents a one shot.

### Ways of Changing Granularity

Changing the granularity of your data is a very common thing to do. There are two ways to do it.

1. **Grouping** — sometimes called **aggregating** — involves going from fine grained data (e.g. shot) to less fine grained data (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 on any particular shot.
2. **Stacking or unstacking** — sometimes called **reshaping** — is less common than grouping. It involves no loss of information, essentially because it crams data that was formerly in unique rows into separate columns. For example: say I have a dataset of points at the *player-game* level. I could move things around so my data is one line for every *player*, but now with 82 separate columns (game 1's points, game 2's, etc), one for each game.

### Grouping

Aggregating data to be less granular via grouping is something data scientists do all the time. Examples: going from shot to game data or from player-game to player-season data.

Grouping necessarily involves some function that says *how* your data gets to this less granular state.

So if we have have shot level data with information about whether a shot went in or not (i.e. a 1 if the shot went in, 0 if not) and wanted to group it to the game level we could take the:

- *sum* to get total made shots for the game
- *average* to get shooting percentage
- *count* to get total number of shot attempts

Let's use our shot data to look at some examples.

## groupby

Pandas handles grouping via the `groupby` function.

```
In [1]: shots.groupby('game_id').sum().head()
Out[1]:
      dist  value  made     x  ...    period  min_left  sec_left
game_id
21900002   2366    396    79 -1233  ...        421       919      4537
21900008   2156    405    70 -1138  ...        435       917      5001
21900054   2343    422    71   689  ...        445       979      5036
21900063   2110    371    83 -1407  ...        405       935      4493
21900110   1979    385    74  -821  ...        414       939      4708
```

This gives us a DataFrame where every column is summed (because we called `.sum()` at the end) 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 `sum` gives us the sum of every column. Usually we'd only be interested in a subset of variables, maybe field goals made or attempted for sum.

```
In [2]: fg_cols = ['fgm', 'fga', 'fg3m', 'fg3a']

In [3]: shots.groupby('game_id').sum()[fg_cols].head()
Out[3]:
      fgm  fga  fg3m  fg3a
game_id
21900002   79  166    24    64
21900008   70  175    14    55
21900054   71  181    14    60
21900063   83  162    19    47
21900110   74  169    16    47
```

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

```
In [4]:  
shots.groupby('game_id').agg({  
    'value': 'mean',  
    'fgm': 'sum',  
    'fga': 'sum',  
    'fg3m': 'sum',  
    'fg3a': 'sum',  
}).head()  
  
--  
Out[4]:
```

	value	fgm	fga	fg3m	fg3a
game_id					
21900002	2.385542	79	166	24	64
21900008	2.314286	70	175	14	55
21900054	2.331492	71	181	14	60
21900063	2.290123	83	162	19	47
21900110	2.278107	74	169	16	47

Note how the new grouped by columns have the same names as the original, non-aggregated versions. But after a `groupby` that name may no longer be what we want. For instance, `value` isn't the best name for the average shot value, which is what this column tells us post grouping.

To fix this, Pandas let's you pass new variable names to `agg` as keywords, with the values being variable, function **tuple** pairs (for our purposes, a tuple is like a list but with parenthesis). The following code is the same as the above, but renames `value` to `ave_value` and adds in some information about shot distance.

```
In [5]:  
shots.groupby('game_id').agg(  
    ave_value = ('value', 'mean'),  
    ave_dist= ('dist', 'mean'),  
    max_dist = ('dist', 'max'),  
    fgm = ('fgm', 'sum'),  
    fga = ('fga', 'sum'),  
    fg3m = ('fg3m', 'sum'),  
    fg3a = ('fg3a', 'sum')).head()  
  
Out[5]:
```

	ave_value	ave_dist	max_dist	fgm	fga	fg3m	fg3a
game_id							
21900002	2.385542	14.253012	33	79	166	24	64
21900008	2.314286	12.320000	53	70	175	14	55
21900054	2.331492	12.944751	29	71	181	14	60
21900063	2.290123	13.024691	72	83	162	19	47
21900110	2.278107	11.710059	27	74	169	16	47

Note you're no longer passing a dictionary, instead `agg` takes arguments, each in a `new_variable`

```
= ('old_variable', 'function-as-string-name') format.
```

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

```
In [6]:  
shots_per_pg = shots.groupby(['game_id', 'player_id']).agg(  
    name = ('name', 'first'),  
    fgm = ('fgm', 'sum'),  
    fga = ('fga', 'sum'),  
    fg3m = ('fg3m', 'sum'),  
    fg3a = ('fg3a', 'sum'),  
    ave_dist= ('dist', 'mean'),  
    max_dist = ('dist', 'max'),  
)  
  
In [7]: shots_per_pg.head()  
Out[7]:
```

		name	fgm	fga	fg3m	fg3a	ave_dist	max_dist
game_id	player_id							
21900002	2544	L. James	7	19	1	5	11.473684	33
	2730	D. Howard	1	3	0	0	1.666667	4
101150		L. Williams	8	14	1	4	17.285714	30
201162		J. Dudley	2	2	2	2	24.500000	25
201580		J. McGee	2	3	0	0	0.333333	1

## 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 is indexed by two or more variables. In the example above, our index is a combination of `game_id` and `player_id`.

You can still use the `loc` method with multi-level indexed DataFrames, but you need to pass it a tuple (again, like a list but with parenthesis):

```
In [8]: shots_per_pg.loc[((21900002, 2544), (21900008, 201143))]  
Out[8]:
```

		name	fgm	fga	fg3m	fg3a	ave_dist	max_dist
game_id	player_id							
21900002	2544	L. James	7	19	1	5	11.473684	33
21900008	201143	A. Horford	5	13	1	6	13.769231	25

I personally find multilevel indexes unwieldy and avoid them when I can by calling the `reset_index` method immediately after running a multi-column `groupby`.

However there are situations where multi-level indexes are the only way to do what you want, like in the second way to aggregate data, which is 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 it, we're just moving data from columns to rows (stacking) or vis versa (unstacking).

For example, let's quick get number of shots made by team and value (2 or 3):

```
In [1]:  
sv = (shots  
      .groupby(['team', 'value'])['made']  
      .sum()  
      .reset_index())  
  
In [2]: sv.head()  
Out[2]:  
   team  value  made  
0  ATL      2    180  
1  ATL      3     59  
2  BKN      2    134  
3  BKN      3     46  
4  BOS      2    171
```

This data is at the *team and shot value* level. Each row is a team-value combination (Atlanta, 2 pointers), and the `made` column tells us the number of that type of shot that team made (180).

But say instead we want this data to be at the *team* level and have two separate columns for number of 2 and 3 pointers. Then we'd do:

```
In [3]: sv_reshaped = sv.set_index(['team', 'value']).unstack()  
  
In [4]: sv_reshaped.head()  
Out[4]:  
      made  
value  2  3  
team  
ATL    180  59  
BKN    134  46  
BOS    171  88  
CHA     49  27  
CHI     66  29
```

This move doesn't cost us any information. Initially, we find made shots for a particular team and shot type by looking in the right row.

After we unstack it, we can find made shots for a particular team and shot type by first finding the team's row (Atlanta), then finding the column for the right type (made 2).

This lets us do things like calculate all shot totals:

```
In [5]: total_made = sv_reshaped.sum(axis=1)

In [6]: total_made.head()
Out[6]:
team
ATL    239
BKN    180
BOS    259
CHA     76
CHI     95
dtype: int64
```

Or figure out % of shots made that were 3s.

```
In [7]: sv_reshaped.columns = [2, 3]

In [8]: (sv_reshaped[3]/total_made).head()
Out[8]:
team
ATL    0.246862
BKN    0.255556
BOS    0.339768
CHA    0.355263
CHI    0.305263
```

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

```
In [9]: sv_reshaped.stack().head()
Out[9]:
team
ATL    2    180
      3    59
BKN    2    134
      3    46
BOS    2    171
```

## Review

In this section we learned 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 team game data into a DataFrame named `dftg`.
- b) Figure out the average number of points each team scored per game.
- c) Figure out the portion of games each team scored 100 points or more.
- d) How many teams scored 150 or more points in at least one game?
- e) Run `dftg.groupby('date')[['team_id']].count()`, compare it with `dftg.groupby(['date'])[['team_id']].sum()`.

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?

### 3.4.3

- a) Make a new DataFrame `dftg2` that's at the team and win-loss level and includes the following info: team id, wl, then average: pts, fgm, fga, fg3m, fg3a and sum of: wl. These columns should be called ave\_pts, ave\_fgm, ave\_fga, ave\_fg3m, ave\_fg3a, and n, respectively.
- b) Because you grouped by more than one column, note `dftg2` has a multi-index. Make those regular columns.
- c) How many teams averaged more than 110 points in their losses?
- d) Make a new DataFrame `dftg3`, that's just like `dftg2` except it uses `team` instead of `team_id`. Keep the multi-index.
- e) How does stacking or unstacking affect the amount of information in your data? Try running `dftg3.unstack()` what's it doing?

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

## 5. Combining Two or More DataFrames

WHERE WE ARE: *There are five main things you can do with Pandas DataFrames. This section is about the fifth, which is combining them.*

*The examples for this section are in the file `03_05_combine.py`. We'll pick up right after loading the `pg`, `games` and `player` DataFrames.*

The last thing we need to know how to do with DataFrames is combine them.

Typically, by *combining* we mean sticking DataFrames together side by side, like books on a shelf. This is also called **merging**, **joining**, or **horizontal concatenation**.

The alternative (which we'll also learn) is stacking DataFrames on top of each other, like a snowman. This is called **appending** or **vertical concatenation**.

### Merging

There are three questions to ask yourself when merging DataFrames. They are:

1. What columns are you joining on?
2. Are you doing a one to one (1:1), one to many (1:m or m:1), or many to many (m:m) type join?
3. What are you doing with unmatched observations?

Let's go over each of these.

#### Merge Question 1. What columns are you joining on?

Say we want to analyze whether guys shoot better when they're playing at home vs playing away. We can't do it at the moment because we don't have a home or away column in our player-game data.

	name	team	game_id	fga	ast	stl	pts
0	L. James	LAL	21900002	19	8	1	18
1	D. Howard	LAL	21900002	3	1	0	3
2	L. Williams	LAC	21900002	14	7	1	21
3	J. Dudley	LAL	21900002	2	0	0	6
4	J. McGee	LAL	21900002	3	0	0	4

Recall our tabular data basics: each *row* is some item in a collection, and each *column* some piece of information.

Here, rows are player-game combinations (Lebron James, game id 21900002). And our information is: player, team, game id, field goal attempts, steals and points. Information we *don't* have: whether the player was home or away.

## Learn to Code with Basketball

---

That information is in a different table, `games`:

	game_id	date	home	away
0	21900001	2019-10-22	TOR	NOP
1	21900002	2019-10-22	LAC	LAL
2	21900003	2019-10-23	CHA	CHI
3	21900004	2019-10-23	IND	DET
4	21900005	2019-10-23	ORL	CLE

We need to link the home and away information in `games` with the player statistics in `pg`. The end result will look something like this:

	name	team	game_id	fga	ast	stl	pts	home	away
0	L. James	LAL	21900002	19	8	1	18	LAC	LAL
1	D. Howard	LAL	21900002	3	1	0	3	LAC	LAL
2	L. Williams	LAC	21900002	14	7	1	21	LAC	LAL
3	J. Dudley	LAL	21900002	2	0	0	6	LAC	LAL
4	J. McGee	LAL	21900002	3	0	0	4	LAC	LAL

To link these two tables we need information in common. In this example, it's `game_id`. Both the `pg` and `games` DataFrames have a `game_id` column, and in both cases it refers to the same game.

So we can do:

```
In [1]: pd.merge(pg, games[['game_id', 'home', 'away']],
               on='game_id').head()
Out[1]:
      name  fgm  fga  fg_pct  pts  ...  home  away
0  L. James    7   19  0.368   18  ...  LAC  LAL
1  D. Howard   1    3  0.333    3  ...  LAC  LAL
2  L. Williams   8   14  0.571   21  ...  LAC  LAL
3  J. Dudley    2    2  1.000    6  ...  LAC  LAL
4  J. McGee     2    3  0.667    4  ...  LAC  LAL
```

Again, this works because `game_id` is in (and means the same thing) in both tables. Without it, Pandas would have no way of knowing which `pg` row was connected to which `game` row.

Here, we're explicitly telling Pandas to link these tables on `game_id` with the `on='game_id'` keyword argument. This argument is optional. If we leave it out, Pandas will default to using the columns the two DataFrames have in common (it'll try to merge on all the columns with the same name).

## Merging is Precise

Keep in mind that to merge successfully, the values in the columns you're linking need to be *exactly* the same.

If you're merging on the `name` column one DataFrame has "J.R. Smith" and the other has "JR Smith" or just "J.Smith" or "Earl Joseph Smith", they won't be merged.

It's your job to modify one or both of them (using the column manipulation functions we talked about earlier) to make them the same so you can combine them properly.

If you're new to working with data you 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 merging on `name` there are two '`D. Robinson`' (Duncan and David), it will lead to unexpected behavior.

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 works too. For example, say we have separate two and three point DataFrames:

```
In [2]: df2 = pg[['game_id', 'player_id', 'fg2m', 'fg2a']]  
In [3]: df3 = pg[['game_id', 'player_id', 'fg3m', 'fg3a']]
```

Both of which are at the player-game level. To combine them by `player_id` and `game_id` we just pass a list of column names to `on`.

```
In [4]: combined = pd.merge(df2, df3, on=['player_id', 'game_id'])
```

## Merge Question 2. Are you doing a 1:1, 1:many (or many:1), or many:many join?

After deciding which columns you're merging on, the next step is figuring out whether you're doing a one-to-one, one-to-many, or many-to-many merge.

In the previous example, we merged two DataFrames together on `player_id` and `game_id`. Neither DataFrame had duplicates on these columns. That is, they each had *one* row with a `game_id` of 21900002 and `player_id` of 2544, *one* row with a `game_id` of 21900002 and `player_id` of 2730, etc. Linking them up was a **one-to-one** (1:1) merge.

One-to-one merges are straightforward; all DataFrames involved (the two we're merging, plus the final, merged product) are at the same level of granularity. This is *not* the case with one-to-many (or many-to-one, same thing) merges.

Say we're working with our `combined` DataFrame from above. Recall this was *combined* two and three point stats at the player-game level.

	game_id	player_id	fg2m	fg2a	fg3m	fg3a
0	21900002	2544	6	14	1	5
1	21900002	2730	1	3	0	0
2	21900002	101150	7	10	1	4
3	21900002	201162	0	0	2	2
4	21900002	201580	2	3	0	0

Now we want to add back in each player's name. We do this by merging it with the player table:

	player_id	name	team	height	weight
0	1713	V. Carter	ATL	6-6	220.0
1	2037	J. Crawford	BKN	6-5	200.0
2	2544	L. James	LAL	6-9	250.0
3	2546	C. Anthony	POR	6-7	238.0
4	2594	K. Korver	MIL	6-7	212.0

The column we're merging on is `player_id`. Since the `player` data is at the player level, it has one row per `player_id`. There are no duplicates:

```
In [5]: player['player_id'].duplicated().any()
Out[5]: False
```

That's *not* true for `combined`, which is at the player-game level. Here, each player shows up multiple times: once in his first game, another time for his second, etc.

```
In [6]: combined['player_id'].duplicated().any()
Out[6]: True
```

In other words, every *one* `player_id` in our `player` table is being matched to *many* `player_ids` in the `combined` table. This is a **one-to-many** merge.

```
In [7]: pd.merge(combined, player[['player_id', 'name', 'height',
                                'weight']]).head()
Out[7]:
   game_id  player_id  fg2m  fg2a  fg3m  fg3a      name  height  weight
0  21900002      2544     6    14     1     5  L. James   6-9  250.0
1  21900054      2544     6    10     2     5  L. James   6-9  250.0
2  21900329      2544     7    14     4     9  L. James   6-9  250.0
3  21900861      2544    12    16     5    11  L. James   6-9  250.0
4  21901232      2544     4    12     2     7  L. James   6-9  250.0
```

(Note that even though we left out the `on='player_id'` keyword argument, Pandas defaulted to it since `player_id` was the one column the two tables had in common.)

One-to-many joins come up often, especially when data is efficiently stored. It's not necessary 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 our `player` table when we need it.

Finally, although it's technically possible to do **many-to-many** (m:m) joins, in my experience this is almost always done unintentionally<sup>3</sup>, usually when merging on columns with inadvertent duplicates.

Note how the Pandas `merge` command for 1:1 and 1:m merges looks exactly the same. Pandas automatically figures out which type you want depending on what your data looks like.

You can pass the type of merge you're expecting using the `validate` keyword. If you do that, Pandas will throw an error if the merge isn't what you say it should be. It's a good habit to get into. It's much better to get an error right away than it is to continue working with data that isn't actually structured the way you thought it was.

Let's try that last `combined` to `player` example using `validate`. We know this isn't really a 1:1 merge, so if we set `validate='1:1'` we should get an error.

```
In [8]: pd.merge(combined, player, validate='1:1')
...
MergeError: Merge keys are not unique in left dataset; not a one-to-one
merge
```

Perfect.

### Merge Question 3. What are you doing with unmatched observations?

So you know which columns you're merging on, and whether you're doing a 1:1 or 1:m join. The final factor to consider: what are you doing with unmatched observations?

Logically, there's no requirement that two tables *have* to include information about the exact same observations.

To demonstrate, let's remake our two and three point data, keeping observations with at least one attempt.

```
In [1]: df2 = pg.loc[pg['fg2a'] > 0,
                   ['game_id', 'player_id', 'fg2m', 'fg2a']]
In [2]: df3 = pg.loc[pg['fg3a'] > 0,
                   ['game_id', 'player_id', 'fg2m', 'fg2a']]
```

So `df2` only has player-games where the player shot at least one two pointer; `df3` has only threes.

```
In [3]: df2.shape
Out[3]: (1894, 4)

In [4]: df3.shape
Out[4]: (1680, 4)
```

<sup>3</sup>There are rare situations where something like this might be useful that we'll touch on more in the SQL section.

Many players in the two point table (not all of them) will have some three point stats. Some players in the three point table won't be in the two point table.

When you merge these, Pandas defaults to keeping only the observations in *both* tables.

```
In [5]: comb_inner = pd.merge(df2, df3)  
In [6]: comb_inner.shape  
Out[6]: (1548, 6)
```

So `comb_inner` *only* has the player-game's where the player had at least one two *and* one three point attempt.

Alternatively, we can keep everything in the left (`df2`) or right (`df3`) table by passing '`left`' or '`right`' to the `how` argument.

```
In [7]: comb_left = pd.merge(df2, df3, how='left')  
In [8]: comb_left.shape  
Out[8]: (1894, 6)
```

Where "left" and "right" just denote the order we passed the DataFrames into `merge` (first one is left, second right).

Now `comb_left` has everyone who had at least one two point attempt, whether they shot any threes or not. Rows in `df2` that weren't in `df3` get missing values for `fg3m` and `fg3a`.

```
In [9]: comb_left.head()  
Out[9]:  
   game_id  player_id  fg2m  fg2a  fg3m  fg3a  
0  21900002       2544     6    14   1.0   5.0  
1  21900002       2730     1     3   NaN   NaN  
2  21900002      101150     7    10   1.0   4.0  
3  21900002      201580     2     3   NaN   NaN  
4  21900002      201976     1     2   0.0   5.0
```

We can also do an "outer" merge, which keeps everything: matches, and non-matches from both left and right tables.

One thing I find helpful when doing non-inner joins is to include the `indicator=True` keyword argument. This adds a column `_merge` indicating whether the observation was in the left DataFrame, right DataFrame, or both.

```
In [10]: comb_outer = pd.merge(df2, df3, how='outer', indicator=True)

In [11]: comb_outer.shape
Out[11]: (2026, 7)

In [12]: comb_outer['_merge'].value_counts()
Out[12]:
both           1548
left_only      346
right_only     132
Name: _merge, dtype: int64
```

This tells us that — out of the 2114 player-game observations in our sample — in 2026 the player had at least one shot attempt. Out of these 2026, 346 only had two point attempts (no threes), 132 only had three point attempts (no twos). The rest (1548) had at least one of each.

## More on pd.merge

All of our examples so far have been neat merges on identically named id columns, but that won't always be the case. Often the columns you're merging on will have different names in different DataFrames.

To demonstrate, let's modify the two and three point tables we've been working with, renaming `player_id` to `fg2_shooter_id` and `fg3_shooter_id` respectively.

```
In [1]: df2.columns = ['game_id', 'fg2_shooter_id', 'fg2m', 'fg2a']

In [2]: df3.columns = ['game_id', 'fg3_shooter_id', 'fg3m', 'fg3a']
```

(Recall 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 `fg2_shooter_id` in `df2` and `fg3_shooter_id` in `df3`. What to do? Simple, just use the `left_on` and `right_on` arguments instead of `on`.

```
In [3]:
pd.merge(df2, df3, left_on=['game_id', 'fg2_shooter_id'],
         right_on=['game_id', 'fg3_shooter_id']).head()

Out[3]:
   game_id  fg2_shooter_id  fg2m  fg2a  fg3_shooter_id  fg3m  fg3a
0  21900002          2544     6    14          2544     1     5
1  21900002         101150     7    10         101150     1     4
2  21900002         201976     1     2         201976     0     5
3  21900002         201980     3     5         201980     7     9
4  21900002         202340     1     2         202340     2     5
```

Sometimes you might want attach one of the DataFrames you're merging by its index. That's also no problem.

Here's an example. Say we want to find each player's maximum number of three point shots and attempts:

```
In [4]:  
max_fg3 = (df3  
    .groupby('fg3_shooter_id')  
    .agg(max_fg3m = ('fg3m', 'max'),  
        max_fg3a = ('fg3a', 'max')))
```

This is a `groupby` on `fg3_shooter_id`, which results in a new DataFrame where the index is `fg3_shooter_id`.

```
In [5]: max_fg3.head()  
Out[5]:  
          max_fg3m  max_fg3a  
fg3_shooter_id  
1713              2          5  
2037              1          2  
2544              5         11  
2546              3          6  
2594              3          5
```

What if we want to add this back into our original, `df3` DataFrame? They both have `fg3_shooter_id`, but one is an index, one a regular column. No problem. Instead of `right_on`, we pass `right_index=True`.

```
In [6]: pd.merge(df3, max_fg3, left_on='fg3_shooter_id',  
                 right_index=True).sample(5)  
Out[6]:  
      game_id  fg3_shooter_id  fg3m  fg3a  max_fg3m  max_fg3a  
1697  21901285        1629067    1     5          2          5  
1321  21901255        203903     1     9          5          9  
1068  21901231        1628366    0     4          3          8  
1746  21901289        201143     3     4          3          6  
1838  21901295        203506     3     5          3         12
```

## pd.merge() Resets the Index

One thing to be aware of with `merge` is that the DataFrame it returns 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 indexed DataFrames. If that's the case, how is Pandas supposed know which of their indexes to use once they're combined? It doesn't, and so resets the index to the 0, 1, 2, ... default.

But if you're relying on Pandas indexing to automatically align everything for you, and doing some merging along the way, this is something you'll want to watch out for.

### **pd.concat()**

If you're combining DataFrames with the same index, you can use the `concat` function (for concatenate) instead of `merge`.

To try it out, let's make our two and three point DataFrames again, this time setting the index to `game_id` and `player_id`.

```
In [1]:  
df2 = (pg.loc[pg['fg2a'] > 0, ['game_id', 'player_id', 'fg2m', 'fg2a']]  
       .set_index(['game_id', 'player_id']))  
  
In [2]:  
df3 = (pg.loc[pg['fg3a'] > 0, ['game_id', 'player_id', 'fg3m', 'fg3a']]  
       .set_index(['game_id', 'player_id']))
```

So `df2`, for example, looks like this:

```
In [3]: df2.head()  
Out[3]:  
          fg2m  fg2a  
game_id  player_id  
21900002  2544      6    14  
           2730      1     3  
          101150      7    10  
          201580      2     3  
          201976      1     2
```

And concatenating `df2` and `df3` gives us:

```
In [4]: pd.concat([df2, df3], axis=1).head()  
Out[4]:  
          fg2m  fg2a  fg3m  fg3a  
game_id  player_id  
21900002  2544    6.0  14.0    1.0    5.0  
           2730    1.0   3.0    NaN    NaN  
          101150    7.0  10.0    1.0    4.0  
          201580    2.0   3.0    NaN    NaN  
          201976    1.0   2.0    0.0    5.0
```

Note we're passing `concat` a *list* of DataFrames. Lists can contain as many items as you want, and `concat` lets you stick together as many DataFrames as you want. This is different than `merge`, which limits you to two.

For example, maybe we have a free throw DataFrame.

```
In [5]:  
df_ft = (pg.loc[pg['fta'] > 0, ['game_id', 'player_id', 'ftm', 'fta']]  
         .set_index(['game_id', 'player_id']))
```

And want to concatenate all three at once:

```
In [6]: pd.concat([df2, df3, df_ft], axis=1).head()  
Out[6]:  
          fg2m  fg2a  fg3m  fg3a  ftm  fta  
game_id  player_id  
21900002  2544    6.0   14.0   1.0   5.0   3.0   4.0  
           2730    1.0    3.0   NaN   NaN   1.0   1.0  
           101150    7.0   10.0   1.0   4.0   4.0   4.0  
           201580    2.0    3.0   NaN   NaN   NaN   NaN  
           201976    1.0    2.0   0.0   5.0   NaN   NaN
```

Like most Pandas functions, `concat` takes an `axis` argument. When you pass `axis=1`, `concat` sticks the DataFrames together side by side horizontally. Both `merge` and `concat` with `axis=1` provide similar functionality. I usually stick with `merge` for straightforward, two DataFrame joins since it's more powerful, and use `concat` if I need to combine more than two DataFrames.

When `axis=1`, you can tell `concat` how to handle mismatched observations using the `join` argument. Options are '`'inner'` or '`'outer'` (the default). Unlike `merge` there's no '`'left'` or '`'right'`' option, which makes sense because `concat` has to handle more than two DataFrames.

## Combining DataFrames Vertically

When `axis=0` (which it is by default), `concat` sticks the DataFrames together on top of each other, like a snowman. Importantly, `concat` is the *only* way to do this. There's no `axis` equivalent in `merge`.

Let's make some DataFrames to try it out:

```
In [1]: bucks = pg.loc[pg['team'] == 'MIL']  
  
In [2]: magic = pg.loc[pg['team'] == 'ORL']
```

We can see these DataFrames are 72 and 152 rows respectively:

```
In [3]: bucks.shape  
Out[3]: (72, 33)  
  
In [4]: magic.shape  
Out[4]: (152, 33)
```

## Learn to Code with Basketball

---

Now let's use `pd.concat` (with its default `axis=0`) to stack these on top of each other. The resulting DataFrame should be  $72 + 152 = 224$  rows.

```
In [5]: pd.concat([bucks, magic]).shape  
Out[5]: (224, 33)
```

Perfect.

In this case, we *know* `bucks` and `magic` don't have any index values in common (because we just created them using `loc`, which keeps original index), but often that's not the case.

For example, let's reset the indexes on both of these.

```
In [6]: bucks_reset = bucks.reset_index(drop=True)  
In [7]: magic_reset = magic.reset_index(drop=True)
```

That resets each index to the default `0, 1, ...` etc. So now `bucks_reset` looks like this:

```
In [8]: bucks_reset.head()  
Out[8]:  
      name  fgm  fga  fg_pct  pts  ...  dd2  start  fg2m  fg2a  
0   K. Korver    0    4   0.000    1  ...    0  False     0     1  
1   E. Ilyasova   1    2   0.500    3  ...    0  False     0     0  
2   B. Lopez     3    6   0.500    8  ...    0  True      1     3  
3   R. Lopez     1    6   0.167    2  ...    0  False     1     4  
4   G. Hill      6   11   0.545   15  ...    0  False     3     7
```

And when we concatenate `bucks_reset` and `magic_reset` we get:

```
In [9]: pd.concat([bucks_reset, magic_reset]).sort_index().head()  
Out[9]:  
      name  fgm  fga  fg_pct  pts  ...  dd2  start  fg2m  fg2a  
0   K. Korver    0    4   0.0    1  ...    0  False     0     1  
0   D. Augustin   4   10   0.4    8  ...    0  False     4     8  
1   E. Ilyasova   1    2   0.5    3  ...    0  False     0     0  
1   A. Aminu      1    2   0.5    2  ...    0  False     1     1  
2   B. Lopez      3    6   0.5    8  ...    0  True      1     3
```

Now our index has duplicates — one `0` from our Bucks DataFrame, another from the Magic DataFrame, etc. Pandas will technically let us do this, but it's probably not what we want. The solution is to pass `ignore_index=True` to `concat`:

```
In [10]: pd.concat([bucks_reset, magic_reset],  
                  ignore_index=True).sort_index().head()  
Out[10]:  
      name  fgm  fga  fg_pct  pts  ...  dd2  start  fg2m  fg2a  
0   K. Korver    0    4   0.000    1  ...    0  False     0     1  
1   E. Ilyasova   1    2   0.500    3  ...    0  False     0     0  
2   B. Lopez     3    6   0.500    8  ...    0   True     1     3  
3   R. Lopez     1    6   0.167    2  ...    0  False     1     4  
4   G. Hill      6   11   0.545   15  ...    0  False     3     7
```

## 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 covered how `pd.concat` can let you combine two or more DataFrames with the same index left to right via the `axis=1` keyword.

Let's wrap up with a quick summary of the differences between `concat` and `merge`.

### `merge`

- usually what you'll use when you need to combine two DataFrames horizontally
- combines *two* DataFrames per `merge` (you can do more by calling `merge` multiple times)
- *only* combines DataFrames horizontally
- let's you combine DataFrames with different indexes
- is more flexible in handling unmatched observations

### `concat`

- the *only* way to stack DataFrames vertically on top of each other
- combines *any number* of DataFrames
- can combine horizontally (`axis=1`) or vertically (`axis=0`)
- *requires* every DataFrame has the same index
- let's you keep observations in *all* DataFrames (`join='inner'`) or observations in *any* DataFrame (`join='outer'`)

## Exercises

### 3.5.1

- a) Load the three datasets in in `./data/problems/combine1/`.

They contain point (field goal and point info), rebound (offensive and defensive), and defensive (blocks and steals) data.

Combine them: b) using `pd.merge`. c) using `pd.concat`.

Note players are only in the point data if they got at least one point or attempted one field goal. Same with reb.csv and def.csv. Make sure your final combined data includes all players, even if they didn't show up in the data. If a player didn't get any points (or an rebound, steal etc) make sure the number is set to 0.

- d) Which do you think is better here, `pd.merge` or `pd.concat`?

### 3.5.2

- a) Load the three datasets in in `./data/problems/combine2/`. It contains the same data, but split "vertically" by position.
- b) Combine them. Make the index of the resulting DataFrame is the default (0, 1, 2, ... )

### 3.5.3

- a) Load the team data in `./data/teams.csv`.
- b) Write a for loop to save subsets of the data frame for each conference (East and West) 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 — a mini programming language that is separate from Python — 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 read into Pandas. What advantages do databases and SQL give us over that? Let's 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 a single spreadsheet is that things can get unwieldy very quickly. For example, say we're building a model to project number of points a player will score in any given game. This data is at the player and game level, but we might want to include less granular data — information about the *player* (years in league, height, weight) or *game* (home or away, days rest).

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 that way.

Think about it: a player's height 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. Instead of one giant table, it would be more efficient to store this data as:

- One player table with just name, team, position, height, weight, year they came in the league.
- One game table with date, home and away team, number of days rest.
- Another player-game table with ONLY the things that vary at this level: points, rebounds, score, 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 of information necessary is called **normalizing** your data. There are at least two benefits:

First, storing data in one place makes it easier to update it.

Say our initial dataset had the incorrect home and away teams for MIL-CHI, first game of the season. 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.

The other advantage 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 shot data; right now we have mostly shot-specific information in there (distance, coordinates on the floor, made or not). 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 shot in the shot 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, shot etc) in its own csv file. Do things really need to be in a SQL database?

Multiple csv files is better than one giant csv, and honestly I don't care if you want to do it this way (I do it myself for quick, smaller projects).

However, it does mean 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 between data.

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 be using them, 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

SQL is short for Structured Query Language. I always pronounce it “sequel”, but some people say the full “S-Q-L”. It’s not really a fully featured programming language, more of a simple way to describe data you want to load from a database.

It’s also important to note SQL is its own thing, *not* part of Python. We’ll be using it with Pandas and inside Python, but that’s not necessary. Other languages have their own way of interacting with SQL, and some people do most of their work in SQL itself.

## 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<sup>1</sup>. 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. It’s 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 are the same.

Another benefit: when you stick to the 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 make one using SQLite, which is the simplest to set up.

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. This code is in `04_sql.py`.

---

<sup>1</sup>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 basketball datasets we’ll be using.

```
import pandas as pd
from os import path
import sqlite3

# handle directories
DATA_DIR = './data'

# create connection
conn = sqlite3.connect(path.join(DATA_DIR, 'basketball-data.sqlite'))

# load csv data
player_game = pd.read_csv(path.join(DATA_DIR, 'player_game.csv'))
player = pd.read_csv(path.join(DATA_DIR, 'players.csv'))
game = pd.read_csv(path.join(DATA_DIR, 'games.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 columns from your player table, you'd do:

```
SELECT *
FROM player
```

Though not required, a loose SQL convention is to put keywords like **SELECT** and **FROM** in uppercase, as opposed to particular column or table name in lowercase.

Because the job of SQL is to get data into Pandas so we can work with it, we'll be writing all of 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 the 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, and will throw an error if it's not.

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   first     last       name ...   team team2  
0      1713    Vince    Carter    V. Carter ...    ATL  None  
1      2037    Jamal  Crawford  J. Crawford ...    BKN  None  
2      2544   LeBron     James    L. James ...    LAL  None  
3      2546 Carmelo Anthony  C. Anthony ...    POR  None  
4      2594      Kyle   Korver    K. Korver ...    MIL  None
```

In the example file, you'll notice this is how we run all of the queries in this chapter (i.e., as strings inside `read_sql`). And you should stick to that as you run your own queries and work through the book.

However, because the `pd.read_sql(..., conn)` is the same every time, 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, name, birthdate as bday  
FROM player
```

Out [3]:

	player_id	name	bday
0	1713	V. Carter	19770126
1	2037	J. Crawford	19800320
2	2544	L. James	19841230
3	2546	C. Anthony	19840529
4	2594	K. Korver	19810317

You just list the columns you want and separate them by commas. Notice the `birthdate AS bday` part of the `SELECT` statement. Though column is stored in the database as `birthdate`, we're renaming it to `bday` on the fly, which can be useful.

## Filtering

What if we want to *filter* our rows, say — only get back Canadian players? We need to add another clause, a `WHERE`:

In [4]:

```
SELECT player_id, name, birthdate AS bday
FROM player
WHERE country = 'Canada'
```

Out [4]:

	player_id	name	bday
0	202684	T. Thompson	19910313
1	202709	C. Joseph	19910820
2	203482	K. Olynyk	19910419
3	203920	K. Birch	19920928
4	203939	D. Powell	19910720

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 `'Canada'`. Double quotes won't work.

Finally, notice we're filtering on `country` (i.e. we're choosing which rows to return depending on the value they have for `country`), 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 `country` with `'Canada'` for every row), but we don't have to.

We can use logic operators like `OR` and `AND` in our `WHERE` clause too.

In [5]:

## Learn to Code with Basketball

---

```
SELECT player_id, name, pos, country
FROM player
WHERE country != 'USA' AND pos == 'Center'
```

Out [5]:

	player_id	name	pos	country
0	101133	I. Mahinmi	Center	France
1	201143	A. Horford	Center	Dominican Republic
2	201188	M. Gasol	Center	Spain
3	202683	E. Kanter	Center	Turkey
4	202684	T. Thompson	Center	Canada

In [6]:

```
SELECT player_id, name, pos, country, school
FROM player
WHERE country != 'USA' OR school = 'North Carolina'
```

Out [6]:

	player_id	name	pos	country	school
0	1713	V. Carter	Guard	USA	North Carolina
1	101107	M. Williams	Forward	USA	North Carolina
2	101133	I. Mahinmi	Center	France	Pau Orthez
3	101141	E. Ilyasova	Forward	Turkey	FC Barcelona
4	200757	T. Sefolosha	Forward	Switzerland	None

To check whether a column is in a list of values you can use **IN**:

In [7]:

```
SELECT player_id, name, pos, country, school
FROM player
WHERE school IN ('North Carolina', 'Duke')
```

Out [7]:~~~ player\_id name pos country school 0 1713 V. Carter Guard USA North Carolina 1 101107 M. Williams Forward USA North Carolina 2 200755 J. Redick Guard USA Duke 3 201961 W. Ellington Guard USA North Carolina 4 201980 D. Green Guard USA North Carolina ~~

SQL also allows negation:

In [8]:

```
SELECT player_id, name, pos, country, school
FROM player
WHERE school NOT IN ('North Carolina', 'Duke')
```

Out [8]:

	player_id	name	pos	country	school
0	2037	J. Crawford	Guard	USA	Michigan
1	2544	L. James	Forward	USA	St. Vincent-St. Mary HS (OH)
2	2546	C. Anthony	Forward	USA	Syracuse
3	2594	K. Korver	Guard	USA	Creighton
4	2730	D. Howard	Center	USA	SW Atl Christian Academy (GA)

## 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 [9]:

```
SELECT
    player.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 [9]:

	name	pos	team	conference	division
0	V. Carter	Guard	ATL	East	Southeast
1	V. Carter	Guard	ATL	East	Atlantic
2	V. Carter	Guard	ATL	East	Central
3	V. Carter	Guard	ATL	West	Southwest
4	V. Carter	Guard	ATL	East	Central
5	V. Carter	Guard	ATL	West	Southwest
6	V. Carter	Guard	ATL	West	Northwest
7	V. Carter	Guard	ATL	West	Pacific
8	V. Carter	Guard	ATL	West	Southwest
9	V. Carter	Guard	ATL	West	Pacific

It's all Vince Carter, and he's in every division.

The problem is we haven't told SQL how the `player` and `team` tables are related.

## Learn to Code with Basketball

---

When you don't include that information, SQL doesn't try to figure it out or complain and give you an error. Instead it returns a cross join, 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 (`V. Carter`, `Guard`, `ATL`) matched up with the first division in the team table (`Southeast`); then the second (`Atlantic`), and so on.

To make it even clearer, let's add in the `team` column from the `team` table too.

In [10]:

```
SELECT
    player.name,
    player.pos,
    player.team,
    team.conference,
    team.division
FROM player, team
```

Out [10]:

	name	pos	player_team	team_team	conference	division
0	V. Carter	Guard	ATL	ATL	East	Southeast
1	V. Carter	Guard	ATL	BOS	East	Atlantic
2	V. Carter	Guard	ATL	CLE	East	Central
3	V. Carter	Guard	ATL	NOP	West	Southwest
4	V. Carter	Guard	ATL	CHI	East	Central
5	V. Carter	Guard	ATL	DAL	West	Southwest
6	V. Carter	Guard	ATL	DEN	West	Northwest
7	V. Carter	Guard	ATL	GSW	West	Pacific
8	V. Carter	Guard	ATL	HOU	West	Southwest
9	V. Carter	Guard	ATL	LAC	West	Pacific

This makes it clear it's a cross join — every line for Vince Carter (and also every other player once we get through Carter) is getting linked up with every team in the `team` table.

Since we have a 457 row player table and 30 row team table, that means we should get back  $457 \times 30 = 13710$  rows. Sure enough:

```
In [11]: df.shape
Out[11]: (13710, 6)
```

What if we added a third table, say a `conference` table with two rows: one for West, one for East. In that case, each of these 13710 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  $457 \times 30 \times 2 = 27420$  rows.

## Learn to Code with Basketball

---

This is almost never what we want<sup>2</sup> (in fact an inadvertent cross join 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 cross join and getting back a row where Vince Carter plays in the Atlantic, so we have specify a `WHERE` clause to filter and keep only the rows that make sense.

In [11]:

```
SELECT
    player.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	V. Carter	Guard	ATL	East	Southeast
1	J. Crawford	Guard	BKN	East	Atlantic
2	L. James	Forward	LAL	West	Pacific
3	C. Anthony	Forward	POR	West	Northwest
4	K. Korver	Guard	MIL	East	Central

Let's walk through it:

First, SQL is doing the full cross join (with 13710 rows).

Then we have a `WHERE`, so we're saying after the cross join 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 30 separate rows for Vince Carter, to only the one row — where his team in the `player` table (`ATL`) equals `ATL` in the `team` table.

That gives us a table of 457 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]:

---

<sup>2</sup>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.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	V. Carter	Guard		ATL	ATL	East	Southeast
1	J. Crawford	Guard		BKN	BKN	East	Atlantic
2	L. James	Forward		LAL	LAL	West	Pacific
3	C. Anthony	Forward		POR	POR	West	Northwest
4	K. Korver	Guard		MIL	MIL	East	Central

I first learned about this cross join-then `WHERE` framework of conceptualizing SQL queries from the book, [The Essence of SQL](#) by David Rozenshtein. It's a great book, but out of print and going for \$125 (as a 119 page paperback) on Amazon as of this writing. It covers more than just cross join-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 update our `WHERE` clause.

In [13]:

```
SELECT
    player.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	...	game_id	plus_minus	dd2	start
0	V. Carter	Guard	ATL	ATL	...	21900867	-8	0	0
1	V. Carter	Guard	ATL	ATL	...	21900969	6	0	0
2	V. Carter	Guard	ATL	ATL	...	21900295	12	0	0
3	J. Crawford	Guard	BKN	BKN	...	21901256	4	0	0
4	L. James	Forward	LAL	LAL	...	21901232	-4	1	1

This is doing the same as above (`player + team` tables) but also combining it with data from `player_games`. Note, that if we had left off our `WHERE` clause, SQL would have done a full, three table `player*team*player_game` cross join. Vince Carter would be matched with each team, then each of *those* rows would be matched with every row from the player-game table, giving us a  $100 \times 32 \times 716 = 2,291,200$  row result.

