NATHAN BRAUN

LEARN TO CODE with FOOTBALL

Learn Python, webscraping, machine learning *and more*, all applied to soccer data

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Learn to Code with Football

v0.1.1

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Copyright Notice

Copyright © 2023 by Nathan Braun. All rights reserved.

By viewing this digital book, you agree that under no circumstances shall you use this book or any portion of it for anything but your own personal use and reference. To be clear: you shall not copy, re-sell, sublicense, rent out, share or otherwise distribute this book, or any other Learn to Code with Soccer/Football digital product, whether modified or not, to any third party.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Contents

Prerequisites: Tooling **1**
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

1. Introduction **8**
The Purpose of Data Analysis . . . 8
What is Data? . . . 8
Example Datasets . . . 10
Shot Data . . . 10
Player/Game Data . . . 11
Player Data . . . 12
Match Data . . . 12
What is Analysis? . . . 13
Types of Data Analysis . . . 14
Summary Statistics . . . 14
Modeling . . . 15
High Level Data Analysis Process . . . 17
1. Collecting Data . . . 17
2. Storing Data . . . 17
3. Loading Data . . . 17
4. Manipulating Data . . . 17
5. Analyzing Data for Insights . . . 18
Connecting the High Level Analysis Process to the Rest of the Book . . . 18
End of Chapter Exercises . . . 19

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

2. Python **22**
Introduction to Python Programming . . . 22
How to Read This Chapter . . . 22
Important Parts of the Python Standard Library . . . 23
Comments . . . 23
Variables . . . 23
Types . . . 25
Interlude: How to Figure Things Out in Python . . . 26
Bools . . . 29
if statements . . . 30
Container Types . . . 30
Unpacking . . . 32
Loops . . . 33
Comprehensions . . . 35
Functions . . . 38
Libraries are Functions and Types . . . 44
os Library and path . . . 44
End of Chapter Exercises . . . 46

3. Pandas **49**
Introduction to Pandas . . . 49
Types and Functions . . . 49
Things You Can Do with DataFrames . . . 50
How to Read This Chapter . . . 50
Part 1. DataFrame Basics . . . 51
Importing Pandas . . . 51
Loading Data . . . 52
DataFrame Methods and Attributes . . . 52
Working with Subsets of Columns . . . 53
Indexing . . . 55
Outputting Data . . . 58
Exercises . . . 59
Part 2. Things You Can Do With DataFrames . . . 60
Introduction . . . 60
1. Modify or Create New Columns of Data . . . 61
Creating or Modifying Columns - Same Thing . . . 61
Math and Number Columns . . . 62
String Columns . . . 63

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Boolean Columns . . . 64
Applying Functions to Columns . . . 65
Dropping Columns . . . 66
Renaming Columns . . . 66
Missing Data in Columns . . . 67
Changing Column Types . . . 68
Review . . . 70
Exercises . . . 71
2. Use Built-In Pandas Functions That Work on DataFrames . . . 72
Summary Statistic Functions . . . 72
Axis . . . 73
Summary Functions on Boolean Columns . . . 74
Other Misc Built-in Summary Functions . . . 76
Review . . . 78
Exercises . . . 79
3. Filter Observations . . . 80
loc . . . 80
Combining Filtering with Changing Columns . . . 84
The `query` Method is an Alternative Way to Filter . . . 85
Review . . . 86
Exercises . . . 88
4. Change Granularity . . . 89
Ways of Changing Granularity . . . 89
Grouping . . . 89
A Note on Multilevel Indexing . . . 92
Stacking and Unstacking Data . . . 92
Review . . . 94
5. Combining Two or More DataFrames . . . 96
Merging . . . 96
Merge Question 1. What columns are you joining on? . . . 96
Merging is Precise . . . 97
Merge Question 2. Are you doing a 1:1, 1:many (or many:1), or many:many join? . . . 98
Merge Question 3. What are you doing with unmatched observations? . . . 100
More on pd.merge . . . 102
`pd.merge()` Resets the Index . . . 104
`pd.concat()` . . . 104
Combining DataFrames Vertically . . . 106
Review . . . 107

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

4. SQL **110**
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 . . . 124
SQL Example — LEFT JOIN, UNION, Subqueries . . . 125
End of Chapter Exercises . . . 129

5. Web Scraping and APIs **130**
Introduction to Web Scraping and APIs . . . 130
Web Scraping . . . 130
HTML and CSS . . . 130
BeautifulSoup . . . 133
Simple vs Nested Tags . . . 136
World Football - Web Scraping Example . . . 140
APIs . . . 149
Two Types of APIs . . . 149
Web APIs . . . 149
HTTP . . . 150
JSON . . . 150
Benefits of APIs . . . 151
Working with APIs - General Process . . . 152
Fantasy Premier League API . . . 153

6. Data Analysis and Visualization **165**
Introduction . . . 165
Distributions . . . 166
Summary Stats . . . 169
Density Plots in Python . . . 173

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Relationships Between Variables . . . 182
Scatter Plots with Python . . . 182
Correlation . . . 190
Line Plots with Python . . . 193
Plot Options . . . 198
Shot Charts . . . 203
Shot Charts As Seaborn Scatter Plots . . . 203
kwargs . . . 207
Contour Plots . . . 211
End of Chapter Exercises . . . 216

7. Modeling **217**
Introduction to Modeling . . . 217
The Simplest Model . . . 217
Linear regression . . . 218
Statistical Significance . . . 223
Regressions hold things constant . . . 226
Fixed Effects . . . 229
Squaring Variables . . . 232
Logging Variables . . . 233
Interactions . . . 235
Logistic Regression . . . 236
Random Forest . . . 238
Classification and Regression Trees . . . 238
Random Forests are a Bunch of Trees . . . 239
Using a Trained Random Forest to Generate Predictions . . . 239
Random Forest Example in Scikit-Learn . . . 240
Random Forest Regressions . . . 245
End of Chapter Exercises . . . 247

8. Intermediate Coding and Next Steps: High Level Strategies **249**
Gall's Law . . . 249
Get Quick Feedback . . . 250
Use Functions . . . 250
DRY: Don't Repeat Yourself . . . 251
Functions Help You Think Less . . . 251
Attitude . . . 251
Review . . . 253

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

9. Conclusion **254**

Appendix A: Places to Get Data **255**
Detailed Academic Event Data . . . 255
Datahub's list . . . 255
Premier League Fantasy API . . . 256
Other Options . . . 256
Kaggle.com . . . 256
Google Dataset Search . . . 256

Appendix B: Anki **257**
Remembering What You Learn . . . 257
Installing Anki . . . 258
Using Anki with this Book . . . 259

Appendix C: Answers to End of Chapter Exercises **261**
1. Introduction . . . 261
2. Python . . . 263
3.0 Pandas Basics . . . 268
3.1 Columns . . . 270
3.2 Built-in Functions . . . 274
3.3 Filtering . . . 276
3.4 Granularity . . . 278
3.5 Combining DataFrames . . . 282
4. SQL . . . 284
6. Summary and Data Visualization . . . 285
7. Modeling . . . 290

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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-soccer-files/releases

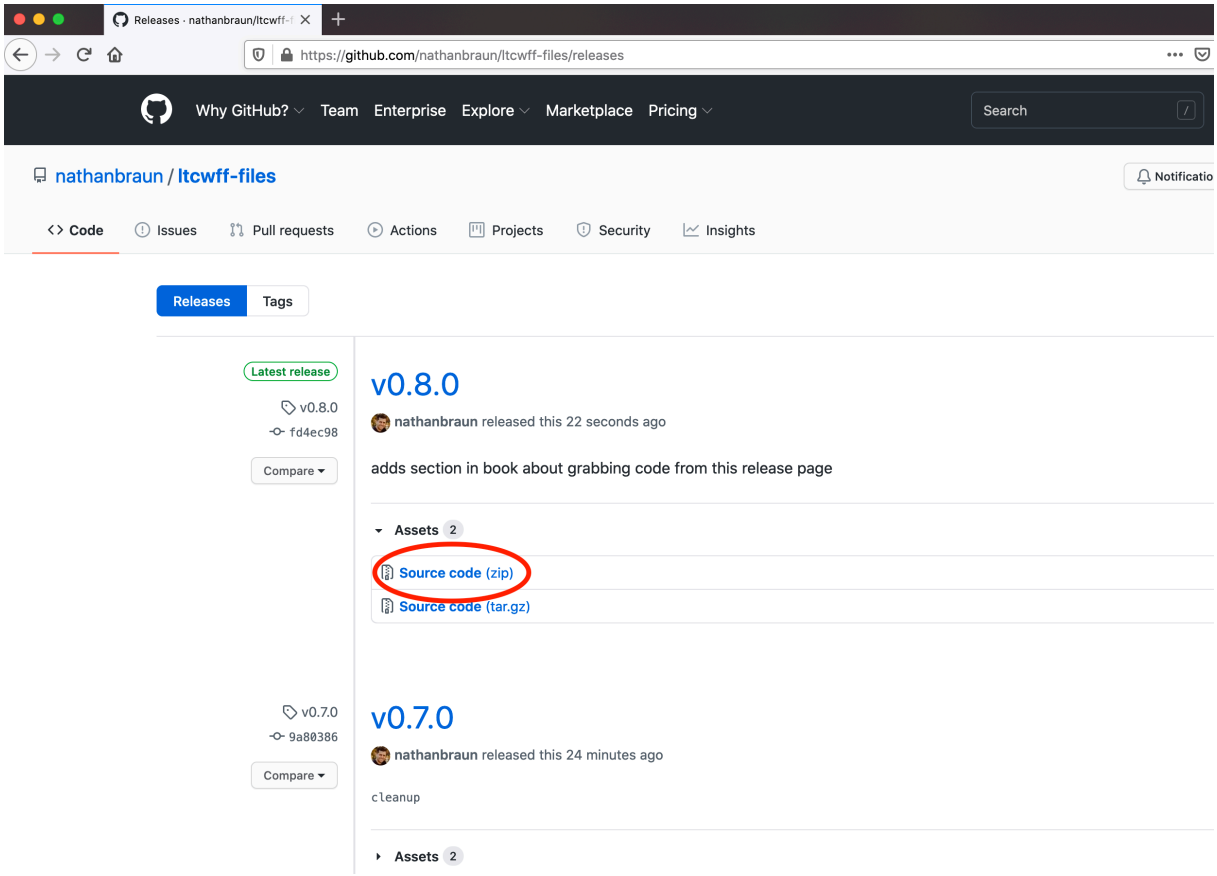

Figure 0.1: LTCWFF Files on GitHub

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

If you're not familiar with Git or GitHub, no problem. Just click the `Source code` link under the latest release to download the files. This will download a file called `code-soccer-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-soccer-files`, which you can do too) you'll see four sub-directories: `code`, `data`, `anki`, `solutions-to-excercises`.

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-soccer-files`

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

`C:\Users\nathanbraun\code-soccer-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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Figure 0.2: Python 3.x on the Anaconda site

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

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

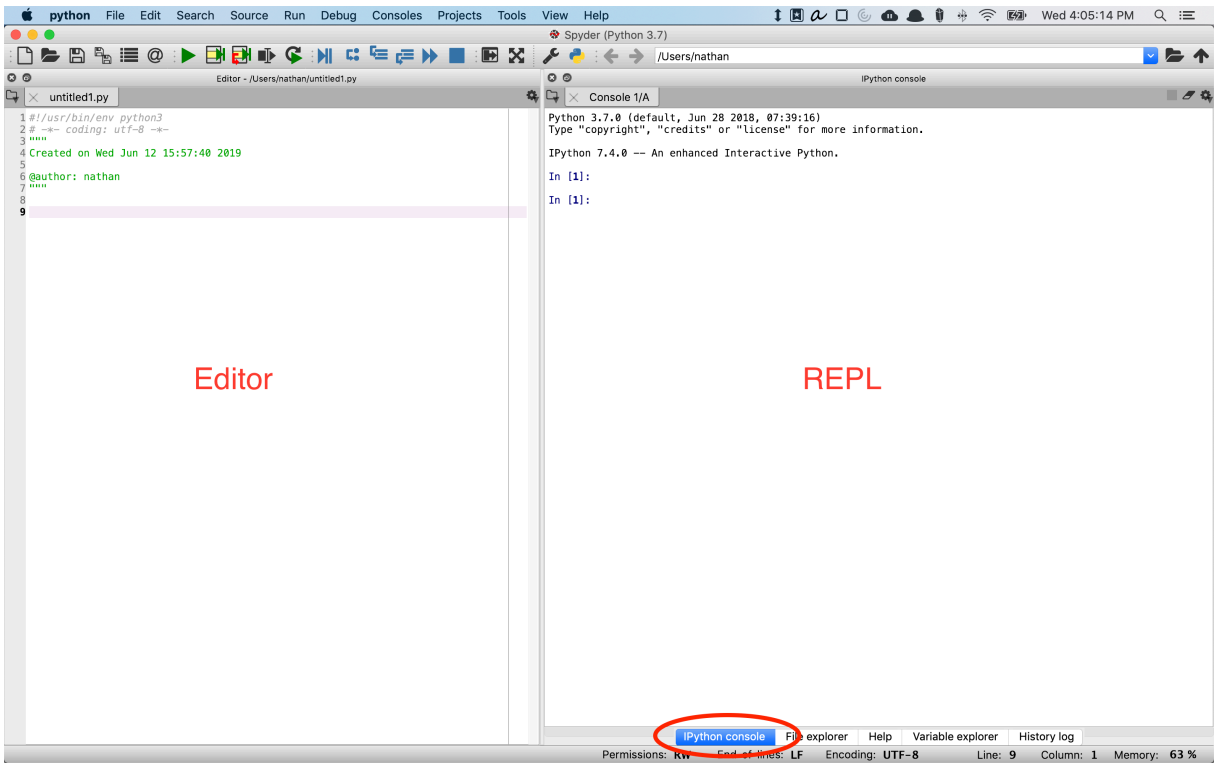

Figure 0.3: Editor and REPL in Spyder

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

And then later:

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

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

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

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

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

Try:

```
In [2]: x = 1

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

ZeroDivisionError: division by zero
```

Since dividing by 0 is against the laws of math[2], Python won't let you do it and will throw (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.

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

1. Introduction

The Purpose of Data Analysis

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

- I want to win my premier league fantasy match, who do I play this week?
- I'm a scout for Bayern, which midfielder do I recommend we sign?
- I'm a mad scientist, how many more championships would Manchester United have won had Cristiano Ronaldo not left for Real Madrid?

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 football game, say the 2018 World Cup Final - France vs Croatia. What would a collection of structured information about it look like?

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

Now **information** — what information might we have about each shot in this collection? Maybe: minute, distance, player (and team) shooting, which foot the player shot it with, whether it went in, etc.

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 (player, minute, etc).

This is an efficient, organized way of presenting information. When we want to know, "who had the first shot in the second half and which foot did they use and did it go in?", we can find the right row and columns, and say "Oh, Antoine Griezmann with his left foot, and no".

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

shot	min	period	name	team	foot	goal
1	20	1H	D. Vida	Croatia	head/body	False
2	23	1H	I. ćRakiti	Croatia	left	False
3	27	1H	I. šćPerii	Croatia	left	True
4	39	1H	A. ćRebi	Croatia	left	False
5	42	1H	I. šćPerii	Croatia	head/body	False
6	42	1H	D. Lovren	Croatia	right	False
7	45	1H	D. Vida	Croatia	head/body	False
8	49	2H	A. Griezmann	France	left	False
9	50	2H	A. ćRebi	Croatia	left	False
10	53	2H	S. Vrsaljko	Croatia	right	False
11	55	2H	K. Mbappé	France	right	False
12	61	2H	P. Pogba	France	right	False
13	62	2H	P. Pogba	France	left	True
14	64	2H	O. Giroud	France	left	False
15	68	2H	K. Mbappé	France	right	True
16	67	2H	A. ćRebi	Croatia	right	False
17	71	2H	M. žćManduki	Croatia	right	True
18	78	2H	S. Vrsaljko	Croatia	right	False
19	80	2H	I. ćRakiti	Croatia	left	False
20	89	2H	N. Fekir	France	left	False
21	91	2H	I. ćRakiti	Croatia	right	False

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 France-Croatia is *flattened* out into one big table).

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

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
shot,min,period,name,team,foot,goal
0,20,1H,D. Vida,Croatia,head/body,False
1,23,1H,I. ćRakiti,Croatia,left,False
2,27,1H,I. šćPerii,Croatia,left,True
3,39,1H,A. ćRebi,Croatia,left,False
4,42,1H,I. šćPerii,Croatia,head/body,False
5,42,1H,D. Lovren,Croatia,right,False
6,45,1H,D. Vida,Croatia,head/body,False
7,49,2H,A. Griezmann,France,left,False
8,50,2H,A. ćRebi,Croatia,left,False
9,53,2H,S. Vrsaljko,Croatia,right,False
10,55,2H,K. Mbappé,France,right,False
11,61,2H,P. Pogba,France,right,False
12,62,2H,P. Pogba,France,left,True
13,64,2H,O. Giroud,France,left,False
14,68,2H,K. Mbappé,France,right,True
15,67,2H,A. ćRebi,Croatia,right,False
16,71,2H,M. žćManduki,Croatia,right,True
17,78,2H,S. Vrsaljko,Croatia,right,False
18,80,2H,I. ćRakiti,Croatia,left,False
19,89,2H,N. Fekir,France,left,False
20,91,2H,I. ćRakiti,Croatia,right,False
```

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. You can open and write csv's in Excel (or whichever spreadsheet program you use) and they'll be in the familiar spreadsheet format. That's one of the main benefits to storing data as 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 the 2018 World Cup. They're 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

It includes: the player shooting, time in the match, and shot location (coordinates on the pitch in x, y form) and distance, as well as information on the type of shot (bodypart, whether it went in, roughly where on net it was, etc).

This data is in the file `./data/shot.csv`.

Player/Game Data

The second dataset this book comes with is at the player-match level. It's in the file `player_match.csv`. The rows (and some of the columns, they don't all fit here) look like this:

```
match_id            name     team       opp  min  shot  goal  started
 2058016          D. Rose  England   Belgium   46     0     0     True
 2058004  Sergio Busquets    Spain    Russia  120     1     0     True
 2057986      O. Toivonen   Sweden   Germany   78     1     1     True
 2058011    Douglas Costa   Brazil   Belgium   32     4     0    False
 2057984       M. Hummels  Germany    Mexico   90     1     0     True
 2057994    L. Dendoncker  Belgium   England   90     0     0     True
 2058009         A. Young  England  Colombia  102     0     0     True
 2057996    G. Krychowiak   Poland   Senegal   90     1     1     True
 2057987       Se-Jong Ju    Korea    Mexico   63     0     0     True
 2057975       L. Balogun  Nigeria   Iceland   90     1     0     True
```

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

This data is a *collection* of player-match combinations. Each row represents one player's statistics for one match (Danny Rose/vs Belgium). If we had 100 players, and they each played four matches, then our dataset would be `100*4=400` rows long.

The columns (variables) are *information*. In the fourth row, each column tells us something about how Douglas Costa did in his game vs Belgium. The `min` column shows us he played 32 minutes, the `shot` column indicates he had 4 shots, and the `goal` column tells us none of them (0) went in.

If we want to look at another player-game (Danny Rose, England vs Belgium or Costa, some other game), we look at a different row.

Notice how our columns can be different types of information, like text (`name`) or numbers (`min`) or semi-hybrid, technically-numbers but we would never do any math with them (`match_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 `year`. It's always `2018` (this data is from the 2018 World Cup). Does this matter? No. We're just asking: "for this particular player/game result, what year was it?" It just happens that for this data the answer is the same for every row.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Player Data

We can also organize our 2018 cup data at the *player* level. This dataset is in `players.csv` and looks like this:

```
    player_name  pos   foot            team  height  weight  birth_date
M. Plattenhardt  DEF   left         Germany     181      76    19920126
       Yong Lee  DEF  right  Korea Republic     180      76    19861224
    L. Torreira  MID  right         Uruguay     168      64    19960211
        I. Sarr  MID  right         Senegal     178      68    19980225
       K. Danso  DEF  right         Austria     190      89    19980919
```

Match Data

Same thing with matches. The file `matches.csv` contains data on every match in the 2018 World Cup:

```
                          label    group        date  venue
26  Russia - Saudi Arabia, 5 - 0  Group A  2018-06-14  Olimpiyskiy
60        Egypt - Uruguay, 0 - 1  Group A  2018-06-15  Centralnyj
59         Morocco - Iran, 0 - 1  Group B  2018-06-15  Krestovskyi
57       Portugal - Spain, 3 - 3  Group B  2018-06-15  Olimpiyskiy
55    Argentina - Iceland, 1 - 1  Group D  2018-06-16  Otkrytiye
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

What is Analysis?

How many footballs are in the following picture?

Figure 0.1: Few soccerballs

Pretty easy question right? What about this one?

Figure 0.2: Many soccerballs

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 shot data this book comes with, you'll notice it's 1366 rows and 24 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 unluckiest? Of course not.

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

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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: mean, (average), median, mode or expected goals (xG).
2. The second type of analysis is building *models* to understand relationships between data.

Summary Statistics

Summary statistics can be complex (expected goals) or more basic (shooting percentage or games missed due to injury), but all of them involve going from raw data to some more useful number.

Stats don't need to be fancy. Take our player data:

```
           name  pos         team  height  weight
      A. NDiaye  MID      Senegal     187      82
T. Alderweireld  DEF      Belgium     187      91
  J. Vertonghen  DEF      Belgium     189      88
     C. Eriksen  MID      Denmark     180      76
     D. Mertens  FWD      Belgium     169      61
    O. Toivonen  FWD       Sweden     192      78
   K. El Ahmadi  MID      Morocco     179      78
    J. Guidetti  FWD       Sweden     185      79
     N. Amrabat  FWD      Morocco     178      77
      N. Chadli  MID      Belgium     187      80
     L. Balogun  DEF      Nigeria     190      81
  D. Tadi\u0107  MID       Serbia     181      76
 L. Sch\u00f6ne  MID      Denmark     177      78
     D. Zakaria  MID  Switzerland     191      80
       A. Iwobi  MID      Nigeria     180      75
     M. Yoshida  DEF        Japan     189      78
      H. Mendyl  DEF      Morocco     179      73
      T. Ebuehi  DEF      Nigeria     187      72
```

What "statistic" might we use to understand the physical characteristics of World Cup football players?

How about, the average?

```
height    182.702041
weight     76.801361
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

The main goal of these single number summary statistics is usually to *summarize and make sense of some past performance* (or existing data), e.g. when deciding who to sign to our fantasy team or arguing online about who contributed the most to Manchester City's championship win.

Stats vary in scope. Some, like WhoScored's ratings, try to be all encompassing, while others get at a particular facet of a player's performance. For example, we might measure "power" by looking at a player's longest shot on goal, or "sportsmanship" by looking at the number of yellow cards.

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. Doing this well can give teams 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[1].

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.

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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?

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 2022 World Cup. I have data on all the qualifying games, and I want to use that to predict each team's number of goals scored during the tournament itself.

Modeling is about relationships. In this case the relationship is between data I have *now* (the number of goals scored in qualifying games in the run up to the 2022 World Cup) and events that will happen in the *future* (number of goals scored during the actual 2022 tournament).

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 on in the Summer of 2022, I have data on the last World Cup in 2018. And I could build a model:

```
goals scored in 2018 World Cup = model(goals scored in 2018 qualifying
games)
```

Training (or *fitting*) this model is the process of using that known/existing/already happened data to find a relationship between the input variables (2018 qualifying games goals scored) and the output variable (goals scored in 2018 World Cup).

Once I establish that relationship, I can feed it new inputs — 22 goals scored in qualifying games — and transform it using my relationship to get back a prediction for the tournament.

The inputs I feed my model might be from past events that have already happened. Often this is done to evaluate model performance. For example, I could put in France's 2018 qualifying stats to see what the model would have predicted for the World Cup in 2018, even though I already know how they did (hopefully it's close[2]).

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 2022 World Cup — I can put in the number of goals Qatar scored in qualifying games and get back a projection for the real tournament.

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. Football data is no exception. Sometimes called *munging*, this means getting your raw data in the right format for analysis.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

End of Chapter Exercises

1.1

Name the granularity for the following, hypothetical datasets:

a)

```
match_id period          name  ngoals  nshots  ave_shot_distance
 2057972     1H  H. Magnússon       0       1          14.282610
 2057963     1H          Isco       0       2          26.838446
 2058010     2H     O. Giroud       0       1          20.313895
 2057974     2H   N. Otamendi       0       1          10.978094
 2058015     E1   A. ćKramari       0       2          16.884146
 2057985     2H   V. Claesson       0       1          21.303812
 2057994     2H    A. Januzaj       1       2          13.504836
 2057983     1H    V. Behrami       0       1          28.585756
 2058004     E2       Rodrigo       0       3          17.726751
 2057960     2H    Iago Aspas       0       1          23.226243
```

b)

```
team_id            team grouping
  14855  Korea Republic        F
  14358          Russia        A
   7047          Sweden        F
  16276         Tunisia        G
   6380          Brazil        E
   3148         Germany        F
  16216         Morocco        B
  12430        Colombia        H
  17929          Panama        G
  12274       Argentina        D
```

c)

```
round_id           start_date  goals  ngames
 4165363  2018-06-14 15:00:00    115      45
 4165364  2018-06-30 14:00:00     24       8
 4165365  2018-07-06 14:00:00     11       4
 4165366  2018-07-10 18:00:00      4       2
 4165367  2018-07-14 14:00:00      2       1
 4165368  2018-07-15 15:00:00      6       1
```

d)

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
     team    win  nmatches  shots  opp_shots  goals  oppgoals
  Belgium  False         2     16         42      1         2
  Belgium   True         5     82         48     14         4
   Brazil  False         2     43         13      2         2
   Brazil   True         3     52         25      6         0
 Colombia  False         2     20         25      5         7
 Colombia   True         2     15         11      4         0
  Croatia  False         1     14          7      2         3
  Croatia   True         6     89         73     18        10
```

e)

```
     team  pos  ave_height  ave_weight
Argentina  DEF  181.250000   76.375000
Argentina  FWD  174.200000   71.600000
Argentina  GKP  188.000000   83.500000
Argentina  MID  177.400000   74.000000
  Austria  DEF  187.166667   83.500000
  Austria  FWD  190.666667   83.333333
  Austria  GKP  189.000000   80.500000
  Austria  MID  181.000000   76.250000
  Belgium  DEF  188.142857   84.857143
  Belgium  FWD  178.500000   74.500000
```

1.2

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

a) Input variables.

b) Output variable.

c) What level of granularity is this model at?

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

1.3

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

f) plotting your data
g) getting data from an API
h) pandas
i) taking the mean of your data
j) combining multiple data sources

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

At their core, variables[1] 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]: goals_scored = 2
```

Another, less common, word for assignment is **binding**, as in `goals_scored` is bound to the number 2.

Now, whenever you use `goals_scored` in your code, the program automatically substitutes it with `2` instead.

```
In [2]: goals_scored
Out[2]: 2

In [3]: 3*goals_scored
Out[3]: 6
```

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 `goals_scored` is `2`, the assignment statement itself, `goals_scored = 2`, 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, `goals_scored` has a value of `2` (from the code we just ran above). So the right hand side, `goals_scored + 1` is evaluated first (2 + 1 = 3), and *then* the result gets (re)assigned to `goals_scored`, overwriting the `2` it held previously.

[1] 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [4]: goals_scored = goals_scored + 1

In [5]: goals_scored
Out[5]: 3
```

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]: keeper_saves = 12  # int
In [7]: ball_speed_kmh = 96.5  # 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 the word *It's*), in which case a string with `'It's a goal'` would give an error.

```
In [8]: starting_fwd = 'Lionel Messi'
In [9]: description = "It's a goal"
```

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

```
In [10]: type(starting_fwd)
Out[10]: str

In [11]: type(keeper_saves)
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 of them. The easiest way to do that is via **f-strings**.

```
In [12]: player_description = f'{description} by {starting_fwd}!'

In [13]: player_description
Out[13]: "It's a goal by Lionel Messi!"
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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]: 'gooaaaaal'.upper()
Out[14]: 'GOOAAAAAL'
```

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]: 'Christiano Ronaldo, Man U'.replace('Man U', 'Real Madrid')
Out[15]: 'Christiano Ronaldo, Real Madrid'
```

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 `'lionel messi'` 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).

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

Note: tab completion on a string directly like this doesn't always work in Spyder. If it's not working for you, assign `'lionel messi'` to a variable and tab complete on that. Like this[2]:

[2]The upside of this Spyder autocomplete issue is you can learn about the programming convention "foo". When dealing

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [16]: foo = 'lionel messi'
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 `'lionel messi'.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]: 'lionel messi'.capitalize()
Out[17]: 'Lionel messi'
```

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]: '  lionel messi'
```

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

```
In [19]: 'lionel messi'
```

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

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

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

> "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]: '  lionel messi'.lstrip()
Out[20]: 'lionel messi'
```

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 es-

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

> pecially 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_goals = 2
In [22]: team2_goals = 1

# these are all bools:
In [23]: team1_won = team1_goals > team2_goals

In [24]: team2_won = team1_goals < team2_goals

In [25]: teams_tied = team1_goals == team2_goals

In [26]: teams_did_not_tie = team1_goals != team2_goals

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_goals = team2_goals)
...
SyntaxError: invalid syntax
```

So `team1_goals == team2_goals` will be `True` if those numbers are the same, `False` if not.

The reverse is `!=`, which means *not equal*. The expression `team1_goals != team2_goals` is `True` if the values are different, `False` if they're the same.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

```
In [30]: shootout = (team1_goals > 3) and (team2_goals > 3)

In [31]: at_least_one_good_team = (team1_goals > 3) or (team2_goals > 3)

In [32]: you_guys_are_bad = not ((team1_goals > 1) or (team2_goals > 1))

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]: 'Nice job team 1!'
```

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**.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Lists

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

```
In [36]: roster_list = ['ruben dias', 'gabriel jesus', 'riyad mahrez']
```

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]: 'ruben dias'

In [38]: roster_list[0:2]
Out[38]: ['ruben dias', 'gabriel jesus']
```

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]: ['gabriel jesus', 'riyad mahrez']
```

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 = {'CB': 'ruben dias',
               'CF': 'gabriel jesus',
               'RW': 'riyad mahrez'}
```

You can access items in a dict like this:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [41]: roster_dict['CB']
Out[41]: 'ruben dias'
```

And add new things to dicts like this:

```
In [42]: roster_dict['LW'] = 'raheem sterling'

In [43]: roster_dict
Out[43]:
{'CB': 'ruben dias',
 'CF': 'gabriel jesus',
 'RW': 'riyad mahrez',
 'LW': 'raheem sterling'}
```

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 = 'RW'

In [45]: roster_dict[pos]
Out[45]: 'riyad mahrez'
```

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

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 `'raheem sterling'` 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]: cb, dm = ['ruben dias', 'fernandinho']
```

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

```
In [47]: cb = 'ruben dias'

In [48]: dm = 'fernandinho'
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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]: cb, dm = ['ruben dias', 'fernandinho', 'riyad mahrez']
...
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 `cb` and `dm` 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 = ['ruben dias', 'gabriel jesus', 'riyad mahrez']
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 `'ruben dias'`, the second `'gabriel jesus'`, 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]: ['Ruben Dias', 'Gabriel Jesus', 'Riyad Mahrez']
```

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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [51]:
for x in roster_dict:
    print(f"position: {x}")
--
position: CB
position: CF
position: RW
position: LW
```

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: CB
player: ruben dias
position: CF
player: gabriel jesus
position: RW
player: riyad mahrez
position: LW
player: raheem sterling
```

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: CB
player: ruben dias
position: CF
player: gabriel jesus
position: RW
player: riyad mahrez
position: LW
player: raheem sterling
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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. Our first `for` loop example, where we wanted to take our list of lowercase players and make a list where they're all properly formatted, is a great candidate.

The list comprehension way of doing that would be:

```
In [54]: roster_list
Out[54]: ['ruben dias', 'gabriel jesus', 'riyad mahrez']

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

In [56]: roster_list_proper
Out[56]: ['Ruben Dias', 'Gabriel Jesus', 'Riyad Mahrez']
```

All list comprehensions take the form [*a* for *b* in *c*] where *c* is the list you're iterating over (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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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]: ['Ruben Dias', 'Gabriel Jesus']
```

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]: ['dias', 'jesus', 'mahrez']
```

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 = 'ruben dias'

In [63]: full_name.split(' ')
Out[63]: ['ruben', 'dias']

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

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`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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_r_only = [
    x for x in roster_list if x.startswith('r')]

In [66]: roster_r_only
Out[66]: ['ruben dias', 'riyad mahrez']
```

Updating our notation, a comprehension technically has the form [*a* for *b* in *c* if *d*], where *if d* is optional.

Above, *d* is `x.startswith('r')`. 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]: 'ruben dias'.startswith('r')
Out[67]: True

In [68]: 'gabriel jesus'.startswith('r')
Out[68]: False

In [69]: 'riyad mahrez'.startswith('r')
Out[69]: True
```

Interestingly, in this comprehension the *a* in our [*a* for *b* in *c* 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 `'r'`.

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

```
In [70]:
roster_r_only_title = [
    x.title() for x in roster_list if x.startswith('r')]
--

In [71]: roster_r_only_title
Out[71]: ['Ruben Dias', 'Riyad Mahrez']
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [72]:
salary_per_player = {
    'ruben dias': 6000000, 'gabriel jesus': 4680000, 'riyad mahrez':
       6240000}
--

In [72]:
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]: {'RUBEN DIAS': 6.0, 'GABRIEL JESUS': 4.68, 'RIYAD MAHREZ': 6.24}
```

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]: 16920000
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

finds the length of a list.

```
In [77]: len(['ruben dias', 'gabriel jesus', 'riyad mahrez'])
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(['ruben dias', 'gabriel jesus', 'riyad mahrez'])` and `3`. We could define variables with it:

```
In [78]: n_goals = len(['ruben dias', 'gabriel jesus', 'riyad mahrez'])

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

Or use it in math.

```
In [80]: 4 + len(['ruben dias', 'gabriel jesus', 'riyad mahrez'])
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 ejected(nyellow, nred):
    """
    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 yellow and red cards and returns a bool
    indicating whether the player is ejected
    """
    return (nred >= 1) or (nyellow >= 2)
```

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

```
In [81]: ejected(1, 0)
Out[81]: False
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Note the arguments `nyellow` and `nred`. These work just like normal variables, except they're 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(nyellow)
...
NameError: name 'nyellow' is not defined
```

You'll get an error. `nyellow` only exists inside the `ejected` 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 ejected_noisy(nyellow, nred):
    """
    this function takes number of yellow and red cards and returns a bool
    indicating whether the player is ejected

    it also prints out nyellow
    """
    print(nyellow)  # works here since we're inside fn
    return (nred >= 1) or (nyellow >= 2)
```

And then when we call it:

```
In [83]: ejected_noisy(0, 1)
0
Out[83]: True
```

Note the `0` in the REPL. Along with returning a bool, `ejected_noisy` prints the value of `nyellow`. 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]: ejected(1)
...
TypeError: ejected() missing 1 required positional argument: 'nred'
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

We got an error. We gave it 1, which got assigned to `nyellow` but `nred` didn't get a value.

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

```
def ejected_wdefault(nyellow, nred=0):
    """
    this function takes number of yellow and red cards and returns a bool
    indicating whether the player is ejected
    """
    return (nred >= 1) or (nyellow >= 2)
```

Now `nred` is **optional** because we gave it a default value. Note `nyellow` is still **required** because it doesn't have a default value. 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]: ejected_wdefault(2)
Out[85]: True
```

But if we run it without `nyellow` we still get an error:

```
In [86]: ejected_wdefault(nred=0)
...
TypeError: ejected_wdefault() missing 1 required positional argument: '
   nyellow'
```

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]: ejected(1, 0)
Out[87]: False
```

The function assigns 1 to `nyellow` and 0 to `nred`. It's in that order (`nyellow`, `nred`) because that's the order we wrote them when we defined `ejected` 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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [88]: ejected?
Signature: ejected(nyellow, nred)
...
Type:      function
```

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

```
In [89]: ejected(nred=0, nyellow=1)
Out[89]: False
```

Keyword arguments are useful because you no longer have to remember the exact argument order. In practice, they're also required to take advantage of default values.

Think about it: presumably your function includes defaults so that you don't have to type in a value for *every* argument, *every* time. But if you're passing some values and not others, how's Python supposed to know which is which?

The answer is keyword arguments.

You're allowed to mix positional and keyword arguments:

```
In [90]: ejected(1, nred=0)
Out[90]: False
```

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)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 = ['FWD', 'MID', 'D', 'GK']

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

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: 4'
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
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 [3] that other people have written using Python [4] 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:

[3] 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.

[4] 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [98]: from os import path

In [99]: DATA_DIR = '/Users/nathan/code-soccer-files/data'

In [100]: path.join(DATA_DIR, 'shots.csv')
Out[100]: '/Users/nathan/code-soccer-files/data/shots.csv'

In [101]: os.path.join(DATA_DIR, 'shots.csv')  # alt way of calling
Out[101]: '/Users/nathan/code-soccer-files/data/shots.csv'
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

End of Chapter Exercises

2.1

Which of the following are valid Python variable names?

a) `_throwaway_data`
b) `n_shots`
c) `1st_half`
d) `shotsOnGoal`
e) `wc_2018_champion`
f) `player position`
g) `@home_or_away`
h) `'num_penalties'`

2.2

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

```
match_minutes = 45
match_minutes = match_minutes + 45
match_minutes = match_minutes + 5
```

2.3

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

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_oconnell` that takes in a player name and returns a `bool` indicating whether the player's name is "Jack O'Connell" — regardless of case or whether the user included the `'`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

2.6

Write a function `a_lot_of_goals` that takes in a number (e.g. 0, 2, or 6) and returns a string `'6 is a lot of goals!` if the number is >= 4 or `"2 is not that many goals"` otherwise.

2.8

Say we have a list:

```
roster = ['ruben dias', 'gabriel jesus', 'riyad mahrez']
```

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

2.9

Say we have a dict:

```
shot_info = {'shooter': 'Robert Lewandowski', 'foot': 'right',
             'went_in': False}
```

a) How would you change `'shooter'` to 'Cristiano Ronaldo'?

b) Write a function `toggle_foot` that takes a dict like `shot_info`, turns `'foot'` to the opposite of whatever it is (so right to left or left to right), and returns the updated dict.

