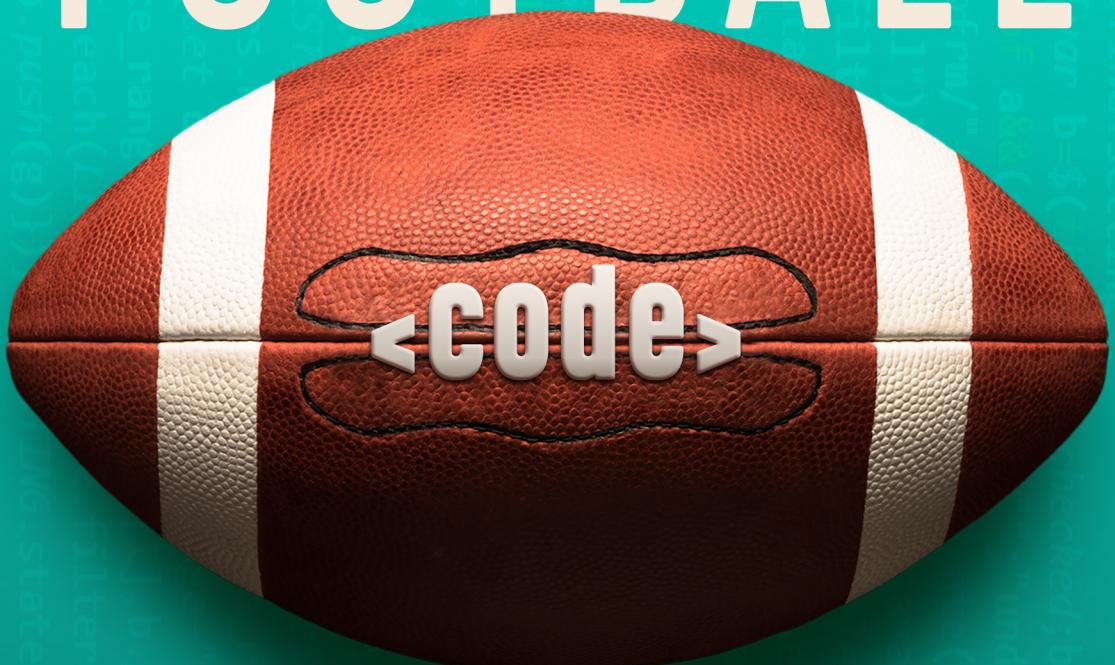


NATHAN BRAUN

2025 DEVELOPER KIT FANTASY FOOTBALL



A project based guide for analyzing your fantasy football team using tools YOU build.

2025 Fantasy Football Developer Kit

v0.19.1

Copyright Notice

Copyright © 2019 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 Fantasy Football digital product, whether modified or not, to any third party.

Contents

Prerequisites: Tooling	1
1. Get the Files Included with this Book	1
2. Set Up your config.ini File	3
Developer Kit License Key	3
1. Introduction	5
Learn to Code with Fantasy Football	5
Building your own tools with this kit	5
The Projects	6
Automatic League Import	6
Who Do I Start Calculator	6
League Analyzer	6
Best Ball Projector	6
Technical Prerequisites: Python and Pandas	6
High Level Strategies for Building Real Programs	7
Gall's Law	7
Get Quick Feedback	7
Use Functions	8
How This Book is Organized	8
2. Monte Carlo Analysis	9
Option 1 for working with distributions: math	9
Option 2 for working with distributions: simulations	10
3. utilities.py	12
How to Read This Chapter	12
Spyder	13
Prerequisites	14
Setup, Authorization, Fantasy Math API, Accessing the Simulations	14
__name == '__main__'	14
Authorization Workflow	15

GraphQL	16
Included Helper Functions	17
generate_token	17
get_players	17
master_player_lookup	17
get_sims	18
4. Introduction to the Data	19
How to Read This Chapter	19
Reminder About Your Working Directory	19
Simulation Data	19
Querying sims	21
Working with simulations	22
Will Herbert outscore Mahomes?	23
Other Monte Carlo Applications	25
Actual scores vs projections	26
Percentiles as luck	30
Correlations	32
Conditional Probabilities	34
Who should I start Monday night?	35
Playing “offense” or “defense” as the underdog or favorite	36
Simulating your entire lineup	37
What Goes into the Simulations	37
5. Quickstart: Analyze Your Team and League	39
League Website Specific Setup	39
Fleaflicker Setup Quickstart	40
Sleeper Setup Quickstart	40
ESPN Setup Quickstart	43
Yahoo Setup Quickstart	44
Setup Continued (All Sites)	49
League Analyzer	50
Who Do I Start Analyzer	52
Conclusion	53
6. Project #1: Who Do I Start	54
WDIS API	54
Projects Connect via APIs	55
Building the WDIS Project	55

Parameters	56
Coding Up a Who Do I Start Calculator	60
Beyond WDIS	66
Plotting	79
WDIS Wrap Up	84
7. Project #2: League Integration	86
Working with Public APIs — General Process	86
1. Authentication	86
2. Finding an endpoint	86
3. Visit endpoint in browser	87
4. Get what you need in Python	90
5. Clean things up	90
Common Outputs	90
1. Team Data	90
2 and 3. Matchup and Team Schedule Data	91
4. Roster Data	91
ESPN Integration	93
Authentication and Setup	93
Connecting to ESPN in Python	93
ESPN Endpoints	95
Roster Data	96
Team Data	115
Schedule Info	118
Wrap Up	121
Fleaflicker Integration	123
Roster Data	123
Team Data	136
Schedule Info	140
Wrap Up	142
Sleeper Integration	143
Roster Data	143
Team Data	160
Schedule Info	162
Wrap Up	168
Yahoo Integration	170
Authentication and Setup	170
Yahoo Walkthrough	173

Yahoo Endpoints	176
Roster Data	177
Team Data	192
Schedule Info	198
Wrap Up	207
Saving League Data to a Database	208
Getting the Data	208
Writing it to a Database	209
Other League Data	211
Wrap Up - League Configuration	211
Auto WDIS - Integrating WDIS with your League	213
Writing to a File	225
Writing WDIS Output to a File	226
Wrap Up	230
8. Project #3: League Analyzer	231
Prerequisites	231
Input Data	231
Analyzing Matchups	233
Analyzing Teams	243
High and Low	245
Aside - walking through totals_by_team	245
Back to analyzing teams	247
Aside: high/low scores and how more data → less variance	249
Back to the team analysis	250
Writing to a File	251
Plots	253
9. Project #4: Best Ball Projector	259
Best Ball Leagues	259
Walkthrough	259
Using the simulations to find the max score	263
Presentation and formatting	275
Sleeper Best Ball Leagues	280
Appendix A: Installing Python	281
Python	281
Editor	283
Console (REPL)	283

Using Spyder	285
Appendix B: ini files	286
Appendix C: Probability + Fantasy Football	288
Distributions: Smoothed Out X's	290
Appendix D: Technical Review	293
Comprehensions	294
f-strings	294
Pandas Functions and the axis argument	295
stack/unstack	297
Review	300
Appendix E: Fantasy Math Web Access	301
License Key and Email	301
Leagues	302
Analyzing a Matchup	304
Results	305
Who Do I Start	307

Prerequisites: Tooling

Before we start we have to do two things:

1. Get the Files Included with this Book

The annual fantasy football developer kit includes three things:

1. This PDF guide.
2. The code — both the step by step versions we go through in this guide, as well as the polished versions — for all the projects.
3. A license key that lets you access the Fantasy Math simulations player projections via API.

If you're reading this, we're good on (1). We can get (2) at:

<https://github.com/nathanbraun/fantasy-developer-kit/releases>

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 `fantasy-developer-kit-vX.X.X.zip`, where X.X.X is the latest version number (v0.0.1 in the screenshot above).

2025 Fantasy Football Developer Kit

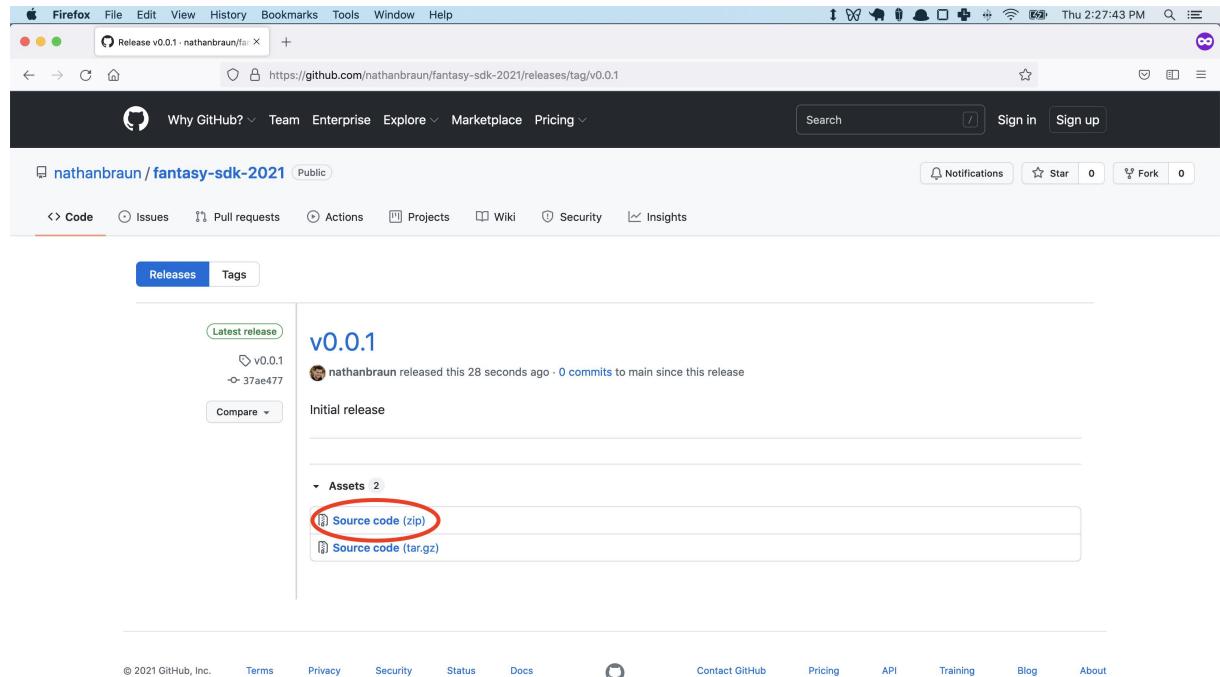


Figure 0.1: Developer Kit Files on Github

When you unzip these (note in the book I've dropped the version number and renamed the directory just `fantasy-developer-kit`, which you can do too) make note of where you put them.

For example, on my mac, I have them in my home directory:

```
/Users/nathanbraun/fantasy-developer-kit
```

Sometimes in this guide I'll refer to files starting with a period, like this:

```
. /projects/wdis/wdis_working.py
```

The period means it's relative to your `fantasy-developer-kit` directory. So the full location of this bath would be:

```
/Users/nathanbraun/fantasy-developer-kit/projects/wdis/wdis_working.py
```

2. Set Up your config.ini File

After you've got the files, make a copy of `config_example.ini` and rename it to `config.ini`. In that file, copy and paste your developer kit license key from SendOwl into the `LICENSE_KEY` spot (no quotes).

Note: **the file needs to be named exactly config.ini**. Though the file is just text, it has a `.ini` file extension.

Some computers hide file extensions by default. If you're going to be a coder, you'll want to see what they are, so I'd recommend changing your settings to show them. [Here's how to do it on a Mac](#) and [here's how to do it on Windows](#).

Developer Kit License Key

You can find your license key on the same page where you downloaded this guide:

File Download

 **2022 Fantasy Football Developer Kit**
License Key: ADB9-FC23 [REDACTED]
3 downloads remaining

 **Learn to Code with Fantasy Football (Book)**
3 downloads remaining

Download All

Powered by **SendOwl** 

Figure 0.2: Sendowl License Key Screenshot

When you bought the book you should have received a link to the download page in your email. If you can't find it, you can sign in here to find it again:

https://transactions.sendowl.com/customer_accounts/197959/login

This link will bring you to a login screen. If it's the first time you've visited it you'll have to enter your email then click 'Forgot password'.

When you have the license key, paste into `LICENSE_KEY` in the `config.ini` file you just made. Then you'll be all set. We'll pick these files back up in a bit.

1. Introduction

Learn to Code with Fantasy Football

Learn to Code with Fantasy Football (LTCWFF) teaches the basics and common themes (e.g. the 5 things you can do with DataFrames) that come up over and over again in Python, Pandas and Data Science.

Once you've learned coding fundamentals, the next steps — fitting the pieces together and putting together your own projects to do “real” work — are different skills.

This developer kit is a follow up to LTCWFF, but it's not “things you can do with DataFrames number 6-10”. Believe it or not, if you've worked through it you already *have* the technical skills to build your own projects.

Building your own tools with this kit

The skills for moving beyond the basics are less technical and more about mindset and general strategies. They take practice and the ability to stay motivated, which are two big benefits to working through projects to analyze your own Fantasy Football team.

In this kit, we'll walk through building four projects that use the **Fantasy Math simulation GraphQL API**.

The only way to build any complicated analysis is by starting simple and working our way up, so these walk-throughs include all the intermediate versions on our way up to the final product. You'll be able to apply all of these projects to your own fantasy teams and leagues immediately. Not only will that hopefully be more interesting, it should help you do better in fantasy too.

If you don't want to wait until you've walked through everything yourself, that's fine. The book includes final versions of each project you can use as a template as well as web-access to the Fantasy Math start-sit model.

Let's look at what we'll build.

The Projects

Automatic League Import

We'll walk through writing some code that can automatically connect to our league website — ESPN, Yahoo, Fleaflicker and Sleeper — so we can pull down rosters and automatically analyze our league and matchups. We'll set up our own SQL database to keep track of everything.

Who Do I Start Calculator

Next we'll build a tool that takes in your lineup, your opponents lineup, a list of guys you're thinking about starting, and returns the probability of winning with each one.

This is the project that will be the most useful in helping you do better in fantasy. It should add a couple of percentage points to your probability of winning every week.

League Analyzer

After that we'll build a tool to analyze our league, getting projections, over-unders and betting lines for each matchup, as well as team-specific stats like probability everyone scores the high or the low.

This is probably the most fun project, and hopefully something your league will enjoy (if you choose to share it with them — I wouldn't blame you for keeping this type of intel to yourself).

Best Ball Projector

Finally, we'll code up a tool that takes different best ball lineups and (accurately) projects total scores and utilization percentages.

This is less useful week to week (since you can't change your best ball team once you draft it), but it's a useful teaching tool + can help you analyze the tradeoffs between positions (e.g. a 3rd QB vs a 7th WR).

This project is a great Pandas refresher.

Technical Prerequisites: Python and Pandas

These projects assume you have familiarity with Python, Pandas and the plotting library seaborn.

If you're unfamiliar with any of them, read chapters two, three and six in LTCWFF. For details on the easiest way to install Python and how I recommend setting things up, see Appendix A.

It's not a substitute for those sections of the book, but I've also included a technical appendix (Appendix D) that reviews some of the Pandas and Python concepts that come up more often in these projects because of the type of data we're working with.

Topics included:

- List and Dictionary Comprehensions
- f-strings
- Pandas functions and the axis argument
- stack/unstack in Pandas

High Level Strategies for Building Real Programs

Gall's Law

"A complex system that works is invariably found to have evolved from a simple system that worked." - John Gall

If you go through all the build-from-scratch versions of these projects one concept that you'll notice again and again 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 these projects and think "there's no way I could ever do that." But if I just sat down and tried to write these complete programs off the top of my head I wouldn't be able to either.

The key is building up to it, starting with simple things that work (even if they're not exactly what you want), and going from there.

I can't stress this enough, and it's exactly what we'll do with these projects.

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 (read eval print loop) on the right, as well as getting comfortable with the short cut keys to quickly move between them. You should keep doing that as you work through these projects. More on how exactly to set this up is in Appendix A.

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.

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.

How This Book is Organized

In the next chapter, we'll load and get familiar with the simulation data we'll be working with.

Then we'll dive into the projects themselves, starting from nothing and working our way up. This book also includes final versions of the projects that you should be able to adopt to your own teams and leagues with minimal configuration.

2. Monte Carlo Analysis

How to think about fantasy football:

1. Every week, **a player's fantasy score is a random draw from some distribution.**
2. The shape of **this distribution is different for every player, every week.** What factors go into it? Anything that might affect performance (talent, opportunity, quality of the opposing defense, how good the player's offensive line is, etc).
3. Your job as a fantasy football player is (usually¹) to **assemble a lineup of guys with distributions as far to the right as you can.** This is what maximizes your probability of winning.

If any of that is confusing, or if you're not sold on player performances as draws from probability distributions (or if you're not sure what a probability distribution even *is*), no problem. See Appendix C at the end of this guide, where we build up this intuition from scratch.

Viewing fantasy football this way is a necessary and good first step; next is figuring out how to actually work with these distributions. In theory, we have two options:

Option 1 for working with distributions: math

The first way to work with distributions is by manipulating certain mathematical equations that — when plotted with X, Y coordinates — make curves.

For example, given some numbers for average and standard deviation, you could draw a Normal distribution by plotting this equation:

$$P(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$$

Figure 0.1: Equation for a normal distribution

But there are some major drawbacks with this approach:

¹Later we'll see that — sometimes, due to correlations between players or player variance — the player who gives us the best chance of winning isn't always the one who we think will score the most points on average.

- One is that they restrict you to certain types of curves and shapes. The above is Normal distribution. Other equations give other curves with other games. Gamma, Poisson, Weibull, etc. There are a lot of curves, but not all data follows these shapes.
- An even bigger problem is that working with the math involved is tedious and difficult at best and literally impossible at worse.

Say I told you Justin Herbert and Patrick Mahomes's projections this week are both normally distributed $\sim N(18,6)$ and $\sim N(20.5,7)$ respectively.

Quick, using the equation in Figure 0.1, tell me — what's the probability Justin Herbert scores more than Patrick Mahomes?

Option 2 for working with distributions: simulations

Simulations are a much easier way of working with distributions, especially when you know how to code. Analysis that works with simulations is also called **Monte Carlo** analysis.

Technically, simulations are a bunch of draws from some data generating process. Often that process is a mathematical distribution like a Normal or Gamma distribution. But we can also work with data generating processes that are difficult or impossible to describe mathematically. In that case simulations are our only option.

In practice, most simulation analysis involves:

1. *Generating data* according to some process that reflects the aspects of reality you care about. Sometimes this means finding the right distribution, other times it means combining distributions or generating data according to other rules.
2. *Asking questions* and analyzing this simulated data via summary statistics or plots.

Here, I've mostly taken care of (1) for you by modeling, and then providing via an API, thousands simulated fantasy scores for each player. During the season, these will be updated each week.

Note although access to these simulations comes with the kit, we'll still be tweaking and putting these simulations together in ways that fall under the data generating process.

So these projects mostly involve (2). But the nature of simulation data makes it a great way to practice the Python and Pandas and DataFrame skills we learned about in Learn to Code with Fantasy Football. All the projects in this guide are Monte Carlo analysis that work with the same raw, underlying simulations that power [Fantasy Math](#).

The developer kit comes with historical access to the API (2021-2023 data, which is what we'll use when walking through the examples), as well as access for the 2025 season.

In this next section we'll look at the Fantasy Math Simulation API and play around with the helper functions I've provided.