Also note the `player_games.*` syntax. This gives us all the columns from that table.

Sometimes table names can get long and unwieldy, especially when working with multiple tables. We could also write above as:

```
SELECT
    p.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.

## Combining Joining and Other Filters

We can also add in other filters, e.g. maybe we want this same query but only the Centers:

```
SELECT
    p.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 == 'Center'
```

## Misc SQL

The basics of `SELECT`, `FROM` and `WHERE` plus the cross join-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.

### 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 all the different calendar days NBA games were on.

In [14]:

```
SELECT DISTINCT season, date
FROM game
```

Out [14]

	season	date
0	2019-20	2019-10-22
1	2019-20	2019-10-23
2	2019-20	2019-10-24
3	2019-20	2019-10-25
4	2019-20	2019-10-26

## UNION

`UNION` lets you stick data on top of each other to form one table. It's similar to `concat` with `axis=0` 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 the data I'm inheriting is in separate tables. I might do:

```
SELECT *
FROM player_data_2020
UNION
SELECT *
FROM player_data_2021
```

## 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 cross join-then `WHERE` framework can only handle inner joins. That is, we'll only keep observations 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 every game. 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 cross join-then `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 using some of these miscellaneous concepts (unions and subqueries), so it might be useful to go through this as a final example.

Feel free to skip if you don't envision yourself using SQL that much and are satisfied doing this in Pandas.

This query gives us points and plus-minus for every player and game in our database, whether or not the player actually played up. We'll walk through it below:

```
1  SELECT a.date, a.team, a.opp, a.first, a.last, b.pts, b.plus_minus
2  FROM
3      (SELECT game_id, date, home as team, away as opp, player_id,
4          player.name, last, first
5      FROM game, player
6      WHERE
7          game.home = player.team
8      UNION
9      SELECT game_id, date, home as team, away as opp, player_id,
10         player.name, last, first
11     FROM game, player
12     WHERE
13         game.away = player.team) AS a
14 LEFT JOIN player_game AS b ON a.game_id = b.game_id AND
15     a.player_id = b.player_id
```

Let's go through it. First, we need a full set of game 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_id, date, home as team, away as opp, player_id, player.name,
       last, first
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 cross join, i.e. we get back a line for every player in every game. So, after the `game, player` cross join here, not only is there a line for Lebron James, game 1, Lakers vs Clippers, but there's a line for Lebron James, game 1, Toronto vs New Orleans too.

This is way more than we need. In the case of Lebron, we want to filter the rows to ones where one of the teams is the Lakers. We do that in our `WHERE` clause. This is all review from above.

The problem is our `game` table is by game, not `team` and `game`. If it were the latter, LAL-LAC game 1 would have two lines, one for the Lakers, one for the Clippers. Instead it's just the one line, with the Clippers in the `home` field, Lakers in `away`.

	game_id	home	away	date
0	21900001	TOR	NOP	2019-10-22
1	21900002	LAC	LAL	2019-10-22
2	21900003	CHA	CHI	2019-10-23
3	21900004	IND	DET	2019-10-23
4	21900005	ORL	CLE	2019-10-23

What this means is, to match up Lebron James to only the Laker games, we're going to have to run this part of the query twice: once when the Lakers are home, another time when they're away, then stick them together with a `UNION` clause

That's what we're doing here:

```
SELECT game_id, date, home as team, away as opp, player_id,
       player.name, last, first
  FROM game, player
 WHERE
       game.home = player.team
UNION
SELECT game_id, date, home as team, away as opp, player_id,
       player.name, last, first
  FROM game, player
 WHERE
       game.away = player.team
```

Above the `UNION` gives us a line for every player's home games (so 42 per player in a normal regular season), 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_game_table` if you want.

In fact, let's do that, giving us:

```
SELECT a.date, a.team, a.opp, a.first, a.last, b.pts, b.plus_minus
  FROM
    full_player_by_game_table AS a
  LEFT JOIN player_game AS b ON a.game_id = b.game_id 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.query("last == 'Redick'")  
Out[15]:  
      date  team  opp  first    last  pts  plus_minus  
0  2019-10-22  TOR  NOP    JJ  Redick  NaN      NaN  
594 2019-10-25  NOP  DAL    JJ  Redick  NaN      NaN  
935 2019-10-26  HOU  NOP    JJ  Redick  NaN      NaN  
1390 2019-10-28  NOP  GSW    JJ  Redick  NaN      NaN  
2002 2019-10-31  NOP  DEN    JJ  Redick  NaN      NaN  
...  ...  ...  ...  ...  ...  ...  ...  
30759 2020-08-06  SAC  NOP    JJ  Redick  NaN      NaN  
31077 2020-08-07  NOP  WAS    JJ  Redick  15.0    3.0  
31372 2020-08-09  NOP  SAS    JJ  Redick  31.0   -6.0  
31844 2020-08-11  SAC  NOP    JJ  Redick  NaN      NaN  
32162 2020-08-13  ORL  NOP    JJ  Redick  NaN      NaN
```

JJ Redick played sporadically in 2019, yet we still have all the rows 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 summarizes points, field goal attempts and field goal makes at the player-game level. Make it only include Central division players. It should have the following columns:

`date, name, fgm, fga, pts`

Rename `pts` to `points` in your query.

### 4.2

- b) Now modify your query so that instead of `name` you have `first` and `last` from the `player` table. Use abbreviations too (`t` for the team table, `pg` for `player_game`, etc).

# 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. Other times you have to get it yourself.

Here we'll learn how to write programs that grab data for you. There are two situations where these programs are especially useful.

The first is when you want to take regular snapshots of data that's changing over time. For example, you could write a program that gets the current injury report for every NBA team and run it every day.

Second is when you need to grab a lot of data at once. Scrapers scale really well. You want detailed game data for every game in a season? Write a function that's flexible enough to get this data for one game. Then run it 1230 times (82 games years \* 30 teams \ 2 since every game has two teams). Copying and pasting that much data would be slow, tedious and error prone.

This chapter covers two ways to get data: web scraping and APIs.

## 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 sites. Once they're up and running, you usually can run them without visiting the site in your browser.

## HTML and CSS

Building web scrapers involves understanding basic HTML + CSS, which — along with JavaScript — is most of the code behind websites today. We'll focus on the minimum required for scraping. So while this section won't teach you how to build your own site, it will make getting data a lot easier.

HTML is a *markup* language, which means it includes both the content you see on the screen (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, e.g. a starting tag of <p> and an ending tag </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 view the HTML tags on a website by right clicking in your browser and selecting 'view source'.

Tags can be nested. The i tag stands for italics, and so:

```
<p><i>This part</i> of the paragraph is in italics.</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 my second paragraph.</p>
```

These ids and classes are there so web designers can specify — separately in a **CSS file** — more rules about how things should look. Maybe the CSS file says intro paragraphs are a larger font. Or paragraphs with the class "main-body" get a different color, 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 which parts of the website to scrape, so it's useful to know what they are.

## Common HTML Tags

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 to 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 analysts, we're usually interested in tabular data, so it's worth exploring HTML tables in more detail.

Tables are good example of nested HTML. Everything is between `table` tags. Inside those, `tr`, `td` and `th` tags denote rows, columns and header columns respectively.

So if we had a table with weekly fantasy points, the HTML for the first two rows (plus the header) might look something like this:

```
<html>
  <table>
    <tr>
      <th>Name</th>
      <th>Date</th>
      <th>Team</th>
      <th>Opp</th>
      <th>Pts</th>
      <th>Reb</th>
    </tr>
    <tr>
      <td>Lebron James</td>
      <td>2019-10-22</td>
      <td>LAL</td>
      <td>LAC</td>
      <td>18</td>
      <td>10</td>
    </tr>
    <tr>
      <td>Giannis Antetokounmpo</td>
      <td>2020-10-06</td>
      <td>MIL</td>
      <td>SAN</td>
      <td>24</td>
      <td>12</td>
    </tr>
  </table>
<html>
```

Note columns (`td` and `th` elements) are always nested inside rows (`tr`).

If you were to save this code in an html file and open it in your browser you'd see:

Name	Date	Team	Opp	Pts	Reb
Lebron James	2019-10-22	LAL	LAC	18	10
Giannis Antetokounmpo	2020-10-06	MIL	SAN	24	12

**Figure 0.1:** Simple HTML Table

## BeautifulSoup

The library BeautifulSoup (abbreviated BS4) is the Python standard for working with HTML. It lets you turn HTML tags into standard data structures like lists and dicts, which you can then put into Pandas.

Let's take our HTML from above, put it in a multi-line string, and load it into BeautifulSoup. Note the following code is in [05\\_01\\_scraping.py](#), and we start from the top of the file.

```
In [1]: from bs4 import BeautifulSoup as Soup

In [2]:

```

Note we're using BeautifulSoup with a regular Python string. BS4 is a *parsing* library. It helps us take a *string* of HTML and turn it into Python data. It's agnostic about where this string comes from. Here, I wrote it out by hand and put it in a file for you. You could use BS4 on this string even if you weren't connected to the Internet, though in practice we almost always get our HTML in real time.

The key type in BeautifulSoup is called a *tag*. Like lists and dicts, tags are containers, which means they hold things. Often they hold other tags.

Once we call `Soup` on our string, every pair of tags in the HTML gets turned into some BS4 tag. Usually they're nested.

For example, our first *row* (the header row) is in a `tr` tag:

## Learn to Code with Basketball

---

```
In [4]: tr_tag = html_soup.find('tr')

In [5]: tr_tag
Out[5]:
<tr>
<th>Name</th>
<th>Date</th>
<th>Team</th>
<th>Opp</th>
<th>Pts</th>
<th>Reb</th>
</tr>

In [6]: type(tr_tag)
Out[6]: bs4.element.Tag
```

(Note: here the `find('tr')` method “finds” the first `tr` tag in our data and returns it.)

We could also have a tag object that represents the whole table.

```
In [7]: table_tag = html_soup.find('table')

In [8]: table_tag
Out[8]:
<table>
<tr>
<th>Name</th>
<th>Date</th>
<th>Team</th>
<th>Opp</th>
<th>Pts</th>
<th>Reb</th>
</tr>
...
<tr>
<td>Giannis Antetokounmpo</td>
<td>2020-10-06</td>
<td>MIL</td>
<td>SAN</td>
<td>24</td>
<td>12</td>
</tr>
</table>

In [9]: type(table_tag)
Out[9]: bs4.element.Tag
```

Or just the first `td` element.

```
In [10]: td_tag = html_soup.find('td')

In [11]: td_tag
Out[11]: <td>Lebron James</td>

In [12]: type(td_tag)
Out[12]: bs4.element.Tag
```

They're all tags. In fact, the whole page — all the zoomed out HTML — is just one giant `html` tag.

## Simple vs Nested Tags

I've found it easiest to mentally divide tags into two types. BeautifulSoup doesn't distinguish between these, so this isn't official terminology, but it's helped me.

### Simple Tags

Simple tags are BS4 tags with just text inside. No other, nested tags. Our `td_tag` above is an example.

```
In [13]: td_tag
Out[13]: <td>Lebron James</td>
```

On simple tags, the key attribute is `string`. This returns the data inside.

```
In [14]: td_tag.string
Out[14]: 'Lebron James'
```

Technically, `string` returns a BeautifulSoup string, which carries around a bunch of extra data. It's good practice to convert them to regular Python strings like this:

```
In [15]: str(td_tag.string)
Out[15]: 'Lebron James'
```

### Nested Tags

Nested BS4 tags contain other tags. The `tr`, `table` and `html` tags above were all nested.

The most important method for nested tags is `find_all`. It takes the name of an HTML tag ('`tr`', '`p`', '`td`' etc) and returns all the matching sub-tags in a list.

So to find all the `th` tags in our first row:

```
In [16]: tr_tag.find_all('th')
Out[16]:
[<th>Name</th>,
 <th>Date</th>,
 <th>Team</th>,
 <th>Opp</th>,
 <th>Pts</th>,
 <th>Reb</th>]
```

Again, nested tags contain other tags. These sub-tags themselves can be simple or nested, but either way `find_all` is how you access them. Here, calling `find_all('th')` returned a list of three simple `th` tags. Let's call `string` to pull their data out.

```
In [17]: [str(x.string) for x in tr_tag.find_all('th')]
Out[17]: ['Name', 'Date', 'Team', 'Opp', 'Pts', 'Reb']
```

Note the list comprehension. Scraping is another situation where basic Python comes in handy.

### Other notes on `find_all` and nested tags.

First, `find_all` works *recursively*, which means it searches multiple levels deep. So we could find all the `td` tags in our `table`, even though they're in multiple rows.

```
In [18]: all_td_tags = table_tag.find_all('td')
```

The result is in one flat list:

```
In [19]: all_td_tags
Out[19]:
[<td>Lebron James</td>,
 <td>2019-10-22</td>,
 <td>LAL</td>,
 <td>LAC</td>,
 <td>18</td>,
 <td>10</td>,
 <td>Giannis Antetokounmpo</td>,
 <td>2020-10-06</td>,
 <td>MIL</td>,
 <td>SAN</td>,
 <td>24</td>,
 <td>12</td>]
```

Second, `find_all` is a *method* that you call on a particular tag. It only searches in the tag its called on. So, while calling `table_tag.find_all('td')` returned data on Lebron *and* Giannis (because

they're both in the table), calling `tr_tag.find_all('td')` on just the last, single row returns data for Giannis only.

```
In [20]: all_rows = table_tag.find_all('tr')

In [21]: first_data_row = all_rows[1] # all_rows[0] is header

In [22]: first_data_row.find_all('td')
Out[22]:
[<td>Lebron James</td>,
 <td>2019-10-22</td>,
 <td>LAL</td>,
 <td>LAC</td>,
 <td>18</td>,
 <td>10</td>]
```

Third, you can search multiple tags by passing `find_all` a *tuple* (for our purposes a tuple is a list with parenthesis instead of brackets) of tag names.

```
In [23]: all_td_and_th_tags = table_tag.find_all(('td', 'th'))

In [24]: all_td_and_th_tags
Out[24]:
[<th>Name</th>,
 <th>Date</th>,
 <th>Team</th>,
 <th>Opp</th>,
 <th>Pts</th>,
 <th>Reb</th>,
 <td>Lebron James</td>,
 <td>2019-10-22</td>,
 <td>LAL</td>,
 <td>LAC</td>,
 <td>18</td>,
 <td>10</td>,
 <td>Giannis Antetokounmpo</td>,
 <td>2020-10-06</td>,
 <td>MIL</td>,
 <td>SAN</td>,
 <td>24</td>,
 <td>12</td>]
```

Finally, remember `find_all` can return lists of both simple and nested tags. If you get back a simple tag, you can run `string` on it:

```
In [25]: [str(x.string) for x in all_td_tags]
Out[25]:
['Lebron James',
 '2019-10-22',
 'LAL',
 'LAC',
 '18',
 '10',
 'Giannis Antetokounmpo',
 '2020-10-06',
 'MIL',
 'SAN',
 '24',
 '12']
```

But if you get back a list of nested tags, you'll have to call `find_all` again.

```
In [26]: all_rows = table_tag.find_all('tr')
In [27]: list_of_td_lists = [x.find_all('td') for x in all_rows[1:]]

In [28]: list_of_td_lists
Out[28]:
[[<td>Lebron James</td>,
 <td>2019-10-22</td>,
 <td>LAL</td>,
 <td>LAC</td>,
 <td>18</td>,
 <td>10</td>],
 [<td>Giannis Antetokounmpo</td>,
 <td>2020-10-06</td>,
 <td>MIL</td>,
 <td>SAN</td>,
 <td>24</td>,
 <td>12</td>]]
```

## Remember the ABA - Web Scraping Example

Note the examples for this section are in the file `05_02_aba.py`. We'll pick up from the top of the file.

Let's build a scraper to get some statistics from *Remember the ABA*, an American Basketball Association website. For the unfamiliar, the ABA was a rival basketball league that spun up in the late 1960s before merging with the NBA in 1976. We'll get data on the ABA's top 10 all time scorers and put it in a DataFrame.

We'll start with the imports

```
from bs4 import BeautifulSoup as Soup
import pandas as pd
import requests
from pandas import DataFrame
```

Besides BeautifulSoup, we're also importing the `requests` library, which we'll use to programmatically visit Remember the ABA. We're going to put our final result in a DataFrame, so we've imported that too.

After you've loaded those imports in your REPL, the first step is using `requests` to visit the URL and store the raw HTML it returns:

```
In [1]: response = requests.get('http://www.remembertheaba.com/
ABAStatistics/ABATop20.html')
```

Let's look at (part of) this HTML.

```
In [2]: print(response.text)
Out[2]:
<tr>
    <td align="left" width="13"><font size="1">40</font></td>
    <td align="left" width="91"><font size="1">Byron Beck</font></td>
    <td width="15" align="left"><font size="1">8353</font></td>
    <td align="left" width="15"><font size="1">694</font></td>
    <td align="left" width="25"><font size="1">18717</font></td>
    <td align="left" width="22"><font size="1">3574</font></td>
    <td align="left" width="25"><font size="1">7049</font></td>
    <td align="left" width="21"><font size="1">.507</font></td>
    <td align="left" width="21"><font size="1">13</font></td>
    <td align="left" width="20"><font size="1">44</font></td>
    <td align="left" width="20"><font size="1">.295</font></td>
    <td align="left" width="21"><font size="1">1192</font></td>
    <td align="left" width="20"><font size="1">1471</font></td>
    <td align="left" width="20"><font size="1">.810</font></td>
    <td width="25">-&nbsp;</td>
    <td align="left" width="24"><font size="1">5165</font></td>
    <td align="left" width="20"><font size="1">945</font></td>
    <td align="left" width="20"><font size="1">2208</font></td>
    <td width="12">-&nbsp;</td>
    <td align="left" width="15"><font size="1">155</font></td>
    <td align="left" width="20"><font size="1">933</font></td>
    <td align="left" width="20"><font size="1">42</font></td>
    <td align="left" width="19"><font size="1">12.0</font></td>
    <td width="16"><font size="1">36</font>&nbsp;</td>
    <td align="left" width="44"><font size="1">67-76</font></td>
</tr>
```

The `text` attribute of `response` is a string with all the HTML on this page. This is just a small snippet of what we get back — there are almost 33k lines — but you get the picture.

Now let's parse it:

```
In [3]: aba_soup = Soup(response.text)
```

Remember, we can treat this top level `aba_soup` object as a giant nested tag. What do we do with nested tags? Run `find_all` on them.

We can never be 100% sure when dealing with sites that we didn't create, but looking at the page, it's probably safe to assume the data we want is on an HTML table.

Let's find all the `table` tags in this HTML.

```
In [3]: tables = aba_soup.find_all('table')
```

Remember `find_all` always returns a list. In this case a list of BS4 `table` tags. Let's check how many tables we got back.

```
In [4]: len(tables)
Out[4]: 1
```

It's just the one here, which is easy, but it's not at all uncommon for sites to have multiple tables. In that 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 it's just the one it's easy.

```
In [5]: aba_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`, `font` etc).

`aba_table` is still a nested tag, so let's run another `find_all`:

```
In [6]: rows = aba_table.find_all('tr')
```

This gives us a list of all the `tr` tags inside our table. Let's find the header row:

```
In [7]: rows[1]
Out[7]:
<tr>
<th align="left" width="13"><font size="1">No</font></th>
<th align="left" width="91"><font size="1">Name</font></th>
<th align="left" width="15"><font size="1">Pnts</font></th>
<th align="left" width="15"><font size="1">GP</font></th>
<th align="left" width="25"><font size="1">Min</font></th>
<th align="left" width="22"><font size="1">FGM</font></th>
<th align="left" width="25"><font size="1">FGA</font></th>
<th align="left" width="21"><font size="1">FG%</font></th>
<th align="left" width="21"><font size="1">3PM</font></th>
<th align="left" width="20"><font size="1">3PA</font></th>
<th align="left" width="20"><font size="1">3P%</font></th>
<th align="left" width="21"><font size="1">FTM</font></th>
<th align="left" width="20"><font size="1">FTA</font></th>
<th align="left" width="20"><font size="1">FT%</font></th>
<th align="left" width="25"><font size="1">OReb</font></th>
<th align="left" width="24"><font size="1">TReb</font></th>
<th align="left" width="20"><font size="1">AST</font></th>
<th align="left" width="20"><font size="1">PF</font></th>
<th align="left" width="12"><font size="1">Dq</font></th>
<th align="left" width="15"><font size="1">Stl</font></th>
<th align="left" width="20"><font size="1">Trn</font></th>
<th align="left" width="20"><font size="1">Blk</font></th>
<th align="left" width="19"><font size="1">PPG</font></th>
<th align="left" width="16"><font size="1">Hi</font></th>
<th align="left" width="44"><font size="1">Years</font></th>
</tr>
```

That'll be useful later for knowing what columns are what.

Now how about some data.

```
In [8]: first_data_row = rows[2]

In [9]: first_data_row
Out[9]:
<tr>
<td align="left" width="13"><font size="1">10</font></td>
<td align="left" width="91"><font size="1">Louie Dampier</font></td>
<td align="left" width="15"><font size="1">13726</font></td>
<td align="left" width="15"><font size="1">728</font></td>
<td align="left" width="25"><font size="1">27770</font></td>
<td align="left" width="22"><font size="1">5290</font></td>
<td align="left" width="25"><font size="1">12047</font></td>
<td align="left" width="21"><font size="1">.439</font></td>
<td align="left" width="21"><font size="1">794</font></td>
<td align="left" width="20"><font size="1">2217</font></td>
<td align="left" width="20"><font size="1">.358</font></td>
<td align="left" width="21"><font size="1">2352</font></td>
<td align="left" width="20"><font size="1">2849</font></td>
<td align="left" width="20"><font size="1">.826</font></td>
<td width="25">- </td>
<td align="left" width="24"><font size="1">2282</font></td>
<td align="left" width="20"><font size="1">4044</font></td>
<td align="left" width="20"><font size="1">1633</font></td>
<td width="12">- </td>
<td width="15">- </td>
<td width="20">- </td>
<td width="20">- </td>
<td align="left" width="19"><font size="1">18.9</font></td>
<td align="left" width="16"><font size="1">55</font></td>
<td align="left" width="44"><font size="1">67-76</font></td>
</tr>
```

It's the first row — Loui Dampier — nice. Note this is *still* a nested tag, so we need to use `find_all` again. The end is in site though.

```
In [10]: first_data_row.find_all('td')
Out[10]:
[<td align="left" width="13"><font size="1">10</font></td>,
 <td align="left" width="91"><font size="1">Louie Dampier</font></td>,
 <td align="left" width="15"><font size="1">13726</font></td>,
 <td align="left" width="15"><font size="1">728</font></td>,
 <td align="left" width="25"><font size="1">27770</font></td>,
 <td align="left" width="22"><font size="1">5290</font></td>,
 <td align="left" width="25"><font size="1">12047</font></td>,
 <td align="left" width="21"><font size="1">.439</font></td>,
 <td align="left" width="21"><font size="1">794</font></td>,
 <td align="left" width="20"><font size="1">2217</font></td>,
 <td align="left" width="20"><font size="1">.358</font></td>,
 <td align="left" width="21"><font size="1">2352</font></td>,
 <td align="left" width="20"><font size="1">2849</font></td>,
 <td align="left" width="20"><font size="1">.826</font></td>,
 <td width="25">- </td>,
 <td align="left" width="24"><font size="1">2282</font></td>,
 <td align="left" width="20"><font size="1">4044</font></td>,
 <td align="left" width="20"><font size="1">1633</font></td>,
 <td width="12">- </td>,
 <td width="15">- </td>,
 <td width="20">- </td>,
 <td width="20">- </td>,
 <td align="left" width="19"><font size="1">18.9</font></td>,
 <td align="left" width="16"><font size="1">55</font></td>,
 <td align="left" width="44"><font size="1">67-76</font></td>]
```

This returns a list of `td` tags. Technically, these each have a `font` tag inside of them, but because they each only have one `font` tag we can still call `string` to get the data out.

```
In [11]: [str(x.string) for x in first_data_row.find_all('td')]
Out[11]:
['10',
 'Louie Dampier',
 '13726',
 '728',
 '27770',
 '5290',
 '12047',
 '.439',
 '794',
 '2217',
 '.358',
 '2352',
 '2849',
 '.826',
 '-\xa0',
 '2282',
 '4044',
 '1633',
 '-\xa0',
 '-\xa0',
 '-\xa0',
 '-\xa0',
 '18.9',
 '55',
 '67-76']
```

Again note the list comprehension. Now that we've got this working, let's put it inside a function that will work on any row.

```
def parse_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')]
```

We have to apply `parse_row` to each row in our data. Since the first row is the header, that data is `rows[1:]`.

```
In [12]: list_of_parsed_rows = [parse_row(row) for row in rows[2:]]
```

Working with lists of lists is a pain, so let's get this into Pandas. The DataFrame constructor is pretty flexible. Let's try passing it `list_of_parsed_rows` and seeing what happens:

```
In [13]: df = DataFrame(list_of_parsed_rows)

In [14]: df.head()
Out[14]:
   0           1    2    3    ...    22    23    24
0  10  Louie Dampier  13726  728    ...    18.9    55  67-76
1  44      Dan Issel  12823  500    ...    25.6    51  70-76
2  24      Ron Boone  12153  662    ...    18.4    42  68-76
3  34     Mel Daniels  11739  628    ...    18.7    56  67-75
4  32    Julius Erving  11662  407    ...    28.7    63  71-76
```

Great. It just doesn't have column names. Let's fix that. We could just look at the table online and assign a list of what they should be called, but let's use our header row:

```
In [15]: df.columns = [str(x.string).lower() for x in rows[1].find_all('th')]
```

We're almost there. Our one remaining issue is that all of the data is in string form, like numbers stored as text in Excel. It's an easy fix via the `astype` method.

We can figure out which should go where:

```
In [16]: str_cols = ['name', 'years']

In [17]: float_cols = ['fg%', '3p%', 'ft%', 'ppg']

In [18]:
int_cols = [x for x in df.columns if not ((x in str_cols) or
                                             (x in float_cols))]
```

Then use the `astype` method. For floats:

```
In [19]: df[float_cols] = df[float_cols].astype(float)
```

And for ints:

```
In [20]: df[int_cols] = df[int_cols].astype(int)
...
ValueError: invalid literal for int() with base 10: '-\xa0'
```

Uh oh. This is a real life error I ran into when writing this chapter. I considered rewriting things to steer around it, but these kind of things come up all the time, so this is good practice.

Let's handle it. Copying the error:

```
ValueError: invalid literal for int() with base 10: '-\xa0'
```

And pasting it into Google brought up a [helpful Stackoverflow page](#) that says, “I solved the error using `pandas.to_numeric`” and gives an example:

```
data.Population1 = pd.to_numeric(data.Population1, errors="coerce")
```

So let's try it:

```
In [21]: df[int_cols] = pd.to_numeric(df[int_cols], errors='coerce')
...
TypeError: arg must be a list, tuple, 1-d array, or Series
```

Still not working. Looks like you have to call it a single column at time? Let's try that. We have a bunch of columns, but we can go through them one at a time with a loop:

```
In [22]:
for int_col in int_cols:
    df[int_col] = pd.to_numeric(df[int_col], errors='coerce')
```

Great, finally, no errors. Taking a look at our data:

```
In [24]: df.head()
Out[24]:
   no      name  pnts   gp   min   ...   blk   ppg   hi  years
0  10  Louie Dampier  13726  728  27770   ...   NaN  18.9  55.0  67-76
1  44      Dan Issel  12823  500  19444   ...   NaN  25.6  51.0  70-76
2  24      Ron Boone  12153  662  21596   ...   NaN  18.4  42.0  68-76
3  34     Mel Daniels  11739  628  22340   ...  351.0  18.7  56.0  67-75
4  32  Julius Erving  11662  407  16550   ...  648.0  28.7  63.0  71-76
```

There we go, we've built our first real web scraper!

## APIs

Above, we learned how to scrape a website using BeautifulSoup to work with HTML. While HTML basically tells the browser what to render onscreen, an API works differently.

### Two Types of APIs

In practice, people mean at least two things by API.

The trouble comes from the acronym — *Application Programming Interface*. It all depends 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.

A more specific example: part of the Pandas API is the `pd.merge` function. The `merge` API 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.

This `merge` API isn’t quite the same thing as the `merge` documentation. Ideally, an API is accurately documented (and Pandas is), but it’s not required. Nor is the API the `merge` function itself exactly. Instead, it’s how the function interacts with the outside world and the programmer. Basically what it takes and returns.

### 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 its URL. Everyone is familiar with the most basic urls, e.g.

[www.fantasymath.com](http://www.fantasymath.com)

But urls can have extra stuff too, e.g.

[api.fantasymath.com/v2/players-comp/?player=michael-jordan&player=larry-bird](http://api.fantasymath.com/v2/players-comp/?player=michael-jordan&player=larry-bird)

At a very basic level: a web API lets you specifically manipulate the URL (e.g. maybe you put in `magic-johnson` instead of `larry-bird`) 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 a predictable, easy to digest way. You can’t just tack on “api”

to the front of any URL, play around with parameters, and start getting back useful data. Many APIs you'll be using come with instructions on what to do and what everything means.

Not every API is documented and meant for public consumption. More commonly, they're built for a website's own, internal purposes. The Fantasy Math API, for example, which is a football API I built, is only called by the [www.fantasymath.com](http://www.fantasymath.com) site. Members can go to fantasymath.com and pick two players from the dropdown. When they click submit, the site (behind the scenes) accesses the API via that URL, and 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.

Many APIs are private, but some are public, which means anyone can access them to get back some data.

We'll look at a specific example of a public basketball API in a bit, but first let's touch on two prerequisite concepts.

## HTTP

This isn't a web programming book, but it will be useful to know a 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 the URL we want to visit, plus some optional data. A web server is basically a computer that's always on, listening for incoming requests. The web server handles the request, then — based on what's 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 a database.

There are other types of requests, but *GET* is likely the only one you'll really need to use when working with public APIs.

## JSON

When you visit (request) a normal site, the response comes back in HTML which your browser displays on the screen. But when you make a request to an API you usually get *data* back instead.

There are different data formats, but the most common by far nowadays 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 (potentially nested) combination of the python equivalent to dict, list, string's, and numbers.

```
"""
{
    "players": ["lebron-james", "joel-embiid"],
    "positions": ["pf", "c"],
    "season": 2022
}
"""
```

To convert our JSON response to something useful to python we just do `resp.json()` (or `json.loads(response.text)`).

But note, this won't work on just anything we throw at it. For example, if we had:

```
"""
{
    "players": ["lebron-james", "joel-embiid",
    "positions": ["pf", "c"],
    "season": 2022
}
"""
```

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.

This is where it's very helpful and powerful to have a good understanding of lists and dicts, and also comprehensions. If you're iffy on those it might be a good time to go back and review those parts of the intro to python section.

## Benefits of APIs

APIs have some benefits over scraping HTML.

Most importantly, because they're specifically designed to present data, they're usually much easier to use and require much less code. Often data is available via an API that you can't get otherwise.

That's not to say APIs don't have their disadvantages. For one, they aren't as common. Anyone who puts out an API 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. More on this in the next section.

## Working with APIs - General Process

Let's talk for a bit about a general process of working with APIs.

### 0. Authentication

Before anything else, we need to make sure we can actually get data from the API, i.e. visit it and get data back.

Authentication requirements depend on the API. Some APIs are *public*, which means anyone can request one of its URLs and get data back. Other APIs require you to *authenticate*, which basically means you need to send along some data (kind of like a password, though not exactly the same) with your request before they'll return results.

### 1. Finding an endpoint

Assuming you're authenticated, the very first step in working with an API is thinking about what we need and finding the right endpoint.

Generally, we'll do this through some combination of:

1. looking at documentation (if it exists)
2. looking at third party blog posts and tutorials
3. looking at other people's public code on github

Usually (1) or (2) is easiest; (3) is a last resort.

### 2. Visit endpoint in browser

After figuring out the endpoint you want to use, the next step is visiting it in your browser. All of these APIs return JSON.

These APIs can return a *lot* of data, and it's helpful to be able to look it all at once in the browser as opposed to trying to make sense of it in the REPL. More on this as we get into some examples.

### 3. Get what you need in Python

Once we've looked at the data in our browser and have figured out what we want, we can connect to the API in Python (using the `requests` library) and get what we need out of it.

#### 4. Get everything into Pandas

Our goal with any type of data in Python should always be getting it into Pandas ASAP.

Almost always, the easiest and best way to do that is by (1) getting list of identically structured dictionaries, then (2) passing that to `DataFrame`.

Re (1): with sports data we're almost always processing some item in a collection. We'll have historical stats for some player and year, then want to get those same stats for a bunch more players and years.

Our general process will be to play around with a specific instance of whatever we're working with (e.g. Dwayne Wade's stats, 2006), write a function that gets what we need out of it, then apply the function to the entire collection (all of Wade's other years, every other player).

We'll work through a few examples next.

#### NBA API

There are a few different NBA APIs out there. Some are via third parties (e.g. [fantasydata.com](#)) and require you to pay to use them.

But it turns out the NBA has its own, mostly internal API, which — as of this writing — anyone can access.

It's not necessarily easy to use — it's a hodge podge of several different endpoints, all of which are undocumented and have a lot of overlap and confusing parts. But that's fine. We're not programming archeologists trying to understand the evolution of the NBA API. We just want code that can get data, and there's more than enough to do that.

We'll run through our 4 step process with this in a second, but if you're new to working with APIs you might be wondering how we even made it this far — where do you even find something like this in the first place?

Answer is literally google, "nba api". Everything we need is a few results down, a very helpful site by nbasense.

<http://nbasense.com/nba-api/>

#### NBA Data API Walkthrough #1

Let's walk through the process of working with this API, starting with authentication.

## Authentication

The easiest way to check authentication is to try out a few URLs and see if you can get data back. On the nbasense documentation, clicking ‘try out an example!’ on the main page eventually brings us to:

[http://data.nba.net/prod/v1/20170201/0021600732\\_boxscore.json](http://data.nba.net/prod/v1/20170201/0021600732_boxscore.json)

Putting it in the browser we can see it returns JSON, vs an error or a message about not being allowed to access this data. Looks like this API is public. Great.

### 1. Finding an endpoint

Now that we know we can get data, let’s figure out what we want. Usually we’d go into it with some idea ahead of time (“I need historical game by game stats for the past five years”), but in this case let’s look around for something simple.

How about — basic information about each team?

According to nbasense, the endpoint that with that info is:

<http://data.nba.net/prod/v1/2021/teams.json>

Let’s try it.

### 2. Visit endpoint in browser

Try copying that URL and putting it in the browser.

What you see might depend on what browser you’re using. In FireFox, I get nicely formatted data:

```
▼ league:  
  ▼ vegas:  
    ▼ 0:  
      city: "Atlanta"  
      fullName: "Atlanta Hawks"  
      isNBAFranchise: true  
      confName: "East"  
      tricode: "ATL"  
      teamShortName: "Atlanta"  
      divName: "Southeast"  
      isAllStar: false  
      nickname: "Hawks"  
      urlName: "hawks"  
      teamId: "1610612737"  
      altCityName: "Atlanta"  
    ▼ 1:  
      nickname: "Celtics"  
      urlName: "celtics"  
      teamId: "1610612738"  
      altCityName: "Boston"  
      tricode: "BOS"  
      teamShortName: "Boston"  
      divName: "Atlantic"  
      isAllStar: false  
      isNBAFranchise: true  
      confName: "East"  
      fullName: "Boston Celtics"  
      city: "Boston"
```

**Figure 0.2:** Formatted JSON

But sometimes you might see a data dump that looks something like this:

```
{ "league" : { "vegas" : [ { "city" : "Atlanta", "fullName" :  
  "Atlanta Hawks", "isNBAFranchise" : true, "confName" : "East",  
  "tricode" : "ATL", "teamShortName" : "Atlanta", "divName" :  
  "Southeast", "isAllStar" : false, "nickname" : "Hawks", "urlName" :  
  "hawks", "teamId" : "1610612737", "altCityName" : "Atlanta" },  
  { "nickname" : "Celtics", "urlName" : "celtics", "teamId" :  
  "1610612738", "altCityName" : "Boston", "tricode" : "BOS",  
  "teamShortName" : "Boston", "divName" : "Atlantic", "isAllStar" :  
  false, "isNBAFranchise" : true, "confName" : "East", "fullName" :  
  "Boston Celtics", "city" : "Boston" }, ... ] }
```

This data will be a lot easier to make sense of if it's formatted, so let's talk about how to do that quick.

**JSON in your Web browser** I find it very helpful to explore the JSON these APIs return in my browser, where I can click around, collapse and expand things, and generally get a high level overview while also being able to zoom in on the parts I need to see.

Sometimes browsers do this automatically, but your browser might display JSON data as a wall of unformatted text.

If it does, the very first thing I'd recommend doing is installing a JSON viewer browser extension. I normally use Firefox, which does this automatically. For Chrome, [this extension](#) looks like a popular one.

With a viewer, we can turn this huge wall of text into specifics we can expand, visualize, etc.

**Back to our data.** Since we're looking at this API in the browser, you can see see it's just a URL that returns data. Here can see this JSON has two, top level fields — `league` and `_internal`. The `league` field has some further nested fields: `vegas`, `standard`, `utah`, `africa` and `sacramento` — most of which have what we want.

What do these locations mean? I have no idea. As far as I can tell, they return the same data. Remember, this is all built for the NBA's own internal, mysterious purposes. They could be code names; or maybe they're where their servers are located.

It doesn't matter. We have some data, let's get it in Python.

### 3. Get what you need in Python

*Note the code for this section is in `./code/05_03_api.py`. We're picking up after the imports.*

The endpoint we're working with:

```
In [1]: teams_url = 'http://data.nba.net/prod/v1/2021/teams.json'
```

We'll use that URL to get the data. In Python, Http requests are handled via the `request` package. Again, `GET` requests are for getting data.

```
In [2]: teams_resp = requests.get(teams_url)
```

This gives us a `response` object. The only thing we need to know about these is how to turn it into a format we're more familiar with. We do that with the `json` method.

```
In [3]: teams_json = teams_resp.json()
```

Note: this real data from a real API, so what you see in your own REPL is going to be different from what I'm showing here. That's probably OK. If the NBA API changes, and any of the following code breaks permanently, I'll update the book.

However, if you are running into issues or just want make sure you're seeing *exactly* what I'm showing here, I've saved a snapshot of the data that matches up with the book. To use it uncomment and run the following in `05_03_api.py`:

```
In [4]:  
with open('./data/json/teams.json') as f:  
    teams_json = json.load(f)
```

This will load the saved data I've included with the book. It's not any different — I got it from hitting this same API — but it's a way to make sure we're using the same thing. Up to you on whether you want to use it.

Either way, looking at `teams_json`, we can see it gives us bunch of data, more than we could ever view in the REPL. Here's the last few lines of what I see:

```
In [5]: teams_json  
Out[5]:  
...  
...  
{'fullName': 'Washington Wizards',  
 'city': 'Washington',  
 'confName': 'East',  
 'isNBAFranchise': True,  
 'divName': 'Southeast',  
 'isAllStar': False,  
 'tricode': 'WAS',  
 'teamShortName': 'Washington',  
 'teamId': '1610612764',  
 'altCityName': 'Washington',  
 'nickname': 'Wizards',  
 'urlName': 'wizards'}],  
 '_internal': {'xsltInCache': 'true',  
 'xsltTransformTimeMillis': '8',  
 'xsltForceRecompile': 'true',  
 'endToEndTimeMillis': '1425',  
 'consolidatedDomKey': 'prod__transform__marty_teams_list__4496649757625'  
,  
 'xsltCompileTimeMillis': '14',  
 'xslt': 'NBA/xsl/league/roster/marty_teams_list.xsl',  
 'igorPath': 'cron,1627034427189,1627034427189|router  
 ,1627034427189,1627034427302|domUpdater,1627034427406,1627034428466|  
 feedProducer,1627034428563,1627034428614',  
 'pubDateTime': '2021-07-23 06:00:28.607 EDT'}}
```

This data is exactly what we see in the browser, just in Python-dict form. Just like the browser, we have our top level keys:

```
In [6]: teams_json.keys()  
Out[6]: dict_keys(['league', '_internal'])
```

It's just a standard, Python dictionary:

```
In [6]: type(teams_json)  
Out[6]: dict
```

#### 4. Get everything into Pandas

Our goal with any data type work in Python should be getting it into Pandas ASAP.

The easiest way to do that is by getting a list of identically structured dicts, and passing that to `DataFrame`. That'll give us a `DataFrame` where our columns are our dict keys, and each row is a specific example.

Here data in that format is a few levels down in the `league` key. It looks like the same data is repeated in `vegas`, `standard`, `utah`, or `sacramento` — so we can take our pick.