2.10

Assuming we've defined our same dict:

```
shot_info = {'shooter': 'Robert Lewandowski', 'foot': 'right',
             'went_in': False}
```

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

a) `shot_info['is_pk']`
b) `shot_info[shooter]`
c) `shot_info['distance'] = 20`

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

2.11

Say we're working with the list:

```
roster = ['ruben dias', 'gabriel jesus', 'riyad mahrez']
```

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 = {'CB': 'ruben dias',
               'CF': 'gabriel jesus',
               'RW': 'riyad mahrez',
               'LW': 'raheem sterling'}
```

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 `'j'` or `'m'`.

2.12

a) Write a function `mapper` that takes a list and a function, applies the function to every item in the list and returns it.

b) Assuming 90 minute matches, use `mapper` with an anonymous function to come up with a list of stoppage time values for the following match times:

```
match_minutes = [95, 92, 90, 91, 97, 95]
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 how 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 football 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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/fantasymath/fantasybook/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 = '/Users/nathan/fantasybook/data' # <- change this
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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       foot   goal  ...  shot_loc_desc    min
0          A. Samedov      right  False  ...            NaN    6.0
1   Yasir Al Shahrani      right  False  ...            NaN    5.0
2          Y. Zhirkov       left  False  ...            NaN    8.0
3        Y. Gazinskiy  head/body   True  ...  goal low left   11.0
4  Mohammad Al Sahlawi       left  False  ...            NaN   21.0

[5 rows x 24 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`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [7]: shots.columns
Out[7]:
Index(['name', 'foot', 'goal', 'dist', 'time', 'match_id', 'period',
       'team_id', 'player_id', 'event_id', 'accurate', 'shot_loc',
       'counter', 'opportunity', 'x1', 'y1', 'x2', 'y2', 'H1', 'H2',
       'E1', 'shot_loc_desc', 'time2', 'min'],
      dtype='object')
```

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

```
In [8]: shots.shape
Out[8]: (1366, 24)
```

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             A. Samedov
1      Yasir Al Shahrani
2             Y. Zhirkov
3           Y. Gazinskiy
4    Mohammad Al Sahlawi
```

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 vice versa. Calling the `to_frame` method will turn any Series into a one-column DataFrame.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [11]: shots['name'].to_frame().head()
Out[11]:
                  name
0           A. Samedov
1    Yasir Al Shahrani
2           Y. Zhirkov
3         Y. Gazinskiy
4  Mohammad Al Sahlawi

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', 'foot', 'goal', 'period']].head()
Out[13]:
                  name       foot   goal period
0           A. Samedov      right  False     1H
1    Yasir Al Shahrani      right  False     1H
2           Y. Zhirkov       left  False     1H
3         Y. Gazinskiy  head/body   True     1H
4  Mohammad Al Sahlawi       left  False     1H

In [14]: type(shots[['name', 'foot', 'goal', 'period']])
Out[14]: pandas.core.frame.DataFrame
```

Notice the difference between the two:

- `shots['name']`
- `shots[['name', 'foot', 'goal', 'period']]`

In the former we have `'name'`.

It's completely replaced by `['name', 'foot', 'goal', 'period']` 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]: shots['name', 'foot', 'goal', 'period'].head()
...
KeyError: ('name', 'foot', 'goal', 'period')
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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', 'foot', 'goal', 'period']].head()
Out[16]:
                  name       foot   goal period
0           A. Samedov      right  False     1H
1    Yasir Al Shahrani      right  False     1H
2           Y. Zhirkov       left  False     1H
3         Y. Gazinskiy  head/body   True     1H
4  Mohammad Al Sahlawi       left  False     1H
```

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       foot  ...        time2   min
shot_id                                   ...
258612244           A. Samedov      right  ...   407.123899   6.0
258612248    Yasir Al Shahrani      right  ...   327.142941   5.0
258612307           Y. Zhirkov       left  ...   526.276996   8.0
258612368         Y. Gazinskiy  head/body  ...   693.396917  11.0
258612558  Mohammad Al Sahlawi       left  ...  1266.276267  21.0
```

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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [18]: shots.head()
Out[18]:
                  name       foot   goal  ...        time2   min
0           A. Samedov      right  False  ...   407.123899   6.0
1    Yasir Al Shahrani      right  False  ...   327.142941   5.0
2           Y. Zhirkov       left  False  ...   526.276996   8.0
3         Y. Gazinskiy  head/body   True  ...   693.396917  11.0
4  Mohammad Al Sahlawi       left  False  ...  1266.276267  21.0
```

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       foot  ...        time2   min
shot_id                                   ...
258612244           A. Samedov      right  ...   407.123899   6.0
258612248    Yasir Al Shahrani      right  ...   327.142941   5.0
258612307           Y. Zhirkov       left  ...   526.276996   8.0
258612368         Y. Gazinskiy  head/body  ...   693.396917  11.0
258612558  Mohammad Al Sahlawi       left  ...  1266.276267  21.0
```

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

```
In [21]:
# reload shots with default 0, 1, ... index
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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [23]: shots.reset_index().head()
Out[23]:
     shot_id                name  ...        time2   min
0  258612244           A. Samedov  ...   407.123899   6.0
1  258612248    Yasir Al Shahrani  ...   327.142941   5.0
2  258612307           Y. Zhirkov  ...   526.276996   8.0
3  258612368         Y. Gazinskiy  ...   693.396917  11.0
4  258612558  Mohammad Al Sahlawi  ...  1266.276267  21.0
```

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 some basic information about 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'] == 'E1') |
                      (shots['period'] == 'E2')),
                      ['name', 'goal', 'period']]

In [25]: shots_ot.head()
Out[25]:
                    name   goal period
shot_id
280217011     Jordi Alba  False     E1
280217158           Koke  False     E1
280217216  Marco Asensio  False     E1
280217250     Iago Aspas  False     E1
280217309          Piqus  False     E1
```

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

```
In [26]: shots_ot.sort_values('name', inplace=True)

In [27]: shots_ot.head()
Out[27]:
                  name   goal period
shot_id
279011647   A. Erokhin  False     E1
263156279  A. ćKramari  False     E2
263156061  A. ćKramari  False     E1
261268830  A. ćKramari  False     E2
263156133  A. ćKramari  False     E1
```

Now, what if we want to go back and add in the `foot` (as in right or left) column from our original, all

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

shots DataFrame?

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

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

In [29]: shots_ot.head()
Out[29]:
                  name   goal period       foot
shot_id
279011647   A. Erokhin  False     E1  head/body
263156279  A. ćKramari  False     E2      right
263156061  A. ćKramari  False     E1       left
261268830  A. ćKramari  False     E2      right
263156133  A. ćKramari  False     E1       left
```

Voila. 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 `foot` values from our old DataFrame.

We're able to do that because `shots`, `shots_ot`, and the Series `shots['foot']` 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)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Exercises

3.0.1

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

3.0.2

Use the `match` DataFrame to create another DataFrame, `match10`, that is the first 10 matches (e.g. the 10 matches with the earliest dates).

3.0.3

Sort `match` by **`label`** in descending order (so Uruguay - Saudi Arabia is on the first line).. On another line, look at `match` in the REPL and make sure it worked.

3.0.4

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

3.0.5

a) Make a new DataFrame, `match_simple`, with just the columns `'date'`, `'home_team'`, `'away_team'`, `'home_score'` and `'away_score'` in that order.

b) Rearrange `match_simple` so the order is `'home_team'`, `'away_team'`, `'date'`, `'home_score'`, `'away_score'`.

c) Using the original `match` DataFrame, add the `'match_id'` column to `match_simple`.

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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['yellow_cards'] = 1

In [2]: pg[['name', 'min', 'yellow_cards']].head()
Out[2]:
              name   min  yellow_cards
0     D. Cheryshev  66.0             1
1  Mário Fernandes  90.0             1
2      I. Akinfeev   NaN             1
3   S. Ignashevich  90.0             1
4       A. Dzagoev  24.0             1
```

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

It's no different than creating it originally.

```
In [3]: pg['yellow_cards'] = 2

In [4]: pg[['name', 'min', 'yellow_cards']].head()
Out[4]:
              name   min  yellow_cards
0     D. Cheryshev  66.0             2
1  Mário Fernandes  90.0             2
2      I. Akinfeev   NaN             2
3   S. Ignashevich  90.0             2
4       A. Dzagoev  24.0             2
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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['shot_pct'] = 100*pg['goal']/pg['shot']

In [6]: pg[['name', 'shot', 'goal', 'shot_pct']].head()
Out[6]:
              name  shot  goal   shot_pct
0     D. Cheryshev     3     2  66.666667
1  Mário Fernandes     0     0        NaN
2      I. Akinfeev     0     0        NaN
3   S. Ignashevich     0     0        NaN
4       A. Dzagoev     0     0        NaN
```

This adds a new column `shot_pct` to our `pg` DataFrame[1].

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[2] of player rank:

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

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

In [9]: pg['ln_pass'] = np.log(pg['pass'])
```

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

```
In [10]: pg['goal_width_ft'] = 24
```

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.

[1] If we wanted, we could also create a separate column (Series), unattached to `pg`, but with the same index like this `shot_pct = 100*pg['goal']/pg['shot']`.

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

```
In [11]: pg[['name', 'team', 'match_id', 'goal_width_ft']].sample(5)
Out[11]:
                   name          team  match_id  goal_width_ft
1181            K. Glik        Poland   2058000             24
991            F. Delph       England   2057993             24
193     Bruno Fernandes      Portugal   2057960             24
20        Osama Hawsawi  Saudi Arabia   2057954             24
1225  Cristiano Ronaldo      Portugal   2058002             24
```

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]:
734       R. WALLACE
886         A. EKDAL
862    SEON-MIN MOON
229      Y. BELHANDA
750         M. NEUER

In [2]: pg['name'].str.replace('.', ' ').sample(5)
Out[2]:
541       F  Armani
346     A  Carrillo
1104       S  Arias
484        A ć Rebi
648       B  Oviedo
```

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

```
In [3]: (pg['name'] + ', ' + pg['pos'] + ' - ' + pg['team']).sample(5)
Out[3]:
391      K. Schmeichel, GKP - Denmark
525        É. Banega, MID - Argentina
1335    A. Christensen, DEF - Denmark
681        V. ćStojkovi, GKP - Serbia
522          T. Ebuehi, DEF - Nigeria
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [4]: pg['name'].str.replace('.', ' ').str.lower().sample(5)
Out[4]:
360        a  griezmann
569         a ć kramari
802            t  kroos
1669          b  pavard
905     t  alderweireld
```

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 defender.

```
In [1]: pg['is_defender'] = (pg['pos'] == 'DEF')

In [2]: pg[['name', 'team', 'is_defender']].sample(5)
Out[2]:
              name     team  is_defender
1523     F. Smolov   Russia        False
277   M. Mohammadi     Iran         True
1064     B. Srarfi  Tunisia        False
1028    M. Dembélé  Belgium        False
694     A. Kolarov   Serbia         True
```

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

```
In [3]: pg['is_a_mid_or_fwd'] = ((pg['pos'] == 'MID') |
                                 (pg['pos'] == 'FWD'))

In [4]: pg['balanced_off'] = (pg['goal'] > 0) & (pg['assist'] > 0)
```

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

```
In [4]:
pg['not_fr_or_eng'] = ~((pg['team'] == 'England') |
                        (pg['team'] == 'France'))
```

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 a goal *or* an assist you could do:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [5]: (pg[['goal', 'assist']] > 0).sample(5)
Out[5]:
       goal  assist
1595   True   False
1281   True   False
913   False   False
1078  False    True
794   False   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 South America. Rather than writing a long boolean expression with many | values, we might do something like:

```
In [1]:
def is_south_america(team):
  """
  Takes some string named team ('England', 'Germany, 'Argentina' etc) and
  checks whether it's in South America.
  """
  return team in ['Brazil', 'Uruguay', 'Colombia', 'Argentina',
                  'Costa Rica', 'Peru']

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

In [3]: pg[['name', 'team', 'is_sa']].sample(5)
Out[3]:
            name            team  is_sa
944       D. Alli         England  False
120  C. Rodríguez         Uruguay   True
277  M. Mohammadi            Iran  False
788   Min-Woo Kim  Korea Republic  False
870   L. Goretzka         Germany  False
```

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

Our function `is_south_america` 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [4]:
pg['is_sa_alternate'] = pg['team'].apply(lambda x: x in [
    ['Brazil', 'Uruguay', 'Colombia', 'Argentina', 'Costa Rica', 'Peru']])
```

Dropping Columns

Dropping a column works like this:

```
In [5]: pg.drop('is_sa_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    TEAM   MIN  ...  NOT_FR_OR_ENG  IS_SA
0     D. Cheryshev  Russia  66.0  ...           True  False
1  Mário Fernandes  Russia  90.0  ...           True  False
2      I. Akinfeev  Russia   NaN  ...           True  False
3   S. Ignashevich  Russia  90.0  ...           True  False
4       A. Dzagoev  Russia  24.0  ...           True  False
```

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 `minutes` to just `min`.

```
In [4]: pg.rename(columns={'min': 'minutes'}, inplace=True)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 *shot percentage* (goals scored/shot attempts).

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

But what should it's value be for players who didn't take any shots?

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 shots and guys who didn't take any. Missing is better.

```
In [2]: pg[['name', 'team', 'goal', 'shot', 'shot_pct']].head(10)
Out[2]:
              name    team  goal  shot  shot_pct
0     D. Cheryshev  Russia     2     3  0.666667
1  Mário Fernandes  Russia     0     0       NaN
2      I. Akinfeev  Russia     0     0       NaN
3   S. Ignashevich  Russia     0     0       NaN
4       A. Dzagoev  Russia     0     0       NaN
5        A. Dzyuba  Russia     1     1  1.000000
6       A. Samedov  Russia     0     2  0.000000
7        F. Smolov  Russia     0     0       NaN
8       Y. Zhirkov  Russia     0     1  0.000000
9     Y. Gazinskiy  Russia     1     1  1.000000
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: pg['shot_pct'].isnull().head(10)
Out[1]:
0    False
1     True
2     True
3     True
4     True
5    False
6    False
7     True
8    False
9    False

In [2]: pg['shot_pct'].notnull().head(10)
Out[2]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7    False
8     True
9     True
```

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

```
In [3]: pg['shot_pct'].fillna(-99).head(10)
Out[3]:
0     0.666667
1   -99.000000
2   -99.000000
3   -99.000000
4   -99.000000
5     1.000000
6     0.000000
7   -99.000000
8     0.000000
9     1.000000
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: pg['date'].sample(5)
Out[1]:
923     20180618
1298    20180701
336     20180616
115     20180625
355     20180616
```

In normal Python, if we wanted to get the year, month and day out of a string like `'20180618'` we would just do:

```
In [2]: date = '20180618'

In [3]: year = date[0:4]

In [4]: month = date[4:6]

In [5]: day = date[6:8]

In [6]: year
Out[6]: '2018'

In [7]: month
Out[7]: '06'

In [8]: day
Out[8]: '18'
```

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
660      F. Calvo  Costa Rica    06  20180622
724   R. Azofeifa  Costa Rica    06  20180627
903   J. Gallardo      Mexico    06  20180627
1460   F. Muslera     Uruguay    07  20180706
1197  J. Cuadrado    Colombia    06  20180628
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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]:
609     6
844     6
1492    7
1583    7
149     6
```

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

```
In [13]: pg.dtypes.head()
Out[13]:
name        object
team        object
minutes    float64
shot         int64
goal         int64
```

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).

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Exercises

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

3.1.2 Add a column to `pm`, `'ob_touches'` that is number of throw-ins plus number of corners.

3.1.3 Add a column `'player_desc'` to pm that takes the form, ' is the ', e.g. `'L. Messi is the Argentina FWD'` for Lional Messi.

3.1.4 Add a boolean column to pm `'at_least_one_throwin'` indicating whether a player had at least one throw-in.

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

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

3.1.7

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

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

3.1.8

a) Make a new column `'air_duel_won_percentage'` indicating the percentage of a players air duels that a player won.

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

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 numeric columns in your DataFrame you can do:

```
In [1]: pg[['shot', 'goal', 'assist', 'pass', 'throw', 'corner']].mean()
Out[1]:
shot       0.817475
goal       0.104728
assist     0.050269
pass      31.599641
throw      1.473369
corner     0.345302
```

We can also do `max`.

```
In [2]:
pg[['name', 'shot', 'goal', 'assist', 'pass', 'throw', 'corner']].max()

Out[2]:
name Š     . Vrsaljko
shot                7
goal                3
assist              2
pass              174
throw              19
corner              8
```

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 `'S. Vrsaljko'` for `name` in our player-game data).

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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[['shot', 'goal', 'assist', 'pass']].mean(axis=0)
Out[3]:
shot       0.817475
goal       0.213046
assist     0.050269
pass      31.599641
```

(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: shots, goals, assists and passes.

```
In [4]: pg[['shot', 'goal', 'assist', 'pass']].mean(axis=1).head()
Out[4]:
0    6.75
1    6.50
2    4.25
3    6.50
4    2.00
```

That number is meaningless here, but sometimes data is structured differently. For example, you might have a DataFrame where the columns are: `name`, `goals1`, `goals2`, `goals3`, ... `goals7` — where each row represents a player and there are (up to) 7 columns, one for potential game each player plays (3 group games, up to 4 knockout games).

Then `axis=0` would give the average number of goals across *all* players for *each game*, which could be interesting, and `axis=1` would give you *each player's* average score for the *whole tournament*, which could also be interesting.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 observations are defenders who scored at least one goal in that game?

```
In [1]: pg['defender_scored'] = (pg['pos'] == 'DEF') & (pg['goal'] > 0)

In [2]: pg['defender_scored'].mean()
Out[2]: 0.020945541591861162

In [3]: pg['defender_scored'].sum()
Out[3]: 35
```

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 have more than 100 passes in a game?

```
In [4]: (pg['pass'] > 100).any()
Out[4]: True
```

Yes. Did everyone in this dataset make ast least one pass?

```
In [5]: (pg['pass'] > 0).all()
Out[5]: False
```

No.

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

For example, to look by row and check if each player won more than 5 air duels *or* got more than 5 interceptions we could do:

```
In [6]: (pg[['air_duel_won', 'interception']] > 5).any(axis=1)
Out[6]:
0       False
1        True
2       False
3        True
4       False
        ...
1666    False
1667    False
1668     True
1669    False
1670    False
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

```
In [7]: (pg[['air_duel_won', 'interception']] > 5).any(axis=1).sum()
Out[7]: 332
```

How often did someone get *both* 5 air duels won *and* interceptions in the same game?

```
In [8]: (pg[['air_duel_won', 'interception']] > 5).all(axis=1).sum()
Out[8]: 14
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 [9]: pg['team'].value_counts()
Out[9]:
Croatia           100
England            99
Belgium            94
France             83
Russia             72
Sweden             70
Brazil             70
Uruguay            70
Spain              57
Colombia           56
Portugal           56
Japan              55
Switzerland        54
Argentina          53
Mexico             46
Senegal            42
Egypt              42
Korea Republic     42
Iran               42
Costa Rica         42
Nigeria            42
Iceland            42
Germany            42
Denmark            41
Saudi Arabia       41
Serbia             40
Panama             40
Morocco            40
Tunisia            38
Poland             34
Peru               26
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [10]: pg['team'].value_counts(normalize=True)
Out[10]:
Croatia           0.059844
England           0.059246
Belgium           0.056254
France            0.049671
Russia            0.043088
Sweden            0.041891
Brazil            0.041891
Uruguay           0.041891
Spain             0.034111
Colombia          0.033513
Portugal          0.033513
Japan             0.032914
Switzerland       0.032316
Argentina         0.031718
Mexico            0.027528
Senegal           0.025135
Egypt             0.025135
Korea Republic    0.025135
Iran              0.025135
Costa Rica        0.025135
Nigeria           0.025135
Iceland           0.025135
Germany           0.025135
Denmark           0.024536
Saudi Arabia      0.024536
Serbia            0.023938
Panama            0.023938
Morocco           0.023938
Tunisia           0.022741
Poland            0.020347
Peru              0.015560
```

So 5.98% (100/1671) of the observations in our player-game data are from Croatia (this makes sense since they played the most games), 5.92% (99/1671) are from England, etc.

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

```
In [11]: pd.crosstab(pg['team'], pg['pos']).head()
Out[11]:
pos          DEF  FWD  GKP  MID
team
Argentina     16   16    4   17
Belgium       28   26    7   33
Brazil        22   19    5   24
Colombia      17   10    4   25
Costa Rica    17    6    3   16
```

Crosstab also takes an optional `normalize` argument.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Exercises

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

3.2.2 Add a column to `pm` that gives the total number of clearances, crosses, assists and key passes. for each player-game. Do it two ways, one with basic arithmetic operators and another way using a built-in pandas function. Call them '`named_pass1`' and '`named_pass1`'. Prove that they're the same.

3.2.3

a) What were the average values for shots, assists, and passes?

b) How many times in our data did someone score at least 1 goal and have at least 1 assist?

c) What % of player performances was that?

d) How many own goals were there total in our sample?

e) What position is most represented in our data? Least?

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 match DataFrame `dfm` 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, `2058017` is the championship `match_id`.

```
In [1]: championship_id = 2058017

In [2]: dfm.loc[championship_id]
Out[2]:
label              France - Croatia, 4 - 2
group                                  NaN
date                   2018-07-15 15:00:00
venue         Olimpiyskiy stadion Luzhniki
dur                                Regular
gameweek                                 0
round_id                           4165368
home                                  4418
away                                  9598
winner                                4418
loser                                 9598
ref                                 378051
ref2                                378038
ref3                                378060
ref4                                377215
home_score                               4
away_score                               2
home_team                           France
away_team                          Croatia
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [3]: group_a_ids = [2057959, 2057958, 2057957, 2057956, 2057955,
                       2057954]

In [4]: dfm.loc[group_a_ids]
Out[4]:
                                 label    group  ...      away_team
match_id                                         ...
2057959     Saudi Arabia - Egypt, 2 - 1  Group A  ...          Egypt
2057958         Uruguay - Russia, 3 - 0  Group A  ...         Russia
2057957   Uruguay - Saudi Arabia, 1 - 0  Group A  ...   Saudi Arabia
2057956           Russia - Egypt, 3 - 1  Group A  ...          Egypt
2057955          Egypt - Uruguay, 0 - 1  Group A  ...        Uruguay
2057954    Russia - Saudi Arabia, 5 - 0  Group A  ...   Saudi Arabia
```

While not technically filtering by rows, you can also pass `loc` a second argument to limit which columns you return. This returns the **`label`**, `group` and `venue` columns for the ids we specified.

```
In [5]: dfm.loc[group_a_ids, ['label', 'group', 'venue']]
Out[5]:
                                 label  ...                        venue
match_id                                ...
2057959     Saudi Arabia - Egypt, 2 - 1  ...               Volgograd Arena
2057958         Uruguay - Russia, 3 - 0  ...                  Samara Arena
2057957   Uruguay - Saudi Arabia, 1 - 0  ...                  Rostov Arena
2057956           Russia - Egypt, 3 - 1  ...           Stadion Krestovskyi
2057955          Egypt - Uruguay, 0 - 1  ...           Stadion Centralnyj
2057954    Russia - Saudi Arabia, 5 - 0  ...  Olimpiyskiy stadion Luzhniki
```

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]: dfm.loc[group_a_ids, 'venue']
Out[6]:
match_id
2057959                 Volgograd Arena
2057958                    Samara Arena
2057957                    Rostov Arena
2057956             Stadion Krestovskyi
2057955             Stadion Centralnyj
2057954    Olimpiyskiy stadion Luzhniki
```

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`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

So say we're just interested in Group B. Let's create our column of booleans that indicate whether a match was part of Group B.

```
In [7]: is_group_b = dfm['group'] == 'Group B'

In [8]: is_group_b.head()  # none of these are
Out[8]:
match_id
2058017    False
2058012    False
2057977    False
2057974    False
2058014    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 [8]: dfm_b = dfm.loc[is_group_b]

In [9]: dfm_b[['label', 'group', 'venue']].head()
Out[9]:
                              label    group                        venue
match_id
2057964      Iran - Portugal, 1 - 1  Group B                Mordovia Arena
2057965      Spain - Morocco, 2 - 2  Group B            Kaliningrad Stadium
2057962   Portugal - Morocco, 1 - 0  Group B  Olimpiyskiy stadion Luzhniki
2057963         Iran - Spain, 0 - 1  Group B                  Kazan Arena
2057960     Portugal - Spain, 3 - 3  Group B     Olimpiyskiy Stadion Fisht
```

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 `is_group_b` has the same index as `dfm` because we created it with `dfm['group'] == 'Group B'` and that's how Pandas works.

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

```
In [10]: dfm_g = dfm.loc[dfm['group'] == 'Group G']

In [11]: dfm_g[['label', 'group', 'venue']].head()
Out[11]:
                              label    group                       venue
match_id
2057994   England - Belgium, 0 - 1  Group G        Kaliningrad Stadium
2057991   Tunisia - England, 1 - 2  Group G            Volgograd Arena
2057992   Belgium - Tunisia, 5 - 2  Group G            Otkrytiye Arena
2057995    Panama - Tunisia, 1 - 2  Group G             Mordovia Arena
2057990    Belgium - Panama, 3 - 0  Group G  Olimpiyskiy Stadion Fisht
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Having to refer to the name of your DataFrame (`dfm.loc[dfm['group'] == 'Group G']` here) multiple times in one line may seem cumbersome at first. It can get a bit verbose, but this is a very common thing to do so get used to it.

Any boolean column or boolean operation works.

```
In [13]: is_group_d = dfm['group'] == 'Group D'

In [14]: dfm_not_d = dfm.loc[~is_group_d]

In [15]: dfm_not_d[['label', 'group', 'venue']].head()
Out[15]:
                                label    group              venue
match_id
2058017       France - Croatia, 4 - 2      NaN           Luzhniki
2058012   Russia - Croatia, 2 - 2 (P)      NaN      Stadion Fisht
2058014       France - Belgium, 1 - 0      NaN        Krestovskyi
2058011       Brazil - Belgium, 1 - 2      NaN        Kazan Arena
2057994      England - Belgium, 0 - 1  Group G        Kaliningrad
```

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]: dfm.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]: dfm.drop_duplicates('venue')[['label', 'group', 'venue']]
Out[2]:
                                label  ...                        venue
match_id                               ...
2058017         France - Croatia, 4 - 2  ...  Olimpiyskiy stadion Luzhniki
2058012     Russia - Croatia, 2 - 2 (P)  ...     Olimpiyskiy Stadion Fisht
2057977        Iceland - Croatia, 1 - 2  ...                 Rostov Arena
2057974      Argentina - Croatia, 0 - 3  ...       Stadion Nizhny Novgorod
2058014         France - Belgium, 1 - 0  ...           Stadion Krestovskyi
2058011         Brazil - Belgium, 1 - 2  ...                 Kazan  Arena
2057994        England - Belgium, 0 - 1  ...           Kaliningrad Stadium
2057968            France - Peru, 1 - 0  ...           Stadion Central nyj
2058013         Sweden - England, 0 - 2  ...                 Samara Arena
2058009   Colombia - England, 1 - 1 (P)  ...               Otkrytiye Arena
2057991        Tunisia - England, 1 - 2  ...               Volgograd Arena
2057997         Colombia - Japan, 1 - 2  ...                Mordovia Arena
2057984         Germany - Mexico, 0 - 1  ...              Stadion Luzhniki
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Note, although it though it might look strange to have

```
[['label', 'group', 'venue']]
```

immediately after `.drop_duplicates('venue')`, it works because the result of `dfm.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]: dfm.duplicated().head()
Out[3]:
match_id
2058017    False
2058012    False
2057977    False
2057974    False
2058014    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]: dfm['group'].duplicated().head()
Out[4]:
match_id
2058017    False
2058012     True
2057977    False
2057974     True
2058014     True
Name: group, dtype: bool
```

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

Combining Filtering with Changing Columns

Often you'll want to combine filtering with modifying columns and update columns *only* for certain rows.

For example, say we want a column "home/away description" that summarizes which team won ("home team won!", "away team won!" or "tied!")

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

We would do that using something like this:

```
In [1]: dfm['home_away_desc'] = np.nan

In [2]: dfm.loc[dfm['home'] == dfm['winner'],
                'home_away_desc'] = 'home team won!'

In [3]: dfm.loc[dfm['away'] == dfm['winner'],
                'home_away_desc'] = 'away team won!'

In [4]: dfm.loc[dfm['winner'] == 0, 'home_away_desc'] = 'tied!'
```

We start by creating an empty column, `home_away_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 `winner` equals what we want) and the one column (`home_away_desc`) and assign the correct value to it.

Note Pandas indexing ability is what allows us to assign, e.g., `'tied!'` only to observations where `winner` equals `0` without worrying that the other observations will be affected.

```
In [5]: dfm['home_away_desc'].value_counts()
Out[5]:
away team won!    27
home team won!    26
tied!              8
```

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 `dfm` so it only includes Group A:

```
In [1]: dfm.query("group == 'Group A'").head()
Out[1]:
                                  label  ...  home_away_desc
match_id                                 ...
2057956           Russia - Egypt, 3 - 1  ...  home team won!
2057959     Saudi Arabia - Egypt, 2 - 1  ...  home team won!
2057954    Russia - Saudi Arabia, 5 - 0  ...  home team won!
2057957   Uruguay - Saudi Arabia, 1 - 0  ...  home team won!
2057958         Uruguay - Russia, 3 - 0  ...  home team won!
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Notice how inside the string we're referring to `group` without quotes, like a variable name. We can refer to any column name like that. String values inside `query` strings (e.g. `'Group A'`) 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. `"group == 'Group A'"`) so that they work well together.

We can also call boolean columns directly:

```
In [2]: dfm['is_group_b'] = dfm['group'] == 'Group B'

In [3]: dfm.query("is_group_b").head()
Out[3]:
                              label    group  ...   is_group_b
match_id                                      ...
2057964      Iran - Portugal, 1 - 1  Group B  ...         True
2057965      Spain - Morocco, 2 - 2  Group B  ...         True
2057962   Portugal - Morocco, 1 - 0  Group B  ...         True
2057963         Iran - Spain, 0 - 1  Group B  ...         True
2057960     Portugal - Spain, 3 - 3  Group B  ...         True
```

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

```
In [4]: dfm.query("group.isnull()")[['label', 'group', 'venue']].head()
Out[4]:
                                label group                        venue
match_id
2058017       France - Croatia, 4 - 2   NaN  Olimpiyskiy stadion Luzhniki
2058012   Russia - Croatia, 2 - 2 (P)   NaN     Olimpiyskiy Stadion Fisht
2058014       France - Belgium, 1 - 0   NaN           Stadion Krestovskyi
2058011       Brazil - Belgium, 1 - 2   NaN                  Kazan' Arena
2058003     France - Argentina, 4 - 3   NaN                  Kazan' Arena
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

and dropping duplicates. Finally, we learned about `query`, which is a shorthand way to filter your data.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Exercises

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

3.3.2 Make smaller DataFrame with just Brazilian players and only the columns: `'player_name'`, `'pos'`, `'foot'`, `'weight'`, `'height'`. Do it two ways: (1) using the `loc` syntax, and (b) another time using the `query` syntax. Call them `dfp_bra1` and `dfp_bra2`.

3.3.3 Make a DataFrame `dfp_no_bra` with the same columns that is everyone EXCEPT Brazilian players, add the `'nationality'` column to it.

3.3.4

a) Are there any duplicates by birth day `dfp` DataFrame? Hint: remember people born in different years can have the same birthday.

b) Divide `dfp` into two separate DataFrames `dfp_dups` and `dfp_nodups`, one with dups (birthday) one without.

3.3.5 Add a new column to `dfp` called `'height_description'` with the values:

- `'tall'` for players whose height is greater than 195 cm
- `'short'` for players less than 175 cm
- missing otherwise

3.3.6 Make a new DataFrame with only observations for which '`height_description`' is missing. Do this with both the (a) loc and (b) query syntax. Call them `dfp_no_desc1` and `dfp_no_desc2`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.*

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 **aggreggating** — 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 (up to) 7 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 scored a goal, `0` if not) and wanted to group it to the game level we could take the:

- *sum* to get total points 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

groupby

Pandas handles grouping via the `groupby` function.

```
In [1]: shots.groupby('match_id').sum().head()
Out[1]:
          goal      dist_m      dist_ft  ...   E1         time2     min
match_id                                 ...
2057954      4  304.845939   999.894679  ...  0.0  53283.658696   879.0
2057955      1  343.561912  1126.883072  ...  0.0  56130.383870   927.0
2057956      2  439.253711  1440.752172  ...  0.0  59121.048323   974.0
2057957      1  323.745765  1061.886108  ...  0.0  65229.073335  1078.0
2057958      1  312.215401  1024.066515  ...  0.0  57359.851560   948.0
```

This gives us a DataFrame where every column is summed (because we called `.sum()` at the end) over `match_id`. Note how `match_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 goals and attempts for sum.

```
In [2]: shots['attempt'] = 1

In [3]: sum_cols = ['goal', 'attempt', 'accurate', 'counter', 'opportunity
    ']

In [4]: shots.groupby('match_id').sum()[sum_cols].head()
Out[4]:
          goal  attempt  accurate  counter  opportunity
match_id
2057954      4       18         6        0           10
2057955      1       18         7        1           14
2057956      2       24         4        0           17
2057957      1       20         6        3           14
2057958      1       15         6        1           11
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [5]:
shots.groupby('match_id').agg({
    'goal': 'sum',
    'attempt': 'count',
    'dist_m': 'mean',
    'dist_ft': 'mean'}).head()

Out[5]:
          goal  attempt     dist_m    dist_ft
match_id
2057954      4       18  16.935885  55.549704
2057955      1       18  19.086773  62.604615
2057956      2       24  18.302238  60.031340
2057957      1       20  16.187288  53.094305
2057958      1       15  20.814360  68.271101
```

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.

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 lets us explicitely name our new average distance columns:

```
In [6]:
shots.groupby('match_id').agg(
    goal = ('goal', 'sum'),
    attempt = ('attempt', 'count'),
    ave_dist_m = ('dist_m', 'mean'),
    ave_dist_ft = ('dist_ft', 'mean')).head()

Out[6]:
          goal  attempt  ave_dist_m  ave_dist_ft
match_id
2057954      4       18   16.935885    55.549704
2057955      1       18   19.086773    62.604615
2057956      2       24   18.302238    60.031340
2057957      1       20   16.187288    53.094305
2057958      1       15   20.814360    68.271101
```

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* team — by passing `groupby` a list.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [7]:
shots_team = shots.groupby(['match_id', 'team_id']).agg(
    goal = ('goal', 'sum'),
    attempt = ('attempt', 'count'),
    ave_dist_m = ('dist_m', 'mean'),
    min_dist_m = ('dist_m', 'min'),
    max_dist_ft = ('dist_ft', 'max'))

In [8]: shots_team.head()
Out[8]:
                  goal  attempt  ave_dist_m  min_dist_m  max_dist_ft
match_id team_id
2057954  14358       4       11   15.320402    6.623716   108.628943
         16521       0        7   19.474503    9.509131   113.867970
2057955  15670       1       11   17.965852    8.356085   110.011571
         16129       0        7   20.848221   11.266750    96.462566
2057956  14358       2       12   20.536720    6.206842   107.088120
```

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 `match_id` and `team_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 [9]: shots_team.loc[[(2057954, 14358), (2058017, 4418)]]
Out[9]:
                  goal  attempt  ave_dist_m  min_dist_m  max_dist_ft
match_id team_id
2057954  14358       4       11   15.320402    6.623716   108.628943
2058017  4418        2        7   19.859032   10.681969    90.257872
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 vice versa (unstacking).

For example, let's quick get average shot distance by player and foot (left, right, or head):

```
In [1]:
fd = shots.query("foot in ('left', 'right')").groupby(
    ['name', 'foot'])['dist_m'].mean().reset_index()

In [2]: fd.head()
Out[2]:
          name   foot     dist_m
0     A. Badri   left  30.932028
1  A. Carrillo   left  16.712171
2  A. Carrillo  right  27.723482
3    A. Cooper  right  26.696061
4    A. Dzyuba   left  16.191779
```

This data is at the *player and foot* level. Each row is a player-foot combination (A. Carillo, left foot), and we have the `dist` column for how far away their shots were on average from the goal.

But say instead we want this data to be at the *player* level and have two separate columns for average distance (left foot and right foot). Then we'd do:

```
In [3]: fd_reshaped = fd.set_index(['name', 'foot']).unstack()

In [4]: fd_reshaped.head()
Out[4]:
                dist_m
foot              left      right
name
A. Badri     30.932028        NaN
A. Carrillo  16.712171  27.723482
A. Cooper          NaN  26.696061
A. Dzyuba    16.191779  11.133510
A. Ekdal           NaN  19.647886
```

This move doesn't cost us any information. Initially, we find average shot distance for a particular player (Carrillo) and foot (left) by looking in the right row.

After we unstack it, we can find average shot distance for a particular player and foot by first finding the player's row, then finding the column for the right foot.

This lets us do things like calculate the average difference between right and left footed shots:

```
In [5]: (fd_reshaped['right'] - fd_reshaped['left']).mean()
Out[5]: 0.09521678124840466
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Or see how many players had further average shots with their right vs left foots:

```
In [6]: fd_reshaped.idxmax(axis=1).value_counts()
Out[6]:
right    197
left     143
```

(Note we haven't seen the `idxmax` function yet, but it's a basic summary function like `mean`, `max`, or `sum`. If you want, play around in the REPL and see if you can figure out exactly what it does).

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

```
In [7]: fd_reshaped_undo = fd_reshaped.stack()

In [8]: fd_reshaped_undo.head()
Out[8]:
name
A. Badri      left     30.932028
A. Carrillo   left     16.712171
              right    27.723482
A. Cooper     right    26.696061
A. Dzyuba     left     16.191779
```

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 vice versa.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Exercises

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

3.4.2

a) Load the player match data into a DataFrame named `dfpm`.

b) Figure out the average number of shots and goals each player scored per match.

c) Figure out the portion of at players that averaged 4 or more shots per game.

3.4.3

a) Make a new DataFrame `dftm` that's at the team/match level and includes the following info: match_id, team, total goals, passes, shots and number of players played. The last three columns should be named `total_goal`, `total_pass`, `total_shot` and `nplayed` respectively.

b) Because you grouped by more than one column, note `dftm` has a multi-index. Make those regular columns.

c) Add a new column boolean `no_goals` indicating whether or not the team scored 0 goals. Compare average total number of passes and shots for values of `no_goals`.

d) Run `dftm.groupby('match_id').count()`, compare it with `dftm.groupby('match_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?

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 how much age matters for players stats. We can't do it at the moment because we don't have any age data in our player-game data.

```
   player_id  match_id            name    team  shot  goal  assist
0       4513   2057954     D. Cheryshev  Russia     3     2       0
1      41123   2057954  Mário Fernandes  Russia     0     0       0
2     101576   2057954      I. Akinfeev  Russia     0     0       0
3     101583   2057954   S. Ignashevich  Russia     0     0       0
4     101590   2057954       A. Dzagoev  Russia     0     0       0
```

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 (Cheryshev, Russa vs Saudi Arabia). And the information we have: player_id, match_id, player name, shots, goals and assists. Information we *don't* have: the players' birthdays.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

```
   player_id      player_name  pos     team  birth_date
0      32793       A. N'Diaye  MID  Senegal    19900306
1         36  T. Alderweireld  DEF  Belgium    19890302
2         48    J. Vertonghen  DEF  Belgium    19870424
3         54       C. Eriksen  MID  Denmark    19920214
4         93  J. Guðmundsson  MID  Iceland    19901027
```

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

```
    player_id  match_id                name  ... goal  assist  birth_date
0        4513   2057954        D. Cheryshev  ...    2       0    19901226
32     101707   2057954           F. Smolov  ...    0       0    19900209
94     292954   2057954  Mohammed Al Burayk  ...    0       0    19920915
90     257800   2057954          A. Golovin  ...    1       2    19960530
37     101857   2057954          Y. Zhirkov  ...    0       0    19830820
```

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

So we can do:

```
In [1]: pd.merge(pg, player[['player_id', 'birth_date']],
                 on='player_id').head(5)
Out[1]:
           name    team   min  ...  player_rank  started  birth_date
0  D. Cheryshev  Russia  66.0  ...       0.0405    False    19901226
1  D. Cheryshev  Russia  74.0  ...       0.0162     True    19901226
2  D. Cheryshev  Russia  38.0  ...       0.0064     True    19901226
3  D. Cheryshev  Russia  59.0  ...      -0.0026    False    19901226
4  D. Cheryshev  Russia  66.0  ...      -0.0063     True    19901226
```

Again, this works because `player_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 `player_id` with the `on='player_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.

Merging is Precise

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

If your `name` column in one DataFrame has "N'Golo Kante" and the other has "NGolo Kante" or just "N.Kante", 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` and you have two `'C.Sanchez'` (both Columbia and Uraguay have Carlos Sanchez's on their teams), 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 passing and shooting statistic DataFrames:

```
In [2]: pass_df = pg[['match_id', 'player_id', 'pass', 'assist']]

In [3]: shot_df = pg[['match_id', 'player_id', 'shot', 'goal']]
```

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

```
In [4]: combined = pd.merge(pass_df, shot_df,
                            on=['match_id', 'player_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 `match_id`. Neither DataFrame had duplicates on these columns. That is, they each had *one* row with a match_id of `2057954` and player_id of `4513`, *one* row with a match_id of `2057954` and player_id of `41123`, 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* rushing and receiving stats at the player-game level.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
   match_id  player_id  pass  assist  shot  goal
0   2057954       4513    22       0     3     2
1   2057954      41123    26       0     0     0
2   2057954     101576    17       0     0     0
3   2057954     101583    26       0     0     0
4   2057954     101590     8       0     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      player_name  pos     team  birth_date
0      32793        A. N'Diaye  MID  Senegal    19900306
1         36  T. Alderweireld  DEF  Belgium    19890302
2         48    J. Vertonghen  DEF  Belgium    19870424
3         54        C. Eriksen  MID  Denmark    19920214
4         93  J. Guðmundsson  MID  Iceland    19901027
```

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 week 1 game, another time for week 2, 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', 'player_name', 'pos',
          'team']]).head()
Out[7]:
   match_id  player_id  pass  ...   player_name  pos    team
0   2057954       4513    22  ...  D. Cheryshev  MID  Russia
1   2057956       4513    28  ...  D. Cheryshev  MID  Russia
2   2057958       4513    10  ...  D. Cheryshev  MID  Russia
3   2058004       4513     6  ...  D. Cheryshev  MID  Russia
4   2058012       4513    18  ...  D. Cheryshev  MID  Russia
```

(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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Finally, although it's technically possible to do **many-to-many** (m:m) joins, in my experience this is almost always done unintentionally[3], 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 goal and assist data, keeping only observations that have at least one goal and assist respectively:

```
In [1]: goal_df = pg.loc[pg['goal'] > 0,
                         ['match_id', 'player_id', 'goal']]

In [2]: assist_df = pg.loc[pg['assist'] > 0,
                           ['match_id', 'player_id', 'assist']]

In [3]: goal_df.shape
Out[3]: (159, 3)

In [4]: assist_df.shape
Out[4]: (82, 3)
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Many players in the goal table (not all of them) will also have at least one assist. Some players in the assist table won't be in the goal table.

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

```
In [5]: comb_inner = pd.merge(goal_df, assist_df)

In [6]: comb_inner.shape
Out[6]: (10, 4)
```

So `comb_inner` *only* has the player-game's where the player had at least one goal *and* one assist. This happened 10 times in the 2018 World Cup.