3. utilities.py

How to Read This Chapter

This chapter — like the rest of coding chapters in the book — is meant to be coded along with. All the code in this chapter are included in the Python files `utilities.py`.

Assuming you're working in Spyder, should open these files in your editor and run the code as you work through this. Since this our first time doing that (in this book), let's do a quick Spyder review.

Spyder

More on this is in [Appendix A](#), but briefly, your screen in Spyder should look like this.

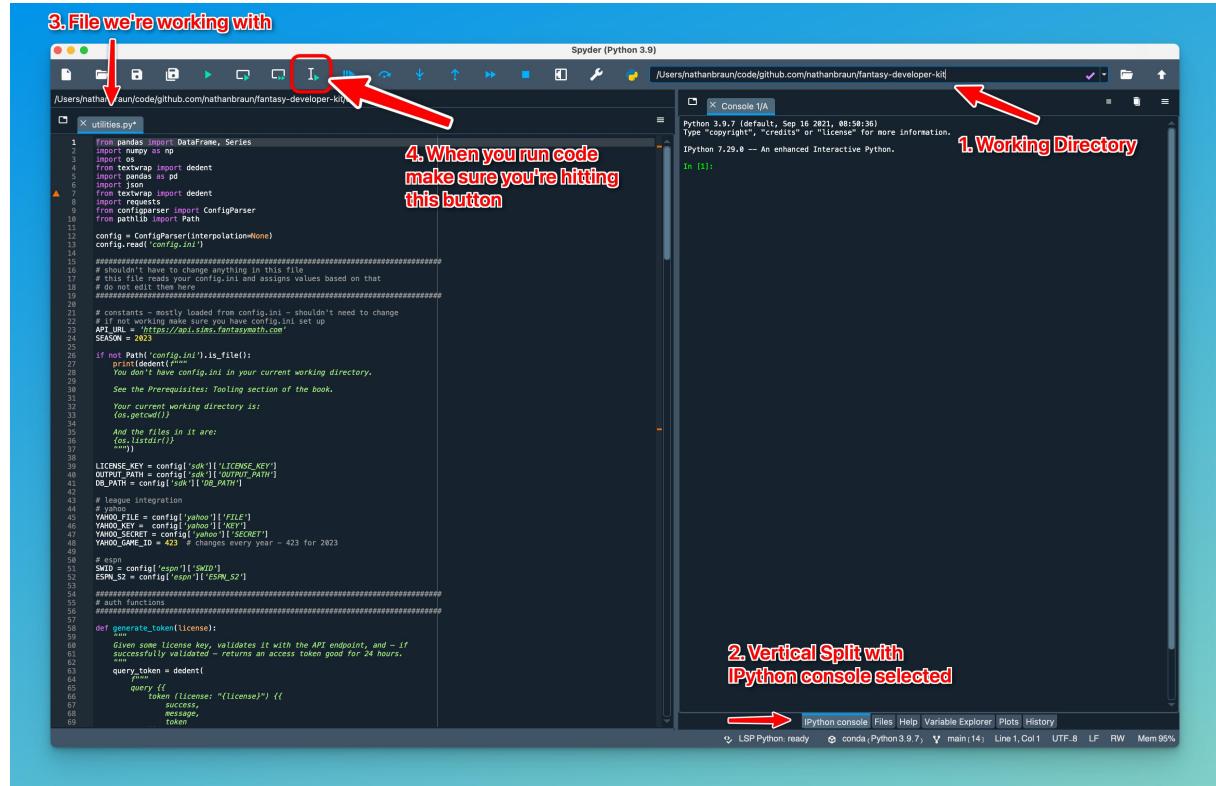


Figure 0.1: Spyder

Notes:

1. Spyder shows your *working directory* up here. **This needs to be your fantasy-developer-kit** (or fantasy-developer-kit-main if you got the files from github) **directory**. To change it, click the Folder icon in the top right.

Note this needs to be your working directory for all the files we run in this book.

2. I like having things set up so my editor is on the left, and my console/REPL is on the right. To get this in Spyder go to View → Window layouts and click on Horizontal split. Then make sure IPython console is selected on the bottom.
3. In this screenshot we can see we have `utilities.py` open in our editor. To open a file click the folder icon.

Finally, to run code, we'll:

4. **Highlight the code we want to run and press the button that looks like a capital I with a play button next to it.** F9 is the keyboard shortcut. Note: *don't* press the regular ► button. That runs the whole file.

When you run some code, I've included what you'll see in the console/REPL here in the book. So if you highlighted `1 + 1` in the editor and pressed F9 your REPL would show:

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

Where the line starting with `In [1]` is what you send, and `Out[1]` is what the REPL prints out.

These are lines [1] for me because this was the first thing I entered in a new REPL session. Don't worry if the numbers you see in `In[]` and `Out[]` don't match exactly what's in this chapter. In fact, they probably won't, because as you run this code 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.

The code usually builds on itself (remember, the REPL keeps track of what you've run previously), so if something's not working, it might be relying on something you haven't run yet.

More on my recommended setup is in [Appendix A](#).

Prerequisites

When you purchased this guide, you received a License Key via email. That license is required to access the simulation API we'll be working with. In the [prerequisites chapter](#), we talked about where to get it and how to put it into `config.ini`. This chapter assumes you've done that.

Setup, Authorization, Fantasy Math API, Accessing the Simulations

```
--name == '__main__'
```

So we have `utilities.py` open. Scroll down to the bottom. You should see a line:

```
if __name__ == '__main__':
    ...
```

With a bunch of code indented below it. This convention is fairly common in Python. It basically lets you separate your functions and parameters (above `__name__ == '__main__'`) and code you may want to run (below it).

Here, above `__name__ == ...` I've written a bunch of utility functions that we'll use in other programs. Below it, I've included an example of how the authorization process works.

Authorization Workflow

Let's run everything in `utilities.py` above `__name__ == '__main__'` (through line 318). Again highlight those lines, then press the `I->` button (or F9).

It should run. This code doesn't *evaluate* to anything (it's just code, it's not like $1 + 1$, which equals 2) so it doesn't print anything `Out []`. Instead after you run it you'll just see `In [2]` waiting for more input.

Next run:

```
In [2] token = generate_token(LICENSE_KEY) ['token']
```

(Quick tip: if it's just one line, you don't have to highlight the code before running it. `I->` will run the line the cursor is on. In practice, you'll often be able to move from line to line in the editor, pressing F9 on the lines you want to run.)

We created a token, when we look at it (keep running the code) we see something like this:

```
In [3]: token
Out[3]: 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJxaWNlbnNlIjoiss
DN0M1MzMtQkM5MDQxRDctOTdDRUE2MEQtMUY2NTRENUiLCJleHAiOjE2MDE1MzgxOTN9.
VtzUrgrq9DMt679Exq1oBW0bOsGP04sf9YHBQeV2NJ4'
```

In general, our basic authorization workflow will be:

1. You use your license key and the `generate_token` function from `utilities.py` to create an *access token*, which is good for 24 hours (after 24 hours you just generate another one).
2. You include the access token every time you query the Fantasy Math API.

To make sure your token is working correctly you can pass it to the `validate` function I've provided.

You should see:

```
In [4]: validate(token)
Out[4]: {'validate': {'validated': True,
                     'message': 'Authentication successful.'}}
```

If you see that you're all set.

GraphQL

The Fantasy Math API is built using GraphQL. I'm not going to spend a lot of time on it, and feel free to skip this section if you're not interested in the details, but GraphQL is basically a way for you as the user of the API to have some say in what data you get back.

You get that by sending some JSON (which remember is similar to Python dictionaries and lists) to the API that describes the data you want.

For example, the Fantasy Math API has a GraphQL endpoint, `available`, which takes a season and week and returns a list of all the player ids you can choose from.

Here's how you'd call it using a multi line string + the `json` package (note we're still at the bottom of `utilities.py`):

```
In [5]:  
    QUERY_STR = """  
        query {  
            available (season: 2023, week: 1) {  
                player_id,  
                name,  
                pos,  
                actual  
            }  
        }  
    """  
  
In [6]:  
r = requests.post(API_URL, json={'query': QUERY_STR},  
                  headers={'Authorization': f'Bearer {token}'})  
  
In [7]: df = DataFrame(json.loads(r.text)['data']['available'])
```

This returns a `DataFrame` of every player in the Fantasy Math API for week 1, 2023.

```
In [8]: df.sample(10)  
Out[8]:  
      player_id          name pos  actual  
248       770  Diontae Johnson  WR    7.80  
542      1200  Harrison Butker  PK    9.40  
76       1288     Jared Goff  QB   18.55  
201       69    Luke Musgrave  TE     NaN  
278      1139    Cooper Kupp  WR     NaN  
440       929    Calvin Ridley  WR   24.10  
205       585    Adam Trautman  TE    8.40  
40        1035    Greg Joseph  PK    4.50  
496      1167    David Njoku  TE    4.40  
265       379   Kadarius Toney  WR    1.00
```

This `DataFrame` includes the `player_id`, `position` and `actual` columns because that's

what we requested via GraphQL. If we wanted to drop, say, `actual`, we could. This flexibility is what GraphQL gets you over a traditional REST API.

Included Helper Functions

In the last section we called the `available` route of the Fantasy Math API directly using the `requests` library and a multi-line string. But for these projects, I've provided a few helper functions to make it easier.

There are four main functions:

generate_token

We've already talked about `generate_token` (and its sidekick `validate`). They're just functions you can use to generate and check your access tokens.

get_players

The purpose of `get_players` is to get a list of eligible Fantasy Math player ids (`player_id`), which you can use to pick out which players' simulations you want.

By default, `get_players` function returns the players available for the current week in the 2025 season. You can ask for eligible IDs from prior weeks by passing a season (2021 or later) and week (1-18) argument.

It also takes a scoring system arguments. Options here are:

`'pass_4'` or `'pass_6'` for the `qb` argument.

`'ppr_0'`, `'ppr_1'` or `'ppr_lover2'` (e.g. standard, ppr or half-ppr) for the `skill` argument.

`'dst_std'` or `'dst_high'` for `dst` scoring. Note you should probably just use `dst_std` unless your league gives a lot of points for turnovers and sacks.

master_player_lookup

The `master_player_lookup` function returns a DataFrame of all player ids (FantasyMath, Fleaflicker, ESPN, Yahoo and Sleeper) for most players going back to 2021. This will be useful for the league integration project.

get_sims

The `get_sims` function returns the actual simulations.

It takes a list of `player_id`'s (which you can get by first calling `get_players`) as well as scoring system arguments (`qb`, `skill`, `dst`).

Just like `get_players`, `get_sims` defaults to the current week in the 2025 season. If you pass it a season and week argument it'll return past simulations.

It also takes an `nsims` argument for the number of simulations to return (up to 500).

4. Introduction to the Data

How to Read This Chapter

Note: the following assumes you have `intro.py` open in your editor, along with a REPL you can send code to. If you have your same session open from last chapter, that's no problem, just type `reset` and `clear` into it to to clear everything else.

Now that we've seen our utility functions, let's take a first look at the data. We'll start in `intro.py` by importing Pandas and our utility functions:

```
In [1]:  
import pandas as pd  
from os import path  
from utilities import (LICENSE_KEY, generate_token, get_players, get_sims,  
name_sims)
```

Reminder About Your Working Directory

Notice we're importing some of the functions and constants from our `utilities.py` file. In order to do this, you *need* your working directory in Spyder to be wherever these files are.

If you ever get an error like:

```
In [2]:  
from utilities import (LICENSE_KEY, generate_token, get_players, get_sims,  
name_sims)  
...  
ModuleNotFoundError: No module named 'utilities'
```

You're not in the right working directory. More on this [here](#).

Simulation Data

OK, back to our first look at the data. After we've successfully imported our utility functions, we'll set some parameters and get an access token:

```
In [3]:  
SEASON = 2023  
WEEK = 3  
SCORING = {'qb': 'pass_4', 'skill': 'ppr_1', 'dst': 'dst_std'}  
  
In [4]: token = generate_token(LICENSE_KEY)['token']
```

Remember, we'll need to pass this token to every API call so it knows we have permission to query the data.

We can start by getting a list of players:

```
In [6]:  
players = get_players(token, **SCORING, season=SEASON,  
                      week=WEEK).set_index('player_id')  
  
In [7]: players.head()  
Out[7]:  
          name  pos  ...  team  actual  
player_id  
1091      Patrick Mahomes  QB  ...  KC  25.68  
498       Jalen Hurts  QB  ...  PHI  19.88  
889       Josh Allen  QB  ...  BUF  21.32  
1094      Joshua Dobbs  QB  ...  ARI  17.06  
329       Zach Wilson  QB  ...  NYJ   6.38
```

Note, this is live data from the Fantasy Math simulation API. I anticipate the API working when you read this. But in case you want to work through these projects in the offseason, or in the future after your annual access has expired. Or if you just want to make sure you're seeing exactly what I'm seeing, I've saved all the data locally.

Let's make a boolean constant that keeps track of which version of the data to use (saved vs live):

```
In [8]: USE_SAVED_DATA = True
```

Now we can use this code:

```
In [9]:  
if USE_SAVED_DATA:  
    players = get_players(token, **SCORING, season=SEASON,  
                          week=WEEK).set_index('fantasymath_id')  
else:  
    players = (pd.read_csv(path.join('data', 'players.csv'))  
               .set_index('fantasymath_id'))
```

It's the same data (I made it by querying and saving a snapshot of the API) but it's a way to make sure we're seeing the exact same thing.

Because we're working with past data (week 3, 2023) the player function returns actual scores. We can use this to see how the model actually performed. If we wanted to get player ids for *this* week in 2025 we'd leave the `week` and `season` arguments off and do: `get_players(token, **SCORING)`. In that case `actual` would be empty.

Querying sims

Once we have a list of player ids, we can pass them to the simulation function:

```
In [10]:  
if USE_SAVED_DATA:  
    sims = pd.read_csv(path.join('data', 'sims.csv'))  
else:  
    sims = get_sims(token, players=list(players.index), week=WEEK,  
                    season=SEASON, nsims=5000, **SCORING)  
  
In [11]: sims.head()  
Out[11]:  
   1091      498     ...      5152      5169  
0  31.373173  22.943242  ...  16.655954  0.079218  
1  15.580346  21.278506  ...  14.725048  7.598453  
2  19.305332  20.131436  ...  0.677490  11.846076  
3   5.358195  32.302340  ...  15.128840  9.830206  
4  19.384985  39.341403  ...  12.699372  13.258936
```

Each simulation is a row (there are 500, though we're just looking at the first 5) and each player is a column.

The columns are named by `player_id`. So the first two columns, 1091 and 498 are Patrick Mahomes and Jalen Hurts:

```
In [12]: players.loc[[1091, 498]]  
Out[12]:  
          name  pos  ...  team  actual  
player_id  
1091      Patrick Mahomes  QB  ...    KC  25.68  
498       Jalen Hurts   QB  ...  PHI  19.88
```

These simulations are OK — we have what we need in `players` to see who's who, but it'd be nice to be able to tell who we're looking at. To do that, I've included a function called `name_sims` (we imported it from `utilities` above) that takes your `sims`, `players` and names the columns. Let's try it:

```
In [13]: nsims = name_sims(sims, players)

In [14]: nsims.head()
Out[14]:
   patrick-mahomes  jalen-hurts  ...      atl      lv
0            31.373173    22.943242  ...  16.655954  0.079218
1            15.580346    21.278506  ...  14.725048  7.598453
2            19.305332    20.131436  ...  0.677490  11.846076
3             5.358195    32.302340  ...  15.128840  9.830206
4            19.384985    39.341403  ...  12.699372  13.258936
```

That's better.

Working with simulations

So we have our simulations. Let's take a player, say Justin Herbert. 500 simulations. What can we do with this?

Well, if we're interested in projecting how many points he'll score on average:

```
In [15]: sims['justin-herbert'].mean()
Out[15]: 21.976058849476672
```

Or the median:

```
In [16]: sims['justin-herbert'].median()
Out[16]: 21.55108079040353
```

These are just your standard Pandas summary functions. But the real benefit of simulations is how they let you work with distributions. We can get a sense of the distribution with summary statistics:

```
In [17]:
nsims['justin-herbert'].describe(percentiles=[0.05, .25, .5, .75, .95])

Out[17]:
count      500.000000
mean       21.976059
std        8.513235
min        0.606958
5%         9.045559
25%        15.900111
50%        21.551081
75%        27.480257
95%        36.132164
max        49.746482
```

Or by visualizing it:

```
g = sns.FacetGrid(nsims, aspect=2)
g = g.map(sns.kdeplot, 'justin-herbert', fill=True)
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle("Justin Herbert's Fantasy Points Distribution - Wk 3, 2023")
```

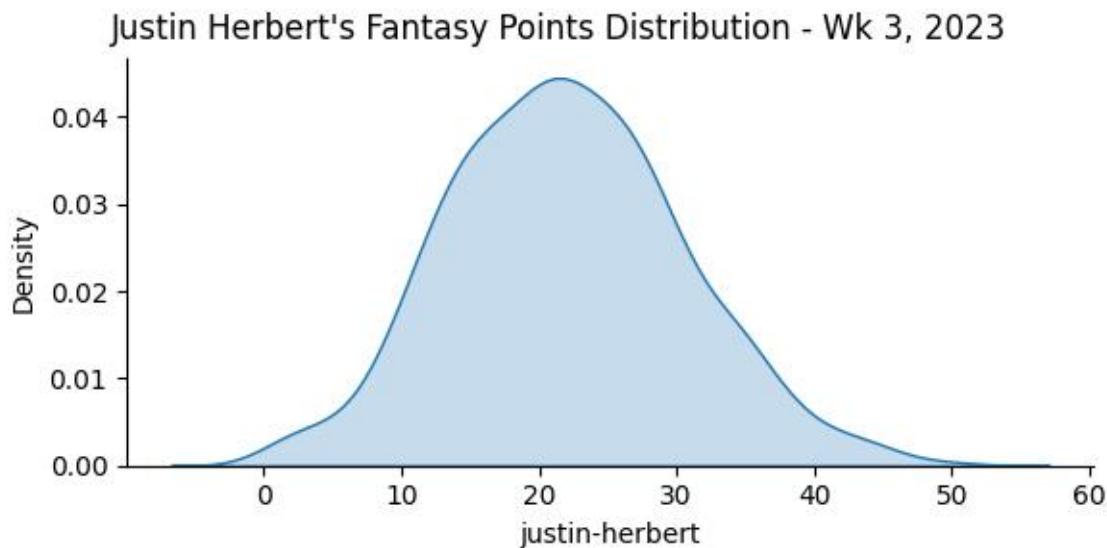


Figure 0.1: Herbert distribution

Will Herbert outscore Mahomes?

Remember above, when I asked for the probability Justin Herbert outscores Mahomes?

Given a little Pandas knowledge, figuring this out via simulations is easy. We have:

```
In [18]: sims[['justin-herbert', 'patrick-mahomes']].head()
Out[18]:
   justin-herbert  patrick-mahomes
0      24.889975      31.373173
1      19.695595      15.580346
2      25.814006      19.305332
3      14.212815       5.358195
4      22.572415      19.384985
```

Then it's just a matter of making a boolean column:

```
In [19]: (sims['justin-herbert'] > sims['patrick-mahomes']).head()
Out[19]:
0    False
1     True
2     True
3     True
4     True
```

And taking the average of it:

```
In [20]: (sims['justin-herbert'] > sims['patrick-mahomes']).mean()
Out[20]: 0.426
```

All of which are basic data manipulation concepts we covered in the first few chapters of Learn to Code with Fantasy Football.