Let's use `standard` since that sounds the most normal. We're looking for a list of dicts, which is what `standard` is:

```
In [7]: type(teams_json['league']['standard'])  
Out[7]: list
```

And here's what the first one looks like:

```
In [8]: teams_json['league']['standard'][0]  
Out[8]:  
{'city': 'Atlanta',  
 'fullName': 'Atlanta Hawks',  
 'isNBAFranchise': True,  
 'confName': 'East',  
 'tricode': 'ATL',  
 'teamShortName': 'Atlanta',  
 'divName': 'Southeast',  
 'isAllStar': False,  
 'nickname': 'Hawks',  
 'urlName': 'hawks',  
 'teamId': '1610612737',  
 'altCityName': 'Atlanta'}
```

So all we have to do is pass `teams_json['league']['standard']` to `DataFrame` and it'll work:

```
In [9]: df_teams = DataFrame(teams_json['league']['standard'])

In [10]: df_teams.head()
Out[10]:
   city      fullName  ...  urlName    teamId altCityName
0  Atlanta  Atlanta Hawks  ...  hawks  1610612737  Atlanta
1  Boston   Boston Celtics  ...  celtics  1610612738  Boston
2 Brooklyn  Brooklyn Nets  ...  nets  1610612751  Brooklyn
3 Charlotte Charlotte Hornets  ...  hornets  1610612766  Charlotte
4 Chicago   Chicago Bulls  ...  bulls  1610612741  Chicago
```

And there we go. We got team data from the NBA API.

## Roster Data

Let's do another one. How about rosters? Let's run through the four steps:

### 1. Find an endpoint

Looking at nbasense, if I go to Data :: Prod :: Roster :: LeagueRosterPlayers on the side it gives me this example endpoint:

<http://data.nba.net/prod/v1/2021/players.json>

### 2. Visit endpoint in browser

Looking at this in our browser, we can see it's a dict with the same fields as last time (`_internal` and `league`, which in turn has the same `vegas`, `standard`, `sacramento` etc fields).

We want to put it in Pandas, so again, keep an eye out for a list of dicts. Looks like that's in the `players` field inside of `league -> standard`.

Looking at it, we can see it has a lot of the fields we'd expect: `firstName`, `lastName`, `personId`, etc. Great.

There are a few non-standard parts too, for example the `teams` field, which contains a nested list of dicts with team info for the given player.

This will be a good opportunity to practice our Python/Pandas/API data rearranging skills, so let's look at it closer in Python.

### 3. Get what you need in Python

Loading it in Python like usual:

```
In [1]:  
players_url = 'http://data.nba.net/prod/v1/2021/players.json'  
players_resp = requests.get(players_url)  
players_json = players_resp.json()
```

(Again, if you want make sure you're using the same data, uncomment the `json.load` lines in `05_03_api.py`.)

We know from the browser that most of it is a giant list of players, and that's it's all pretty standard apart from a few teams field. Here's a single player (note I'm not showing the nested data here, we'll deal with that next):

```
In [2]: player0 = players_json['league']['standard'][0]  
  
In [3]: player0  
Out[3]:  
{'firstName': 'Precious',  
 'lastName': 'Achiuwa',  
 'temporaryDisplayName': 'Achiuwa, Precious',  
 'personId': '1630173',  
 'teamId': '1610612761',  
 'jersey': '5',  
 'isActive': True,  
 'pos': 'F',  
 'heightFeet': '6',  
 'heightInches': '8',  
 'heightMeters': '2.03',  
 'weightPounds': '225',  
 'weightKilograms': '102.1',  
 'dateOfBirthUTC': '1999-09-19',  
 ...  
 'nbaDebutYear': '2020',  
 'yearsPro': '1',  
 'collegeName': 'Memphis',  
 'lastAffiliation': 'Memphis/Nigeria',  
 'country': 'Nigeria'}
```

### 4. Get it into Pandas

A DataFrame needs a list of flat dictionaries, which — apart from the nested fields — is what we have. So we could ignore these and get a list of regular columns with:

```
In [4]:  
player_cols = [key for key, value in player0.items() if type(value) is not  
              (dict or list)]  
  
In [5]: player_cols  
Out[5]: ['firstName', 'lastName', 'temporaryDisplayName', 'personId',  
         'teamId', 'jersey', 'isActive', 'pos', 'heightFeet',  
         'heightInches', 'heightMeters', 'weightPounds',  
         'weightKilograms', 'dateOfBirthUTC', 'teams', 'nbaDebutYear',  
         'yearsPro', 'collegeName', 'lastAffiliation', 'country']
```

Then get a regular, player level DataFrame with:

```
In [6]: df_players = DataFrame(players_json['league']['standard'])[  
                           player_cols]  
  
In [7]: df_players.head()  
Out[7]:  
      firstName   lastName   ...   lastAffiliation   country  
0   Precious     Achiuwa   ...   Memphis/Nigeria   Nigeria  
1   Steven       Adams    ...   Pittsburgh/New Zealand   New Zealand  
2   Bam          Adebayo   ...   Kentucky/USA   USA  
3   Santi        Aldama   ...   Loyola-Maryland/Spain   Spain  
4   LaMarcus    Aldridge   ...   Texas-Austin/USA   USA
```

Let's set that aside – we can save it as a csv or put it as a `players` table in a SQL database, whatever.

Now let's look at the `teams` field. Let's look at a specific example.

```
In [8]: player0['teams']  
Out[8]:  
[{'teamId': '1610612748', 'seasonStart': '2020', 'seasonEnd': '2020'},  
 {'teamId': '1610612761', 'seasonStart': '2021', 'seasonEnd': '2021'}]
```

Looks like this field is its own, nested list of dicts with a players historical team information.

We can put this in a DataFrame:

```
In [9]: DataFrame(player0['teams'])  
Out[9]:  
      teamId  seasonStart  seasonEnd  
0   1610612748        2020        2020  
1   1610612761        2021        2021
```

which works.

So it looks like what this will do is give us is a table of every player and team they played for. This will be at the player-team level — which note may not necessarily be the same as player-team-season since players often switch teams within a single season.

## Learn to Code with Basketball

---

I think what we want is to put this code in a function that'll take a single player dictionary, and return his historical team info. We know the above is for Precious Achiuwa (because he's `player0`) but we'll want to keep track of this in our function too:

```
In [10]:  
def player_teams_to_df(player_dict):  
    df = DataFrame(player_dict['teams'])  
    df['personId'] = player_dict['personId']  
    return df
```

Trying it out on Achiuwa:

```
In [11]: player_teams_to_df(player0)  
Out[11]:  
      teamId seasonStart seasonEnd personId  
0  1610612748        2020        2020  1630173  
1  1610612761        2021        2021  1630173
```

Cool. Now let's use this function, `pd.concat` and a list comprehension to get a big DataFrame of historical team info for *all* players:

```
In [12]:  
df_player_team = pd.concat([player_teams_to_df(x) for x in  
                           players_json['league']['standard']], ignore_index=True)
```

Looking at it:

```
In [13]: df_player_team.head()  
Out[13]:  
      teamId seasonStart seasonEnd personId  
0  1610612748        2020        2020  1630173  
1  1610612761        2021        2021  1630173  
2  1610612760        2013        2019  203500  
3  1610612740        2020        2020  203500  
4  1610612763        2021        2021  203500
```

It looks like it worked.

Cool. So this gives us player-team level data for every player in the NBA. Along with our team, and player data, this is a decent start.

There are other interesting endpoints, and you can play with them, but let's change gears a bit.

## nba\_api Wrapper

We went through the last section because I wanted you to get familiar with general process for working with API endpoints — HTTP, JSON etc.

But sometimes people build *wrappers* (sometimes called *clients*) with APIs. This is code that calls these endpoints under the hood and with the data.

Clients are *language specific*. There's an NBA API wrapper written in PHP (another programming language), and another for Javascript. There are multiple Python wrappers.

A lot of time big, well funded companies release and maintain clients because they want to make it easier for developers to use their stuff. So Facebook (Meta?) has a Python (and Javascript, and PHP) client for their advertising API, that lets programmers manage ads, and view clicks and stuff.

In this case, the NBA API client (wrapper) was *not* built by the NBA. Instead, some kind, NBA data enthusiast (who goes by the username [swar](#) on github) made it for us, with a lot of help from the community. Thank you swar and friends!

Some general benefits to using clients or wrappers:

- The main benefit is it's much simpler. The NBA API is a hodge podge of 6 different endpoints, many with duplicate data. This is obviously something that's evolved over time. My guess is the NBA wanted to put out new data but couldn't change older APIs people were relying on previously, but it doesn't matter. Using [nba\\_api](#) means we don't have to worry about that.
- Related: if an NBA endpoint *does* change (very possible), swar or someone else can just tweak the [nba\\_api](#) client and it'll work for everyone.
- It's also very likely we'll get better data. We saw above how there is a lot of duplication in the NBA API data. Hopefully swar understands the difference between standard, vegas and utah or whatever.

Drawbacks to clients:

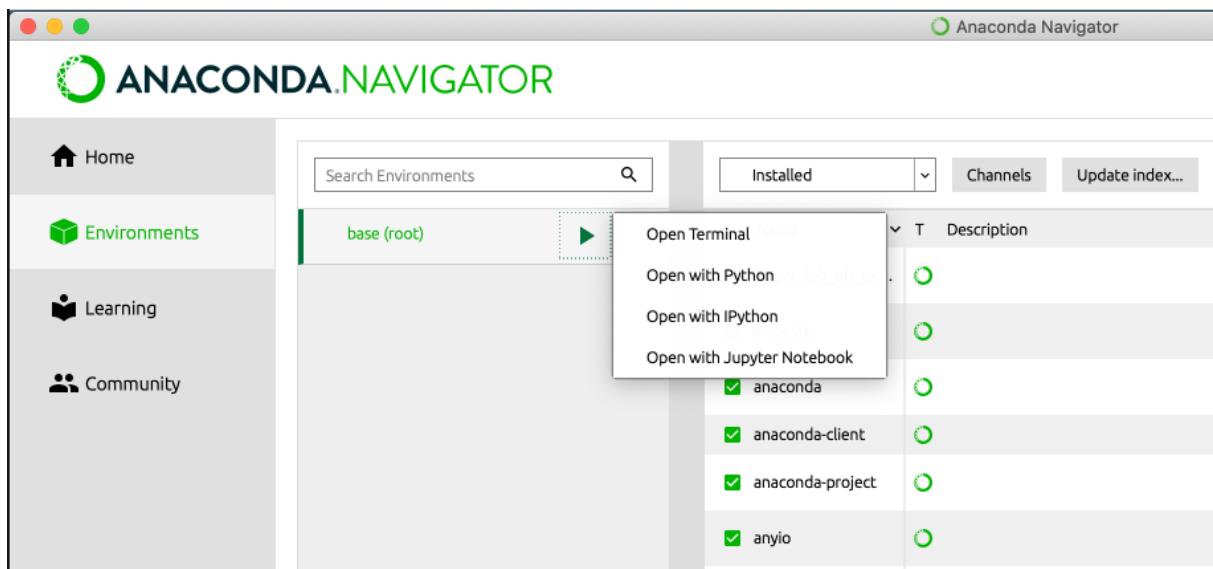
- It's more complicated. Using a client means we can't (usually) just visit the endpoint in the browser and look at the raw data we want. Instead, it's doing a lot of stuff under the hood.
- Not as transferable. Learning [nba\\_api](#) isn't going to help you with other APIs or necessarily even other API clients. Instead, to use it, we need to learn the wrapper's syntax and how they want us to do things.
- We have to install them. The [nba\\_api](#) package doesn't come included in Anaconda/Spyder.

Those drawbacks are why I wanted to show you how to work with API directly. But in this case the NBA API Python wrapper is powerful and popular enough that seems worth getting familiar with it.

## Installing nba\_api

So far, Anaconda has come with every Python package we've needed, but it doesn't come with [nba\\_api](#). Here's how to get it:

1. Open up Anaconda Navigator.
2. Click on [Environments](#) on the left side bar.
3. This will bring up a list of your environments. Click on the 'play button' style green triangle in environment you're using (usually there will just be one environment: [base \(root\)](#), but if you know you're using a different one click that).
4. Select [Open Terminal](#).



**Figure 0.3:** Opening Terminal in Anaconda

5. A text console will open up. Type:

```
pip install nba_api
```

6. Then close it once it installs. Then restart Spyder.

## Player Data in nba\_api

Remember, this Python package is just a wrapper around the underlying NBA API and NBA data.

One thing that's very normal with APIs (and data in general) is that everything has an id — players, games, teams etc. This means a lot of the endpoints take some id and return data about whatever you're asking for.

But these ids don't mean anything outside of the NBA API — e.g. the player id for LeBron James is 2544 — so that raises an interesting question. How do we get started and get ahold of some ids in the first place?

The answer is the `nba_api` package comes with two functions to get us started. These are:

```
from nba_api.stats.static import teams, players
```

According to documentation, this part of the code doesn't even call the NBA api itself — the info and ids are all hardcoded and included in the package. Presumably this means the package needs to be updated to add rookies every year, which is a good thing to keep in mind.

*Note: the following code is in `05_04_wrapper.py`. We'll pick up after the regular imports.*

So we can do:

```
In [1]: from nba_api.stats.static import teams, players

In [2]: player_df = DataFrame(players.get_players())

In [3]: player_df.head()
Out[3]:
   id      full_name first_name    last_name  is_active
0  76001      Alaa Abdelnaby     Alaa  Abdelnaby    False
1  76002      Zaid Abdul-Aziz    Zaid  Abdul-Aziz    False
2  76003  Kareem Abdul-Jabbar  Kareem  Abdul-Jabbar    False
3    51  Mahmoud Abdul-Rauf  Mahmoud  Abdul-Rauf    False
4  1505      Tariq Abdul-Wahad   Tariq  Abdul-Wahad    False
```

And teams:

```
In [4]: team_df = DataFrame(teams.get_teams())

In [5]: team_df.head()
Out[5]:
   id      full_name ...      state year Founded
0  1610612737      Atlanta Hawks ...  Atlanta  1949
1  1610612738      Boston Celtics ... Massachusetts  1946
2  1610612739      Cleveland Cavaliers ...  Ohio  1970
3  1610612740      New Orleans Pelicans ... Louisiana  2002
4  1610612741      Chicago Bulls ... Illinois  1966
```

Aside: one question to test your getting-data-from-Python-to-Pandas knowledge, what type of data do you think `players.get_players()` and `teams.get_teams()` are returning<sup>1</sup>? Note we're passing them to DataFrame and getting a nice, simple data back.

---

<sup>1</sup>Answer: a list of dicts.

## Getting data with nba\_api

Once we have our player and team data and along with the relevant ids, getting data with [nba\\_api](#) is pretty straightforward.

Remember our normal api process: find an endpoint, visit endpoint in browser, get what you need in Python, and put it in Pandas.

With [nba\\_api](#) this process is similar:

### 1. Find an nba\_api endpoint

Just like with the regular API, the first step is finding an endpoint. Normally I think of “endpoint” as being the ending of a URL, like:

[prod/v1/20170201/0021600732\\_boxscore.json](#)

But here they’re functions and types in the [nba\\_api](#) package. That’s fine.

A complete list of endpoints is here, under the [Endpoints](#) section:

[https://github.com/swar/nba\\_api/blob/master/docs/table\\_of\\_contents.md](https://github.com/swar/nba_api/blob/master/docs/table_of_contents.md)

Once an endpoint sounds promising, we can click on it to see what data it returns what we need to call it. The documentation mentions [CommonPlayerInfo](#) so let’s start with that. Clicking on it brings us to:

[https://github.com/swar/nba\\_api/blob/master/docs/nba\\_api/stats/endpoints/commonplayerinfo.md](https://github.com/swar/nba_api/blob/master/docs/nba_api/stats/endpoints/commonplayerinfo.md)

This page shows us what it takes (Parameters) as well as the data it returns (JSON). So, it looks like here [CommonPlayerInfo](#) takes a player id and returns a lot of stuff.

### 2. Import nba\_api Endpoint in Python

Once we’ve decided on an endpoint, we load it into Python like this:

```
from nba_api.stats.endpoints.endpointname import EndPointName
```

So the last part of the `from` is lowercase (`endpointname`) and the part we `import` is CamelCase.

(Note CamelCase means capitalizing new words, with no spaces in between. It’s called that because words have humps, like a camel.)

So to use [CommonPlayerInfo](#) we do this:

```
from nba_api.stats.endpoints.commonplayerinfo import CommonPlayerInfo
```

And `TeamYearByYearStats` would be:

```
from nba_api.stats.endpoints.teamyearbyyearstats import  
TeamYearByYearStats
```

The documentation lists the parameters and whether they're required. As a starting point that's helpful, but I've found it's misleading. Often it flags parameters as 'required' when they're not.

For example the docs say `TeamYearByYearStats` "requires" a team id, as well as bunch of other stuff — league id, per mode etc.

But if we actually look at it in the REPL:

```
In [6]: from nba_api.stats.endpoints.teamyearbyyearstats import  
TeamYearByYearStats

In [7]: TeamYearByYearStats?  
Init signature:  
TeamYearByYearStats(  
    team_id,  
    league_id='00',  
    per_mode_simple='Totals',  
    season_type_all_star='Regular Season',  
    proxy=None,  
    headers=None,  
    timeout=30,  
    get_request=True,  
)  
Docstring:      <no docstring>  
File:          .../site-packages/nba_api/stats/endpoints/  
              teamyearbyyearstats.py  
Type:          type  
Subclasses:
```

We can see `league_id`, `per_mode_simple` and `season_type_all_star` default to '`'00'`', '`'Totals'`', and '`'Regular Season'`' respectively. This means they aren't actually required.

I'd get in the habit of looking at every endpoint in the REPL like this before stressing out about some parameter that's "required" but not necessarily clear.

### 3. Calling the Endpoint

Once we have our endpoint loaded, we can call it. Let's do the `TeamYearByYearStats` endpoint. It takes a team id, which we can grab using our static, teams data we got above.

```
In [8]: teams_df.head()
Out[8]:
      id          full_name  ...      state year_founded
0  1610612737      Atlanta Hawks  ...    Atlanta      1949
1  1610612738      Boston Celtics  ...  Massachusetts  1946
2  1610612739  Cleveland Cavaliers  ...        Ohio      1970
3  1610612740  New Orleans Pelicans  ...    Louisiana      2002
4  1610612741      Chicago Bulls  ...    Illinois      1966
```

Let's do the Pelicans.

```
In [9]: pelicans_id = '1610612740'
In [9]: pelicans_data = TeamYearByYearStats(pelicans_id)
```

If we look at this:

```
In [10]: pelicans_data
Out[10]: <nba_api.stats.endpoints.teamyearbyyearstats.Tea...
          mYearByYearStats
at
0x12f946290>
```

It doesn't tell us much. Let's get it into Pandas —

#### 4. Getting Data in Pandas

Once we call our endpoint, `nba_api` has some built in functions to get data out. Specifically — and this part is the same for every endpoint — we have to call `get_data_frames()`. This returns a list of DataFrames. Usually the one we want is the first item in the list.

So here we do:

```
In [11]: pelicans_df = pelicans_data.get_data_frames()[0]
In [12]: pelicans_df.head()
Out[12]:
      TEAM_ID          TEAM_CITY  ...    BLK    PTS  PTS_RANK
0  1610612740      New Orleans  ...  394  7699      19
1  1610612740      New Orleans  ...  346  7529      17
2  1610612740      New Orleans  ...  310  7252      30
3  1610612740  New Orleans/Oklahoma City  ...  311  7611      25
4  1610612740  New Orleans/Oklahoma City  ...  347  7833      25
```

And that works.

## Play by Play Data Example

Let's do another one. Running through our steps:

### 1. Find an nba\_api endpoint

We start by looking at our [list of endpoints](#) for something we want. Let's get some play by play data with [PlayByPlayV2](#). Clicking on it, we can see it takes a `game_id` parameter.

Note we don't have any game ids yet — we just have our player and team DataFrames — so we'll have to figure out how to get those. In the meantime, it looks like they include an example (in the Valid URL section), `0021700807`, so we can use that.

### 2. Import nba\_api endpoint in Python

We can import it in Python using our standard `nba_api` syntax:

```
In [1]: from nba_api.stats.endpoints.playbyplayv2 import PlayByPlayV2
```

Let's look it in the REPL:

```
In [2]: PlayByPlayV2?
Init signature:
PlayByPlayV2(
    game_id,
    end_period='0',
    start_period='0',
    proxy=None,
    headers=None,
    timeout=30,
    get_request=True,
)
Docstring:      <no docstring>
File:          .../site-packages/nba_api/stats/endpoints/playbyplayv2.py
Type:          type
Subclasses:
```

### 3. Calling the Endpoint

So the only thing we have to give it is a `game_id`. Let's try it with the example id from in the documentation.

```
In [3]: game_id = '0021700807'
In [3]: pbp_data = PlayByPlayV2(game_id)
```

Again, looking at this:

```
In [4]: pbp_data  
Out[4]: <nba_api.stats.endpoints.playbyplayv2.PlayByPlayV2 at 0x12faf6d40>
```

doesn't show much. We need get it in Pandas.

#### 4. Getting Data in Pandas

Again, with `nba_api`, after calling an endpoint, it's always the same code to get data into a `DataFrame`:

```
In [5]: pbp_df = pbp_data.get_data_frames()[0]
```

Looking at some of the columns:

```
In [6]: pbp_df[['HOMEDESCRIPTION', 'VISITORDESCRIPTION']].head(10)  
Out[6]:  
          HOMEDESCRIPTION           VISITORDESCRIPTION  
0            None                  None  
1  Jump Ball: Tip to Gibson      None  
2            None  MISS Wiggins 27' 3PT Jump Shot  
3            None    Gibson REBOUND (Off:1 Def:0)  
4            None  MISS Gibson 1' Tip Layup Shot  
5  Thomas REB (Off:0 Def:1)      None  
6            None    MISS Thompson 1      None  
7            None  Wiggins REBOUND (Off:0 Def:1)  
8            None  MISS Gibson 8' Jump Shot  
9  James REB (Off:0 Def:1)      None
```

Looks like that worked. This is cool, but it's just the one game id. What should we do if we want to get the other ones?

#### Play by Play Data for Multiple Games

If we want to use `PlayByPlayV2` on other games, we need to find some game ids. We do *that* by calling an endpoint that returns some.

Looking at the [endpoint list](#), my first thought was something like 'schedule' but I don't see that on there. There are some game log endpoints though. How about `TeamGameLog`?

It looks like it takes a team id (which we have) and gives back game id (among other stats). That'll work.

Running through the rest of process quick:

```
In [7]: from nba_api.stats.endpoints.teamgamelog import TeamGameLog

In [7]: TeamGameLog?
Init signature:
TeamGameLog(
    team_id,
    season='2021-22',
    season_type_all_star='Regular Season',
    date_from_nullable='',
    date_to_nullable='',
    league_id_nullable='',
    proxy=None,
    headers=None,
    timeout=30,
    get_request=True,
)
Docstring:      <no docstring>
File:          .../site-packages/nba_api/stats/endpoints/teamgamelog.py
Type:          type
Subclasses:
```

So the only thing it requires is a `team_id`, perfect. Let's do the Bucks this time. Finding it in our teams DataFrame:

```
In [8]: team_df.query("abbreviation == 'MIL'")[['id', 'full_name']]
Out[8]:
      id      full_name
12  1610612749  Milwaukee Bucks

In [9]: bucks_id = '1610612749'
```

And calling the `TeamGameLog` endpoint:

```
In [10]: mil_log = TeamGameLog(bucks_id).get_data_frames()[0]

In [11]: mil_log.head()
Out[11]:
   Team_ID      Game_ID      GAME_DATE      MATCHUP  ...  TOV  PF  PTS
0  1610612749  0022101218  APR 10, 2022  MIL @ CLE  ...  12  14  115
1  1610612749  0022101203  APR 08, 2022  MIL @ DET  ...  7  20  131
2  1610612749  0022101198  APR 07, 2022  MIL vs. BOS  ...  11  19  127
3  1610612749  0022101183  APR 05, 2022  MIL @ CHI  ...  14  19  127
4  1610612749  0022101167  APR 03, 2022  MIL vs. DAL  ...  9  22  112
```

Perfect. So now we can use the `Game_ID` column to get the play by play data from the games we want. Let's grab some of these and put them in a list:

```
In [12]: mil_game_ids = list(mil_log['Game_ID'].head())
In [13]: mil_game_ids
Out[13]: ['0022101218', '0022101203', '0022101198', '0022101183', '0022101167']
```

If we wanted to get the play by play data for all these games and stick them on top of each other in one table we could do something like this:

```
In [14]:
mil_pbp_df1 = pd.concat([
    PlayByPlayV2(x).get_data_frames()[0] for x in mil_game_ids],
    ignore_index=True)
```

And that works:

```
In [15]: mil_pbp_df1.head()
Out[15]:
   GAME_ID  EVENTNUM ... VIDEO_AVAILABLE_FLAG
0 0022101218      2 ...          0
1 0022101218      4 ...          1
2 0022101218      7 ...          1
3 0022101218      9 ...          0
4 0022101218     11 ...          1
```

The only issue is this doesn't handle errors very well.

Occasionally — whether it's our fault (we've mistyped a game id) or the NBAs (the server takes too long to respond) — a `PlayByPlayV2` call on a specific game id might not work and throw an error.

If that happens when all our `PlayByPlayV2` calls are written inside a list comprehension like this the whole thing errors out and we end up with no data.

We might have a list of 100 game ids and — if 99 work and 1 doesn't — we end up with no data.

So in a case like this it's probably better to explicitly allow for (and handle) errors.

## Interlude: Exceptions in Python

Allowing for errors is basic part of any programming language. We haven't covered it up till this point because — in type of interactive, REPL based, data processing work we do — it's not super important. In a REPL, if you run into an error, no big deal, just edit your code and try something else.

Compare this to, say, the autopilot software that runs on an airplane. In that case if you run into an error — maybe you got some faulty sensor data and divided by 0 or something — you definitely do *not*

want the whole program to just throw up its hands and say “error”. You want the plane to handle the error and try something else.

Let’s look at example of an error in Python. What if we call `PlayByPlayV2` with an obvious non-game id. So instead of `'0021700807'` let’s try it with `'XXXXXXXXXX'`.

```
In [16]: PlayByPlayV2('XXXXXXXXXX')
...
JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

I’ve truncated the output, but we can see there’s a problem. It’s not necessarily super helpful — something like “not a valid game id” would be a better error — but something is obviously wrong.

So now let’s say we want to handle this type of invalid game id error. If we get a valid id, we’ll return the DataFrame, if it’s invalid we’ll just return an empty DataFrame.

Here’s how you’d do that. Note I’m going to put it inside a function so we can try it out with different values:

```
1 def get_pbp_data(game_id):
2     try:
3         pbp_df = PlayByPlayV2(game_id).get_data_frames()[0]
4     except Exception as _:
5         pbp_df = DataFrame()
6
7     return pbp_df
```

The new part is the `try ... except ...` lines. Let’s go through it.

Lines 2-3 mean we’re *trying* some code (calling `PlayByPlayV2` and getting the data out in this case). Lines 4-5 are where we tell Python what to do if there’s an error (make `pbp_df` an empty DataFrame).

The `except Exception` part means we’re catching *any* error. That’s sort of broad (maybe we just want to catch the specific `JSONDecodeError` we saw before), but it’s fine. I honestly don’t use `try ... except ...` that much. Maybe it’d be different if we were writing airplane software.

So now we can run our function with a valid game id:

```
In [17]: get_pbp_data('0021700807')
Out[17]:
   GAME_ID  EVENTNUM  ...  VIDEO_AVAILABLE_FLAG
0  0021700807      2  ...          0
1  0021700807      4  ...          1
2  0021700807      7  ...          1
3  0021700807      8  ...          1
4  0021700807      9  ...          1
...
450 0021700807    654  ...          0
451 0021700807    655  ...          0
452 0021700807    659  ...          1
453 0021700807    661  ...          1
454 0021700807    662  ...          0
```

And it works. If we run it with an invalid game id:

```
In [18]: get_pbp_data('XXXXXXXXXXXX')
Out[18]:
Empty DataFrame
Columns: []
Index: []
```

We just get an empty DataFrame. Perfect.

So using this `try ... except ...` syntax, this is how I would write code to run through a bunch of game ids, getting the play by play data and when I can, and skipping (and keeping track of) the game ids where I can't.

Let's add a fake `game_id` to our list to make sure it works.

```
In [19]: mil_game_ids = mil_game_ids + ['XXXXXXXXXXXX']

In [20]: mil_game_ids
Out[20]:
['0022101218',
 '0022101203',
 '0022101198',
 '0022101183',
 '0022101167',
 'XXXXXXXXXXXX']
```

And here's the code.

Note what we're doing is basically, looping through each `game_id` and trying to get the data. If it works, we add it to our data. If not we note the game id (by adding it to our list of `bad_game_ids`) and move on.

```
pbp_df_all = DataFrame()
bad_game_ids = []
for gid in mil_game_ids:
    print(gid)
    try:
        working_df = PlayByPlayV2(gid).get_data_frames()[0]
    except Exception as _:
        bad_game_ids = bad_game_ids + [gid]
    continue # continue means stop here and go to next item in the
            # for loop

pbp_df_all = pd.concat([pbp_df_all, working_df], ignore_index=True)
```

Running it prints this:

```
Out[21]:
0022101218
0022101203
0022101198
0022101183
0022101167
XXXXXXXXXX
```

We can check to make sure it worked. Looking at the values of `game_id` in our play by play data:

```
In [22]: pbp_df_all['GAME_ID'].value_counts()
Out[22]:
0022101183    484
0022101203    482
0022101218    467
0022101198    463
0022101167    459
Name: GAME_ID, dtype: int64
```

We can see they're all there. And our bad ids:

```
In [23]: bad_game_ids
Out[23]: ['XXXXXXXXXX']
```

Perfect.

After getting this data, we can store them in a SQL database (see chapter 4), or as csv's or whatever. Note usually though we *would* want to store this data, as opposed to treating the NBA API as our storage that we call whenever we want to do some analysis.

First, it's faster. It's much more efficient to store data locally (or even get it directly from a database online) than to re-hit a networked API every time we need the data

Second, it's the polite thing to do. Hosting and maintaining an API costs money. It's usually not a big deal playing around with it or grabbing data occasionally, but we don't need to overload servers when we don't have to.

Finally, storing the data means you'd have it if anything ever happened to the API.

# 6. Data Analysis and Visualization

## Introduction

In the first section of the book we defined *analysis* as “deriving useful insights from data”.

One way to derive insights is through *modeling*. We’ll cover that in the next chapter.

This chapter is about everything else. Non-modeling analysis usually takes one of three forms:

1. Understanding the **distribution** of a single variable.
2. Understanding the **relationship** between two or more variables.
3. Summarizing a bunch of variables via one **composite** score.

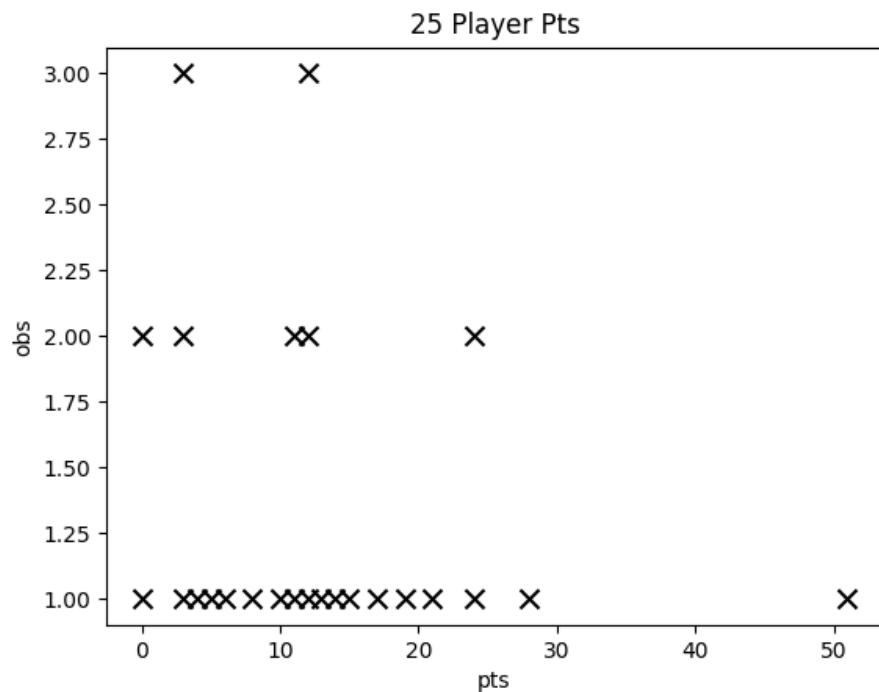
## Distributions

A distribution is a way to convey the *frequency* of outcomes for some variable.

For example, take player points per game. Here are 25 random player performances from the 2019-2020 season:

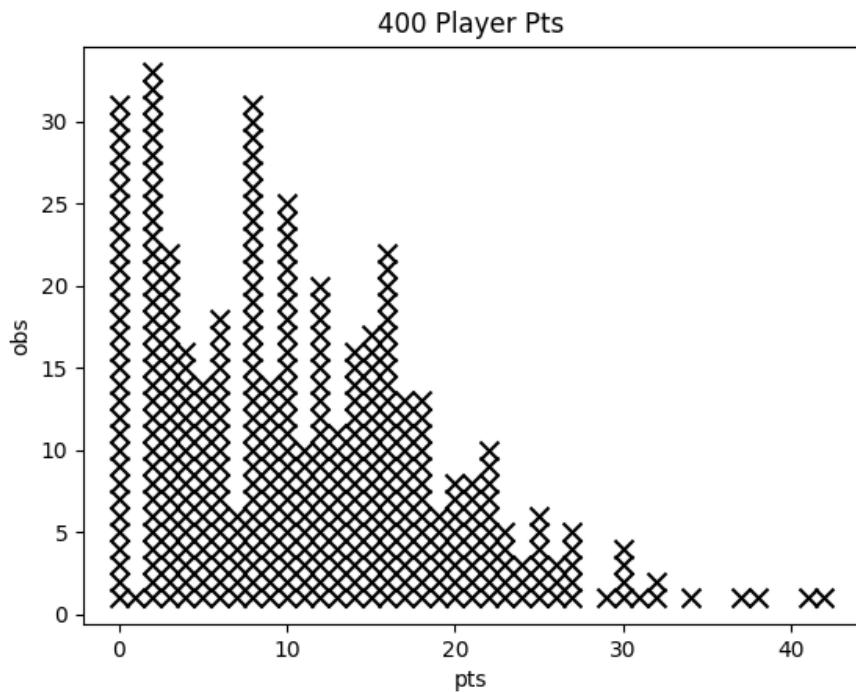
name	date	team	opp	pts
J. Isaac	20191125	ORL	DET	10
R. Bullock	20200105	NYK	LAC	9
B. Forbes	20191129	SAS	LAC	12
J. Green	20200804	LAC	PHX	10
L. Williams	20200806	LAC	DAL	6
J. Green	20191129	LAC	SAS	16
N. Vucevic	20200212	ORL	DET	19
K. Alexander	20200806	MIA	MIL	0
R. Westbrook	20200811	HOU	SAS	20
J. Ingles	20200304	UTA	NYK	8
L. Thomas	20200804	BKN	MIL	6
K. O Quinn	20200811	PHI	PHX	9
D. Green	20200128	GSW	PHI	9
I. Smith	20200807	WAS	NOP	18
D. Green	20191022	LAL	LAC	28
I. Bonga	20200301	WAS	GSW	6
J. Collins	20200226	ATL	ORL	26
I. Bonga	20200811	WAS	MIL	10
K. Johnson	20200807	SAS	UTA	4
T. McConnell	20200806	IND	PHX	8
T. Herro	20200806	MIA	MIL	20
J. McGee	20200803	LAL	UTA	0
W. Iwundu	20191220	ORL	POR	2
C. Silva	20200806	MIA	MIL	0
T. Ross	20191129	ORL	TOR	7

Let's arrange these smallest to largest, marking each with an x and stacking x's when a player's points shows up multiple times. Make sense? Like this:



**Figure 0.1:** 25 Player Points

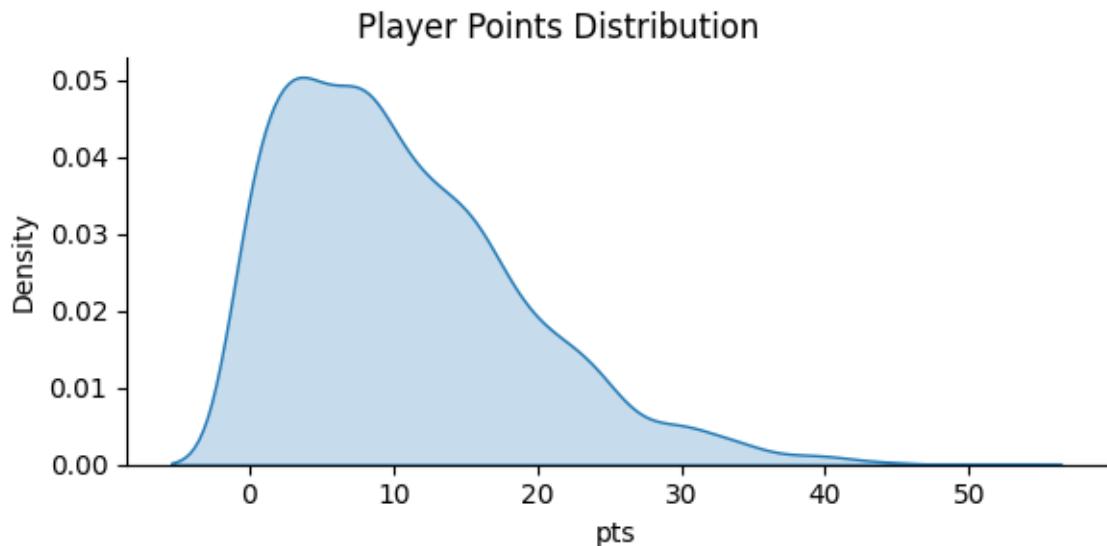
Interesting, let's up it to 400 observations:



**Figure 0.2:** 400 Player Points

These stacked x's show us the *distribution* of points scored. Distributions are the key to statistics. When a player goes into a game and says, "I wonder how many points I'll score tonight?" he's picking out one of these little x's 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, points only come in whole numbers, 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 - Player Points

The technical name for these curves are **kernel densities**<sup>1</sup>.

We're not going to get into the math, but for our purposes (mainly plotting them), we can mentally treat these curves as smoothed out stacks of `x`'s.

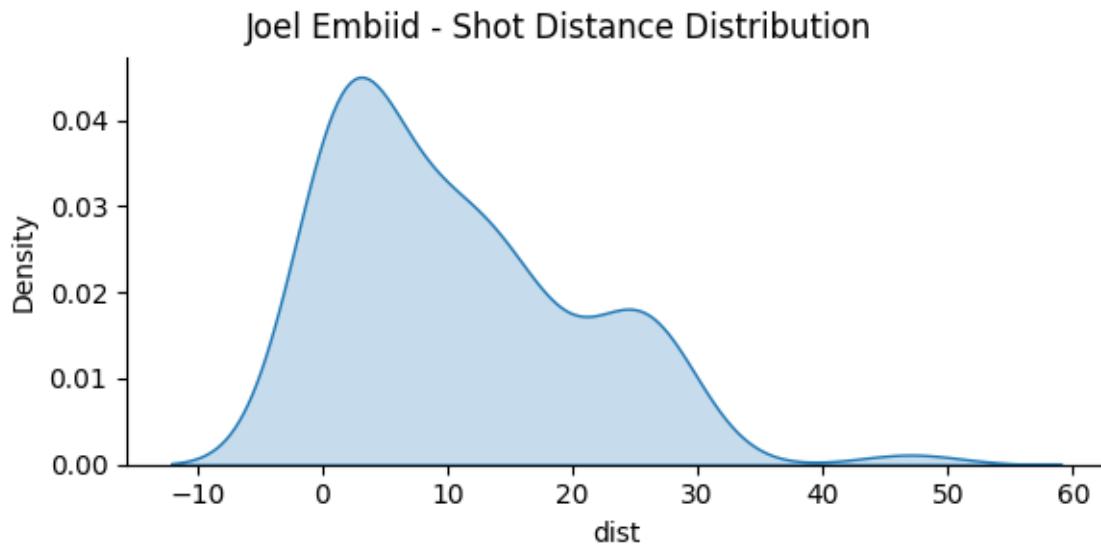
Don't worry about the values of the y-axis (0.05, 0.04, etc). They're just set (or *scaled*) so that the whole area under the curve equals 1. That's convenient — half the area under the curve is 0.5, a quarter is 0.25, etc — but it doesn't change the shape. We could take this exact same curve and double the y values. It wouldn't look any different, but the area under it would be 2 instead of 1.

## Summary Stats

Let's look at another distribution, say Joel Embiid's shot distance:

---

<sup>1</sup>Kernel densities are more flexible than traditional probability density functions like the normal or student T distribution. They can follow the “bumpiness” of our stacked `x`'s — but they're more complicated mathematically.



**Figure 0.4:** Kernel Density - Shot Distance - Joel Embiid

This gives us a good overview of what to expect: usually Embiid is shooting from inside 16 feet (about 75% of the area under the curve is to the left of 16). 40% of his shots are inside 5 feet. About 15% are three pointers.

Viewing and working with complete distributions like this is the gold standard for understanding a variable, but it isn't always practical.

**Summary statistics** describe (or *summarize*) a distribution with just a few numbers. Sometimes they're called *point estimates*. This makes sense because they're reducing the entire two dimensional distribution to a single number (or *point*).

For example, at the **median**, half area under the curve is above, half below. Here, Embiid's median shot distance is 8.5 feet.

The median splits the distribution 50-50, but you can divide it wherever you want. 10-90, 90-10, 79.34-20.66, whatever.

These are called **percentiles**, and they denote the area to the *left* of the curve. So at the 10th percentile, 10% of the area under the curve is to the left. The 10th percentile of Embiid's shot distance is 1 foot.

## Summary Statistics in Pandas

*Note: the code for this section is in the file `06_01_summary.py`. We'll pick up right after loading the shot, game and team data into DataFrames `dfs`, `dfg`, `dft` respectively.*

Pandas lets us calculate any percentile we want with the `quantile` function. Here's shot distance:

```
In [1]: dfs['dist'].quantile(.9)  
Out[1]: 26.0
```

So 90% of shots were taken within 26 feet; 10% of shots were outside that.

You can also calculate multiple statistics — including several percentiles — at once in Pandas with `describe`.