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

```
In [7]: comb_left = pd.merge(goal_df, assist_df, how='left')

In [8]: comb_left.shape
Out[8]: (159, 4)
```

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 goal, whether they had any assists or not. Rows in `goal_df` that weren't in `assist_df` get missing values for `assist`.

```
In [9]: comb_left.head()
Out[9]:
   match_id  player_id  goal  assist
0   2057954       4513     2     NaN
1   2057954     101669     1     NaN
2   2057954     102157     1     NaN
3   2057954     122561     5     NaN
4   2057954     257800     1     2.0
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

In [11]: comb_outer.shape
Out[11]: (318, 5)

In [12]: comb_outer['_merge'].value_counts()
Out[12]:
left_only     236
right_only     72
both           10
```

This tells us that — out of the 318 player-game observations in our sample where a player scored either a goal *or* an assist, 236 were just a goal, 72 just an assist, and 10 times both.

So we know 10 times in the 2018 World Cup, a player scored a goal *and* got an assist. Out of curiosity, did the same player ever do this twice?

```
In [13]: comb_outer.query("_merge == 'both'")['player_id'].value_counts()
Out[13]:
25776     2
257800    1
69400     1
14836     1
20751     1
3682      1
8287      1
101590    1
14812     1
```

Yes. Who was it?

```
In [14]: player.query("player_id == 25776")[
                      ['player_name', 'pos', 'team']]
Out[14]:
    player_name  pos     team
454   W. Khazri  MID  Tunisia
```

Wahbi Khazri from Tunisia. Nice.

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

To demonstrate, let's modify the goal and assist tables we've been working with, renaming `player_id` to `scorer_id` and `passer_id` respectively.

```
In[1]: goal_df.columns = ['match_id', 'scorer_id', 'goal']

In[2]: assist_df.columns = ['match_id', 'passer_id', 'assist']
```

(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 `scorer_id` in `goals_df` and `passer_id` in `assist_df`. What to do? Simple, just use the `left_on` and `right_on` arguments instead of `on`.

```
In [3]:
pd.merge(goal_df, assist_df, left_on=['match_id', 'scorer_id'],
         right_on=['match_id', 'passer_id']).head()
Out[3]:
   match_id  scorer_id  goal  passer_id  assist
0   2057954     257800     1     257800       2
1   2057977      69400     1      69400       1
2   2057992      25776     1      25776       1
3   2057995      25776     1      25776       1
4   2057999      14836     1      14836       1
```

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 goals in a game.

```
In [4]:
max_goals = (goal_df
               .groupby('scorer_id')
               .agg(max_goals = ('goal', 'max')))
```

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

```
In [5]: max_goals.head()
Out[5]:
           max_goals
scorer_id
48                 1
122                1
123                1
261                1
3304               1
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [6]: pd.merge(goal_df, max_goals, left_on='scorer_id',
              right_index=True).head()
Out[6]:
      match_id  scorer_id  goal  max_goals
0      2057954       4513     2          2
56     2057956       4513     1          2
1290   2058004       4513     1          2
1503   2058012       4513     1          2
5      2057954     101669     1          1
```

`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 goal and assist DataFrames again, this time setting the index to `match_id` and `player_id`.

```
In [1]:
goal_df = (pg.loc[pg['goal'] > 0, ['match_id', 'player_id', 'goal']]
           .set_index(['match_id', 'player_id']))

In [2]:
assist_df = (pg.loc[pg['assist'] > 0, ['match_id', 'player_id', 'assist']]
             .set_index(['match_id', 'player_id']))
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [4]: goal_df.head()
Out[4]:
                    goal
match_id player_id
2057954  4513          2
         101669        1
         102157        1
         122561        5
         257800        1
```

And concatenating `goal_df` and `assist_df` gives us:

```
In [5]: pd.concat([goal_df, assist_df], axis=1).head()
Out[5]:
                    goal  assist
match_id player_id
2057954  4513        2.0     NaN
         101669      1.0     NaN
         102157      1.0     NaN
         122561      5.0     NaN
         257800      1.0     2.0
```

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 tackle DataFrame.

```
In [6]:
tackle_df = (pg.loc[pg['tackle'] > 0, ['match_id', 'player_id', 'tackle']]
            .set_index(['match_id', 'player_id']))
```

And want to concatenate all three at once:

```
In [7]: pd.concat([goal_df, assist_df, tackle_df], axis=1).head()
Out[7]:
                    goal  assist  tackle
match_id player_id
2057954  4513        2.0     NaN     NaN
         101669      1.0     NaN     NaN
         102157      1.0     NaN     1.0
         122561      5.0     NaN     NaN
         257800      1.0     2.0     1.0
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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]: mids = pm.loc[pm['pos'] == 'MID']

In [2]: fwds = pm.loc[pm['pos'] == 'FWD']
```

We can see these DataFrames are 637 and 384 rows respectively:

```
In [3]: mids.shape
Out[3]: (637, 31)

In [4]: fwds.shape
Out[4]: (384, 31)
```

Now let's use `pd.concat` (with its default `axis=0`) to stack these on top of each other. The resulting DataFrame should be `637 + 384 = 1021` rows.

```
In [5]: pd.concat([mids, fwds]).shape
Out[5]: (1021, 31)
```

Perfect.

In this case, we *know* `mids` and `fwds` 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]: mids_reset = mids.reset_index(drop=True)

In [7]: fwds_reset = fwds.reset_index(drop=True)
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [8]: mids_reset.head()
Out[8]:
           name    team   min  shot  ...  pos     side  started
0  D. Cheryshev  Russia  66.0     3  ...  MID     left    False
1    A. Dzagoev  Russia  24.0     0  ...  MID  central     True
2    A. Samedov  Russia  64.0     2  ...  MID    right     True
3    Y. Zhirkov  Russia  90.0     1  ...  MID     left     True
4  Y. Gazinskiy  Russia  90.0     1  ...  MID  central     True
```

And when we concatenate `mids_reset` and `fwds_reset` we get:

```
In [9]: pd.concat([mids_reset, fwds_reset]).sort_index().head()
Out[9]:
           name    team   min  shot  ...  pos     side  started
0  D. Cheryshev  Russia  66.0     3  ...  MID     left    False
0     A. Dzyuba  Russia  20.0     1  ...  FWD  central    False
1     F. Smolov  Russia  70.0     0  ...  FWD  central     True
1    A. Dzagoev  Russia  24.0     0  ...  MID  central     True
2    A. Samedov  Russia  64.0     2  ...  MID    right     True
```

Now our index has duplicates — one `0` from our midfielder DataFrame, another from the forward 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([mids_reset, fwds_reset],
             ignore_index=True).sort_index().head()
Out[10]:
           name    team   min  shot  ...  pos     side  started
0  D. Cheryshev  Russia  66.0     3  ...  MID     left    False
1    A. Dzagoev  Russia  24.0     0  ...  MID  central     True
2    A. Samedov  Russia  64.0     2  ...  MID    right     True
3    Y. Zhirkov  Russia  90.0     1  ...  MID     left     True
4  Y. Gazinskiy  Russia  90.0     1  ...  MID  central     True
```

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`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

`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'`)

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Exercises

3.5.1

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

They contain name (player and team), shot (shots and goals), pass (assists and crosses), and out of bound (throw ins and corners) data.

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

Note players are only in the shot data if they got had at least one shot or goal. Same with pass.csv and ob.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 have any shots (or passes, throw ins 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 group (A, B, C ...) 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 player goals per match. This data is at the player and match level, but we might want to include less granular data — information about the *player* (position, club team, height) or *match* (venue, weather).

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 position (usually) and height and match stay the same every match, and venue or weather is the same for every player playing in a given match. 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.
- One match table with date, venue, weather.
- Another player-match table with ONLY the things that vary at this level: shots, goals, passes, etc.

In each one we would want an id column — i.e. the player table would have a "player id", the match table a "match id", and the player-match table both player and match 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 incorrectly had the wrong venue for Russia vs Saudi Arabia, Group A, 6/14. If that information is in a single `match` 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, time left, right or left foot, whether it scored a goal). But imagine if we had to store every single thing we might care about — player height, weight, club team, 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, match, player-match 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

The other thing is SQL is good at is 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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[1]. Also, though SQL has some syntax for updating and creating data tables, I also usually handle writing to databases inside Pandas.

But SQL is good for loading (not necessarily modifying) and joining your raw, initial, normalized tables. Its also OK for filtering, e.g. if you want to only load RB data or just want to look at a certain week.

There are a few benefits to limiting SQL to the basics like this. One is that SQL dialects and commands can change depending on the database you're using (Postgres, SQLite, MS SQL, etc), but most of the basics 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`.

[1] One exception: SQL can be more memory efficient if your data is too large to load into Pandas, but that usually isn't a problem with the medium sized football datasets we'll be using.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

# handle directories
DATA_DIR = './data'

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

# load csv data
player_match = pd.read_csv(path.join(DATA_DIR, 'player_match.csv'))
player = pd.read_csv(path.join(DATA_DIR, 'players.csv'))
game = pd.read_csv(path.join(DATA_DIR, 'matches.csv'))
team = pd.read_csv(path.join(DATA_DIR, 'teams.csv'))

# and write it to sql
player_match.to_sql('player_match', 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.:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
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      player_name  pos  ... passport  team_id     team
0      32793       A. N'Diaye  MID  ...      686    19314  Senegal
1         36  T. Alderweireld  DEF  ...       56     5629  Belgium
2         48    J. Vertonghen  DEF  ...       56     5629  Belgium
3         54       C. Eriksen  MID  ...      208     7712  Denmark
4         93  J. Guðmundsson  MID  ...      352     7839  Iceland
```

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, player_name AS name, team, pos, foot
FROM player
```

```
Out [3]:
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
   player_id             name     team  pos   foot
0      32793       A. N'Diaye  Senegal  MID  right
1         36  T. Alderweireld  Belgium  DEF  right
2         48    J. Vertonghen  Belgium  DEF   left
3         54       C. Eriksen  Denmark  MID  right
4         93  J. Guðmundsson  Iceland  MID   left
```

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

Filtering

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

In [4]:

```
SELECT player_id, player_name AS name, pos, foot
FROM player
WHERE team = 'Japan'
```

Out [4]:

```
   player_id          name  pos   foot
0        703    M. Yoshida  DEF  right
1     101592      K. Honda  MID   left
2      37896  E. Kawashima  GKP  right
3      95010  H. Yamaguchi  MID  right
4      14730      T. Usami  FWD  right
```

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

Finally, notice we're filtering on `team` (i.e. we're choosing which rows to return depending on the value they have for `team`), even though it's not in our `SELECT` statement. That's fine. We could include it if we wanted (in which case we'd have a column `team` with `'Japan'` 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]:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
SELECT player_id, player_name AS name, team, pos, foot
FROM player
WHERE team = 'Japan' AND pos == 'MID'
```

Out [5]:

```
   player_id          name   team  pos   foot
0     101592      K. Honda  Japan  MID   left
1      95010  H. Yamaguchi  Japan  MID  right
2      14816     S. Kagawa  Japan  MID  right
3      14836       T. Inui  Japan  MID  right
4      14929     M. Hasebe  Japan  MID  right
```

In [6]:

```
SELECT player_id, player_name AS name, team, pos, foot
FROM player
WHERE team = 'Japan' OR pos == 'GKP'
```

Out [6]:

```
   player_id          name       team  pos   foot
0        703    M. Yoshida      Japan  DEF  right
1     101576   I. Akinfeev     Russia  GKP  right
2     101592      K. Honda      Japan  MID   left
3       3397  Arrizabalaga      Spain  GKP  right
4       3551  W. Caballero  Argentina  GKP  right
```

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

In [7]:

```
SELECT player_id, player_name AS name, pos, foot
FROM player
WHERE pos IN ('DEF', 'MID')
```

Out [7]:

```
   player_id             name  pos   foot
0      32793       A. N'Diaye  MID  right
1         36  T. Alderweireld  DEF  right
2         48    J. Vertonghen  DEF   left
3         54       C. Eriksen  MID  right
4         93  J. Guðmundsson  MID   left
```

SQL also allows negation:

In [8]:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
SELECT player_id, player_name AS name, team, pos, foot
FROM player
WHERE team NOT IN ('Japan', 'Iceland')
```

Out [8]:

```
   player_id             name     team  pos   foot
0      32793        A. N'Diaye  Senegal  MID  right
1         36  T. Alderweireld  Belgium  DEF  right
2         48    J. Vertonghen  Belgium  DEF   left
3         54        C. Eriksen  Denmark  MID  right
4        122        D. Mertens  Belgium  FWD  right
```

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 group they're in (in the `team` table).

We might try adding a new table to our `FROM` clause like this:

In [9]:

```
SELECT
    player.player_name AS name,
    player.pos,
    player.team,
    team.grouping
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 grouping
0  A. N'Diaye  MID  Senegal        F
1  A. N'Diaye  MID  Senegal        A
2  A. N'Diaye  MID  Senegal        F
3  A. N'Diaye  MID  Senegal        G
4  A. N'Diaye  MID  Senegal        E
5  A. N'Diaye  MID  Senegal        F
6  A. N'Diaye  MID  Senegal        B
7  A. N'Diaye  MID  Senegal        H
8  A. N'Diaye  MID  Senegal        G
9  A. N'Diaye  MID  Senegal        D
```

It's all Alfred N'Diaye (from Senegal), and he shows up in every group.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

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 (`A. N'Diaye`, `MID`, `Senegal`) matched up with the first team in the team table (`Korea`); then the second (`Russia`), and so on.

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

In [10]:

```
SELECT
    player.player_name as name,
    player.pos,
    player.team as player_team,
    team.team as team_team,
    team.grouping
FROM player, team
```

Out [10]:

```
               name  pos player_team team_team grouping
0        A. N'Diaye  MID     Senegal   Senegal        H
1   T. Alderweireld  DEF     Belgium   Belgium        G
2     J. Vertonghen  DEF     Belgium   Belgium        G
3        C. Eriksen  MID     Denmark   Denmark        C
4   J. Guðmundsson   MID     Iceland   Iceland        D
5        D. Mertens  FWD     Belgium   Belgium        G
6       O. Toivonen  FWD      Sweden    Sweden        F
7      K. El Ahmadi  MID     Morocco   Morocco        B
8       J. Guidetti  FWD      Sweden    Sweden        F
9        N. Amrabat  FWD     Morocco   Morocco        B
```

This makes it clear it's a cross join — every line for Alred N'Diaye (and also every other player once N'Diaye is done) is getting linked up with every team in the `team` table.

Since we have a 735 row player table and 32 row team table, that means we should get back `735*32 = 23520` rows. Sure enough:

```
In [11]: df.shape
Out[11]: (23520, 5)
```

What if we added a third table, say a `pos` table with four rows: one for MID, one for DEF, etc. In that case, each of these 23520 rows in the table above gets matched yet again with *each* of the four rows in the `position` table. This table would be `735*32*4 = 94080` rows.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

This is almost never what we want[2] (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 Alfred N'Diaye plays for Korea, so we have to specify a `WHERE` clause to filter and keep only the rows that make sense.

In [12]:

```
SELECT
    player.player_name as name,
    player.pos,
    player.team,
    team.grouping
FROM player, team
WHERE player.team = team.team
```

Out [12]:

```
              name  pos     team grouping
0       A. N'Diaye  MID  Senegal        H
1  T. Alderweireld  DEF  Belgium        G
2    J. Vertonghen  DEF  Belgium        G
3       C. Eriksen  MID  Denmark        C
4  J. Guðmundsson  MID  Iceland        D
```

Let's walk through it:

First, SQL is doing the full cross join (with 23520 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 32 separate rows for Alfred N'Diaye, to only the one row — where his team in the `player` table (`Senagal`) equals `Senegal` in the `team` table.

That gives us a table of 735 rows — the same number of players we originally started with — and includes the group info for each.

Again, adding in the `team` column from table team makes it more clear:

In [13]:

[2] Can you think of a time when it would be what you want? I can think of one: if you had a table of teams, and another of weeks 1-17 and wanted to generate a schedule so that every team had a line for every week. That's it though.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
SELECT
    player.player_name as name,
    player.pos,
    player.team as player_team,
    team.team as team_team,
    team.grouping
FROM player, team
WHERE player.team = team.team
```

Out [13]:

```
              name  pos player_team team_team grouping
0       A. N'Diaye  MID     Senegal   Senegal        H
1  T. Alderweireld  DEF     Belgium   Belgium        G
2    J. Vertonghen  DEF     Belgium   Belgium        G
3       C. Eriksen  MID     Denmark   Denmark        C
4  J. Guðmundsson  MID     Iceland   Iceland        D
5       D. Mertens  FWD     Belgium   Belgium        G
6      O. Toivonen  FWD      Sweden    Sweden        F
7     K. El Ahmadi  MID     Morocco   Morocco        B
8      J. Guidetti  FWD      Sweden    Sweden        F
9       N. Amrabat  FWD     Morocco   Morocco        B
```

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 [14]:

```
SELECT
    player.player_name as name,
    player.pos,
    team.team,
    team.city,
    team.grouping,
    player_match.*
FROM player, team, player_match
WHERE
    player.team = team.team AND
    player_match.player_id = player.player_id
```

Out [14]:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
              name  pos     team  ...     side player_rank started
0       A. N'Diaye  MID  Senegal  ...  central      0.0059       1
1       A. N'Diaye  MID  Senegal  ...  central      0.0035       1
2  T. Alderweireld  DEF  Belgium  ...    right      0.0046       1
3  T. Alderweireld  DEF  Belgium  ...    right      0.0052       1
4  T. Alderweireld  DEF  Belgium  ...    right      0.0032       1
```

This is doing the same as above (`player` + `team` tables) but also combining it with data from `player_match`. Again, if we had left off our `WHERE` clause, SQL would have done a full, three table `player*team*player_match` cross join. Alred N'Diaye would be matched with each team, then each of *those* rows would be matched with every row from the player-match table, giving us a `735*32*1671=39,301,920` row result.

Also note the `player_match.*` 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.player_name as name,
    p.pos,
    t.team,
    t.city,
    t.grouping,
    pm.match_id,
    pm.pass
FROM player AS p, team AS t, player_match AS pm
WHERE
    p.team = t.team AND
    pm.player_id = p.player_id
```

We just specify the full names once (in `FROM`), then add an alias with `AS`. Then in the `SELECT` and `WHERE` clauses we can use the alias instead.

Combing Joining and Other Filters

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```sql
SELECT
    p.player_name as name,
    p.pos,
    t.team,
    t.city,
    t.grouping,
    pm.match_id,
    pm.pass
FROM player AS p, team AS t, player_match AS pm
WHERE
    p.team = t.team AND
    pm.player_id = p.player_id AND
    p.pos == 'FWD'
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 referee combinations for the games:

In [15]:

```
SELECT DISTINCT ref, ref2
FROM game
```

Out [15]

```
      ref    ref2
0  378051  378038
1  378204  378144
2  384946  384962
3  380597  380580
4  378232  378231
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 multiple groups and the data I'm inheriting is in separate tables. I might do:

```
SELECT *
FROM player_data_group_a
UNION
SELECT *
FROM player_data_group_b
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 number of passes, shots and goals for every player and match in our database, whether or not the player actually played. We'll walk through it below:

```
1   SELECT a.*, b.min, b.pass, b.shot, b.goal
2   FROM
3       (SELECT match_id, label, home as team, away as opp, player_id,
4         player_name
5       FROM game, player
6       WHERE game.home = player.team_id
7       UNION
8       SELECT match_id, label, home as team, away as opp, player_id,
9         player_name
10      FROM game, player
11      WHERE game.away = player.team_id) AS a
12  LEFT JOIN player_match AS b ON a.match_id = b.match_id AND
13            a.player_id = b.player_id
```

Let's go through it. First, we need a full set of rows for every player. We do that in a subquery (lines 3-11) and call the resulting table `a`.

This subquery involves a `UNION`, let's look at the top part (lines 3-6).

```
SELECT match_id, label, home as team, away as opp, player_id, player_name
FROM game, player
WHERE game.home = player.team_id
```

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 Lionel Messi, Argentina vs Iceland, but there's a line for Lionel Messi, Korea vs Germany too.

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

The problem is our `match` table is by game, not team and game. If it were the latter, Argentina vs Iceland would have *two* lines, one for Argentina, one for Iceland. Instead it's just the one line, with Argentina in the `home` field, Iceland in `away`.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
    match_id                 date   home_team     away_team
...
55   2057972  2018-06-16 13:00:00   Argentina       Iceland
22   2057967  2018-06-16 16:00:00        Peru       Denmark
23   2057973  2018-06-16 19:00:00     Croatia       Nigeria
58   2057979  2018-06-17 12:00:00  Costa Rica        Serbia
34   2057984  2018-06-17 15:00:00     Germany        Mexico
29   2057978  2018-06-17 18:00:00      Brazil   Switzerland
```

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

That's what we're doing here:

```
SELECT match_id, label, home as team, away as opp, player_id, player_name
FROM game, player
WHERE game.home = player.team_id
UNION
SELECT match_id, label, home as team, away as opp, player_id, player_name
FROM game, player
WHERE game.away = player.team_id
```

Above the `UNION` gives us a line for every player's home games (so 2 per player), below a line for every away game. Stick them on top of each other and we have what we want.

It's all in a subquery, which we alias as `a`. So once you understand that, you can ignore it and mentally replace everything in parenthesis with `full_player_table` if you want.

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

```
    SELECT a.*, b.min, b.pass, b.shot, b.goal
    FROM full_player_table AS a
    LEFT JOIN player_match AS b ON a.match_id = b.match_id AND a.player_id
        = b.player_id
```

From there it's the standard left join syntax: `LEFT JOIN table_name ON` ...

Remember what we're going for: a line for every player, every game, whether the player played in the game or not. Messi played every game, so let's look at someone else, say Fyodor Kudryahov, from Russia:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [16]: df.query("player_id == 101647")
Out[16]:
      match_id                        label   team  ...  pass  shot goal
5      2057954  Russia - Saudi Arabia, 5 - 0  14358  ...   NaN   NaN  NaN
99     2057956         Russia - Egypt, 3 - 1  14358  ...   3.0   0.0  0.0
200    2057958       Uruguay - Russia, 3 - 0  15670  ...  38.0   0.0  0.0
2220   2058004     Spain - Russia, 1 - 1 (P)   1598  ...  12.0   0.0  0.0
2593   2058012   Russia - Croatia, 2 - 2 (P)  14358  ...  38.0   0.0  0.0
```

We can see he has row for each of Russia's game, even though he didn't play in their first vs Saudi Arabia. Perfect.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 at shots, goals, and passes at the player-match level. Make it only include players from grouping C. It should have the following columns:

`date`, `name`, `team`, `goal`, `shot`, `pass`

Rename `name` to `player` in your query.

4.2

b) Now modify your query to add in nationality from from `player` table. Use abbreviations too (`t` for the team table, `pm` for `player_match`, etc).

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 weather at every Premier league stadium and run it every day.

Second is when you need to grab a lot of data at once. Scrapers scale really well. You want stats for every Premier league match? Write a function that's flexible enough to get this data for one game. Then run it 380 times (38 matches * 20 teams \ 2 since every match 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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>Shots</th>
     <th>Goals</th>
    </tr>
    <tr>
     <td>Lionel Messi</td>
     <td>2018-06-16</td>
     <td>Argentina</td>
     <td>Iceland</td>
     <td>7</td>
     <td>0</td>
    </tr>
    <tr>
     <td>Luka Modric</td>
     <td>2018-06-21</td>
     <td>Croatia</td>
     <td>Argentina</td>
     <td>2</td>
     <td>1</td>
    </tr>
  </table>
<html>
```

Note columns (`td` and `th` elements) are always nested inside rows (`tr`).

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

If you were to save this code in an html file and open it in your browser you'd see:

Name	Date	Team	Opp	Shots	Goals
Lionel Messi	2018-06-16	Argentina	Iceland	7	0
Luka Modric	2018-06-21	Croatia	Argentina	2	1

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: from bs4 import BeautifulSoup as Soup

In [2]:
table_html = """
<html>
  <table>
    <tr>
     <th>Name</th>
     <th>Date</th>
     <th>Team</th>
     <th>Opp</th>
     <th>Shots</th>
     <th>Goals</th>
    </tr>
    <tr>
     <td>Lionel Messi</td>
     <td>2018-06-16</td>
     <td>Argentina</td>
     <td>Iceland</td>
     <td>7</td>
     <td>0</td>
    </tr>
    <tr>
     <td>Luka Modric</td>
     <td>2018-06-21</td>
     <td>Croatia</td>
     <td>Argentina</td>
     <td>2</td>
     <td>1</td>
    </tr>
  </table>
<html>
"""

In [3]: html_soup = Soup(table_html)
```

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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
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>Shots</th>
<th>Goals</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>Shots</th>
<th>Goals</th>
</tr>
...
<tr>
<td>Luka Modric</td>
<td>2018-06-21</td>
<td>Croatia</td>
<td>Argentina</td>
<td>2</td>
<td>1</td>
</tr>
</table>

In [9]: type(table_tag)
Out[9]: bs4.element.Tag
```

Or just the first `td` element.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [10]: td_tag = html_soup.find('td')

In [11]: td_tag
Out[11]: <td>Lionel Messi</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>Lionel Messi</td>
```

On simple tags, the key attribute is `string`. This returns the data inside.

```
In [14]: td_tag.string
Out[14]: 'Lionel Messi'
```

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]: 'Lionel Messi'
```

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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [16]: tr_tag.find_all('th')
Out[16]:
[<th>Name</th>,
 <th>Date</th>,
 <th>Team</th>,
 <th>Opp</th>,
 <th>Shots</th>,
 <th>Goals</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', 'Shots', 'Goals']
```

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>Lionel Messi</td>,
 <td>2018-06-16</td>,
 <td>Argentina</td>,
 <td>Iceland</td>,
 <td>7</td>,
 <td>0</td>,
 <td>Luka Modric</td>,
 <td>2018-06-21</td>,
 <td>Croatia</td>,
 <td>Argentina</td>,
 <td>2</td>,
 <td>1</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 Messi *and* Modric (because

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

they're both in the table), calling `tr_tag.find_all('td')` on just the last, single row returns data for Modric 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>Lionel Messi</td>,
 <td>2018-06-16</td>,
 <td>Argentina</td>,
 <td>Iceland</td>,
 <td>7</td>,
 <td>0</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>Shots</th>,
 <th>Goals</th>,
 <td>Lionel Messi</td>,
 <td>2018-06-16</td>,
 <td>Argentina</td>,
 <td>Iceland</td>,
 <td>7</td>,
 <td>0</td>,
 <td>Luka Modric</td>,
 <td>2018-06-21</td>,
 <td>Croatia</td>,
 <td>Argentina</td>,
 <td>2</td>,
 <td>1</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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [25]: [str(x.string) for x in all_td_tags]
Out[25]:
['Lionel Messi',
 '2018-06-16',
 'Argentina',
 'Iceland',
 '7',
 '0',
 'Luka Modric',
 '2018-06-21',
 'Croatia',
 'Argentina',
 '2',
 '1']
```

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>Lionel Messi</td>,
  <td>2018-06-16</td>,
  <td>Argentina</td>,
  <td>Iceland</td>,
  <td>7</td>,
  <td>0</td>],
 [<td>Luka Modric</td>,
  <td>2018-06-21</td>,
  <td>Croatia</td>,
  <td>Argentina</td>,
  <td>2</td>,
  <td>1</td>]]
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

World Football - Web Scraping Example

Note the examples for this section are in the file `05_02_wfb.py`. We'll pick up from the top of the file.

Let's build a scraper to get some statistics from worldfootball.net. Let's get the data from this page:

https://www.worldfootball.net/schedule/eng-premier-league-2022-2023-spieltag/38/

Which has the British Premier League results for the 2022-2023 season. Specifically, we want this part:

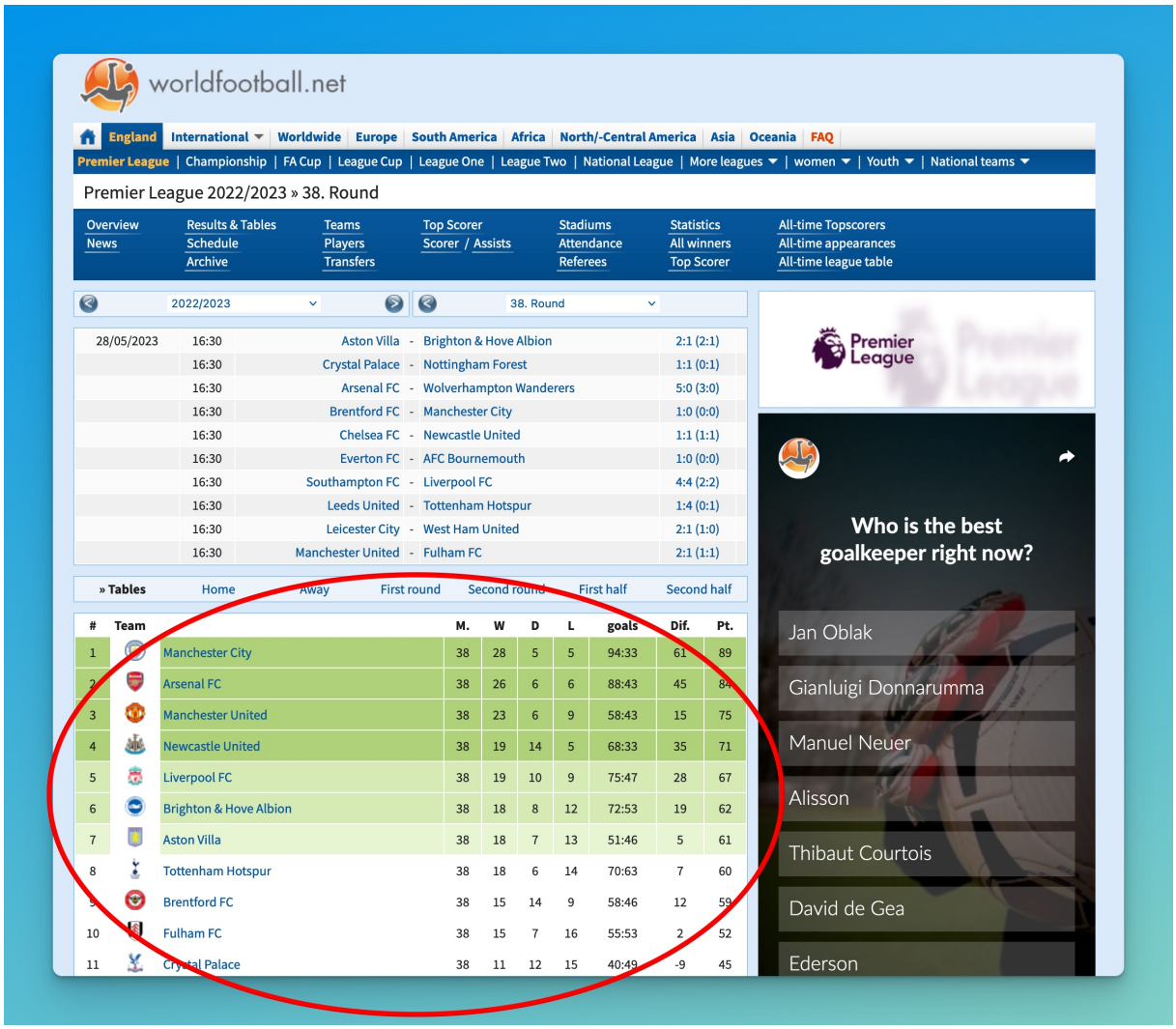

Figure 0.2: World Football

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

We'll start with the imports:

```
from bs4 import BeautifulSoup as Soup
import requests
from pandas import DataFrame
```

Besides BeautifulSoup, we're also importing the `requests` library, which we'll use to programatically visit MFF. 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('https://www.myfootballfacts.com/premier-
    league/all-time-premier-league/premier_league_goal_statistics/')
```

Let's look at (part of) this HTML.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [2] print(response.text)
Out[2]:
...
<table class="standard_tabelle" cellpadding="3" cellspacing="1">
<tr>
<th align="center">#</th>
<th colspan="2">Team</th>
<th align="center">M.</th>
<th align="center">W</th>
<th align="center">D</th> <th align="center">L</th>
<th align="center">goals</th>
<th align="center">Dif.</th>
<th align="center">Pt.</th>
</tr>
<tr>
<td bgcolor="#AFD179" align="center">1</td>
<td bgcolor="#FFFFFF" align="center">
<img src="..." width="20" height="20" valign="center" alt="Manchester City
    , England" title="Manchester City, England" />
</td>
<td bgcolor="#AFD179">
<a href="/teams/manchester-city/" title="Manchester City">Manchester City
    </a>
</td>
<td bgcolor="#AFD179" align="center">38</td>
<td bgcolor="#AFD179" align="center">28</td>
<td bgcolor="#AFD179" align="center">5</td> <td bgcolor="#AFD179" align="
    center">5</td>
<td bgcolor="#AFD179" align="center">94:33</td>
<td bgcolor="#AFD179" align="center">61</td>
<td bgcolor="#AFD179" align="center">89</td>
</tr>
<tr>
<td bgcolor="#AFD179" align="center">2</td>
<td bgcolor="#FFFFFF" align="center">
<img src="..." width="20" height="20" valign="center" alt="Arsenal FC,
    England" title="Arsenal FC, England" />
</td>
<td bgcolor="#AFD179">
<a href="/teams/arsenal-fc/" title="Arsenal FC">Arsenal FC</a>
</td>
<td bgcolor="#AFD179" align="center">38</td>
<td bgcolor="#AFD179" align="center">26</td>
<td bgcolor="#AFD179" align="center">6</td> <td bgcolor="#AFD179" align="
    center">6</td>
<td bgcolor="#AFD179" align="center">88:43</td>
<td bgcolor="#AFD179" align="center">45</td>
<td bgcolor="#AFD179" align="center">84</td>
</tr>
<tr>
...
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 more than 11 thousand lines — but you get the picture.

Now let's parse it:

```
In [3]: soup = Soup(response.text)
```

Remember, we can treat this top level `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 = 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]: 6
```

There's six, but looking at them briefly in the REPL (printing out `tables[0]`, `tables[1]`, etc), it's clear we want this one:

```
In [5]: results_table = tables[3]
```

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).

`results_table` is still a nested tag, so let's run another `find_all`:

```
In [6]: rows = results_table.find_all('tr')
```

This gives us a list of all the `tr` tags inside our table. We can see the header row is the first one:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [7]: rows[0]
Out[7]:
<tr>
<th align="center">#</th>
<th colspan="2">Team</th>
<th align="center">M.</th>
<th align="center">W</th>
<th align="center">D</th> <th align="center">L</th>
<th align="center">goals</th>
<th align="center">Dif.</th>
<th align="center">Pt.</th>
</tr>
```

That'll be useful later for knowing what columns are what.

Now how about some data.

```
In [8]: first_data_row = rows[1]

In [9]: first_data_row
Out[9]:
<tr>
<td align="center" bgcolor="#AFD179">1</td>
<td align="center" bgcolor="#FFFFFF">
<img alt="Manchester City, England" height="20" src="..." title="
    Manchester City, England" valign="center" width="20"/>
</td>
<td bgcolor="#AFD179">
<a href="/teams/manchester-city/" title="Manchester City">Manchester City
    </a>
</td>
<td align="center" bgcolor="#AFD179">38</td>
<td align="center" bgcolor="#AFD179">28</td>
<td align="center" bgcolor="#AFD179">5</td> <td align="center" bgcolor="#
    AFD179">5</td>
<td align="center" bgcolor="#AFD179">94:33</td>
<td align="center" bgcolor="#AFD179">61</td>
<td align="center" bgcolor="#AFD179">89</td>
</tr>
```

It's the first row — stats for Man City, which was the best team in the 2022-2023 season. Nice. Note this is *still* a nested tag, so we need to use `find_all` again. The end is in site though.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [10]: first_data_row.find_all('td')
Out[10]:
[<td align="center" bgcolor="#AFD179">1</td>,
 <td align="center" bgcolor="#FFFFFF">
 <img alt="Manchester City, England" height="20" title="Manchester City,
    England" valign="center" width="20"/>
 </td>,
 <td bgcolor="#AFD179">
 <a href="/teams/manchester-city/" title="Manchester City">Manchester City
    </a>
 </td>,
 <td align="center" bgcolor="#AFD179">38</td>,
 <td align="center" bgcolor="#AFD179">28</td>,
 <td align="center" bgcolor="#AFD179">5</td>,
 <td align="center" bgcolor="#AFD179">5</td>,
 <td align="center" bgcolor="#AFD179">94:33</td>,
 <td align="center" bgcolor="#AFD179">61</td>,
 <td align="center" bgcolor="#AFD179">89</td>]
```

This returns a list of `td` tags. Now we can call `str` on them to get the data out:

```
In [11]: [str(x.string) for x in first_data_row.find_all('td')]
Out[11]: ['1', 'None', 'None', '38', '28', '5', '5', '94:33', '61', '89']
```

Note this is pretty good, but it's showing `None` in two spots instead of the team name. Looking at the website, the first `None` is the team logo. Makes sense we can't get it in data form. But the second should be the name. Looking at it specifically, it's this:

```
In [12]: first_data_row.find_all('td')[2]
Out[12]:
<td bgcolor="#AFD179">
<a href="/teams/manchester-city/" title="Manchester City">Manchester City
    </a>
</td>
```

The problem is this one is *still* a nested tag — it has the link tag (`<a>`) inside of it. If we call `find('a')` on it, *then* `.string`, we can get it out.

```
In [13]: first_data_row.find_all('td')[2].find('a').string
Out[13]: 'Manchester City'
```

Let's write a function that can handle this — it'll return the basic `string` if there, but also work on nested tags with links in them too. Something like this:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
def string_from_simple_or_a(td):
    # first try to find the string
    # if it exists, just return that
    simple_string = td.string
    if simple_string is not None:
        return str(simple_string)

    else:
        # if there is no string, try to find a link
        a_tag = td.find('a')

        # if that exists, return that - otherwise return None
        if a_tag is not None:
            return str(a_tag.string)
        else:
            return None
```

Now we can run this on every `td` tag in our `first_data_row`:

```
In [14]: [string_from_simple_or_a(x) for x in first_data_row.find_all('td'
    )]
Out[14]: ['1', None, 'Manchester City', '38', '28', '5', '5', '94:33', '61
    ', '89']
```

Perfect.

Now that we've got this working, let's put it inside a function that will work on any row.

```
def parse_row(row):
    """
    Take in a tr tag and get the data out of it in the form of a list of
    strings.
    """
    return [string_from_simple_or_a(x) for x in row.find_all('td')]
```

We have to apply `parse_row` to each row in our data. Since the first row is a header, our data is: `rows[3:]`.

```
In [15]: list_of_parsed_rows = [parse_row(row) for row in rows[1:]]
```

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 `lists_of_parsed_rows` and seeing what happens:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [16]: df = DataFrame(list_of_parsed_rows)

In [17]: df
Out[17]:
     0     1                        2   3   4   5   6      7    8   9
0    1  None          Manchester City  38  28   5   5  94:33   61  89
1    2  None                Arsenal FC  38  26   6   6  88:43   45  84
2    3  None        Manchester United  38  23   6   9  58:43   15  75
3    4  None         Newcastle United  38  19  14   5  68:33   35  71
4    5  None              Liverpool FC  38  19  10   9  75:47   28  67
5    6  None   Brighton & Hove Albion  38  18   8  12  72:53   19  62
6    7  None               Aston Villa  38  18   7  13  51:46    5  61
7    8  None        Tottenham Hotspur  38  18   6  14  70:63    7  60
8    9  None              Brentford FC  38  15  14   9  58:46   12  59
9   10  None                 Fulham FC  38  15   7  16  55:53    2  52
10  11  None            Crystal Palace  38  11  12  15  40:49   -9  45
11  12  None                Chelsea FC  38  11  11  16  38:47   -9  44
12  13  None  Wolverhampton Wanderers  38  11   8  19  31:58  -27  41
13  14  None          West Ham United  38  11   7  20  42:55  -13  40
14  15  None           AFC Bournemouth  38  11   6  21  37:71  -34  39
15  16  None        Nottingham Forest  38   9  11  18  38:68  -30  38
16  17  None                Everton FC  38   8  12  18  34:57  -23  36
17  18  None            Leicester City  38   9   7  22  51:68  -17  34
18  19  None              Leeds United  38   7  10  21  48:78  -30  31
19  20  None            Southampton FC  38   6   7  25  36:73  -37  25
```

Almost what we want, just a few minor issues. First, it doesn't have column names. Let's fix that. We could parse them, but it's easiest to just name them what we want.

```
In [18]:
df.columns = ['standings', 'logo', 'team', 'matches', 'wins', 'draws',
              'losses', 'goals', 'diff', 'points']

In [19]: df.head()
Out[19]:
  standings  logo               team matches ...  goals diff points
0         1  None    Manchester City      38 ...  94:33   61     89
1         2  None         Arsenal FC      38 ...  88:43   45     84
2         3  None  Manchester United      38 ...  58:43   15     75
3         4  None   Newcastle United      38 ...  68:33   35     71
4         5  None       Liverpool FC      38 ...  75:47   28     67
```

Next, the logo column isn't doing anything — we're not going to show the logo in our data. So let's drop it. We could run `df.drop('logo', axis=1)`. That'd work the first time we ran it, but if for some reason we run more times it'll throw an error trying to drop a column that no longer exists.