So Herbert had a 42.6% of outscoring Mahomes in week 3, 2023.

This means Mahomes distribution is further to the right. If we want to see what that looks like:

```
In [21]:
nsims_long = nsims[['justin-herbert', 'patrick-mahomes']].stack().
    reset_index()
nsims_long.columns = ['sim_n', 'player', 'pts']

g = sns.FacetGrid(nsims_long, hue='player', aspect=2)
g.map(sns.kdeplot, 'pts', fill=True)
g.add_legend()
```

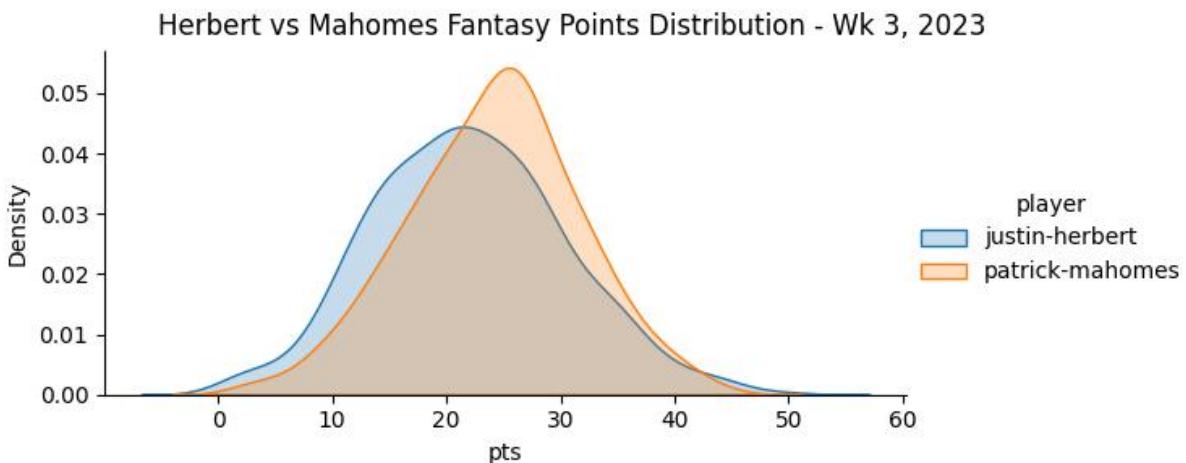


Figure 0.2: Herbert vs Mahomes

Other Monte Carlo Applications

What else should we ask? When working with simulations, you're really only limited by your imagination. For example, what's the probability Justin Herbert outscores both Matt Stafford AND Justin Fields by at least 11.5 points?

Can you imagine working with the probability distribution equations above to figure that out mathematically? But, again, with simulations it's easy:

```
In [22]:  
    (sims['justin-herbert'] >  
     sims[['matthew-stafford', 'justin-fields']].max(axis=1) + 11.5).  
     mean()  
--  
Out[22]: 0.158
```

So about an 16% chance.

The other benefit: Monte Carlo analysis is flexible. You can take into account whatever you want, as long as you figure out how to build it into the data generating process.

Projecting best ball distributions

For example, say we're interested in projecting QB points for our best ball team, where we have both Herbert and Stafford.

This is something that's impossible to figure out numerically, but easy to build into your simulation:

1. Take a draw from each player, then —
2. Use the higher of the two scores for your QB spot .
3. Analyze that QB score with some basic summary stats.

In this case our data generating process isn't just approximating reality, it's describing it exactly. This is how best ball leagues work.

```
In [23]: sims['bb_qb'] = sims[['justin-herbert', 'matthew-stafford']].max(axis=1)

In [24]: sims[['bb_qb', 'justin-herbert', 'matthew-stafford']].describe()
Out[24]:
      bb_qb  justin-herbert  matthew-stafford
count  500.000000    500.000000    500.000000
mean   24.183287    21.976059    15.842432
std    7.519747     8.513235    8.145466
min    4.254415     0.606958    0.027773
25%   19.197732    15.900111    9.984359
50%   23.801324    21.551081   15.836984
75%   28.720037    27.480257   21.395072
max   49.746482    49.746482   40.711039
```

So on average, we'd expect a best ball duo of Herbert and Stafford to score about 24.18 points, vs 21.98 and 15.84 for Herbert and Stafford individually.

What if we want to see how much adding Kirk Cousins would raise our score?

```
In [25]:
sims['bb_qb2'] = sims[['justin-herbert', 'matthew-stafford',
                       'kirk-cousins']].max(axis=1)

In [26]:
sims[['bb_qb2', 'bb_qb', 'justin-herbert', 'matthew-stafford',
      'kirk-cousins']].describe().round(2)
Out[26]:
      bb_qb2  bb_qb  justin-herbert  matthew-stafford  kirk-cousins
count  500.00  500.00    500.00      500.00      500.00
mean   27.03  24.18    21.98      15.84      21.64
std    7.01   7.52     8.51      8.15      8.15
min   12.35   4.25     0.61      0.03      0.37
25%   22.15  19.20    15.90      9.98     16.07
50%   26.38  23.80    21.55     15.84     20.99
75%   31.69  28.72    27.48     21.40     26.96
max   52.09  49.75    49.75     40.71     52.09
```

By modifying the data generating process and drawing a bunch of simulations, we're able to answer questions about our data it'd be impossible to figure out any other way.

Actual scores vs projections

Because we're looking at data from last year, we know how many points these players actually scored, in the `actual` column:

```
In [27]: players.head()
Out[27]:
          name  pos  ...  team  actual
player_id
1091      Patrick Mahomes  QB  ...  KC  25.68
498       Jalen Hurts  QB  ...  PHI  19.88
889       Josh Allen  QB  ...  BUF  21.32
1094      Joshua Dobbs  QB  ...  ARI  17.06
329       Zach Wilson  QB  ...  NYJ  6.38
```

So Mahomes scored 25.68 points in Week 2, 2023. How does that compare with what we simulated for him beforehand?

Well, our median Mahomes simulation was:

```
In [28]: nsims['patrick-mahomes'].median()
Out[28]: 24.49372359176565
```

So pretty close. What about percentile-wise?

```
In [29]: (25.68 > nsims['patrick-mahomes']).mean()
Out[29]: 0.56
```

So Mahomes score in Week 3 beat 56% of his simulations, i.e. it was a *56th percentile* score.

Let's get projected (by taking the average of the simulations) for all the quarterbacks:

```
In [26]: qbs = players.loc[players['pos'] == 'QB']

In [27]: qbs['proj'] = sims.mean().round(2)

In [28]: qbs.sort_values('proj', ascending=False).head(15)[
            ['name', 'proj', 'actual']]
Out[28]:
```

player_id		name	proj	actual
1091		Patrick Mahomes	24.11	25.68
494		Justin Herbert	21.98	29.30
2085		Kirk Cousins	21.64	25.68
498		Jalen Hurts	21.35	19.88
893		Lamar Jackson	21.24	28.18
889		Josh Allen	21.01	21.32
493		Tua Tagovailoa	19.95	28.36
327		Trevor Lawrence	19.10	14.36
1297		Dak Prescott	18.67	14.36
1883		Geno Smith	17.72	15.44
328		Justin Fields	17.56	10.66
1088		Deshawn Watson	17.25	21.16
492		Joe Burrow	17.24	8.16
1288		Jared Goff	17.19	18.02
156		Brock Purdy	16.61	20.30

Now let's calculate each quarterback's percentile too:

```
In [30]:  
def fpts_percentile(row):  
    return (row['actual'] > sims[row.name]).mean()  
--  
  
In [31]: qbs['pctile'] = qbs.apply(fpts_percentile, axis=1)  
  
In [32]:  
qbs.sort_values('proj', ascending=False).head(15)[['name', 'proj', 'actual',  
        'pctile']]  
--  
Out[32]:
```

player_id		name	proj	actual	pctile
1091		Patrick Mahomes	24.11	25.68	0.560
494		Justin Herbert	21.98	29.30	0.808
2085		Kirk Cousins	21.64	25.68	0.694
498		Jalen Hurts	21.35	19.88	0.394
893		Lamar Jackson	21.24	28.18	0.808
889		Josh Allen	21.01	21.32	0.536
493		Tua Tagovailoa	19.95	28.36	0.854
327		Trevor Lawrence	19.10	14.36	0.246
1297		Dak Prescott	18.67	14.36	0.284
1883		Geno Smith	17.72	15.44	0.402
328		Justin Fields	17.56	10.66	0.206
1088		Deshaun Watson	17.25	21.16	0.706
492		Joe Burrow	17.24	8.16	0.136
1288		Jared Goff	17.19	18.02	0.558
156		Brock Purdy	16.61	20.30	0.688

So Justin Herbert scored 29.30 points vs his projection of 21.98. About an 80th percentile week. We'd expect to see an 80th performance about 1 out of every 5 times. And sure enough, if you look at the 15 players above, three of them (Herbert, Lamar Jackson, Deshaun Watson) have a pctile > 80.

Similarly, we should expect to see 10% of the players under 10 pctile, 20% under 20, etc. And actually it's pretty close:

```
In [33]: qbs['pctile'].describe()
          percentiles=[.1, .2, .3, .4, .5, .6, .7, .8, .9])
Out[33]:
count    31.000000
mean     0.487032
std      0.264878
min      0.014000
10%      0.140000
20%      0.232000
30%      0.290000
40%      0.394000
50%      0.520000
60%      0.560000
70%      0.694000
80%      0.738000
90%      0.808000
max      0.892000
```

That is, 14% of players had 10th percentile or less draws, 23% of players, 20th percentile, etc. That's the sign of a good model.

Percentiles as luck

A player's actual point percentile is one way to quantify "luck".

Good draws from further right in the distribution have higher percentiles, and are luckier. Worse draws, from further left, are unlucky.

Let's look at the top 10 luckiest fantasy QB performances in Week 3, 2023:

```
In [34]:
qbs.sort_values('pctile', ascending=False).head(10)[['name', 'proj', 'actual',
                                                       'pctile']]
--
```

player_id	name	proj	actual	pctile
497	Jordan Love	14.49	24.26	0.892
2301	Andy Dalton	12.94	23.54	0.884
493	Tua Tagovailoa	19.95	28.36	0.854
1094	Joshua Dobbs	11.12	17.06	0.808
893	Lamar Jackson	21.24	28.18	0.808
494	Justin Herbert	21.98	29.30	0.808
150	Kenny Pickett	13.83	18.50	0.738
1757	Jimmy Garoppolo	13.19	17.66	0.716
1088	Deshawn Watson	17.25	21.16	0.706
2085	Kirk Cousins	21.64	25.68	0.694

(Aside: remember — this is from week 3, 2023. Now in 2025, we know Jordan Love is good (he just became the highest paid QB ever). But these distributions were built using the best information we had in September 2023. Back then opinions about Love were much more up in the air, which meant his projected distribution was further to the left. So while Love might have been good the whole time, and his success isn't "luck" from his POV, you were indeed lucky (or ahead of the fantasy consensus) to get these kind of points out of Love back in early 2023.)

And here are the unluckiest players:

In [35]:				
qbs.sort_values('ptile', ascending=False).tail(10)[['name', 'proj', 'actual', 'ptile']]				
Out[35]:				
player_id		name	proj	actual
890	Baker Mayfield	14.47	10.04	0.290
1297	Dak Prescott	18.67	14.36	0.284
327	Trevor Lawrence	19.10	14.36	0.246
1689	Derek Carr	14.40	8.12	0.232
155	Desmond Ridder	12.83	6.34	0.216
328	Justin Fields	17.56	10.66	0.206
2082	Ryan Tannehill	11.63	4.16	0.140
492	Joe Burrow	17.24	8.16	0.136
677	Daniel Jones	14.63	3.98	0.082
151	Sam Howell	13.30	0.60	0.014

(Note you could reverse what we Love above and apply it to someone like Fields or Lawrence)

Would you rather be lucky or good?

Would you rather be lucky or good? Well it depends. Patrick Mahomes is good. He has a very rightward projected point distribution. He wasn't particularly lucky in week 3, with a 56th percentile draw.

But because he's good, that still meant 25.68 points. Meanwhile, Andy Dalton was very lucky in week 3 with a 88th percentile draw, but since he's not that good (the median of his distribution is about 13 points) he still scored less than Mahomes.

So really, the best performing players are usually lucky *and* good. We can see that by looking at the top performers:

```
In [36]:  
qbs.sort_values('actual', ascending=False).head(10)[['name', 'proj', '  
actual',  
          'pctile']]  
--  
Out[36]:
```

	name	proj	actual	pctile
player_id				
494	Justin Herbert	21.98	29.30	0.808
493	Tua Tagovailoa	19.95	28.36	0.854
893	Lamar Jackson	21.24	28.18	0.808
1091	Patrick Mahomes	24.11	25.68	0.560
2085	Kirk Cousins	21.64	25.68	0.694
497	Jordan Love	14.49	24.26	0.892
2301	Andy Dalton	12.94	23.54	0.884
889	Josh Allen	21.01	21.32	0.536
1088	Deshawn Watson	17.25	21.16	0.706
156	Brock Purdy	16.61	20.30	0.688

Two of the top three are good QBs (Herbert and Lamar) who got fairly lucky (80th percentile) draws. There's also Tua, who is a bit less good but got an even luckier draw (85th percentile). Then there's Mahomes.

In reality, high score are a combo of luck and skill. But, since you can't predict luck, you should just focus on the skill part, which is what you do when you start players with the best distributions.

Correlations

One of the most useful things about these simulations is that they're correlated.

Correlation summarizes the tendency of numbers to move together. The usual way of calculating correlation (Pearson's) summarizes this tendency by giving you a number between -1 and 1.

Variables with a -1 correlation move perfectly in opposite directions; variables that move in the same direction have a correlation of 1. A correlation of 0 means the variables have no relationship.

For example, the correlation between a QB and his WR1 is about 0.35. As quarterbacks tend to score more, their WR1's tend to score more too. It's not a sure thing — it's possible the QB had a big game by throwing to his WR2 or his TE — but their scores tend to move together.

You can see here that the projections for Herbert and Keenan Allen — though the data is purely simulated — have a correlation around 0.40, just like it'd be in real life.

```
In [1]: nsims[['justin-herbert', 'keenan-allen']].corr()
Out[1]:
      justin-herbert  keenan-allen
justin-herbert      1.000000   0.400943
keenan-allen       0.400943   1.000000
```

Same with Herbert and the DST he was playing against (Vikings this week) — except the correlation between a QB and the DST he's playing against is even stronger, and negative:

```
In [2]: nsims[['justin-herbert', 'min']].corr()
Out[2]:
      justin-herbert    min
justin-herbert      1.000000 -0.547627
min                -0.547627  1.000000
```

Note: these correlated projections definitely aren't an accident or something that just happened. It took a lot of work behind the scenes to (a) figure out what these historical correlations between players in the same game were, (b) come up with a way to simulate draws from players while preserving them, and (c) make sure the distributions were accurate overall.

It's turned out well. We ran these separately above, but what's great about these projections is that they're all correlated simultaneously, that is — Herbert is positively correlated with Keenan Allen (and Ekeler, Mike Williams and his K etc) and negatively correlated with the Vikings DST, all at the same time.

```
In [3]: nsims[['justin-herbert', 'keenan-allen', 'min']].corr()
Out[3]:
      justin-herbert  keenan-allen    min
justin-herbert      1.000000   0.400943 -0.547627
keenan-allen       0.400943   1.000000 -0.236117
min                -0.547627  -0.236117  1.000000
```

This is a *matrix*, e.g. you follow player on the left, and the player on the top and get any correlation. Any player is perfectly correlated with himself, which is why all the diagonals equal 1.

You can see Keenan is also negatively correlated with the Raiders D, but not as much as Murray.

Let's add a few more:

```
In [4]: nsims[['justin-herbert', 'kirk-cousins', 'keenan-allen',
              'justin-jefferson', 'min', 'jordan-love']].corr().round(2)
Out[4]:
          herbert  cousins  keenan  jefferson   min  jordan-love
justin-herbert    1.00    0.22    0.40    0.10 -0.55   -0.01
kirk-cousins      0.22    1.00    0.09    0.27 -0.07   -0.03
keenan-allen      0.40    0.09    1.00    0.11 -0.24   -0.01
justin-jefferson   0.10    0.27    0.11    1.00 -0.08   -0.04
min            -0.55   -0.07   -0.24   -0.08  1.00   -0.04
jordan-love       -0.01   -0.03   -0.01   -0.04 -0.04    1.00
```

We can see Herbert and Cousins — opposing QBs — are positively correlated. Many times (not every time, otherwise the correlation would be 1) gameflow leads to QBs either both doing well (e.g. a shootout) or poorly (e.g. a grind out game).

I also added Jordan Love (who was playing the Saints) so you can see he's not correlated with anyone at all. His correlations with all the LAC-LV players are around around 0 (they're not perfectly 0 since there's noise in the data). This makes sense, since he's playing hundreds of miles away.

Conditional Probabilities

Another way of seeing how these correlations affect players scores is by looking at conditional probabilities. For example, with PPR scoring in week 3, 2023, Keenan Allen's regular, unconditional projected points distribution looked like thi

```
In [5]: nsims['keenan-allen'].describe()
Out[5]:
count    500.000000
mean     17.493676
std      9.447631
min      0.102908
25%     10.331722
50%     16.813358
75%     24.955462
max     44.548022
```

But what if we want to know Allen's range of outcomes, *conditional* on Herbert scoring at least 30 points, an then again when Herbert scores 12 or less.

The fact that their simulations are positively correlated means these distributions will be higher and lower respectively:

```
In [6]:  
pd.concat([  
    nsims.loc[nsims['justin-herbert'] > 30, 'keenan-allen'].describe(),  
    nsims.loc[nsims['justin-herbert'] < 12, 'keenan-allen'].describe()],  
    axis=1)  
--  
Out[6]:  
          keenan-allen   keenan-allen  
count      83.000000      57.000000  
mean      24.816760      11.357782  
std       9.114331      7.474711  
min       0.425437      0.330261  
25%      19.976145      4.729811  
50%      25.946795      11.630839  
75%      30.683469      16.518819  
max      42.979666      29.838108
```

Who should I start Monday night?

Factoring in these correlations (along with the shape of a players point distribution generally) allows us to be more precise with the type of questions we want to answer.

For example, say I was going into MNF week 3, 2023 wondering who should I start: Mike Evans or Tee Higgins?

In terms of a simple, “who will score more points?” the simulations say Evans.

```
In [7]: print((nsims['mike-evans'] > nsims['tee-higgins']).mean())  
Out[7]: 0.564
```

And that's fine. But what if it's Monday night, I'm down 40 points and I also have Joe Burrow, who is positively correlated with Tee Higgins?

```
In [8]: print((nsims[['joe-burrow', 'mike-evans']].sum(axis=1) > 50).mean()  
())  
Out[8]: 0.064  
  
In [9]: print((nsims[['joe-burrow', 'tee-higgins']].sum(axis=1) > 50).mean()  
())  
Out[9]: 0.09
```

My chances with Higgins (9%) and Burrow much better than my chances with Mike Evans (6%) and Burrow, even though Evans usually scores more. This is what taking positional correlations into account allows you to do.