```
In [2]: dfs[['dist', 'value']].describe()  
Out[2]:  
          dist           value  
count    16876.000000  16876.000000  
mean      13.432745   2.377992  
std       10.621506   0.484900  
min       0.000000   2.000000  
25%      2.000000   2.000000  
50%      13.000000   2.000000  
75%      24.000000   3.000000  
max      80.000000   3.000000
```

## Mean aka Average aka Expected Value

The second line of `describe` gives the **mean**. Other terms for the mean include *average* or *expected value*. Expected value refers to the fact that the mean is the probability weighted sum of all outcomes.

So take our shot distances. Here's how often distances 0-9 feet show up in our sample:

```
In [3]: dfs['dist'].value_counts(normalize=True).sort_index().head(10)  
Out[3]:  
0      0.080410  
1      0.113356  
2      0.084499  
3      0.041716  
4      0.025421  
5      0.022754  
6      0.019021  
7      0.017362  
8      0.017421  
9      0.016651
```

So 8% of shots were from 0 feet, 11% from 1 feet, 8.4% from 2 feet, etc.

So the *expected value* of shot distance would be:

$$0.08 \times 0 + 0.11 \times 1 + 0.08 \times 2 + 0.04 \times 3 + \dots = 13.43$$

This is the same thing as summing up all the individual distances in our data and dividing by however many there are (16,876 here). That's not 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".

Also note the normal average you learned in school is just a special case where the probability of every observation is the same. For example, say we want to calculate the average (or *expected*) weight in pounds of a random member of the Bucks starting lineup. There are five players, so the "probability" of picking any given one is  $1/5=0.20$ .

Then we have:

$$0.2 \times 242 + 0.2 \times 222 + 0.2 \times 282 + 0.2 \times 220 + 0.2 \times 214 = 236$$

Which of course is the same as summing up  $242 + 222 + 282 + 220 + 214$  and dividing by 5.

## Variance

Other summary stats summarize dispersion — how close or far apart the observations are to each other. The standard deviation, for instance, is (basically) the average *distance to the mean*. To calculate it you (1) figure out the average of all your observations, (2) figure out how far each observation is from that average, and (3) take the average of that<sup>2</sup>. It's smaller when values are tightly clustered around their mean, larger when values are more dispersed.

## Distributions vs Summary Stats

There are times when using summary statistics in place of distributions is useful. Usually it's because distributions are difficult to manipulate mathematically.

For example, say you've projected each player's point distribution for a game. And you want to use these player distributions to predict total score for a whole team. Summary stats make this easy. You: (1) figure out the mean for each player, then (2) add them up. That's a lot easier than trying to combine player distributions and take the mean of that.

But although summary statistics are convenient, I think working with distributions directly is under-rated, especially because Python + a few other libraries make it so easy.

---

<sup>2</sup>Technically, there are some slight modifications, but it's the general idea.

## Density Plots in 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 a library called matplotlib, which is very powerful, but is also trickier to learn<sup>3</sup>. One of the main problems is that there are multiple ways of doing the same thing, which can get confusing.

So instead we'll use the *seaborn* library, which is built on top of matplotlib. Though not as widely used, seaborn is still very popular and comes bundled with Anaconda. We'll go over some specific parts that I think provide the best mix of functionality and ease of use.

*The code below is in 06\_01\_summary.py. Note we're picking up in the plotting section, so we've imported the libraries and loaded our data. Note the convention is to import seaborn as sns.*

When making density plots in seaborn you basically have control over three things. Let's call them "levers". By manipulating these levers (and the data you're feeding to it) you can make a million different plots.

All three of these levers are columns in your data. Two of them are optional.

### Basic, One-Variable Density Plot

The only thing we *have* to tell seaborn is the name of the variable we're plotting. This is a column in our data.

Let's look at shot distance (our column `dist` in our data `dfs`). This is how we'd do it:

```
In [1]: g = sns.FacetGrid(dfs).map(sns.kdeplot, 'dist', shade=True)
```

This code has two parts: it's (1) making a *FacetGrid*, which is a powerful, seaborn plotting type. Then (2) it's *mapping* the `kdeplot` function (kernel density plot) to it, with shading option on. Shading is optional, but I think it looks better.

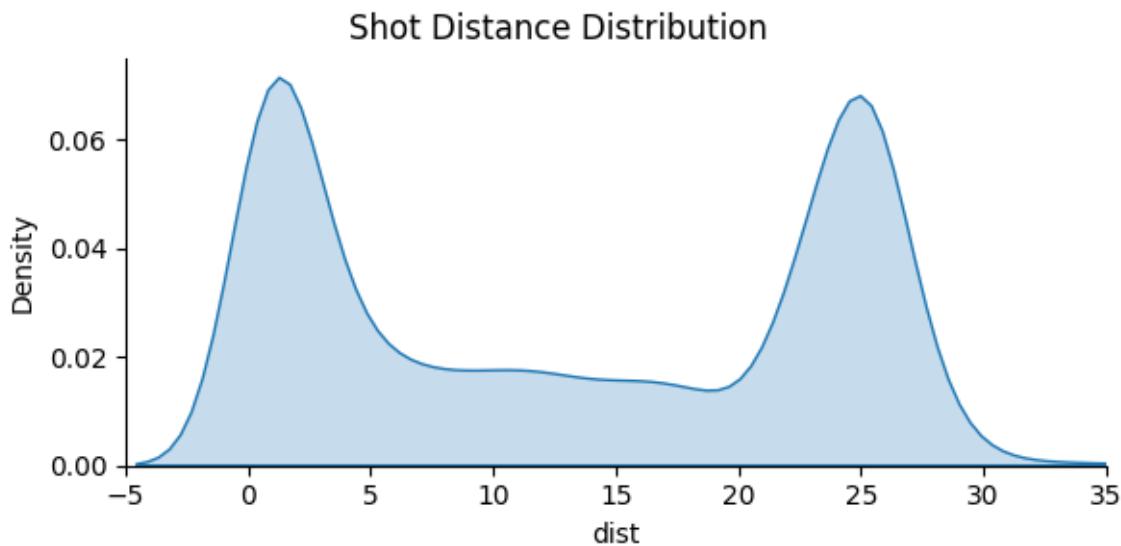
I usually write it on two lines because I think it's a little easier to understand:

```
In [2]:  
g = (sns.FacetGrid(dfs)  
     .map(sns.kdeplot, 'dist', shade=True))
```

---

<sup>3</sup>Note: "trickier" is relative, if after reading this book you wanted to sit down with the documentation and master matplotlib, you could do it, and it wouldn't be hard. It's just not as intuitive as Pandas or Seaborn (where mastering a few basic concepts will let you do a million things) and might involve more memorization.

Here's what it looks like:



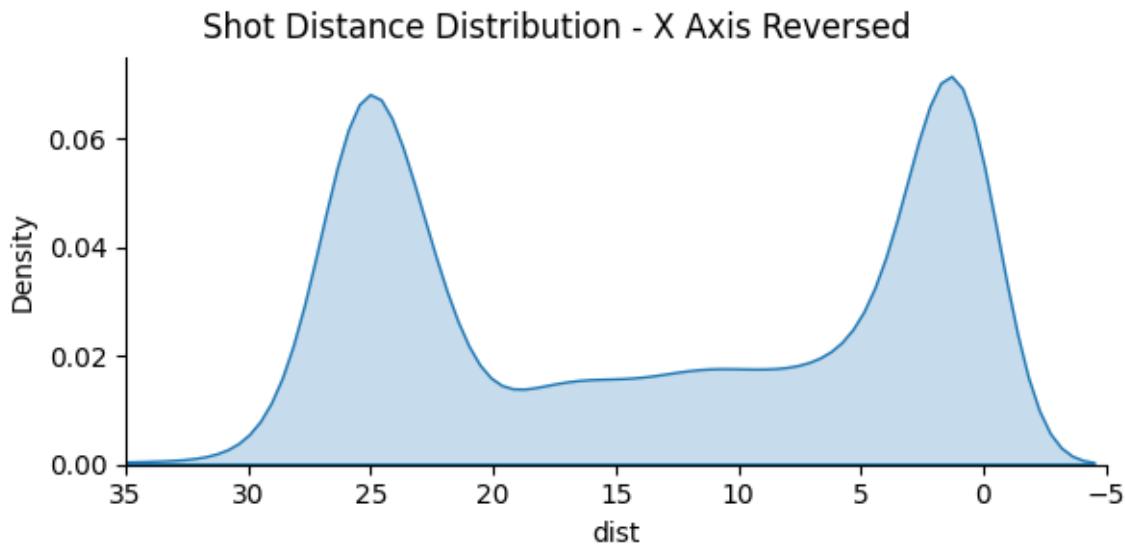
**Figure 0.5:** Distribution of Shot Distance

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 them later.

There are faster ways to make this specific plot, but I'd recommend sticking with the FacetGrid-then-map approach because it's easy to extend.

Personally, I think it looks better with the x-axis inverted, i.e. the shorter shots on the right. Maybe because that's how they show shot charts on TV. I didn't actually know how to do this off the top of my head, but a bit of Googling + Stack Overflow shows this works:

```
In [3]: g.ax.invert_xaxis()
```



**Figure 0.6:** Distribution of Shot Distance with X-Axis Reversed

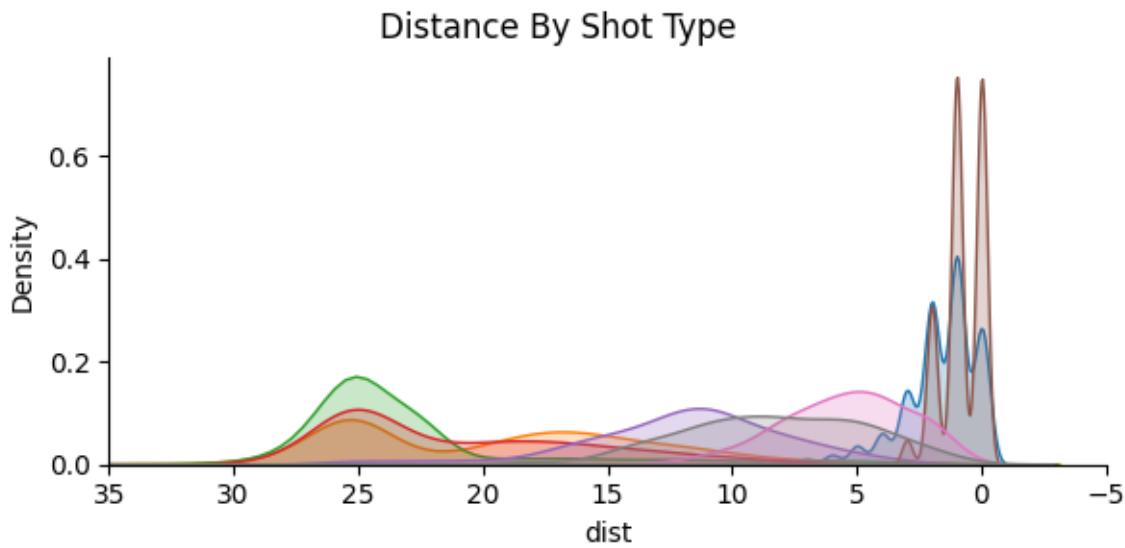
Perfect.

### Seaborn “Levers” - Slicing and Dicing Plots

Seaborn’s killer feature is how easy it makes creating and comparing multiple plots.

For example, say we want separate plots by shot type. Now we can introduce our second lever, the `hue` keyword.

```
In [4]:  
g = (sns.FacetGrid(dfs, hue='shot_type')  
    .map(sns.kdeplot, 'dist', shade=True))
```



**Figure 0.7:** Distribution of Distance by Shot Type

### Aside: Jittering

Looking at this, I'm not a huge fan of the “spikiness” in the distributions for layups and dunks, where we see big peaks at 0, 1, 2, etc. This is happening because of the data was collected in x, y coordinates that had to take on whole numbers. In real life you'd expect something more continuous.

One common way to fix things like this is by adding a small amount of random data to each observation. This is called *jittering*, and can improve the visuals.

We can do this with the `random` library, which is built into Python:

```
In [5]: import random
```

`random` lets you take draws from miscellaneous distributions. For example, here's how you'd get a random number between 0 and 1:

```
In [6]: random.uniform(0, 1)
Out[6]: 0.6388718454170276
```

(Note you'll see something different; this is a random).

We could add a random number between 0 and 1 to all of our layup and dunk distances, and it'd work fine. The only problem is it'd be slightly *biased*. If we took the average before and after jittering, the after-jitter average would be higher because `random.uniform(0, 1)` is always positive. Instead,

let's use a random number that's centered around 0. How about the classic, bell-shaped, *normal* distribution? In `random` it's called `gauss`:

Here are 10 values:

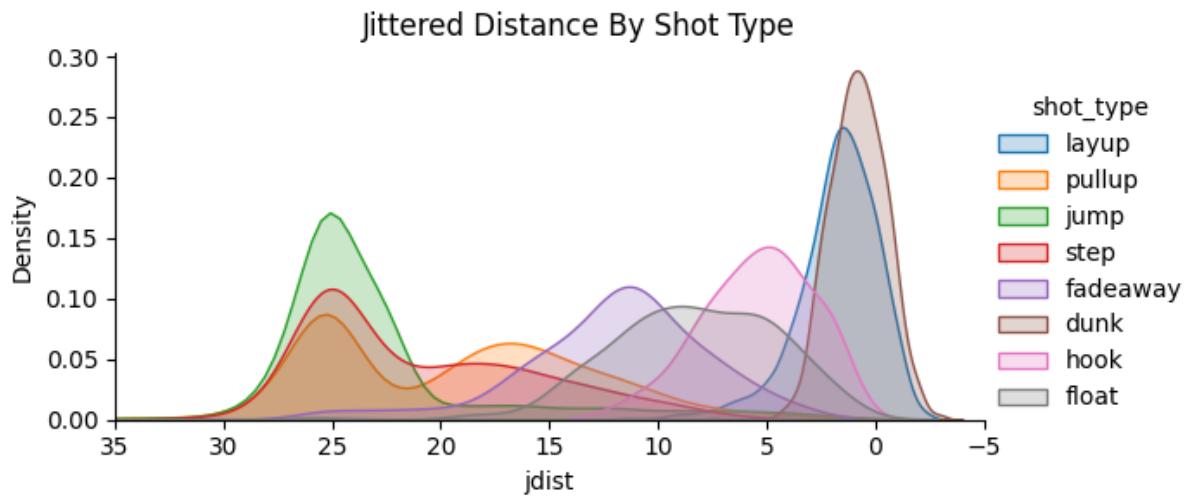
```
In [7]: [random.gauss(0, 1) for _ in range(10)]  
Out[7]:  
[0.8313728638876453,  
 0.17057235664185233,  
 0.2536999616218848,  
 -0.18657013635142755,  
 -1.2022458299176941,  
 0.5579663225316337,  
 -0.44036955593965005,  
 2.835632599157855,  
 0.11302542631295645,  
 0.9804724156385064]
```

So let's make a new column, `jdist` (for jittered distance) that's the same as `dist`, but has a `random.gauss(0, 1)` added to every layup and dunk distance:

```
In [8]:  
dfs['jdist'] = dfs['dist']  
  
dfs.loc[df['shot_type'] == 'layup', 'dist'] = dfs['dist'].apply(  
    lambda x: x + random.gauss(0, 1))  
  
dfs.loc[df['shot_type'] == 'dunk', 'dist'] = dfs['dist'].apply(  
    lambda x: x + random.gauss(0, 1))
```

And redoing our plot:

```
In [9]:  
g = (sns.FacetGrid(dfs, hue='shot_type')  
     .map(sns.kdeplot, 'jdist', shade=True))
```



**Figure 0.8:** Distribution of Jittered Distance by Shot Type

That's better. Great. Now you know about jittering. Back to seaborn.

### More Seaborn Levers

So far: we made a distribution plot with a single variable. Then we added the `hue` argument to draw multiple plots.

Remember, `hue` is a *column* of data. By passing the `shot_type` column, we're telling seaborn to plot different distributions of `jdist` (with different colors, or hues) for each value of `shot_type`. So we have one density plot of `jdist` when `shot_type='dunk'`, another for `shot_type='stepback'` etc.

This plot does a nice job showing the distribution of distance from the basket across different types of shots. What if we wanted to add in another dimension, say, whether the shot went in?

We can use our third lever — the `col` keyword.

```
In [10]:  
g = (sns.FacetGrid(dfs, hue='shot_type', col='made')  
     .map(sns.kdeplot, 'jdist', shade=True))
```



**Figure 0.9:** Distribution of Distance by Shot Type and Made or Not

This draws separate plots (on different columns) for every value of whatever you pass to `col`. So we have two plots: one when `made=True`, another for `made=False`.

Within each of these, the `hue=shot_type` draws separate curves for each shot type.

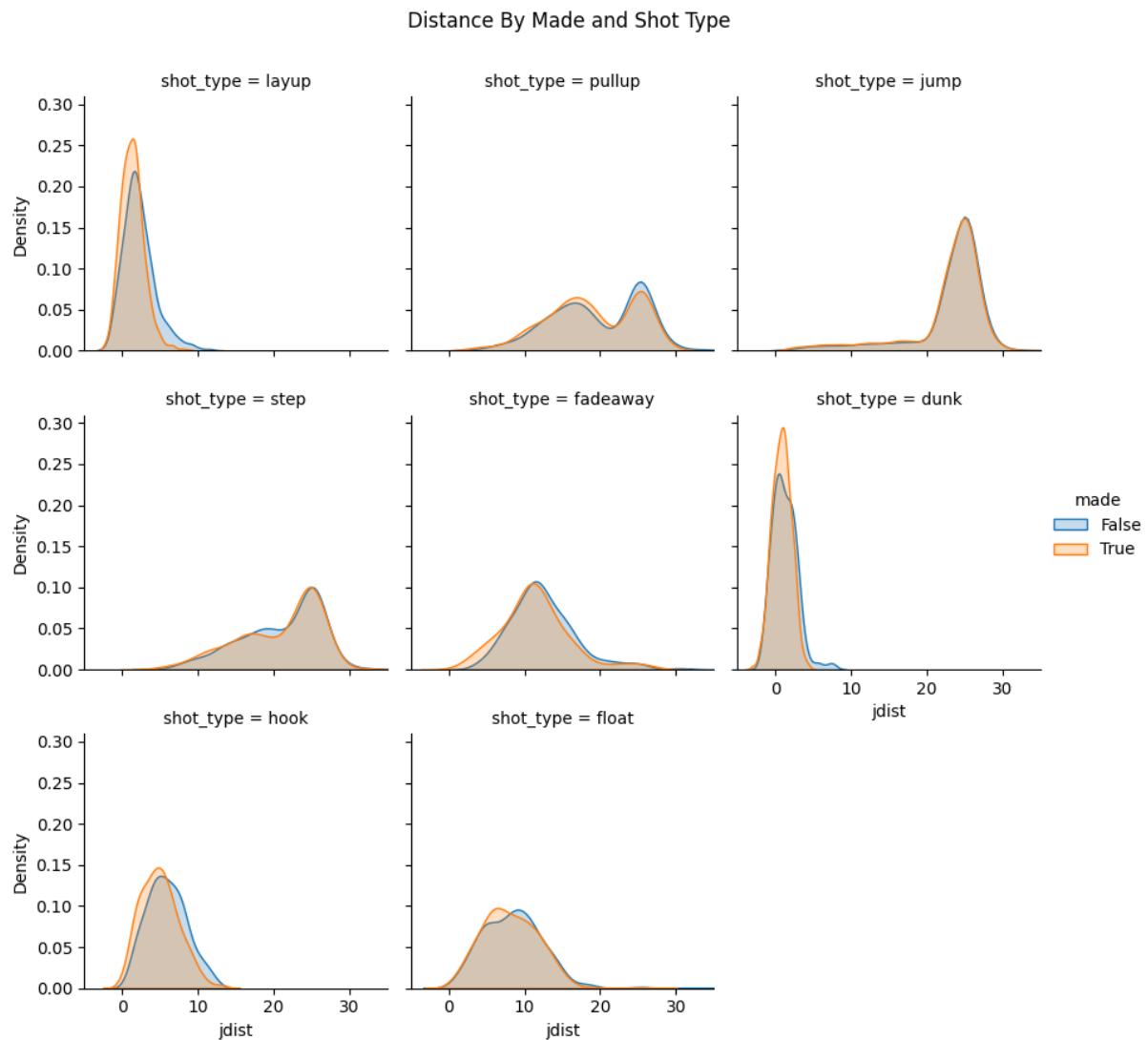
Are made shots closer to the basket? It looks like it for layups and dunks, but it's hard to tell for the other shot types. Let's try swapping `hue` and `col` around.

Before we do it see if you can mentally imagine (or at least describe in words) what this will do.

Imagined it? OK, let's try it:

```
In [11]:  
g = (sns.FacetGrid(dfs, col='shot_type', hue='made', col_wrap=3)  
     .map(sns.kdeplot, 'jdist', shade=True))
```

(Note, don't worry about the `col_wrap` argument for now, it's aesthetic and we'll cover it later.)



**Figure 0.10:** Distribution of Distance by Made and Shot Type

With `col='shot_type'`, we can see seaborn is making a separate plot (in its own column) for each value of `shot_type`. Within each of those, its drawing two distributions, one for made shots, the other for missed.

This makes it clear that — for most shots — players are more likely to make closer shots. This is especially true for hook shots. It doesn't appear to make a difference for regular jumpers.

Again, all three of these variables — `jdist`, `shot_type`, and `made` 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.

## Manipulating data for seaborn

Seaborn needs everything in a certain format, but sometimes data is structured differently.

*Note: the following code uses our games data, which we loaded into a DataFrame `dfg` at the beginning of `06_01_summary.py`.*

Say we want to plot the distributions of points scored by home vs away teams using our game data. That's no problem conceptually, except our points data is in separate columns.

We currently have this:

```
In [1]:  
dfg[['date', 'home_team', 'away_team', 'home_pts', 'away_pts']].head()  
  
Out[1]:  
      date  home_team  away_team  home_pts  away_pts  
0  2019-10-22       TOR       NOP      130      122  
1  2019-10-22       LAC       LAL      112      102  
2  2019-10-23       CHA       CHI      126      125  
3  2019-10-23       IND       DET      110      119  
4  2019-10-23       ORL       CLE      94       85
```

But seaborn needs something more like this:

```
      date  team  pts  location  
0  2019-10-22  TOR  130    home  
1  2019-10-22  LAC  112    home  
5  2019-10-22  NOP  122   away  
6  2019-10-22  LAL  102   away  
2  2019-10-23  CHA  126    home  
3  2019-10-23  IND  110    home  
4  2019-10-23  ORL  94     home  
7  2019-10-23  CHI  125   away  
8  2019-10-23  DET  119   away  
9  2019-10-23  CLE  85    away
```

They contain the same information, we've just changed the granularity (from game to team-game) 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 (`game`) and a location (`home` or `away`) — moves that score (e.g. `game['home_score']`) to a score column, then adds in another column indicating which location we're dealing with. So this:

```
def home_away_pts_df(df, location):
    df = df[['date', f'{location}_team', f'{location}_pts']]
    df.columns = ['date', 'team', 'pts']
    df['location'] = location
    return df
```

And to use it with `home`:

```
In [2]: home_away_pts_df(dfg, 'home').head()
Out[2]:
      date team  pts location
0  2019-10-22  TOR  130    home
1  2019-10-22  LAC  112    home
2  2019-10-23  CHA  126    home
3  2019-10-23  IND  110    home
4  2019-10-23  ORL   94    home
```

That's half of what we want. We just need to call it separately for home and away, then stick the resulting DataFrames on top each other (like a snowman). Recall vertical stacking is done with the `concat` function, which takes a list of DataFrames.

So we need a list of DataFrames: one with home teams, another with away teams, then we need to pass them to `concat`. Let's do it using a list comprehension.

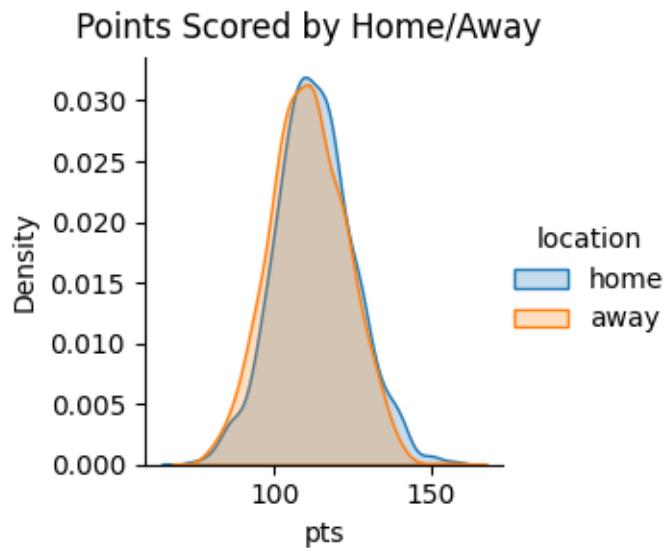
```
In [3]:
pts_long = pd.concat([
    home_away_pts_df(dfg, loc) for loc in ['home', 'away']],
    ignore_index=True)
```

Now we have what we want: the points and home/away in two separate columns. And we can pass it to seaborn:

```
In [4]:
g = (sns.FacetGrid(pts_long, hue='location')
     .map(sns.kdeplot, 'pts', shade=True))
```

The plot is easy once our data is 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 0.11:** Distribution of Points by Home/Away

And it does indeed look like the home team's point distribution is shifted slightly right vs the away team.

#### Extended Example: Game to team-game - all stats

We just went from the game level, to the game and team level for the `pts` column. Now, let's write some code to do this for *all* the home and away columns. This is really more Pandas practice than it is seaborn, but it will let us make some interesting plots, so let's do it here.

Basically, what we want to do is take this function:

```
def home_away_pts_df(df, location):
    df = df[['date', f'{location}_team', f'{location}_pts']]
    df.columns = ['date', 'team', 'pts']
    df['location'] = location
    return df
```

— and apply it to *all* the columns, not just the points. So we'll end up with something like this:

	team	opp_team	pts	opp_pts	...	fg_pct	opp_fg_pct	stl	opp_stl
0	TOR	NOP	130	122	...	0.408	0.422	7	4
1	LAC	LAL	112	102	...	0.519	0.435	8	4
2	CHA	CHI	126	125	...	0.511	0.467	3	11
3	IND	DET	110	119	...	0.461	0.526	8	5
4	ORL	CLE	94	85	...	0.430	0.375	12	8

Notice how we're including both the team points in a column (`pts`), *and* also the opponents points (`opp_pts`). We're doing this for every stat (`fgm`, `opp_fgm`, etc).

Just like with points, we'll have to run it on home and away teams separately.

It's always easier to start with a real example of what we're trying to do, so let's start with the home teams:

```
In [5]: location = 'home'
```

When teams are home, their opponents are away:

```
In [6]: opp = 'away'
```

Now we can identify all the “team” columns (`pts`, `fgm`, `fga` etc):

```
In [7]: team_cols = [x for x in dfg.columns if x.startswith(location)]
```

```
In [8]: team_cols
```

```
Out[8]:
```

```
['home_team', 'home_pts', 'home_fgm', 'home_fga', 'home_fg_pct',
 'home_fg3m', 'home_fg3a', 'home_fg3_pct', 'home_ftm', 'home_fta',
 'home_ft_pct', 'home_oreb', 'home_dreb', 'home_reb', 'home_ast',
 'home_stl', 'home_blk', 'home_tov', 'home_pf', 'home_plus_minus']
```

And the “opponent” columns (`opp_pts`, `opp_fgm`, `opp_fga`):

```
In [9]: opp_cols = [x for x in dfg.columns if x.startswith(opp)]
```

```
In [10]: opp_cols
```

```
Out[10]:
```

```
['away_team', 'away_pts', 'away_fgm', 'away_fga', 'away_fg_pct',
 'away_fg3m', 'away_fg3a', 'away_fg3_pct', 'away_ftm', 'away_fta',
 'away_ft_pct', 'away_oreb', 'away_dreb', 'away_reb', 'away_ast',
 'away_stl', 'away_blk', 'away_tov', 'away_pf', 'away_plus_minus']
```

Now let's rename these, just like we did in `home_away_pts_df`.

```
In [11]: df_team = dfg[team_cols]

In [12]: df_team.columns = [x.replace(f'{location}_', '') for x in
                           df_team.columns]

In [13]: df_team.head()
Out[13]:
   team  pts  fgm  fga  fg_pct  ...  reb  ast  stl  blk  tov  pf
0  TOR  130   42  103  0.408  ...    57   23    7   3   16  24
1  LAC  112   42   81  0.519  ...    45   24    8   5   14  25
2  CHA  126   45   88  0.511  ...    41   28    3   4   19  18
3  IND  110   41   89  0.461  ...    36   26    8   6   15  28
4  ORL   94   37   86  0.430  ...    46   24   12    4   13  18
```

And do the same for the opponent columns:

```
In [14]: df_opp = dfg[opp_cols]

In [15]: df_opp.columns = [x.replace(opp, 'opp') for x in df_opp.columns]

In [16]: df_opp.head()
Out[16]:
   opp_team  opp_pts  opp_fgm  opp_fga  opp_fg_pct  ...  opp_reb  opp_ast  opp_stl  opp_blk  opp_tov  opp_pf
0      NOP     122      43     102     0.408  ...        9      19      34
1      LAL     102      37      85     0.519  ...        7      14      24
2      CHI     125      49     105     0.511  ...        4      10      20
3      DET     119      41      78     0.461  ...        6      18      21
4      CLE      85      33      88     0.430  ...        2      16      15
```

Cool. So what the first row is telling us is our team (TOR) scored 130 points, made 42 field goals, etc. Their opponent (NOP) scored 122 and made 43 field goals.

We can stick the team and opponent data together horizontally with `concat` since they have the same index:

```
In [17]: dfg_wide = pd.concat([df_team, df_opp], axis=1)

In [18]: dfg_wide.head()
Out[18]:
   team  pts  fgm  fga  fg_pct  ...  opp_stl  opp_blk  opp_tov  opp_pf
0  TOR  130   42  103  0.408  ...        4       9      19      34
1  LAC  112   42   81  0.519  ...        4       7      14      24
2  CHA  126   45   88  0.511  ...       11       4      10      20
3  IND  110   41   89  0.461  ...        5       6      18      21
4  ORL   94   37   86  0.430  ...        8       2      16      15
```

That's a good start, but remember that's just the *home* teams. We also want this same DataFrame for *away* teams, e.g. we want a row when NOP is `team` and TOR is `opp_team`.

## Learn to Code with Basketball

---

The easiest way to do that is put what we have so far in a function:

```
def home_away_all(df, location):
    if location == 'home':
        opp = 'away'
    elif location == 'away':
        opp = 'home'

    team_cols = [x for x in df.columns if x.startswith(location)]
    team_cols

    opp_cols = [x for x in df.columns if x.startswith(opp)]
    opp_cols

    df_team = df[team_cols]
    df_team.columns = [x.replace(f'{location}_', '') for x in df_team.columns]
    df_team.head()

    df_opp = df[opp_cols]
    df_opp.columns = [x.replace(opp, 'opp') for x in df_opp.columns]
    df_opp.head()

    return pd.concat([df_team, df_opp], axis=1)
```

Then call in on the away teams:

```
In [19]: home_away_all(dfg, 'away').head()
Out[19]:
   team  pts  fgm  fga  fg_pct  ...  opp_stl  opp_blk  opp_tov  opp_pf
0  NOP  122   43  102  0.422  ...       7       3      16      24
1  LAL  102   37   85  0.435  ...       8       5      14      25
2  CHI  125   49  105  0.467  ...       3       4      19      18
3  DET  119   41   78  0.526  ...       8       6      15      28
4  CLE   85   33   88  0.375  ...      12       4      13      18
```

Perfect.

Once we have that we stick them together vertically so we have all teams:

```
In [20]
dfg_wide = pd.concat([
    home_away_all(dfg, 'home'),
    home_away_all(dfg, 'away')])
```

The only other thing is that there are a few columns that don't fit the home/away format. Specifically:

```
In [21]:  
other_cols = [x for x in dfg.columns  
             if not ((x.startswith('home')) or (x.startswith('away')))]  
  
In [22]: other_cols  
Out[22]: ['game_id', 'date', 'min', 'bubble', 'sample', 'season']
```

Let's add those into our `dfg_wide` data too, creating a `win` column while we're at it:

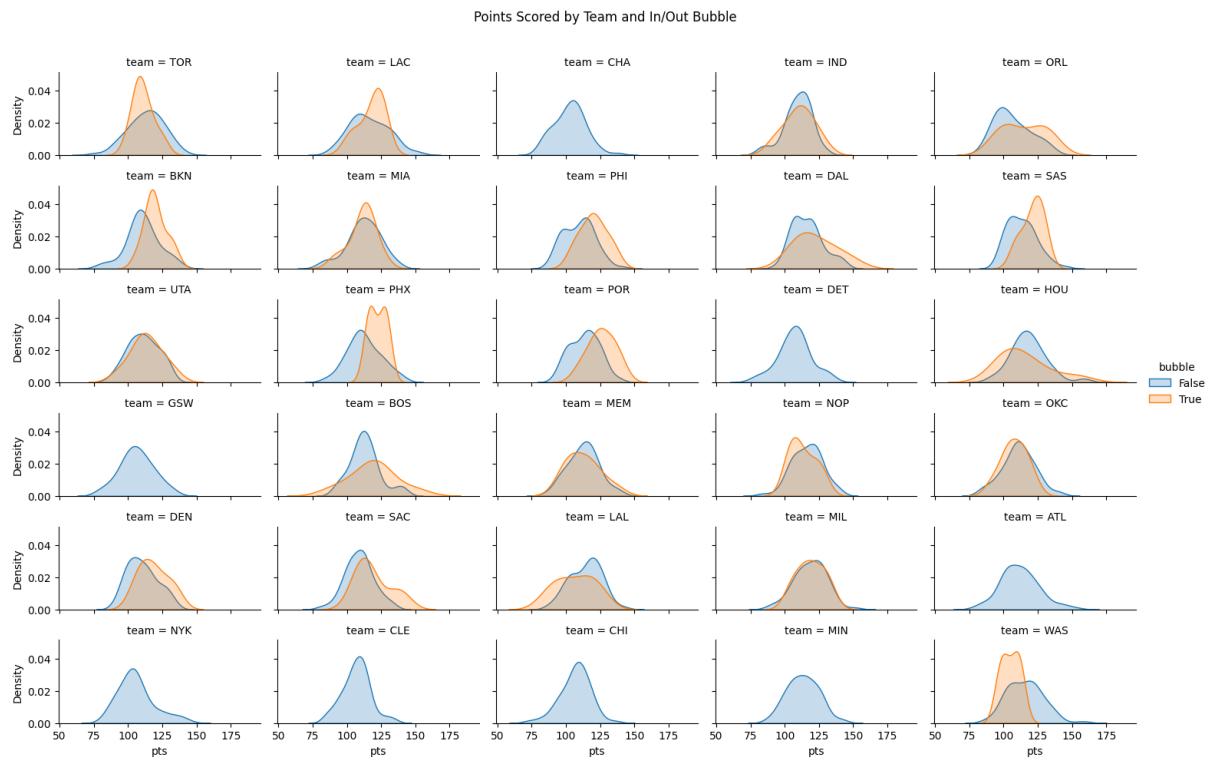
```
In [23]: df_all = pd.concat([dfg_wide, dfg[other_cols]], axis=1)  
  
In [23]: df_all['win'] = dfg_wide['pts'] > dfg_wide['opp_pts']  
  
In [24]: df_all.head()  
Out[24]:  
   team  pts   fgm   fga   fg_pct   fg3m   ...   bubble   sample   season   win  
0  TOR  130    42   103   0.408     14   ...   False   False  2019-20  True  
1  LAC  112    42    81   0.519     11   ...   False   True  2019-20  True  
2  CHA  126    45    88   0.511     23   ...   False   False  2019-20  True  
3  IND  110    41    89   0.461      8   ...   False   False  2019-20 False  
4  ORL   94    37    86   0.430      9   ...   False   False  2019-20  True
```

Perfect. Now our data is in the first, team-game format, and we can do interesting things like look at scoring by team and bubble:

```
In [25]:  
g = (sns.FacetGrid(df_all, hue='bubble', col='team', col_wrap=5)  
     .map(sns.kdeplot, 'pts', shade=True))  
g.add_legend()
```

## Learn to Code with Basketball

---



**Figure 0.12:** Distribution of Points by Team and Bubble

## Relationships Between Variables

Another common analysis task is to look at the *relationship* between two (or more) variables.

For example, maybe we're interested in the relationship between field goal percentage and total points. How much do they tend to move together?

### Scatter Plots with Python

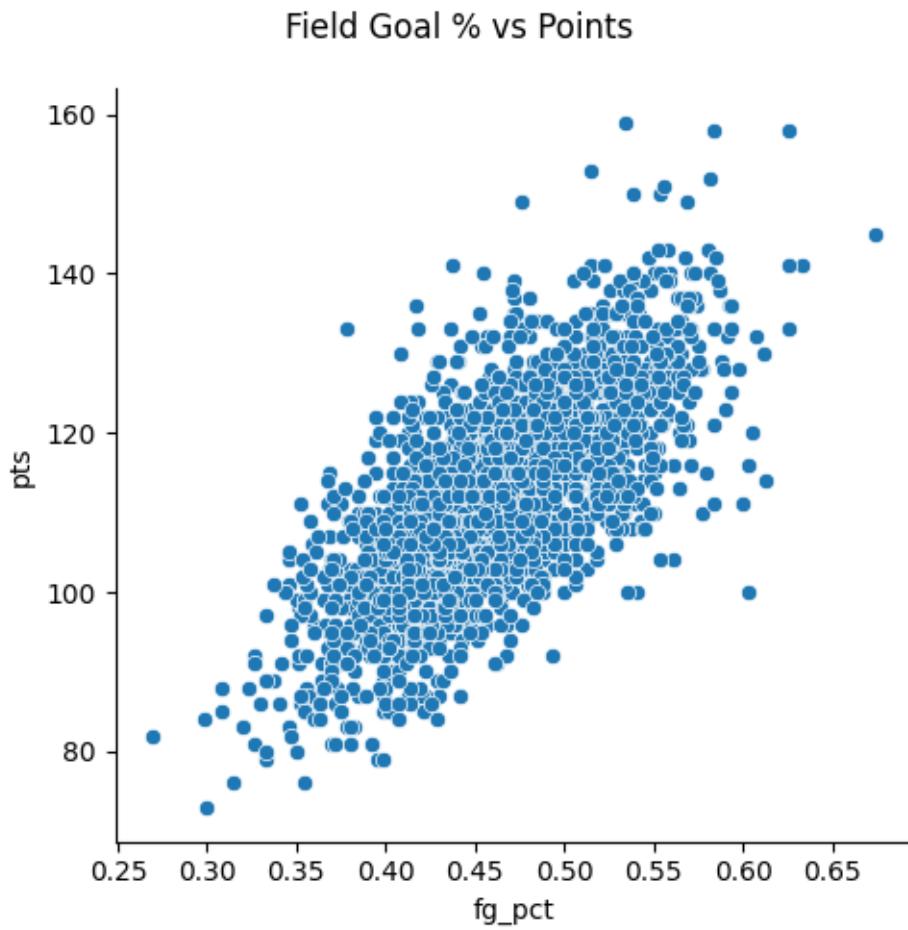
The most useful tool for visualizing relationships is a *scatter plot*. Scatter plots are just plots of points on a standard coordinate system. One of your variables is your x axis, the other your y axis. Each observation gets placed on the graph. The result is a sort of “cloud” of points. The more the cloud moves from one corner of the graph to another, the stronger the relationship between the two variables.

So say we want to look at the relationship between field goal percentage and points.

*Note: we're still in 06\_01\_summary.py. We're using df\_all, which we created in the section above.*

To make scatter plots in seaborn we use the basic function `relplot` (for relationship plot):

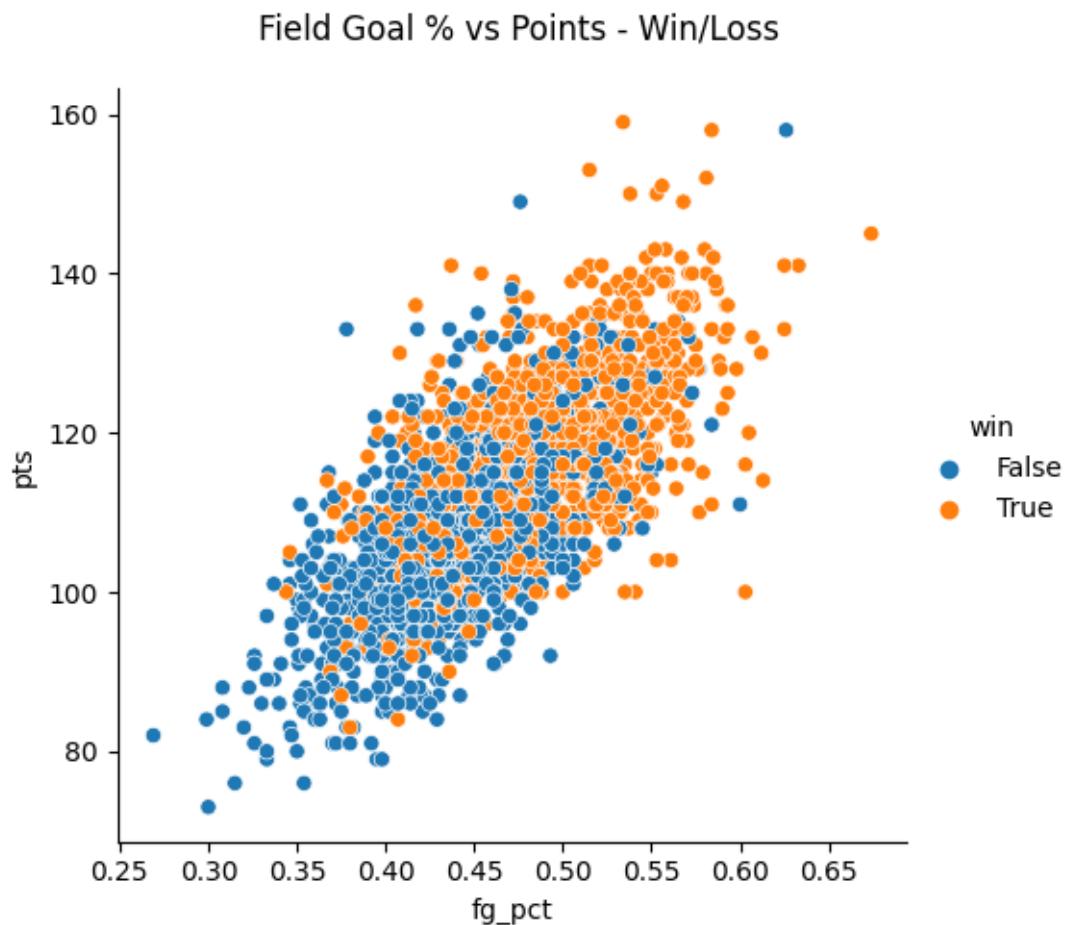
```
In [1]: sns.relplot(x='fg_pct', y='pts', data=df_all)
```



**Figure 0.13:** Field Goal Percentage vs Points

It looks this cloud of points is moving up and to the right, showing a *positive* relationship between field goal percentage and points. This makes sense. Later, we'll look at other, numeric ways of quantifying the strength of a relationship, but for now let's color our plots by whether the team won or not. Just like the density plots, we do this using the `hue` keyword.