Instead we can do:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [20]: df = df[[x for x in df.columns if x != 'logo']]

In [21]: df.head()
Out[21]:
  standings               team matches ... losses  goals diff points
0         1    Manchester City      38 ...      5  94:33   61     89
1         2         Arsenal FC      38 ...      6  88:43   45     84
2         3  Manchester United      38 ...      9  58:43   15     75
3         4   Newcastle United      38 ...      5  68:33   35     71
4         5       Liverpool FC      38 ...      9  75:47   28     67
```

The other thing is the goals column is formatted sort of weird, e.g. `94:33` for (`scored:allowed`). Let's put those in two separate columns:

```
In [22]: df['goals_for'] = df['goals'].str[:2]

In [23]: df['goals_against'] = df['goals'].str[3:]

In [24]: df.head()
Out[24]:
  standings               team matches  ... points goals_for goals_against
0         1    Manchester City      38  ...     89        94            33
1         2         Arsenal FC      38  ...     84        88            43
2         3  Manchester United      38  ...     75        58            43
3         4   Newcastle United      38  ...     71        68            33
4         5       Liverpool FC      38  ...     67        75            47
```

note the bottom two rows — they're summary rows, not really data we want. So let's get rid of them:

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. In this case all our numbers are integers (round, whole numbers).

```
In [25]:
int_cols = ['standings', 'matches', 'wins', 'draws', 'losses', 'diff',
            'points', 'goals_for', 'goals_against']

In [26]: df[int_cols] = df[int_cols].astype(int)
```

And we're done. Taking a look at our data:

```
In [27]: df.head()
Out[27]:
   standings               team  matches  ...  goals_for  goals_against
0          1    Manchester City       38  ...         94             33
1          2         Arsenal FC       38  ...         88             43
2          3  Manchester United       38  ...         58             43
3          4   Newcastle United       38  ...         68             33
4          5       Liverpool FC       38  ...         75             47
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 it's URL. Everyone is familiar with the most basic urls, e.g.

`www.fantasymath.com`,

But urls can have extra stuff too, e.g.

`api.fantasymath.com/v2/players-comp/?player=lionel-messi&player=cristiano-ronaldo`

At a very basic level: a web API lets you specifically manipulate the URL (e.g. maybe you put in `neymar` instead of `cristiano-ronaldo`) and get data back. It does this via the same mechanisms (HTTP) that regular, non-API websites when you go to some given URL.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 an American football API I built, is only called by the `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 football public 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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": ["lionel-messi", "luka-modric"],
    "positions": ["fwd", "mid"],
    "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": ["lionel-messi", "luka-modric",
    "positions": ["fwd", "mid"],
    "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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 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. Lionel Messi, 2012), write a function that gets what we need out of it, then apply the function to the entire collection (all of Messi's other years, every other player).

We'll work through a few examples next.

Fantasy Premier League API

The Fantasy Premier League website at https://premierleague.com has a good API that's public. Note, this isn't a FPL book; I don't play it myself (though writing this book makes me want to).

But, it's a good, free example of a football API, and has a lot of general data. We'll stick to more general, non-fantasy specific portions of the endpoint.

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?

Answer is literally google, "premier league api". Everything we need is a few results down, a very helpful writeup by Frenzel Timothy on Medium:

https://medium.com/@frenzelts/fantasy-premier-league-api-endpoints-a-detailed-guide-acbd5598eb19

Football Data API Walkthrough #1

Let's walk through the process of working with this API, starting with authentication.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Authentication

The easiest way to check authentication is to try out a few URLs and see if you can get data back. On Tim's documentation, the first link he lists:

```
https://fantasy.premierleague.com/api/bootstrap-static/
```

Putting it in the browser we can it returns data, 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 Premier League team?

According to Tim, the endpoint we've just found looks like it has some info on that. Again:

```
https://fantasy.premierleague.com/api/bootstrap-static/
```

Let's try it.

2. Visit endpoint in browser

Try copying that URL and putting it in the browser.

What you see will depend on what browser you're using. In FireFox, I get nicely formatted data:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
JSON    Raw Data    Headers
Save  Copy  Collapse All  Expand All (slow)  Filter JSON
▸ events:                    […]
▸ game_settings:             {…}
▸ phases:                    […]
▾ teams:
  ▾ 0:
      code:                  3
      draw:                  0
      form:                  null
      id:                    1
      loss:                  0
      name:                  "Arsenal"
      played:                0
      points:                0
      position:              0
      short_name:            "ARS"
      strength:              4
      team_division:         null
      unavailable:           false
      win:                   0
      strength_overall_home: 1200
      strength_overall_away: 1270
      strength_attack_home:  1150
      strength_attack_away:  1210
      strength_defence_home: 1190
      strength_defence_away: 1220
      pulse_id:              1
  ▸ 1:                       {…}
```


Figure 0.3: Formatted JSON

Sometimes you might see a data dump that looks something like this:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
{"events":[{"id":1,"name":"Gameweek
1","deadline_time":"2022-08-05T17:30:00Z","average_entry_score":0,"
    finished":false,"data_checked":false,"highest_scoring_entry":null,"
    deadline_time_epoch":1659720600,"deadline_time_game_offset":0,"
    highest_score":null,"is_previous":false,"is_current":false,"is_next":
    true,"cup_leagues_created":false,"h2h_ko_matches_created":false,"
    chip_plays":[{"chip_name":"bboost","num_played":9775},{"chip_name":"3xc
    ","num_played":20173}],"most_selected":null,"most_transferred_in":null,
    "top_element":null,"top_element_info":null,"transfers_made":0,"
    most_captained":null,"most_vice_captained":null},{"id":2,"name":"
    Gameweek
2","deadline_time":"2022-08-13T10:00:00Z","average_entry_score":0,"
    finished":false,"data_checked":false,"highest_scoring_entry":null,"
    deadline_time_epoch":1660384800,"deadline_time_game_offset":0,"
    highest_score":null,"is_previous":false,"is_current":false,"is_next":
    false,"cup_leagues_created":false,"h2h_ko_matches_created":false,"
    chip_plays":[{"chip_name":"bboost","num_played":95038},{"chip_name":"
    freehit","num_played":102410},{"chip_name":"wildcard","num_played"
    :277209},{"chip_name":"3xc","num_played":269514}],"most_selected":277,"
    most_transferred_in":272,"top_element":142,"top_element_info":{"id"
    :142,"points":18},"transfers_made":12038724,"most_captained":233,"
    most_vice_captained":277},{"id":3,"name":"Gameweek
3","deadline_time":"2022-08-20T10:00:00Z","average_entry_score":0,"
    finished":false,"data_checked":false,"highest_scoring_entry":null,"
    deadline_time_epoch":1660989600,"deadline_time_game_offset":0,"
    highest_score":null,"is_previous":false,"is_current":false,"is_next":
    false,"cup_leagues_created":false,"h2h_ko_matches_created":false,"
    chip_plays":[{"chip_name":"bboost","num_played":94049},{"chip_name":"
    freehit","num_played":117627},{"chip_name":"wildcard","num_played"
    :372083},{"chip_name":"3xc","num_played":138714}],"most_selected":277,"
    most_transferred_in":419,"top_element":null,"top_element_info":null,"
    transfers_made":15553648,"most_captained":277,"most_vice_captained"
    :277},{"id":4,"name":"Gameweek
```


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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 a bunch of fields — `events`, `game_settings`, `phases`, `teams`, etc. The `teams` field has what we want.

Let's get it in Python:

3. Get what you need in Python

Note the code for this section is in `./code/05_03_fpl.py`. We'll pick up after the imports.

The endpoint we're working with:

```
In [1]: fpl_url = 'https://fantasy.premierleague.com/api/bootstrap-static/
    '
```

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]: fpl_resp = requests.get(fpl_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]: fpl_json = fpl_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 FPL 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 it's the book, I've saved a snapshot of the data that matches up what we show here. To use it uncomment and run the following in `05_03_fpl.py`:

```
In [4]:
with open('./data/json/fpl.json') as f:
    fpl_json = json.load(f)
```

This will load the saved FPL JSON 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 `fpl_json`, we can see it gives us a massive amount of data, more than we could ever view in the REPL. Here's the last few lines:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [5]: fpl_json
Out[5]:
 ...
  {'id': 3,
   'plural_name': 'Midfielders',
   'plural_name_short': 'MID',
   'singular_name': 'Midfielder',
   'singular_name_short': 'MID',
   'squad_select': 5,
   'squad_min_play': 2,
   'squad_max_play': 5,
   'ui_shirt_specific': False,
   'sub_positions_locked': [],
   'element_count': 221},
  {'id': 4,
   'plural_name': 'Forwards',
   'plural_name_short': 'FWD',
   'singular_name': 'Forward',
   'singular_name_short': 'FWD',
   'squad_select': 3,
   'squad_min_play': 1,
   'squad_max_play': 3,
   'ui_shirt_specific': False,
   'sub_positions_locked': [],
   'element_count': 57}]}
```

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]: fpl_json.keys()
Out[6]: dict_keys(['events', 'game_settings', 'phases', 'teams',
                   'total_players', 'elements', 'element_stats',
                   'element_types'])
```

4. Get everything into Pandas

Again, our goal with any type of data in Python should always 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.

In this case this step is super easy, because the `teams` field *already* gives a list of identically structured dicts.

```
In [7]: type(fpl_json['teams'])
Out[7]: list
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Here's what the first one looks like:

```
In [8]: fpl_json['teams'][0]
Out[8]:
{'code': 3,
 'draw': 0,
 'form': None,
 'id': 1,
 'loss': 0,
 'name': 'Arsenal',
 'played': 0,
 'points': 0,
 'position': 0,
 'short_name': 'ARS',
 'strength': 4,
 'team_division': None,
 'unavailable': False,
 'win': 0,
 'strength_overall_home': 1200,
 'strength_overall_away': 1270,
 'strength_attack_home': 1150,
 'strength_attack_away': 1210,
 'strength_defence_home': 1190,
 'strength_defence_away': 1220,
 'pulse_id': 1}
```

So all we have to do is pass `fpl_json['teams']` to `DataFrame` and it'll work:

```
In [8]: df_teams = DataFrame(fpl_json['teams'])

In [9]: df_teams.head()
Out[9]:
          name short_name  id  ...  strength_defence_away pulse_id
0      Arsenal        ARS   1  ...                   1220        1
1  Aston Villa        AVL   2  ...                   1090        2
2    Brentford        BRE   3  ...                   1120      130
3     Brighton        BHA   4  ...                   1120      131
4      Burnley        BUR   5  ...                   1100       43
```

And there we go. We got Premier League team data from the FPL API.

Match Data

Let's do another one. How about some game (aka match or fixture) data. Let's run through the four steps:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

1. Find an endpoint

Looking at Frenzel's writeup, it looks like we can get match level data at the fixtures endpoint:

https://fantasy.premierleague.com/api/fixtures/

2. Visit endpoint in browser

Let's look at this end point in our browser. We can see it's a giant list of matches, each with basic information: home, away teams, time, results, etc.

The only non-standard part is in the `stats` field, which contains information on players ("elements") and the various stat categories.

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]:
match_url = 'https://fantasy.premierleague.com/api/fixtures/'
match_resp = requests.get(match_url)
match_json = match_resp.json()
```

(Again, if you want make sure you're using the same data as me, uncomment the `json.load` line in `05_03_fpl.py`.)

We know from the browser that it's a giant list of matches, and that's it's all pretty standard. Here's a single match:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [2]: match0 = match_json[0]

In [3]: match0
Out[3]:
{'code': 2292810,
 'event': 1,
 'finished': False,
 'finished_provisional': False,
 'id': 1,
 'kickoff_time': '2022-08-05T19:00:00Z',
 'minutes': 0,
 'provisional_start_time': False,
 'started': False,
 'team_a': 1,
 'team_a_score': None,
 'team_h': 7,
 'team_h_score': None,
 'stats': [],
 'team_h_difficulty': 3,
 'team_a_difficulty': 2,
 'pulse_id': 74911}
```

4. Get it into Pandas

A DataFrame needs a list of flat dictionaries, which — apart from the `stats` field — is what we have. So we could ignore `stats` and do:

```
In [4]: match_cols = [key for key in match0 if key != 'stats']

In [5]: df_match = DataFrame(match_json)[match_cols]
```

And this would give us a good `match` level DataFrame.

```
In [6]:
df_match[['id', 'team_a', 'team_h', 'team_a_score', 'team_h_score',
    'kickoff_time']].head()
--
Out[6]:
   id  team_a  team_h team_a_score team_h_score          kickoff_time
0   1       1       7         None         None  2022-08-05T19:00:00Z
1   4      12       9         None         None  2022-08-06T11:30:00Z
2   2       2       3         None         None  2022-08-06T14:00:00Z
3   5      20      11         None         None  2022-08-06T14:00:00Z
4   7      16      15         None         None  2022-08-06T14:00:00Z
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Player Data

There are other interesting endpoints, and you can play with them, but the only other thing we'll for sure want is some player information. Right now we just have element — apparently the FPL API word for "player ID" — but we have no information on which player is who, position etc.

So let's get that quick. Looking at this, it looks like all that data is in our original bootstrap-static endpoint:

https://fantasy.premierleague.com/api/bootstrap-static/

If you've been following along, you should still have `fpl_json` loaded in the REPL, but if not you'll have to go back and load it again

We already turned the `teams` field of this data into a DataFrame, now for `elements`.

Looking at a specific example:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: fpl_json['elements'][0]
Out[1]:
{'chance_of_playing_next_round': None,
 'chance_of_playing_this_round': None,
 'code': 58822,
 'cost_change_event': 0,
 'cost_change_event_fall': 0,
 'cost_change_start': 0,
 'cost_change_start_fall': 0,
 'dreamteam_count': 0,
 'element_type': 2,
 'ep_next': '2.3',
 'ep_this': None,
 'event_points': 0,
 'first_name': 'Cedric',
 'form': '0.0',
 'id': 1,
 'in_dreamteam': False,
 'news': '',
 'news_added': None,
 'now_cost': 45,
 'photo': '58822.jpg',
 'points_per_game': '2.3',
 'second_name': 'Alves Soares',
 ...
 'goals_scored': 1,
 'assists': 1,
 'clean_sheets': 3,
 'goals_conceded': 27,
 'own_goals': 0,
 'penalties_saved': 0,
 'penalties_missed': 0,
 'yellow_cards': 3,
 'red_cards': 0,
 'saves': 0,
 'bonus': 3,
 'bps': 292,
 'influence': '318.4',
 'creativity': '327.1',
 'threat': '111.0',
 'ict_index': '75.8',
 'influence_rank': 202,
 'influence_rank_type': 79,
 'creativity_rank': 112,
 'creativity_rank_type': 24,
 'threat_rank': 235,
 'threat_rank_type': 78,
 'ict_index_rank': 195,
 'ict_index_rank_type': 61,
 'corners_and_indirect_freekicks_order': 2,
 'corners_and_indirect_freekicks_text': '',
 'direct_freekicks_order': 3,
 'direct_freekicks_text': '',
 'penalties_order': None,
 'penalties_text': ''}
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

This is a lot of data (which I truncated) but if we look closely at the first and second name fields we can see this is data on Cedric Alves Soares, right back for Arsenal. The dict looks normal — no fields are nested dictionaries or lists or anything — which means again we can pass this straight to DataFrame.

```
In [3]: df_players = DataFrame(fpl_json['elements'])

In [4]: df_players.sample(5)
Out[4]:
       first_name       second_name  team  ... yellow_cards red_cards
465        Harvey             White    18  ...            0         0
54          Jaden  Philogene-Bidace     2  ...            0         0
237         Youri         Tielemans    10  ...            3         0
449  Pierre-Emile          Højbjerg    18  ...            3         0
327          Cole            Palmer    13  ...            0         0
```

Great.

After getting this data, we can store it 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 FPL 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) that it is 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Distributions

A distribution is a way to convey the *frequency* of outcomes for some variable.

For example, take our player-match data. How many passes did each player have per per match?

Let's take, say 200, random player-matches with number of passes from the 2018 World Cup. Here are the first 25:

```
           name       team            opp   min  pos  pass
L. Augustinsson     Sweden        England  90.0  DEF    40
      G. Torres     Panama        Belgium  27.0  FWD     7
   B. Bjarnason    Iceland        Nigeria  90.0  MID    20
     P. Jansson     Sweden  Korea Republic  90.0  DEF    47
  Marco Asensio      Spain           Iran  10.0  MID     9
      R. Varane     France        Belgium  90.0  DEF    38
      E. Hazard    Belgium          Japan  90.0  FWD    55
     I. Kutepov     Russia          Egypt  90.0  DEF    18
 B. Sigurðarson    Iceland        Croatia  19.0  FWD     4
     I. šćPerii    Croatia      Argentina  80.0  MID    13
     M. Dembélé    Belgium        England  12.0  MID    11
     V. Ćorluka    Croatia      Argentina   NaN  DEF     2
 Renato Augusto     Brazil    Switzerland  24.0  MID    14
    L. Piszczek     Poland        Senegal  83.0  DEF    42
     T. Meunier    Belgium         Panama  90.0  DEF    38
       F. Baloy     Panama        England  21.0  DEF    17
        Y. Mina   Colombia         Poland  90.0  DEF    36
     M. Yoshida      Japan       Colombia  90.0  DEF    93
      K. Mbappé     France        Denmark  14.0  FWD    11
  J. Mascherano  Argentina        Nigeria  90.0  DEF    78
     D. Mertens    Belgium         France  30.0  FWD    18
    Ahmed Fathy      Egypt        Uruguay  90.0  DEF    41
   J. Hernández     Mexico         Sweden  90.0  FWD    20
      Trézéguet      Egypt   Saudi Arabia  81.0  MID    20
     Iago Aspas      Spain         Russia  40.0  FWD    17
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Let's arrange these from smallest to largest, marking each with an x and stacking x's when a number shows up multiple times. Make sense? Like this:

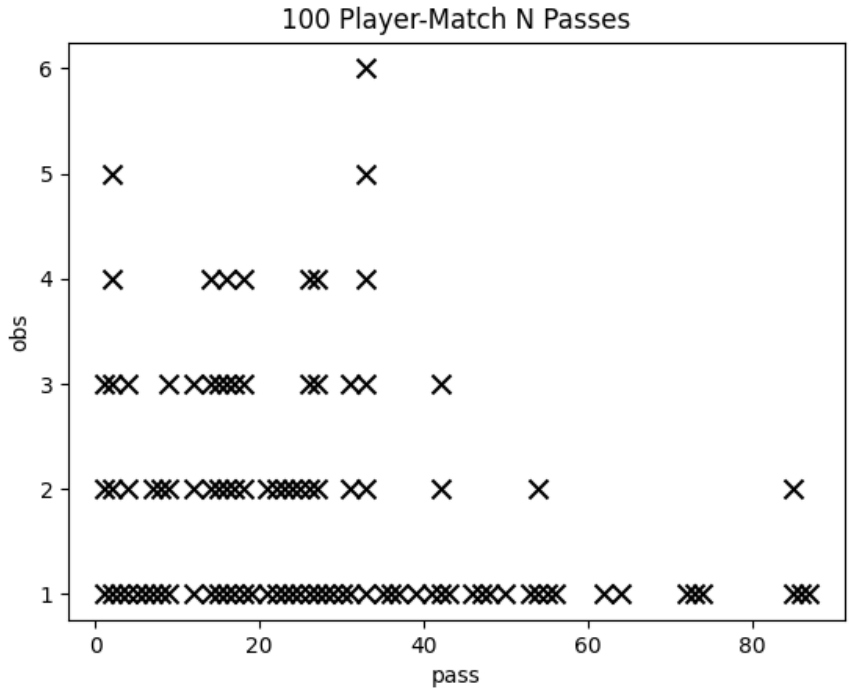

Figure 0.1: 100 Player-Match N Passes

Interesting, let's up it to 500 observations:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

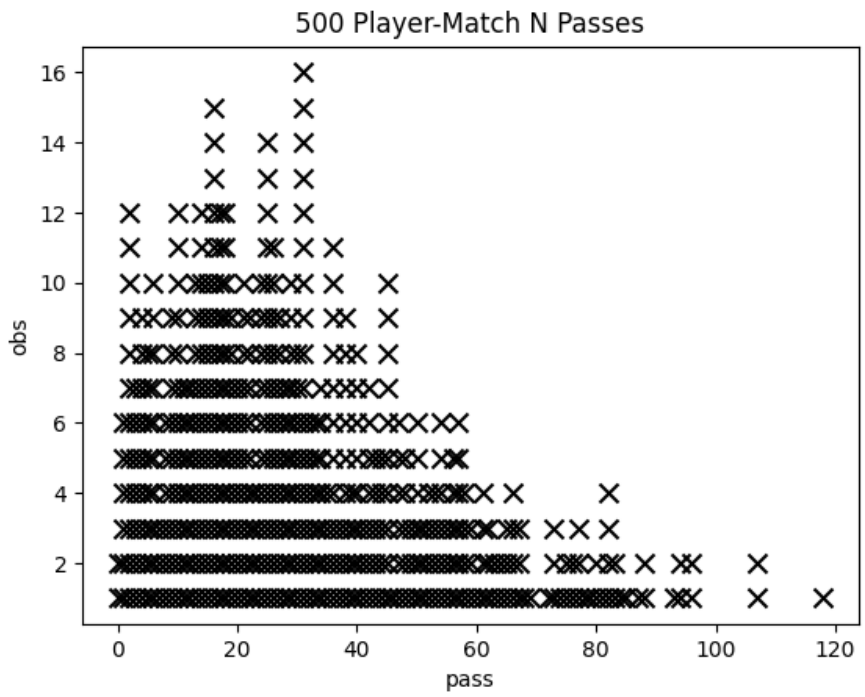

Figure 0.2: 500 Player-Match N Passes

These stacked x's show us the *distribution* of number of passes. Distributions are the key to statistics. When we player goes into a World Cup game and says, "I wonder how many passes I'll have today" the answer is he's picking out one of these little x's at random. Each *x* is equally likely to get picked. Each number of *passes* is not. There are a lot more x's between 20-40 passes than there are between 60 and 80.

In this case, passes come in discrete numbers, but it may make more sense to treat points as a *continous* 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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

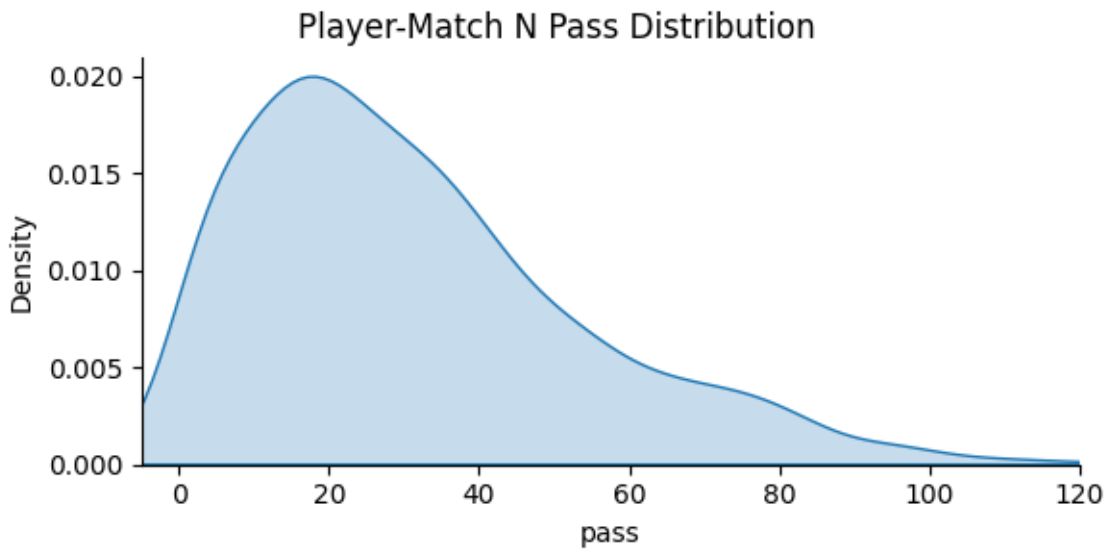

Figure 0.3: Kernel Density - N of Passes

The technical name for these curves are **kernel densities**[1].

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 (0.020, 0.015, 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

So we have our number of passes distribution:

[1] 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

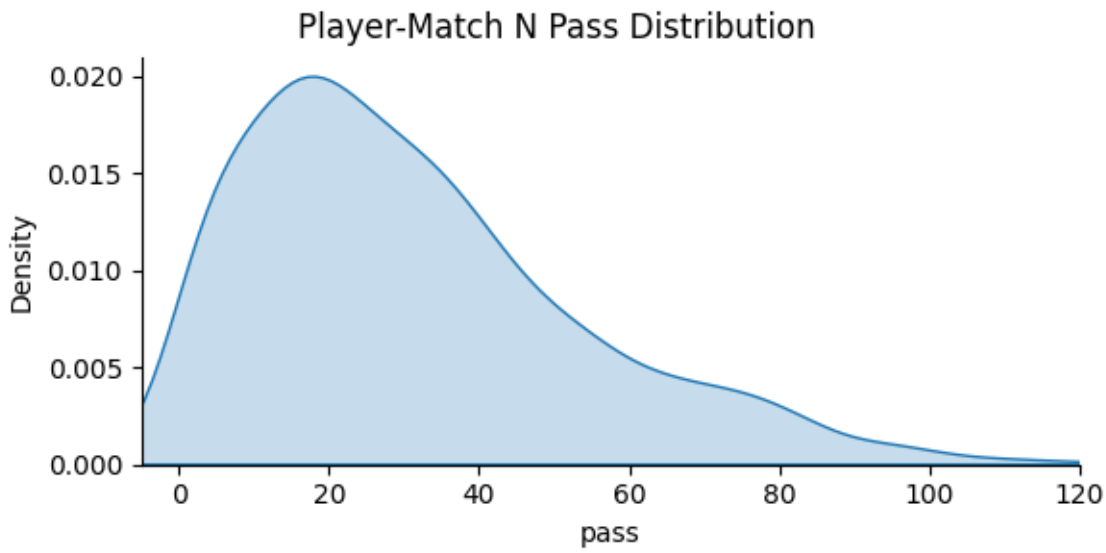

Figure 0.4: Kernel Density - N of Passes

This gives us a good overview of what to expect: most players pass the ball under 40 times a match. A few are making 100+ passes.

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, the median is 27 passes.

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 2018 player-match passes is 6.

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 loading and processing our various datasets.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Pandas lets us calculate any percentile we want with the `quantile` function. Let's look at number of passes in our player-match data:

```
In [1]: dfpm['pass'].quantile(.9)
Out[1]: 65.0
```

So player pass the ball more than 65 times in only 10% of their games.

You can also calculate multiple statistics — including several percentiles — at once in Pandas with `describe`.

```
In [2]: dfpm[['pass', 'shot']].describe()
Out[2]:
              pass         shot
count  1671.000000  1671.000000
mean     31.599641     0.817475
std      23.397261     1.157462
min       0.000000     0.000000
25%      14.000000     0.000000
50%      27.000000     0.000000
75%      43.000000     1.000000
max     174.000000     7.000000
```

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 number of shots by player per match. Here's how often each value shows up in our data:

```
In [3]: dfpm['shot'].value_counts(normalize=True).sort_index().head(10)
Out[3]:
0    0.536804
1    0.254339
2    0.120886
3    0.055057
4    0.018552
5    0.007181
6    0.004189
7    0.002992
```

So 54% of players take 0 shots, 25% take 1 shot, 12% take two shots, etc.

So the *expected value* of number of shots per player would be:

```
0.54*0 + 0.25*1 + 0.12*2 ... = 0.82
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

This is the same thing as summing up all the individual player-match shot values in our data and dividing by however many there are (1,671 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 kg of a random member of Frances starting lineup.

```
    player_name  pos  weight
T. Alderweireld  DEF      91
  J. Vertonghen  DEF      88
     D. Mertens  FWD      61
      N. Chadli  MID      80
   Y. Tielemans  MID      72
    Y. Carrasco  FWD      71
      A. Witsel  MID      81
     T. Meunier  DEF      82
   K. De Bruyne  MID      68
   M. Batshuayi  FWD      78
   T. Vermaelen  DEF      80
      R. Lukaku  FWD      94
      D. Boyata  DEF      84
    M. Fellaini  MID      85
     V. Kompany  DEF      85
    S. Mignolet  GKP      87
     M. Dembélé  MID      88
    K. Casteels  GKP      86
    T. Courtois  GKP      94
      T. Hazard  FWD      69
     A. Januzaj  MID      75
    C. Kabasele  DEF      84
      E. Hazard  FWD      74
  L. Dendoncker  MID      76
```

There are 24 players, so the "probability" of picking any given one is `1/24=0.04167`.

Then we have:

```
0.04167*91 + 0.04167*88 + ... = 80.54
```

Which of course is the same as summing up `91 + 88 + 61 + ...` and dividing by `24`.

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 calcu-

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

late 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[2]. 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 pass distribution for a game. And you want to use these player distributions to predict total passes 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 individual player distributions and take the mean of that.

But although summary statistics are convenient, I think working with distributions directly is underrated, especially because Python + a few other libraries make it so easy.

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[3]. One of the main problems is that there are multiple ways of doing the same thing, which can get confusing.

So instead we'll 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.

We're still in `06_01_summary.py`. 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.

[2] Technically, there are some slight modifications, but it's the general idea.

[3] 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 passes (our column `pass` in our data `dfpm`). This is how we'd do it:

```
In [1]: g = (sns.FacetGrid(dfpm).map(sns.kdeplot, 'pass', 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(dfpm)
     .map(sns.kdeplot, 'pass', shade=True))
```

Here's what it looks like:

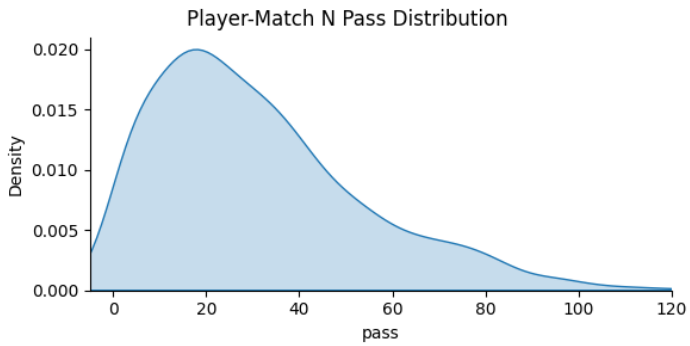

Figure 0.5: Distribution of Standard Fantasy Pts

Note, I've added a title and changed some height and width options to make things clearer. Yours won't show that yet. We'll cover 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.

Seaborn "Levers" - Slicing and Dicing Plots

Seaborn's killer feature is how easy it makes creating and comparing multiple plots.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

For example, say we want separate plots by player position (MID, D, FWD). Now we can introduce our second lever, the `hue` keyword.

```
In [3]:
g = (sns.FacetGrid(dfpm, hue='pos')
     .map(sns.kdeplot, 'pass', shade=True))
```

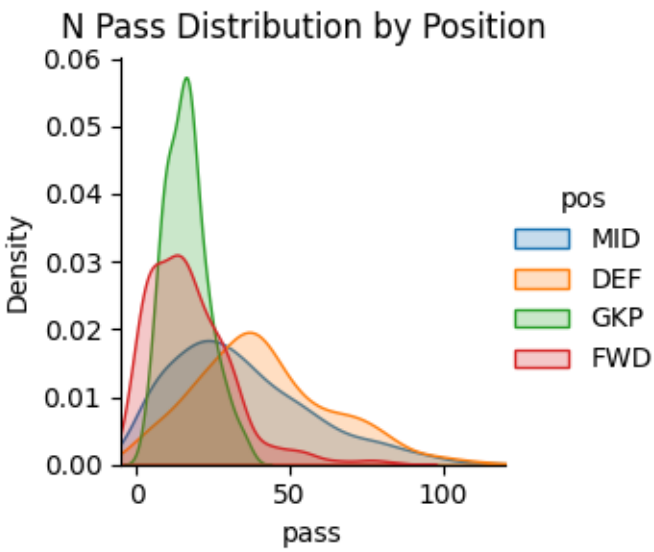

Figure 0.6: Distribution of Passes by Position

Both number of passes and position are *columns* of data. By setting `hue='pos'`, we're telling seaborn to plot different distributions of `pass` (with different colors, or hues) for each value of `pos`. So we have one density of `pass` when `pos='D'`, another for `pos='MID'` etc.

This plot does a nice job showing the distribution of number of passes across different positions. What if we wanted to add in another dimension, say, which side of the field they played on.

We can use our third lever — the `col` keyword.

```
In [4]:
g = (sns.FacetGrid(dfpm, hue='pos', col='side')
     .map(sns.kdeplot, 'pass', shade=True))
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

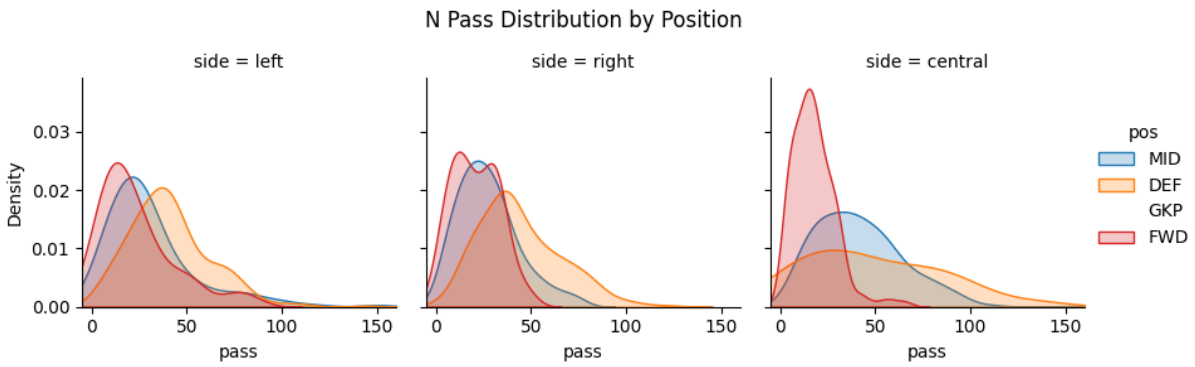

Figure 0.7: Number of Passes by Position and Side

This draws separate plots (on different columns) for every value of whatever you pass to `col`. So we have three plots: one when `side='left'`, another for `side='right'` and one when `side='central'`.

Within each of these, the `hue=pos` draws separate plots for each position.

Note, goalies no longer show up, because they don't have a side.

This is cool, but it's bugging me that the order of our plots is left, right, *then* central. Central should be in the middle.

Until now I actually didn't know how to do this, but as usual a quick Google search ("seaborn facegrid order col") brings up a relevant stackoverflow. We just need to add the `col_order` keyword:

```
In [5]:
g = (sns.FacetGrid(dfpm, hue='pos', col='side',
           col_order=['left', 'central', 'right'])
     .map(sns.kdeplot, 'pass', shade=True))
```

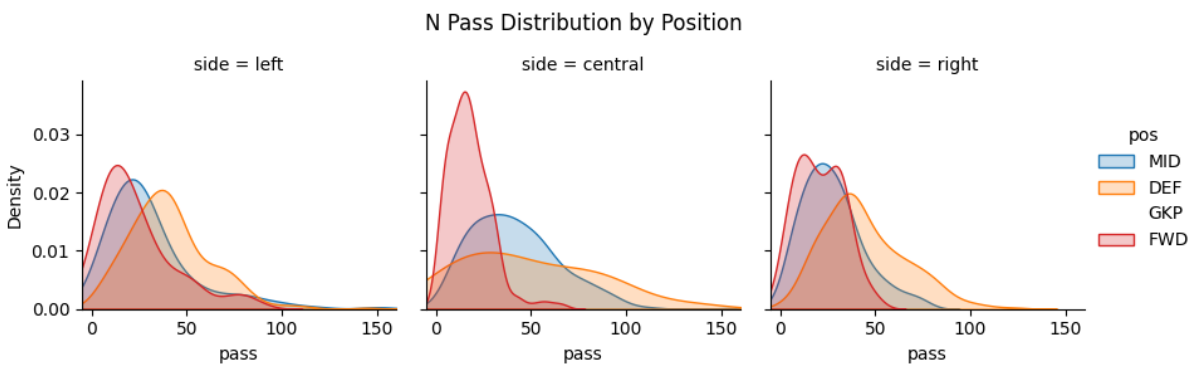

Figure 0.8: Number of Passes by Position - Ordered

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Great. There's also a `row` argument. I don't actually find myself using it that much, but here I suppose we could draw this same plot with positions on different rows.

Note goalies don't have a "side", so let's put them in the middle:

```
In [6]: dfpm.loc[dfpm['pos'] == 'GKP', 'side'] = 'central'
```

Then plot it. I haven't tested this, but I'll assume `row_order` works the same as `col_order`:

```
In [7]:
g = (sns.FacetGrid(dfpm, hue='pos', col='side', row='pos',
                   col_order=['left', 'central', 'right'],
                   row_order=['FWD', 'MID', 'DEF', 'GKP'])
     .map(sns.kdeplot, 'pass', shade=True))
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

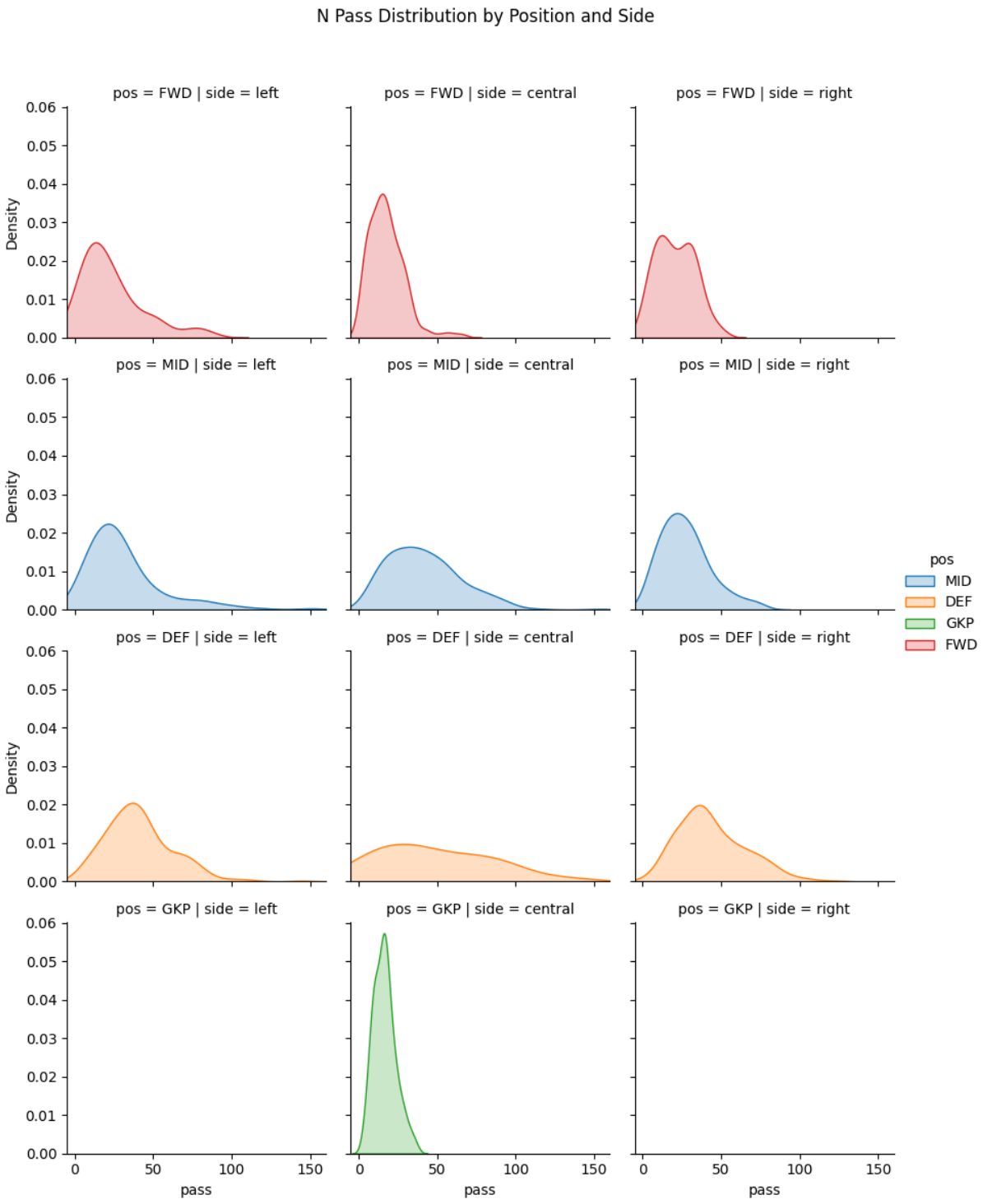

Figure 0.9: Number of Passes by Position - Rows

Awesome.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Again, all three of these variables — `pass`, `pos`, and `side` 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 match data, which we loaded into a DataFrame `dfm` at the beginning of 06_01_summary.py.

Say we want to plot the distributions of goals scored by home vs away teams using our match data. That's no problem conceptually, except our goals data is in separate columns.