Playing “offense” or “defense” as the underdog or favorite

Let's look at a simplified fantasy matchup. Say we're in a 1 QB, 1 DST league. Our opponent this week is good, with Justin Herbert (2nd ranked QB) and the Cowboys Defense (3rd ranked DST). Our QB is the solidly average (10th ranked QB) Geno Smith, and our DST options aren't great either. We're trying to decide whether to start the Steelers (ranked 16) or Minnesota (ranked 19).

Pittsburgh DST averages more points than the Vikings DST:

```
In [10]: nsims[['min', 'pit']].describe()
Out[10]:
      min          pit
count  500.000000  500.000000
mean    7.078593   7.657250
std     5.308242   6.036458
min     0.008882   0.026179
25%    2.772932   2.548929
50%    6.077938   6.454133
75%   10.293269  11.583480
max   27.438368  31.974950
```

In a vacuum, we'd definitely start them. But the Vikings D is playing against Herbert this week, so their scores are negatively correlated. If Herbert has a bad game, the Vikings D is more likely to have a good game, and vis versa. Pittsburgh (who is playing the Raiders) is uncorrelated (the ~0.03 correlation is noise).

```
In [11]: nsims[['justin-herbert', 'min', 'pit']].corr()
Out[11]:
           justin-herbert      min          pit
justin-herbert        1.000000 -0.547627  0.034049
min                 -0.547627  1.000000 -0.030473
pit                  0.034049 -0.030473  1.000000
```

Because we're the heavy underdog (Geno and a not great D vs Herbert and the top ranked D), the negative correlation between MIN and Herbert helps us. Basically, we're down so much that we need two things to go “right” for us to win: a bad Herbert game for our opponent, and a good DST game from us. Both of those together are more likely to happen when we start the Vikings. And we can see it in the data:

```
In [12]: print((nsims[['geno-smith', 'pit']].sum(axis=1) >
               nsims[['justin-herbert', 'dal']].sum(axis=1)).mean())
Out[12]: 0.312

In [13]: print((nsims[['geno-smith', 'min']].sum(axis=1) >
               nsims[['justin-herbert', 'ne']].sum(axis=1)).mean())
Out[13]: 0.324
```

Starting the Vikings gives us slightly better odds, even though Pittsburgh is the better D.

Again, this is only because we're a heavy underdog. What happens if increase our overall odds by starting Mahomes instead of Geno?

```
In [14]:  
print((nsims[['patrick-mahomes', 'pit']].sum(axis=1) >  
      nsims[['justin-herbert', 'dal']].sum(axis=1)).mean())  
--  
0.49  
  
In [14]:  
print((nsims[['patrick-mahomes', 'min']].sum(axis=1) >  
      nsims[['justin-herbert', 'dal']].sum(axis=1)).mean())  
--  
0.454
```

Now we can see the Vikings are the much worse option, lowering our odds almost 4 percentage points.

Simulating your entire lineup

Of course, the natural extension is figuring out — not just the probability your QB and WR outscore your opponents QB — but the probability ALL your players outscore ALL your opponents, taking into account *all* correlations in *all* games.

That's the precise question we care about in fantasy football (vs the similar but subtly different "who will score more"), and it's one these simulations will help you answer later on in the book.

What Goes into the Simulations

Exactly *how* these simulations are calculated is beyond the scope of this book. I'm not saying that because I'm trying to keep anything from you. Rather, they're done with an advanced modeling technique (Bayesian modeling with Markov chain Monte Carlo sampling using the Python package PyMC) that requires a lot more math to explain.

In the last chapter of LTCWFF, we talked about modeling, and gave examples of models that would predict (say) a players projected fantasy points given a bunch of input variables (weather, consensus ranking, etc)¹.

This model and both similarities and differences.

¹: In fact, this is how point projections on most major sites are still done.

The similarities: like more basic models, the Fantasy Math models are taking some (A) inputs and generating (B) output. The difference: in this case that output — rather than being some single number — is sophisticated set of (blended) distributions.

The main things that go into these models: 1. Weekly expert consensus rankings (higher ranked players are better, players who experts are split on have wider ranges of outcomes) 2. Betting info, i.e. game totals and spreads.

Along with:

3. Historical correlations between players in the same game (e.g. a QB vs his WR1).

5. Quickstart: Analyze Your Team and League

Hopefully you're starting to get a sense of what these simulations will let you do. In the rest of the book we'll be use them to slowly build up real tools

But this kit comes with working, polished copies of the code for all the projects, and I wanted to show you how to get started with them here.

The most convenient way to use is by having them automatically grab information from your league, and that's what we'll do here.

This guide gives instructions + code for connecting to ESPN, Yahoo, Fleaflicker and Sleeper leagues. The setup is different depending on the site. It's easy for Fleaflicker and Sleeper; a bit more involved for Yahoo and ESPN.

League Website Specific Setup

See below for [Fleaflicker](#), [Sleeper](#), [ESPN](#), and [Yahoo](#) specific instructions. Once you've followed them, you can jump down to the next section.

Fleaflicker Setup Quickstart

To get what we need from Fleaflicker you need to know two things. Your:

- league id
- team id

You can get these by going to your main team page:

<https://www.fleaflicker.com/nfl/leagues/316893/teams/1605156>

Here, my league id is 316893 and my team id is 1605156. Make note of what these are (or at least don't close the browser page). When you have that you can jump to the [next section](#).

Sleeper Setup Quickstart

To get connect to Sleeper you need to know your league and team id. You can find your league in your league URL:

<https://sleeper.app/leagues/this-is-your-league-id>

For example, my league id is 1002102487509295104.

Once you have that visit:

https://api.sleeper.app/v1/league/league_id/users

in your browser.

It'll return a bunch of JSON showing data on everyone in the league. Find yourself (look at the `display_name` field) and make note of which spot you're in in the list.

For example, here:

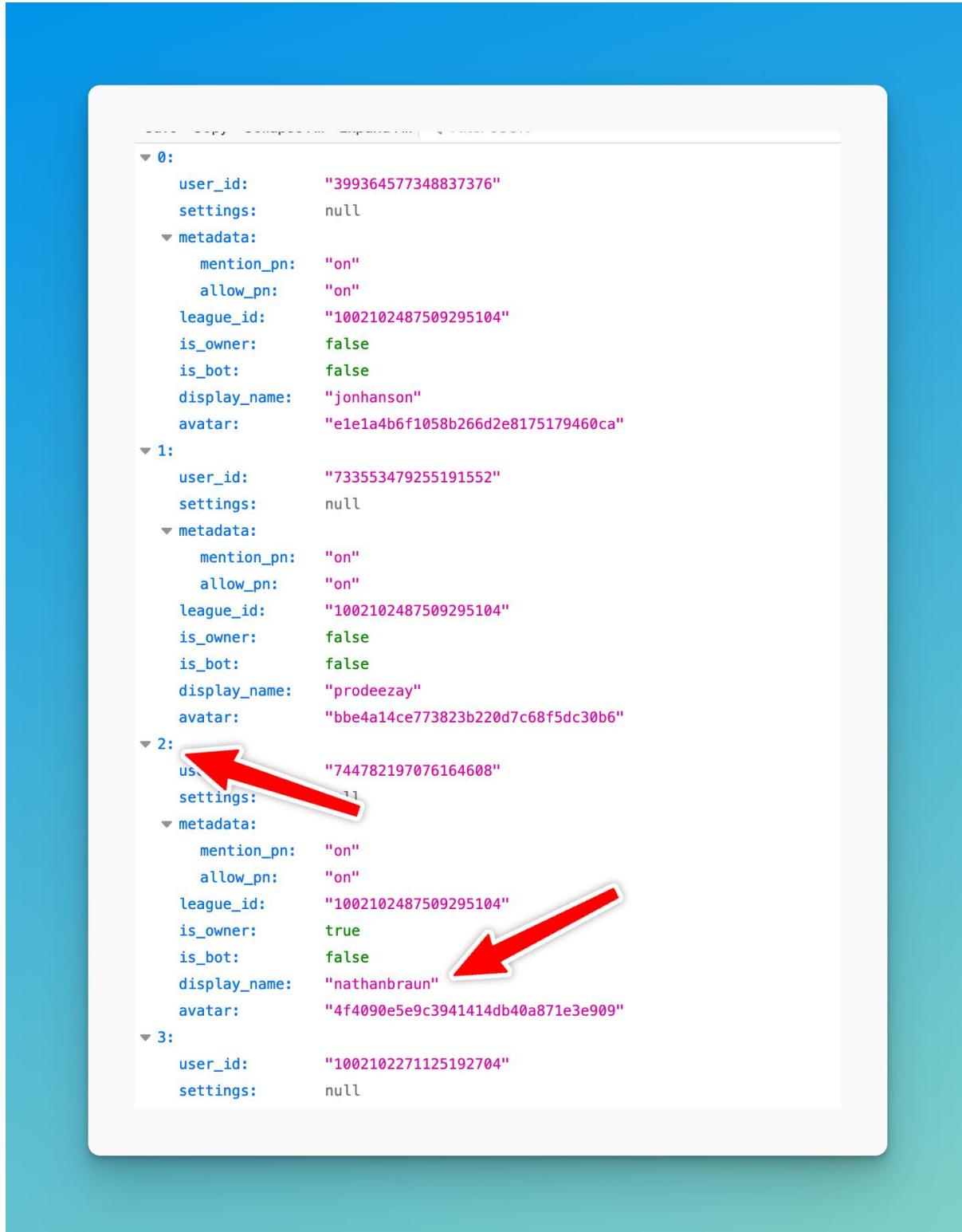


Figure 0.1: Finding team id in Sleeper

My team id is 2 (it's the third spot, the ids start at 0).

When you have that you can jump to the [next section](#).

ESPN Setup Quickstart

Note: this is meant to get you up and running and quickly as possible. For more on what we're doing and why we have to do it check out the [ESPN league integration section](#).

1. Log into ESPN fantasy and go to your main team page.

Navigate to your main team page. The link for mine is:

<https://fantasy.espn.com/football/team?leagueId=242906&seasonId=2021&teamId=11&fromTeamId=11¹>

In the URL we can see both the league id (mine is 242906) and team id (mine is 11). Make note of these, or just keep the browser tab open for a minute, because we'll use them in a bit.

2. Find your ESPN Authorization Cookies

After you've logged into ESPN fantasy, right click anywhere on the page and click *Inspect*. Then click on the *Storage* tab, go down to *Cookies*, then <https://fantasy.espn.com>. There will be a lot of cookies there, but we're looking for two: `swid` and `espn_s2`. Find and copy them into your `config.ini` file (see the [prerequisites](#) section for more on config.ini).

Mine (with a bunch X'd out so no one uses them to log into my ESPN account and drop all my players) are:

```
[espn]
SWID = XXXXXXXX-0004-4E26-AE4E-XXXXXXXXXXXX
ESPN_S2 = AEBd3f6XXXXXXX...XXXXXXXXY8gJl3rF%2Btr0zcYyK5Fst1Q3tzfP
```

And we're done!

This login and copy cookies to your code workflow is kind of hackish, but you should only have to do it once. If it stops working (i.e. you get a 401 not authorized error) you can log back in and grab new ones.

How that we have these you can go down to the [league setup section](#).

¹Notice that if you try to click this you won't see anything. It's a private league. That's fine. Just do it with your own league.

Yahoo Setup Quickstart

Note: this is meant to get you up and running as quickly as possible, so I keep the explaining to a minimum. For more on what we're doing and why we have to do it see the full [Yahoo league integration](#) section.

Create a Yahoo App

1. Login to your Yahoo Fantasy Football league.
2. Go to <https://developer.yahoo.com/apps/create/> and create an app. Note you need to be logged into the Yahoo account.
3. You'll see this screen:

Create Application

Application Name

Description

Homepage URL

Redirect URI(s)

Please specify any additional redirect uris.

API Permissions

Select private user data APIs that your application needs to access.

Fantasy Sports
 Oath Ad Platforms
 Relationships (Social Directory)
 OpenID Connect Permissions

By clicking Create App, you agree to be bound by the [Yahoo Developer Network Terms of Use](#).

[Create App](#) [Cancel](#)

Figure 0.2: Create Yahoo Application

Put in the following:

Application Name (Required) — I put in `the yahoo fantasy football developer kit app` but you can put in whatever you want.

Redirect URI(s) (Required) — This is required, so it needs to a valid URL, but we won't be using it. Just put in `localhost:8080`

API Permissions (Required) — Check the `Fantasy Sports` box and leave the `Read` option selected.

4. Then click `Create App`. Yahoo will create your app, and redirect you to a page with a `Client ID` and `Client Secret`. Make note of these (or at least don't close the browser), we'll use them in a minute.

Install yahoo_oauth

Now we have our app, but we still need the part where it links up to your Yahoo account and asks for permissions.

We'll outsource all of this to a third party library, `yahoo_oauth`.

So far, Anaconda has come with every Python package we've needed. But to connect to Yahoo fantasy we'll need a third party package. Here's how to get it:

1. Open up Anaconda Navigator.
2. Click on `Environments` on the left side bar.
3. This will bring up a list of your environments. Click on the 'play button' style green triangle in environment you're using (usually there will just be one environment: `base (root)`, but if you know you're using a different one click that).
4. Select `Open Terminal`.

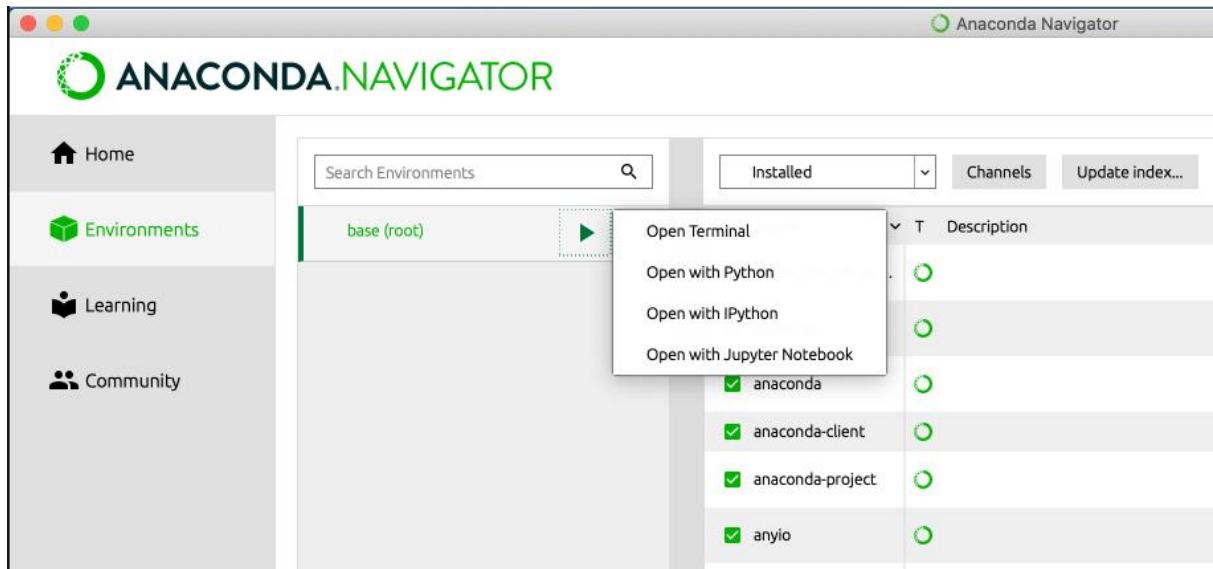


Figure 0.3: Opening Terminal in Anaconda

5. A text console will open up. Type:

```
pip install yahoo_oauth
```

6. Then close it after it installs. Then restart Spyder.

Connecting our app to our Yahoo data

To start, grab your consumer key (aka client id) and secret (aka client secret) from the Yahoo app you made a minute ago and put them in your `config.ini` file:

```
[yahoo]
CONSUMER_KEY =
    dj0yJmk9VTZXXXXXXXXXXXXXXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk
CONSUMER_SECRET = 56ec2XXXXXXXXXX8aa1eff00c8
```

(Note those are my keys, partially X'd out)

Once that's done open up `./code/integration/yahoo_working.py`.

In the REPL, run through line 18. This takes care of some setup.

Now, run this line:

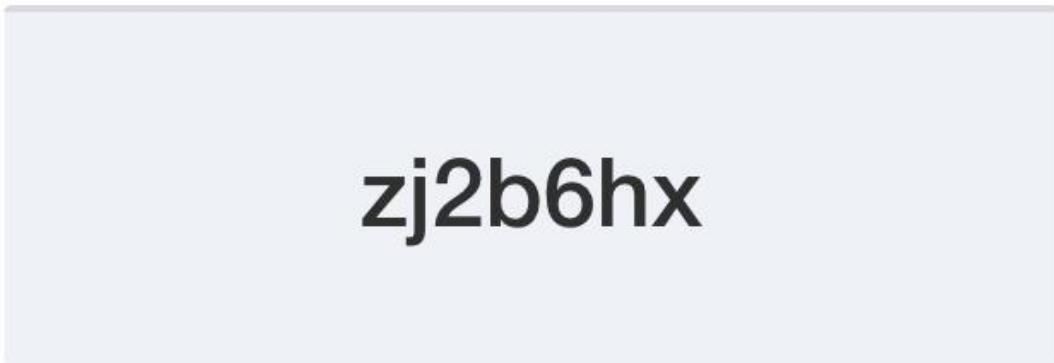
```
In [2]: OAUTH = OAuth2(None, None, from_file=YAHOO_FILE)
[2021-09-07 14:43:55,761 DEBUG] [yahoo_oauth.oauth.__init__] Checking
...
Enter verifier :
```

A browser window will open asking you for permission to connect to your Yahoo account. You'll also see a message pop up in the REPL: `Enter verifier`

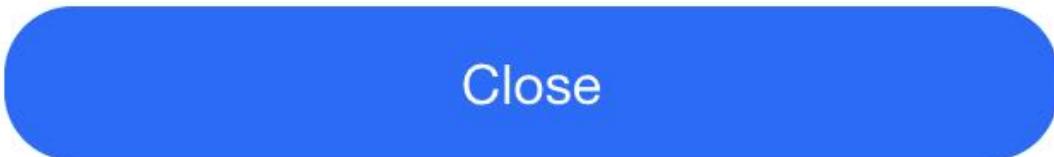
Click `Agree` and it'll take you to a page with a code. Mine is `zj2b6hx`.

Sharing approval

Use this code to connect and share your Yahoo info with fantasy football developer kit integration



zj2b6hx



Close

Figure 0.4: Yahoo Oauth Code

Copy and paste the code into the REPL, which is still waiting with its `Enter verifier` message.

Finally, we're in. You should only have to do it once. Now you can continue with the [league setup section](#).

Setup Continued (All Sites)

Now that you have your site-specific stuff working, find your developer kit Python files, and open up [hosts/league_setup.py](#) in Spyder.

On line 11 you'll see a dictionary of league info. I have one already filled in. Add yours in the same format (feel free to keep mine in if you want, I'll use it in examples later).

Note the first key is a name/tag. We'll use this later so make it something easy to remember and type.

```
LEAGUES = {  
    'nate-league': {  
        'host': 'fleaflicker',  
        'league_id': 34958,  
        'scoring': {'qb': 'pass_6', 'skill': 'ppr_1', 'dst': 'dst_high'},  
        'team_id': 217960},  
    ... # enter your league in same format  
}
```

Once you've entered all your leagues (you can enter as many as you want) run the whole file in the REPL.