```
In [2]: sns.relplot(x='fg_pct', y='pts', hue='win', data=df_all)
```



**Figure 0.14:** Field Goal Percentage vs Points, Win-Loss

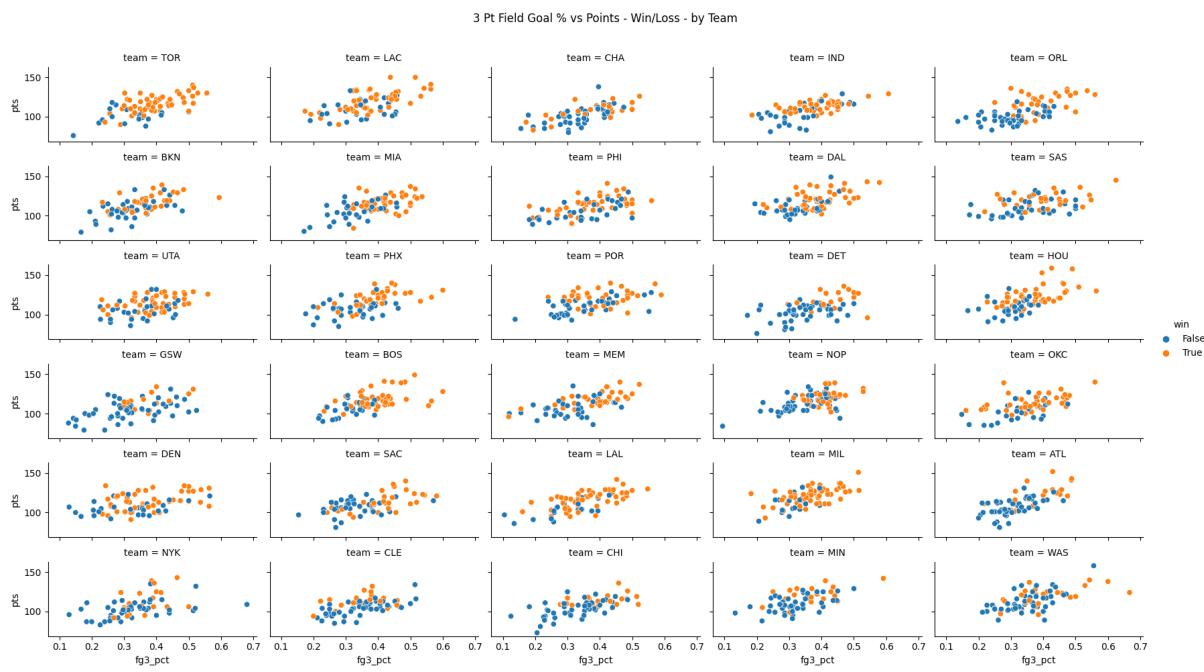
This is sort of interesting. As we'd expect, the top-right observations are more likely to win.

Let's add team. Just like density plots, we can do this with the `col` argument.

```
In [3]:  
sns.relplot(x='fg3_pct', y='pts', col='team', hue='win', col_wrap=5,  
            data=df_all)
```

## Learn to Code with Basketball

---



**Figure 0.15:** 3 Pt Field Goal Percentage vs Points, by Win-Loss and Team

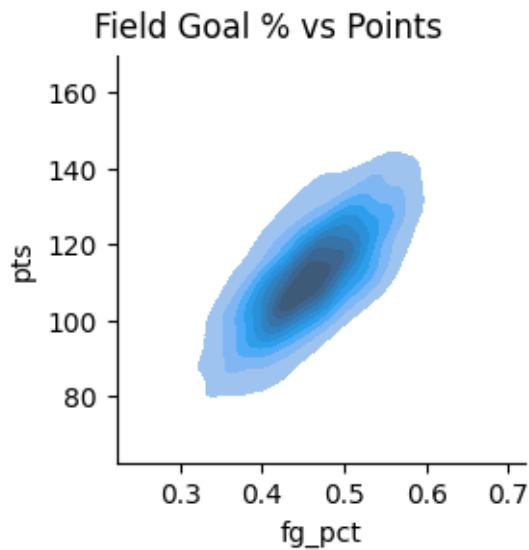
## Contour Plots

One visually interesting tweak is to turn these scatter charts into *contour* plots.

Earlier we started out by looking at distributions as stacked x's and moved to area under the curve. Contour plots are basically a way to do this with scatter plots.

The contour version of points vs field goal percentage:

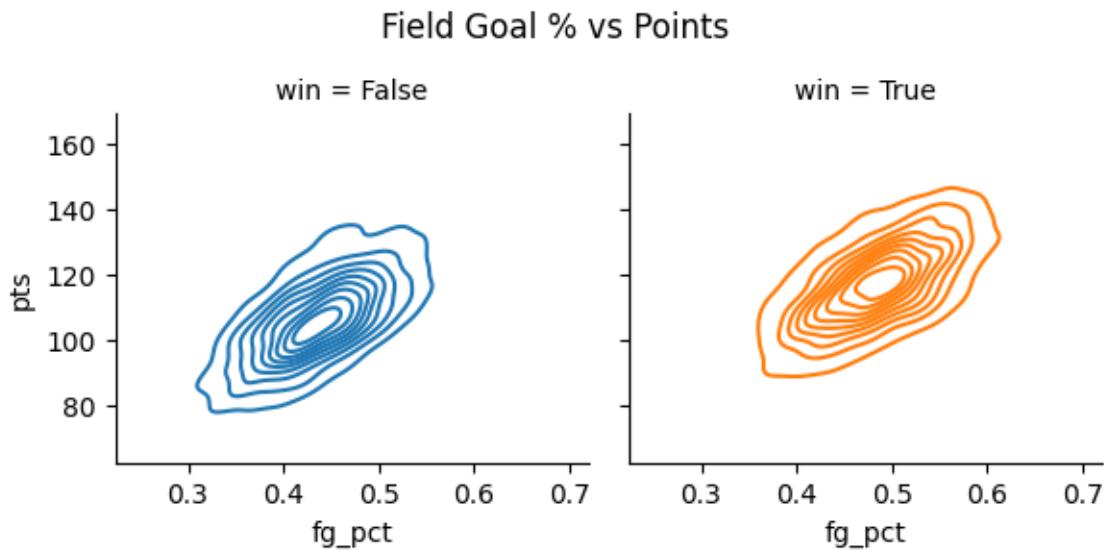
```
In [7]:  
g = (sns.FacetGrid(df_all)  
     .map(sns.kdeplot, 'fg_pct', 'pts', shade=True))
```



**Figure 0.16:** Field Goal Percentage vs Points - Contour

Note we included the `shade=True` option. We can leave that off, in which case we just get the lines, which is more like what'd you see on a map. This can be useful for visualizing finer distinctions between relationships or adding other hues:

```
In [8]:  
g = (sns.FacetGrid(df_all, hue='win', col='win')  
     .map(sns.kdeplot, 'fg_pct', 'pts'))
```



**Figure 0.17:** Field Goal Percentage vs Points by Win - No Shading

## Correlation

Just like a median or mean summarizes a variable's distribution, there are statistics that summarize the strength of the relationship between variables.

The most basic is **correlation**. The usual way of calculating correlation (called *Pearson's*) summarizes the tendency of variables to move together with a number between -1 and 1.

Variables with a -1 correlation move perfectly together in opposite directions; variables with a 1 correlation move perfectly together in the same direction.

Note: “move perfectly together” doesn’t necessarily mean “exactly the same”. Variables that are exactly the same *are* perfectly correlated, but so are simple, multiply-by-a-number transformations. For example, number of three pointers (say `n`) is perfectly correlated with the number of points scored from three pointers (`3*n`).

A correlation of 0 means the variables have no relationship. They’re *independent*.

One interesting way to view correlations across multiple variables at once (though still in pairs) is with a *correlation matrix*. In a correlation matrix, the variables you’re interested in are the rows and columns. To check the correlation between any two variables, you find the right row and column and look at the value.

For example, let’s look at a correlation matrix for a few of our team-game stats. In Pandas you get a correlation matrix with the `corr` function:

```
In [1]:  
df_all[['pts', 'reb', 'win', 'fg3_pct', 'fgm', 'blk',  
       'opp_fg3_pct']].corr().round(2)  
  
Out[1]:  
      pts   reb   win  fg3_pct    fgm    blk  opp_fg3_pct  
pts  1.00  0.11  0.46   0.55  0.83  0.06     0.06  
reb  0.11  1.00  0.32  -0.10  0.09  0.17    -0.34  
win  0.46  0.32  1.00   0.34  0.39  0.17    -0.34  
fg3_pct  0.55 -0.10  0.34   1.00  0.43  0.01     0.01  
fgm   0.83  0.09  0.39   0.43  1.00  0.08     0.07  
blk   0.06  0.17  0.17   0.01  0.08  1.00    -0.08  
opp_fg3_pct  0.06 -0.34 -0.34   0.01  0.07 -0.08     1.00
```

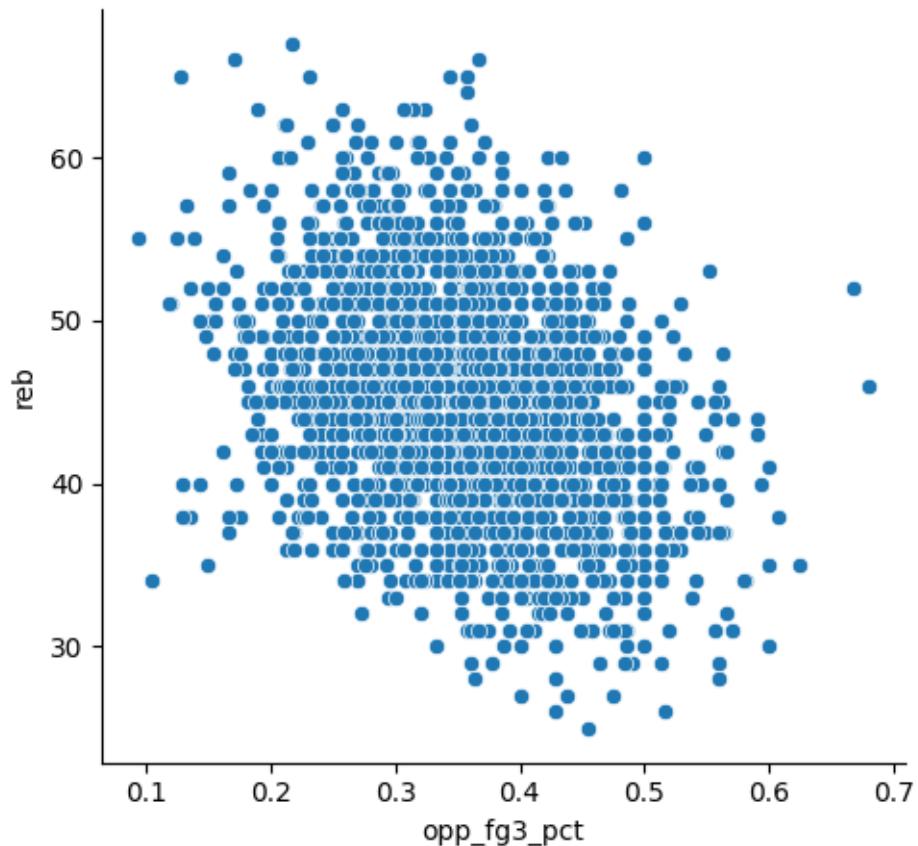
Note that the diagonal elements are all 1. Every variable is perfectly correlated with itself. Also note the matrix is symmetrical around the diagonal. This makes sense. Correlation is like multiplication — order doesn't matter. The correlation between points and field goals made is the same as the correlation between field goals made and points.

To pick out any individual correlation pair, we can look at the row and column we're interested in. So we can see the correlation between opponents three point percentage and number of rebounds is -0.34.

Here's that in scatter plot form:

```
In [2]: g = sns.relplot(x='opp_fg3_pct', y='reb', data=df_all)
```

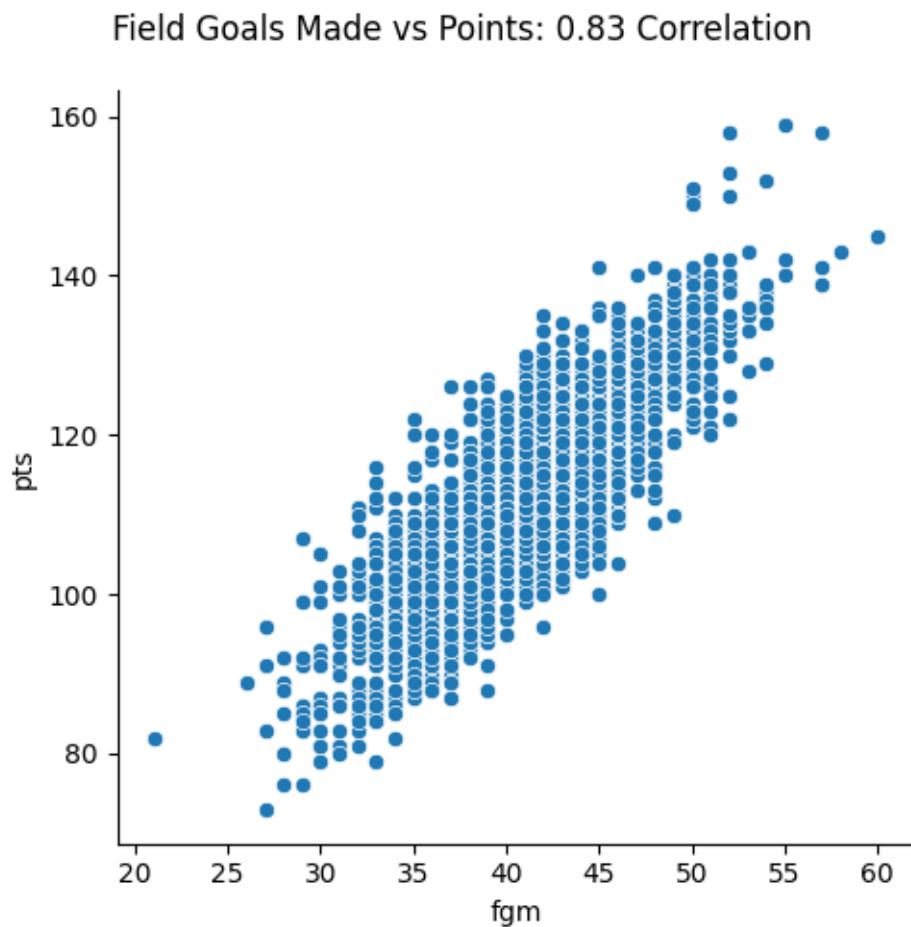
Opponent 3 Point % vs Number of Rebounds: -0.34 Correlation



**Figure 0.18:** Opp 3 Pt % vs Rebounds: -0.34 correlation

So that's a correlation of -0.34. Compare that to the correlation between field goals made and number of points, with is 0.83:

```
In [3]: g = sns.relplot(x='fgm', y='pts', data=df_all)
```



**Figure 0.19:** FGM vs Points: 0.83 correlation

These 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 in general. But when one of the variables is some measure of *time* (e.g. game number 1-82, year 2009-2022, time left in game) a lineplot is usually more useful.

You make a lineplot by passing the argument `kind='line'` to `relplot`. When working with line plots, 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.

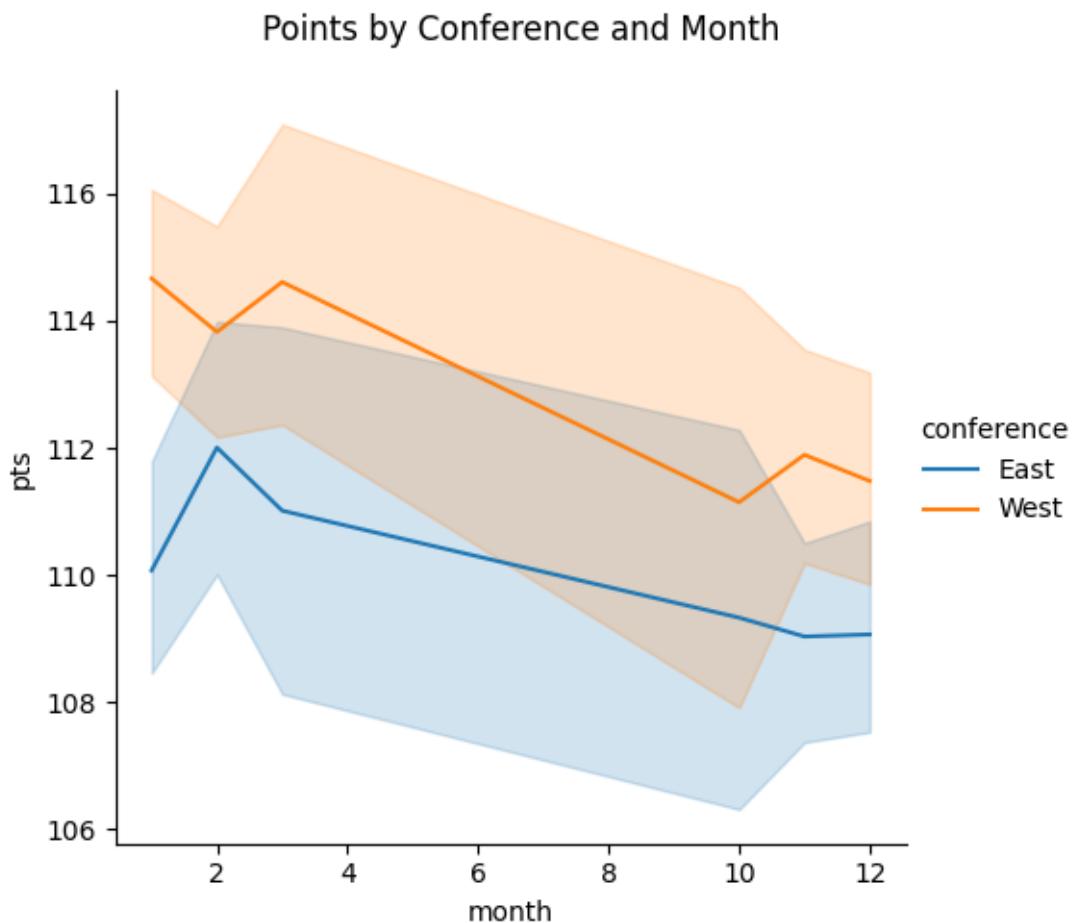
Let's try plotting points by conference and month. We'll pick up after we've merged team division and conference information into `df_all`.

First we need to get the month variable. We can extract it from `date` like this:

```
In [1]: df_all['month'] = df_all['date'].str[5:7].astype(int)
```

Now we can do a line plot of points by team and conference:

```
In [2]:  
g = sns.relplot(x='month', y='pts', kind='line',  
                 data=df_all.query("~bubble"), hue='conference')
```



**Figure 0.20:** Points by Month, Conference

Woah. What's happening here? We wanted a *line* plot. We are getting some lines, but we're also seeing a bunch of shading.

We told seaborn we wanted month on the *x* axis and wins on the *y* axis. And we wanted different plots (hues) for each conference.

But remember, our data is at the *team* level, so for any given month (our *x* axis) and conference (*hue*) there are multiple observations. Say we're looking at November 2019, Eastern Conference:

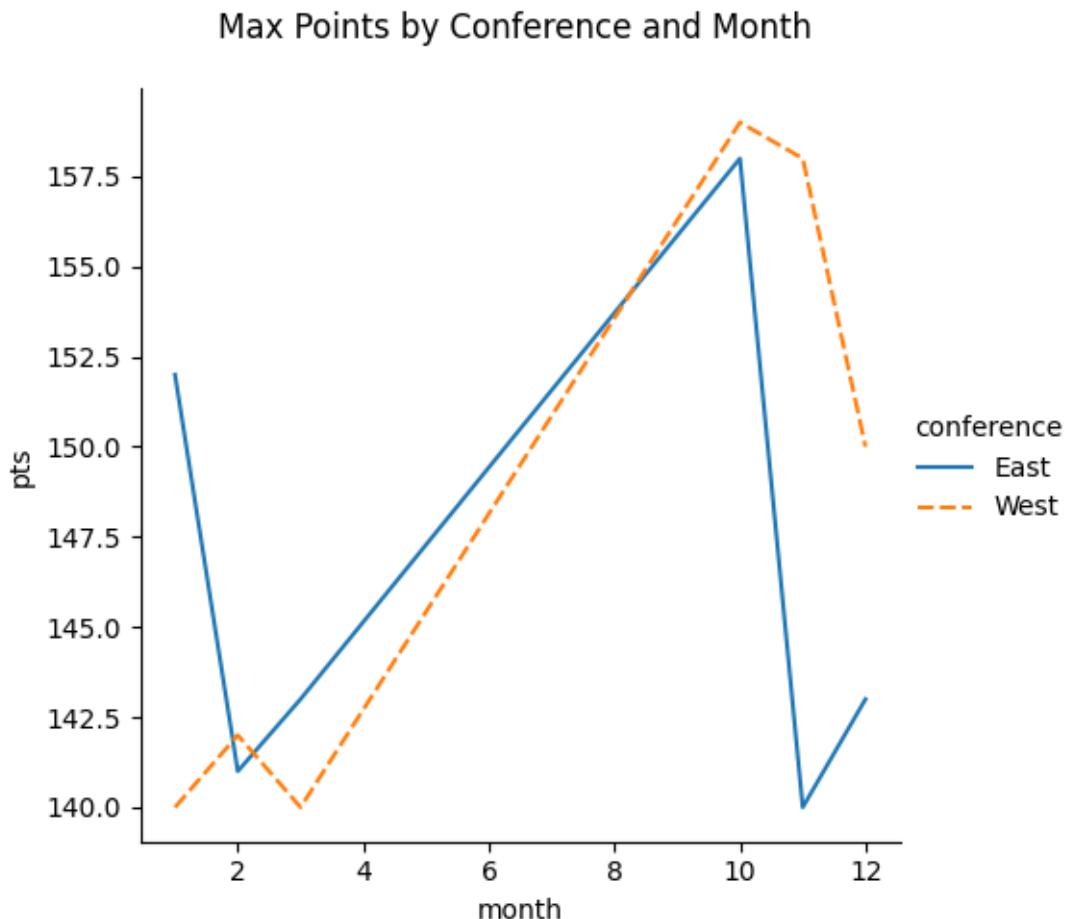
```
In [3]:  
df_all.loc[(df_all['month'] == 11) & (df_all['conference'] == 'East'),  
           ['date', 'team', 'opp_team', 'pts', 'opp_pts', 'conference']]  
       .sort_values('date').head()
```

```
Out[3]:  
      date team opp_team  pts  opp_pts conference  
973  2019-11-01  DET     CHI  106      112      East  
358  2019-11-01  BKN     HOU  123      116      East  
210  2019-11-01  IND     CLE  102      95       East  
1143 2019-11-01  BOS     NYK  104      102      East  
1822 2019-11-01  NYK     BOS  102      104      East
```

Instead of plotting separate lines for each team, 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 month and conference — say the *maximum* points by conference and month, it'll plot just the single lines.

```
In [4]:  
max_pts = (df_all.query("~bubble").groupby(['conference', 'month'],  
                                         as_index=False)['pts'].max())  
  
In [5]:  
g = sns.relplot(x='month', y='pts', kind='line', style='conference',  
                 data=max_pts, hue='conference')
```



**Figure 0.21:** Max Points vs Month

Note: if you look close, you'll see we included an additional fourth lever: `style`. It controls how the lines look (dotted, dashed, solid etc).

Like the density examples, the line version of `relplot` includes options for separating plots by columns.

## Plot Options

Seaborn provides 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.

We covered the most important parts of these plots above. But there are a few other cosmetic options that are useful. These are mostly the same for every type of plot we've looked at, so we'll go through

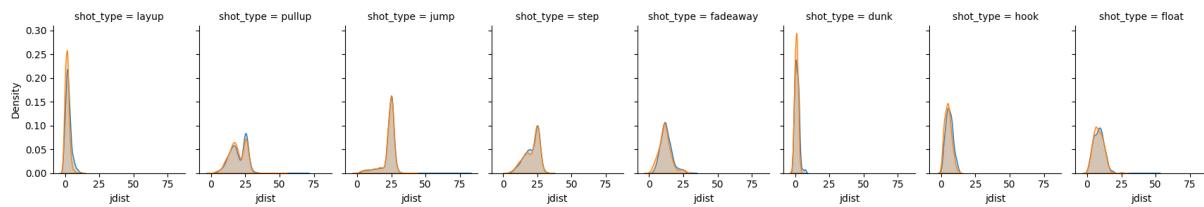
them all with one example.

Let's use our jittered distance by shot type distribution plot from earlier.

```
In [1]:  
g = (sns.FacetGrid(df, col='shot_type', hue='made')  
     .map(sns.kdeplot, 'dist', shade=True))
```

## Wrapping columns

By default, seaborn will spread all our columns out horizontally. With multiple columns that quickly becomes unwieldy:

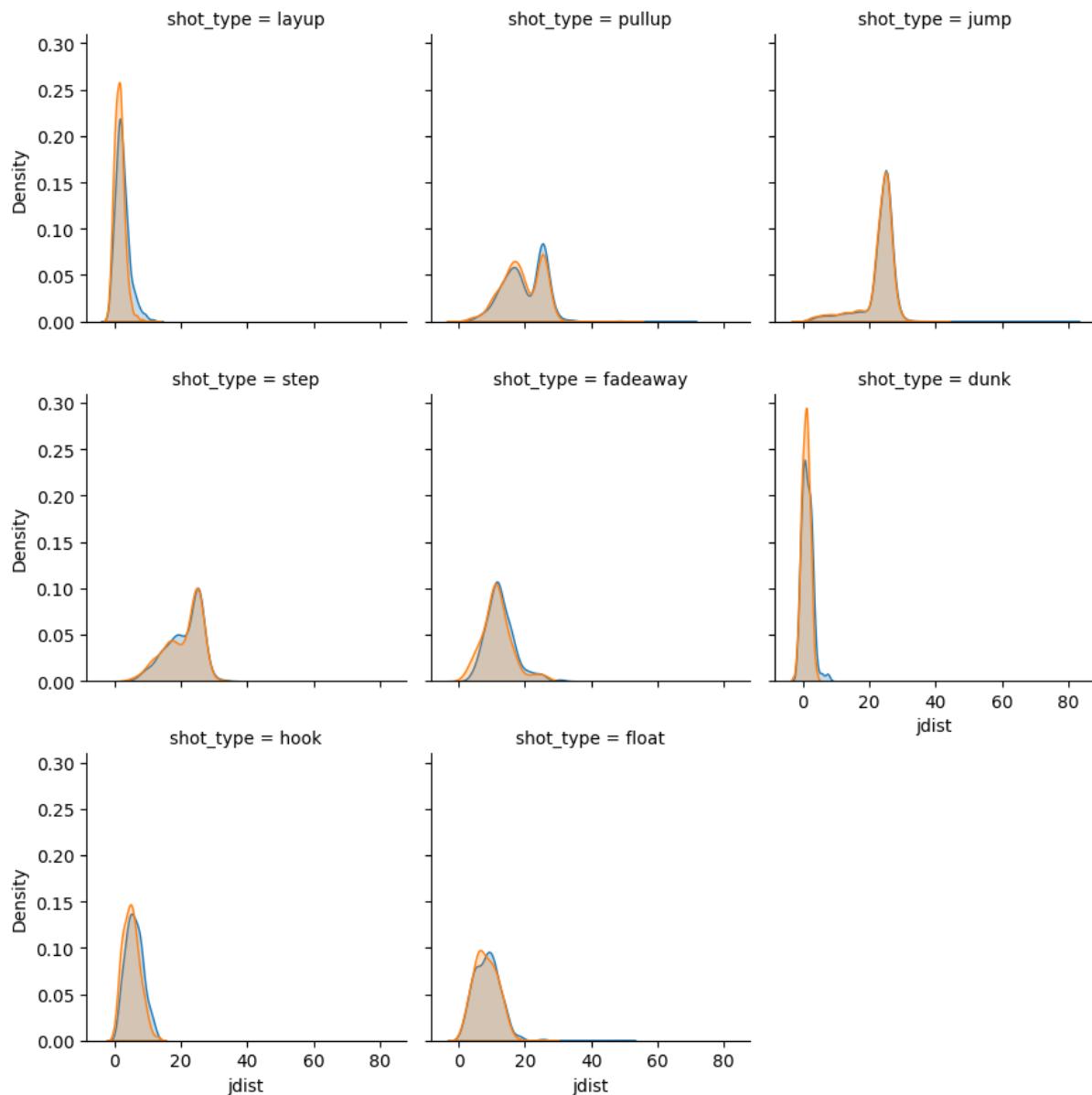


**Figure 0.22:** Distance by Shot Type - No col\_wrap

We can fix it with the `col_wrap` keyword, which will make seaborn start a new row after some number of columns.

```
In [2]:  
g = (sns.FacetGrid(df, col='shot_type', hue='made', col_wrap=3)  
     .map(sns.kdeplot, 'dist', shade=True))
```

Here it makes the plot much more readable.



**Figure 0.23:** Distribution by Shot Type - col\_wrap=3

### 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.

```
In [3]:  
g.fig.subplots_adjust(top=0.9) # adding a title  
g.fig.suptitle('Distribution of Shot Distances by Type')
```

I'm not sure why it's two steps — it 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.

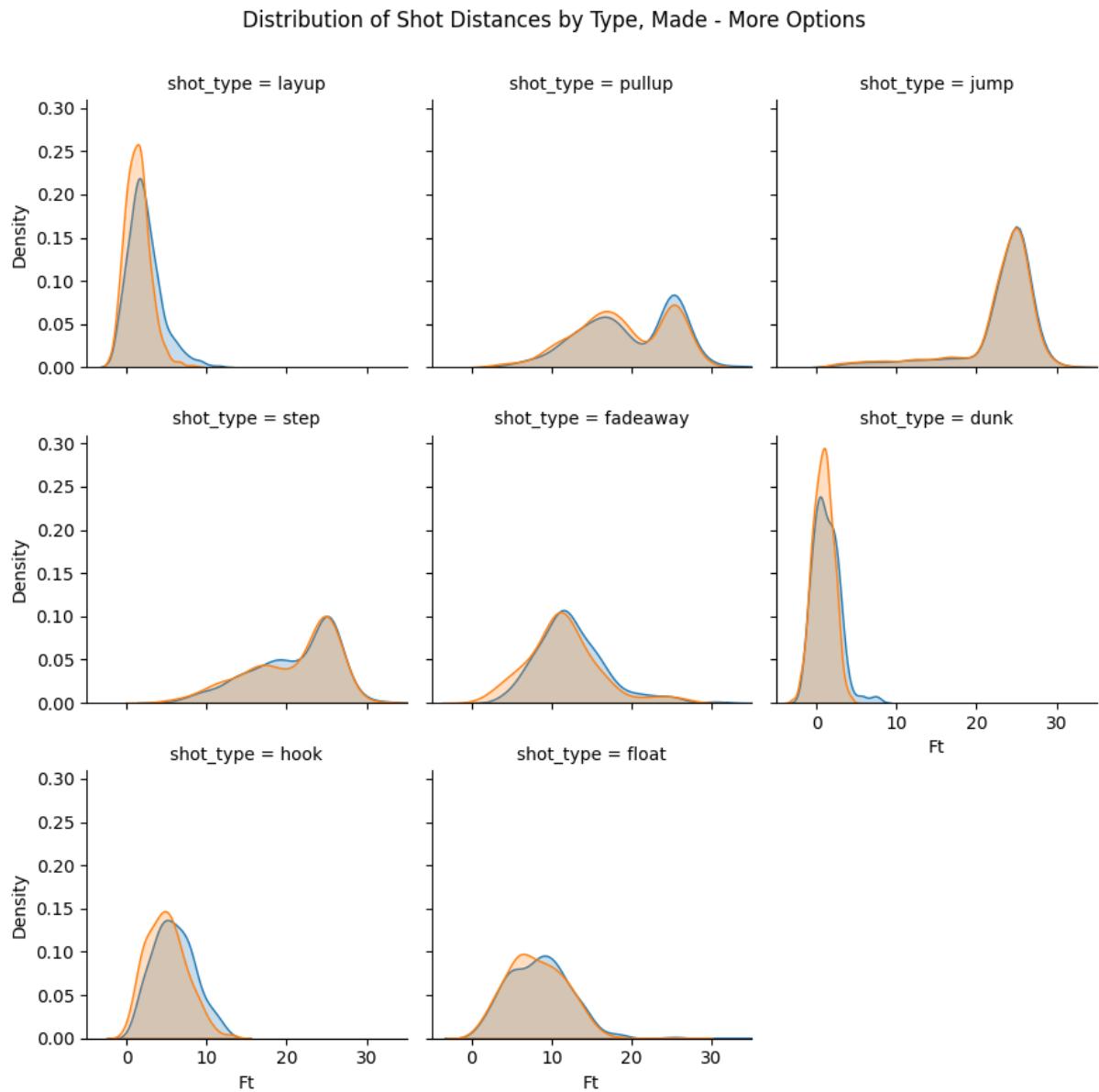
You do that via the `set` method.

```
In [4]:  
g.set(xlim=(-5, 40))
```

`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.

```
In [5]:  
g.set_xlabels('Ft')  
g.set_ylabels('Density')
```



**Figure 0.24:** Distribution by Shot Type - More Options

### Legend

For `relplot`, Seaborn will automatically add a legend if you use the `hue` keyword. If you don't want it you can pass `legend=False`.

For our `FacetGrid` then `map` approach you need to add it yourself:

```
In [6]: 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 1 (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`.

```
In [7]: g.savefig('shot_dist_type_made.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 usually 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.

## Shot Charts

We have a bunch of shot data, so let's see if we can make some shot charts.

### Shot Charts As Seaborn Scatter Plots

The key to making shot charts is that our shot data includes x, y coordinates:

*Note: the following code is in 06\_02\_shot\_chart.py. We'll pick up after loading the shot data and doing some light processing.*

```
In [1]: dfs[['name', 'dist', 'value', 'made', 'x', 'y']].head()
```

```
Out[1]:
```

	name	dist	value	made	x	y
0	L. James	2	2	True	-9	23
1	L. Shamat	26	3	False	201	178
2	D. Green	25	3	True	125	221
3	P. Beverley	26	3	False	117	239
4	A. Davis	18	2	False	96	162

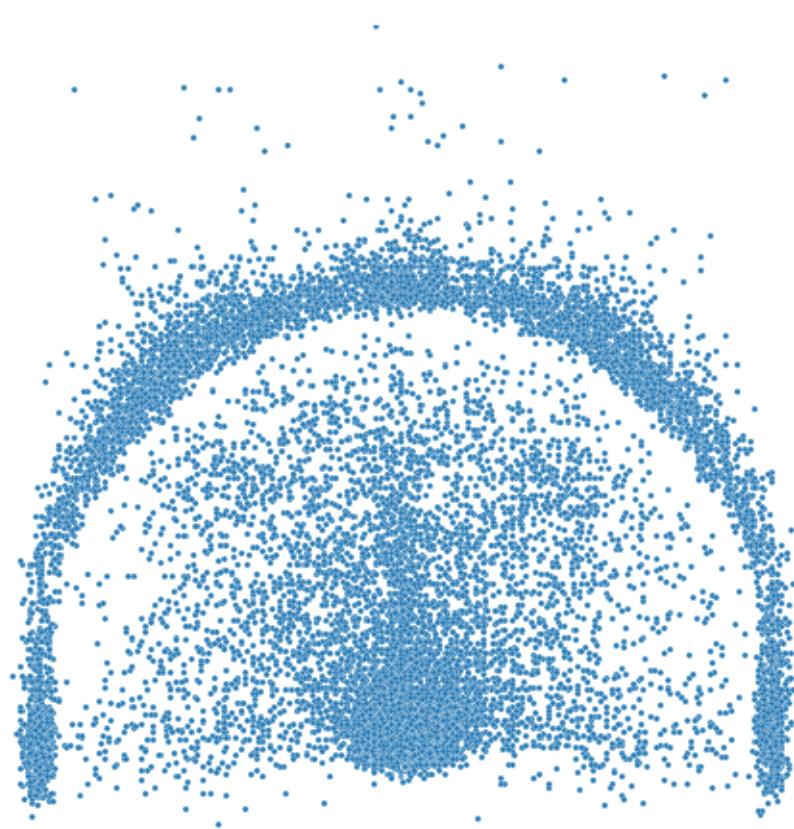
These coordinates represent a location on half of an NBA court. If we're standing at midcourt and looking at the basket, `y` gives our location left to right, `x` to the baseline.

By playing around with it, it looks to me like `x` is about -50 to 400 and `y` can be anywhere from -250 to 250. I'm not sure how they landed on this range of values specifically, but apparently that maps up to the 47 feet by 50 feet dimensions of half an NBA court.

We've already seen how a scatter plot shows two dimensions (columns) of data. So far we've been using it to visualize how variables move together, but there's no reason we can't use it for physical, x y coordinates too.

So let's do it, starting with all of the shot data in our sample:

```
In [2]:  
g = sns.relplot(data=df, x='x', y='y', kind='scatter', s=5)  
g.set(xlim=(-250, 250), ylim=(-50, 400), yticks=[], xticks=[],  
      xlabel=None, ylabel=None)  
g.despine(left=True, bottom=True)
```



**Figure 0.25:** Shot Chart - All Data

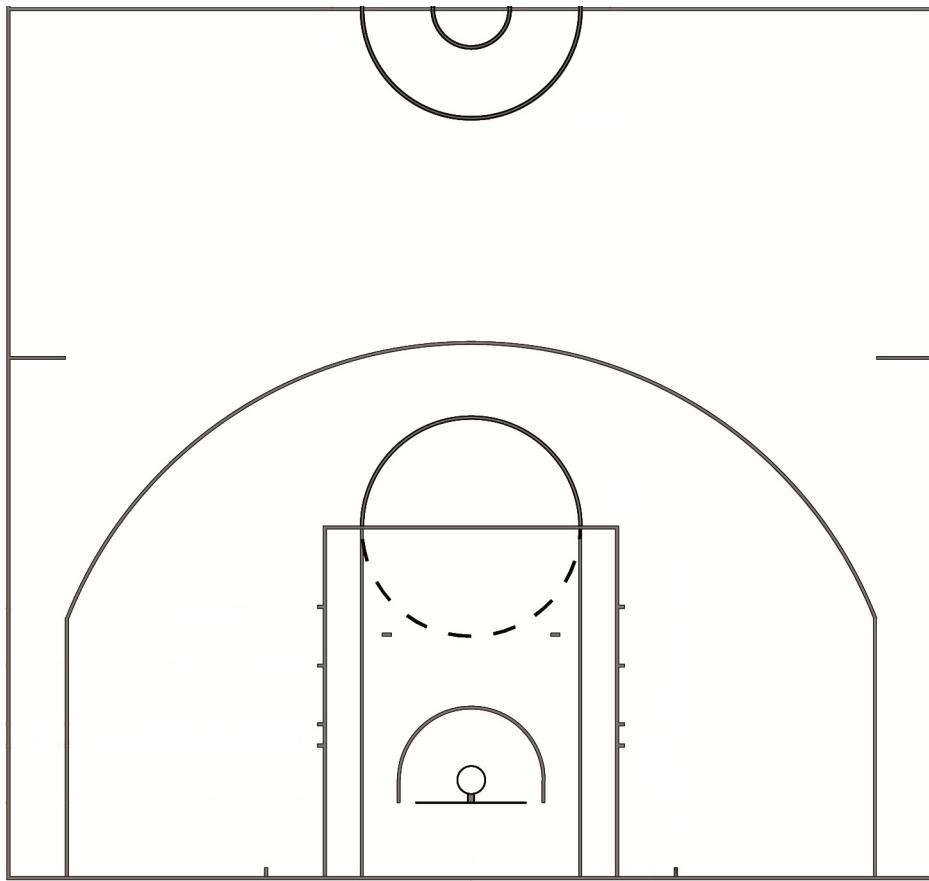
(Note I'm setting the `xlim` and `ylim` to the appropriate ranges (I mostly figured this out through trial and error) and getting rid of the ticks, axes and labels that show up on regular graphs.)

This looks pretty good — it's definitely a recognizable shot chart — but let's try adding a court background - with three point line, paint, etc.

Note: I did *not* know how to do this off the top of my head — it involved a combination of Googling, stackoverflow and tinkering. So I wouldn't worry about memorizing these options or even understanding exactly what they do. Just — if you want to make some shot charts in the future, come back and look at this code.

What we basically will do is put some seaborn scatter plots (which gives us all the hue, column, style etc options) on top of an image of a basketball court.

I found this one:



**Figure 0.26:** NBA Court Background

Which I've included in the code and data files that came with this book.

To add it to our scatter plot, we have to load the image into matplotlib with:

```
In [3]: import matplotlib.image as mpimg  
In [4]: map_img = mpimg.imread('./data/nba_court.jpg')
```

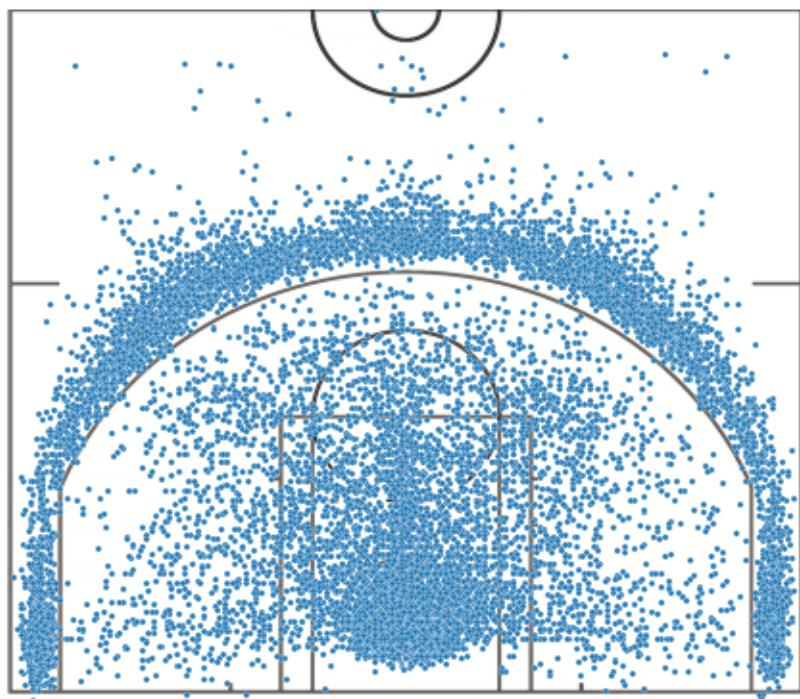
Then, after we make our scatterplot in seaborn:

```
In [5]:  
g = sns.relplot(data=df, x='x', y='y', kind='scatter', s=5)  
g.set(xlim=(-250, 250), ylim=(-50, 400), yticks=[], xticks=[],  
      xlabel=None, ylabel=None)  
g.despine(left=True, bottom=True)
```

We can add it as the background to our plot with:

```
In [6]:  
for ax in g.fig.axes:  
    ax.imshow(map_img, zorder=0, extent=[-250, 250, -30, 400])
```

Shot Chart - All Shots



**Figure 0.27:** Shot Chart - All Data

And there we go!

Again, don't worry about the `mpimg` and `ax.imshow` stuff. Just make a note to come back here when you want to make some shot charts.

To make it easier and extendible, let's put this code in a function:

```
In [7]:  
def shot_chart(df, **kwargs):  
    g = sns.relplot(data=df, x='x', y='y', kind='scatter', **kwargs)  
    g.set(xlim=(-250, 250), ylim=(-30, 400), yticks=[], xticks=[],  
          xlabel=None, ylabel=None)  
    g.despine(left=True, bottom=True)  
  
    for ax in g.fig.axes:  
        ax.imshow(map_img, zorder=0, extent=[-250, 250, -30, 400])  
  
    return g
```

## kwargs

If you try to call a normal Python function with extra arguments, you get an error.

Take this function:

```
def add2(num1, num2):  
    return num1 + num2
```

And calling it with an extra argument:

```
In [1]: add2(num1=4, num2=5, num3=1)  
...  
TypeError: add2() got an unexpected keyword argument 'num3'
```

To get around that, you can add `**kwargs` (short for keyword arguments) when defining your function. It sort of “gobbles up” any extra arguments:

```
def add2_flexible(num1, num2, **kwargs):  
    return num1 + num2
```

Now it works:

```
In [4]: add2_flexible(num1=4, num2=5, num3=1, num4=4)  
Out[4]: 9
```

We've included `**kwargs` in `shot_chart`. Why? Well, `shot_chart` is mostly just a wrapper around seaborn's `relplot` function.

As we've seen above, `sns.relplot` has a lot of options — our main levers `hue` and `col`, but also `col_wrap`, `aspect`, `height` etc. It'd be nice to be able to set these options via our `shot_chart` function.

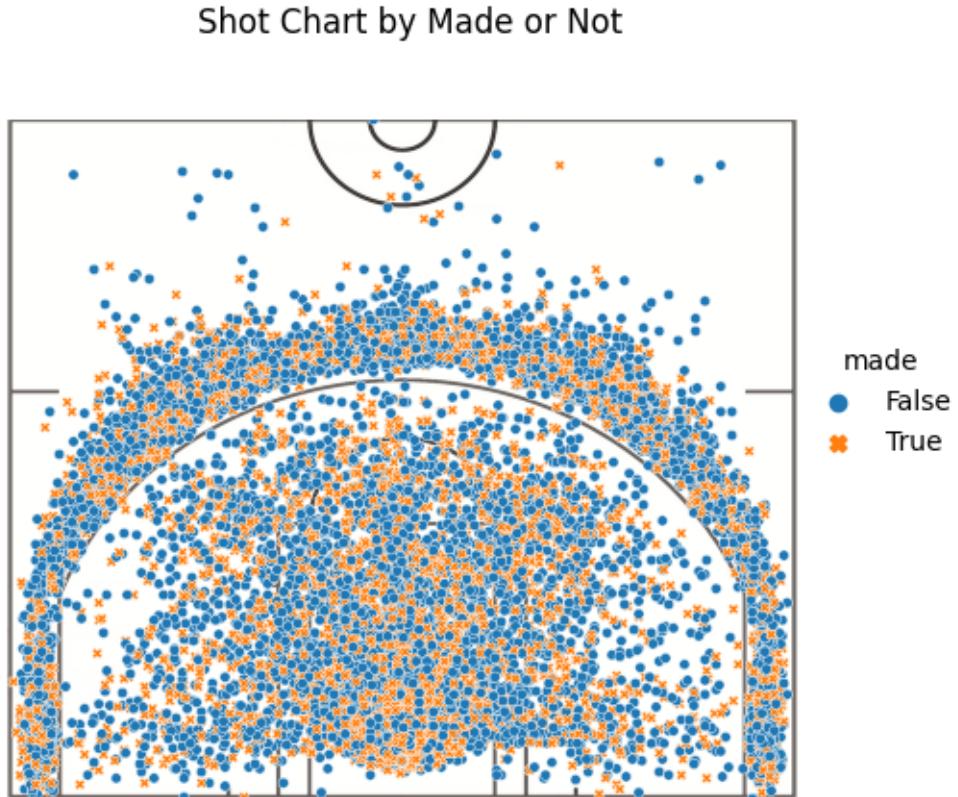
We can do this through `**kwargs`. We're using it to take any extra keyword arguments from `shot_chart`, and pass them to seaborn.

This means can do things like:

```
In [5]: g = shot_chart(dfs, hue='made', style='made', s=15)
```

where `hue='made'`, `style='goal'` and `s=15` (for size of the dots) are the extra keyword arguments.

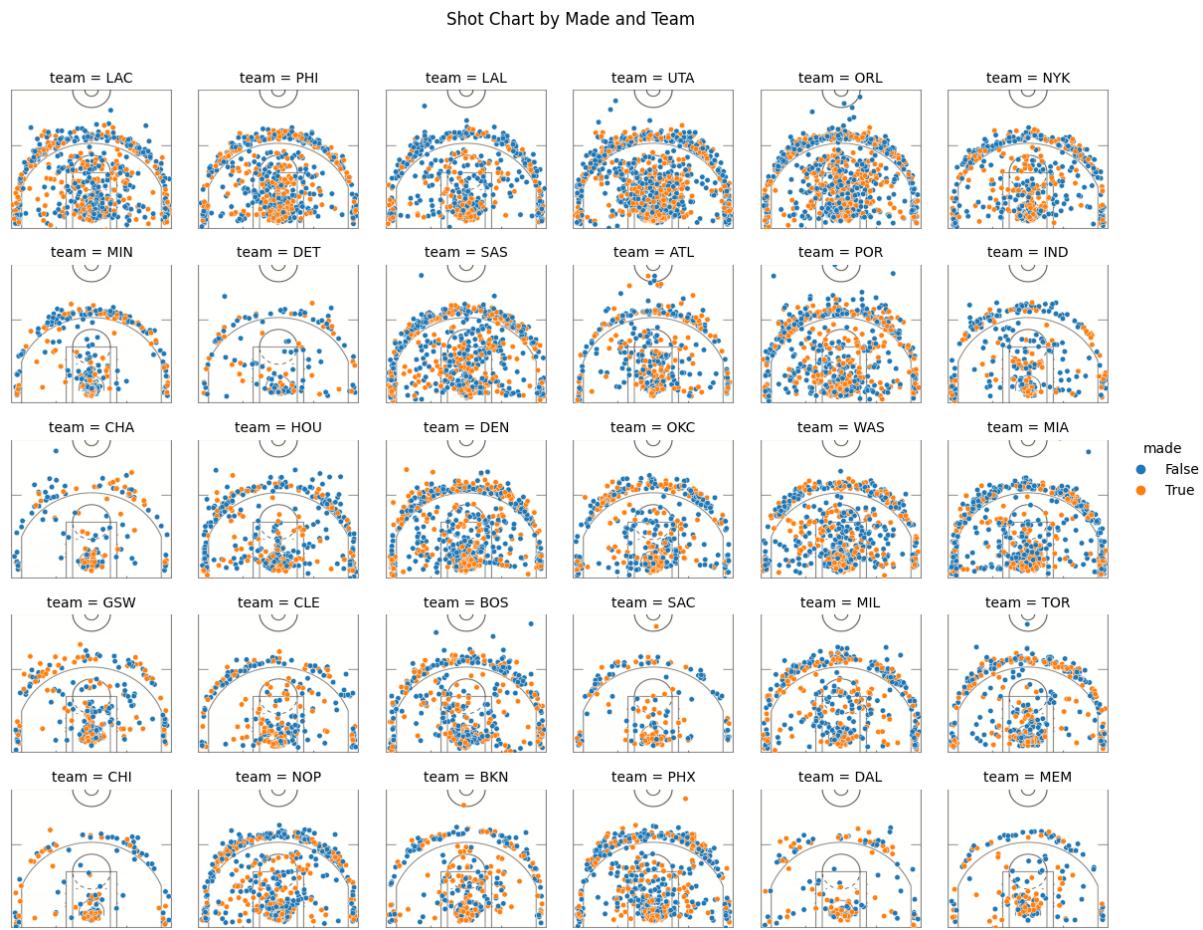
These are stored in `**kwargs`, and get passed straight to `sns.relplot`, giving us:



**Figure 0.28:** Shot Chart - Made/Not

Wee can use all our normal `relplot` arguments, including the plot options we talked about in the last section.

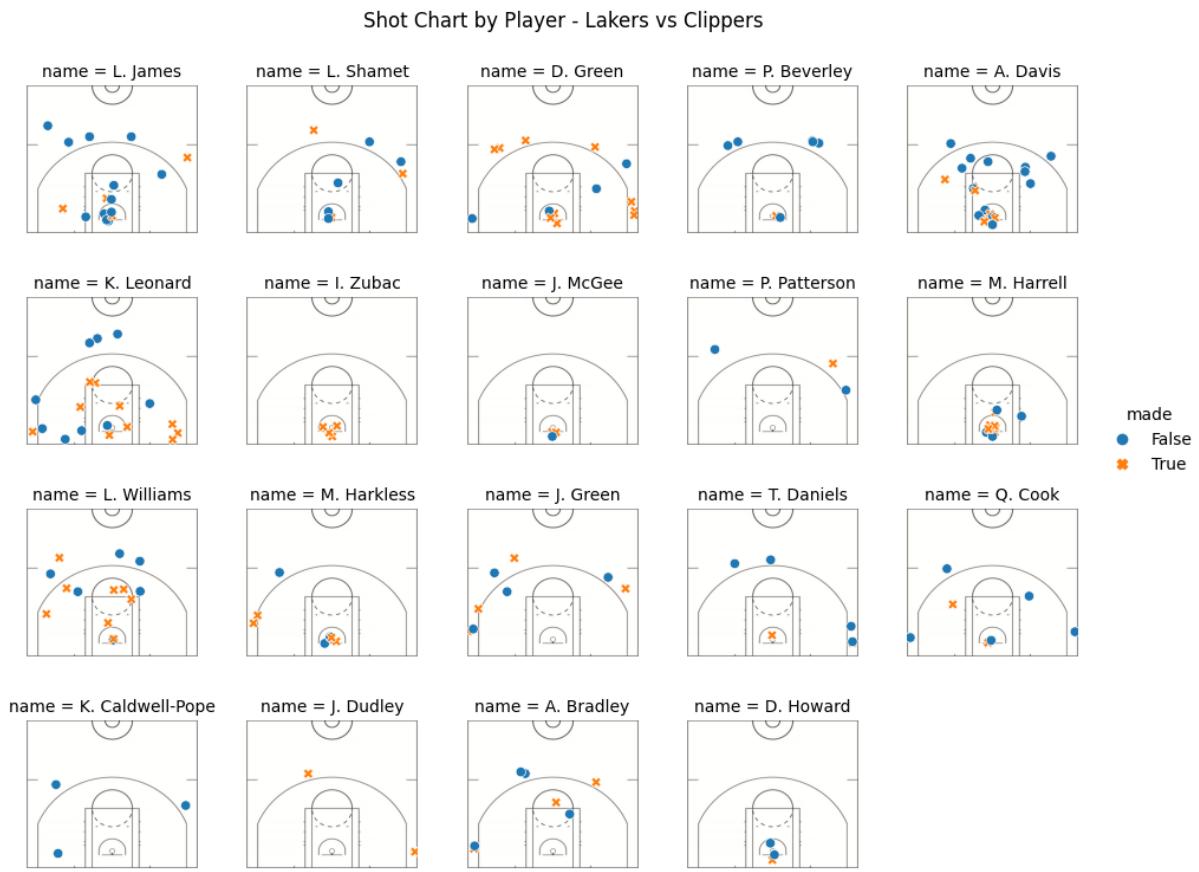
```
In [6]:  
shot_chart(dfs, hue='made', col='team', col_wrap=6, aspect=1.2, height=2)
```



**Figure 0.29:** Shot Chart by Shot Type

We could also look at a shot chart for one single game. Here's Lakers vs Clippers, with each player on his own plot:

```
In [10]:  
g = shot_chart(df.query("game_id == 21900002"), hue='made', style='made',  
               col='name', col_wrap=5, height=2)
```



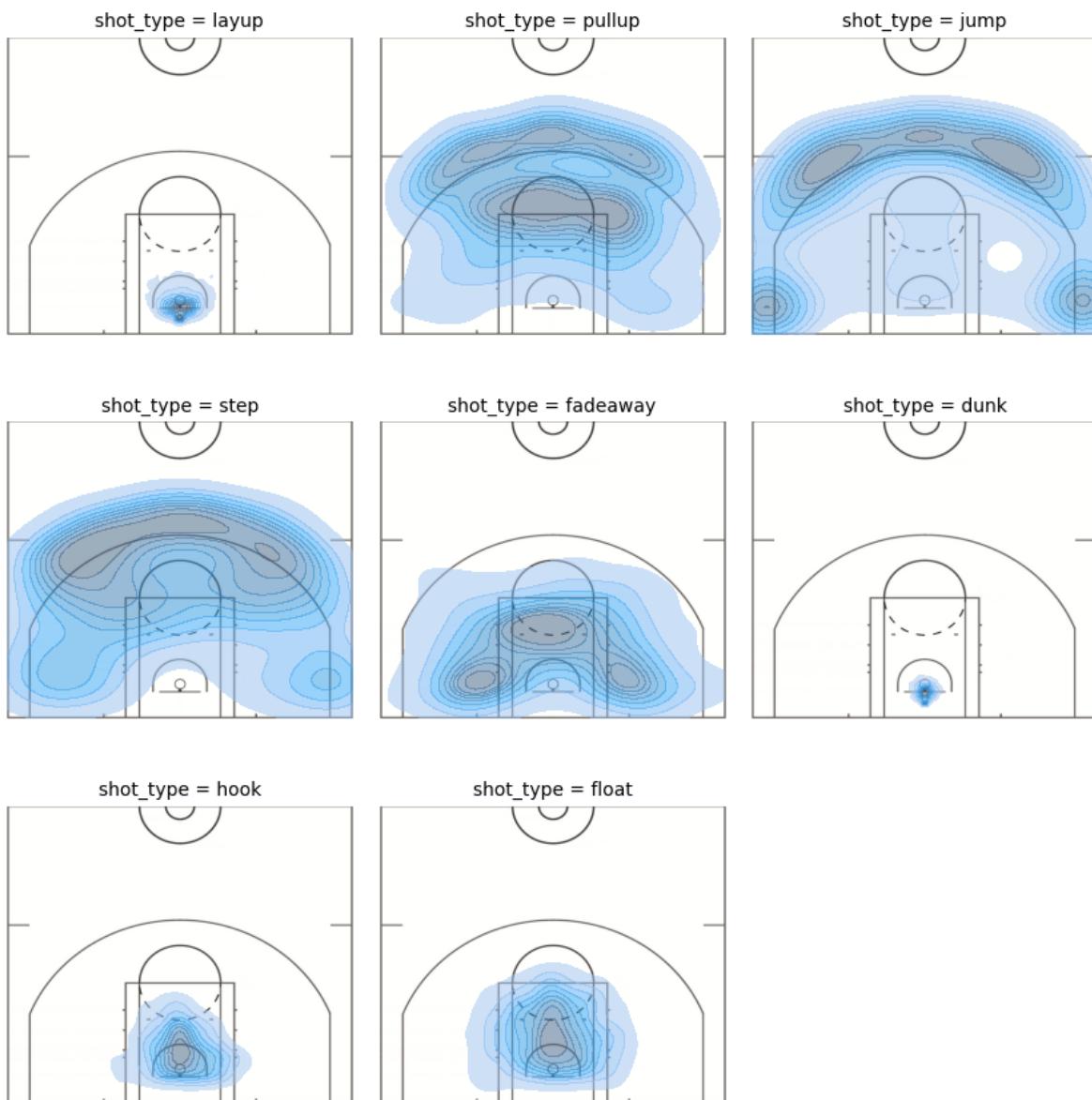
**Figure 0.30:** Shot Chart - LAC vs LAL, by Player

## Contour Plots

One visually interesting tweak is to turn these shot charts with points into contour plots.

Here's a contoured shot plot by shot type:

```
In [10]:
g = (sns.FacetGrid(df, col='shot_type', col_wrap=3)
     .map(sns.kdeplot, 'x', 'y', alpha=0.5, shade=True)
     .add_legend())
g.set(yticks=[], xticks=[], xlabel=None, ylabel=None)
g.despine(left=True, bottom=True)
for ax in g.fig.axes:
    ax.imshow(map_img, zorder=0, extent=[-250, 250, -30, 400])
```



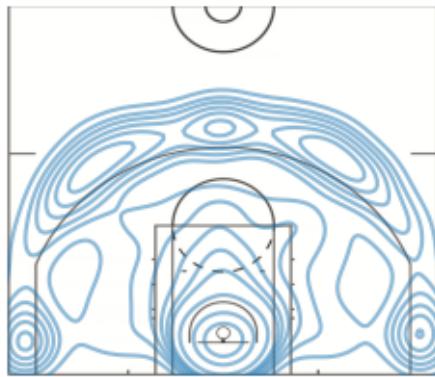
**Figure 0.31:** Shot Chart by Type

Note we're back to the `FacetGrid` then `map` plot.

It's also sometimes interesting to do these plots without shading:

```
In [11]
g = (sns.FacetGrid(df)
      .map(sns.kdeplot, 'x', 'y', alpha=0.5)
      .add_legend())
g.set(yticks=[], xticks=[], xlabel=None, ylabel=None)
g.despine(left=True, bottom=True)
for ax in g.fig.axes:
    ax.imshow(map_img, zorder=0, extent=[-250, 250, -30, 400])
```

Then they end up looking more like lines on a map:



**Figure 0.32:** Lined Shot Chart

In general, I think being able to slice and dice and plot scatter and contour charts like this opens up a lot of possibilities. I always like seeing what people make — feel free to shoot me an email ([nate@nathanbraun.com](mailto:nate@nathanbraun.com)) or tag me on Twitter — (@nathanbraun) with any interesting analysis.

## End of Chapter Exercises

### 6.1

- a) Using the team game data, plot the distribution of three point attempts. Make sure to give your plot a title.

Now modify your plot to show the distribution of three point attempts by whether the team won. Do it (b) as separate colors on the same plot, and (c) as separate plots.

- (d) Sometimes it's effective to use the multiple keywords ("levers") to display redundant information, experiment with this.
- (e) Plot the three point attempts by team, with each team on its own plot. Make sure to limit the number of columns so your plot isn't just one wide row.

### 6.2

- (a) Plot the relationship between three point attempts and free throw percentage. Again, make sure your plot has a title.
- (b) Jitter three point attempts and run the scatter plot again.
- (c) It's hard to tell whether this cloud of points is moving upward to the right or not. Check the correlation between these two variables numerically.

# 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*. In this chapter, we'll look at models more in depth and learn how to build them in Python.

### The Simplest Model

Let's say we want a model that takes in distance to the hoop and predicts whether a basket from there will go in or not. So we might have something like:

```
basket or not = model(feet to basket)
```

### Terminology

First some terminology: the variable "basket or not" is our **output variable**<sup>1</sup>. There's always exactly one output variable.

The variable "feet to basket" is our **input variable**<sup>2</sup>. In this case we just have one, but we could have as many as we want. For example:

```
basket or not = model(feet to basket, time left in game)
```

OK. Back to:

```
basket or not = model(feet to basket)
```

Here's a question: what is the simplest implementation for `model(...)` we might come up with?

How about:

---

<sup>1</sup>Other terms for this variable include: *left hand side variable* (it's to the left of the equals sign); *dependent variable* (its value *depends* on the value of distance to the basket), or *y variable* (traditionally output variables are denoted with *y*, inputs with *x*'s).

<sup>2</sup>Other words for input variables include: *right hand side*, *independent*, *explanatory*, or *x* variables.

```
model(...)= No
```

So give it any distance from the hoop, and our model spits out: “no, it will not be a basket”. Since the majority of shots don’t go in (average field goal percentage in our data is 46%), this model will be more accurate than not! But since it never says anything besides no, it’s not that interesting or useful.

What about:

```
prob_basket = 1 + -0.01*distance_in_ft + -0.0000001*distance_in_ft ^ 2
```

So from 1 foot out we’d get a probability of 0.99, 3 feet 0.97, 10 feet 0.90 and 99 feet 0.000002. This is more interesting. I made the numbers up, so it isn’t a *good* model (for 50 feet it gives about a 0.50 probability of a shot going in, which is way too high). But it shows how a model transforms inputs to an output using some mathematical function.

## Linear regression

This type of model format:

```
output variable =  
    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 (input variable), 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 parts of the equation. The particular letters don’t matter (though people have settled on conventions). The point is to provide an abstract way of thinking about and referring to parts of our model.

A linear equation can have more than one data term in it, which is why statisticians use *b<sub>0</sub>* and *b<sub>1</sub>* 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: *b<sub>0</sub>*, *b<sub>1</sub>*, and *b<sub>2</sub>* are different numbers, and *x<sub>1</sub>* and *x<sub>2</sub>* are different columns of data. The notation just ensures you can include as many variables as you need to (just add another number).

In our probability-of-basket model that I made up, `x1` was feet from the basket, and `x2` was feet from the basket squared. We had:

```
prob_basket = b0 + b1*(distance_in_feet) + b2*(distance_in_feet ^ 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 and one output variable. So we need a three column DataFrame with distance, distance squared, and made basket or not. We need it at the shot level. Let's do it.

*Note: the code for this section is in the file `07_01_ols.py`. We'll pick up from the top of the file.*

```
1 import pandas as pd
2 import statsmodels.formula.api as smf
3 from os import path
4
5 DATA_DIR = './data'
6
7 # load
8 df = pd.read_csv(path.join(DATA_DIR, 'shots.csv'))
9
10 df['dist_sq'] = df['dist']**2
11 df['made'] = df['made'].astype(int)
```

Besides loading our libraries and the data, this first section of the code also does some minor processing:

First, we had to make `dist_sq` by squaring our `dist` variable. Note exponents in Pandas use the `**` operator. I don't necessarily expect you to have known that off the top of your head. However, I do expect you to figure it out via Google after trying `^` and getting an error message.

This initial `made` variable is a column of booleans (`True` if the shot went in, `False` otherwise). That's fine except our model can only operate on actual numbers. We need to convert this boolean column into its numeric equivalent.

The way to do this while making everything easy to interpret is by transforming `made` 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)<sup>3</sup>.

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. All for this:

<sup>3</sup>Notice even though we have two outcomes — made basket or not — we just have the one column. There'd be no benefit to including an extra column `missed` because it'd be the complete opposite of `made`; it doesn't add any information. In fact, including two perfectly correlated variables like this in your input variables breaks the model, and most statistical programs will drop the unnecessary variable automatically.

```
In [1]: df[['made', 'dist', 'dist_sq']].head()
Out[1]:
   made  dist  dist_sq
0      1     2       4
1      0    26     676
2      1    25     625
3      0    26     676
4      0    18     324
```

Once we have our table, we just need to pass it to our modeling function, which we get from the third party library `statsmodels`. There are different ways to use it, but I've found the easiest is via the formula API.

We imported this at the top:

```
import statsmodels.formula.api as smf
```

We'll be using the `ols` function. OLS stands for Ordinary Least Squares, and is another term for basic, standard linear regression.

Compared to getting the data in the right format, actually running the model in Python is trivial. We just 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 [2]: model = smf.ols(formula='made ~ dist + dist_sq', data=df)
In [3]: results = model.fit()
```

Once we've done that, we can look at the results:

```
In [4]: results.summary2()
Out[4]:
"""
Results: Ordinary least squares
=====
Model: OLS Adj. R-squared: 0.050
Dependent Variable: made AIC: 23533.8158
Date: 2022-06-08 09:59 BIC: 23557.0168
No. Observations: 16876 Log-Likelihood: -11764.
Df Model: 2 F-statistic: 441.0
Df Residuals: 16873 Prob (F-statistic): 2.15e-187
R-squared: 0.050 Scale: 0.23609
-----
          Coef.  Std.Err.      t    P>|t|      [0.025  0.975]
-----
Intercept   0.6197  0.0067  91.8279  0.0000  0.6065  0.6330
dist        -0.0177  0.0010 -17.7935  0.0000 -0.0197 -0.0158
dist_sq      0.0003  0.0000   8.2325  0.0000  0.0002  0.0003
-----
Omnibus:     68294.390 Durbin-Watson: 2.012
Prob(Omnibus): 0.000 Jarque-Bera (JB): 2269.755
Skew:         0.163 Prob(JB): 0.000
Kurtosis:    1.233 Condition No.: 797
=====
"""


```

We get back a lot of information from 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 making a basket when you're right next to — 0 feet away — the basket. The other coefficients are next to `dist` and `dist_sq`.

So instead of my made up formula from earlier, the formula that best fits this data is:

```
0.6197 + -0.017*dist + 0.0003*(dist ^ 2).
```

Let's test it out with some values:

```
In [5]:  
def prob_of_make(yds):  
    b0, b1, b2 = results.params  
    return (b0 + b1*yds + b2*(yds**2))  
  
In [6]: prob_of_make(1)  
Out[6]: 0.6022954444724247  
  
In [7]: prob_of_make(25)  
Out[7]: 0.3431953624855063  
  
In [8]: prob_of_make(30)  
Out[8]: 0.3277962585781258
```

Seems reasonable. Let's use the `results.predict` method to predict it for every value of our data.

```
In [9]: df['made_hat'] = results.predict(df)  
  
In [10]: df[['made', 'made_hat']].head()  
Out[10]:  
   made  made_hat  
0      1  0.585380  
1      0  0.339051  
2      1  0.343195  
3      0  0.339051  
4      0  0.387104
```

Seems reasonable.

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 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` to make the residuals as small as possible<sup>4</sup>.

The proportion of variation in the output variable that your model "explains" (the rest of variation is in the residuals) is called R^2 ("R squared", often written R2). It's always between 0-1. An R2 of 0 means your model explains nothing. An R2 of 1 means your model is perfect: your `yhat` always equals `y`; every residual is 0.

---

<sup>4</sup>Technically, OLS regression finds the coefficients that make the total sum of each *squared* residual as small as possible. Squaring a residual makes sure it's positive. Otherwise the model could just "minimize" the sum of residuals by underpredicting everything to get a bunch of negative numbers. Note "squares as small as possible" is partly the name, ordinary "least squares" regression.

## Statistical Significance

If we look at our regression results, we can see there are a bunch of columns in addition to the coefficients. All of these are getting at the same thing, the **significance** of each coefficient.

Statistical significance is bit tricky to explain, but it basically gets at: is the effect we're observing 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 — which we usually can't observe, and the observed/measured relationship, which we *can*.

For example, consider flipping a fair coin.

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) that 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 guess, then flip a few coins — we probably will observe *some* relationship in our data.

Measures of statistical significance are meant to help you tell whether any 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. Assuming that's the case, (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.

To 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.

*The code for this section is in the file `07_02_ols.py`. We'll pick up after the imports.*

```
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.summary()
"""
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
=====
"""

```

Since we're working with randomly generated data you'll get something different, but according to my 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 the significance columns. The one to pay attention to is  $P > |t|$ . It says: (1) start by assuming no true relationship between your guess and probability of calling the flip correctly. Then (2), if that were the case, you'd see a relationship as "strong" as the one we observed about 21% of

the time.

So looks like we had a semi-unusual draw, about 80th percentile, but nothing *that* crazy.

(Note: 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 see how often you get a 9 or 10.)

Traditionally, the rule has been for statisticians and social scientists to call a result **significant** if the P value is less than 0.05, 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.

Usually, people running regressions *want* to see an interesting, significant result. This is a problem because there are many, many people running regressions. If we have 100 researchers running a regression on relationships that don't actually exist, 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 they're describing statistical noise.

The real situation is worse, because usually even one person can run enough variations of a regression — adding in variables here, making different data assumptions there — to get an interesting and significant result.

But if you keep running regressions until you find something you like, the traditional interpretation of a 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 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 distance from the basket and probability of the shot going in. Distance clearly has an effect. Looking at `P > |t|` it says:

1. Start by assuming no true relationship between distance to the basket and probability of making a shot.
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 the basket actually is related to the 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 shot data comes with a variable called `dunk`, which is 1 if the shot was an attempted dunk, 0 otherwise (layup, jumper, etc).

Let's see how dunking a shot affects the probability of the shot going in.

*The code for this example is in `07_03_ols2.py`. We'll pick up on line 26, after you've loaded the shot data into a DataFrame named `df` and done some light processing on shot types.*

```
In [1]:  
model = smf.ols(formula=  
    """  
        made ~ dunk  
    """", data=df)  
results = model.fit()  
results.summary2()  
  
Out[1]:  
"""  
Results: Ordinary least squares  
=====  
Model: OLS Adj. R-squared: 0.048  
Dependent Variable: made AIC: 23552.5229  
Date: 2022-06-07 10:04 BIC: 23567.9901  
No. Observations: 16876 Log-Likelihood: -11774.  
Df Model: 1 F-statistic: 860.2  
Df Residuals: 16874 Prob (F-statistic): 1.83e-184  
R-squared: 0.049 Scale: 0.23637  
-----  
Coef. Std.Err. t P>|t| [0.025 0.975]  
-----  
Intercept 0.4333 0.0039 112.5409 0.0000 0.4258 0.4408  
dunk[T.True] 0.4808 0.0164 29.3289 0.0000 0.4486 0.5129  
-----  
Omnibus: 63998.527 Durbin-Watson: 2.038  
Prob(Omnibus): 0.000 Jarque-Bera (JB): 2557.914  
Skew: 0.238 Prob(JB): 0.000  
Kurtosis: 1.153 Condition No.: 4  
=====  
"""
```

Let's look at the coefficients to practice reading them. According to this, a dunk attempt increases the probability a shot goes in by 0.48. This is statistically significant.

But if you think about it, there are two reasons a dunk attempt is more likely to go in: 1. The player is physically placing the ball in the basket (vs floating it, trying to bounce it off the backboard, or just chucking it up). Not every dunk goes in (we've all seen dunks go flying off the rim the other way), but it definitely helps. 2. But also, to even attempt a dunk you need to be a lot closer to the basket.

This last one is obvious, but we can also see it in the data:

```
In [2]: df.groupby('dunk')['dist'].mean()
Out[2]:
dunk
False    14.168705
True     0.828142
Name: dist, dtype: float64
```

On average, dunk attempts are inside 1 foot from the basket, vs non-dunks, which average 14 feet. But what if we want to quantify *just* the dunking part (not distance) on shot probability.

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.

We know dunks are a lot closer to the basket, but we're not explicitly controlling for that.

To do so, we can add distance to the model. Let's run it:

```
In [3]:  
model = smf.ols(formula=  
    """  
        made ~ dunk + dist  
    """", data=df)  
results = model.fit()  
results.summary2()  
  
Out[3]:  
"""  
    Results: Ordinary least squares  
=====  
Model: OLS Adj. R-squared: 0.073  
Dependent Variable: made AIC: 23108.1478  
Date: 2022-06-07 10:10 BIC: 23131.3487  
No. Observations: 16876 Log-Likelihood: -11551.  
Df Model: 2 F-statistic: 667.7  
Df Residuals: 16873 Prob (F-statistic): 8.26e-280  
R-squared: 0.073 Scale: 0.23021  
-----  
      Coef.  Std.Err.    t   P>|t|  [0.025  0.975]  
-----  
Intercept  0.5427  0.0064  84.8671  0.0000  0.5301  0.5552  
dunk[T.True]  0.3778  0.0169  22.3721  0.0000  0.3447  0.4109  
dist       -0.0077  0.0004 -21.2662  0.0000 -0.0084 -0.0070  
-----  
Omnibus: 70096.276 Durbin-Watson: 2.017  
Prob(Omnibus): 0.000 Jarque-Bera (JB): 2276.793  
Skew: 0.228 Prob(JB): 0.000  
Kurtosis: 1.259 Condition No.: 79  
=====  
"""
```

Now that distance from the hoop is explicitly accounted for, we know that the `dunk` coefficient measures *only* the effect of physically placing the ball in the basket, *not* the fact dunks (unless you're MJ in the 1987 dunk contest) are closer to the basket. And we can see the effect drops, from 0.58 to 0.38.

Let's try adding `layup`:

```
In [4]:
model = smf.ols(formula=
"""
    made ~ dunk + dist + layup
    """, data=df)
results = model.fit()
results.summary2()

Out[4]:
"""
Results: Ordinary least squares
=====
Model: OLS                    Adj. R-squared: 0.075
Dependent Variable: made      AIC: 23083.4563
Date: 2022-06-07 10:13        BIC: 23114.3909
No. Observations: 16876       Log-Likelihood: -11538.
Df Model: 3                  F-statistic: 454.7
Df Residuals: 16872          Prob (F-statistic): 3.91e-284
R-squared: 0.075              Scale: 0.22986
-----
          Coef.   Std.Err.      t    P>|t|    [0.025  0.975]
-----
Intercept      0.4913    0.0118  41.5957  0.0000    0.4682  0.5145
dunk[T.True]   0.4273    0.0194  22.0223  0.0000    0.3893  0.4653
layup[T.True]  0.0669    0.0129  5.1678   0.0000    0.0415  0.0923
dist           -0.0055   0.0006  -9.7324  0.0000   -0.0066 -0.0044
-----
Omnibus:      70806.630   Durbin-Watson: 2.012
Prob(Omnibus): 0.000      Jarque-Bera (JB): 2254.673
Skew:          0.230      Prob(JB): 0.000
Kurtosis:     1.270      Condition No.: 106
=====
"""

```

This is interesting.

Let's look at all the coefficients to practice reading them. The *Intercept* (sometimes denoted *b0*) is 0.49. This says the probability of making a shot that's 0 feet away from the basket, that's *not* a layup or a dunk, is 0.49.

The coefficient on *dist* says every foot away from the basket lowers our the probability the shot goes in by 0.0055. So if we're 18 and a half feet away from the hoop, our probability of making the shot is:

```
In [5]: 0.4913 -0.0055*18.5
Out[5]: 0.38955
```

The coefficients on *dunk* and *layup* tell us dunking and laying up the ball increase our probability of making the shot by 0.4273 and 0.0669 respectively. Note we *add* this to the intercept. So our proba-

bility of making a dunk from 1 foot away:

```
In [6]: 0.4913 -0.0055*1 + 0.4273  
Out[6]: 0.9131
```

And a layup from 2 feet out:

```
In [7]: 0.4913 -0.0055*2 + 0.0669  
Out[7]: 0.5472
```

## Fixed Effects

We've seen how dummy variables work for binary, true or false data, but what about something with more than two categories?

Not just `made` or `dunk` or not, but all shot types (`layup`, `fadeaway`, `jumper` etc) or position (`SG`, `PG`, `SF`, `PF`, `C`).

These are called categorical variables or “fixed effects” and the way we handle them is by putting our one categorical variable (position) into a series of dummies that give us the same information (`is_sg`, `is_pg`, `is_pf`, `is_sf`, `is_c`).