We currently have this:

```
In [1]:
(dfm[['date', 'home_team', 'away_team', 'home_score', 'away_score']]
 .sort_values('date').head())

Out[1]:
                   date  home_team     away_team  home_score  away_score
26  2018-06-14 15:00:00     Russia  Saudi Arabia           5           0
60  2018-06-15 12:00:00      Egypt       Uruguay           0           1
59  2018-06-15 15:00:00    Morocco          Iran           0           1
57  2018-06-15 18:00:00   Portugal         Spain           3           3
55  2018-06-16 13:00:00  Argentina       Iceland           1           1
```

But we actually need something more like this:

```
                   date          team  score location
0  2018-06-14 15:00:00  Saudi Arabia      0     away
1  2018-06-14 15:00:00        Russia      5     home
2  2018-06-15 12:00:00         Egypt      0     home
3  2018-06-15 12:00:00       Uruguay      1     away
4  2018-06-15 15:00:00       Morocco      0     home
5  2018-06-15 15:00:00          Iran      1     away
6  2018-06-15 18:00:00      Portugal      3     home
7  2018-06-15 18:00:00         Spain      3     away
8  2018-06-16 13:00:00     Argentina      1     home
9  2018-06-16 13:00:00       Iceland      1     away
```

They contain the same information, we've just changed the granularity (from match to team-match) 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

First let's build a function that — given our data (`match`) and a location (`home` or `away`) — moves that score (e.g. `match['home_score']`) to a score column, then adds in another column indicating which location we're dealing with. So this:

```
def home_away_score_df(df, location):
    df = df[['match_id', 'date', f'{location}_team', f'{location}_score']]
    df.columns = ['match_id', 'date', 'team', 'score']
    df['location'] = location
    return df
```

And to use it with `home`:

```
In [2]: home_away_score_df(dfm, 'home').head()
Out[2]:
   match_id                 date       team  score location
0   2058017  2018-07-15 15:00:00     France      4     home
1   2058012  2018-07-07 18:00:00     Russia      2     home
2   2057977  2018-06-26 18:00:00    Iceland      1     home
3   2057974  2018-06-21 18:00:00  Argentina      0     home
4   2058014  2018-07-10 18:00:00     France      1     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]:
score_long = pd.concat([
    home_away_score_df(dfm, 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 passing it to seaborn:

```
In [4]:
g = (sns.FacetGrid(score_long, hue='location')
     .map(sns.kdeplot, 'score', 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:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

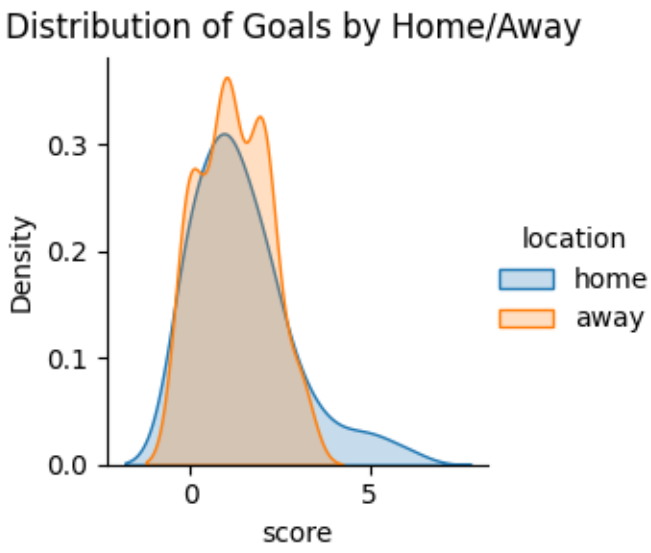

Figure 0.10: Distribution of Goals by Home/Away

Of course, this kind of a silly example since home and away are meaningless in World Cup games (all games took place in Russia) and any difference we see here are random. But you get the idea

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 players' height and weight. These obviously tend to move together, but what's the best way to visualize that?

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 let's look at the relationship between players height and weight.

Note: we're still in `06_01_summary.py`. We're using `dfp`, which we loaded and processed at the top of the file.

To make scatter plots in seaborn we use the basic function `relplot` (for relationship plot):

```
In [1]: g = sns.relplot(x='weight', y='height', data=dfp)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

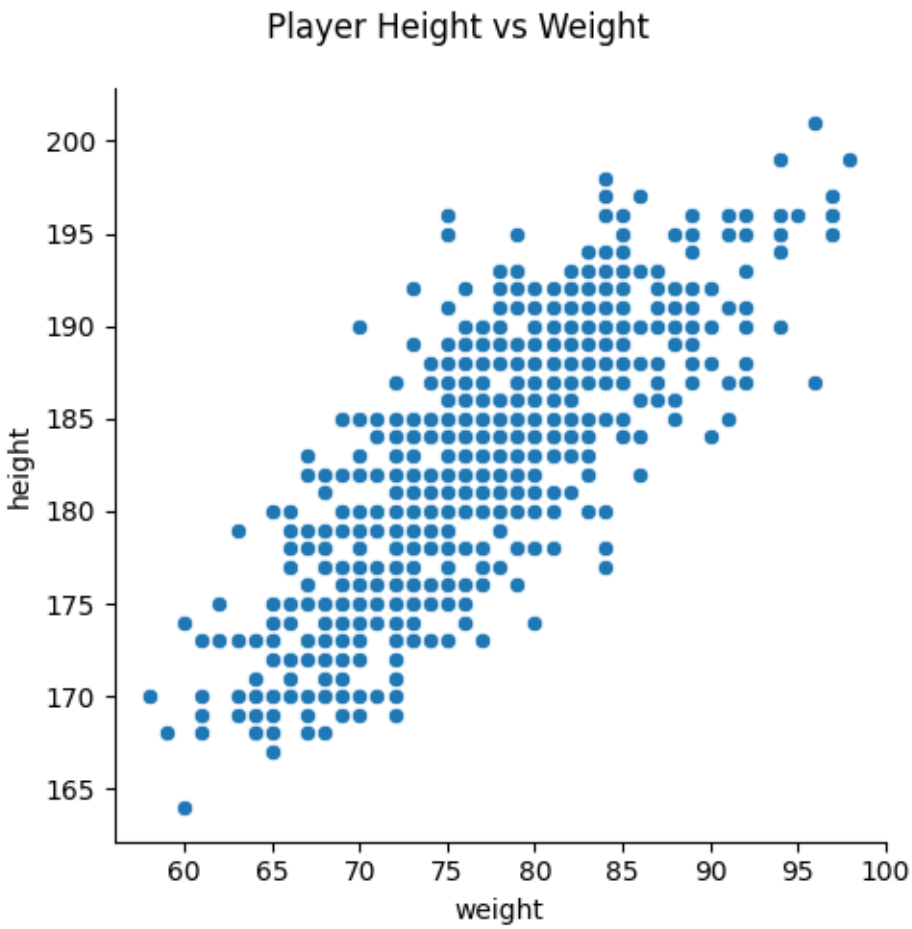

Figure 0.11: Player Height vs Weight

This cloud of points is moving up and to the right, showing a *positive* relationship between weight and height. 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 position. Just like the density plots, we do this using the `hue` keyword.

```
In [2]: g = sns.relplot(x='weight', y='height', hue='pos', data=dfp)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

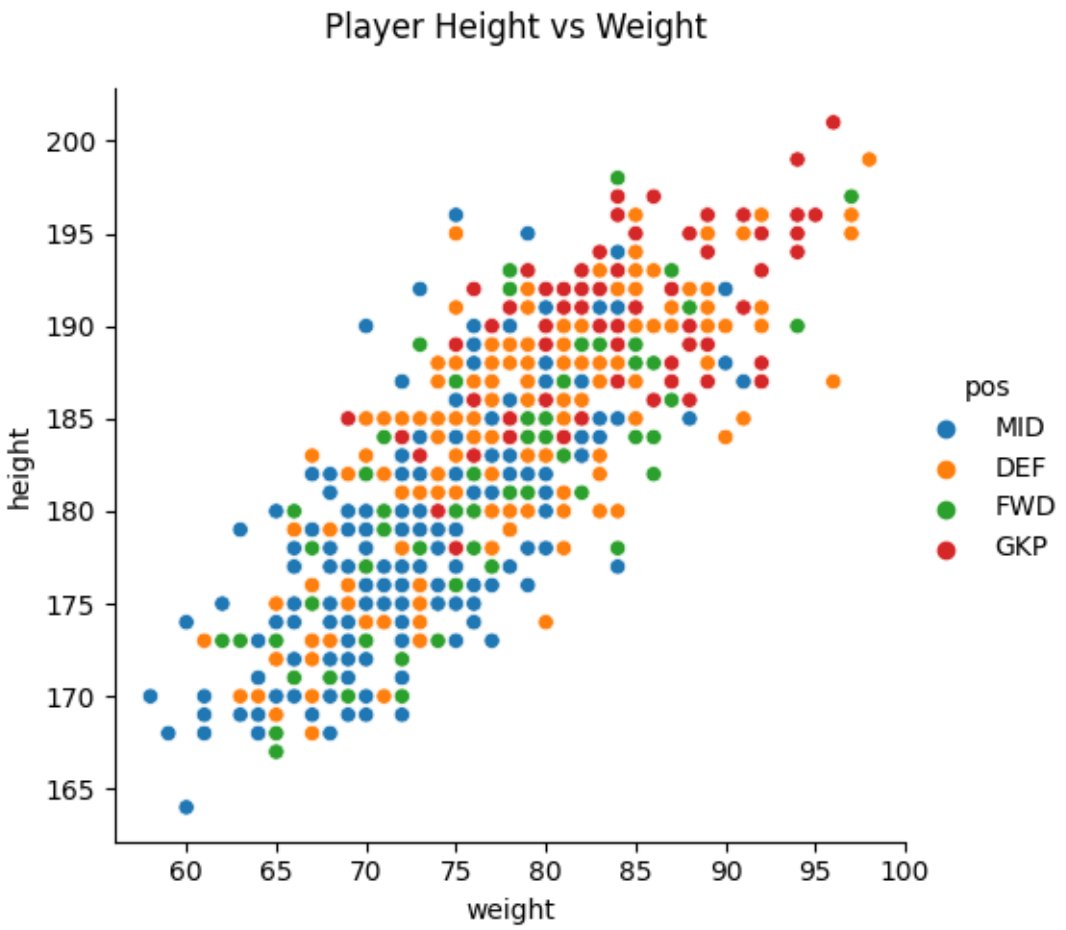

Figure 0.12: Player Height vs Weight, by Position

Aside: Jittering

Our graph makes it pretty clear that height and weight come in nice round, whole numbers. This makes our scatter plot look like more of a regularly spaced grid, vs a cloud of points.

This says more about our our data and how it was collected (e.g. rounded to the nearest kg) than anything else, and can look a bit unnatural.

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 often improve the visuals.

We can do this with the `random` library, which is built into Python:

```
In [3]: import random
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

`random` lets you take draws from miscellaneous distributions. For example, here's how you'd get a random number between 0 and 1:

```
In [4]: random.uniform(0, 1)
Out[4]: 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 weight and height data, and it'd work fine. The only problem is it'd be slightly biased 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 [5]: [random.gauss(0, 1) for _ in range(10)]
Out[5]:
[0.8313728638876453,
 0.17057235664185233,
 0.2536999616218848,
 -0.18657013635142755,
 -1.2022458299176941,
 0.5579663225316337,
 -0.44036955593965005,
 2.835632599157855,
 0.11302542631295645,
 0.9804724156385064]
```

So let's make new new columns, `jheight` and `jweight` (for jittered height and weight) that are the same as regular `height` and `weight`, but have a `random.gauss(0, 1)` added.

```
In [6]:
dfp['jheight'] = dfp['height'].apply(lambda x: x + random.gauss(0, 1))
dfp['jweight'] = dfp['weight'].apply(lambda x: x + random.gauss(0, 1))
```

And redoing our plot:

```
In [7]: g = sns.relplot(x='jweight', y='jheight', hue='pos', data=dfp)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

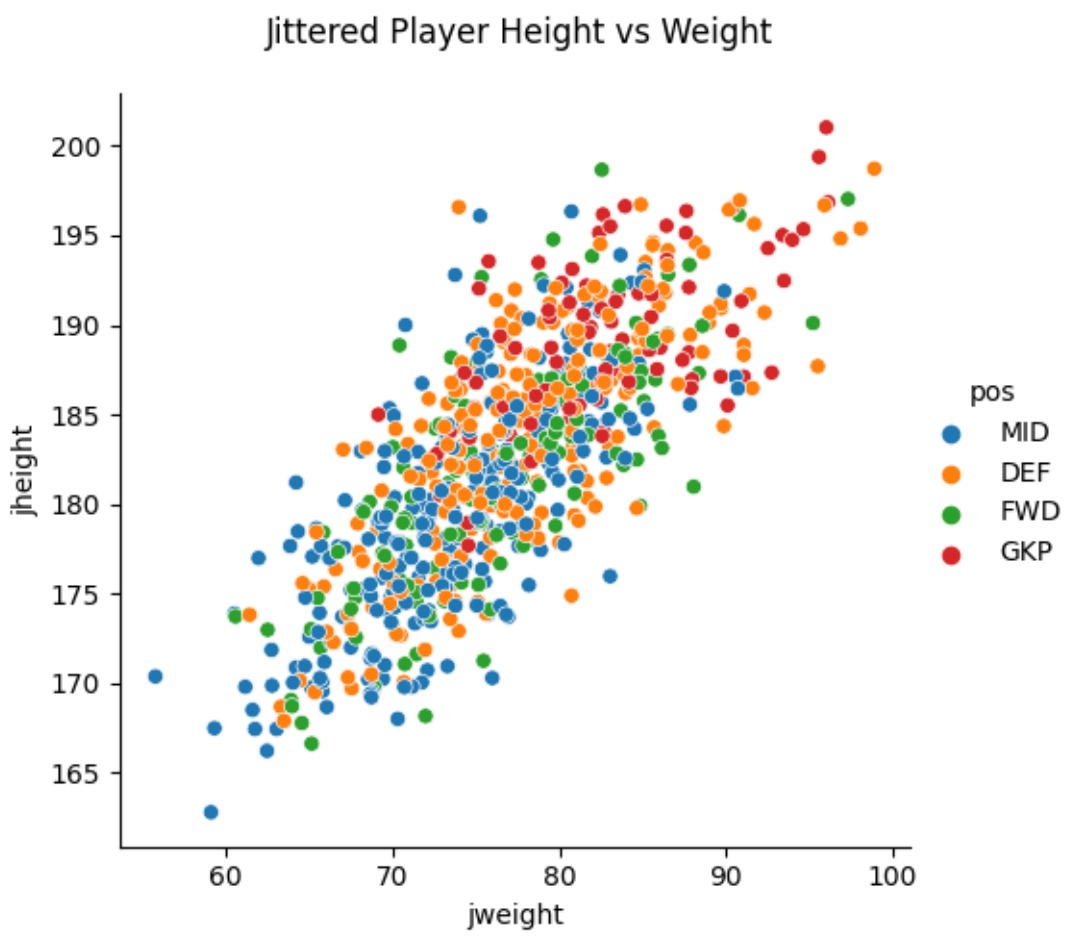

Figure 0.13: Player Height vs Weight, by Position

That's better. Back to seaborn.

Seaborn scatter plots also take the column argument:

```
In [3]:
g = sns.relplot(x='jweight', y='jheight', hue='pos', col='team',
                col_wrap=5, data=dfp)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Figure 0.14: Player Height vs Weight, by Team

Though in this specific case a distribution plot looks sort of cool:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

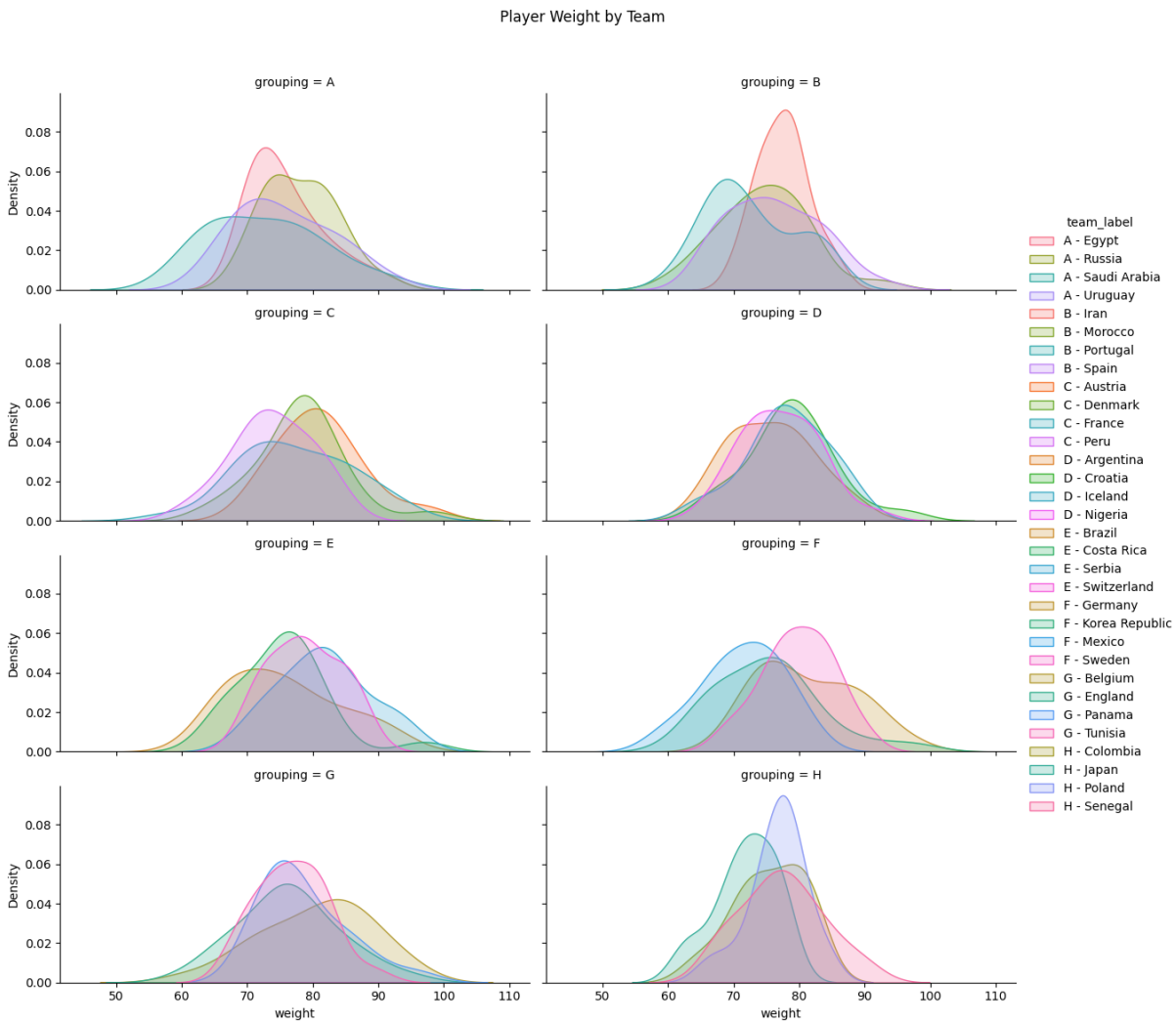

Figure 0.15: Player Weight by Team

Contour Plots

One visually interesting tweak is to turn these scatter charts into *contour* plots.

Earlier we looked 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 height by weight plot (note the curves are smoothing things out, so we no longer have to jitter):

```
In [7]:
g = (sns.FacetGrid(dfp, col='pos', hue='pos', col_wrap=2)
     .map(sns.kdeplot, 'weight', 'height', shade=True))
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

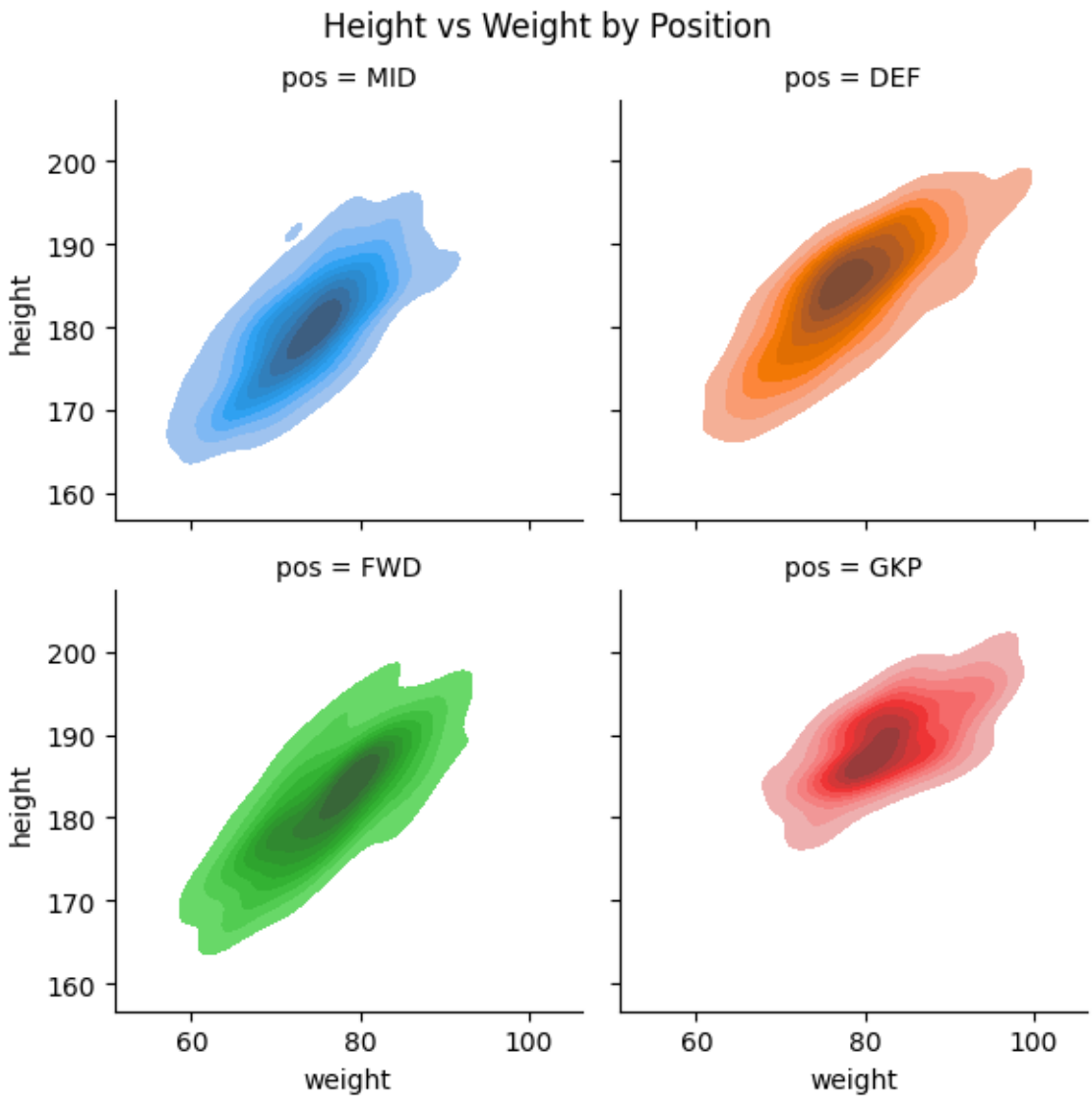

Figure 0.16: Player Height vs Weight, Contour Plot

This plot does a good job of showing how much taller goalies are vs other positions.

Here 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:

```
In [8]:
g = (sns.FacetGrid(players, col='pos', hue='pos')
     .map(sns.kdeplot, 'weight_lb', 'height_in'))
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

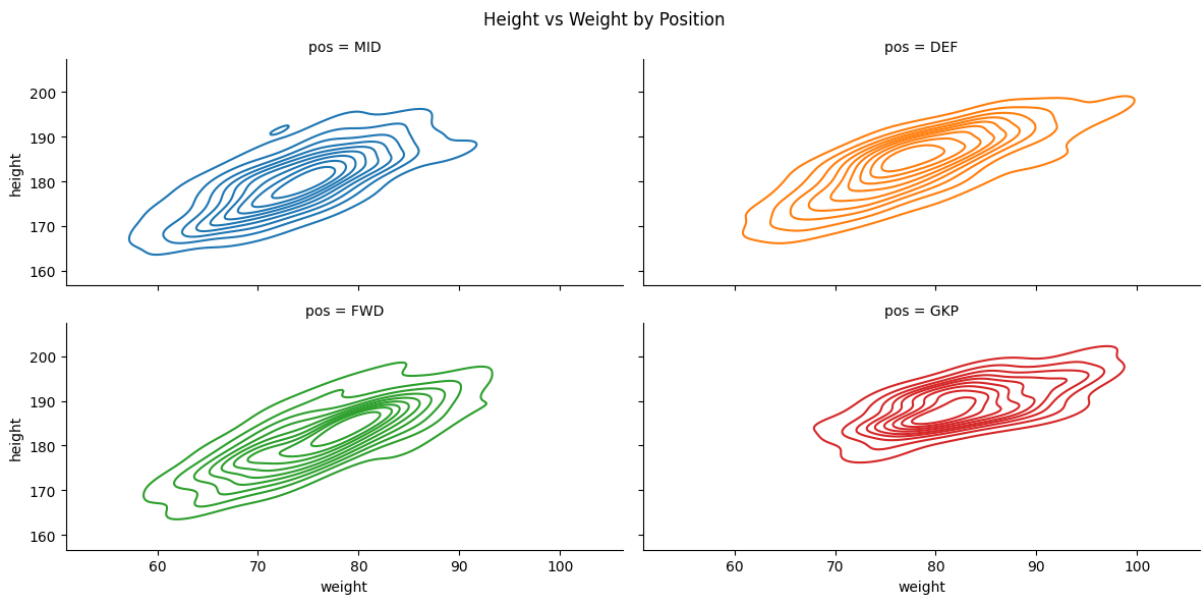

Figure 0.17: Player Height vs Weight, Contour Plot - No Shading

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, shot distance in meters (say `n`) is perfectly correlated shot distance in feet (`3.28*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.

Note: we're still in `06_01_summary.py`. We're using our team-match level stats, which we loaded in the DataFrame `dftm` at the top of the file.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Let's look at a correlation matrix for some team-match level stats. In Pandas you get a correlation matrix with the `corr` function.

```
In [1]:
(dftm[['shot', 'goal', 'shot_opp', 'goal_opp', 'pass_opp', 'pass', 'win']]
  .corr()
  .round(2))

Out[1]:
          shot  goal  shot_opp  goal_opp  pass_opp  pass   win
shot      1.00  0.22     -0.30     -0.01     -0.46  0.58  0.20
goal      0.22  1.00     -0.01      0.30      0.10  0.20  0.50
shot_opp -0.30 -0.01      1.00      0.22      0.58 -0.46 -0.12
goal_opp -0.01  0.30      0.22      1.00      0.20  0.10 -0.44
pass_opp -0.46  0.10      0.58      0.20      1.00 -0.60 -0.10
pass      0.58  0.20     -0.46      0.10     -0.60  1.00  0.12
win       0.20  0.50     -0.12     -0.44     -0.10  0.12  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 targets and carries is the same as the correlation between carries and targets.

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 number of shots and goals in a game is `0.22`.

Here's that in (jittered) scatter plot form:

```
In [2]:
dftm['jshot'] = dftm['shot'].apply(lambda x: x + random.gauss(0, 1))
dftm['jgoal'] = dftm['goal'].apply(lambda x: x + random.gauss(0, 1))

sns.relplot(x='jshot', y='jgoal', data=dftm)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

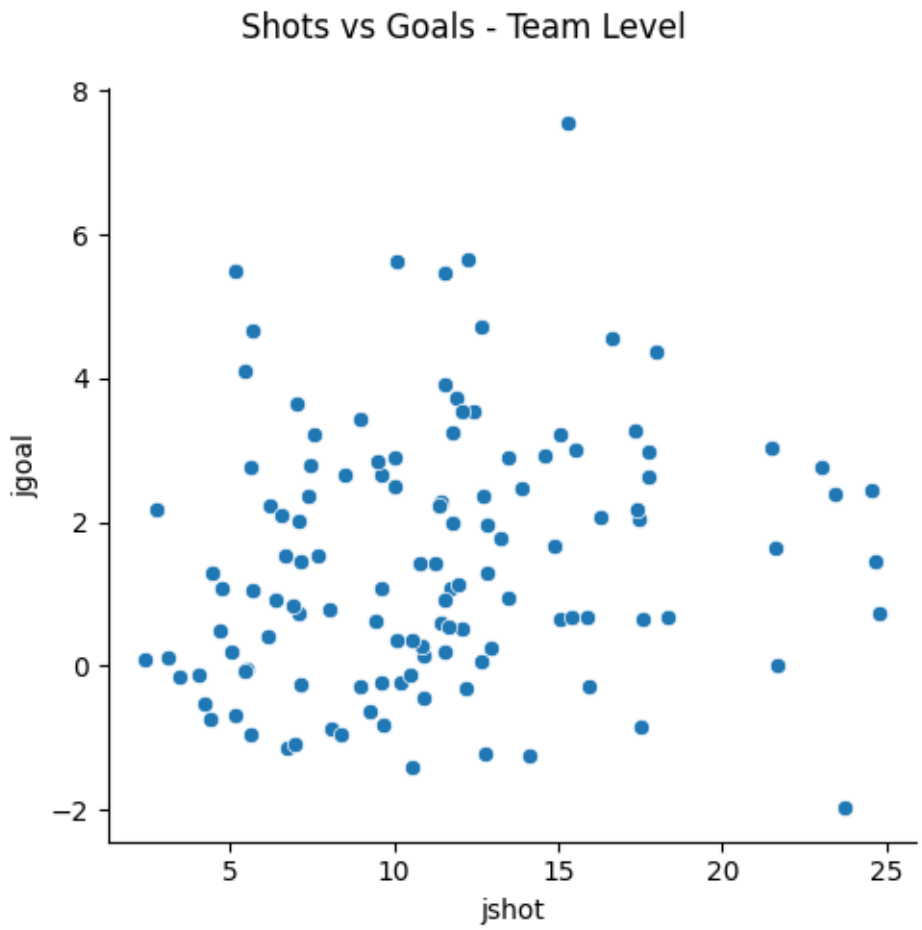

Figure 0.18: Shots vs Goals - 0.22 Correlation

So that's a correlation of 0.22. Compare that to the correlation between number of passes and shots, which is 0.58.

```
In [3]: g = sns.relplot(x='jshot', y='pass', data=tm)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

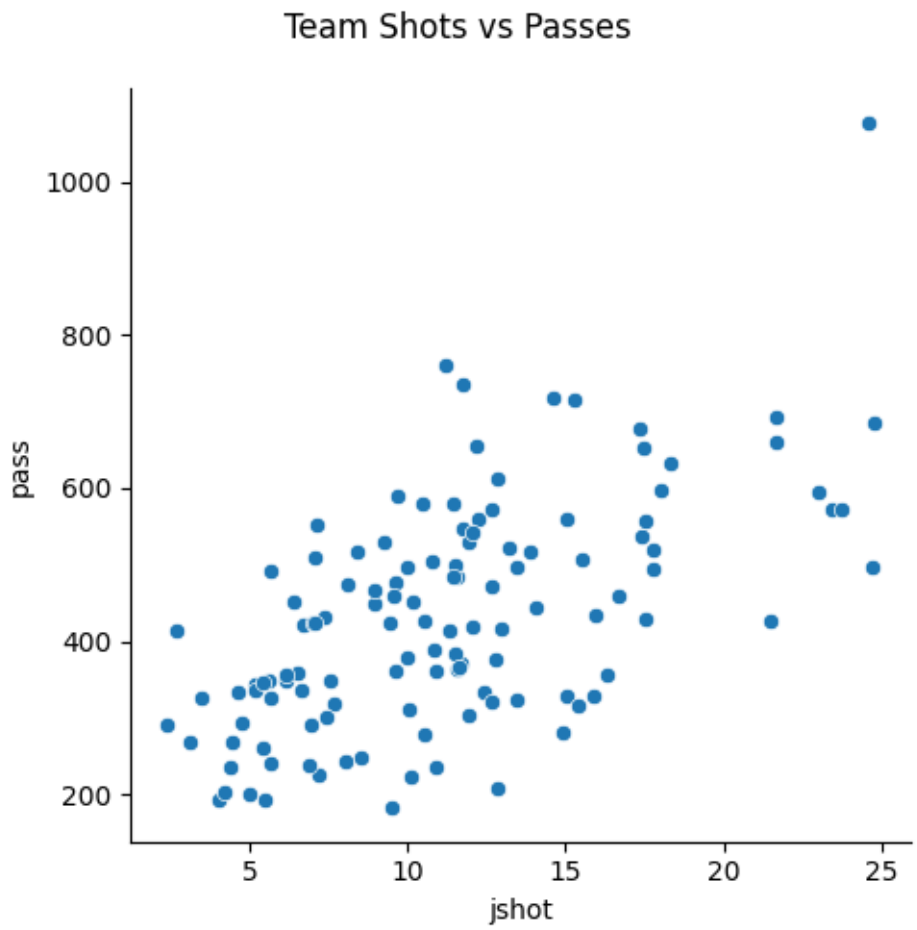

Figure 0.19: Shots vs Passes - 0.58 Correlation

These points in a cloud around the bottom-left to upper-right diagonal.

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. minute in game, year 2009-2022, month) 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.

So maybe we want to look at total goals scored over time. First let's total home and away scores to get total goals for the match:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: dfm['total_goals'] = dfm['home_score'] + dfm['away_score']
```

Then do our line plot.

```
In [2]:
g = sns.relplot(x='day', y='total_goals', kind='line', data=dfm)
```

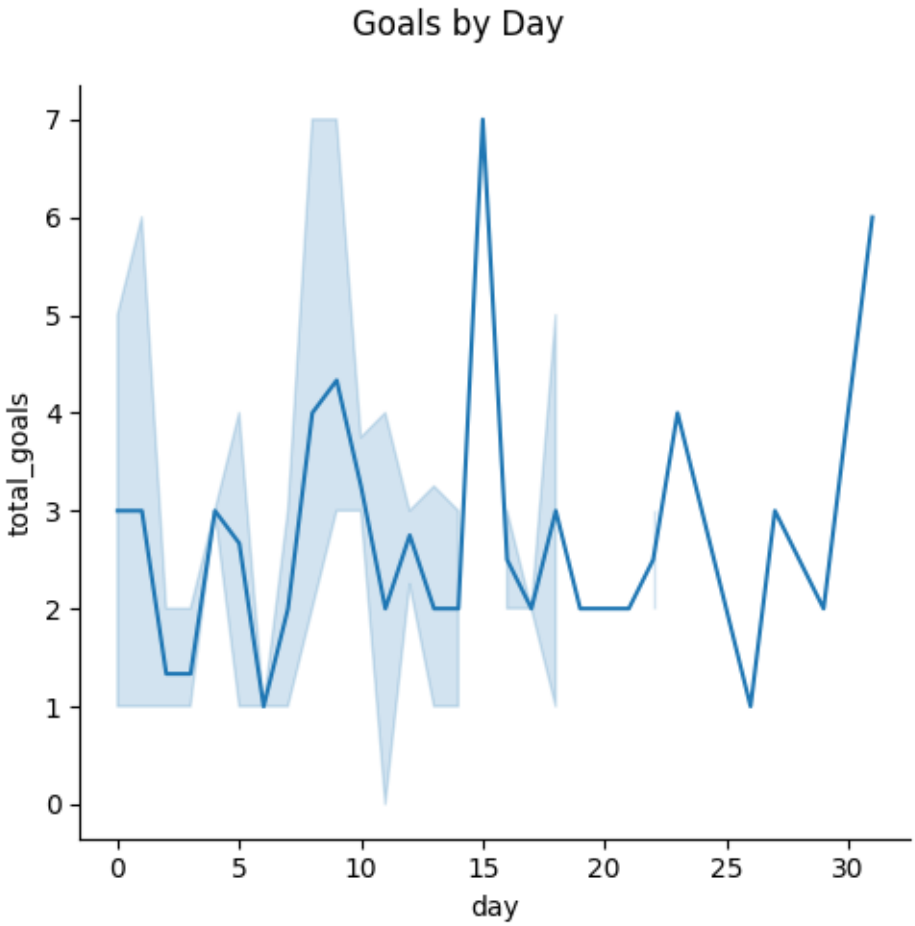

Figure 0.20: Goals by Day

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 day of the tournament on the `x` axis and `total_goals` (the sum of home and away goals) on the `y` axis.

But remember, our data is at the *match* level, so for any given day (our x axis) and score (y) there are multiple observations. Say we're looking at day 1 of the tournament:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [3]: dfm.query("day == 1")
Out[3]:
    match_id                        label    group  ... day total_goals
55   2057972  Argentina - Iceland, 1 - 1  Group D  ...   1           2
57   2057960     Portugal - Spain, 3 - 3  Group B  ...   1           6
59   2057961       Morocco - Iran, 0 - 1  Group B  ...   1           1
```

Instead of plotting separate lines for each match, 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 each row — say the average daily goals, it'll plot just the single lines.

```
In [4]:
ave_total_goals = dfm.groupby('day')['total_goals'].mean().reset_index()

In [5]:
g = sns.relplot(x='day', y='total_goals', kind='line', data=
    ave_total_goals)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

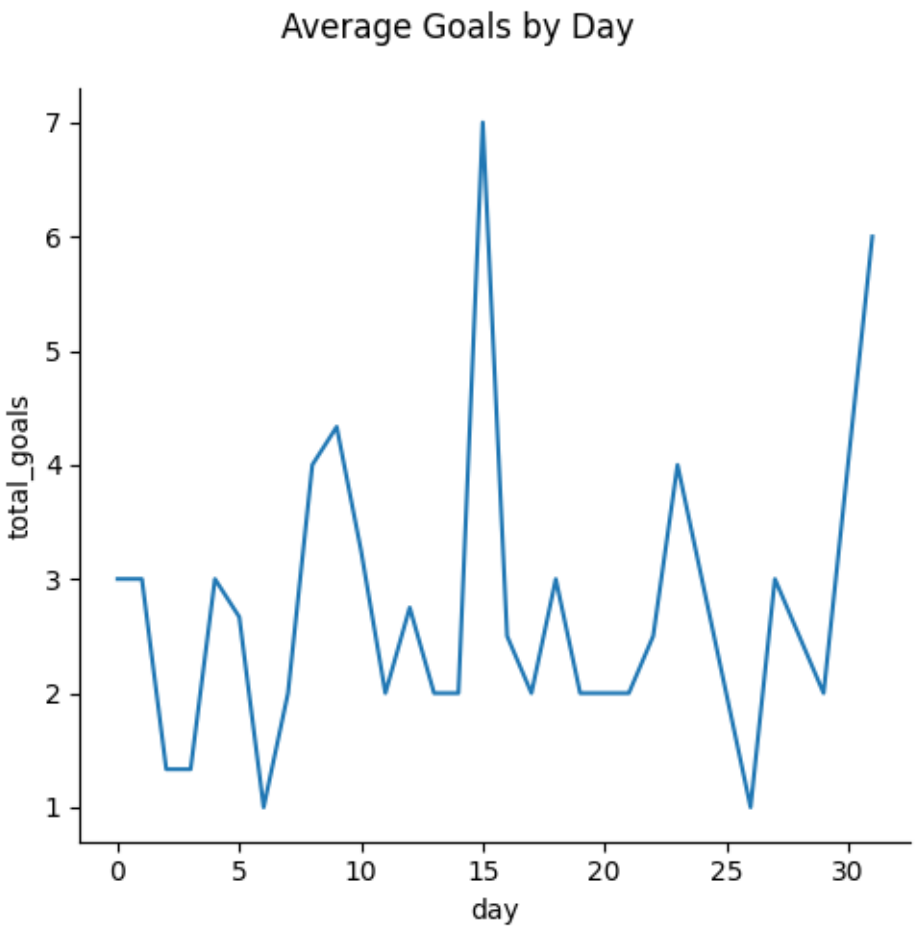

Figure 0.21: Ave Goals by Day

Like the density examples, the line version of `relplot` includes options for separating plots by hues, columns and rows.

Also, any measure of time works. Maybe we want to look at shot distance by minute in the period and foot:

```
In [6]:
g = sns.relplot(x='min_period', y='dist_m', kind='line', hue='foot',
                row='period', data=dfs)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