Once that's done you're ready to get started with the analysis.

If you ever need to change anything or add a league, no problem. Just update the `LEAGUES` variable in [./hosts/league_setup.py](#) and run this again.

League Analyzer

Once you've done the setup, using the tools is straightforward.

Let's start with the league analyzer. Open up `league.py`. Note some parameters starting on line 16.

```
LEAGUE = 'nate-league'  
WEEK = 2  
WRITE_OUTPUT = False
```

These are straightforward.

- `LEAGUE` is the league you're using (it needs to be in the `league_setup.py` file)
- `WEEK` is the current fantasy week
- `WRITE_OUTPUT` is a boolean indicating whether or not you want to write the output out to a file.
If it's `False`, the code will print the results to the REPL.

Once you've set those you can run the file. It'll give you probabilities for all matchups, as well as projections by team.

Here's mine for this week:

```
*****  
Matchup Projections, Week 1 - 2024  
*****  
  
 wp_a  wp_b  over_under  spread    ml      team_a    team_b  
0  0.50  0.50    246.68    0.0   100  craigsoccer99  nbraun  
1  0.81  0.19    235.50   26.5  -424  steverynear  Brusda  
2  0.68  0.32    231.16   14.5  -213  strausskahn  Ryne  
3  0.50  0.50    247.06    0.0  -100    Deising  sporeilly  
4  0.55  0.45    245.16    4.0  -121    Stefense  komp  
5  0.38  0.62    256.14   -8.0   160  JonHanson  pdizz
```

And:

```
*****
Team Projections, Week 1 - 2024
*****
```

owner_name	mean	std	5%	25%	50%	...	p_high	p_low
pdizz	132.74	23.36	94.22	116.74	132.66	...	0.16	0.03
steverynear	131.41	23.82	93.43	115.64	130.14	...	0.16	0.03
Stefense	124.84	23.09	87.92	109.79	124.13	...	0.09	0.06
JonHanson	124.16	22.51	87.92	107.94	124.33	...	0.08	0.06
strausskahn	123.63	22.25	87.09	108.49	124.15	...	0.08	0.06
Deising	123.54	23.29	85.61	107.65	122.06	...	0.09	0.07
craigsoccer99	123.31	21.82	86.66	108.90	124.21	...	0.07	0.06
nbraun	123.05	23.05	84.68	107.55	123.68	...	0.07	0.08
sporeilly	123.02	21.62	86.24	109.10	122.61	...	0.09	0.06
komp	121.33	22.60	84.86	105.80	120.57	...	0.07	0.08
Ryne	109.29	21.92	74.85	93.75	107.35	...	0.03	0.18
Brusda	104.99	20.49	70.06	91.40	104.79	...	0.02	0.24

I usually just run it like this, then send a screenshot to my league group chat. It always engenders some lively discussions.

Note if you run this mid-week (say after the Thursday night game or before Sunday or Monday night football) it'll take into account player's actual scores and update the projections based on that.

Who Do I Start Analyzer

This one is in `wdis.py`.

Note the parameters starting on line 17. They're the same as the last file:

- `LEAGUE` is the league you're using (it needs to be in the `league_setup.py` file)
- `WEEK` is the current fantasy week
- `WRITE_OUTPUT` is a boolean indicating whether or not you want to write the output out to a file.
If it's `False`, the code will print the results to the REPL.

Once you've set those you can run the file. Under the hood it's: getting your matchup data (your team and your opponents starters), then rotating through your bench options calculating the probability you win for each one. Then it returns the lineup that maximizes your chances.

If you're currently starting someone you shouldn't be, it'll alert you and tell you who to start instead.

Here's mine for this week:

```
Recommended Starters:
```

```
QB: Geno Smith
RB1: Christian McCaffrey
RB2: Tony Pollard
WR1: Calvin Ridley
WR2: Keenan Allen
TE: Juwan Johnson
K: Jake Elliott
D/ST: DAL
```

```
WARNING: Not maximizing probability of winning!
```

```
Start:  
Juwan Johnson
```

```
Instead of:  
Jake Ferguson
```

Another interesting thing you can do is look at the detailed `df_start` DataFrame. This gives you detailed information (average, median, and 5, 25, 75, 95 percentile team scores) along with:

- Your win probability, `wp`.
- The probability this ends up being the `wrong` player and one of your backups scores higher.
- The probability you'll really `regret` your starting decision (you lose by starting this person when you would have won with a backup).

In [1]: df_start.round(2)								
Out[1]:								
pos	name	mean	std	5%	...	wp	wrong	regret
QB	Geno Smith	130.67	23.83	92.25	...	0.69	0.70	0.10
	Deshawn Watson	129.99	23.50	91.68	...	0.67	0.74	0.12
	Kirk Cousins	129.04	23.97	90.68	...	0.65	0.76	0.14
	Mac Jones	127.44	23.61	89.30	...	0.64	0.80	0.14
RB1	Christian McCaffrey	130.67	23.83	92.25	...	0.69	0.11	0.00
	Rico Dowdle	116.05	23.03	78.96	...	0.51	0.95	0.18
	Elijah Mitchell	116.09	22.89	78.31	...	0.50	0.94	0.19
RB2	Tony Pollard	130.67	23.83	92.25	...	0.69	0.18	0.01
	Elijah Mitchell	119.22	22.49	82.79	...	0.55	0.91	0.14
	Rico Dowdle	119.18	22.77	83.08	...	0.54	0.91	0.15
WR1	Calvin Ridley	130.67	23.83	92.25	...	0.69	0.47	0.05
	Amari Cooper	127.57	23.58	86.66	...	0.64	0.64	0.09
	Romeo Doubs	122.08	22.69	84.13	...	0.57	0.89	0.16
WR2	Keenan Allen	130.67	23.83	92.25	...	0.69	0.46	0.04
	Amari Cooper	127.86	23.92	89.82	...	0.65	0.65	0.08
	Romeo Doubs	122.37	22.97	86.00	...	0.59	0.89	0.14
TE	Juwan Johnson	130.67	23.83	92.25	...	0.69	0.48	0.03
	Jake Ferguson	130.33	23.61	93.26	...	0.68	0.52	0.04
K	Jake Elliott	130.67	23.83	92.25	...	0.69	0.00	0.00
D/ST	DAL	130.67	23.83	92.25	...	0.69	0.00	0.00

Notice how there's *always* some probability you pick the wrong player. That's part of fantasy football. When you have a lot of backups — like I do at QB — the probability you're wrong can be pretty high (above 50%). The thing is your probability of being wrong is even higher with all the other guys. That's why you start the guy you do.

Conclusion

So that's the quickstart. I encourage you to get it running, because it's really pretty useful + fun.

In the rest of the guide, we'll walk through building these tools (starting from the basics and going step by step) and more.

6. Project #1: Who Do I Start

Our first project will be using the simulations and API to build our own who do I start tool.

We want to able to give our tool our team, our opponents team and who we're thinking about starting, and have it tell us who maximizes our chances of winning and by how much.

That's the most important part, but it'd be nice if our tool did a few other things, including project our team totals with the various options, make some nice plots, and tell us the probability we're making the *wrong* decision.

WDIS API

A piece of code's *API* describes exactly how it works¹. The functions included and what specifically they take in and return. Let's go over the API for this project.

Note: I'm technically cheating here because I went back and wrote this section *after* writing this project. Normally, you wouldn't have filled in all details yet. That's fine.

In this project, our end result will be two functions each of which take:

- `sims` — a DataFrame of FantasyMath simulations
- `team1` — a regular Python list of fantasymath ids with all the players on team 1 (“your” team, aka the one you’re calculating the wdis options for)
- `team2` — list of fantasymath ids for team 2 (your opponent)
- `wdis` — a list of fantasymath ids for the players you’re trying to decide between

One function (`wdis_plus`) will return some summary stats and numerical analysis — probability of winning with each player in `wdis` etc. The other (`wdis_plot`) will return some data visualizations.

Again — while it’s worth thinking about — you wouldn’t normally have the whole API planned at the outset like this. I’m just mentioning it here so you know how this project works even if you skip building it yourself.

¹Remember, people mean two things when they talk about APIs. Web APIs, where you make a request to a URL and get a response with data (usually JSON) back, and *code APIs*, which basically just means a precise description of the code’s functions and what they take and return. We’re talking about the latter here.

Projects Connect via APIs

Mapping out an API like this also lets us keep our projects separate.

For example, after this who do I start project, our next project is connecting to our league website and getting lineups, rosters, etc.

The code we write to do this will be very different depending on site (ESPN vs Yahoo vs Fleaflicker). We'll walk through all of them, but it's helpful to know that all we need for our WDIS calculator are the raw simulations (which we have), starting lineups, and bench options.

We don't have to worry about how the site specific data grabbing is implemented, just as long as it gets us a list of fantasymath ids. The projects fit together; there are natural boundaries.

It also makes it easier to skip around. Maybe you'd rather start with the league integration project. No problem. You can skip ahead and work through it until you're able to scrape starting lineups and bench options as lists of fantasymath ids. Then you'll know you have everything you'll need for the wdis project.

Building the WDIS Project

Now, finally, we're ready to build something.

A final version you can use with minimal tweaking is in `./wdis_manual.py`. If you're in a hurry to start getting your own who do I start advice, definitely open that up and play around with the parameters (`team1`, `team2`, the scoring settings etc) and run this code as is.

My code uses a real life example I had with my own team a few years ago but we'll talk about subbing in your own team and start-sit decision to make it more interesting.

Otherwise, let's walk through building this tool from scratch. The code for this walk-through is in `./projects/wdis/wdis_working.py`.

Let's start with our imports:

```
import pandas as pd
from os import path
import seaborn as sns
from pandas import Series
from utilities import (generate_token, get_sims, LICENSE_KEY, get_players,
                      OUTPUT_PATH)
```

Reminder: because we're importing our own code in `utilities`, we need to make sure we're working (i.e. by setting Spyder's *working directory* in the top right corner) in the right directory.

The other thing we'll do right away is get an access token.

```
In [1]: token = generate_token(LICENSE_KEY) ['token']

In [2]: token
Out[2]: 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJsaWNlbnNlIjoiODI0Ri1COEIyLTY3REQtNTc0NSIsImV4cCI6MTYyODcwOTM3Nn0.
BgmCZXpJzi4xOSLKsqYhRrFzU5ZoEeLQLajNGzwwrjY'
```

Parameters

When I'm coding something like this, I like to start with a specific example (e.g. week 1, 2021, my team), even if I know I'm ultimately building a tool for more general use (*any* week and team).

A real example is simpler and makes it much easier to get feedback on the code I'm writing, vs an extended, hypothetical example I can only run when the code is finished.

I usually hard code a few parameters too. For example, we know `get_sims` takes number of sims, scoring and (optional) season and week arguments, along with a list of players, so I'll put those at the top of the file (right after my imports).

There's no hard and fast rule, but often I make these parameters uppercase, especially if I view them as constants (e.g. not necessarily changing within one single run of our analysis).

```
In [3]: 

WEEK = 1
SEASON = 2021
NSIMS = 500
SCORING = {'qb': 'pass6', 'skill': 'ppr', 'dst': 'high'}

team1 = ['jalen-hurts', 'clyde-edwards-helaire', 'saquon-barkley',
         'keenan-allen', 'cooper-kupp', 'dallas-goedert', 'jason-myers',
         'tb']
team1_ids = [498, 518, 904, 1954, 1139, 974, 1871, 5180]

team2 = ['justin-jefferson', 'antonio-gibson', 'noah-fant',
         'christian-mccaffrey', 'tyler-lockett', 'matthew-stafford',
         'matt-gay', 'gb']
team2_ids = [551, 567, 755, 1105, 1542, 2621, 776, 5162]
```

Note: these teams and opponents were a real life example I had from 2021. The code will work with these values (and you'll see what I see here in the book when running it), but you're welcome to fill in your own team (`team1`) and opponent (`team2`) and bench options for a start-sit decision too.

The only real caveat is you need to make sure you're working with valid Fantasy Math player ids.

You can find these using the `get_players` function.

For example (I printed off the first 20 players here):

```
In [4]: valid_players = get_players(token, season=SEASON, week=WEEK, **  
SCORING)  
  
In [5]: (valid_players[['name', 'player_id']])[:20]  
Out[5]:  
      name  player_id  
0    Trevor Lawrence      327  
1      Zach Wilson       329  
2      Mac Jones        331  
3     Najee Harris       344  
4   Kenneth Gainwell      345  
5  Javonte Williams      346  
6    Michael Carter       348  
7  Rhamondre Stevenson      349  
8     Jaret Patterson      350  
9     Chuba Hubbard       351  
10    Elijah Mitchell      353  
11     Kylin Hill        355  
12    Mekhi Sargent       359  
13    Larry Rountree       362  
14     Ja Marr Chase       371  
15    DeVonta Smith       372  
16    Rondale Moore       373  
17     Jaylen Waddle       374  
18  Amon-Ra St. Brown      377  
19     Elijah Moore        378
```

To start these are hard coded (i.e. I went through my `valid_players` table and found the ids of the guys I wanted) but later we'll see how to get these ids automatically from our league site.

Also, this is live data from the Fantasy Math simulation API. I anticipate the API working when you read this. But you might want to work through these projects in the offseason, or in the future after your annual access has expired. In any case, so you can 100% see exactly what's shown here, I've saved all the data locally.

Let's make a boolean constant that keeps track of what we want to do:

```
In [6]: USE_SAVED_DATA = True
```

Now we can change our `valid_players` call to this:

```
In [7]:  
if USE_SAVED_DATA:  
    valid_players = pd.read_csv(path.join('projects', 'wdis', 'data',  
                                         'valid_players.csv'))  
                                         .set_index('player_id')  
else:  
    valid_players = get_players(token, season=SEASON, week=WEEK,  
                                **SCORING).set_index('player_id')
```

So if `USE_SAVED_DATA` is `True`, we'll load the snapshot I've saved. Otherwise we'll hit the API.

Next we need to get the simulations for those players from the API. Besides the access token and scoring rules, the `get_sims` function takes a list of players we want simulations for. We'll limit it to our lineup + bench options.

```
In [8]: players = team1 + team2 + bench
```

I've saved a snapshot of this data too. Note if you want to walk through this using your own start-sit decision (with your own player ids), you'll need to set `USE_SAVED_DATA = False`.

```
In [9]:  
if USE_SAVED_DATA:  
    sims = pd.read_csv(path.join('projects', 'wdis', 'data', 'sims.csv'))  
    sims.columns = [int(x) for x in sims.columns]  
else:  
    sims = get_sims(token, players, week=WEEK, season=SEASON, nsims=NSIMS,  
                    **SCORING)
```

Here's what that data looks like:

```
In [10]: sims  
Out[10]:  
      498        518        904     ...        703        705        906  
0  27.122777  7.557770  10.731900  ...  4.834787  13.669192  7.456091  
1  36.821143  17.371515  11.662721  ...  3.357973  8.792438  1.726840  
2  16.259885  22.811825  8.673472  ...  5.689259  15.939823  22.064444  
3  28.420762  9.006972  21.791620  ...  13.090076  22.503080  5.701639  
4  25.108344  28.735289  28.975998  ...  4.877228  0.174620  25.152749
```

Recall how each Pandas DataFrame includes an index. Here it's just the default, which is 0-499. In this case we can think of it as simulation ID. So the sims in the first row (`sim ID == 0`) are correlated with each other.

For now, let's make a version with names instead of ids to make it easier to follow what's going on. Remember I've provided this `name_sims` function in `utilities.py`.

```
In [11]: nsims = name_sims(sims, valid_players)

In [12]: nsims.head()
Out[12]:
      jalen-hurts  clyde-helaire  ...  darrell-henderson  ronald-jones
0    27.122777      7.557770  ...          13.669192      7.456091
1    36.821143      17.371515  ...          8.792438      1.726840
2    16.259885      22.811825  ...         15.939823     22.064444
3    28.420762      9.006972  ...         22.503080      5.701639
4    25.108344      28.735289  ...          0.174620     25.152749
```

Coding Up a Who Do I Start Calculator

Let's look at my RB2 spot, where in Week 1, 2021 I had to decide between Clyde Edwards-Helaire, Darrell Henderson, Ronald Jones and Tony Pollard.

Let's write some code to figure out the probability of winning with each of those players. We'll start simple, then extend it to make it more flexible.

Remember in Pandas you select subsets of columns by passing a list to DataFrame, so if we want to look at just the sims for my team:

```
In [13]: nsims[team1].head()
Out[13]:
   jalen-hurts  clyde-edwards-helaire ... dallas-goedert      tb
0    27.122777           7.557770 ...  15.281437  3.173092
1    36.821143          17.371515 ...  21.290151  4.013744
2    16.259885          22.811825 ...  12.530967  9.251356
3    28.420762           9.006972 ...  9.043176 17.572380
4    25.108344          28.735289 ...  7.610788  6.898449
```

To get total team score, we need to add all these together, which we do with the `sum` function.

Remember, by default most Pandas function operate on the columns. So:

```
In [14]: nsims[team1].sum()
Out[14]:
jalen-hurts            12205.926956
clyde-edwards-helaire  8285.814094
saquon-barkley         6895.262466
keenan-allen           8553.888021
cooper-kupp             7648.181083
dallas-goedert          4808.567960
jason-myers            4251.672991
tb                      6911.850096
```

sums up player point totals over all the simulations. That's not useful. What we want is to sum things up by *row* so that we have total Team 1 score for simulation 0, another for sim 1, etc. We get this by passing `axis=1` to sum.

```
In [15]: nsims[team1].sum(axis=1).head()
Out[15]:
0      93.641759
1     139.811889
2      97.781934
3     113.214746
4     134.904767
dtype: float64
```

So in the first simulation, my team scored 93.64 points. In the second, 139.81. These are the first 5 values out of 500 simulations.

Here's Team 2:

```
In [16]: nsims[team2].sum(axis=1).head()
Out[16]:
0    158.991433
1    86.806277
2    108.607376
3    105.672497
4    115.706523
```

This is nice, but what we *really* want is to see who scored more and see who beats who and how often. We're looking for a column of booleans:

```
In [17]: team1_beats_team2 = (nsims[team1].sum(axis=1) >
                           nsims[team2].sum(axis=1))

In [18]: team1_beats_team2.head()
Out[18]:
0    False
1     True
2    False
3     True
4     True
dtype: bool
```

Then we can take the average of this boolean column to see how often we beat our opponent (i.e. how often `team1_beats_team2` is `True`).

```
In [19]: print(team1_beats_team2.mean())
0.422
```

Dang. Not looking good for Team 1.

But that's our probability of winning given our current lineups. Now let's build a simple function that lets us repeat this calculation given our team, our opponent's team, a guy we might want to start, and the sims.

```
In [20]:
def simple_wdis(sims, team1, team2, wdis):
    team1_wdis = team1 + [wdis]
    return (sims[team1_wdis].sum(axis=1) > sims[team2].sum(axis=1)).mean()
```

We can call this a bunch of times using a loop:

```
In [21]:  
team1_no_wdis = ['jalen-hurts', 'saquon-barkley', 'keenan-allen',  
                 'cooper-kupp', 'dallas-goedert', 'jason-myers', 'tb']  
  
wdis = ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones',  
        'tony-pollard']  
  
for player in wdis:  
    print(player)  
    print(simple_wdis(nsims, team1_no_wdis, team2, player))  
--  
clyde-edwards-helaire  
0.422  
darrell-henderson  
0.346  
ronald-jones  
0.314  
tony-pollard  
0.274
```

This definitely works (there's a glimmer of light shining through this rock wall we're tunneling through), but we can make it better.