So `is_sg` is 1 if player is a SG, 0 otherwise, etc. Except we don't need *all* of these. If there are only 5 positions, and a player has to be one, then we know if a player isn't a SG, PG, SF, or PF then we know he must be a C. That means we can (by *can* I mean have to so that 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 [1]: pd.get_dummies(df['shot_type']).head()  
Out[1]:  
   basic  dunk  fadeaway  float  hook  layup  pullup  step  
0      0      0        0      0      0      1      0      0  
1      0      0        0      0      0      0      1      0  
2      1      0        0      0      0      0      0      0  
3      1      0        0      0      0      0      0      0  
4      1      0        0      0      0      0      0      0
```

Again, *all* of these variables would be redundant — there are only eight shot types, so you can pass the `drop_first=True` argument to `get_dummies` to have it return only seven columns.

While it's useful to know what's going on behind the scenes, in practice, programs like `statsmodels` can automatically convert categorical data to a set of fixed effects by wrapping the variable in `C(...)` like this:

```
In [2]:  
model = smf.ols(formula="made ~ C(shot_type) + dist + dist_sq", data=df)  
results = model.fit()  
results.summary2()  
  
Out[2]:  
"""  
Results: Ordinary least squares  
=====  
Model: OLS Adj. R-squared: 0.076  
Dependent Variable: made AIC: 23062.2850  
Date: 2022-06-08 11:45 BIC: 23139.6214  
No. Observations: 16876 Log-Likelihood: -11521.  
Df Model: 9 F-statistic: 155.5  
Df Residuals: 16866 Prob (F-statistic): 6.53e-284  
R-squared: 0.077 Scale: 0.22949  
-----  
Coef. Std.Err. t P>|t| [0.025 0.975]  
-----  
Intercept 0.5950 0.0239 24.9138 0.0000 0.5482 0.6418  
C(shot_type)[T.dunk] 0.3300 0.0276 11.9743 0.0000 0.2760 0.3841  
C(shot_type)[T.fadeaway] -0.0649 0.0228 -2.8487 0.0044 -0.1095 -0.0202  
C(shot_type)[T.float] -0.0821 0.0195 -4.2054 0.0000 -0.1204 -0.0438  
C(shot_type)[T.hook] -0.0707 0.0272 -2.6011 0.0093 -0.1240 -0.0174  
C(shot_type)[T.layup] -0.0235 0.0225 -1.0446 0.2962 -0.0675 0.0206  
C(shot_type)[T.pullup] -0.0022 0.0126 -0.1761 0.8602 -0.0269 0.0225  
C(shot_type)[T.step] 0.0249 0.0192 1.2986 0.1941 -0.0127 0.0624  
dist -0.0135 0.0018 -7.6683 0.0000 -0.0170 -0.0101  
dist_sq 0.0001 0.0000 3.8220 0.0001 0.0001 0.0002  
-----  
Omnibus: 71509.496 Durbin-Watson: 2.010  
Prob(Omnibus): 0.000 Jarque-Bera (JB): 2233.875  
Skew: 0.232 Prob(JB): 0.000  
Kurtosis: 1.279 Condition No.: 5706  
=====  
* The condition number is large (6e+03). This might indicate  
strong multicollinearity or other numerical problems.  
"""
```

Let's zoom in on our fixed effect coefficients:

	Coef.	Std. Err.	t	P> t	[0.025	0.975]
Intercept	0.5950	0.0239	24.9138	0.0000	0.5482	0.6418
C(shot_type)[T.dunk]	0.3300	0.0276	11.9743	0.0000	0.2760	0.3841
C(shot_type)[T.fadeaway]	-0.0649	0.0228	-2.8487	0.0044	-0.1095	-0.0202
C(shot_type)[T.float]	-0.0821	0.0195	-4.2054	0.0000	-0.1204	-0.0438
C(shot_type)[T.hook]	-0.0707	0.0272	-2.6011	0.0093	-0.1240	-0.0174
C(shot_type)[T.layup]	-0.0235	0.0225	-1.0446	0.2962	-0.0675	0.0206
C(shot_type)[T.pullup]	-0.0022	0.0126	-0.1761	0.8602	-0.0269	0.0225
C(shot_type)[T.step]	0.0249	0.0192	1.2986	0.1941	-0.0127	0.0624

Again, including *all* shot types would be redundant, so `statsmodel` automatically dropped one, in this case '`basic`'.

If we wanted we could have `statsmodel` drop a different shot type, say '`layup`':

```
In [3]:  
model = smf.ols(  
    formula="made ~ C(shot_type, Treatment(reference='layup')) + dist +  
    dist_sq", data=df)  
results = model.fit()  
results.summary2()  
  
Out[3]:  
"""  
Results: Ordinary least squares  
=====  
Model: OLS Adj. R-squared: 0.076  
Dependent Variable: made AIC: 23062.2850  
Date: 2022-06-08 11:50 BIC: 23139.6214  
No. Observations: 16876 Log-Likelihood: -11521.  
Df Model: 9 F-statistic: 155.5  
Df Residuals: 16866 Prob (F-statistic): 6.53e-284  
R-squared: 0.077 Scale: 0.22949  
-----  
Coef. Std.Err. t P>|t| [0.025 0.975]  
-----  
Intercept 0.5715 0.0076 75.4999 0.0000 0.5567 0.5864  
C(shot_type) [T.basic] 0.0235 0.0225 1.0446 0.2962 -0.0206 0.0675  
C(shot_type) [T.dunk] 0.3535 0.0173 20.4922 0.0000 0.3197 0.3873  
C(shot_type) [T.fadeaway] -0.0414 0.0249 -1.6670 0.0955 -0.0901 0.0073  
C(shot_type) [T.float] -0.0587 0.0178 -3.2967 0.0010 -0.0935 -0.0238  
C(shot_type) [T.hook] -0.0473 0.0232 -2.0384 0.0415 -0.0927 -0.0018  
C(shot_type) [T.pullup] 0.0212 0.0223 0.9529 0.3406 -0.0224 0.0649  
C(shot_type) [T.step] 0.0483 0.0274 1.7619 0.0781 -0.0054 0.1021  
dist -0.0135 0.0018 -7.6683 0.0000 -0.0170 -0.0101  
dist_sq 0.0001 0.0000 3.8220 0.0001 0.0001 0.0002  
-----  
Omnibus: 71509.496 Durbin-Watson: 2.010  
Prob(Omnibus): 0.000 Jarque-Bera (JB): 2233.875  
Skew: 0.232 Prob(JB): 0.000  
Kurtosis: 1.279 Condition No.: 5145  
=====  
"""
```

Now we need to interpret all of our shot type coefficients relative to layup. How does a floater change the probability of making a shot? It lowers it 0.0587 compared to attempting a layup. What about a dunk? It increases the probability of a made shot by 0.3535 vs a layup.

Let's calculate a few probabilities. Let's save the intercept and distance coefficients so we can reuse them:

```
In [4]: b0 = 0.5715  
In [5]: b_dist = -0.0135  
In [6]: b_dist2 = 0.0001
```

Now what's the probability of making a 25 foot step back?

```
In [7]: b0 + b_dist*25 + b_dist2*(25^2) + 0.0483  
Out[7]: 0.285
```

What about an 15 foot pullup?

```
In [8]: b0 + b_dist*15 + b_dist2*(15^2) + 0.0212  
Out[8]: 0.3915
```

Here's a trickier one — what about a 3 foot layup? Note there's no layup coefficient — `statsmodels` dropped that one. So how do we calculate this?

In this case we know the shot was a layup when all the other shot variables above are 0. Layup is our default, baseline case. So it's just:

```
In [9]: b0 + b_dist*3 + b_dist2*(3^2)  
Out[9]: 0.5311
```

No shot coefficient necessary.

## Squaring Variables

When we run a linear regression, we're assuming certain things about how the world works, namely that a change in one of our `x` variables always means the *same* change in `y`.

Take our distance-make probability model:

```
prob_scoring = b0 + b1*dist
```

By modeling this as a linear relationship, we're assuming a one unit change in distance always has the *same* effect on probability the basket goes in. This effect (`b1`) is the same whether we're going from 1 to 2 feet from the basket, or 24 to 25 feet.

Is this a good assumption? In this case probably not. I'd expect your odds of making a shot go down a lot more from 1 to 2 feet than 24 to 25 feet.

Does this mean we have to abandon linear regression?

No, because there are tweaks we can make that — while keeping things linear — help make our model more realistic. All of these tweaks basically involve keeping the linear framework ( $y = b_0 + x_1 \cdot b_1 + x_2 \cdot b_2 \dots x_n \cdot b_n$ ), while transforming the  $x$ 's to model different situations.

In this case — where we think the relationship between make probability and distance might vary by distance — we can square distance and include it in the model.

```
prob_scoring = b0 + b1*dist + b2*dist^2
```

This allows the effect to change depending on where we are in distance. For early values,  $distance^2$  is relatively small, and  $b_2$  doesn't come into play as much — later it does<sup>5</sup>.

In the model without squared terms,  $make\_probability = b_0 + b_1 \cdot distance$ , this derivative is just  $b_1$ , which should match your intuition. One increase in distance leads to a  $b_1$  decrease in make probability.

With squared terms like  $make\_probability = b_0 + b_1 \cdot distance + b_2 \cdot distance^2$  the precise value of the derivative is less intuitive, but pretty easy to figure out with the [power rule](#). It says a one unit increase in distance will lead to a  $b_1 + 2 \cdot b_2 \cdot (distance \text{ we're currently at})$ .

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 the output, inputs, or both. This lets you move from absolute to relative differences and interpret coefficients as percent changes.

So if we have our distance regression:

```
prob_scoring = b0 + b1*dist
```

We'd interpret  $b_1$  as the decrease in probability of making a shot associated with moving one more foot away from the hoop.

If we did:

```
prob_scoring = b0 + b1*ln(dist)
```

We'd interpret  $b_1$  as the decrease in make probability given a one *percent change* in shot distance.

Let's run this. First we need to calculate the natural log of our variable. Note mathematically you can't take the natural log of 0, so let's give every shot a minimum of 6 inches (0.5 feet).

---

<sup>5</sup>Calculus is a branch of math that's all about analyzing how a change in one variable affects another. In calculus parlance, saying "how probability of making a basket changes as distance changes" is the same as saying, "the derivative of make probability with respect to distance".

```
In [1]: df['ln_dist'] = np.log(df['dist'].apply(lambda x: max(x, 0.5)))
```

Running the regression:

```
In [2]:
model = smf.ols(formula='made ~ ln_dist', data=df)
results = model.fit()
results.summary2()

Out[2]:
"""
Results: Ordinary least squares
=====
Model: OLS           Adj. R-squared: 0.064
Dependent Variable: made          AIC: 23268.6800
Date: 2022-06-08 13:19  BIC: 23284.1473
No. Observations: 16876          Log-Likelihood: -11632.
Df Model: 1             F-statistic: 1161.
Df Residuals: 16874         Prob (F-statistic): 3.74e-246
R-squared: 0.064          Scale: 0.23242
-----
      Coef.  Std.Err.      t    P>|t|  [0.025  0.975]
-----
Intercept   0.6411  0.0065  98.8433  0.0000  0.6283  0.6538
ln_dist     -0.0919  0.0027 -34.0732  0.0000 -0.0972 -0.0866
-----
Omnibus: 72657.622  Durbin-Watson: 1.995
Prob(Omnibus): 0.000  Jarque-Bera (JB): 2133.221
Skew: 0.178  Prob(JB): 0.000
Kurtosis: 1.295  Condition No.: 5
=====
```

We can see getting 1% further away from the basket lowers our probability of making the shot by about 0.09.

Note you can log an input variable (like distance above) or an output variable. If we were working with season-team totals and had some regression like:

$\ln(\text{wins}) = b_0 + b_1 \ln(\text{pts scored}) + b_2 \ln(\text{pts allowed})$

Then  $b_1$  would be the *percent change* in wins given a *one percent change* in total points scored.

## 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.

For example, earlier we ran a regression on make probability as a function of shot type (dunk, layup, etc) and distance from the basket.

But by including these variables separately (our shot type variables, then our distance variables) we're assuming that distance effects make probability the same for every shot. Is that true?

Probably not. For example, I'd imagine being an extra foot away from the basket on a layup has a bigger impact than being an extra foot away on a regular jumper.

To see if this is true, 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:

```
make_prob = b0 + b1*dist + ...
```

To this:

```
make_prob = b0 + b1*dist + b2(dist*is_layup)+ ...
```

Then **b1** is the impact of being an extra foot away, and **b1 + b2** is the effect of being one foot away on layups specifically.

Let's run this regression:

```
In [3]: df['is_layup'] = df['shot_type'] == 'layup'
```

```
In [4]:  
model = smf.ols(formula=  
    """  
        made ~ dist + dist:is_layup  
    """", data=df)  
results = model.fit()  
results.summary2()
```

```
Out[5]:
```

```
"""  
Results: Ordinary least squares  
=====
```

Model:	OLS	Adj. R-squared:	0.059
Dependent Variable:	made	AIC:	23371.5427
Date:	2022-06-08 13:31	BIC:	23394.7437
No. Observations:	16876	Log-Likelihood:	-11683.
Df Model:	2	F-statistic:	526.7
Df Residuals:	16873	Prob (F-statistic):	1.27e-222
R-squared:	0.059	Scale:	0.23383

```
-----  
            Coef.  Std.Err.      t    P>|t|  [0.025  0.975]  
-----  
Intercept      0.6503  0.0070  92.5756  0.0000  0.6366  0.6641  
dist          -0.0123  0.0004 -32.3300  0.0000 -0.0130 -0.0116  
dist:is_layup[T.True] -0.0512  0.0034 -15.2137  0.0000 -0.0578 -0.0446  
-----  
Omnibus:      69787.594     Durbin-Watson:   2.013  
Prob(Omnibus): 0.000       Jarque-Bera (JB): 2208.947  
Skew:          0.158       Prob(JB):       0.000  
Kurtosis:      1.256       Condition No.: 34  
=====
```

```
"""
```

So we can see via the coefficient on `dist:is_layup[T.True]` that every extra foot away from the basket lowers the shot probability an additional -0.0512 for layups specifically.

## Logistic Regression

So far we've been using a normal linear regression, also called Ordinary Least Squares (OLS). This works well for modeling continuous output variables like points scored or number of rebounds.

When applied to 0-1, true or false type output variables (shot made or not or not) — we interpret OLS coefficients as changes in *probabilities*. E.g., in our last regression, being an extra foot away from the

basket decreased the probability we made a shot by 0.0123 (0.0635 for layups).

That's fine, but in practice modeling probabilities with OLS often leads to predictions that are outside the 0-1 range. We can avoid this by running a **logistic** regression instead of OLS.

You can use all the same tricks (interactions, fixed effects, squared, dummy and logged variables) on the right hand side, we're just working with a different model.

Logit:

```
1/(1 + exp(-(b0 + b1*x1 + ... + bn*xn)))
```

Vs Ordinary Least Squares:

```
b0 + b1*x1 + ... + bn*xn
```

In `statsmodels` it's just a one line change.

```
In [1]:  
model = smf.logit(formula=  
    """  
        made ~ layup + dist + dist:layup  
    """", data=df)  
logit_results = model.fit()  
logit_results.summary2()  
  
--  
Optimization terminated successfully.  
    Current function value: 0.659379  
    Iterations 5  
Out[1]:  
"""  
    Results: Logit  
=====  
Model:          Logit           Pseudo R-squared: 0.044  
Dependent Variable: made          AIC:            22263.3619  
Date:          2022-06-08 13:42 BIC:            22294.2965  
No. Observations: 16876          Log-Likelihood: -11128.  
Df Model:         3              LL-Null:        -11643.  
Df Residuals:     16872          LLR p-value:  4.0003e-223  
Converged:       1.0000          Scale:          1.0000  
No. Iterations:   5.0000  
----  
          Coef.  Std.Err.      z    P>|z|    [0.025  0.975]  
----  
Intercept        0.5495    0.0415  13.2259  0.0000   0.4681  0.6310  
layup[T.True]    0.2014    0.0620   3.2477  0.0012   0.0799  0.3230  
dist             -0.0480   0.0021 -22.7194  0.0000  -0.0521 -0.0438  
dist:layup[T.True] -0.2687  0.0205 -13.0904  0.0000  -0.3089 -0.2284  
=====  
"""
```

Now to calculate the probability of making a shot given some distance (and layup indicator), we need to do something similar to before, but then run it through the logistic function.

```
In [2]:  
def prob_made_logit(dist, is_layup):  
    b0, b1, b2, b3 = logit_results.params  
    value = (b0 + b1*is_layup + b2*dist + b3*is_layup*dist)  
    return 1/(1 + math.exp(-value))  
  
In [3]: prob_made_logit(0, 1)  
Out[3]: 0.6793817567896825  
  
In [4]: prob_made_logit(15, 0)  
Out[4]: 0.45758877947023197  
  
In [5]: prob_made_logit(2, 1)  
Out[5]: 0.529377879897726
```

A logit model guarantees our predicted probability will be between 0 and 1. You should always use a logit instead of OLS when you're modeling some yes or no type outcome.

## Random Forest

Both linear and logistic regression are useful for:

1. Analyzing the relationships between data (looking at the coefficients).
2. Making predictions (running new data through a model to see what it predicts).

Random Forest models are much more of a black box. They're more flexible and make fewer assumptions about your data. This makes them great for prediction (2), but almost useless for analyzing relationships between variables (1).

Unlike linear or logistic regression, where your `y` variable has to be continuous or 0/1, Random Forests work well for classification problems, and we'll build one later in the chapter.

But let's start with some theory.

### 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, if we're trying to assign shot type (dunk, layup, step back) maybe the first split is on "distance from the hoop", where shots from within 2 feet go one direction, outside that go another.

Let's follow the 0-2 feet split. We good go to another split that looks at player height — if it's above the split point (say 6'11) we'll predict "dunk", below that, "layup".

Meanwhile the 2+ feet branch also continues onto its own, different split. Maybe it goes to shot clock — if there's more than X seconds on it, it goes one way, less another.

CART trees involve many split points. Details on how these points are selected are beyond the scope of this book, but essentially the computer goes through all possible variables and potential splits and picks the one that separates the data the "best". Then it sets that data aside and starts the process over with each subgroup. The final result is a bunch of if-then decision rules.

You can tell your program when to stop doing splits, either by: (1) telling it to keep going until all observations in a branch are "pure" (all classified the same thing), (2) telling it to split only a certain number of times, or (3) splitting until a branch reaches a certain number of samples.

Python seems to have sensible defaults for this, and 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 — if your output variable is continuous — the average 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.

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.

The default in Python is for each Random Forest to create 100 CART trees, but this is a parameter you have control over.

So the final result is stored as some number of trees, each trained on a different, random sample of your data. If you think about it, *Random Forest* is the perfect name for this model.

## **Using a Trained Random Forest to Generate Predictions**

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 for one particular observation, 80 of the trees might say dunk, 15 layup and 5 floater 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 a certain structure. This is nice for accuracy, but it also means random forests are much more susceptible to things like overfitting. 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 of Random Forest model.

*This example is in 07\_05\_random\_forest.py. We'll pick up right after importing our libraries and loading our shot data into a DataFrame named df, and doing some processing.*

In this example, we'll try to classify shot type (layup, pullup, floater, dunk, hook, fadeaway, step back, or other) given location and time left in the game data. That is, we'll model shot type as a function of:

```
In [1]: xvars = ['dist', 'x', 'y', 'period', 'time_left']  
In [2]: yvar = 'shot_type'
```

Let's look at a few of these observations:

```
In [3]: df[xvars + [yvar]].sample(10)  
Out[3]:  
   dist    x    y  period  time_left shot_type  
14209     6    2   67      4    9.866667    other  
8351      4   -5   48      3    7.800000    float  
9679      4   -11   46      1   10.066667  pullup  
13783     25   192  168      2    3.883333  pullup  
11471     14    82  120      1    4.633333  pullup  
11139     8   -42   73      1   10.333333    hook  
14125     10   101    6      2    9.100000  fadeaway  
6859      1   -8     8      3    7.350000    dunk  
6953      17   -3   175      1    6.266667    other  
1045     17   -63  168      1   9.250000  pullup
```

Along with a look at the variable we're predicting:

```
In [4]: df[yvar].value_counts(normalize=True)
Out[4]:
other      0.412064
layup      0.278621
pullup     0.118808
float       0.071996
dunk        0.055167
fadeaway    0.033717
hook        0.029628
```

So we're going to use our input variables: `dist`, `x`, `y`, `period` and `time_left` to predict our output variable `shot_type`.

### Holdout Set

Because tree based models like Random Forest are so flexible, it's meaningless to evaluate them on the same data you used to build the model — they'll perform too well. Instead, it's good practice to take a **holdout** set, i.e. set aside some portion of the data where you *know* the outcome you're trying to predict (shot type here) so you can evaluate the model on data that wasn't used to train it.

Scikit-learn's `train_test_split` function automatically does that. Here we have it randomly split our data 80/20 — 80% to build the model, 20% to test it.

```
In [5]: train, test = train_test_split(df, test_size=0.20)
```

Running the model takes place on two lines. Note the `n_estimators` option. That's the number of different trees the algorithm will run.

```
In [6]:
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.

Technically, we've just run our first Random Forest model. We can't see anything interesting yet because 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 dataset with some basic Pandas.

```
In [7]: test['shot_type_hat'] = model.predict(test[xvars])
In [8]: test['correct'] = (test['shot_type_hat'] == test['shot_type'])
In [9]: test['correct'].mean()
Out[9]: 0.6792061611374408
```

About 68%, not bad. Note, a Random Forest model includes randomness (hence the name) so when you run this yourself, you'll get something different.

Another thing it's interesting to look at is how confident the model is about each prediction. Remember, this model ran 100 different trees. Each of which classified every observation into one of: layup, pullup, float, dunk, hook, fadeaway, stepback or other. If the model assigned some shot dunk for 51/100 trees and layup for the other 49/100, we can interpret it as relatively unsure in its prediction.

Let's run each of our test samples through each of our 100 trees and check the frequencies. We can do this with the `predict_proba` method on `model`:

```
In [10]: model.predict_proba(test[xvars])
Out[10]:
array([[0.   , 0.09, 0.02, ..., 0.   , 0.57, 0.32],
       [0.   , 0.   , 0.   , ..., 0.   , 1.   , 0.   ],
       [0.   , 0.   , 0.   , ..., 0.   , 0.89, 0.11],
       ...,
       [0.   , 0.01, 0.29, ..., 0.44, 0.23, 0.01],
       [0.   , 0.   , 0.   , ..., 0.   , 0.96, 0.04],
       [0.   , 0.1 , 0.2 , ..., 0.02, 0.5 , 0.15]])
```

This is just a raw, unformatted matrix. Let's put it into a DataFrame, making sure to give it the same index as `test`:

```
In [9]:
probs = DataFrame(model.predict_proba(test[xvars]),
                  index=test.index,
                  columns=model.classes_)

In [10]: probs.head()
Out[10]:
      dunk  fadeaway  float  hook  layup  other  pullup
6781  0.00     0.09   0.02   0.0    0.00   0.57    0.32
9791  0.00     0.00   0.00   0.0    0.00   1.00    0.00
12449 0.00     0.00   0.00   0.0    0.00   0.89    0.11
14285 0.04     0.00   0.00   0.0    0.96   0.00    0.00
6220  0.00     0.00   0.01   0.0    0.00   0.49    0.50
```

We're looking at the first 5 rows of our holdout dataset here. We can see the model says the first observation has a 32% of being a pullup, a 9% of being a fadeaway, and 57% of being other.

Let's bring in the actual, known shot type from our test dataset.

```
In [11]:
results = pd.concat([test[['name', 'dist', 'shot_type', 'shot_type_hat',
                           'correct']], probs], axis=1)
```

We can look at this to see how our model did for different shot types.

```
In [12]:  
results.groupby('shot_type')[['correct', 'layup', 'pullup',  
    'float', 'dunk', 'hook', 'fadeaway', 'other']].mean().round(2)
```

Out[12]:

shot_type	correct	layup	pullup	float	dunk	hook	fadeaway	other
dunk	0.09	0.72	0.00	0.01	0.25	0.01	0.00	0.01
fadeaway	0.13	0.06	0.19	0.21	0.00	0.06	0.16	0.31
<b>float</b>	0.42	0.20	0.12	0.31	0.01	0.10	0.09	0.17
hook	0.10	0.35	0.04	0.28	0.02	0.14	0.05	0.12
layup	0.87	0.71	0.01	0.06	0.15	0.04	0.01	0.03
other	0.86	0.01	0.16	0.03	0.00	0.01	0.02	0.77
pullup	0.27	0.01	0.31	0.08	0.00	0.01	0.06	0.54

Somewhat surprisingly, the model performs worst on dunks, getting them correct only 9% of the time. They're usually misclassified as layups, which makes sense. One thing that might help — if we some player info (like height and weight) in the model. I'd expect taller players would dunk the ball more often.

Working with a holdout dataset let's us do interesting things and is conceptually easy to understand. It's also noisy, especially with small datasets. Different, random holdout sets can give widely fluctuating accuracy numbers. This isn't ideal, which is why an alternative called cross validation is more common.

## Cross Validation

**Cross validation** reduces noise, basically by taking *multiple* holdout sets and blending them together.

How it works: 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.

Besides being less noisy, cross validation lets you get more out of your data. Every observation contributes, vs only the 80% (or whatever percentage you use) with a train-test split. One disadvantage is its more computationally intensive since you're running 10x as many models.

To run cross validation, you create model like we did above. But instead of calling `fit` on it, you pass it to the `scikit-learn` function `cross_val_score`.

```
In [1]: model = RandomForestClassifier(n_estimators=100)
```

```
In [2]: 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. This gives back 10 separate scores, which we can look at and average.

```
In [3]: scores
Out[3]:
array([0.67476303, 0.67535545, 0.6735782 , 0.66706161, 0.67298578,
       0.66883886, 0.68168346, 0.67753408, 0.67753408, 0.67279194])

In [4]: scores.mean()
Out[4]: 0.6742126506853356
```

Again, your results will vary, both due to the randomness of the Random Forest models, as well as the cross validation splits.

### Feature Importance in Random Forest

Finally, although we don't have anything like the coefficients we get with linear regressions, the model does output some information on which variables are most important (e.g. made the biggest difference in being able to split correctly or not).

Scikit learn lets you do that with the `feature_importance_` attribute on your fitted model.

(Note: when you get behind basic linear regression and start to move into scikit-learn and machine learning, people start calling input data *features* instead of variables or columns.)

The feature importance info isn't available after a cross validation — only after a regular model run — so let's run our model again first.

Since it's our final model, let's run it on everything, training + test. Again, this is *not* a good thing to do when deciding between models and which variables to include. You should use cross validation or a holdout set for that. But once we've used cross validation to pick a model, we can run it with everything to include the most data possible.

```
In [1]: model = RandomForestClassifier(n_estimators=100)

In [2]: model.fit(dfs[xvars], dfs[yvar])
Out[2]: RandomForestClassifier()
```

Then to look at the feature importances:

```
In [3]: Series(model.feature_importances_, xvars).sort_values(
            ascending=False)
Out[3]:
dist      0.284295
y         0.242617
x         0.227179
time_left 0.194377
period    0.051532
```

So distance from the hoop is most important, followed by x, y coordinates (these sum to 0.4698, and so together are more important than distance) and time left in game info.

There you go, you've run your first Random Forest model.

## Random Forest Regressions

`RandomForestClassifier` is the `scikit-learn` model for modeling an output variable with discrete categories (shot type, position, made basket or not, etc).

If you're modeling a continuous valued variable (like points, rebounds, or plus-minus) you do the exact same thing, but with `RandomForestRegressor` instead.

When might you want to use `RandomForestRegressor` vs the OLS and `statsmodels` techniques we covered earlier?

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 a Random Forest<sup>6</sup>.

---

<sup>6</sup>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_make` and the `apply` function, create a new column in the data `make_hat_alt` – how does it compare to `results.predict(df)`?
- b) Add `C(value)` to the probability of made basket model, and look the results. Controlling for everything else in our regression, which is a three or two more likely to go in? Why do you think this is?
- c) Run the same model without the `C(value)` syntax, creating a dummy variable 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 (a) 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

Load your team-game data into a DataFrame named `dftg`.

- a) Run a logit model regressing three point percentage, offensive and defensive rebounds, steals, turnovers and blocks on whether a team wins.

Which helps a team more — one more steal or avoiding a turnover? Why do you think this is?

- b) Add some team fixed effects to the model. What type of teams would you expect to have relatively higher coefficients in this model?

### 7.4

- a) Use your team-game data to build a random forest classification model that predicts team based on stats.
- b) There are a few variables here that it would *not* make sense to use. What are they?
- c) Run a cross validation on your model. What's the average and range of % of observations your model got right?
- d) Is this model better than random? What percentage would you expect to get right if you picked a team out at random?
- e) What are the most important features in this model?

## 8. Intermediate Coding and Next Steps: High Level Strategies

If you've made it this far 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!). But moving beyond the basics is less "things you can do with DataFrames #6-10" and more about mindset, high level strategies and getting experience. 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 level (e.g. Gall's Law) and low level (get your code working then put it in a function), but all of them should help as you move beyond the self-contained examples in this book.

### 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 mind as you start working on your own projects is *Gall's Law*.

Applied to programming, it 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 version of some project or even some of the extended examples in this book and think "there's no way I could ever do that." But if I just sat down and tried to write these complete programs off the top of my head I wouldn't be able to 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 tunneling through a giant wall of rock. When starting, your job is to get a tiny hole through to the other side, even if it 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 DataFrames together might be complex enough that you'll want to examine your outputs to make sure everything works.

As you get more experience and practice 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

A 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 on the right, as well as getting comfortable with the shortcut 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 you write it helps you spot and fix these errors as they happen.

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 write, it's going to be way more painful when you eventually do.

## Use Functions

For me, the advice above (start simple + get quick feedback) usually means writing simple, working code in the "top level" (the main, regular Python file; as opposed to inside a function).

Then — after I've examined the outputs in the REPL and am confident some particular piece works — I'll usually put it inside a function.

Functions have two benefits: (1) DRY and (2) letting you set aside and abstract parts of your code.

## DRY: Don't Repeat Yourself

A popular maxim among programmers is “DRY” for [Don’t Repeat Yourself](#)<sup>1</sup>.

For example: say you need to run some similar code a bunch of times. Maybe it’s code that summarizes points by position, and you need to run it for all the SG, PGs, PFs etc.

The naive approach would be to get it working for one position, then copy and paste it a bunch of times, making 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. This is tedious at best and error-prone at worst.

But if you put the code in a function, with arguments to allow for your slightly different use cases, you only have to fix it in one spot when you 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 remember our win probability calculation every time we want to do it. We just use the function.

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 probability!) when reading through our program.

This is another reason it’s usually better to use 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 if you adopt a certain mindset.

---

<sup>1</sup>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

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 what's 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 = {'PF': 'kevin durant',
               'SG': 'kyrie irving',
               'PG': 'james harden'}
```

And turn all the player names to uppercase.

When we're first starting out, maybe we do something like:

```
In [1]: roster_dict1 = {}
In [2]: for pos in roster_dict:
            roster_dict1[pos] = roster_dict[pos].upper()

In [3]: roster_dict1
Out[3]: {'PF': 'KEVIN DURANT', 'SG': 'KYRIE IRVING', 'PG': 'JAMES HARDEN'}
```

Then we learn about comprehensions and realize we could just do this on one line.

```
In [4]: roster_dict2 = {pos: roster_dict[pos].upper()
                        for pos in roster_dict}

In [5]: roster_dict2
Out[5]: {'PF': 'KEVIN DURANT', 'SG': 'KYRIE IRVING', 'PG': 'JAMES HARDEN'}
```

Then later we learn about `.items` in dictionary comprehensions and realize we can write the same thing:

```
In [6]: roster_dict3 = {pos: name.upper()
                        for pos, name in roster_dict.items()}

In [7]: roster_dict3
Out[7]: {'PF': 'KEVIN DURANT', 'SG': 'KYRIE IRVING', 'PG': 'JAMES HARDEN'}
```

This 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 well designed. Over time these things will accumulate and eventually you’ll be able to do things that people who don’t care about their code 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 that you should be consciously thinking about these decisions and be *deliberate*; 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. If you’ve made it this far, you probably do too. I’d encourage you to cultivate 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.
- Don’t repeat yourself and think more clearly by putting common code in functions.
- Care about the design your code and keep trying to get better at.

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 on *any* topic, not just basketball. 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](http://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](mailto:nate@nathanbraun.com)

# Appendix A: Places to Get Data

This appendix lists a few places to get NBA data.

## **nba\_api Python wrapper**

After looking around a bit, I think the best option for most people going to be to use the [nba\\_api](#) Python wrapper. That's what I used to get the sample datasets we've been using here. See chapter 5 for an in-depth look at it.

## **Other Options**

### **Josh Gonzales's List of NBA Data Sources**

After doing some googling, I stumbled across [this list of NBA data sources by Josh Gonzales on Medium](#). From looking at them, I think most of the best ones ultimately are getting their data from [nba\\_api](#), so that's still my recommendation.

### **Kaggle.com**

Kaggle.com is best known for its modeling and machine learning competitions, but it also has a dataset search engine with some basketball related datasets.

<https://www.kaggle.com/datasets>

### **Google Dataset Search**

Google has a dataset search engine with some interesting datasets:

<https://toolbox.google.com/datasetsearch>

# 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 Nielsen 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.

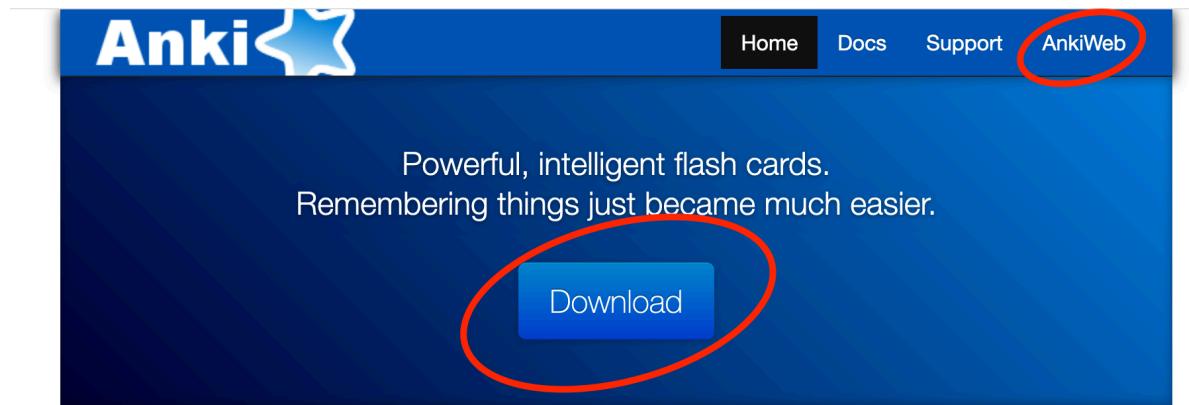
Anki should be useful to everyone reading this book — after all, you bought this book because you want to remember it — but it'll be particularly helpful for readers who don't have the opportunity to program in Python or Pandas regularly. When I learned how to code, I found it didn't necessarily "stick" 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.

## 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 apk file (included with the book) you're importing.

For instance, after you're done with the prerequisite Tooling section of the book, 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) player (and game)
- b) team
- c) team-game
- d) shot type
- e) jersey number

### 1.2

- a) Combined shooting percentage over last 5 games; total numbers of shots over last 5 games.
- b) Number of points a team will score in a game.
- c) This model is at the team and game level.
- d) This is subjective, but I'd say the biggest limitation is it includes no information about the teams upcoming opponent and how good their defense is. That's a big factor in how many points a team will score.

### 1.3

- 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) `_throwaway_data`. Valid. Python programmers often start variables with `_` if they're throwing away or temporary, short term variables.
- b) `n_shots`. Valid.
- c) `3_pt_percentage`. Not valid. Can't start with a number.
- d) `numOfBoards`. Valid, though convention is to split words with `_`, not camelCase.
- e) `flagrant2`. Valid. Numbers OK as long as they're not in the first spot
- f) `coach name`. Not valid. No spaces
- g) `@home_or_away`. Not valid. Only non alphanumeric character allowed is `_`
- h) `'ft_attempts'`. Not valid. A string (wrapped in quotes), not a variable name. Again, only non alphanumeric character allowed is `_`

### 2.2

```
In [1]:  
weekly_points = 100  
weekly_points = weekly_points + 28  
weekly_points = weekly_points + 5  
  
In [2]: weekly_points # 133  
Out[2]: 133
```

### 2.3

```
In [1]:  
def commentary(player, play):  
    return f'{player} with the {play}!'  
  
--  
  
In [2]: commentary('Lebron', 'dunk')  
Out[2]: 'Lebron with the dunk!'
```

### 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 [3]: 'lebron james'.islower() # should return True
Out[3]: True

In [4]: 'Lebron James'.islower() # should return False
Out[4]: False
```

## 2.5

```
In [5]:
def is_fox(player):
    return player.replace("'", '').lower() == 'deaaron fox'

In [6]: is_fox('lebron james')
Out[6]: False

In [7]: is_fox("De'Aaron Fox")
Out[7]: True

In [8]: is_fox("DEAARON FOX")
Out[8]: True
```

## 2.6

```
In [9]:
def is_good_score(score):
    if score >= 100:
        return f'{score} is a good score'
    else:
        return f'{score}'s not that good"
In [10]: is_good_score(90)
Out[10]: "90's not that good"

In [11]: is_good_score(130)
Out[11]: '130 is a good score'
```

## 2.7

Here's what I came up with. The last three use list comprehensions.

```
roster[0:2]
roster[:2]
roster[:-1]
[x for x in roster if x != 'james harden']
[x for x in roster if not x.startswith('j')]
[x for x in roster if x in ['kevin durant', 'kyrie irving']]
```

## 2.8a

```
In [12]: shot_info['shooter'] = 'Devon Booker'

In [13]: shot_info
Out[13]: {'shooter': 'Devon Booker', 'is_3pt': True, 'went_in': False}
```

## 2.8b

```
In [14]:
def toggle3(info):
    info['is_3pt'] = not info['is_3pt']
    return info

In [15]: shot_info
Out[15]: {'shooter': 'Devon Booker', 'is_3pt': True, 'went_in': False}

In [16]: toggle3(shot_info)
Out[16]: {'shooter': 'Devon Booker', 'is_3pt': False, 'went_in': False}
```

## 2.9

- a) No. 'is\_ft' hasn't been defined.
- b) No, `shooter` is a variable that hasn't been defined, the key is 'shooter'.
- c) Yes.

## 2.10a

```
In [17]:
for x in roster:
    print(x.split(' ') [-1])

durant
irving
harden
```

## 2.10b

```
In [18]: {player: len(player) for player in roster}
Out[18]: {'kevin durant': 12, 'kyrie irving': 12, 'james harden': 12}
```

### 2.11a

```
In [19]: [pos for pos in roster_dict]
Out[19]: ['PF', 'SG', 'PG', 'C']
```

### 2.11b

```
In [20]:
[player for _, player in roster_dict.items()
 if player.split(' ')[-1][0] in ['h', 'j']]
Out[20]: ['james harden', 'deandre jordan']
```

### 2.12a

```
def mapper(my_list, my_function):
    return [my_function(x) for x in my_list]
```

### 2.12b

```
In [21]: mapper(list_of_n_3pt_made, lambda x: x*3)
Out[21]: [15, 18, 3, 0, 12, 12]
```

## 3.0 Pandas Basics

### 3.0.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
games = pd.read_csv(path.join(DATA_DIR, 'games.csv'))
```

### 3.0.2

```
# works because data is sorted by date already
In [2]: games50 = games.head(50)

# this is better if don't want to assume data is sorted
In [3]: games50 = games.sort_values('date').head(50)
```

### 3.0.3

```
In [4]: games.sort_values('home_pts', ascending=False, inplace=True)

In [5]: games.head()
Out[5]:
   game_id  home  away  ...  bubble  sample  season
281    21900282   HOU   ATL  ...   False   False  2019-20
60     21900061   WAS   HOU  ...   False   False  2019-20
686    21900687   ATL   WAS  ...   False   False  2019-20
704    21900705   MIL   WAS  ...   False   False  2019-20
181    21900182   LAC   ATL  ...   False   False  2019-20
```

Note: if this didn't work when you printed it on a new line in the REPL you probably forgot the `inplace=True` argument.

### 3.0.4

```
In [6]: type(games.sort_values('home_pts')) # it's a DataFrame
Out[6]: pandas.core.frame.DataFrame
```

### 3.0.5a

```
In [7]:
game_simple = games[['date', 'home', 'away', 'home_pts', 'away_pts']]
```

### 3.0.5b

```
In [8]:  
game_simple = game_simple[['home', 'away', 'date', 'home_pts', 'away_pts']]
```

### 3.0.5c

```
In [9]: game_simple['game_id'] = games['game_id']
```

### 3.0.5d

```
In [10]: games.to_csv(path.join(DATA_DIR, 'game_simple.txt'), sep='|')
```

## 3.1 Columns

### 3.1.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
pg = pd.read_csv(path.join(DATA_DIR, 'player_game.csv'))
```

### 3.1.2

```
In [1]: pg['net_takeaways'] = pg['stl'] - pg['tov']

In [2]: pg['net_takeaways'].head()
Out[2]:
0    -4
1     0
2    -1
3     0
4    -1
```

### 3.1.3

```
In [3]: pg['player_desc'] = pg['name'] + ' is the ' + pg['team'] + ' '
...: + pg['pos']

In [4]: pg['player_desc'].head()
Out[4]:
0    L. James is the LAL Forward
1    D. Howard is the LAL Center
2    L. Williams is the LAC Guard
3    J. Dudley is the LAL Forward
4    J. McGee is the LAL Center
```

### 3.1.4

```
In [5]: pg['bad_game'] = (pg['fga'] > 20) & (pg['pts'] < 15)

In [6]: pg['bad_game'].head()
Out[6]:
0    False
1    False
2    False
3    False
4    False
```

### 3.1.5

```
In [7]: pg['len_last_name'] = (pg['name']
                             .apply(lambda x: len(x.split('.')[-1])))

In [8]: pg['len_last_name'].head()
Out[8]:
0    6
1    7
2    9
3    7
4    6
```

### 3.1.6

```
In [9]: pg['game_id'] = pg['game_id'].astype(str)
```

### 3.1.7a

```
In [10]:
pg.columns = [x.replace('_', ' ') for x in pg.columns]
pg.head()

Out[10]:
      name  fgm  ...  bad game  len last name
0    L. James    7  ...   False        6
1    D. Howard   1  ...   False        7
2  L. Williams   8  ...   False        9
3    J. Dudley   2  ...   False        7
4    J. McGee    2  ...   False        6
```

### 3.1.7b

```
In [11]:
pg.columns = [x.replace(' ', '_') for x in pg.columns]
pg.head()

Out[11]:
      name  fgm  ...  bad_game  len_last_name
0    L. James    7  ...   False        6
1    D. Howard   1  ...   False        7
2  L. Williams   8  ...   False        9
3    J. Dudley   2  ...   False        7
4    J. McGee    2  ...   False        6
```

### 3.1.8a

```
In [12]:  
pg['oreb_percentage'] = pg['oreb']/pg['reb']  
pg['oreb_percentage'].head()  
  
Out[12]:  
0    0.1  
1    0.5  
2    0.2  
3    NaN  
4    0.5
```

### 3.1.8b

'oreb\_percentage' is offensive rebounds divided by total rebounds. Since you can't divide by 0, `oreb_percentage` is missing whenever a player had 0 rebounds.

To replace all the missing values with -99:

```
In [13]: pg['oreb_percentage'].fillna(-99, inplace=True)  
  
In [14]: pg['oreb_percentage'].head()  
Out[14]:  
0    0.1  
1    0.5  
2    0.2  
3   -99.0  
4    0.5
```

### 3.1.9

```
In [15]: pg.drop('oreb_percentage', axis=1, inplace=True)  
  
In [16]: pg.head()  
Out[16]:  
      name  fgm  ...  bad_game  len_last_name  
0    L. James    7  ...    False          6  
1    D. Howard   1  ...    False          7  
2  L. Williams   8  ...    False          9  
3    J. Dudley   2  ...    False          7  
4    J. McGee    2  ...    False          6
```

If you forget the `axis=1` Pandas will try to drop the *row* with the index value '`oreb_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 `'oreb_percentage'` column. Nothing happens to the original `pg`, though we could reassign it if we wanted like this:

```
pg = pg.drop('oreb_percentage', axis=1)
```

## 3.2 Built-in Functions

### 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 [1]: pg['total_shots1'] = pg['fga'] + pg['fta']

In [2]: pg['total_shots2'] = pg[['fga', 'fta']].sum(axis=1)

In [3]: (pg['total_shots1'] == pg['total_shots2']).all()
Out[3]: True
```

### 3.2.3a

```
In [4]: pg[['pts', 'fga', 'reb']].mean()
Out[4]:
pts      10.659413
fga      8.403500
reb      4.242668
```

### 3.2.3b

```
In [5]: ((pg['pts'] >= 40) & (pg['reb'] >= 10)).sum() # 3
Out[5]: 3
```

### 3.2.3c

```
In [6]:
(((pg['pts'] >= 40) & (pg['reb'] >= 10)).sum() / (pg['pts'] >= 40).sum())
--
Out[6]: 0.3
```

### 3.2.3d

```
In [7]: pg['fg3a'].sum()  
Out[7]: 6809
```

### 3.2.3e

Most: 14, least: 8

```
In [8]: pg['team'].value_counts() # ORL - 152 times  
Out[8]:  
ORL    152  
UTA    136  
LAC    116  
LAL    113  
WAS    107  
PHX    104  
SAS     97  
IND     95  
NOP     87  
PHI     85  
MIA     81  
DEN     75  
NYK     73  
MIL     72  
POR     72  
OKC     71  
BKN     64  
DAL     61  
TOR     59  
BOS     56  
HOU     54  
MEM     52  
GSW     47  
CLE     41  
DET     40  
ATL     33  
SAC     32  
MIN     20  
CHI     10  
CHA      9
```

### 3.3 Filtering

#### 3.3.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
dftg = pd.read_csv(path.join(DATA_DIR, 'team_games.csv'))
```

#### 3.3.2a

```
In [1]: dftg_chi1 = dftg.loc[dftg['team'] == 'CHI',
                           ['team', 'date', 'pts', 'fgm', 'fga']]
In [2]: dftg_chi1.head()
Out[2]:
   team      date  pts  fgm  fga
2053  CHI  2020-03-10  108   42   84
2054  CHI  2020-03-08  107   39   92
2055  CHI  2020-03-06  102   40   97
2056  CHI  2020-03-04  108   45   94
2057  CHI  2020-03-02  109   44   94
```

#### 3.3.2b

```
In [3]: dftg_chi2 = dftg.query("team == 'CHI'")[
                           ['team', 'date', 'pts', 'fgm', 'fga']]
```

#### 3.3.3

```
In [4]: dftg_no_chi = dftg.loc[dftg['team'] != 'CHI',
                           ['team', 'date', 'pts', 'fgm', 'fga']]
In [5]: dftg_no_chi.head()
Out[5]:
   team      date  pts  fgm  fga
0  HOU  2020-08-14   96   35   80
1  HOU  2020-08-12  104   32   86
2  HOU  2020-08-11  105   36   89
3  HOU  2020-08-09  129   43   91
4  HOU  2020-08-06  113   36   84
```

### 3.3.4a

Yes.