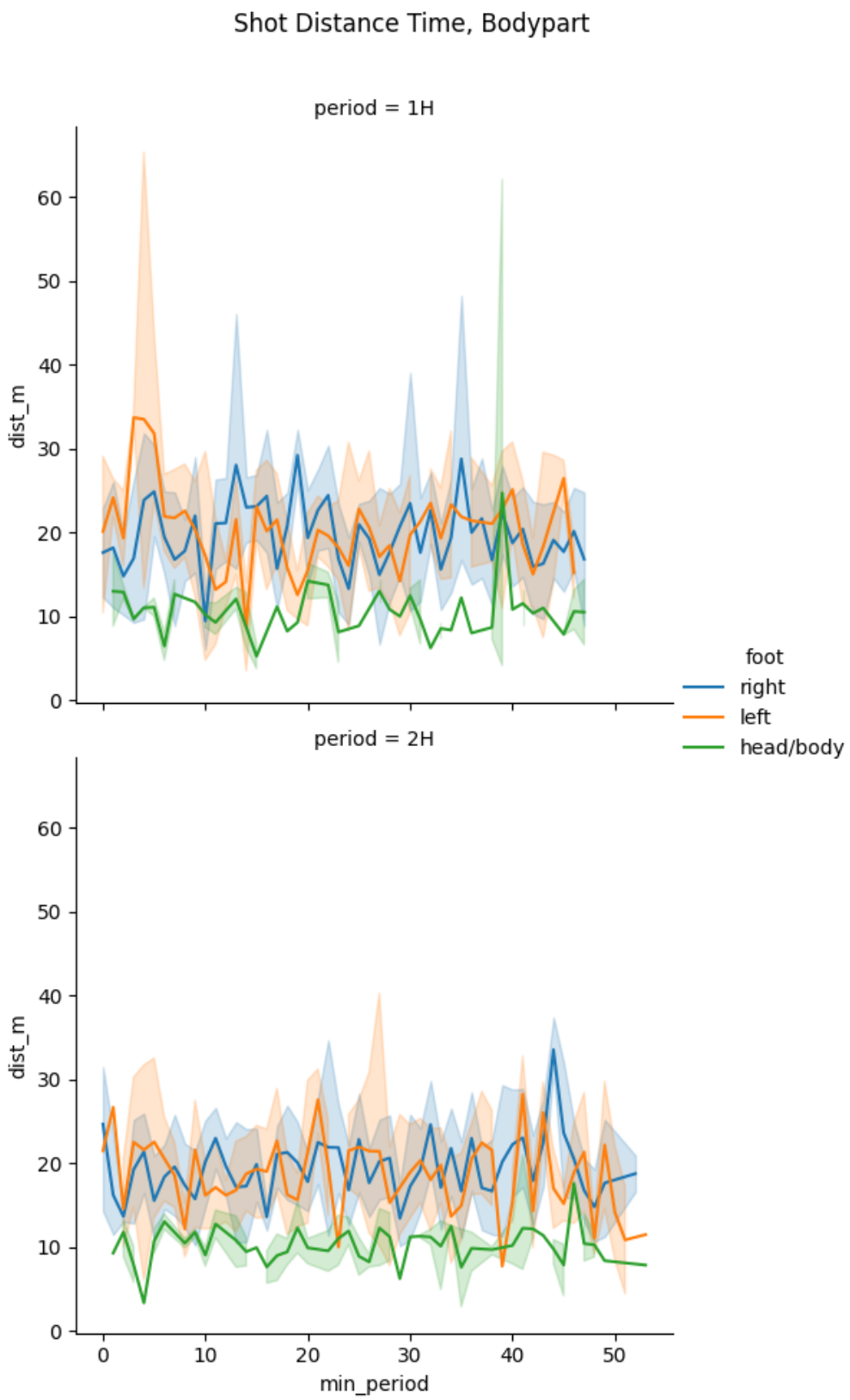

Figure 0.22: Shot Distance by Minute, Bodypart

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 number of passes by position distribution plot from earlier.

```
In [1]:
g = (sns.FacetGrid(dfpm, col='pos')
     .map(sns.kdeplot, 'pass', shade=True))
```

Wrapping columns

By default, seaborn will spread all our columns out horizontally. With multiple columns that quickly becomes unwieldy:

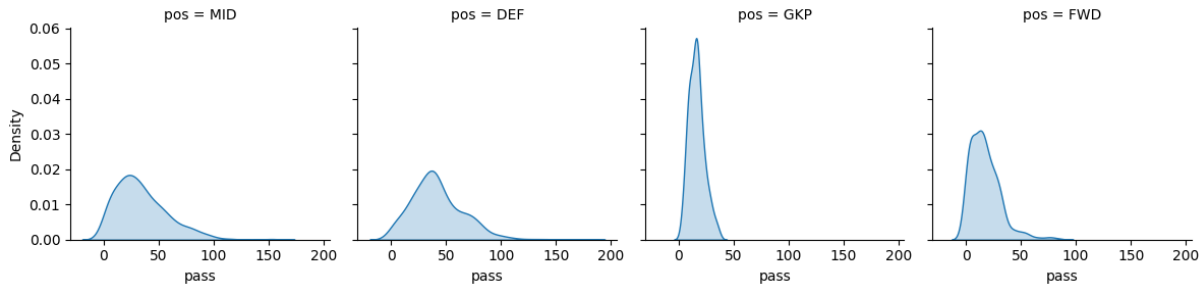

Figure 0.23: Passes by Position - No col_wrap

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(dfpm, col='pos', col_wrap=2)
     .map(sns.kdeplot, 'pass', shade=True))
```

Here it makes the plot much more readable.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

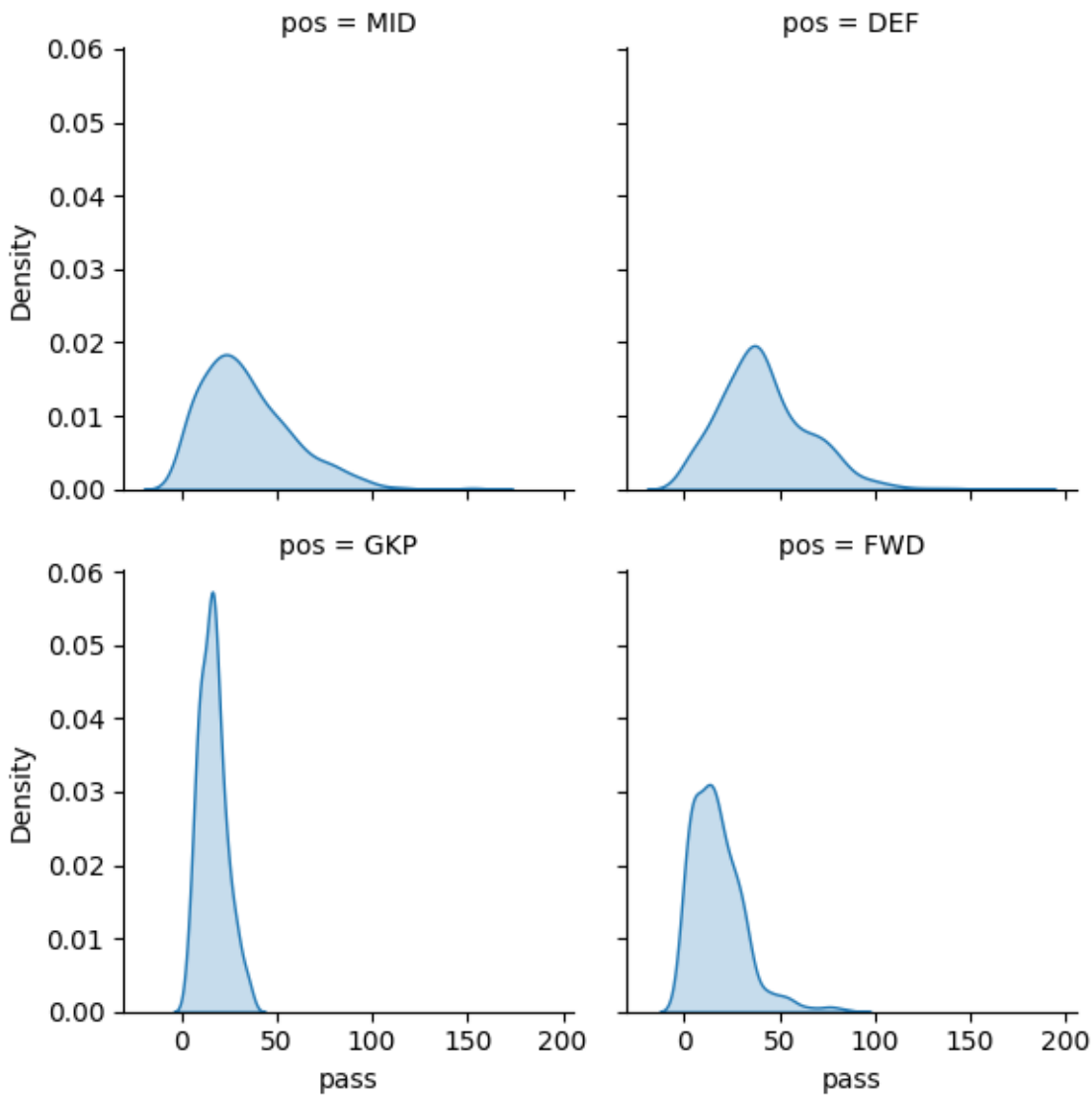

Figure 0.24: Passes by Position - col_wrap=2

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)
g.fig.suptitle('Distribution of No of Passes by Position')
```

This is something that seems like it should be easier, but it's a small price to pay for the overall flexi-

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

bility 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, 120))
```

`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')
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

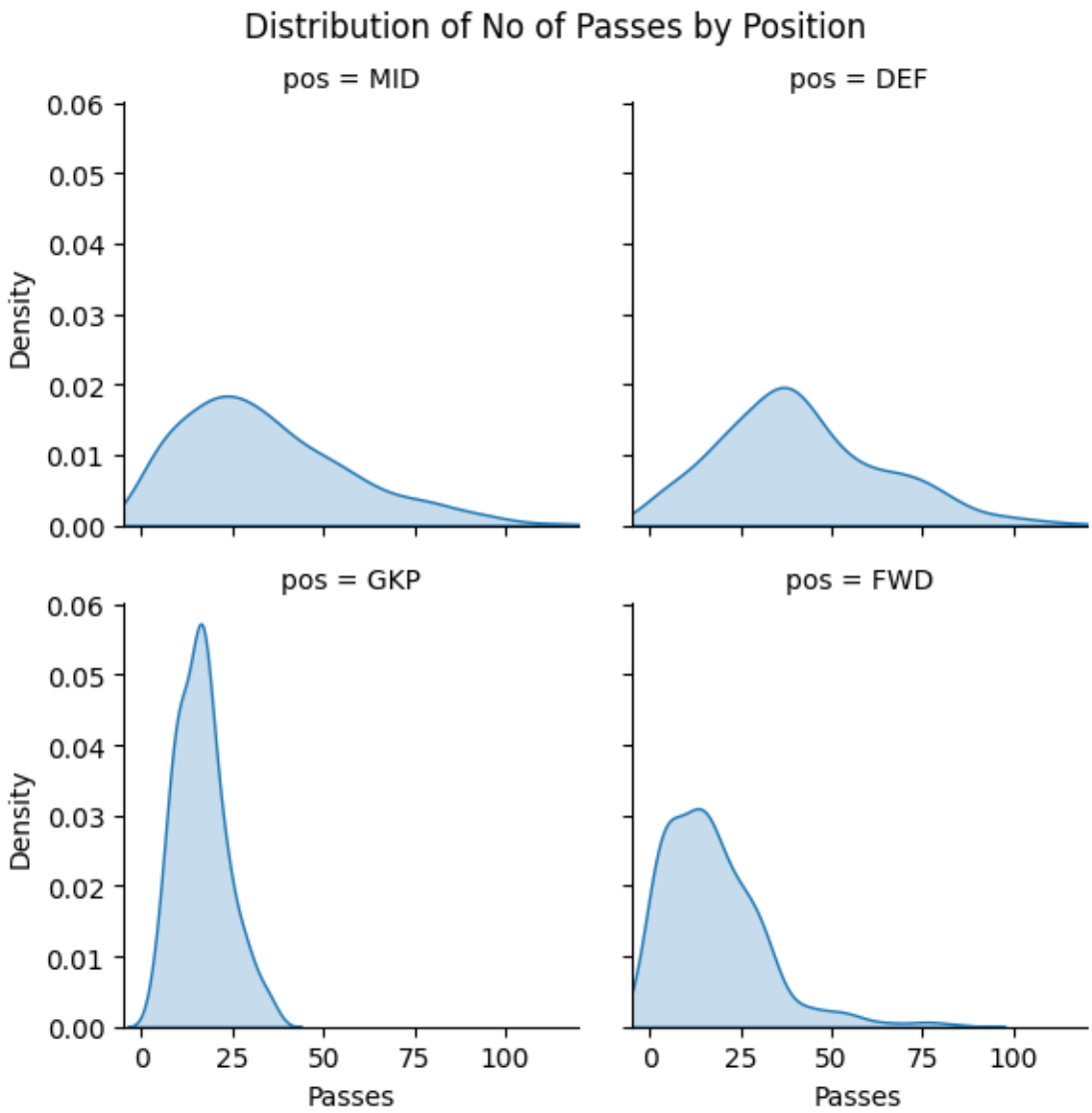

Figure 0.25: Passes by Position - more options

Legend

For `relplot`, Seaborn will automatically add a legend you use the `hue` keyword. If you don't want it you can pass it `legend=False`.

For our `FacetGrid` then `map` approach you need to add it yourself:

```
In [6]: g.add_legend()
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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('no_passes_by_position.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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Shot Charts

We have a bunch of shot data. 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_m', 'foot', 'goal', 'x', 'y']].head(5)
Out[1]:
                  name     dist_m       foot   goal   x   y
0           A. Samedov  12.987566      right  False  90  31
1    Yasir Al Shahrani  16.559476      right  False  87  73
2           Y. Zhirkov  17.013624       left  False  86  70
3         Y. Gazinskiy   8.506812  head/body   True  93  40
4  Mohammad Al Sahlawi  15.975528       left  False  86  62
```

These coordinates represent a location on a football pitch. If we're standing at midfield and looking at the goal, y gives our location left to right, x to the sideline.

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 we have.

```
In [2]:
g = sns.relplot(data=dfs, x='x', y='y', kind='scatter', s=5)
g.set(yticks=[], xticks=[], xlabel=None, ylabel=None)
g.despine(left=True, bottom=True)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

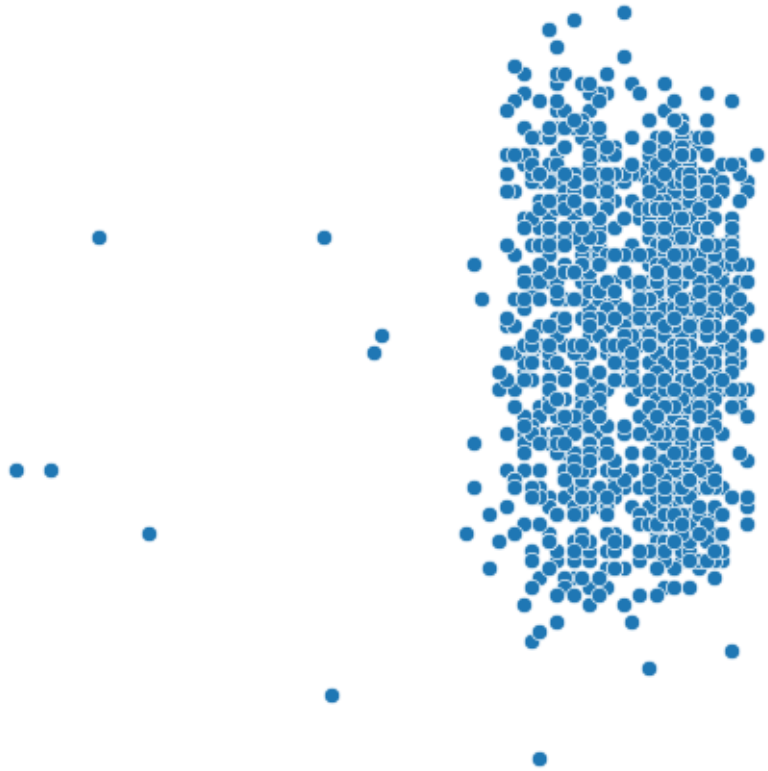

Figure 0.26: Shot Chart - All Data

(Note I'm getting rid of the ticks, axes and labels that show up on regular graphs.)

This maybe looks like it could be some shot data, but it's hard to tell. Let's try adding a field background - with goal line, center circle, box, 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 football field.

I found and edited this one:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

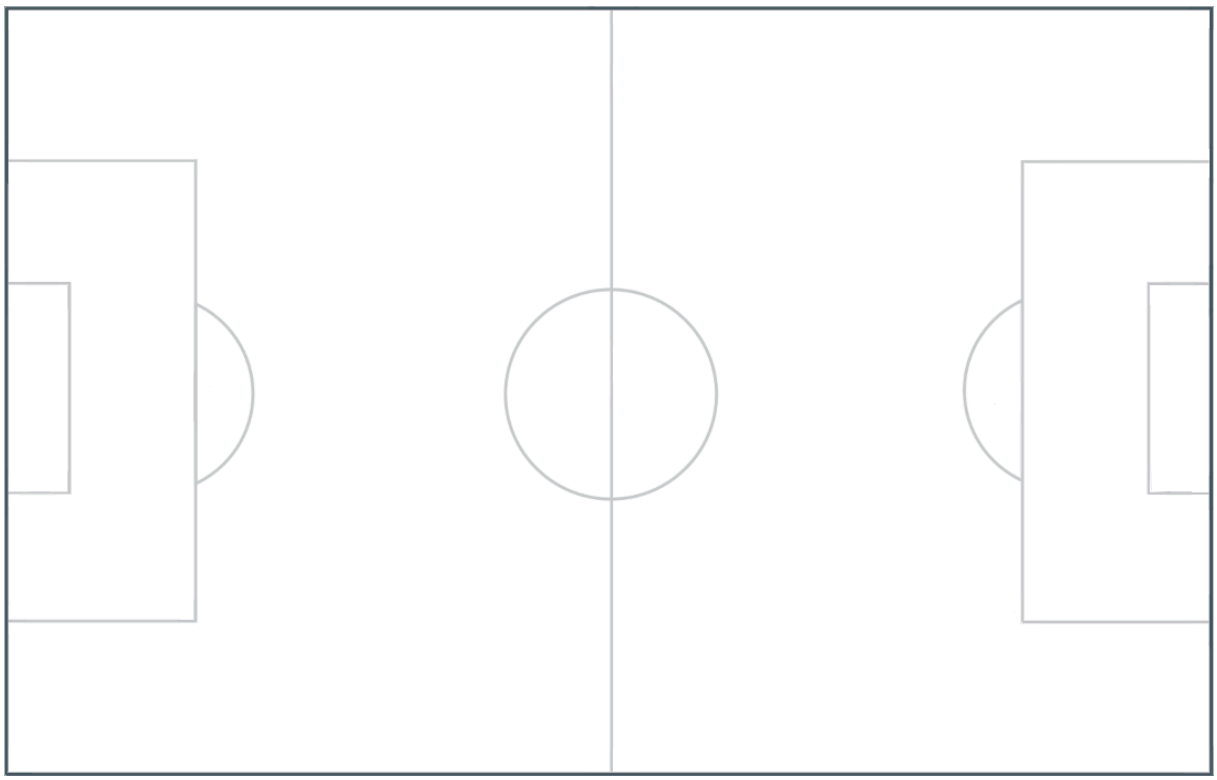

Figure 0.27: Football Field Background

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/soccer_field.png')
```

Then, after we make our scatterplot in seaborn:

```
In [5]:
g = sns.relplot(data=dfs, x='x', y='y', kind='scatter', s=10)
g.set(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=[0, 120, 0, 75])
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

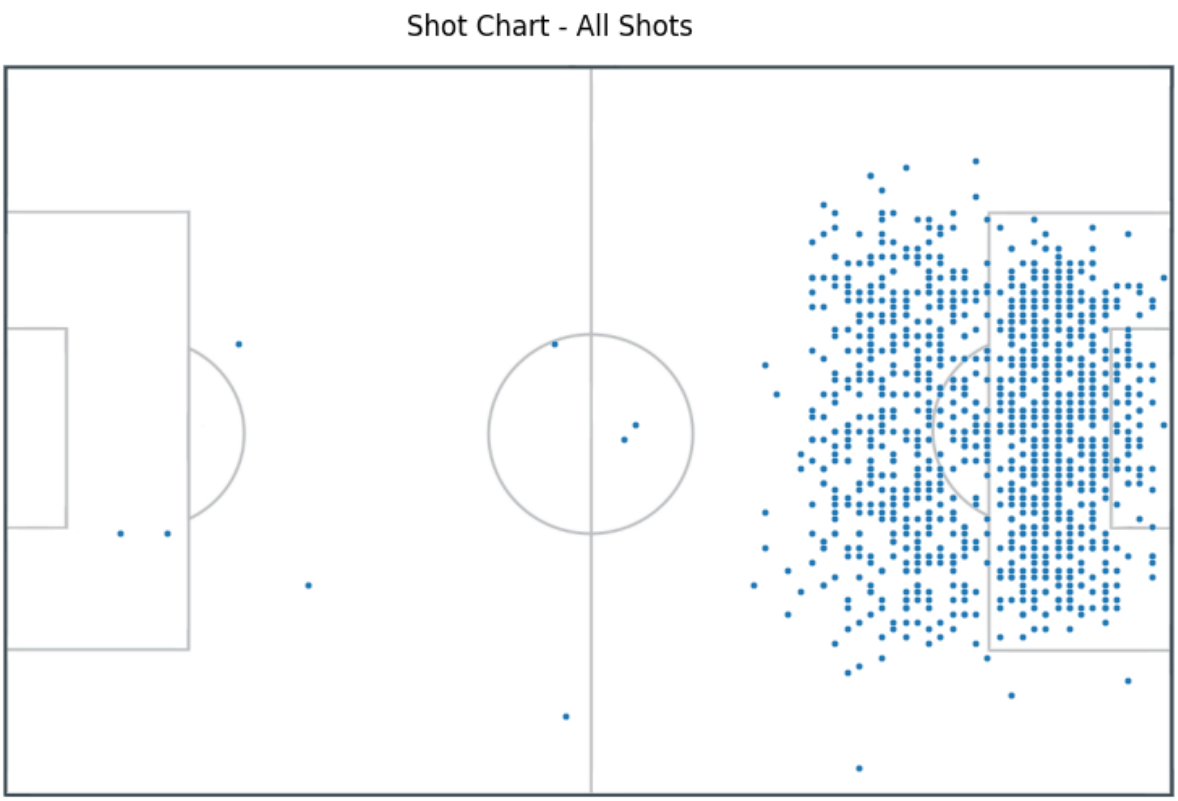

Figure 0.28: Shot Chart - All Data

Nice!

Again, don't worry about the `mpimg` and `ax.imshow` stuff. Just come back to this file when you want to make some shot charts.

For a final touch, let's try adding a bit of jitter:

```
In [7]: dfs['xj'] = dfs['x'].apply(lambda x: x + random.gauss(0, 1))

In [8]: dfs['yj'] = dfs['y'].apply(lambda x: x + random.gauss(0, 1))
```

And take a look:

```
In [9]:
g = sns.relplot(data=dfs, x='xj', y='yj', kind='scatter', s=10)
for ax in g.fig.axes:
    ax.imshow(map_img, zorder=0, extent=[0, 120, 0, 75])
g.set(yticks=[], xticks=[], xlabel=None, ylabel=None)
g.despine(left=True, bottom=True)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

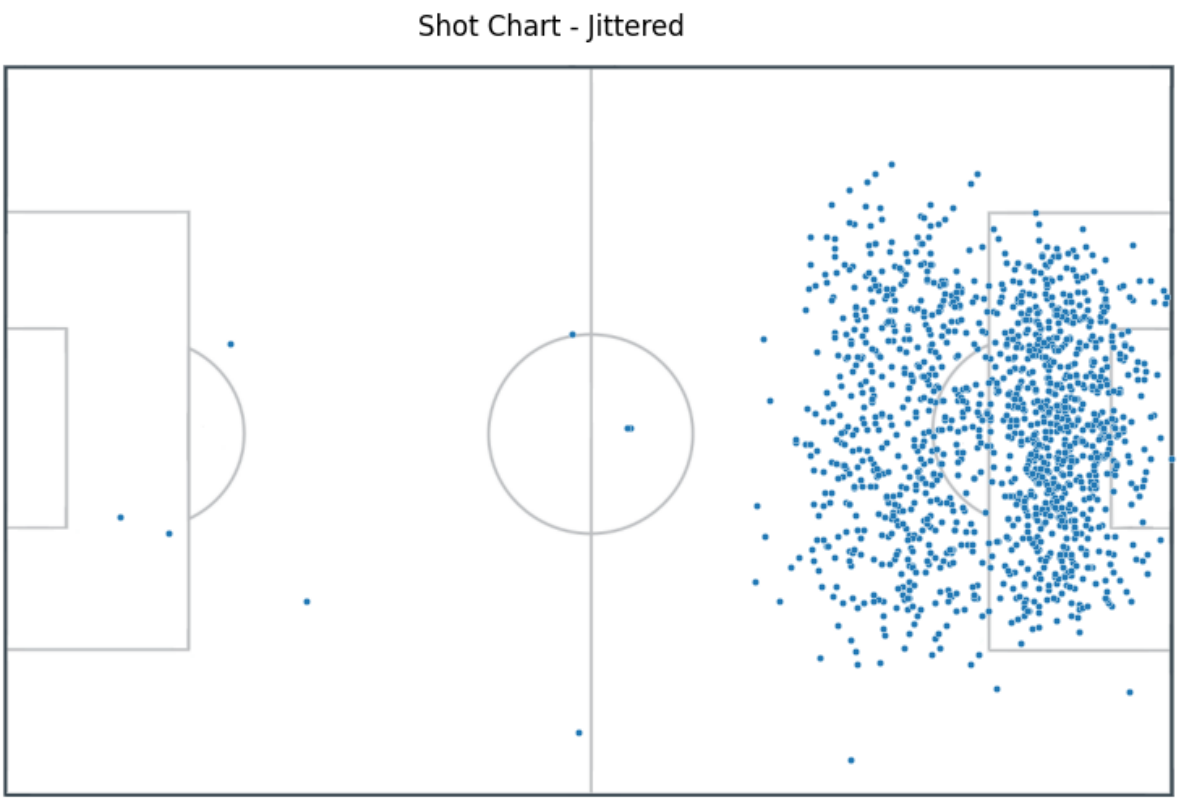

Figure 0.29: Shot Chart - Jittered

To make this easier and extendible, let's put this code in a function:

```
In [10]:
def shot_chart(df, **kwargs):
    g = sns.relplot(data=df, x='xj', y='yj', kind='scatter', **kwargs)
    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=[0, 115, 0, 74])

    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
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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]: shot_chart(dfs, hue='goal', style='goal', s=10)
```

where `hue='goal'`, `style='goal'` and `s=10` (for size of the dots on the graph) are the extra keyword arguments (note there's no argument specifically named `hue` or `style` in `shot_chart`). These are stored in `**kwargs`, and get passed straight to `sns.relplot`, giving us:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

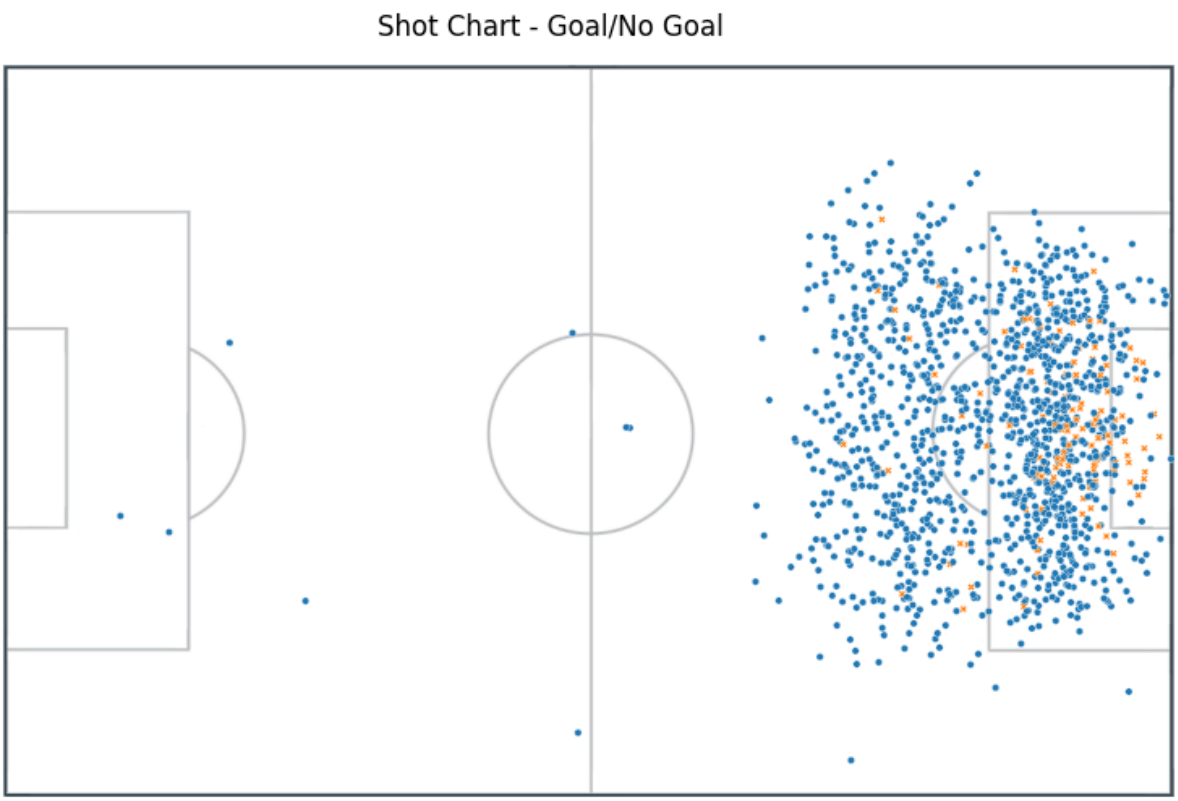

Figure 0.30: Shot Chart - Goal/No Goal

Wee can use all our normal seaborn arguments, including the plot options we talked about in the last section. For example, to plot shot charts by bodypart, we can do:

```
In [9]: shot_chart(dfs, row='foot', hue='foot', s=10)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

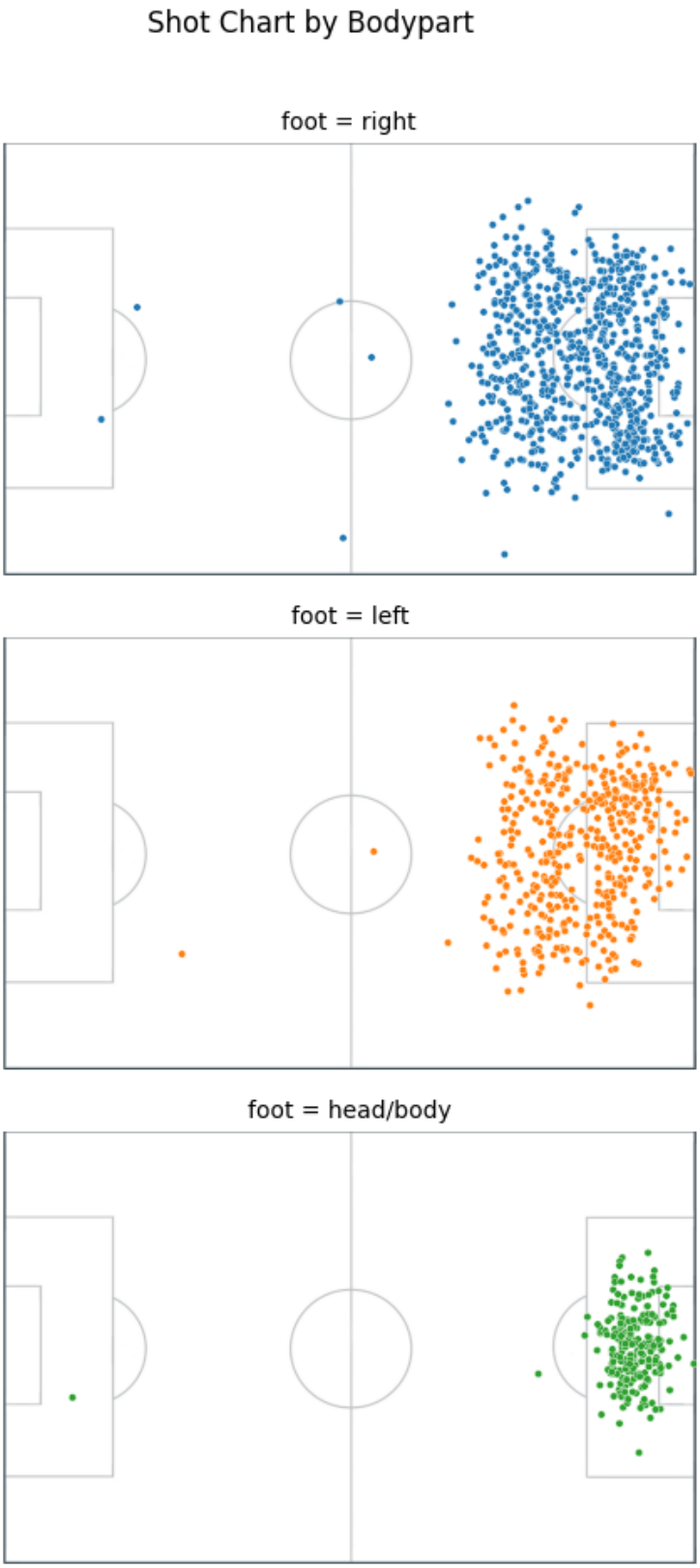

Figure 0.31: Shot Chart by Bodypart

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Contour Plots

One visually interesting tweak is to turn these shot charts with points into *contour* plots. Let's try that with the bodypart plot from above, adding in goal too:

```
In [10]:
g = (sns.FacetGrid(dfs, row='foot', hue='foot', col='goal')
     .map(sns.kdeplot, 'x', 'y', alpha=0.5, shade=True))
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=[0, 120, 0, 75])
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

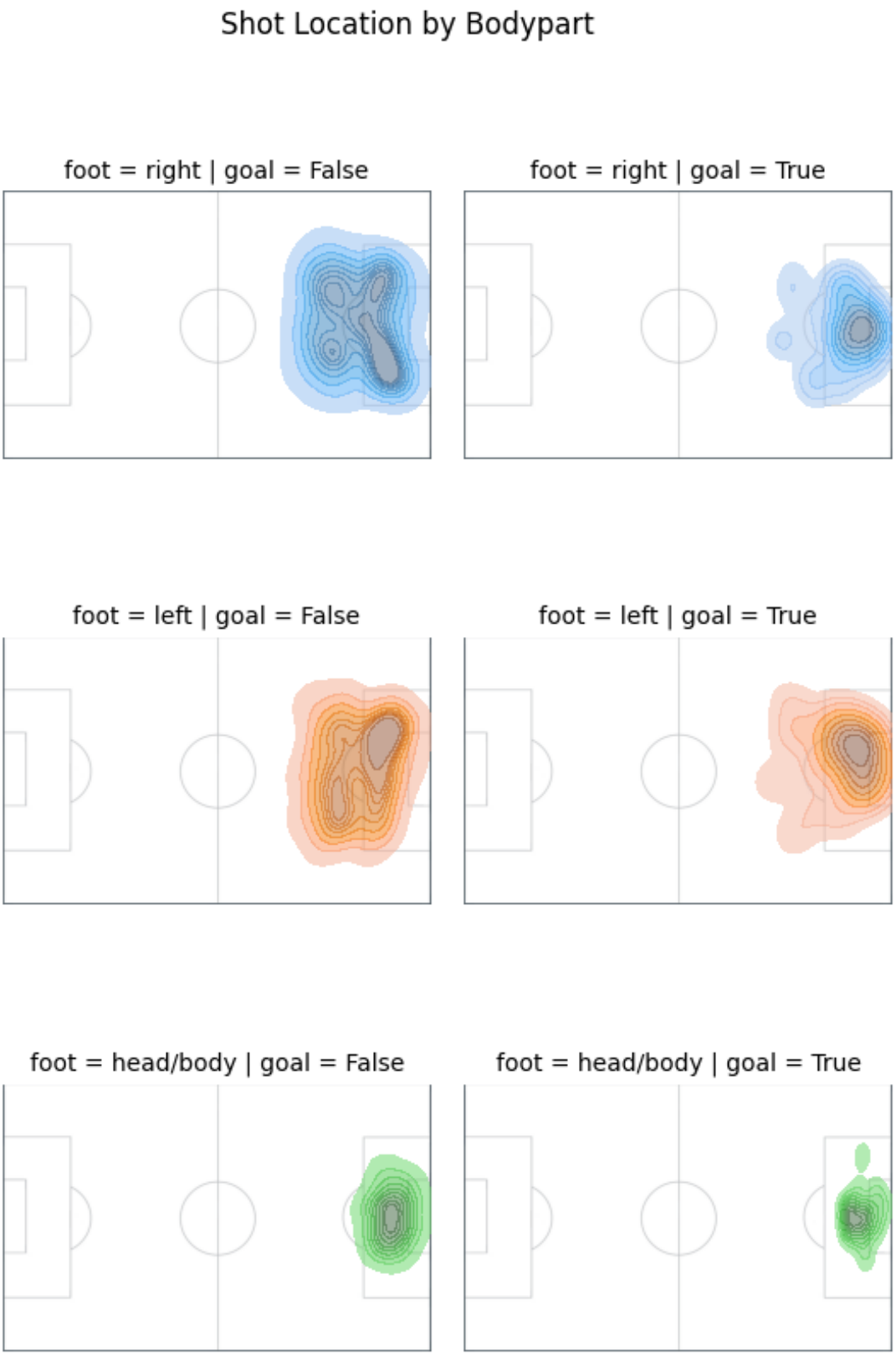

Figure 0.32: Shot Chart by Bodypart

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Note we're back to the `FacetGrid` then `map` plot.

It's also sometimes interesting to do these plots without shading. Let's try separating that by team:

```
In [11]
g = (sns.FacetGrid(dfs, col='team', col_wrap=4, height=2, hue='team')
     .map(sns.kdeplot, 'x', 'y', alpha=0.5))
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=[0, 120, 0, 75])
```

Then they end up looking more like lines on a map:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

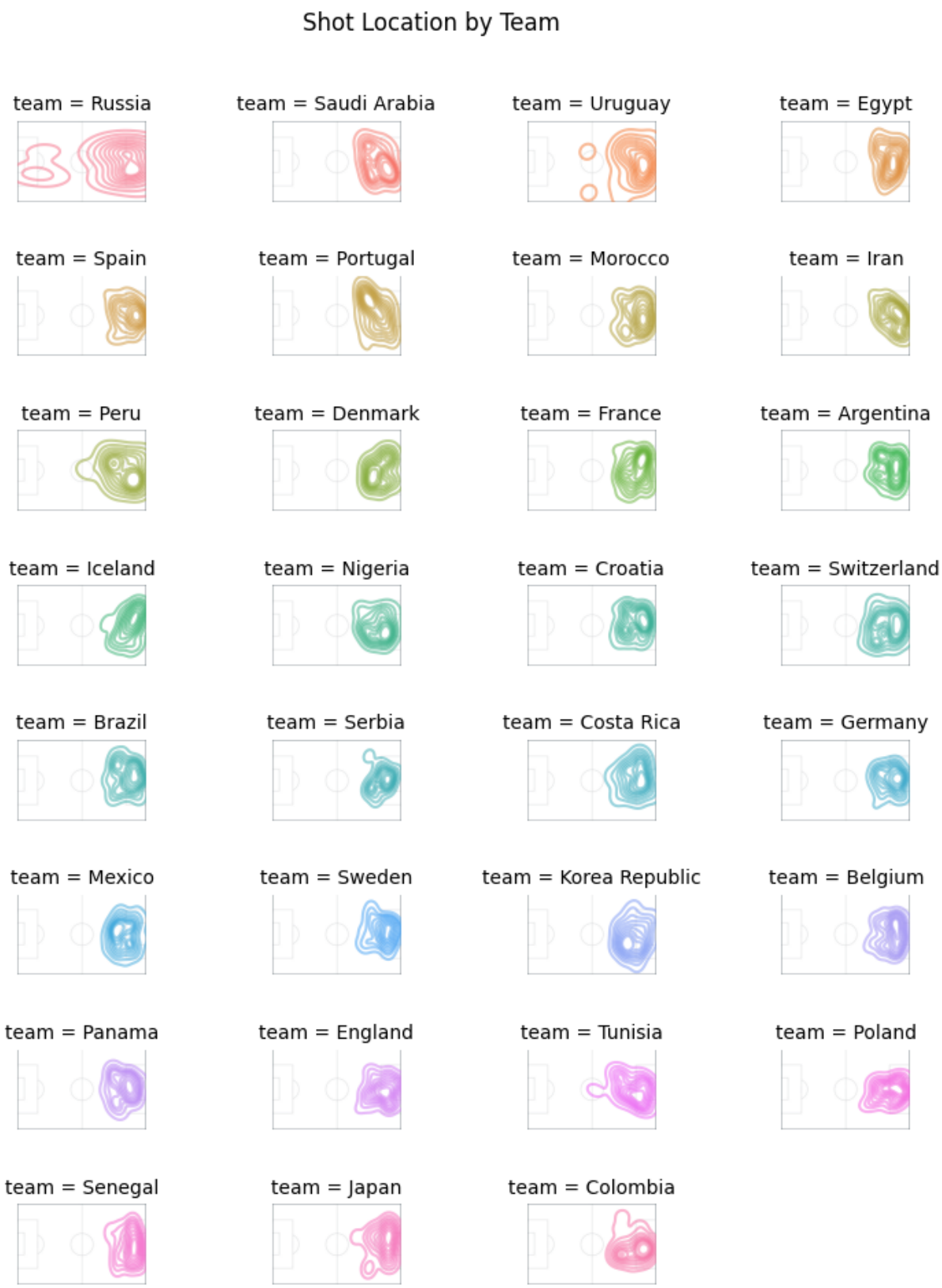

Figure 0.33: Shot Chart by Team

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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) or tag me on Twitter — (@nathanbraun) with any interesting analysis.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

End of Chapter Exercises

6.1

a) Using the team game data, plot the distribution of total passes. Make sure to give your plot a title.

Now modify your plot to show the distribution of passes 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.

(3) Plot the pass distributions by team, with each team is on it's 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 passes and opponents passes. Again, make sure your plot has a title.

(b) Based on this plot, it looks like there's a negative relationship between passes and number of opponent passes. What is the correlation between the two?

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 net and predicts whether a shot from there will be a goal or not. So we might have something like:

```
goal or not = model(meters to the net)
```

Terminology

First some terminology: the variable "goal or not" is our **output variable**[1]. There's always exactly one output variable.

The variable "meters to net" is our **input variable** [2]. In this case we just have one, but we could have as many as we want. For example:

```
goal or not = model(meters to the net, time left in game)
```

OK. Back to:

```
goal or not = model(meters to the net)
```

Here's a question: what is the simplest implementation for `model(...)` we might come up with?

How about:

[1] 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 goal), or *y* variable (traditionally output variables are denoted with y, inputs with x's).

[2] Other words for input variables include: *right hand side*, *independent*, *explanatory*, or *x* variables.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
model(...)= No
```

So give it any yards to go, and our model spits out: "no, it will not be a goal". Since the vast majority of shots don't actually score goals, this model will be very accurate! But since it never says anything besides no, it's not that interesting or useful.