For starters, it's annoying to have to call this function four times, why can't we just pass in a list of our WDIS candidates and get all the probabilities at once?

Let's update it (note the dictionary comprehension — usually anything you can do with loops you can with a comprehension — in the return statement):

```
In [22]:  
def simple_wdis2(sims, team1, team2, wdis):  
    return {  
        player: float((sims[team1 + [player]].sum(axis=1) >  
                      sims[team2].sum(axis=1)).mean())  
        for player in wdis}
```

And calling it:

```
In [23]: simple_wdis2(nsims, team1_no_wdis, team2, wdis)  
Out[23]:  
{'clyde-edwards-helaire': np.float64(0.422),  
 'darrell-henderson': np.float64(0.346),  
 'ronald-jones': np.float64(0.314),  
 'tony-pollard': np.float64(0.274)}
```

That's better. But here's another nitpick: I don't necessarily like having to make sure all my WDIS guys aren't included in `team1`.

It makes it harder to quickly try out new positions. For example, if — after checking my RB2 spot — I want to run through all the FA kickers, I have to edit `team1` to add a RB, and then remove Jason Myers, etc.

(Aside: note what we're doing here, thinking about how we'll use our code, and noticing the things that might be annoying about it. This is a good guide in terms of which direction to move.)

Let's modify it again so that it takes a *full* `team1`, a list of `wdis` players and automatically figures out who you're asking about.

```
In [24]:  
def simple_wdis3(sims, team1, team2, wdis):  
  
    # there should be one player that overlaps in wdis and team1  
    team1_no_wdis = [x for x in team1 if x not in wdis]  
  
    # another way to do this is using python sets  
    # team1_no_wdis_alt = set(team1) - set(wdis)  
  
    return {  
        player: float((sims[team1_no_wdis + [player]].sum(axis=1) >  
                      sims[team2].sum(axis=1)).mean()) for player in wdis}
```

Again, running it:

```
In [25]: simple_wdis3(nsims, team1, team2, wdis)  
Out[25]:  
{'clyde-edwards-helaire': 0.422,  
 'darrell-henderson': 0.346,  
 'ronald-jones': 0.314,  
 'tony-pollard': 0.274}  
  
In [26]: simple_wdis3(nsims, team1, team2,  
                      ['saquon-barkley', 'darrell-henderson'])  
Out[26]: {'saquon-barkley': 0.422, 'darrell-henderson': 0.376}
```

Note that our function is making some assumptions though.

We're assuming there's exactly one player in common between `team1` and `wdis`. We're also assuming `wdis` contains at least one player.

If we've made a mistake in either this will return an error, or worse, no error but an incorrect result.

It's probably good idea to build in some checks — especially if anyone besides us will be using this code — but even for our future self if we go away and pick it up later.

For your own internal use, simple `assert` statements are fine. These throw an error anytime some condition (a boolean) isn't met.

```
In [27]: assert False
```

```
AssertionError:
```

If this is a function that other people (or customers) will be using maybe you want it to have a more informative error message.

While we're adding the checks via `assert` let's also wrap our dict in a Pandas Series which lets us do things like sort by probability of winning.

Again, this is another example of going from simple → more complex. A dict (which is part of the standard library) is simpler than a Series.

```
In [28]:
```

```
def simple_wdis4(sims, team1, team2, wdis):  
  
    # there should be one player that overlaps in wdis and team1  
    team1_no_wdis = [x for x in team1 if x not in wdis]  
  
    # some checks  
    current_starter = [x for x in team1 if x in wdis]  
    assert len(current_starter) == 1  
  
    bench_options = [x for x in wdis if x not in team1]  
    assert len(bench_options) >= 1  
  
    return Series({  
        player: float((sims[team1_no_wdis + [player]].sum(axis=1) >  
                      sims[team2].sum(axis=1)).mean()) for player in wdis}  
    ).sort_values(ascending=False)
```

So we have:

```
In [29]: simple_wdis4(nsims, team1, team2, wdis)
```

```
Out[29]:
```

clyde-edwards-helaire	0.422
darrell-henderson	0.346
ronald-jones	0.314
tony-pollard	0.274

Again, we're assuming exactly one player in common between `team1` and `wdis`. So far player has been `clyde-edwards-helaire`. Let's try leaving him out and breaking our function:

```
In [30]:
```

```
simple_wdis4(nsims, team1, team2, ['darrell-henderson', 'ronald-jones',  
                                    'tony-pollard'])
```

```
AssertionError:
```

Looks like it's working. I'm satisfied with the state of our `simple_wdis` function for now.

Again, I'm not just walking through this slowly in order to make things clearer or as a teaching tool — this iterative, build simple then improve process is actually how I would go about writing a function like this for my own use.

Beyond WDIS

Here's where we're at with our WDIS analysis.

```
team1 = ['jalen-hurts', 'clyde-edwards-helaire', 'saquon-barkley',
         'keenan-allen', 'cooper-kupp', 'dallas-goedert', 'jason-myers',
         'tb']

team2 = ['justin-jefferson', 'antonio-gibson', 'noah-fant',
         'christian-mccaffrey', 'tyler-lockett', 'matthew-stafford',
         'matt-gay', 'gb']

current_starter = 'clyde-edwards-helaire'
bench_options = ['darrell-henderson', 'ronald-jones', 'tony-pollard']
team1_sans_starter = list(set(team1) - set([current_starter]))
```

Let's see if there's any other cool things working with these simulations might tell us.

For one, it'd be nice to see some predictions about our overall team score.

```
In [1]: nsims[team1].sum(axis=1).describe()
Out[1]:
count    500.000000
mean     119.122327
std      23.194248
min      57.623535
25%     102.502233
50%     118.425215
75%     135.305736
max     193.410037
```

This is our projected score starting Edwards-Helaire. It's also nice to see summary stats for *all* our guys in table form, i.e. "stick" all these columns together side by side. (Remember, sticking columns or rows together calls for `pd.concat()`.)

```
In [2]:  
stats = pd.concat([(nsims[team1_sans_starter].sum(axis=1) + sims[x]).  
    describe()  
        for x in wdis], axis=1)  
stats.columns = wdis # make column names = players  
  
In [3]: stats  
Out[3]:  
      clyde-helaire  darrell-henderson  ronald-jones  tony-pollard  
count    500.000000          500.000000          500.000000          500.000000  
mean     119.122327         113.617518         112.235841         108.838242  
std      23.194248          22.822798          23.052998          22.400854  
min      57.623535          57.531070          46.055698          48.543564  
25%     102.502233          97.613458          97.287324          94.187508  
50%     118.425215          112.063858          111.223416          107.759402  
75%     135.305736          127.908247          126.599510          123.085188  
max     193.410037          191.988978          189.323107          181.053567
```

It's personal preference, but I find it easier to read when rows are players instead of columns.

Let's transpose (flip the rows and columns around, we can do it with `.T`) it. We also don't really need count, min or max.

```
In [4]: stats.T.drop(['count', 'min', 'max'], axis=1) # drop unnec  
        columns  
Out[4]:  
      mean      std ...      50%      75%  
clyde-edwards-helaire  119.122327  23.194248 ...  118.425215  135.305736  
darrell-henderson     113.617518  22.822798 ...  112.063858  127.908247  
ronald-jones          112.235841  23.052998 ...  111.223416  126.599510  
tony-pollard           108.838242  22.400854 ...  107.759402  123.085188
```

We know CEH is *most likely* to score the highest, but that definitely doesn't mean he always will.

Let's calculate the probability one of our backups scores more than him. We'll call it the probability of starting the wrong guy.

Doing this is easy: first, we'll figure out the highest simulation from our bench guys:

```
In [5]: nsims[bench_options].max(axis=1).head()  
Out[5]:  
0    13.669192  
1    8.792438  
2    22.064444  
3    22.503080  
4    25.152749
```

And see how often that best-bench guy scores more than CEH:

```
In [6]:  
pd.concat([nsims[bench_options].max(axis=1),  
          nsims[current_starter]], axis=1).head()  
Out[6]:  
          0  clyde-edwards-helaire  
0  13.669192           7.557770  
1  8.792438            17.371515  
2  22.064444           22.811825  
3  22.503080           9.006972  
4  25.152749           28.735289
```

It's another boolean column:

```
In [7]: (nsims[bench_options].max(axis=1) > nsims[current_starter]).mean()  
Out[7]: 0.398
```

So even though we should start CEH, about 40% of the time one of our bench guys will outscore him. This is something most fantasy owners might beat themselves up over, but we know better. That's the benefit to taking a probabilistic approach to fantasy football.

Now, for (the opposite of?) fun, let's calculate how often we'll *really* regret our decision. That is, **how often will we lose because we started CEH instead of one of his backups?** Understanding going in that there's a possibility of this happening, and knowing what that possibility is might help us come to terms with if it does happen.

With Pandas, this is just a matter of some simple math functions:

```
In [8]:  
team1_w_starter = nsims[team1_sans_starter].sum(axis=1) + nsims[  
                      current_starter]  
  
team1_w_best_backup = (nsims[team1_sans_starter].sum(axis=1) +  
                       nsims[bench_options].max(axis=1))  
  
team2_total = nsims[team2].sum(axis=1)
```

The actual calculation is some logical, boolean arithmetic. How often do we win with our best backup AND lose with our starter:

```
In [9]:  
regret_col = ((team1_w_best_backup > team2_total) &  
              (team1_w_starter < team2_total))  
--  
In [10]: regret_col.mean()  
Out[10]: 0.036
```

So starting CEH will lose us our matchup less than 4% of the time. Let's do the opposite — how often

will CEH *win* our matchup?

```
In [11]:  
nailed_it_col = ((team1_w_best_backup < team2_total) &  
                  (team1_w_starter > team2_total))  
  
In [12]: print(nailed_it_col.mean())  
0.076
```

Interesting.

Now let's put these into some functions so we can see these probabilities assuming we start different guys.

```
In [11]:  
def sumstats(starter):  
    team_w_starter = (nsims[team1_sans_starter].sum(axis=1) +  
                      nsims[starter])  
    stats_series = (team_w_starter  
                    .describe(percentiles=[.05, .25, .5, .75, .95])  
                    .drop(['count', 'min', 'max']))  
    stats_series.name = starter  
    return stats_series  
  
def win_prob(starter):  
    team_w_starter = nsims[team1_sans_starter].sum(axis=1) + nsims[starter]  
    return (team_w_starter > team2_total).mean()  
  
def wrong_prob(starter, bench):  
    return (nsims[bench].max(axis=1) > nsims[starter]).mean()  
  
def regret_prob(starter, bench):  
    team_w_starter = nsims[team1_sans_starter].sum(axis=1) + nsims[starter]  
    team_w_best_backup = (nsims[team1_sans_starter].sum(axis=1) +  
                          nsims[bench].max(axis=1))  
  
    return ((team_w_best_backup > team2_total) &  
            (team_w_starter < team2_total)).mean()  
  
def nailed_it_prob(starter, bench):  
    team_w_starter = nsims[team1_sans_starter].sum(axis=1) + nsims[starter]  
    team_w_best_backup = (nsims[team1_sans_starter].sum(axis=1) +  
                          nsims[bench].max(axis=1))  
  
    return ((team_w_best_backup < team2_total) &  
            (team_w_starter > team2_total)).mean()
```

Note: these functions (sumstats, win_prob, etc) are using variables and data we've defined above (`nsims`, `team1`, `team1_sans_starter` etc) even though we're not passing them as arguments.

This is an intermediate move, and is allowed as long as these functions are evaluated AFTER the variables we're using (`nsims`, `team1`) have already been defined. We defined + ran these earlier in the REPL so we're fine.

Now it's easy to see these probabilities for our other starter and bench options.

```
# current_starter = 'clyde-edwards-helaire'

In [12]: print(sumstats('clyde-edwards-helaire'))
mean    119.122327
std     23.194248
5%      82.749104
25%     102.502233
50%     118.425215
75%     135.305736
95%     159.268768

In [13]: print(win_prob(current_starter))
0.422

In [14]: print(wrong_prob(current_starter, bench_options))
0.398

In [15]: print(regret_prob(current_starter, bench_options))
0.036

In [16]: print(nailed_it_prob(current_starter, bench_options))
0.076
~~~

That's with our current start, Clyde Edwards-Helaire. Now let's look at
one of
our backups:

~~~ {.python}
# now with next best alternative, henderson
In [17]: print(sumstats('darrell-henderson'))
mean    113.617518
std     22.822798
5%      77.324589
25%     97.613458
50%     112.063858
75%     127.908247
95%     152.734038

In [18]: print(win_prob('darrell-henderson'))
0.346

In [19]:
print(wrong_prob('darrell-henderson',
                  ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard']))
--
0.812

In [20]:
print(regret_prob('darrell-henderson',
                  ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard']))
--
0.112

In [21]:
print(nailed_it_prob('darrell-henderson',
                  ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard']))
--
```

So, while we're going to really regret starting CEH 3.6% of the time, starting Henderson will cost us the game even more — 11% of the time.

The correct conclusion: in fantasy, regret is unavoidable. All you can look at is your process from week to week.

It'd be nice to fold in these interesting side functions into our main WDIS analysis function, so we can submit teams, wdis and get back all these stats and probabilities for everyone.

Looking at these functions (`sumstats`, `win_prob`, `wrong_prob`, `regret_prob`, `nailed_it_prob`) they all just take starter and (sometimes) bench guys.

We can work with this, but it'd be nice to have some code that — given a list of WDIS candidates — goes through and automatically makes all the permutations of starter and bench guys.

Let's code that up:

```
In [20]:  
def start_bench_scenarios(wdis):  
    """  
    Return all combinations of start, backups for all players in wdis.  
    """  
    return [{  
        'starter': player,  
        'bench': [x for x in wdis if x != player]  
    } for player in wdis]
```

So we have:

```
In [21]: wdis  
Out[21]: ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones',  
         'tony-pollard']  
  
In [22]: start_bench_scenarios(wdis)  
Out[22]: [{  
    'starter': 'clyde-edwards-helaire',  
    'bench': ['darrell-henderson', 'ronald-jones', 'tony-pollard']},  
    {'starter': 'darrell-henderson',  
    'bench': ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard']},  
    {'starter': 'ronald-jones',  
    'bench': ['clyde-edwards-helaire', 'darrell-henderson', 'tony-pollard']},  
    {'starter': 'tony-pollard',  
    'bench': ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones']}
```

Note: I immediately wrote it inside a function, because I knew what I wanted and how to go about doing it, but if you're unsure there's no harm at all in starting even simpler.

Let's do that.

Remember we're working with `wdis`:

```
In [23]: wdis
Out[23]: ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones',
          'tony-pollard']
```

And want a starter and bench options for each player. Always useful to start with a specific example:

```
In [24]: player = 'clyde-edwards-helaire'

In [25]: [x for x in wdis if x != player]
Out[25]: ['darrell-henderson', 'ronald-jones', 'tony-pollard']
```

And we want that in a dict:

```
In [26]: {'starter': player, 'bench': [x for x in wdis if x != player]}
Out[26]:
{'starter': 'clyde-edwards-helaire',
 'bench': ['darrell-henderson', 'ronald-jones', 'tony-pollard']}
```

It's doing what we want with CEH, so now do it for every player in a comprehension, and we'll be all set.

```
In [27]:
[{'starter': player, 'bench': [x for x in wdis if x != player] }
 for player in wdis]
-- 
Out[27]:
[{'starter': 'clyde-edwards-helaire',
 'bench': ['darrell-henderson', 'ronald-jones', 'tony-pollard']},
 {'starter': 'darrell-henderson',
 'bench': ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard']},
 {'starter': 'ronald-jones',
 'bench': ['clyde-edwards-helaire', 'darrell-henderson', 'tony-pollard']},
 ],
 {'starter': 'tony-pollard',
 'bench': ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones']}
```

Now we can call `start_bench_scenarios`, then use list of dictionaries we get back to analyze the probabilities of winning, being wrong and regretting our decision across all the scenarios.

We can also think a bit about presentation. I liked that table of stats (team mean, std deviation, percentiles) for each player.

Let's start with that, then maybe add in our other columns.

```
In [28]: df = pd.concat([sumstats(player) for player in wdis], axis=1)
```

```
In [29]: df = df.T
```

```
In [30]: df.head()
```

```
Out[30]:
```

	mean	std	...	75%	95%
clyde-edwards-helaire	119.122327	23.194248	...	135.305736	159.268768
darrell-henderson	113.617518	22.822798	...	127.908247	152.734038
ronald-jones	112.235841	23.052998	...	126.599510	151.775863
tony-pollard	108.838242	22.400854	...	123.085188	145.881436

Once we have that, it's not hard go through the other functions and add them as columns to the data.

```
In [31]:
```

```
wps = [win_prob(player) for player in wdis]
```

```
df['wp'] = wps
```

```
In [32]: df.head()
```

```
Out[32]:
```

	mean	std	...	95%	wp
clyde-edwards-helaire	119.122327	23.194248	...	159.268768	0.422
darrell-henderson	113.617518	22.822798	...	152.734038	0.346
ronald-jones	112.235841	23.052998	...	151.775863	0.314
tony-pollard	108.838242	22.400854	...	145.881436	0.274

For this next one I'll skip the extra step and just putting it immediately in the DataFrame. I'm also using our `start_bench_scenarios` function.

```
In [33]: df['wrong'] = [wrong_prob(scen['starter'], scen['bench'])  
                      for scen in scenarios]
```

```
In [34]: df.head()
```

```
Out[34]:
```

	mean	std	...	wp	wrong
clyde-edwards-helaire	119.122327	23.194248	...	0.422	0.398
darrell-henderson	113.617518	22.822798	...	0.346	0.812
ronald-jones	112.235841	23.052998	...	0.314	0.846
tony-pollard	108.838242	22.400854	...	0.274	0.944

Finally, regret and nailed it:

```
In [35]:  
df['regret'] = [regret_prob(**scen) for scen in scenarios]
```

```
In [36]:
```

```
df['nailed'] = [nailed_it_prob(**scen) for scen in scenarios]
```

(Remember, putting `**` in front of a dictionary let's you turn it into keyword arguments, so passing `**scen` is the same as passing `scen['starter']`, `scen['bench']` above².)

Our final results:

```
In [37]: df.round(2)
Out[37]:
          mean    std     5%    25% ...   regret   nailed
clyde-edwards-helaire  119.12  23.19  82.75 102.50 ...    0.04    0.08
darrell-henderson      113.62  22.82  77.32  97.61 ...    0.11    0.02
ronald-jones            112.24  23.05  77.98  97.29 ...    0.14    0.01
tony-pollard             108.84 22.40  71.51  94.19 ...    0.18    0.00
```

Usually, when I'm working on code like this, I like to do what we've done above, keeping my code in the main, top level Python program (i.e. not inside a function).

When I'm satisfied with it — when we have this final `df` that looks like what we want — I'll move everything inside a function.

Let's do that now.