```
In [6]: dftg[['fga', 'fgm', 'fg3a', 'fg3m']].duplicated().any()
Out[6]: True

In [7]: dftg[['fga', 'fgm', 'fg3a', 'fg3m']].duplicated().sum() # 32
Out[7]: 32
```

### 3.3.4b

```
In [8]: dups = dftg[['fga', 'fgm', 'fg3a', 'fg3m']].duplicated(keep=False)

In [9]: dftg_fg_dup = dftg.loc[dups]

In [10]: dftg_fg_no_dup = dftg.loc[~dups]
```

### 3.3.5

```
In [11]:
import numpy as np

dftg['three_pt_desc'] = np.nan
dftg.loc[dftg['fg3_pct'] > .5, 'three_pt_desc'] = 'great'
dftg.loc[dftg['fg3_pct'] <= .25, 'three_pt_desc'] = 'brutal'
dftg[['fg3_pct', 'three_pt_desc']].sample(5)

Out[11]:
    fg3_pct three_pt_desc
1912      0.257        NaN
687       0.333        NaN
776       0.415        NaN
1083      0.281        NaN
1287      0.543     great
```

### 3.3.6a

```
In [12]: dftg_no_desc1 = dftg.loc[dftg['three_pt_desc'].isnull()]
```

### 3.3.6b

```
In [13]: dftg_no_desc2 = dftg.query("three_pt_desc.isnull()")
```

## 3.4 Granularity

### 3.4.1

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 whether or not Lebron made every single shot to just knowing how many points he scored *total*, that's a loss of information.

### 3.4.2a

```
import pandas as pd
from os import path

DATA_DIR = './data'
dftg = pd.read_csv(path.join(DATA_DIR, 'team_games.csv'))
```

### 3.4.2b

```
In [1]: dftg.groupby('team')['pts'].mean()

Out[1]:
team
ATL    111.761194
BKN    111.777778
BOS    113.652778
CHA    102.876923
CHI    106.846154
CLE    106.892308
DAL    117.013333
DEN    111.287671
DET    107.242424
GSW    106.338462
HOU    117.805556
IND    109.438356
LAC    116.347222
LAL    113.436620
MEM    112.630137
MIA    112.041096
MIL    118.671233
MIN    113.250000
NOP    115.847222
NYK    105.803030
OKC    110.416667
ORL    107.273973
PHI    110.739726
PHX    113.616438
POR    114.972973
SAC    110.097222
SAS    114.056338
TOR    112.750000
UTA    111.291667
WAS    114.416667
```

### 3.4.2c

Just change sum of 'yards\_gained' to mean:

```
In [2]: dftg['gt_100'] = dftg['pts'] >= 100  
In [3]: dftg.groupby('team')['gt_100'].mean()  
Out[3]:  
team  
ATL    0.880597  
BKN    0.888889  
BOS    0.875000  
CHA    0.646154  
CHI    0.769231  
CLE    0.784615  
DAL    0.973333  
DEN    0.876712  
DET    0.772727  
GSW    0.738462  
HOU    0.930556  
IND    0.849315  
LAC    0.902778  
LAL    0.873239  
MEM    0.849315  
MIA    0.876712  
MIL    0.958904  
MIN    0.859375  
NOP    0.958333  
NYK    0.696970  
OKC    0.833333  
ORL    0.657534  
PHI    0.767123  
PHX    0.876712  
POR    0.905405  
SAC    0.875000  
SAS    0.929577  
TOR    0.861111  
UTA    0.861111  
WAS    0.902778
```

### 3.4.2d

5

```
In [4]: dftg['gt_150'] = dftg['pts'] >= 150

In [5]: dftg.groupby('team')['gt_150'].any()
Out[5]:
team
ATL      True
BKN     False
BOS     False
CHA     False
CHI     False
CLE     False
DAL     False
DEN     False
DET     False
GSW     False
HOU      True
IND     False
LAC      True
LAL     False
MEM     False
MIA     False
MIL      True
MIN     False
NOP     False
NYK     False
OKC     False
ORL     False
PHI     False
PHX     False
POR     False
SAC     False
SAS     False
TOR     False
UTA     False
WAS      True

In [6]: dftg.groupby('team')['gt_150'].any().sum()
Out[6]: 5
```

### 3.4.2e

```
In [7]: dftg.groupby('date')['team_id'].count().head()
Out[7]:
date
2019-10-22      4
2019-10-23     22
2019-10-24      6
2019-10-25     18
2019-10-26     20

In [8]: dftg.groupby('date')['team_id'].sum().head()
Out[8]:
date
2019-10-22    6442450994
2019-10-23   35433480576
2019-10-24   9663676486
2019-10-25  28991029551
2019-10-26  32212255022
```

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.

### 3.4.3a

```
In [1]:
dftg2 = dftg.groupby(['team_id', 'wl']).agg(
    ave_pts = ('pts', 'mean'),
    ave_fgm = ('fgm', 'mean'),
    ave_fga = ('fga', 'mean'),
    ave_fg3m = ('fg3m', 'mean'),
    ave_fg3a = ('fg3a', 'mean'),
    n = ('team_id', 'count'))
```

### 3.4.3b

```
In [2]: dftg2.reset_index(inplace=True)
```

**3.4.3c**

```
# loc
In [3]: (dftg2.loc[dftg2['wl'] == 'L', 'ave_pts'] > 110).sum() # 4
Out[3]: 4

# or with query
In [4]: (dftg2.query("wl == 'L'")['ave_pts'] > 110).sum() # 4
Out[4]: 4
```

**3.4.3d**

```
In [5]:
dftg3 = dftg.groupby(['team', 'wl']).agg(
    ave_pts = ('pts', 'mean'),
    ave_fgm = ('fgm', 'mean'),
    ave_fga = ('fga', 'mean'),
    ave_fg3m = ('fg3m', 'mean'),
    ave_fg3a = ('fg3a', 'mean'),
    n = ('team', 'count'))
```

**3.4.3e**

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 team-win/loss level to the team level. If we stacked it, we'd go from rows being:

team	wl	ave_pts	ave_fgm	ave_fga	ave_fg3m	ave_fg3a	n
ATL	L	107.382979	39.127660	90.936170	11.382979	36.425532	47
	W	122.050000	44.200000	89.650000	13.500000	35.200000	20
BKN	L	107.216216	39.027027	91.135135	12.459459	38.918919	37
	W	116.600000	41.828571	89.314286	13.714286	37.314286	35
BOS	L	105.416667	38.041667	90.208333	10.958333	35.541667	24

To:

wl	ave_pts	ave_fgm	...	ave_fg3m	ave_fg3a	n	W	L	W
team	L	W	L	...	W	L	W	L	W
ATL	107.0	122.0	39.0	...	14.0	36.0	35.0	47	20
BKN	107.0	117.0	39.0	...	14.0	39.0	37.0	37	35
BOS	105.0	118.0	38.0	...	13.0	36.0	34.0	24	48
CHA	100.0	107.0	37.0	...	13.0	34.0	34.0	42	23
CHI	103.0	114.0	39.0	...	14.0	36.0	34.0	43	22

```
In [6]: dftg3.unstack().head()
Out[6]:
      ave_pts          ave_fgm   ...    ave_fg3a      n
wl        L            W       L   ...        W    L    W
team
ATL  107.382979  122.050000  39.127660   ...  35.200000  47  20
BKN  107.216216  116.600000  39.027027   ...  37.314286  37  35
BOS  105.416667  117.770833  38.041667   ...  34.041667  24  48
CHA  100.500000  107.217391  36.761905   ...  34.391304  42  23
CHI  102.953488  114.454545  38.511628   ...  34.318182  43  22
```

### 3.4.4

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.

## 3.5 Combining DataFrames

### 3.5.1a

```
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'))
```

### 3.5.1b

```
In [1]: df_comb1 = pd.merge(df_pts, df_reb)

In [2]: df_comb1 = pd.merge(df_comb1, df_def, how='left')

In [3]: df_comb1 = df_comb1.fillna(0)
```

### 3.5.1c

```
In [4]:
df_comb2 = pd.concat([df_pts.set_index(['player_id', 'game_id']),
                      df_reb.set_index(['player_id', 'game_id']),
                      df_def.set_index(['player_id', 'game_id'])],
                     join='outer', axis=1)

In [5]: df_comb2 = df_comb2.fillna(0)
```

### 3.5.1d

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.

### 3.5.2a

```
import pandas as pd
from os import path

DATA_DIR = './data'
df_c = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'center.csv'))
df_f = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'forward.csv'))
df_g = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'guard.csv'))
```

### 3.5.2b

```
In [6]: df = pd.concat([df_c, df_f, df_g], ignore_index=True)
```

### 3.5.3a

```
import pandas as pd
from os import path

DATA_DIR = './data'
dft = pd.read_csv(path.join(DATA_DIR, 'teams.csv'))
```

### 3.5.3b

```
In [7]:
for conf in ['East', 'West']:
    (dft
        .query(f"conference == '{conf}'")
        .to_csv(path.join(DATA_DIR, f'dft_{conf}.csv'), index=False))
```

Write a two line for loop to save subsets of the ADP data frame for each position.

### 3.5.3c

```
In [8]:
df = pd.concat([pd.read_csv(path.join(DATA_DIR, f'dft_{conf}.csv'))
    for conf in ['East', 'West']], 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.

### 4.1

```
SELECT
    date, name, fgm, fga, pts
FROM player_game, team
WHERE
    team.team = player_game.team AND
    team.division = 'Central'
```

### 4.2

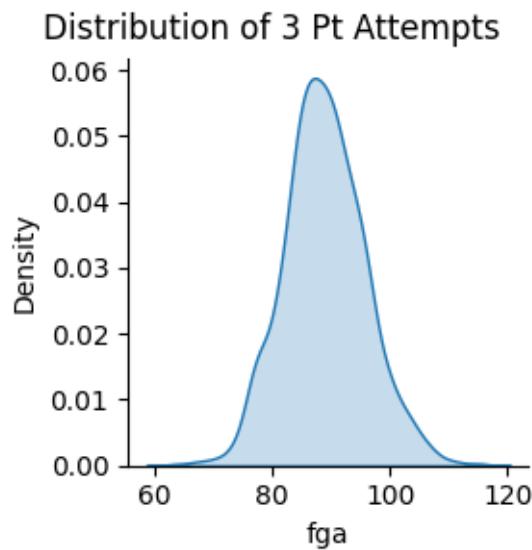
```
SELECT
    p.first, p.last, date, fgm, fga, pts
FROM player_game AS pg, team AS t, player AS p
WHERE
    t.team = pg.team AND
    t.division = 'Central' AND p.player_id = pg.player_id
```

## 6. Summary and Data Visualization

Assuming you've loaded the team game data into a DataFrame named dftg and imported seaborn as sns.

### 6.1a

```
g = (sns.FacetGrid(dftg)
      .map(sns.kdeplot, 'fga', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of 3 Pt Attempts')
```

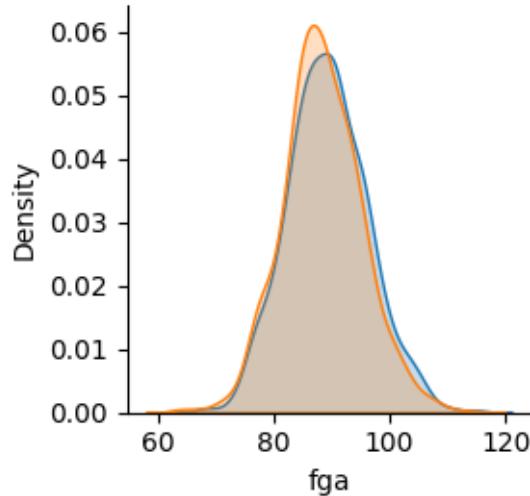


**Figure 0.1:** Solution 6-1a

### 6.1b

```
g = (sns.FacetGrid(dftg, hue='wl')
      .map(sns.kdeplot, 'fga', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of 3 Pt Attempts by Win/Loss B')
```

Distribution of 3 Pt Attempts by Win/Loss B

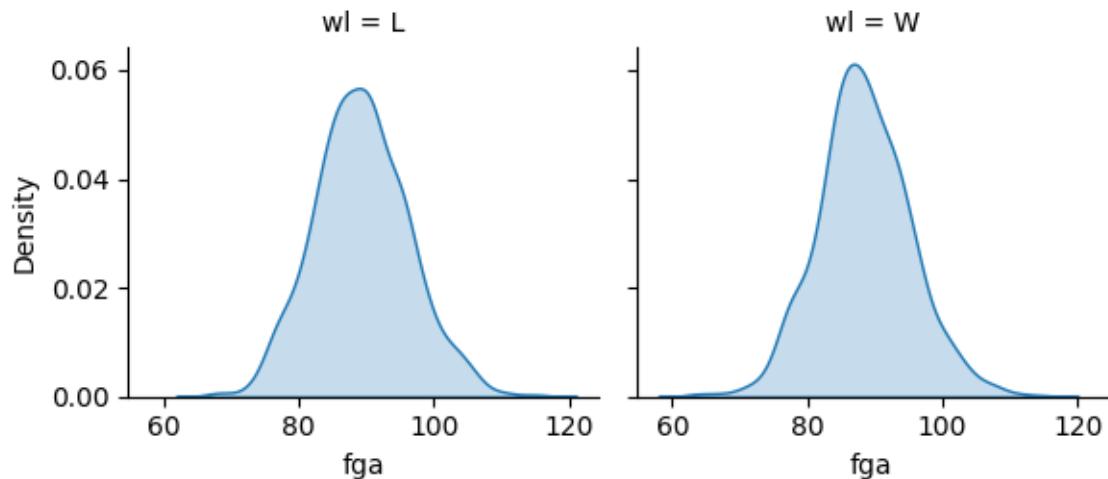


**Figure 0.2:** Solution 6-1b

### 6.1c

```
g = (sns.FacetGrid(dftg, col='wl')
      .map(sns.kdeplot, 'fga', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of 3 Pt Attempts by Win/Loss C')
```

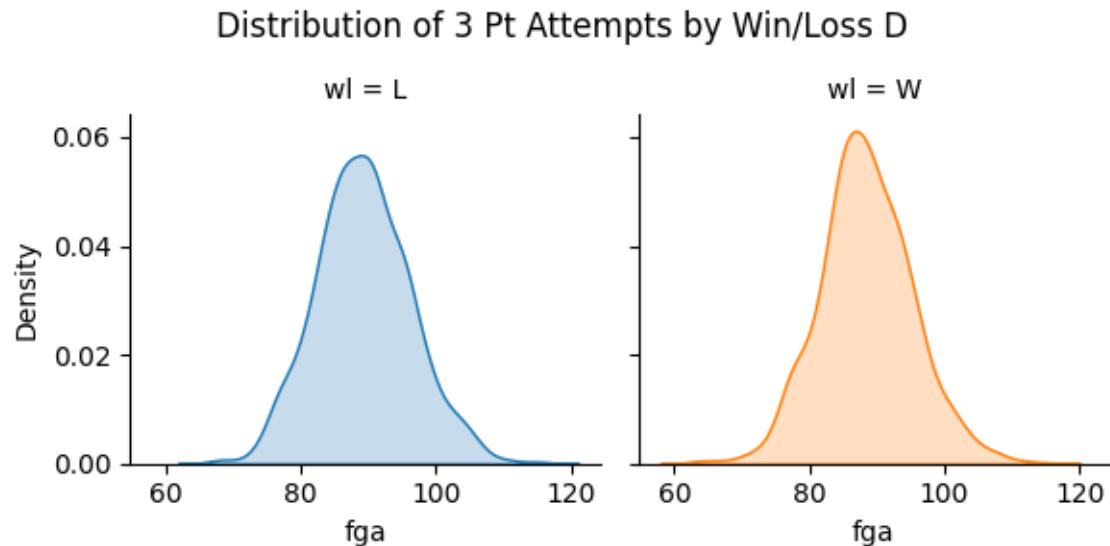
Distribution of 3 Pt Attempts by Win/Loss C



**Figure 0.3:** Solution 6-1c

### 6.1d

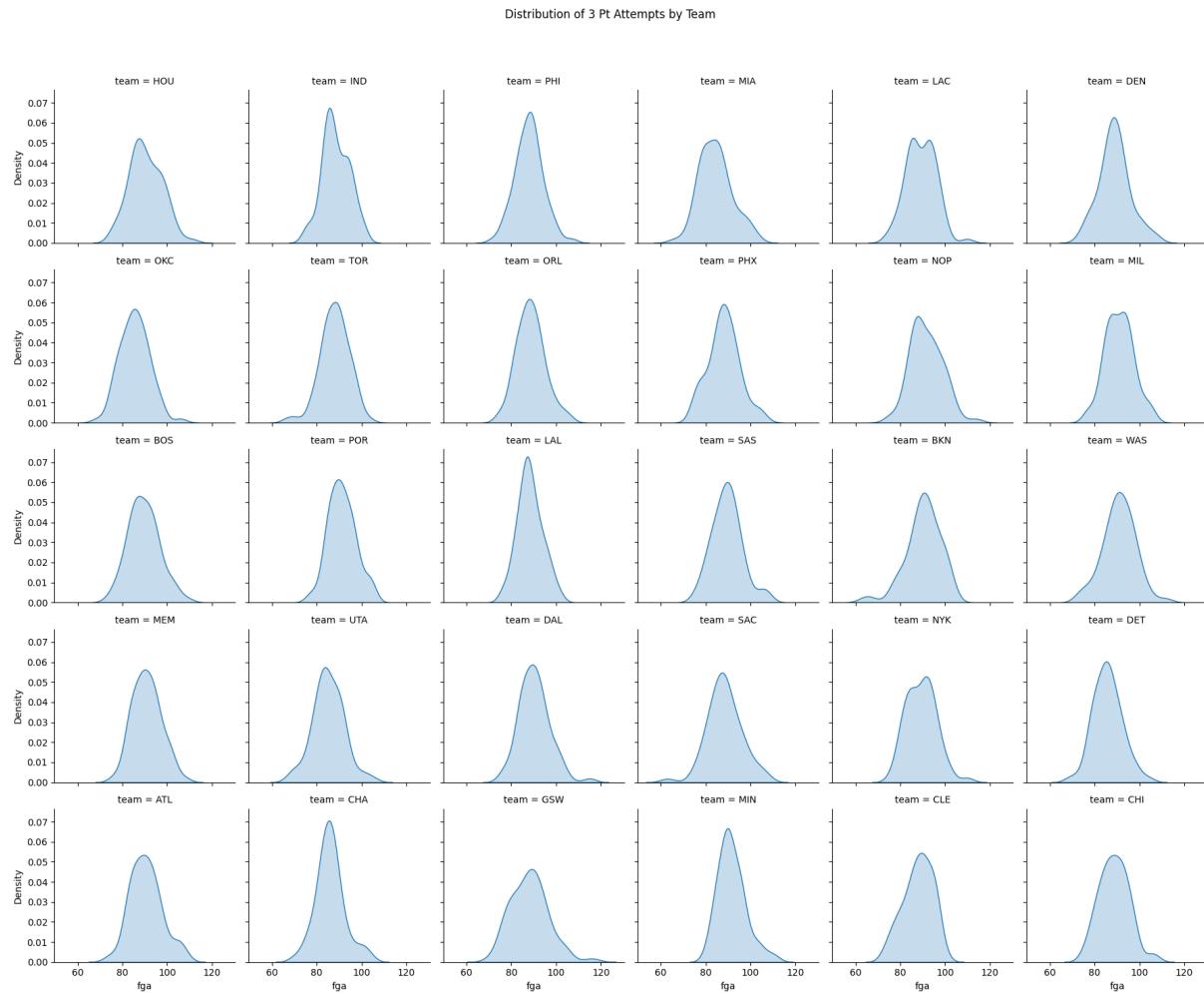
```
g = (sns.FacetGrid(dftg, col='wl', hue='wl')
      .map(sns.kdeplot, 'fga', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of 3 Pt Attempts by Win/Loss D')
```



**Figure 0.4:** Solution 6-1d

## 6.1e

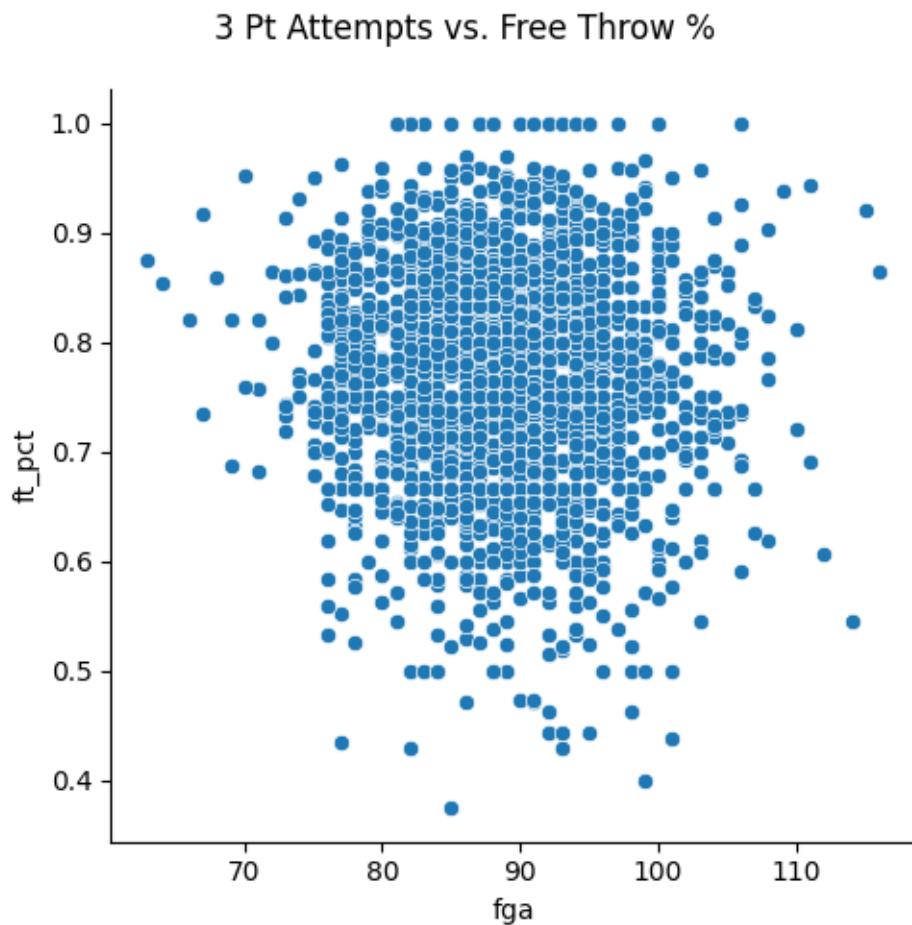
```
g = (sns.FacetGrid(dftg, col='team', col_wrap=6)
      .map(sns.kdeplot, 'fga', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of 3 Pt Attempts by Team')
```



**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')
```



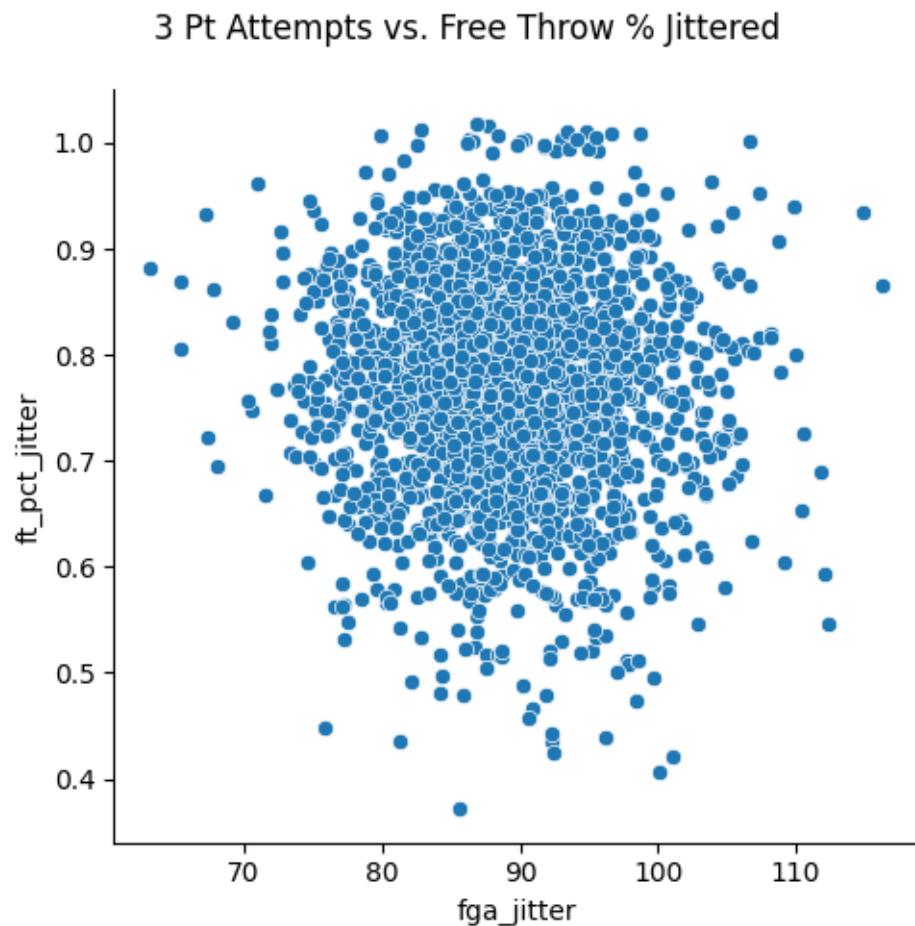
**Figure 0.6:** Solution 6-2a

## 6.2b

```
import random

dftg['fga_jitter'] = dftg['fga'].apply(lambda x: x + random.gauss(0, 1))
dftg['ft_pct_jitter'] = dftg['ft_pct'].apply(
    lambda x: x + random.gauss(0, 0.01))

g = sns.relplot(x='fga_jitter', y='ft_pct_jitter', data=dftg)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('3 Pt Attempts vs. Free Throw % Jittered')
```



**Figure 0.7:** Solution 6-2b

## 6.2c

```
In [1]: dftg[['fga', 'ft_pct']].corr()
Out[1]:
      fga    ft_pct
fga    1.000000 -0.039877
ft_pct -0.039877  1.000000
```

## 7. Modeling

### 7.1a

To apply `prob_of_make` to our yardage data.

```
In [2]:  
def prob_of_make(dist):  
    b0, b1, b2 = results.params  
    return (b0 + b1*dist + b2*(dist**2))  
  
In [3]: df['made_hat_alt'] = df['dist'].apply(prob_of_make)
```

The two should be the same. With `'made_hat_alt'` we're just doing manually what `results.predict(df)` is doing behind the scenes. Let's look at them:

```
In [4]: df[['made_hat', 'made_hat_alt']].head()  
Out[4]:  
   made_hat  made_hat_alt  
0  0.585380      0.585380  
1  0.339051      0.339051  
2  0.343195      0.343195  
3  0.339051      0.339051  
4  0.387104      0.387104
```

To check whether they're the same:

```
In [5]: (df['made_hat_alt'] == df['made_hat']).all()  
Out[5]: False
```

Given the first five rows are exactly the same, this is weird. Sometimes computers can't handle slight rounding errors. Let's check whether they're within some tiny difference of each other:

```
In [6]: import numpy as np  
  
In [7]: (np.abs(df['made_hat'] - df['made_hat_alt']) < .00000001).all()  
Out[7]: True
```

**7.1b**

```
In [5]:  
model_b = smf.ols(  
    formula='made ~ dist + dist_sq + C(value)', data=df)  
results_b = model_b.fit()  
results_b.summary2()  
--  
Out[5]:  
!!!!  
      Results: Ordinary least squares  
=====  
Model:                 OLS           Adj. R-squared:     0.054  
Dependent Variable: made          AIC:            23451.7042  
Date:                  2022-07-20 11:33 BIC:            23482.6388  
No. Observations:   16876        Log-Likelihood:   -11722.  
Df Model:                3           F-statistic:       323.5  
Df Residuals:           16872      Prob (F-statistic): 2.94e-204  
R-squared:               0.054      Scale:             0.23493  
-----  
              Coef.    Std.Err.      t    P>|t|    [0.025    0.975]  
-----  
Intercept      0.6364    0.0070   91.2645  0.0000    0.6228    0.6501  
C(value)[T.3]  0.1416    0.0154   9.1816  0.0000    0.1114    0.1719  
dist         -0.0218    0.0011  -20.0371  0.0000   -0.0240   -0.0197  
dist_sq        0.0002    0.0000    6.6065  0.0000    0.0002    0.0003  
-----  
Omnibus:        69580.869    Durbin-Watson:     2.003  
Prob(Omnibus):  0.000        Jarque-Bera (JB):  2223.683  
Skew:            0.166        Prob(JB):        0.000  
Kurtosis:        1.253        Condition No.: 1834  
=====  
* The condition number is large (2e+03). This might indicate  
strong multicollinearity or other numerical problems.  
!!!!
```

Looking at the results in `results_b.summary2()` we can see the coefficient on `C(value)[T.3]` is 0.1416, which is statistically significant. So a three point shot is *more* likely to go compared to a two pointer.

So the question is: besides distance to the basket (which we're controlling for) what difference between two and three point shots might explain why more three's go in? My guess is it's because better shooters probably take more threes.

### 7.1c

Wrapping a variable in `C()` turns it into a categorical variable (that's 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]: df['is3'] = df['value'] == 3

In [7]:
model_d = smf.ols(formula='made ~ dist + dist_sq + is3', data=df)
results_d = model_d.fit()
results_d.summary2()

Out[7]:
"""
      Results: Ordinary least squares
=====
Model:                 OLS           Adj. R-squared:     0.054
Dependent Variable: made          AIC:            23451.7042
Date:    2022-07-20 11:42  BIC:            23482.6388
No. Observations: 16876          Log-Likelihood:   -11722.
Df Model:                  3           F-statistic:      323.5
Df Residuals:             16872        Prob (F-statistic): 2.94e-204
R-squared:                0.054        Scale:            0.23493
-----
          Coef.    Std.Err.      t    P>|t|    [0.025    0.975]
-----
Intercept      0.6364    0.0070   91.2645  0.0000    0.6228    0.6501
is3[T.True]    0.1416    0.0154   9.1816  0.0000    0.1114    0.1719
dist       -0.0218    0.0011  -20.0371  0.0000   -0.0240   -0.0197
dist_sq      0.0002    0.0000    6.6065  0.0000    0.0002    0.0003
-----
Omnibus:        69580.869    Durbin-Watson:    2.003
Prob(Omnibus):  0.000          Jarque-Bera (JB): 2223.683
Skew:           0.166          Prob(JB):        0.000
Kurtosis:       1.253          Condition No.: 1834
=====
* The condition number is large (2e+03). This might indicate
strong multicollinearity or other numerical problems.
"""
```

Yes, the coefficients are the same. To see:

### 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 [1]:
sims_1k = Series([run_sim_get_pvalue() for _ in range(1000)])
sims_1k.mean()
-- 
Out[1]: 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:

```
In [1]: from math import log  
  
In [2]: p = 0.05  
  
In [3]: g_mean = 1/p  
  
In [4]: g_median = -1/log(1-p, 2)  
  
In [5]: g_mean, g_median  
Out[5]: (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 [6]: sim_time_till_sig_100.mean()  
Out[6]: 19.3  
  
In [7]: sim_time_till_sig_100.median()  
Out[7]: 15.0
```

**7.3a**

```
In [8]: dftg = pd.read_csv(path.join(DATA_DIR, 'team_games.csv'))  
  
In [9]: dftg['win'] = (dftg['wl'] == 'W').astype(int)  
  
In [10]::  
model_a = smf.logit(formula=  
        """  
        win ~ fg3_pct + oreb + dreb + stl + tov + blk  
        """", data=dftg)  
results_a = model_a.fit()  
results_a.summary2()  
  
Optimization terminated successfully.  
    Current function value: 0.487771  
    Iterations 6  
Out[10]:  
"""  
Results: Logit  
=====  
Model:           Logit          Pseudo R-squared: 0.296  
Dependent Variable: win          AIC:            2080.1984  
Date:           2022-07-20 11:47 BIC:            2119.8060  
No. Observations: 2118          Log-Likelihood: -1033.1  
Df Model:         6             LL-Null:         -1468.1  
Df Residuals:     2111          LLR p-value:  1.1633e-184  
Converged:        1.0000        Scale:           1.0000  
No. Iterations:   6.0000  
-----  
      Coef.    Std.Err.      z     P>|z|    [0.025    0.975]  
-----  
Intercept -13.9194    0.6849 -20.3243  0.0000  -15.2617 -12.5771  
fg3_pct    13.2392    0.7841  16.8842  0.0000   11.7023  14.7760  
oreb       0.0481    0.0154   3.1272  0.0018   0.0179   0.0782  
dreb       0.2360    0.0130  18.1795  0.0000   0.2105   0.2614  
stl        0.2405    0.0209  11.5270  0.0000   0.1996   0.2814  
tov        -0.1299   0.0144  -9.0301  0.0000  -0.1581  -0.1017  
blk        0.0986    0.0228   4.3255  0.0000   0.0539   0.1433  
=====  
"""
```

Marginal effects:

```
In [11]: margeff = results_a.get_margeff()
```

```
In [12]: margeff.summary()
```

```
Out[12]:
```

```
"""
```

```
    Logit Marginal Effects
```

```
=====
Dep. Variable:                      win
Method:                            dydx
At:                               overall
=====
```

	dy/dx	std err	z	P> z	[0.025
	0.975]				
fg3_pct	2.1328	0.092	23.070	0.000	1.952
2.314					
oreb	0.0077	0.002	3.153	0.002	0.003
0.013					
dreb	0.0380	0.001	26.742	0.000	0.035
0.041					
stl	0.0387	0.003	13.011	0.000	0.033
0.045					
tov	-0.0209	0.002	-9.691	0.000	-0.025
-0.017					
blk	0.0159	0.004	4.394	0.000	0.009
0.023					

```
"""
```

A steal helps a team more (0.04) than avoiding a turnover (-0.02). I'm not 100% sure why would be – my guess is because the stealing team gets gets the ball in transition and is probably more likely to score. Vs a turnover, where play can reset (e.g. if a team throws it out of bounds or gets a 24 second violation). More evidence for this interpretation: the effect of an additional defensive rebound is similar (where the team also gets the ball in transition) is similar to steal.

### 7.3b

Adding team fixed effects:

## Learn to Code with Basketball

---

```
In [13]:  
model_b = smf.logit(formula=  
    """  
        win ~ fg3_pct + oreb + dreb + stl + tov + blk + C(team)  
    )  
    """", data=dftg)  
results_b = model_b.fit()  
results_b.summary2()  
  
--  
Optimization terminated successfully.  
    Current function value: 0.458665  
    Iterations 7  
Out[13]:  
"""  
Results: Logit  
=====  
Model:          Logit          Pseudo R-squared: 0.338  
Dependent Variable: win          AIC:            2014.9047  
Date:          2022-07-20 12:03 BIC:            2218.6009  
No. Observations: 2118          Log-Likelihood: -971.45  
Df Model:       35             LL-Null:         -1468.1  
Df Residuals:   2082           LLR p-value:  7.6110e-186  
Converged:      1.0000          Scale:           1.0000  
No. Iterations: 7.0000  
-----  
          Coef.    Std.Err.     z    P>|z|    [0.025    0.975]  
-----  
Intercept     -15.0468   0.8039  -18.7182  0.0000  -16.6223  -13.4712  
C(team)[T.BKN]  0.3168   0.4310   0.7352  0.4622  -0.5278   1.1615  
C(team)[T.BOS]  0.8441   0.4503   1.8746  0.0608  -0.0384   1.7266  
C(team)[T.CHA]  0.7371   0.4517   1.6317  0.1027  -0.1483   1.6224  
C(team)[T.CHI] -0.0254   0.4568  -0.0556  0.9557  -0.9207   0.8699  
...  
C(team)[T.SAC]  0.4158   0.4346   0.9567  0.3387  -0.4360   1.2676  
C(team)[T.SAS] -0.3492   0.4394  -0.7947  0.4268  -1.2105   0.5120  
C(team)[T.TOR]  1.0505   0.4444   2.3636  0.0181  0.1794   1.9216  
C(team)[T.UTA]  1.1776   0.4328   2.7210  0.0065  0.3294   2.0257  
C(team)[T.WAS]  0.0074   0.4347   0.0171  0.9864  -0.8445   0.8593  
fg3_pct        13.8315  0.8256   16.7526  0.0000  12.2133  15.4497  
oreb           0.0651   0.0165   3.9317  0.0001  0.0326   0.0975  
dreb           0.2411   0.0140   17.1767  0.0000  0.2136   0.2686  
stl            0.2514   0.0227   11.0864  0.0000  0.2070   0.2959  
tov            -0.1291  0.0154  -8.3904  0.0000  -0.1592  -0.0989  
blk            0.1060   0.0247   4.2913  0.0000  0.0576   0.1544  
=====  
"""
```

Because we're controlling for three point percentage, rebounds, steals, turnovers and blocks, I'd expect teams with high coefficients to be good in areas we're *not* controlling for (e.g. free throws, two

point field goals, three point attempts) to have high coefficients.

#### 7.4a

To run the model:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, cross_val_score

xvars = ['min', 'pts', 'fgm', 'fga', 'fg_pct', 'fg3m', 'fg3a', 'fg3_pct',
         'ftm', 'fta', 'ft_pct', 'oreb', 'dreb', 'reb', 'ast', 'stl', 'blk',
         'tov',
         'pf', 'plus_minus', 'bubble', 'win']

yvar = 'team'

model = RandomForestClassifier(n_estimators=100)
```

#### 7.4b

I'm specifically thinking of team\_id, which technically is numeric and so could be used in a Random Forest, but would predict team perfectly and make the whole model pointless. I also dropped game id and our sample dummy.

#### 7.4c

```
In [14]: scores = cross_val_score(model, dftg[xvars], dftg[yvar], cv=10)
scores.mean()

In [15]: scores.mean()
Out[15]: 0.1251274255566485

In [16]: scores.min()
Out[16]: 0.08018867924528301

In [17]: scores.max()
Out[17]: 0.15566037735849056
```

#### 7.4d

If we were guessing at random we'd expect to get 1/30 right.

```
In [18]: 1/30
Out[18]: 0.033333333333333333
```

So our model does about 4x as well as guessing a team at random.

## 7.4e

```
In [19]: model.fit(dftg[xvars], dftg[yvar]) # fit on entire dataset
Out[19]: RandomForestClassifier()

In [20]:
Series(model.feature_importances_, xvars).sort_values(ascending=False)

Out[20]:
fg3a          0.062110
ft_pct        0.057719
plus_minus   0.054185
fg3_pct       0.053945
fg_pct         0.053194
ast            0.053130
fga             0.051420
pts             0.051077
pf              0.050525
dreb            0.050397
fta             0.049641
reb             0.048667
tov             0.048475
stl             0.046616
ftm             0.045859
fgm             0.045255
blk             0.044584
oreb            0.044491
fg3m            0.042945
min            0.031048
win             0.007795
bubble          0.006921
```