What about:

```
prob goal = 1 + -0.01*distance in m + -0.0000001*distance in m ^ 2
```

So from 1 meter out we'd get a probability of 0.99, 3 meters 0.97, 10 meters 0.90 and 99 meters 0.000002. This is more interesting. I made the numbers up, so it isn't a *good* model (for 50 meters it gives about a 0.50 probability of a goal, 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 `b0` and `b1` instead of `b` and `m`. So we can have:

```
y = b0 + b1*x1 + b2*x2 + ... + ... bn*xn
```

Up to any number n you can think of. As long as it's a bunch of `x*b` terms added together it's a linear equation. Don't get tripped up by the notation: `b0`, `b1`, and `b2` are different numbers, and `x1` and `x2` are different columns of data. The notation just ensures you can include as many variables as you need to (just add another number).

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

In our probability-of-a-goal model that I made up, `x1` was meters from the net, and `x2` was meters from the net squared. We had:

```
prob goal = b0 + b1*(distance in meters)+ b2*(distance in meters ^ 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 goal scored 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.

```
import pandas as pd
import statsmodels.formula.api as smf
from os import path

DATA_DIR = './data'

df = pd.read_csv(path.join(DATA_DIR, 'shots.csv'))

df['dist_m_sq'] = df['dist_m']**2
df['goal'] = df['goal'].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_m_sq` by squaring our `dist_m` 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 `goal` 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 `goal` 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) [3].

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:

[3] Notice even though we have two outcomes — goal 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 `goal`; 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: df[['goal', 'dist_m', 'dist_m_sq']].head()
Out[1]:
   goal     dist_m   dist_m_sq
0     0  12.987566  168.676860
1     0  16.559476  274.216235
2     0  17.013624  289.463402
3     1   8.506812   72.365850
4     0  15.975528  255.217498
```

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 [3]: model = smf.ols(formula='goal ~ dist_m + dist_m_sq', data=df)

In [4]: results = model.fit()
```

Once we've done that, we can look at the results:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [5]: results.summary2()
Out[5]:
"""
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.068
Dependent Variable: goal             AIC:                366.9415
Date:               2022-07-08 13:04 BIC:                382.6004
No. Observations:   1366             Log-Likelihood:     -180.47
Df Model:           2                F-statistic:        50.75
Df Residuals:       1363             Prob (F-statistic): 5.51e-22
R-squared:          0.069            Scale:              0.076425
-------------------------------------------------------------------
               Coef.   Std.Err.     t     P>|t|    [0.025   0.975]
-------------------------------------------------------------------
Intercept      0.2985    0.0229  13.0560  0.0000   0.2537   0.3434
dist_m        -0.0148    0.0017  -8.5768  0.0000  -0.0182  -0.0114
dist_m_sq      0.0001    0.0000   5.0956  0.0000   0.0001   0.0002
------------------------------------------------------------------
Omnibus:             699.924      Durbin-Watson:         1.997
Prob(Omnibus):       0.000        Jarque-Bera (JB):      3093.164
Skew:                2.559        Prob(JB):              0.000
Kurtosis:            8.305        Condition No.:         2122
==================================================================
* The condition number is large (2e+03). This might indicate
strong multicollinearity or other numerical problems.
"""
```

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 goal when you're right next to — 0 feet away — the net. The other coefficients are next to `dist_m` and `dist_m_sq`.

So instead of my made up formula from earlier, the formula that best fits this data is:

`0.2985 + -0.0148*meters + 0.0001*(meters ^ 2)`.

Let's test it out with some values:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [6]:
def prob_of_goal(meters):
    b0, b1, b2 = results.params
    return (b0 + b1*meters + b2*(meters**2))

In [7]: prob_of_goal(1)
Out[7]: 0.2838389299586142

In [8]: prob_of_goal(15)
Out[8]: 0.1088113102073702

In [9]: prob_of_goal(25)
Out[9]: 0.018905161012311905
```

Seems reasonable. Let's use the `results.predict` method to predict it for every value of our data.

```
In [10]: df['goal_hat'] = results.predict(df)

In [11]: df[['goal', 'goal_hat']].head(5)
Out[11]:
   goal  goal_hat
0     0  0.068017
1     0  0.023154
2     0  0.040137
3     1  0.178944
4     0  0.100453
```

We can see, for the first five observations, our `goal_hat` was the highest on the shot that was actually a goal. Not bad.

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[4].

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.

[4]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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
coin = ['H', 'T']

# make empty DataFrame
df = DataFrame(index=range(100))

# now fill it with a "guess" and a "flip"
df['guess'] = [random.choice(coin) for _ in range(100)]
df['result'] = [random.choice(coin) for _ in range(100)]

# did we get it right or not?
df['right'] = (df['guess'] == df['result']).astype(int)
```

Now let's run a regression on it:

```
model = smf.ols(formula='right ~ C(guess)', data=df)
results = model.fit()
results.summary2()
"""
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.006
Dependent Variable: right            AIC:                146.5307
Date:               2019-07-22 14:09 BIC:                151.7411
No. Observations:   100              Log-Likelihood:     -71.265
Df Model:           1                F-statistic:        1.603
Df Residuals:       98               Prob (F-statistic): 0.208
R-squared:          0.016            Scale:              0.24849
-------------------------------------------------------------------
                Coef.   Std.Err.     t     P>|t|    [0.025  0.975]
-------------------------------------------------------------------
Intercept       0.6170    0.0727   8.4859  0.0000   0.4727  0.7613
C(guess)[T.T]  -0.1265    0.0999  -1.2661  0.2085  -0.3247  0.0717
-----------------------------------------------------------------
Omnibus:             915.008       Durbin-Watson:          2.174
Prob(Omnibus):       0.000         Jarque-Bera (JB):       15.613
Skew:                -0.196        Prob(JB):               0.000
Kurtosis:            1.104         Condition No.:          3
==================================================================

"""
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 meters from the net and probability of a 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 net and probability of a shot going in.
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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

So either this was a major, major fluke, or how close you are to the net 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 `foot`, which is either `'left'`, `'right'` or `'head/body'` depending on which foot or bodypart the shooter used for the shot.

Let's use that to see how a heading the ball 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 20, after you've loaded the shot data into a DataFrame named `dfs` and done some light processing, including making a `header` variable.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]:
model = smf.ols(formula=
        """
        goal ~ header
        """, data=dfs)
results = model.fit()
results.summary2()

Out[1]:
"""
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.005
Dependent Variable: goal             AIC:                455.5472
Date:               2022-06-07 10:31 BIC:                465.9865
No. Observations:   1366             Log-Likelihood:     -225.77
Df Model:           1                F-statistic:        7.521
Df Residuals:       1364             Prob (F-statistic): 0.00618
R-squared:          0.005            Scale:              0.081606
-------------------------------------------------------------------
                  Coef.   Std.Err.    t     P>|t|   [0.025  0.975]
-------------------------------------------------------------------
Intercept         0.0806    0.0085  9.5342  0.0000  0.0640  0.0972
header[T.True]    0.0571    0.0208  2.7425  0.0062  0.0163  0.0980
-----------------------------------------------------------------
Omnibus:             778.096      Durbin-Watson:         1.990
Prob(Omnibus):       0.000        Jarque-Bera (JB):      3990.224
Skew:                2.842        Prob(JB):              0.000
Kurtosis:            9.148        Condition No.:         3
==================================================================
"""
```

Let's look the coefficients to practice reading them. According to this, attempting a header increases the probability a shot goes in by 0.0571. This is pretty large, considering non-headers (according the intercept) have a probability of 0.0806.

But let's think about header attempts for a second. On average, we'd expect them to be attempted a lot closer to the goal. Looking at the data that's what we see:

```
In [2]: dfs.groupby('header')['dist_m'].mean()
Out[2]:
header
False    19.595073
True     10.639813
Name: dist_m, dtype: float64
```

On average, headers come are 10 feet from the net, vs regular shots, which are almost 20. But what if we want to quantify *just* the impact of taking a header (not distance) on shot probability.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 headers are taken a lot closer to the goal, 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=
        """
        goal ~ header + dist_m
        """, data=dfs)
results = model.fit()
results.summary2()
--
Out[3]:
"""
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.050
Dependent Variable: goal             AIC:                392.6176
Date:               2022-07-08 13:08 BIC:                408.2765
No. Observations:   1366             Log-Likelihood:     -193.31
Df Model:           2                F-statistic:        37.12
Df Residuals:       1363             Prob (F-statistic): 2.01e-16
R-squared:          0.052            Scale:              0.077875
------------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
------------------------------------------------------------------
Intercept          0.2208   0.0191 11.5668 0.0000  0.1834  0.2583
header[T.True]    -0.0069   0.0218 -0.3176 0.7509 -0.0497  0.0359
dist_m            -0.0072   0.0009 -8.1456 0.0000 -0.0089 -0.0054
------------------------------------------------------------------
Omnibus:             721.248      Durbin-Watson:         1.985
Prob(Omnibus):       0.000        Jarque-Bera (JB):      3281.097
Skew:                2.644        Prob(JB):              0.000
Kurtosis:            8.449        Condition No.:         68
==================================================================
"""
```

Now that distance from the net is explicitly accounted for, we know that the `header` coefficient measures *only* the impact on probability of shooting with your head (vs a foot).

And we can see the coefficient changes sign! It goes from 0.0507 to -0.0069. So, yeah on the surface, headers go in more often than regular shots, but we can see here that's because, on average, they're usually taken so much closer to the goal.

What this means is if — from any given distance — a player has a choice between taking a header or kicking the ball, attempting a header isn't going to increase the probability of a goal. It might even

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

hurt.

Note: I say *might* hurt here, because — even though the coefficient on `header` is negative — it's not statistically significant (the p-value is 0.7509).

Let's look at the other coefficients to practice reading them. The *Intercept* (sometimes denoted `b0`) is 0.2208. This says the probability of making a shot that's 0 feet away from the nt, that's *not* header, is 0.2208.

The coefficient on `dist` says every meter away from the net lowers our the probability the shot goes in by 0.0072. So if we're 20 feet away from the net, our probability of making a regular (non-header) shot is:

```
In [5]: 0.2208 -0.0069*20
Out[5]: 0.0828
```

And a header from 9 meters out:

```
In [6]: 0.2208 -0.0069*20 - 0.0069
Out[6]: 0.0759
```

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 whether a shot was a `goal` or not, but what body part is was taken with (left foot, right foot or head).

These are called categorical variables or "fixed effects" and the way we handle them is by putting our one categorical variable (bodypart) into a series of dummies that give us the same information (`is_left`, `is_right`, `is_head`).

So `is_head` is `1` if a shot was a header, `0` otherwise, etc. Except we don't need *all* of these. If there are only 3 shot types in our data (left foot, right foot, header) every shot has to be one — if a shot wasn't taken with a playres left foot or head, then we know it must be a right footed shot. 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [1]: pd.get_dummies(dfs['foot']).head()
Out[1]:
   head/body  left  right
0          0     0      1
1          0     0      1
2          0     1      0
3          1     0      0
4          0     1      0
```

Again, *all* of these variables would be redundant — there are only three options downs, so you can pass the `drop_first=True` argument to `get_dummies` to have it return only two 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="goal ~ C(foot) + dist_m + dist_m_sq", data=dfs)
results = model.fit()
results.summary2()

Out[2]:
"""
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.069
Dependent Variable: goal             AIC:                367.9501
Date:               2022-07-08 13:11 BIC:                394.0483
No. Observations:   1366             Log-Likelihood:     -178.98
Df Model:           4                F-statistic:        26.14
Df Residuals:       1361             Prob (F-statistic): 6.61e-21
R-squared:          0.071            Scale:              0.076370
------------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
------------------------------------------------------------------
Intercept          0.2840   0.0244 11.6350 0.0000  0.2361  0.3319
C(foot)[T.left]    0.0413   0.0247  1.6695 0.0952 -0.0072  0.0899
C(foot)[T.right]   0.0360   0.0233  1.5488 0.1217 -0.0096  0.0817
dist_m            -0.0161   0.0019 -8.5633 0.0000 -0.0198 -0.0124
dist_m_sq          0.0002   0.0000  5.3680 0.0000  0.0001  0.0002
------------------------------------------------------------------
Omnibus:             698.222      Durbin-Watson:         2.005
Prob(Omnibus):       0.000        Jarque-Bera (JB):      3077.477
Skew:                2.553        Prob(JB):              0.000
Kurtosis:            8.292        Condition No.:         3134
==================================================================
* The condition number is large (3e+03). This might indicate
strong multicollinearity or other numerical problems.
"""
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Let's zoom in on our fixed effect coefficients:

```
------------------------------------------------------------------
                    Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
------------------------------------------------------------------
Intercept           0.2840   0.0244 11.6350 0.0000  0.2361  0.3319
C(foot)[T.left]     0.0413   0.0247  1.6695 0.0952 -0.0072  0.0899
C(foot)[T.right]    0.0360   0.0233  1.5488 0.1217 -0.0096  0.0817
dist_m             -0.0161   0.0019 -8.5633 0.0000 -0.0198 -0.0124
dist_m_sq           0.0002   0.0000  5.3680 0.0000  0.0001  0.0002
```

Again, including *all* shot types would be redundant, so `statsmodel` automatically dropped one, in this case headers.

Let's think about this. How do these dummies look for observations where `foot='right'`? Well, when `C(foot)[T.right]` (`statsmodel`'s fancy way of a saying `is_right`) is 1, the other shot columns are 0. When a row has `C(foot)[T.right]`, that column is 1, everything else is 0.

Fine. But what about headers? In this case we know the shot was a header when both the right and left foot columns above are 0. Header is our default, baseline case, even though they happen the least often.

```
In [3]: dfs['foot'].value_counts()
Out[3]:
right        716
left         425
head/body    225
Name: foot, dtype: int64
```

That means we need to right and left relative to header. How does a left footed shot change the probability of scoring? It increases it 0.0431 compared to a header. What about right? 0.0360 vs a header.

It won't actually change the model at all, but if we wanted, we could have `statsmodel` drop a different shot type, say right footed shots.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [4]:
model = smf.ols(
    formula="goal ~ C(foot, Treatment(reference='right')) + dist_m +
       dist_m_sq",
    data=dfs)
results = model.fit()
results.summary2()

Out[4]:
"""
                     Results: Ordinary least squares
============================================================
Model:               OLS                 Adj. R-squared: 0.069
Dependent Variable:  goal                AIC:         367.9501
Date:                2022-07-08 13:12    BIC:         394.0483
No. Observations:    1366                Log-Likelihood:  -178
Df Model:            4                   F-statistic:    26.14
Df Residuals:        1361                Prob (F):    6.61e-21
R-squared:           0.071               Scale:         0.0763
-------------------------------------------------------------
                Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
-------------------------------------------------------------
Intercept       0.3200   0.0274 11.6610 0.0000  0.2662  0.3739
C[T.head]      -0.0360   0.0233 -1.5488 0.1217 -0.0817  0.0096
C[T.left]       0.0053   0.0169  0.3122 0.7549 -0.0279  0.0385
dist_m         -0.0161   0.0019 -8.5633 0.0000 -0.0198 -0.0124
dist_m_sq       0.0002   0.0000  5.3680 0.0000  0.0001  0.0002
-------------------------------------------------------------
Omnibus:              698.222       Durbin-Watson:       2.005
Prob(Omnibus):        0.000         Jarque-Bera (JB):   3077.4
Skew:                 2.553         Prob(JB):            0.000
Kurtosis:             8.292         Condition No.:        2993
================================================================
"""
```

Now we can interpret our fixed effects coefficients as probability added compared to a right footed shot. So a header decreases it by 0.036, and shooting with your left foot doesn't change much (coefficient is 0.0053, but it's not sigficant).

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-goal probability model:

```
prob scoring = b0 + b1*dist
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

By modeling this as a linear relationship, we're assuming a one unit change in distance always has the *same* effect on probability of scoring a goal. This effect (`b1`) is the same whether we're going from 1 to 2 meters from the net, or 25 to 26 meters.

Is this a good assumption? In this case probably not.

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 = b0 + x1*b1 + x2*b2 ... xn*bn`), 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 `b2` doesn't come into play as much — later it does [5].

In the model without squared terms, `goal probability = b0 + b1*distance`, this derivative is just `b1`, which should match your intuition. One increase in distance leads to a `b1` decrease in goal probability.

With squared terms like `goal probabilitiy = b0 + b1*distance + b2*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 `b1 + 2*b2*(distance 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 let's 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 `b1` as the decrease in probability of making a shot associated with moving one more foot away from the hoop.

[5] *Calculus* is a branch of math that's all about analyzing how a change in one variable affects another. In calculus parlance, saying "how probablity of scoring a goal changes as distance changes" is the same as saying, "the derivative of goal probability with respect to distance".

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

If we did:

```
prob scoring = b0 + b1*ln(dist)
```

We'd interpret `b1` as a the decrease in goal probability given a one *percent change* in shot distance.

Let's run this. First we need to calculate the natural log of our variable:

```
In [1]: dfs['ln_dist'] = np.log(dfs['dist_m'])
```

Running the regression:

```
In [2]:
model = smf.ols(formula='goal ~ ln_dist', data=dfs)
results = model.fit()
results.summary2()

Out[2]:
"""
                  Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.082
Dependent Variable: goal             AIC:                344.8149
Date:               2022-07-08 13:16 BIC:                355.2542
No. Observations:   1366             Log-Likelihood:     -170.41
Df Model:           1                F-statistic:        123.3
Df Residuals:       1364             Prob (F-statistic): 1.71e-27
R-squared:          0.083            Scale:              0.075252
-------------------------------------------------------------------
              Coef.   Std.Err.     t      P>|t|    [0.025   0.975]
-------------------------------------------------------------------
Intercept     0.5336    0.0406   13.1350  0.0000   0.4539   0.6133
ln_dist      -0.1600    0.0144  -11.1055  0.0000  -0.1883  -0.1317
------------------------------------------------------------------
Omnibus:             692.218      Durbin-Watson:         1.989
Prob(Omnibus):       0.000        Jarque-Bera (JB):      3049.133
Skew:                2.524        Prob(JB):              0.000
Kurtosis:            8.300        Condition No.:         17
==================================================================
```

We can see getting 1% further away from the goal lowers our probability of making the shot by about 0.16.

Note you can log an input variable (like distance above) or an output variable. If we were working with tournament-team totals and had some regression like:

```
ln(wins)= b0 + b1*ln(goals scored)+ b2*ln(goals allowed)
```

Then `b1` would be the *percent change* in wins given a one *percent change* in total goals scored.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 goal probability as a function of body part (left foot, right foot, head) and distance from the basket.

But by including these variables separately (our body part 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 goal on a header has a bigger impact that neing an extra foot away on a regular kick.

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:

```
goal prob = b0 + b1*dist
```

To this:

```
goal prob = b0 + b1*dist + b2(dist*is_header)
```

Then `b1` is the impact of being an extra foot away, an and `b1` + `b2` is the effect of being an one foot away on headers specifically.

Let's run this regression:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [3]: dfs['is_header'] = dfs['foot'] == 'head/body'

In [4]:
model = smf.ols(formula=
        """
        goal ~ dist_m + dist_m:is_header
        """, data=dfs)
results = model.fit()
results.summary2()


Out[4]:
"""
                    Results: Ordinary least squares
====================================================================
Model:                 OLS                Adj. R-squared:       0.050
Dependent Variable:    goal               AIC:                  392.6703
Date:                  2022-07-08 13:16   BIC:                  408.3293
No. Observations:      1366               Log-Likelihood:       -193.34
Df Model:              2                  F-statistic:          37.09
Df Residuals:          1363               Prob (F-statistic):   2.07e-16
R-squared:             0.052              Scale:                0.077878
--------------------------------------------------------------------
                          Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
--------------------------------------------------------------------
Intercept                  0.2189   0.0173 12.6256 0.0000  0.1849  0.2530
dist_m                    -0.0071   0.0008 -8.5607 0.0000 -0.0087 -0.0055
dist_m:is_header[T.True] -0.0004   0.0016 -0.2196 0.8262 -0.0035  0.0028
--------------------------------------------------------------------
Omnibus:                721.388        Durbin-Watson:           1.984
Prob(Omnibus):          0.000          Jarque-Bera (JB):        3282.767
Skew:                   2.644          Prob(JB):                0.000
Kurtosis:               8.451          Condition No.:           47
====================================================================

"""
```

So we can see via the coefficient on `dist_m:is_header[T.True]` that every extra foot away from the goal lowers the shot probability an additional -0.0015 for headers 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 goals scored or number of shots.

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*.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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=
        """
        goal ~ dist_m + dist_m:is_header
        """, data=dfs)
logit_results = model.fit()
logit_results.summary2()

Optimization terminated successfully.
         Current function value: 0.255887
         Iterations 8
Out[1]:
"""
                            Results: Logit
=====================================================================
Model:                 Logit              Pseudo R-squared:   0.128
Dependent Variable:    goal               AIC:                727.2096
Date:                  2022-07-08 13:17   BIC:                742.8686
No. Observations:      1366               Log-Likelihood:     -360.60
Df Model:              2                  LL-Null:            -413.41
Df Residuals:          1363               LLR p-value:        1.1721e-23
Converged:             1.0000             Scale:              1.0000
No. Iterations:        8.0000
---------------------------------------------------------------------
                          Coef.  Std.Err.    z    P>|z|   [0.025  0.975]
---------------------------------------------------------------------
Intercept                 0.0859   0.2601  0.3301 0.7413 -0.4239  0.5956
dist_m                   -0.1593   0.0187 -8.5188 0.0000 -0.1959 -0.1226
dist_m:is_header[T.True] -0.0446   0.0245 -1.8179 0.0691 -0.0926  0.0035
=====================================================================

"""
```

Now to calculate the probability of making a shot given some distance (and header indicator), we need to do something similar to before, but then run it through the logistic function.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [2]:
def prob_goal_logit(dist, is_header):
    b0, b1, b2  = logit_results.params
    value = (b0 + b1*dist + b2*dist*is_header)
    return 1/(1 + math.exp(-value))

--

In [3]: prob_goal_logit(20, 0)
Out[3]: 0.0431088596283838

In [4]: prob_goal_logit(14, 1)
Out[4]: 0.05907488159031021

In [5]: prob_goal_logit(14, 0)
Out[5]: 0.10487284058049676
```

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 classify player position (FWD, MID, DEF, GK) based on some game stats maybe the first split is on "number of shots taken", players who took 1 or more shots go one direction, and players who took 0 go another way.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Let's follow the 1+ split. We could go to another split that looks at number of accurate passes — if it's above the split point (say 25 passes) we'll MID, below that FWD.

Meanwhile the 0 shot branch also continues onto its own, different split. Maybe it it goes to throw ins — if a player had more than X, 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).

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 MID, 15 FWD and 5 DEF 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 player match data into a DataFrame named `df`, and doing some processing.

In this example, we'll try to classify position (defender, midfielder, forward, goalkeeper) given location and time left in the game data. That is, we'll model shot type as a function of:

```
In [1]:
xvars = ['shot', 'goal', 'assist', 'pass', 'pass_accurate', 'tackle', '
    accel',
         'counter', 'opportunity', 'keypass', 'own_goal', 'interception',
         'smart', 'clearance', 'cross', 'air_duel', 'air_duel_won',
         'gk_leave_line', 'gk_save_attempt', 'throw', 'corner', 'started']

In [2]: yvar = 'pos'
```

Let's look at a few of these observations:

```
In [3]: df[xvars + [yvar]].head()
Out[3]:
   shot  goal  assist  pass  ...  throw  corner  started  pos
0     3     2       0    22  ...      0       0    False  MID
1     0     0       0    26  ...     16       0     True  DEF
2     0     0       0    17  ...      0       0     True  GKP
3     0     0       0    26  ...      0       0     True  DEF
4     0     0       0     8  ...      0       0     True  MID
```

Along with a look at the variable we're predicting:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [4]: df[yvar].value_counts(normalize=True)
Out[4]:
MID    0.381209
DEF    0.318372
FWD    0.229803
GKP    0.070616
```

So we're going to use our input variables, number of: shots, goals, assists, etc to predict our output variable player position.

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['pos_hat'] = model.predict(test[xvars])

In [8]: test['correct'] = (test['pos_hat'] == test[yvar])

In [9]: test['correct'].mean()
Out[9]: 0.7134328358208956
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

About 71%, 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: MID, FWD, DEF, GK. If the model assigned some player MID for 51/100 trees and FWD for the other 49/100, we can interpret it as relatively unsure in it's 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([[0.03, 0.71, 0.  , 0.26],
       [0.02, 0.37, 0.  , 0.61],
       [0.07, 0.61, 0.  , 0.32],
       ...,
       [0.66, 0.02, 0.  , 0.32],
       [0.21, 0.04, 0.  , 0.75],
       [0.23, 0.  , 0.  , 0.77]])
```

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 [11]:
probs = DataFrame(model.predict_proba(test[xvars]),
                  index=test.index,
                  columns=model.classes_)

In [12]: probs.head()
Out[12]:
       DEF   FWD  GKP   MID
1423  0.03  0.71  0.0  0.26
823   0.02  0.37  0.0  0.61
113   0.07  0.61  0.0  0.32
788   0.66  0.06  0.0  0.28
759   0.04  0.76  0.0  0.20
```

We're looking at the first 5 rows of our holdout dataset here. We can see the model says the first observation has a 71% of being a forward, a 26% of being a midfielder, and 3% of being a defender.

Let's bring in the actual, known position from our test dataset.

```
In [11]:
results = pd.concat([
    test[['name', 'pos', 'pos_hat', 'correct']],
    probs], axis=1)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

We can look at a few examples:

```
In [12]: results.sample(10).round(2)
Out[12]:
           name       team  pos pos_hat  correct   DEF   FWD   GKP   MID
    Diego Costa      Spain  FWD     MID    False  0.04  0.42  0.03  0.51
 A. Christensen   Denmark  DEF     DEF     True  0.89  0.04  0.00  0.07
    Tarek Hamed      Egypt  MID     MID     True  0.12  0.02  0.00  0.86
      S. Coates   Uruguay  DEF     MID    False  0.43  0.01  0.00  0.56
       D. ćTadi    Serbia  MID     MID     True  0.15  0.10  0.01  0.74
      M. Fabián    Mexico  MID     FWD    False  0.00  0.67  0.00  0.33
        M. Berg    Sweden  FWD     FWD     True  0.01  0.89  0.00  0.10
       M. Borja  Colombia  FWD     FWD     True  0.02  0.91  0.00  0.07
  Cédric Soares  Portugal  DEF     DEF     True  0.94  0.01  0.00  0.05
        C. Vela    Mexico  FWD     MID    False  0.04  0.43  0.01  0.52
```

And also aggregate to see how our model did overall for different positions.

```
In [12]:
results.groupby('pos')[['correct', 'FWD', 'MID', 'DEF', 'GKP']].mean()

Out[12]:
      correct       FWD       MID       DEF       GKP
pos
DEF  0.785714  0.091453  0.238801  0.665868  0.003878
FWD  0.569620  0.519512  0.375451  0.099594  0.005443
GKP  1.000000  0.007917  0.034583  0.014167  0.943333
MID  0.694030  0.218276  0.559252  0.214561  0.007910
```

Not surprisingly, the model performs the best on goalkeepers, getting all of them correct. It has the hardest time distinguishing between forwards and midfielders, which again is somethign we'd expect.

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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([0.79761905, 0.77245509, 0.71257485, 0.69461078, 0.73053892,
       0.64670659, 0.76646707, 0.74251497, 0.68862275, 0.71257485])

In [4]: scores.mean()
Out[4]: 0.7264684915882521
```

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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(df[xvars], df[yvar])
Out[2]: RandomForestClassifier()
```

Then to look at the feature importances:

```
In [3]: Series(model.feature_importances_, xvars).sort_values(ascending=
   False)
Out[3]:
pass_accurate      0.114883
pass               0.110250
throw              0.104457
clearance          0.083418
air_duel           0.071086
interception       0.068678
air_duel_won       0.048437
counter            0.048119
gk_save_attempt    0.047468
shot               0.043869
gk_leave_line      0.041971
cross              0.041059
opportunity        0.036506
accel              0.029256
goal               0.025802
tackle             0.023815
corner             0.023619
keypass            0.017150
started            0.014158
assist             0.004856
own_goal           0.001143
smart              0.000000
```

So number of passes (and accurate passes) were most important, followed by number of throw ins. Makes sense.

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 goals or passes) you do the exact same thing, but with `RandomForestRegressor` instead.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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[6].

[6] Of course, there are other, more advanced models than Random Forest available in scikit-learn too. See the documentation for more.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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_goal` 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(period)` to the probability of scoring model, and look the results, What period is a shot most likley to go in?

c) What's your best guess at to why the coefficients on extra periods are not significantly different from 0?

d) Run the same model without the `C(period)` syntax, creating a variables for period 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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-match data into a DataFrame named `dftm`.

a) Run a logit model regressing number of shots and passes, on whether a team wins. But before you do – make prediction about what you expect to to see for the coefficients? (Note: for some reason, statsmodels throws an error if you include a variable named 'pass' in your model so you'll have to rename it).

b) What do you think will happen to the other coefficients if you add goals scored to the model? Try it.

7.4

a) Use your team-game data to build a random forest classification model that predicts team based on stats.

b) Run a cross validation on your model.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 so 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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

DRY: Don't Repeat Yourself

A popular maxim among programmers is "DRY" for Don't Repeat Yourself [1].

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.

[1] DRY comes from a famous (but old) book called The Pragmatic Programmer, which is good, but first came out in 1999 and is a bit out of date technically. It also takes a bit of a different (object oriented) approach that we do here

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

First, it's helpful to take a sort of "pride" (pride isn't exactly the right word but it's close) in your code. You should appreciate and care about well designed, functioning code and strive to write it yourself. The guys who coined DRY talk about this as a sense of *craftsmanship*.

Of course, your standards for whats well-designed will change over time, but that's OK.

Second, you should be continuously growing and improving. You'll be able to do more faster if you deliberately try to get better as a programmer — experimenting, learning and pushing yourself to write better code — especially early on.

One good sign you're doing this well is if you can go back to code you've written in the past and are able to tell approximately when you wrote it.

For example, say we want to modify this dictionary:

```
roster_dict = {'CB': 'ruben dias',
               'CF': 'gabriel jesus',
               'RW': 'riyad mahrez'}
```

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]: {'CB': 'RUBEN DIAS', 'CF': 'GABRIEL JESUS', 'RW': 'RIYAD MAHREZ'}
```

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]: {'CB': 'RUBEN DIAS', 'CF': 'GABRIEL JESUS', 'RW': 'RIYAD MAHREZ'}
```

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]: {'CB': 'RUBEN DIAS', 'CF': 'GABRIEL JESUS', 'RW': 'RIYAD MAHREZ'}
```

This illustrates a few points:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 football. We've covered a lot of material, and there many ways you could go from here.

I'd recommend starting on your own analysis ASAP (see the appendix for a few ideas on places to look for data). Especially if you're self-taught, diving in and working on your own projects is by far the fastest and most fun way to learn.

When I started building the website that became www.fantasymath.com I knew nothing about Python, Pandas, SQL, web scraping, or machine learning. I learned all of it because I wanted to beat my friends in fantasy football.

The goal of this book has been to make things easier for people who feel the same way. Judging by the response so far, there are a lot of you. I hope I've been successful, but if you have any questions, errata, or other feedback, don't hesitate to get in touch — nate@nathanbraun.com

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Appendix A: Places to Get Data

This appendix lists a few places to get football data.

Detailed Academic Event Data

Researchers in Italy, with funding from the EU, have released a free, open source set of in-depth event data here:

https://figshare.com/collections/Soccer_match_event_dataset/4415000/5

A description of it is in the article, A public data set of spatio-temporal match events in football competitions

This is where I got the 2018 World Cup Data we've been using in this book. In all, this dataset has extremely comprehensive data on the following competitions:

- 2018 World Cup
- 2016 European Cup
- 2017-2018 data on first divisions for the English, German, French, Spanish and Italian leagues

All the data is at the figshare link above, but it's in JSON, and I had to do a decent amount of processing to get it in the format we use in this book. You should be able to do something similar after reading the Python and Pandas chapters here, but at some point I might try to process this and put up easier-to-access versions (or at least my code that does that).

The only problem with this data is that I'm not familiar with any ongoing/current efforts to update it (e.g. for the 2022 World Cup or ongoing Eurpoean leagues). But I'll keep an eye out.

Datahub's list

DataHub.io has a large list of football data sources:

https://datahub.io/collections/football

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Premier League Fantasy API

If you're interested in the Premier League, the best place to get updated data is probably the Premier League Fantasy API. See chapter 5 for an in-depth look at it.

https://fantasy.premierleague.com/api/bootstrap-static/

Other Options

Kaggle.com

Kaggle.com is best known for its modeling and machine learning competitions, but it also has a dataset search engine with some football related datasets.

https://www.kaggle.com/datasets

Google Dataset Search

Google has a dataset search engine with some interesting datasets:

https://toolbox.google.com/datasetsearch

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

Anki is optional — all the material in the cards is also in the book — but I strongly recommend at least trying it out.

If you're on the fence, here's a good essay by YCombinator's Michael Nielson for inspiration:

http://augmentingcognition.com/ltm.html

Like Nielsen, I personally have hundreds of Anki cards covering anything I want to remember long term — programming languages (including some on Python and Pandas), machine learning concepts, book notes, optimal blackjack strategy, etc.

Anki should be useful to everyone reading this book — after all, you bought this book because you want to remember it — but it'll be particularly helpful for readers who don't have the opportunity to program in Python or Pandas regularly. When I learned how to code, I found it didn't necessarily "stick" until I was able to do it often — first as part of a sports related side project, then at my day job. I still think working on your own project is a great way to learn, but not everyone is able to do this immediately. Anki will help.

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/

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

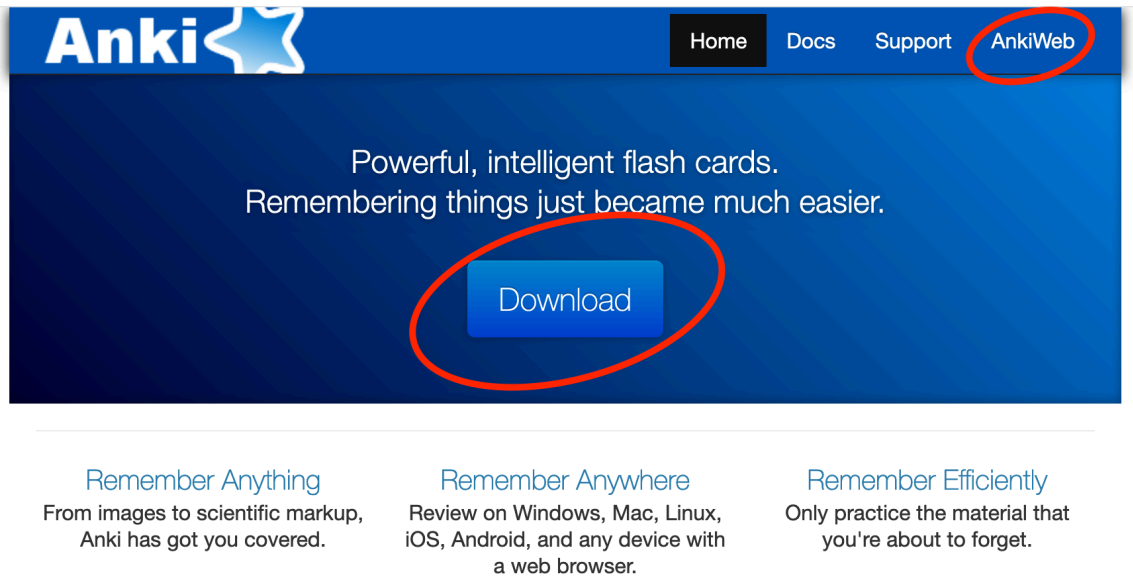

Figure 0.1: Anki Website

You should also make a free AnkiWeb account (in the upper right hand corner) and login with it on the desktop version so that you can save your progress and sync it with the mobile app.

Then install the mobile app. I use AnkiDroid, which is free and works well, but is only for Android.

The official IPhone version costs $25. It would be well worth it to me personally and goes towards supporting the creator of Anki, but if you don't want to pay for it you can either use the computer version or go to https://ankiweb.net on your phone's browser and review your flash cards there. I've also included text versions of the cards if you want to use another flashcard program.

Once you have the mobile app installed, go to settings and set it up to sync with your AnkiWeb account.

Using Anki with this Book

Anki has a ton of settings, which can be a bit overwhelming. You can ignore nearly all of them to start. By default, Anki makes one giant deck (called Default), which you can just add everything to. This is how I use it and it's worked well for me.

Once you've read and understand a section of the book and want to add the Anki cards for it, open up the desktop version of Anki, go to File -> Import… and then find the name of the apkg file (included with the book) you're importing.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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-half (and game)
b) team
c) worldcup round
d) team and win/loss
e) team-position

1.2

a) Wind speed, temperature.

b) Number of total goals scored

c) This model is at the game level.

d) This is subjective, but I'd say the biggest limitation is it includes no information about the teams involved (i.e. whether they're good and high or low scoring), just the weather.

1.3

a) manipulating data
b) analyzing data
c) manipulating data
d) loading data
e) collecting data

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

2. Python

2.1

a) `_throwaway_data`. Valid. Python programmers often start variables with _ if they're throw-away or temporary, short term variables.
b) `n_shots`. Valid.
c) `1st_half`. Not valid. Can't start with a number.
d) `shotsOnGoal`. Valid, though convention is to split words with _, not camelCase.
e) `wc_2018_champion`. Valid. Numbers OK as long as they're not in the first spot
f) `player position`. Not valid. No spaces
g) `@home_or_away`. Not valid. Only non alphanumeric character allowed is _
h) `'num_penalties'`. Not valid. A string (wrapped in quotes), not a variable name. Again, only non alphanumeric character allowed is _

2.2

```
In [1]:
match_minutes = 45
match_minutes = match_minutes + 45
match_minutes = match_minutes + 5

In [2]: match_minutes # 95
Out[2]: 95
```

2.3

```
In [3]:
def commentary(player, play):
    return f'{player} with the {play}!'
--

In [4]: commentary('Messi', 'goal')
Out[4]: 'Messi with the goal!'
```

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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

We can test it like:

```
In [5]: 'lionel messi'.islower()  # should return True
Out[5]: True

In [6]: 'Lionel Messi'.islower()  # should return False
Out[6]: False
```

2.5

```
In [7]:
def is_oconnell(player):
    return player.replace("'", '').lower() == 'jack oconnell'

In [8]: is_oconnell('lionel messi')
Out[8]: False

In [9]: is_oconnell("Jack O'Connell")
Out[9]: True

In [10]: is_oconnell("JACK OCONNELL")
Out[10]: True
```

2.6

```
In [11]:
def a_lot_of_goals(goals):
    if goals >= 4:
        return f'{goals} is a lot of goals!'
    else:
        return f'{goals} is not that many goals'

In [12]: a_lot_of_goals(3)
Out[12]: '3 is not that many goals'