²Note this only works because `regret_prob` takes arguments named `starter` and `bench`, if these were named something else we'd get an error.

```
In [37]:
def wdis_plus(sims, team1, team2, wdis):

    # do some validity checks
    current_starter = set(team1) & set(wdis)
    assert len(current_starter) == 1

    bench_options = set(wdis) - set(team1)
    assert len(bench_options) >= 1

    team_sans_starter = list(set(team1) - current_starter)

    scenarios = start_bench_scenarios(wdis)
    team2_total = sims[team2].sum(axis=1) # opp

    # note these functions all work with sims, even though they don't take
    # sims as an argument
    # it works because everything inside wdis_plus has access to sims
    # if these functions were defined outside of wdis_plus it wouldn't
    # work
    # this is an example of lexical scope: https://stackoverflow.com/a
    # /53062093
    def sumstats(starter):
        team_w_starter = sims[team_sans_starter].sum(axis=1) + sims[
            starter]
        team_info = (team_w_starter
                     .describe(percentiles=[.05, .25, .5, .75, .95])
                     .drop(['count', 'min', 'max']))

        return team_info

    def win_prob(starter):
        team_w_starter = sims[team_sans_starter].sum(axis=1) + sims[
            starter]
        return (team_w_starter > team2_total).mean()

    def wrong_prob(starter, bench):
        return (sims[bench].max(axis=1) > sims[starter]).mean()

    def regret_prob(starter, bench):
        team_w_starter = sims[team_sans_starter].sum(axis=1) + sims[
            starter]
        team_w_best_backup = (sims[team_sans_starter].sum(axis=1) +
                              sims[bench].max(axis=1))

        return ((team_w_best_backup > team2_total) &
                (team_w_starter < team2_total)).mean()

    def nailed_it_prob(starter, bench):
        team_w_starter = sims[team1_sans_starter].sum(axis=1) + sims[
            starter]
        team_w_best_backup = (sims[team1_sans_starter].sum(axis=1) +
                              sims[bench].max(axis=1))

        return ((team_w_best_backup < team2_total) &
                (team_w_starter > team2_total)).mean()

v0.19.1      return ((team_w_best_backup < team2_total) &
                    (team_w_starter > team2_total)).mean()
```

And calling it:

```
In [38]: wdis_plus(nsims, team1, team2, wdis)
Out[38]:
      mean      std    ... wrong  regret  nailed
clyde-edwards-helaire  119.122327  23.194248  ...  0.398   0.036   0.076
darrell-henderson     113.617518  22.822798  ...  0.812   0.112   0.020
ronald-jones          112.235841  23.052998  ...  0.846   0.144   0.012
tony-pollard           108.838242  22.400854  ...  0.944   0.184   0.002
```

Now we have a reusable function we can use for any similar start decision.

For example, one thing I like to do is run all the available FA kickers and defenses through the model.

This help helpful because:

1. Kickers are sort of a crap shoot (their projected distributions overlap a ton and the difference between “good” and “bad” fantasy kickers is small).
2. Kickers are relatively highly correlated with other players on their team and the DST they’re going against.

So often there’s someone under the radar that might let me squeeze out a bit of win probability. It’s like matchup-specific streaming.

Let’s do that (note we have to hit the API again for all the kicker simulations since we didn’t get them originally).

Here were the FA kickers available to me this week:

```
In [1]:
fa_kickers = ['jake-elliott', 'tristan-vizcaino', 'josh-lambo', 'greg-joseph',
              'evan-mcpherson', 'chase-mclaughlin', 'ryan-santoso',
              'aldrick-rosas', 'jason-sanders', 'daniel-carlson',
              'rodrigo-blankenship', 'brandon-mcmanus', 'joey-slye',
              'graham-gano', 'nick-folk', 'cairo-santos']

fa_kicker_ids = [1188, 869, 1635, 1035, 435, 872, 1079, 1477, 1003, 981,
                  592,
                  1839, 864, 2713, 2939, 1837]

if USE_SAVED_DATA:
    k_sims = pd.read_csv(path.join('projects', 'wdis', 'data', 'k_sims.csv'))
    k_sims.columns = [int(x) for x in k_sims.columns]
else:
    k_sims = get_sims(token, fa_kicker_ids, week=WEEK, season=SEASON,
                      nsims=1000, **SCORING)

nk_sims = name_sims(k_sims, valid_players)

nsims_plus = pd.concat([nsims, nk_sims], axis=1)

wdis_k = fa_kickers + ['jason-myers']
```

And running our function:

```
In [2]: df_k = wdis_plus(nsims_plus, team1, team2, wdis_k)

In [3]: df_k
Out[3]:
```

	mean	std	...	wrong	regret	nailed
jason-myers	119.122327	23.194248	...	0.934	0.106	0.004
rodrigo-blankenship	118.187120	23.428079	...	0.952	0.112	0.000
greg-joseph	118.701181	23.742779	...	0.930	0.120	0.002
jason-sanders	118.537626	23.139394	...	0.944	0.120	0.002
aldrick-rosas	118.222963	22.783675	...	0.950	0.122	0.004
cairo-santos	118.539435	23.245674	...	0.922	0.124	0.002
chase-mclaughlin	118.338669	23.347040	...	0.934	0.126	0.002
ryan-santoso	118.874974	23.394266	...	0.910	0.128	0.004
daniel-carlson	118.867519	23.334523	...	0.936	0.128	0.000
evan-mcpherson	117.699540	23.162740	...	0.956	0.130	0.002
joey-slye	118.046092	23.437308	...	0.938	0.130	0.006
nick-folk	118.349606	23.141877	...	0.946	0.130	0.000
tristan-vizcaino	118.121511	23.672072	...	0.942	0.136	0.002
josh-lambo	118.595436	22.808140	...	0.946	0.136	0.004
brandon-mcmanus	118.069199	23.094033	...	0.948	0.140	0.000
graham-gano	118.025492	23.087204	...	0.960	0.140	0.000
jake-elliott	118.404608	23.479945	...	0.952	0.142	0.002

There we go. We can see Myers maximizes my chances of winning by .006 over the next guy. Every edge counts.

Plotting

Now let's do some plotting.

In Learn to Code with Fantasy Football we learned how the Python library seaborn makes plots very easy, provided the data is in the right format.

Let's start by summing up our totals together.

```
In [1]:  
points_wide = pd.concat([  
    sims[team1].sum(axis=1),  
    sims[team2].sum(axis=1)  
, axis=1)  
  
points_wide.columns = ['team1', 'team2']
```

```
In [2]: points_wide.head()  
Out[2]:  
      team1      team2  
0  93.641759  158.991433  
1  139.811889   86.806277  
2   97.781934  108.607376  
3  113.214746  105.672497  
4  134.904767  115.706523
```

This data is “wide”, which means we have our point totals in two separate columns. To work with seaborn, we need it to be long, like this:

```
      sim  team    points  
0     0  team1  93.641759  
1     0  team2  158.991433  
2     1  team1  139.811889  
3     1  team2   86.806277  
4     2  team1   97.781934
```

The easiest way to do this is to *stack* the data. That moves information from the columns (team1 and team2) to rows. Like this:

```
In [3]: points_wide.stack().head()
Out[3]:
0  team1      93.641759
   team2      158.991433
1  team1      139.811889
   team2      86.806277
2  team1      97.781934
```

Both `stack` and it's inverse `unstack` work with DataFrame indices, which means the team info is part of the index. To treat it like a normal column we can call `reset_index` after stacking the data.

```
In [4]: points_long = points_wide.stack().reset_index()
In [4]: points_long.columns = ['sim', 'team', 'points']

In [5]: points_long.head()
Out[5]:
   sim    team    points
0     0  team1  93.641759
1     0  team2  158.991433
2     1  team1  139.811889
3     1  team2  86.806277
4     2  team1  97.781934
```

The stack → reset_index → rename columns pattern a lot in this book, so it's worth reading the above section again (it's also covered in Appendix D) if it doesn't make sense.

Now we have what we need for seaborn.

Remember, despite my `best efforts`, it takes two lines to create density plots, like this:

```
In [6]:
g = sns.FacetGrid(points_long, hue='team', aspect=4)
g = g.map(sns.kdeplot, 'points', fill=True)
```

Then a few more to add a legend, title etc:

```
In [7]:
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Total Team Fantasy Points Distributions')
```

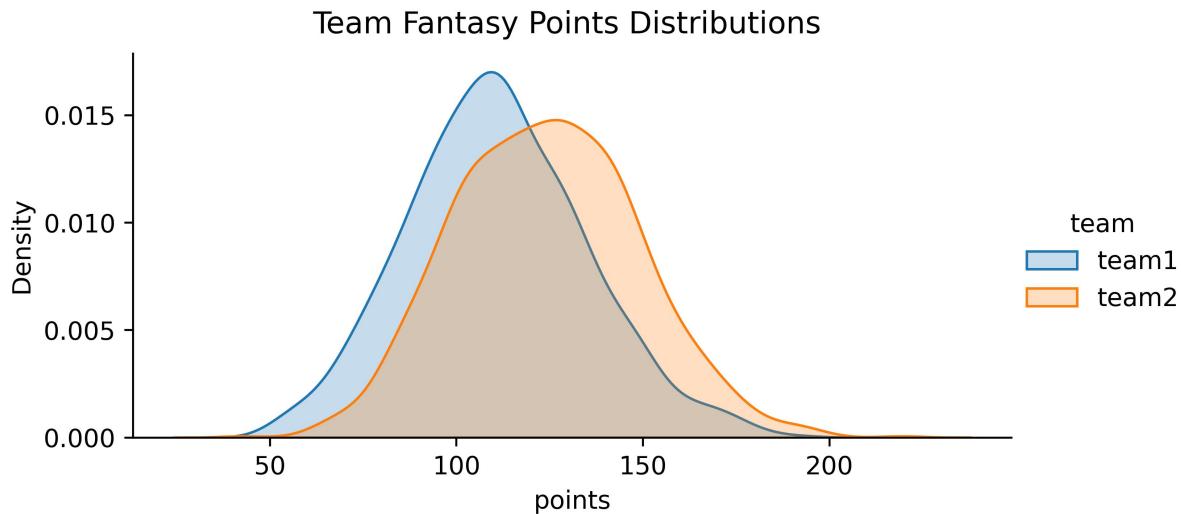


Figure 0.1: Team 1 vs Team 2

That's nice, but similar to above, let's extend it to ALL our WDIS options and plot them all at once.

We'll do the same thing we did earlier, this time creating a separate column for each player, and adding in our opponents total too.

```
In [8]:  
points_wide = pd.concat(  
    [nsims[team1_sans_starter].sum(axis=1) + nsims[player] for player in  
     wdis],  
    axis=1)  
points_wide.columns = wdis  
points_wide['opp'] = nsims[team2].sum(axis=1)  
--  
  
In [9]: points_wide.head()  
Out[9]:  
   clyde-edwards-helaire  darrell-henderson ... tony-pollard          opp  
0            93.641759      99.753182 ...    90.918777  158.991433  
1           139.811889     131.232813 ...   125.798348  86.806277  
2            97.781934      90.909932 ...    80.659367 108.607376  
3           113.214746     126.710855 ...   117.297851 105.672497  
4           134.904767     106.344098 ...   111.046707 115.706523
```

This is the first step. The rest is the same as before.

```
In [10]:
points_long = points_wide.stack().reset_index()
points_long.columns = ['sim', 'team', 'points']
-- 

In [11]: points_long.head(10)
Out[11]:
   sim              team    points
0    0  clyde-edwards-helaire  93.641759
1    0        darrell-henderson  99.753182
2    0          ronald-jones  93.540081
3    0          tony-pollard  90.918777
4    0                  opp  158.991433
5    1  clyde-edwards-helaire 139.811889
6    1        darrell-henderson 131.232813
7    1          ronald-jones 124.167215
8    1          tony-pollard 125.798348
9    1                  opp  86.806277
```

And the plot:

```
In [12]:
g = sns.FacetGrid(points_long, hue='team', aspect=2)
g = g.map(sns.kdeplot, 'points', fill=True)
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle('Team Fantasy Points Distributions - WDIS Options')
```

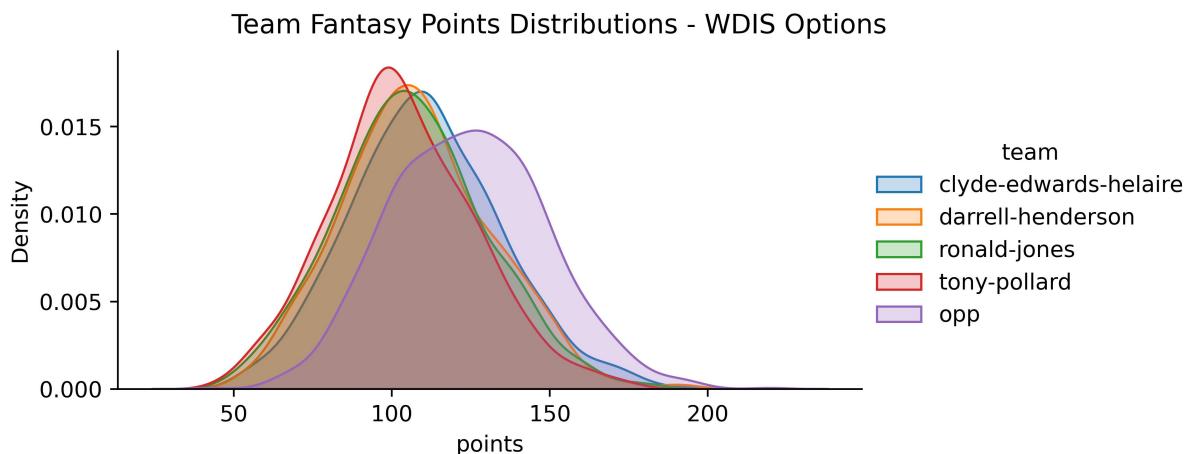


Figure 0.2: WDIS Options vs Opponent

It's working. Now let's put everything in a function. The final code:

```
def wdis_plot(nsims, team1, team2, wdis):

    # do some validity checks
    current_starter = set(team1) & set(wdis)
    assert len(current_starter) == 1

    bench_options = set(wdis) - set(team1)
    assert len(bench_options) >= 1

    #
    team_sans_starter = list(set(team1) - current_starter)

    # total team points under allt he starters
    points_wide = pd.concat(
        [nsims[team_sans_starter].sum(axis=1) + nsims[player] for player
         in
         wdis], axis=1)

    points_wide.columns = wdis

    # add in opponent
    points_wide['opp'] = nsims[team2].sum(axis=1)

    # shift data from columns to rows to work with seaborn
    points_long = points_wide.stack().reset_index()
    points_long.columns = ['sim', 'team', 'points']

    # actual plotting portion
    g = sns.FacetGrid(points_long, hue='team', aspect=4)
    g = g.map(sns.kdeplot, 'points', fill=True)
    g.add_legend()
    g.fig.subplots_adjust(top=0.9)
    g.fig.suptitle('Team Fantasy Points Distributions - WDIS Options')

    return g
```

The only other thing is that — since these team distribution plots are so close together — it might be useful to look at the plots of our WDIS players individually.

Let's do that quick.

```
In [13]:  
# plot wdis players  
pw = sims[wdis].stack().reset_index()  
pw.columns = ['sim', 'player', 'points']  
  
In [14]:  
g = sns.FacetGrid(pw, hue='player', aspect=2)  
g = g.map(sns.kdeplot, 'points', shade=True)  
g.add_legend()  
g.fig.subplots_adjust(top=0.9)  
g.fig.suptitle(f'WDIS Projections')
```

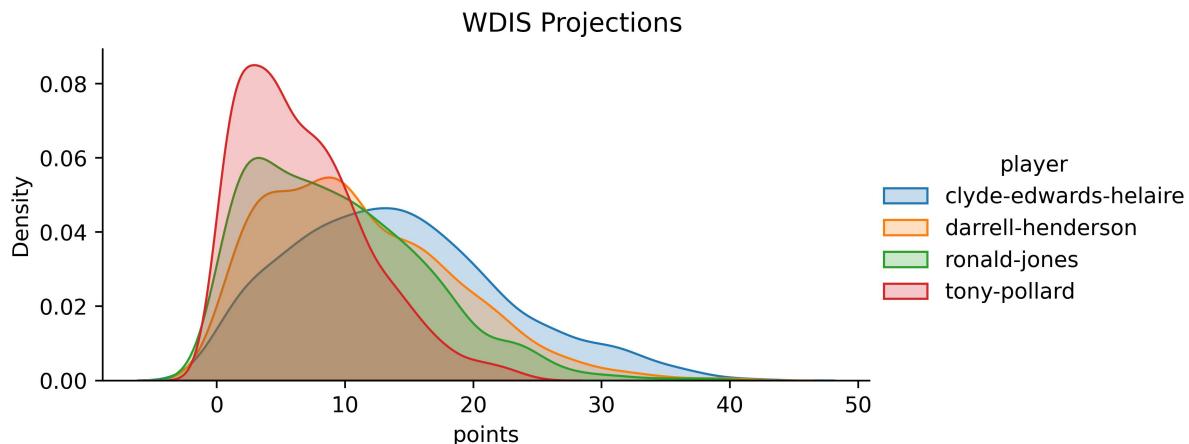


Figure 0.3: WDIS Options - Player Plots

Looks good.

WDIS Wrap Up

Now you should be able to add a few points of win probability to all your matchups for the rest of the season. See [./code/wdis_manual.py](#) for the cleaned up version. There I've renamed our `wdis_plus` and `wdis_plot` functions `calculate` and `plot` respectively.

That way we can import it:

```
import wdis
```

And run:

```
wdis.calculate(sims, team1, team2, bench)  
wdis.plot(sims, team1, team2, bench)
```

Now that we have this function, let's write some code to connect to our league so we don't have to populate our `team1`, `team2` and `bench` list of player ids by hand.

7. Project #2: League Integration

In this section we'll write code that connects to our ESPN, Yahoo, Fleaflicker, or Sleeper league and automatically gets the data we need to for the who do I start and league analysis projects.

Working with Public APIs — General Process

Most of the APIs we'll be using in this section aren't meant for people like us. Instead, they're what these platforms use internally to power their own websites. This means (1) they return a *lot* of data, most of which we won't use, and (2) they don't come with detailed, helpful instructions.

Documentation ranges from technically complete but not super informative (Fleaflicker) to a bunch of PHP examples that haven't been updated in 10 years (Yahoo) to non-existent/a few third party blog posts (ESPN).

Let's talk for a bit about general process of working with these APIs.

1. Authentication

The very first step is authentication, i.e. making sure we can actually put in URL and get data back.

The authentication experience differs by platform. Fleaflicker and Sleeper don't require authentication at all, and let anyone look at any league data. This makes things a lot easier for us.

ESPN requires authentication to access private leagues, and we'll do some mild hacking that'll allow you to query data for the leagues you're in.

Authentication in Yahoo is a pain (don't let this scare you, we'll walk through it step by step), and involves getting API credentials + installing a third party authentication library.

2. Finding an endpoint

After we're authenticated, the next step is thinking about what we need and finding the right endpoint. Generally, this is done 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

Again, the difficulty of this varies. All Fleaflicker's endpoints are documented, so it's pretty easy to figure out how to get what you want.

Yahoo and ESPN not so much. The walk throughs in this book are based on a heavy dose of (2) and (3) + trial and error. Shoutouts in particular to [uberfastman/yfpy](#) and Steven Morse, who wrote up a [blog post for ESPN](#).

3. Visit endpoint in browser

After finding the endpoint you want, the next step is visiting it in your browser and exploring it. All of these APIs return JSON, which is a bunch of nested dicts and lists.

And again, league hosting platforms are complex sites, with lots of edge cases and info they want to show. These APIs are powering all that, and so they need to have a lot of info a lot of data. We don't need most of it.

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.