In [13]: a_lot_of_goals(7)
Out[13]: '7 is a lot of goals!'
```

2.7

Here's what I came up with. The last two use list comprehensions.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [14]: roster = ['ruben dias', 'gabriel jesus', 'riyad mahrez']

In [15]: roster[0:2]
Out[15]: ['ruben dias', 'gabriel jesus']

In [16]: roster[:2]
Out[16]: ['ruben dias', 'gabriel jesus']

In [17]: roster[:-1]
Out[17]: ['ruben dias', 'gabriel jesus']

In [18]: [x for x in roster if x != 'riyad mahrez']
Out[18]: ['ruben dias', 'gabriel jesus']

In [19]: [x for x in roster if x in ['ruben dias', 'gabriel jesus']]
Out[19]: ['ruben dias', 'gabriel jesus']
```

2.8a

```
In [20]:
shot_info = {'shooter': 'Robert Lewandowski', 'foot': 'right', 'went_in':
    False}

In [21]: shot_info['shooter'] = 'Cristiano Ronaldo'

In [22]: shot_info
Out[22]: {'shooter': 'Cristiano Ronaldo', 'foot': 'right', 'went_in':
   False}
```

2.8b

```
In [23]:
def toggle_foot(info):
    if info['foot'] == 'right':
        info['foot'] = 'left'
    else:
        info['foot'] = 'right'
    return info

In [24]: shot_info
Out[24]: {'shooter': 'Cristiano Ronaldo', 'foot': 'right', 'went_in':
   False}

In [25]: toggle_foot(shot_info)
Out[25]: {'shooter': 'Cristiano Ronaldo', 'foot': 'left', 'went_in': False
   }
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

2.9

a) No. `'is_pk'` hasn't been defined.
b) No, `shooter` is a variable that hasn't been defined, the key is `'shooter'`.
c) Yes.

2.10a

```
In [26]: roster = ['ruben dias', 'gabriel jesus', 'riyad mahrez']

In [27]:
for x in roster:
  print(x.split(' ')[-1])
--
dias
jesus
mahrez
```

2.10b

```
In [28]: {player: len(player) for player in roster}
Out[28]: {'ruben dias': 10, 'gabriel jesus': 13, 'riyad mahrez': 12}
```

2.11a

```
In [29]:
roster_dict = {'CB': 'ruben dias',
               'CF': 'gabriel jesus',
               'RW': 'riyad mahrez',
               'LW': 'raheem sterling'}

In [30]: [pos for pos in roster_dict]
Out[30]: ['CB', 'CF', 'RW', 'LW']
```

2.11b

```
In [31]:
[player for _, player in roster_dict.items()
    if player.split(' ')[-1][0] in ['j', 'm']]
--
Out[31]: ['gabriel jesus', 'riyad mahrez']
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

2.12a

```
In [32]:
def mapper(my_list, my_function):
    return [my_function(x) for x in my_list]
```

2.12b

```
In [33]: match_minutes = [95, 92, 91, 91, 97, 95]

In [34]: mapper(match_minutes, lambda x: x - 90)
Out[34]: [5, 2, 1, 1, 7, 5]
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.0 Pandas Basics

3.0.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
dfm = pd.read_csv(path.join(DATA_DIR, 'matches.csv'))
```

3.0.2

```
In [3]: dfm10 = dfm.sort_values('date').head(10)
```

3.0.3

```
In [3]: dfm.sort_values('label', ascending=False, inplace=True)

In [4]: dfm.head()
Out[4]:
    match_id                          label  ...     away_team day
27   2057957  Uruguay - Saudi Arabia, 1 - 0  ...  Saudi Arabia   6
41   2057958        Uruguay - Russia, 3 - 0  ...        Russia  10
39   2058002      Uruguay - Portugal, 2 - 1  ...      Portugal  16
37   2058010        Uruguay - France, 0 - 2  ...        France  21
14   2057991       Tunisia - England, 1 - 2  ...       England   4
```

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 [5]: type(dfm.sort_values('label'))  # it's a DataFrame
Out[5]: pandas.core.frame.DataFrame
```

3.0.5a

```
In [6]:
match_simple = dfm[['date', 'home_team', 'away_team', 'home_score',
                    'away_score']]
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.0.5b

```
In [7]:
match_simple = match_simple[['home_team', 'away_team', 'date', 'home_score
    ',
                             'away_score']]
```

3.0.5c

```
In [8]: match_simple['match_id'] = dfm['match_id']
```

3.0.5d

```
In [9]: dfm.to_csv(path.join(DATA_DIR, 'match_simple.txt'), sep='|')
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.1 Columns

3.1.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
pm = pd.read_csv(path.join(DATA_DIR, 'player_match.csv'))
```

3.1.2

```
In [1]: pm['ob_touches'] = pm['throw'] + pm['corner']

In [2]: pm['ob_touches'].head()
Out[2]:
0     0
1    16
2     0
3     0
4     0
```

3.1.3

```
In [3]: pm['player_desc'] = pm['name'] + ' is the ' + pm['team'] + ' '
          + pm['pos']

In [4]: pm['player_desc'].head()
Out[4]:
0       D. Cheryshev is the Russia MID
1    Mário Fernandes is the Russia DEF
2        I. Akinfeev is the Russia GKP
3     S. Ignashevich is the Russia DEF
4         A. Dzagoev is the Russia MID
```

3.1.4

```
In [5]: pm['at_least_one_throwin'] = pm['throw'] > 0

In [6]: pm['at_least_one_throwin'].head()
Out[6]:
0    False
1     True
2    False
3    False
4    False
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.1.5

```
In [7]:
pm['len_last_name'] = (pm['name']
                       .apply(lambda x: len(x.split(' ')[-1])))
--

In [8]: pm['len_last_name'].head()
Out[8]:
0    10
1    15
2     9
3    12
4     8
Name: len_last_name, dtype: int64
```

3.1.6

```
In [9]: pm['match_id'] = pm['match_id'].astype(str)
```

3.1.7a

```
In [10]: pm.columns = [x.replace('_', ' ') for x in pm.columns]

In [11]: pm.head()
Out[11]:
              name    team  ...  at least one throwin  len last name
0     D. Cheryshev  Russia  ...                 False             10
1  Mário Fernandes  Russia  ...                  True             15
2      I. Akinfeev  Russia  ...                 False              9
3   S. Ignashevich  Russia  ...                 False             12
4       A. Dzagoev  Russia  ...                 False              8
```

3.1.7b

```
In [12]: pm.columns = [x.replace(' ', '_') for x in pm.columns]

In [13]: pm.head()
Out[13]:
              name    team  ...  at_least_one_throwin  len_last_name
0     D. Cheryshev  Russia  ...                 False             10
1  Mário Fernandes  Russia  ...                  True             15
2      I. Akinfeev  Russia  ...                 False              9
3   S. Ignashevich  Russia  ...                 False             12
4       A. Dzagoev  Russia  ...                 False              8
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.1.8a

```
In [14]: pm['air_duel_won_percentage'] = pm['air_duel_won']/pm['air_duel']

In [15]: pm['air_duel_won_percentage'].head()
Out[15]:
0    1.0
1    0.7
2    NaN
3    0.9
4    1.0
```

3.1.8b

`'air_duel_won_percentage'` is air duels won divided by total air duels. Since you can't divide by 0, `air_duel_won_percentage` is missing whenever a player had 0 rebounds.

To replace all the missing values with `-99`:

```
In [16]: pm['air_duel_won_percentage'].fillna(-99, inplace=True)

In [17]: pm['air_duel_won_percentage'].head()
Out[17]:
0     1.0
1     0.7
2   -99.0
3     0.9
4     1.0
```

3.1.9

```
In [18]: pm.drop('air_duel_won_percentage', axis=1, inplace=True)

In [19]: pm.head()
Out[19]:
              name    team  ...  at_least_one_throwin  len_last_name
0     D. Cheryshev  Russia  ...                 False             10
1  Mário Fernandes  Russia  ...                  True             15
2      I. Akinfeev  Russia  ...                 False              9
3   S. Ignashevich  Russia  ...                 False             12
4       A. Dzagoev  Russia  ...                 False              8
```

If you forget the `axis=1` Pandas will try to drop the *row* with the index value `'air_duel_won_percentage'`. Since that doesn't exist, it'll throw an error.

Without the `inplace=True`, Pandas just returns a new copy of `pm` without the `'air_duel_won_percentage'` column. Nothing happens to the original `pm`, though we could reassign it if we wanted like this:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
pm = pm.drop('air_duel_won_percentage', axis=1)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.2 Built-in Functions

3.2.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
pm = pd.read_csv(path.join(DATA_DIR, 'player_match.csv'))
```

3.2.2

```
In [1]: (pm['named_pass1'] = pm['clearance'] + pm['cross'] +
         pm['assist'] + pm['keypass'])

In [2]: pm['named_pass2'] = (
          pm[['clearance', 'cross', 'assist', 'keypass']].sum(axis=1))

In [3]: (pm['named_pass1'] == pm['named_pass2']).all()
Out[3]: True
```

3.2.3a

```
In [4]: pm[['shot', 'assist', 'pass']].mean()
Out[4]:
shot       0.817475
assist     0.050269
pass      31.599641
```

3.2.3b

```
In [5]: ((pm['goal'] >= 1) & (pm['assist'] >= 1)).sum()  # 10
Out[5]: 10
```

3.2.3c

```
In [6]: ((pm['goal'] >= 1) & (pm['assist'] >= 1)).sum()/(pm.shape[0])
Out[6]: 0.005984440454817474
```

3.2.3d

```
In [7]: pm['own_goal'].sum()
Out[7]: 10
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.2.3e

```
In [8]: pm['pos'].value_counts()
Out[8]:
MID    637
DEF    532
FWD    384
GKP    118
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.3 Filtering

3.3.1

```
import pandas as pd
from os import path

DATA_DIR = './data'
dfp = pd.read_csv(path.join(DATA_DIR, 'players.csv'))
```

3.3.2a

```
In [1]:
dfp_bra1 = dfp.loc[dfp['team'] == 'Brazil',
                ['player_name', 'pos', 'foot', 'weight', 'height']]

In [2]: dfp_bra1.head()
Out[2]:
           player_name  pos   foot  weight  height
49             Marcelo  DEF   left      80     174
65          Filipe Luis  DEF   left      73     182
86   Philippe Coutinho  FWD  right      68     171
106             Danilo  DEF  right      78     184
141            Ederson  GKP   left      89     187
```

3.3.2b

```
In [3]:
dfp_bra2 = dfp.query("team == 'Brazil'")[['player_name', 'pos', 'foot',
    'weight', 'height']]
```

3.3.3

```
In [5]:
dfp_no_bra = dfp.loc[dfp['team'] != 'Brazil', ['team', 'player_name', 'pos
    ',
    'foot', 'weight', 'height']]

In [6]: dfp_no_bra.head()
Out[6]:
      team      player_name  pos   foot  weight  height
0  Senegal       A. N'Diaye  MID  right      82     187
1  Belgium  T. Alderweireld  DEF  right      91     187
2  Belgium    J. Vertonghen  DEF   left      88     189
3  Denmark       C. Eriksen  MID  right      76     180
4  Iceland  J. Guðmundsson  MID   left      77     186
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.3.4a

Yes.

```
In [7]: dfp['bday'] = dfp['birth_date'].astype(str).str[-4:]

In [8]: dfp['bday'].duplicated().any()
Out[8]: True
```

3.3.4b

```
In [9]: dups = dfp[['bday']].duplicated(keep=False)

In [10]: dfp_dups = dfp.loc[dups]

In [11]: dfp_no_dups = dfp.loc[~dups]
```

3.3.5

```
In [12]:
import numpy as np

dfp['height_description'] = np.nan
dfp.loc[dfp['height'] > 190, 'height_description'] = 'tall'
dfp.loc[dfp['height'] < 170, 'height_description'] = 'short'
dfp[['height', 'height_description']].sample(5)

Out[12]:
     height height_description
95      187                NaN
126     178                NaN
650     173                NaN
117     191               tall
58      171                NaN
```

3.3.6a

```
In [13]: dfp_no_desc1 = dfp.loc[dfp['height_description'].isnull()]
```

3.3.6b

```
In [14]: dfp_no_desc2 = dfp.query("height_description.isnull()")
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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 which shots Messi scored on (and how much time was left in the match, etc) to just knowing he scored, say, 2 goals total, that's a loss of information.

3.4.2a

```
import pandas as pd
from os import path

DATA_DIR = './data'
dfpm = pd.read_csv(path.join(DATA_DIR, 'player_match.csv'))
```

3.4.2b

```
In [1]: dfpm.groupby('player_id')['shot', 'goal'].mean()

Out[1]:
                shot      goal
player_id
12          0.000000  0.000000
36          0.833333  0.000000
48          0.333333  0.166667
54          2.666667  0.000000
93          1.500000  0.000000
...              ...       ...
437417      0.333333  0.000000
447821      1.000000  1.000000
448079      1.000000  0.000000
448210      0.000000  0.000000
552555      0.500000  0.000000
```

3.4.2c

Just change `sum` of `'yards_gained'` to `mean`:

```
In [2]:
player_ave = dfpm.groupby('player_id')['shot', 'goal'].mean()
(player_ave['shot'] >= 4).mean()  # 1.05%

Out[2]: 0.01054481546572935
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.4.3a

```
In [3]:
dftm = dfpm.groupby(['match_id', 'team']).agg(
    total_goal = ('goal', 'sum'),
    total_pass = ('pass', 'sum'),
    total_shot = ('shot', 'sum'),
    nplayed = ('player_id', 'count'))

In [4]: dftm.head()
Out[4]:
                        total_goal  total_pass  total_shot  nplayed
match_id team
2057954  Russia                  5         311          11       14
         Saudi Arabia            0         516           7       14
2057955  Egypt                   0         421           7       14
         Uruguay                 1         579          11       14
2057956  Egypt                   1         420          12       14
```

3.4.3b

```
In [5]: dftm.reset_index(inplace=True)
```

3.4.3c

```
In [6]: dftm['no_goals'] = dftm['total_goal'] == 0

In [7]: dftm.groupby('no_goals')['total_pass', 'total_shot'].mean()
Out[7]:
          total_pass  total_shot
no_goals
False     441.436782   11.850575
True      411.371429    9.571429
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.4.3d

```
In [8]: dftm.groupby('team')['match_id'].count()
Out[8]:
team
Argentina         4
Belgium           7
Brazil            5
Colombia          4
Costa Rica        3
Croatia           7
Denmark           3
Egypt             3
England           7
France            6
Germany           3
Iceland           3
Iran              3
Japan             4
Korea Republic    3
Mexico            4
Morocco           3
Nigeria           3
Panama            3
Peru              2
Poland            3
Portugal          4
Russia            5
Saudi Arabia      3
Senegal           3
Serbia            3
Spain             4
Sweden            5
Switzerland       4
Tunisia           3
Uruguay           5
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [9]: dfpm.groupby('team')['match_id'].sum()
Out[9]:
team
Argentina         109072999
Belgium           193452310
Brazil            144059398
Colombia          115248082
Costa Rica         86435188
Croatia           205799637
Denmark            84377256
Egypt              86434180
England           203742464
France            170813734
Germany            86435412
Iceland            86434936
Iran               86434432
Japan             113190035
Korea Republic     86435440
Mexico             94667602
Morocco            82318506
Nigeria            86434936
Panama             82319704
Peru               53507155
Poland             69971930
Portugal          115246432
Russia            148174392
Saudi Arabia       84376221
Senegal            86435944
Serbia             82319225
Spain             117304492
Sweden            144059734
Switzerland       111131311
Tunisia            78203723
Uruguay           144058348
```

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.

3.4.4

Stacking is when you change the granularity in your data, but shift information from rows to columns (or vice versa) so it doesn't result in any loss on information.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.5 Combining DataFrames

3.5.1a

```
import pandas as pd
from os import path

DATA_DIR = './data'
df_name = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'name.csv')
    )
df_shot = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'shot.csv')
    )
df_pass = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'pass.csv')
    )
df_ob = pd.read_csv(path.join(DATA_DIR, 'problems/combine1', 'ob.csv'))
```

3.5.1b

```
In [1]: df_comb1 = pd.merge(df_name, df_shot, how='left')

In [2]: df_comb1 = pd.merge(df_comb1, df_pass, how='left')

In [3]: df_comb1 = pd.merge(df_comb1, df_ob, how='left')

In [4]: df_comb1 = df_comb1.fillna(0)
```

3.5.1c

```
In [5]:
df_comb2 = pd.concat([df_name.set_index(['player_id', 'match_id']),
                      df_shot.set_index(['player_id', 'match_id']),
                      df_pass.set_index(['player_id', 'match_id']),
                      df_ob.set_index(['player_id', 'match_id'])], join='
                         outer',
                     axis=1)

In [6]: 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.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

3.5.2a

```
import pandas as pd
from os import path

DATA_DIR = './data'
df_d = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'def.csv'))
df_f = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'fwd.csv'))
df_m = pd.read_csv(path.join(DATA_DIR, 'problems/combine2', 'mid.csv'))
```

3.5.2b

```
In [7]: df = pd.concat([df_d, df_f, df_m], 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 [8]:
for group in ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']:
    (dft
        .query(f"grouping == '{group}'")
        .to_csv(path.join(DATA_DIR, f'dft_{group}.csv'), index=False))
```

3.5.3c

```
In [9]:
df = pd.concat([pd.read_csv(path.join(DATA_DIR, f'dft_{group}.csv'))
    for group in ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']], ignore_index=
        True)
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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

```sql
SELECT
    date, name AS player, team.team, goal, shot, pass
FROM
    player_match, team
WHERE
    team.team = player_match.team AND
    team.grouping = 'C'
```

4.2

```sql
SELECT
    date, name AS player, t.team, goal, shot, pass, height, weight
FROM
    player_match AS pm,
    team AS t,
    player AS p
WHERE
    t.team = pm.team AND
    t.grouping = 'C' AND
    pm.player_id = p.player_id
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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(dftm)
     .map(sns.kdeplot, 'pass', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Passes')
```

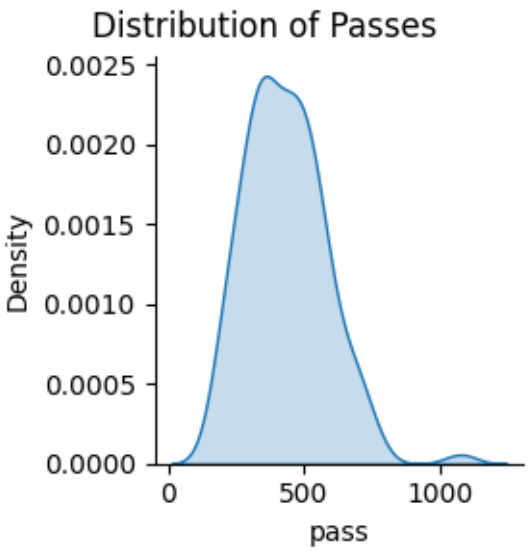

Figure 0.1: Solution 6-1a

6.1b

```
g = (sns.FacetGrid(dftm, hue='win')
    .map(sns.kdeplot, 'pass', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Passes by Win/Loss B')
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

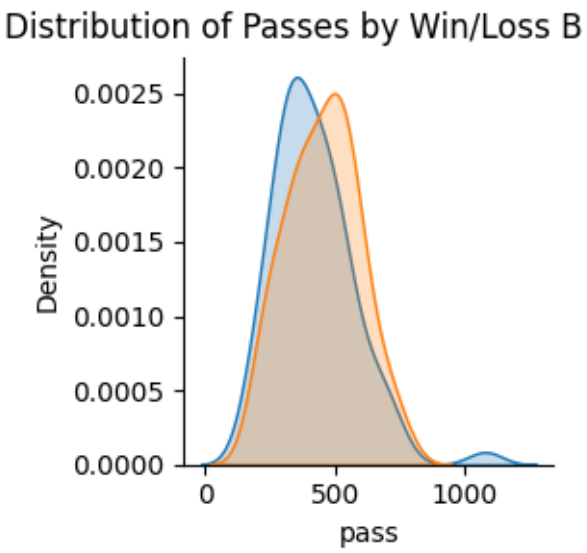

Figure 0.2: Solution 6-1b

6.1c

```
g = (sns.FacetGrid(dftm, col='win')
    .map(sns.kdeplot, 'pass', shade=True))
g.fig.subplots_adjust(top=0.8)
g.fig.suptitle('Distribution of Passes by Win/Loss C')
```

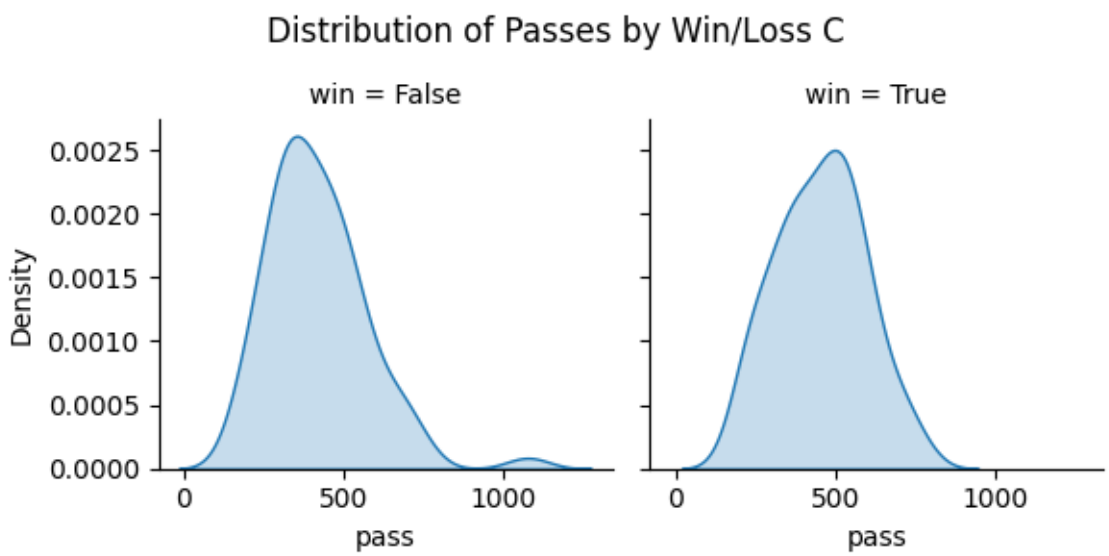

Figure 0.3: Solution 6-1c

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

6.1d

```
g = (sns.FacetGrid(dftm, col='win', hue='win')
    .map(sns.kdeplot, 'pass', shade=True))
g.fig.subplots_adjust(top=0.8)
g.fig.suptitle('Distribution of Passes by Win/Loss D')
```

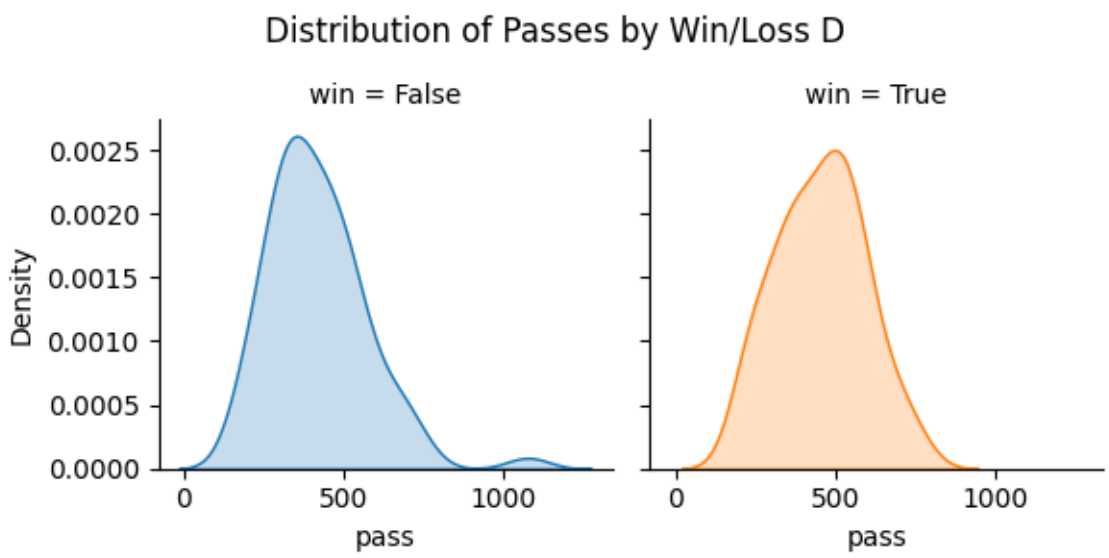

Figure 0.4: Solution 6-1d

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

6.1e

```
g = (sns.FacetGrid(dftm, col='team', col_wrap=6)
    .map(sns.kdeplot, 'pass', shade=True))
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Distribution of Passes by Team')
```

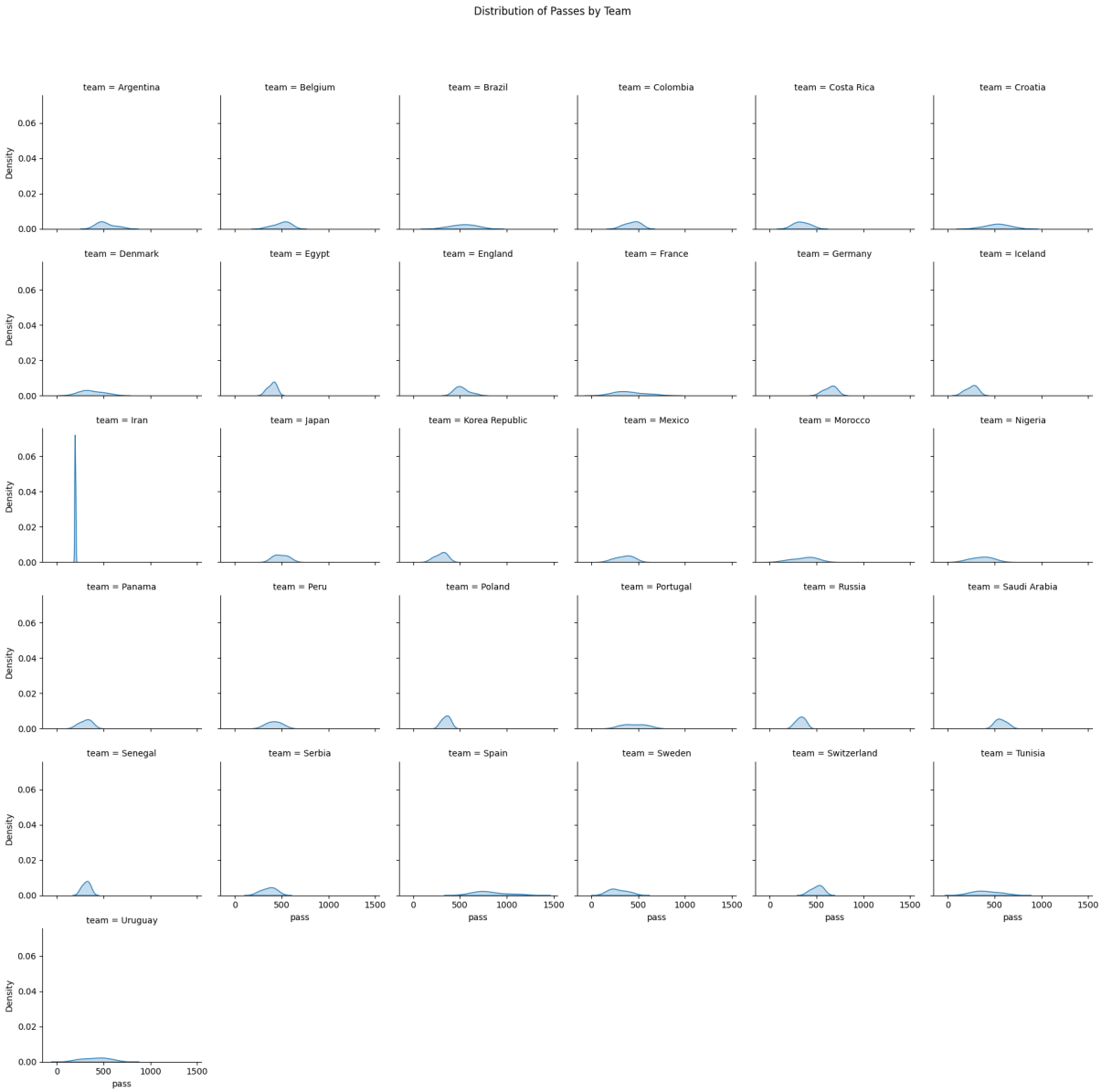

Figure 0.5: Solution 6-1e

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

6.2a

```
g = sns.relplot(x='pass', y='pass_opp', data=dftm)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Passes vs Opponent Passes')
```

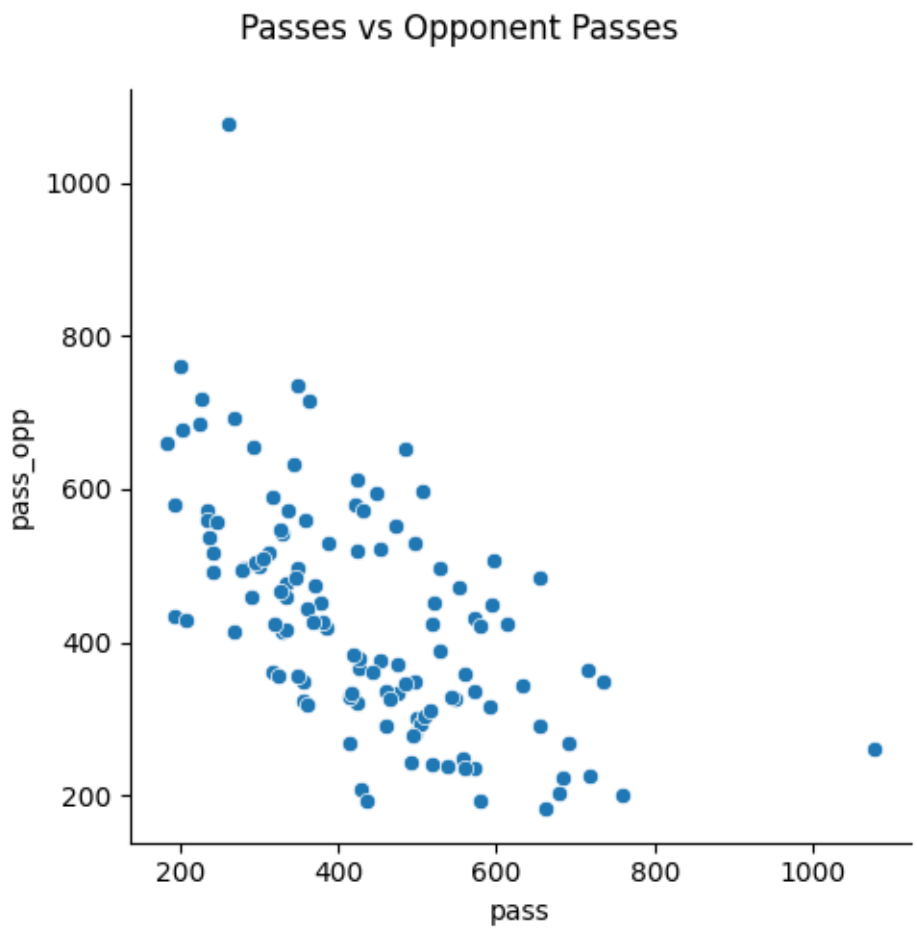

Figure 0.6: Solution 6-2a

6.2b

```
In [1]: dftm[['pass', 'pass_opp']].corr()
Out[1]:
              pass  pass_opp
pass      1.000000 -0.595039
pass_opp -0.595039  1.000000
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

7. Modeling

7.1a

To apply `prob_of_goal` to our yardage data.

```
In [1]:
def prob_of_goal(meters):
    b0, b1, b2 = results.params
    return (b0 + b1*meters + b2*(meters**2))

In [2]: df['made_hat_alt'] = df['dist'].apply(prob_of_make)
```

The two should be the same. With `'goal_hat_alt'` we're just doing manually what `results.predict(df)` is doing behind the scenes. Let's look at them:

```
In [3]: df[['goal_hat', 'goal_hat_alt']].head()
Out[3]:
   goal_hat  goal_hat_alt
0  0.130441      0.130441
1  0.092865      0.092865
2  0.088355      0.088355
3  0.182858      0.182858
4  0.098753      0.098753
```

To check whether they're the same:

```
In [5]: (df['goal_hat'] == df['goal_hat_alt']).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['goal_hat'] - df['goal_hat_alt']) < .00000001).all()
Out[7]: True
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

7.1b

```
In [8]:
model_b = smf.ols(formula='goal ~ dist_m + dist_m_sq + C(period)',
                  data=df)
results_b = model_b.fit()
results_b.summary2()

Out[8]:
"""
                 Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.070
Dependent Variable: goal             AIC:                367.5479
Date:               2022-07-21 16:46 BIC:                398.8658
No. Observations:   1366             Log-Likelihood:     -177.77
Df Model:           5                F-statistic:        21.41
Df Residuals:       1360             Prob (F-statistic): 1.13e-20
R-squared:          0.073            Scale:              0.076292
------------------------------------------------------------------
                   Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
------------------------------------------------------------------
Intercept          0.2792   0.0245 11.4147 0.0000  0.2312  0.3272
C(period)[T.2H]    0.0348   0.0153  2.2762 0.0230  0.0048  0.0648
C(period)[T.E1]   -0.0080   0.0661 -0.1207 0.9039 -0.1376  0.1217
C(period)[T.E2]    0.0096   0.0575  0.1664 0.8679 -0.1033  0.1224
dist_m            -0.0149   0.0017 -8.5919 0.0000 -0.0183 -0.0115
dist_m_sq          0.0001   0.0000  5.1737 0.0000  0.0001  0.0002
------------------------------------------------------------------
Omnibus:             696.024      Durbin-Watson:         1.999
Prob(Omnibus):       0.000        Jarque-Bera (JB):      3054.676
Skew:                2.545        Prob(JB):              0.000
Kurtosis:            8.268        Condition No.:         6142
==================================================================
* The condition number is large (6e+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(period)[T.2H]` is 0.0348, which is statistically significant. So a shot is more likely to go in the second the second half.

7.1c

There just isn't as much shot data from the extra periods (~40 shots total vs ~1300 in periods 1-2). Every game has 2 periods for sure; most games don't have extra periods. So with a lot less data things can be much more noisy/random and it's hard to get a clear signal of what's going on.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

It'd be like flipping a coin 4 times and trying to make a judgement about whether it's fair or not.

```
In [9]: df['is_2h'] = df['period'] == '2H'

In [10]: df['is_e1'] = df['period'] == 'E1'

In [11]: df['is_e2'] = df['period'] == 'E2'

In [12]:
model_d = smf.ols(formula='goal ~ dist_m + dist_m_sq + is_2h + is_e1 +
    is_e2',
                  data=df)
results_d = model_d.fit()
results_d.summary2()  # yes

Out[12]:
"""
                  Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.070
Dependent Variable: goal             AIC:                367.5479
Date:               2022-07-21 16:49 BIC:                398.8658
No. Observations:   1366             Log-Likelihood:     -177.77
Df Model:           5                F-statistic:        21.41
Df Residuals:       1360             Prob (F-statistic): 1.13e-20
R-squared:          0.073            Scale:              0.076292
------------------------------------------------------------------
                    Coef.  Std.Err.    t    P>|t|   [0.025  0.975]
------------------------------------------------------------------
Intercept           0.2792   0.0245 11.4147 0.0000  0.2312  0.3272
is_2h[T.True]       0.0348   0.0153  2.2762 0.0230  0.0048  0.0648
is_e1[T.True]      -0.0080   0.0661 -0.1207 0.9039 -0.1376  0.1217
is_e2[T.True]       0.0096   0.0575  0.1664 0.8679 -0.1033  0.1224
dist_m             -0.0149   0.0017 -8.5919 0.0000 -0.0183 -0.0115
dist_m_sq           0.0001   0.0000  5.1737 0.0000  0.0001  0.0002
------------------------------------------------------------------
Omnibus:              696.024      Durbin-Watson:         1.999
Prob(Omnibus):        0.000        Jarque-Bera (JB):      3054.676
Skew:                 2.545        Prob(JB):              0.000
Kurtosis:             8.268        Condition No.:         6142
==================================================================
* The condition number is large (6e+03). This might indicate
strong multicollinearity or other numerical problems.
"""
```

Yes, the coefficients are the same.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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)
    ])
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

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
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

7.3a

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

In [2]: dftm['win'] = dftm['win'].astype(int)

In [3]: dftm['npass'] = dftm['pass']

In [4]:
model_a = smf.logit(formula=
                    """
                    win ~ shot + npass
                    """, data=dftm)
results_a = model_a.fit()
results_a.summary2()
--
Optimization terminated successfully.
         Current function value: 0.659106
         Iterations 4
Out[4]:
"""
                        Results: Logit
================================================================
Model:              Logit            Pseudo R-squared: 0.030
Dependent Variable: win              AIC:              166.8218
Date:               2022-07-21 16:50 BIC:              175.2339
No. Observations:   122              Log-Likelihood:   -80.411
Df Model:           2                LL-Null:          -82.917
Df Residuals:       119              LLR p-value:      0.081572
Converged:          1.0000           Scale:            1.0000
No. Iterations:     4.0000
-----------------------------------------------------------------
             Coef.   Std.Err.     z     P>|z|    [0.025   0.975]
-----------------------------------------------------------------
Intercept   -1.2736    0.6023  -2.1144  0.0345  -2.4541  -0.0930
shot         0.0835    0.0464   1.7974  0.0723  -0.0075   0.1745
npass       -0.0000    0.0015  -0.0024  0.9981  -0.0030   0.0030
================================================================
"""
```

7.3b

Adding goals

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [5]:
model_b = smf.logit(formula=
                    """
                    win ~ shot + npass + goal
                    """, data=dftm)
results_b = model_b.fit()
results_b.summary2()
--
Optimization terminated successfully.
         Current function value: 0.530662
         Iterations 6
Out[5]:
"""
                         Results: Logit
==================================================================
Model:              Logit            Pseudo R-squared: 0.219
Dependent Variable: win              AIC:              137.4814
Date:               2022-07-21 16:52 BIC:              148.6975
No. Observations:   122              Log-Likelihood:   -64.741
Df Model:           3                LL-Null:          -82.917
Df Residuals:       118              LLR p-value:      6.3060e-08
Converged:          1.0000           Scale:            1.0000
No. Iterations:     6.0000
-------------------------------------------------------------------
               Coef.   Std.Err.     z     P>|z|    [0.025   0.975]
-------------------------------------------------------------------
Intercept     -1.9826    0.7051  -2.8117  0.0049  -3.3646  -0.6006
shot           0.0678    0.0535   1.2669  0.2052  -0.0371   0.1728
npass         -0.0012    0.0018  -0.6674  0.5045  -0.0047   0.0023
goal           0.9800    0.2203   4.4481  0.0000   0.5482   1.4118
==================================================================
```

More shots contribute to winning via more goals, so it makes sense that — *controlling for number of goals* — more shots on their own don't nec do anything to help a team win (e.g. they're no longer significant). Of course, the other component of winning is goals *allowed*, so — to the extent more shots on goal mean the other team isn't shooting and scoring on you — they could still be helpful.

7.4a

To run the model:

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, cross_val_score

dfpm = pd.read_csv(path.join(DATA_DIR, 'player_match.csv'))

xvars = ['min', 'shot', 'goal', 'goal_allowed', 'assist', 'pass',
    'pass_accurate', 'tackle', 'accel', 'counter', 'opportunity', 'keypass
        ',
    'own_goal', 'interception', 'smart', 'clearance', 'cross', 'air_duel',
    'air_duel_won', 'throw', 'corner', 'started']

yvar = 'pos'

dfpm[xvars] = dfpm[xvars].fillna(-99)
```

7.4b

```
In [10]: model = RandomForestClassifier(n_estimators=100)

In [11]: scores = cross_val_score(model, dfpm[xvars], dfpm[yvar], cv=10)

In [12]: scores.mean()
Out[12]: 0.7378243512974051
```

7.4c

Filling missing variables with a negative number in a random forest works because the algorithm picks a split point on each variable

So if we have a bunch of normal values for (say) shots — 0, 1, 0, 2 etc, then we have some that are missing with values -99, the algorithm will just pick a split point somewhere between -99 and 0, where everything below will go one way, above another.

This works as a way for the CART algorithm to clearly identify any missing values.

7.4d

If we were guessing at random we'd expect to get 1/30 right.

Prepared exclusively for mbellido@knox.edu Transaction: 0141058347

```
In [13]: model.fit(dfpm[xvars], dfpm[yvar])
Out[13]: RandomForestClassifier()

In [14]: Series(model.feature_importances_, xvars).sort_values(ascending=
    False)
Out[14]:
pass              0.110580
pass_accurate     0.107167
throw             0.098936
min               0.098713
clearance         0.080519
air_duel          0.065890
interception      0.064152
goal_allowed      0.047155
counter           0.046195
air_duel_won      0.041892
cross             0.039638
shot              0.039396
opportunity       0.034519
accel             0.026144
started           0.023500
tackle            0.022568
corner            0.022008
keypass           0.016721
goal              0.008358
assist            0.004805
own_goal          0.001144
smart             0.000000
dtype: float64
```


Prepared exclusively for mbellido@knox.edu Transaction: 0141058347