Your browser might display JSON data as a wall of unformatted text. For example, here's how the Fleaflicker API looks to me in Chrome:

2025 Fantasy Football Developer Kit

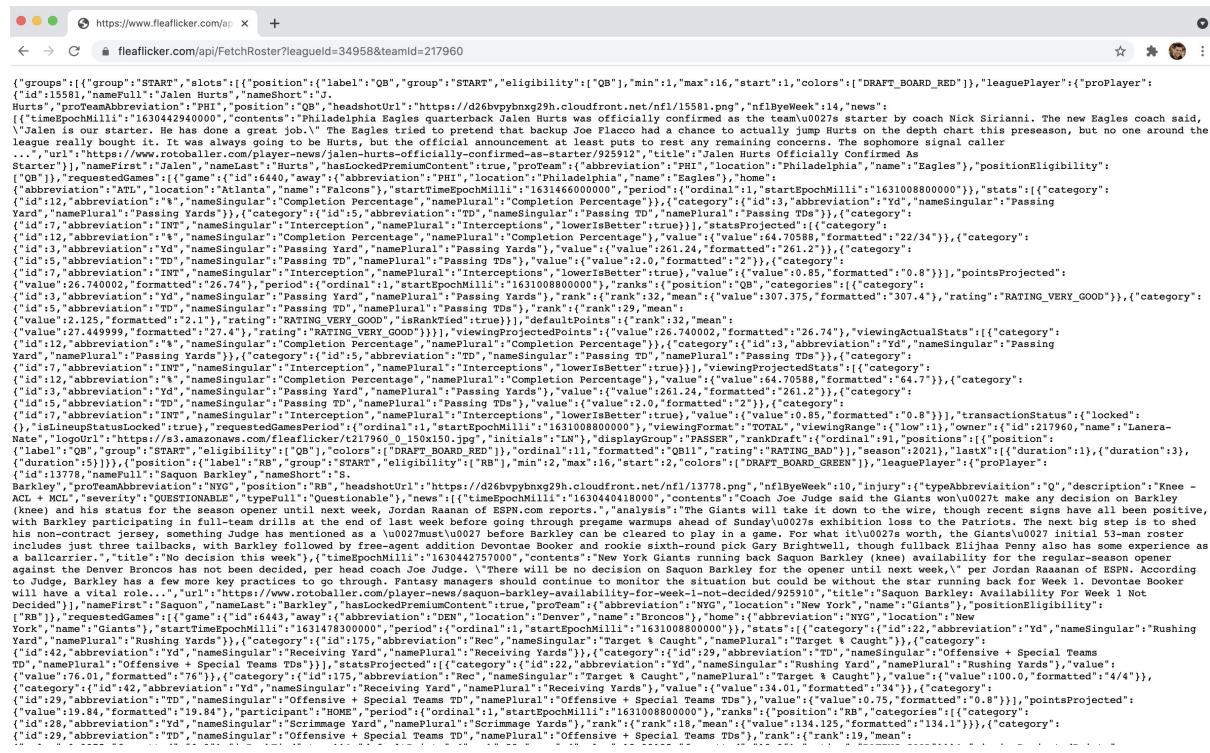
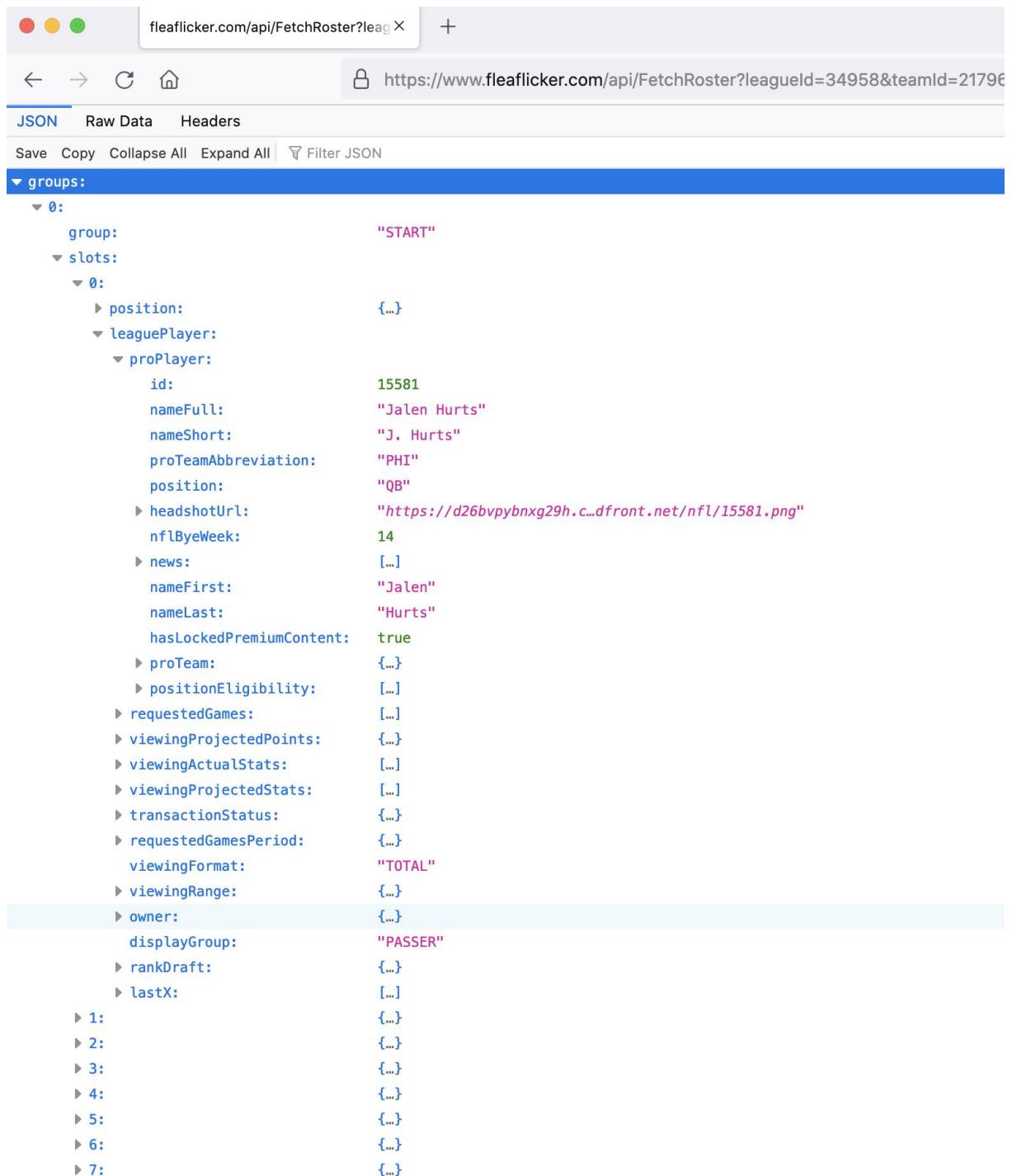


Figure 0.1: Unformatted JSON in Chrome

This is impossible to make sense of. So the 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 formatted bullets we can expand, visualize, etc.

2025 Fantasy Football Developer Kit



The screenshot shows a Firefox browser window with the URL <https://www.fleaflicker.com/api/FetchRoster?leagueId=34958&teamId=21796>. The page displays a JSON object representing a roster entry for Jalen Hurts. The JSON structure includes a 'group' field set to "START", a 'slots' array containing one slot for a 'proPlayer' (Jalen Hurts), and a 'lastX' field.

```
group: "START"
slots:
  0:
    position: {...}
    leaguePlayer:
      proPlayer:
        id: 15581
        nameFull: "Jalen Hurts"
        nameShort: "J. Hurts"
        proTeamAbbreviation: "PHI"
        position: "QB"
        headshotUrl: "https://d26bvpbynng29h.c.dfront.net/nfl/15581.png"
        nflByeWeek: 14
        news: [...]
        nameFirst: "Jalen"
        nameLast: "Hurts"
        hasLockedPremiumContent: true
      proTeam: {...}
      positionEligibility: [...]
      requestedGames: [...]
      viewingProjectedPoints: [...]
      viewingActualStats: [...]
      viewingProjectedStats: [...]
      transactionStatus: [...]
      requestedGamesPeriod: [...]
      viewingFormat: "TOTAL"
      viewingRange: [...]
      owner: [...]
      displayGroup: "PASSE"
      rankDraft: [...]
      lastX: [...]
    1: {...}
    2: {...}
    3: {...}
    4: {...}
    5: {...}
    6: {...}
    7: {...}
```

Figure 0.2: Formatted JSON in FireFox

4. Get what you need in Python

Once we've looked at the JSON in our browser and have an idea what we want, we can access the API in Python and get what we need out of it.

In these projects we're almost always processing collections of things. So we'll query our team roster endpoint and get back a collection of players (e.g. a list of player dictionaries). Or we'll get info on all the teams in our league.

Our general process will be to pick out one item from the collection (e.g. the Aaron Rodgers dict), play around with it to create a function to get only what we need out of it, then apply the function to the entire collection (all the players or teams).

5. Clean things up

After we have working code that gets everything we'll need, it's time to clean things up. I've done this in "final" versions of all the code.

In this case, part of cleaning things up is standardizing what we end up with so it's the same for every platform. That way we can talk about a general process from going from site data → WDIS for example, vs having to look at ESPN → WDIS, Yahoo → WDIS, separately.

More on our standardized outputs next.

Common Outputs

Though they have a lot of similarities, obviously, the code we write to access the data is different for every site. I'd recommend working through only the platforms you use.

But importantly, no matter which host we're on, we'll make sure we end up with *exactly* the same outputs, and that these outputs will easily plug into our other projects.

Here are the 5 pieces of info we'll need when we're done with each platform:

1. Team Data

We need data about all the teams in our league. This will be at the *team* level (e.g. 12 rows for a 12 team league). It will include:

- `team_id`
- `owner_id`

- `owner_name`
- `league_id`

The function that gets this data will be called `get_teams_in_league`.

2 and 3. Matchup and Team Schedule Data

In order to automatically analyze matchups, we need schedule data.

We'll store this at the *game* level (so 6 rows per week for a 12 team league). At some point though we'll want it at the *team* and *week* level too (12 rows per week for a 12 team league) so we'll also make a helper functions to go back and forth.

The `schedule` table will include columns for:

- `team1_id`
- `team2_id`
- `matchup_id`
- `season`,
- `week`
- `league_id`

The function that returns this data will be called `get_league_schedule`.

4. Roster Data

Unlike the first three tables, which won't ever change in a given season. The `roster` table is a snapshot at some moment in time. It includes player information and whether (and what position) we're starting them. It also includes any actual points, e.g. if we're getting data on a Friday and the TNF guys have already played, we'll want to keep track of what they've scored.

Here are the fields: - `player_id` - `name` - `player_position` - `team_position` - `start` - `team_id` - `actual`

The function that gets this data will get roster information for all teams and be called `get_league_rosters`

And that's it! If we can write some code that gets us the above, we'll have everything we need to hook into our wdis (last chapter) and weekly league analyzer (next chapter) projects.

The next step is platform specific. For each site, we'll go through and get all the data above in the same format, so that final, top level functions:

- `get_league_rosters`
- `get_teams_in_league`
- `get_league_schedule`

All have the same names.

When we're done, we'll pick back up together and connect these outputs to the rest of our projects.

ESPN Integration

Note: this section covers how to get data from leagues hosted on ESPN. Skip this and find the relevant section (Yahoo, Fleaflicker, Sleeper) if your league is hosted on something else.

Authentication and Setup

To work with ESPN you need to know two things. Your:

- league id
- team id

You can get those by logging into ESPN fantasy and going to your main team page. Mine is:

<https://fantasy.espn.com/football/team?leagueId=242906&seasonId=2021&teamId=11&fromTeamId=11>

Here, my league id is 242906 and my team id is 11.

Notice that if you try to click on the link above you won't see anything. It's a private league. That's fine. Just do it with your own league.

Connecting to ESPN in Python

In this section we'll be working through [./projects/integration/espn_working.py](#), so open it up. We'll pick up right after the imports.

Put your league and team id in [espn_working.py](#):

```
In [1]:  
LEAGUE_ID = 395209553  
TEAM_ID = 12
```

ESPN has a “public” API. But you can only use it to get data on leagues that you’re in. How it works:

Normally you login to ESPN on a browser like Chrome or Firefox. When you do, ESPN tells your browser to save a tiny bit of data (a “cookie”). Then, when you come back later — either to visit the site like normal or to hit up the API in your browser — ESPN looks for the cookie, finds it, and lets you in without making you login again.

It turns out that you need to have these cookies to access the ESPN API too.

After you’ve logged into your ESPN fantasy league, with the same browser, try going to this API url:

<https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/segments/2023/leagues/395209553?view=mRoster>

where you replace 395209553 with whatever your league id is.

If you've logged in previously, you should be seeing some data. You are now accessing the ESPN API!

This works because your browser stores your ESPN cookies, and you're using this same browser to access the API.

In Python (as opposed to Chrome or Firefox), we access urls using the `requests` library, like this:

```
response = requests.get(roster_url)
```

But Python does *not* have your cookies. So if we tried this, we'd get back this not authorized message:

```
In [2]: response.json()
Out[2]:
{'messages': ['You are not authorized to view this League.'],
 'details': [{'message': 'You are not authorized to view this League.',
   'shortMessage': 'You are not authorized to view this League.',
   'resolution': None,
   'type': 'AUTH_LEAGUE_NOT_VISIBLE',
   'metaData': None}]}]
```

So, unlike the browser, Python doesn't have our cookies, and ESPN won't let us in. What should we do?

The answer is get our cookies out of our browser and put them into Python.

Getting your ESPN Authorization Cookies in Python

In the browser, *after* you've logged into ESPN fantasy, right click anywhere on the page and click *Inspect*. Then click on the *Storage* tab, go down to *Cookies*, then <https://fantasy.espn.com>. There will be a lot of cookies there, but we're looking for two: `swid` and `espn_s2`. Find and copy them into your `config.ini` file.

Mine (with a bunch X'd out so no one uses them to log into my ESPN account and drop all my players) are:

```
[espn]
SWID = XXXXXXXX-0004-4E26-AE4E-XXXXXXXXXXXX
ESPN_S2 = AEBd3f6XXXXXXX...XXXXXXXXY8gJl3rF%2Btr0zcYyK5Fst1Q3tzfP
```

This login and copy cookies to your code workflow is kind of hackish, but you should only have to do it once. If it stops working (i.e. you get a 401 not authorized error) you can log back in and grab new ones.

Note: after pasting these to `config.ini`, you may have to quit and restart Spyder for `utilities.py` to pick them up.

ESPN Endpoints

OK. That should take care of authentication (we'll make sure in a second). Now we need to find an endpoint to hit.

The ESPN API is basically undocumented, apart from blog posts like [this one from Steven Morse](#), which is what we'll use as our starting point.

According to Steven, ESPN API endpoints take the following form:

`https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/seasons/2023/segments/0/leagues/LEAGUE_ID?view=VIEW`

Where `LEAGUE_ID` is your ESPN league id, and `VIEW` is one of:

- `mTeam`
- `mBoxscore`
- `mRoster`
- `mSettings`
- `kona_player_info`
- `player_wl`
- `mSchedule`

So our approach will be:

1. Think about the data we want (rosters, team information, etc).
2. Try one of the endpoints above out in the browser.
3. Figure out if (and where) it has the information we need.
4. Get that information out using Python.

Remember, all of these APIs return JSON, which are nested dicts and lists of dicts and lists. **Almost always, we'll be manipulating these data structures until we have list of Python dictionaries that all have the same keys. Then we'll pass these to DataFrame.**

Roster Data

Let's start with roster data. Looking at our list of endpoints, `mRoster` seems like the logical thing to try:

```
In [1]:  
roster_url = (f'https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/  
seasons/{SEASON}' +  
    f'/segments/0/leagues/{LEAGUE_ID}?view=mRoster')
```

So let's access it using `requests`. Note our `get` request is including the ESPN authentication cookies we got from the browser:

```
In [2]: roster_json = requests.get(roster_url,  
                                cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()
```

This is how you get live data for the current week and how you'll analyze your own team, and you should run it and make sure it worked.

However — to make it easier to follow along, let's use some specific data I saved from the ESPN API between Thursday and Sunday, Week 2, 2023.

We can load that data like this:

```
In [3]:  
with open('./projects/integration/raw/espn/roster.json') as f:  
    roster_json = json.load(f)
```

Just like a normal API endpoint, we can view this file in the browser. Just find it in Finder or Windows Explorer and open it up. When you do look at it (remember you should have a JSON viewer installed) we'll see something like this:

2025 Fantasy Football Developer Kit

```
JSON Raw Data Headers
Save Copy Collapse All Expand All (slow) Filter JSON

▶ draftDetail: {...}
  gameId: 1
  id: 242906
  scoringPeriodId: 1
  seasonId: 2021
  segmentId: 0
▶ status: {...}

▼ teams:
  ▼ 0:
    id: 1
    ▶ roster:
      appliedStatTotal: 43.06
      ▶ entries:
        ▼ 0:
          acquisitionDate: 1630203984939
          acquisitionType: "DRAFT"
          injuryStatus: "NORMAL"
          lineupSlotId: 2
          pendingTransactionIds: null
          playerId: 3068267
          ▶ playerPoolEntry:
            appliedStatTotal: 0
            id: 3068267
            keeperValue: 2
            keeperValueFuture: 1
            lineupLocked: false
            onTeamId: 1
            ▶ player:
              active: true
              defaultPositionId: 2
              ▶ draftRanksByRankType: {...}
                droppable: true
              ▶ eligibleSlots: [...]
              firstName: "Austin"
              fullName: "Austin Ekeler"
              id: 3068267
              injured: false
              injuryStatus: "QUESTIONABLE"
              lastName: "Ekeler"
              lastNewsDate: 1631445778000
              ▶ outlooks: {...}
              ▶ ownership: {...}
              proTeamId: 24
              ▶ rankings: {...}
              ▶ seasonOutlook: "Ekeler missed six games ... a mid-to-back-end RB1."
              ▶ stats: [...]
              universeId: 1
              ▶ ratings: {...}
              rosterLocked: false
              status: "ONTEAM"
              tradeLocked: false
              status: "NORMAL"
        ▶ 1:
      
```

Figure 0.3: ESPN Roster JSON

We're looking for roster data, which is under the teams field. So let's get that:

```
In [4]: list_of_rosters = roster_json['teams']
```

My approach to these things is: get the collection of things I'm interested in (teams/rosters in this case), usually it'll be in a list. Then once I have that collection, I pick out a specific example and look at it more.

So let's take the first roster in `list_of_rosters` and put it in `roster0`:

```
In [5]: roster0 = list_of_rosters[0]
```

```
In [6]: roster0
```

```
Out[6]:
```

```
...
'proTeamId': 0,
'scoringPeriodId': 1,
'seasonId': 2021,
'statSourceId': 1,
'statSplitTypeId': 1,
'stats': {'74': 0.207511225,
'75': 0.302526474,
'76': 0.095015248,
'77': 0.503590937,
'78': 0.626803217,
'79': 0.12321228,
'80': 1.034058366,
'81': 1.047967524,
'82': 0.013909157,
'83': 1.745160528,
'84': 1.977297215,
'85': 0.232136686,
'86': 1.875782815,
'87': 1.900628801,
'88': 0.024845985,
'198': 0.207511225,
'199': 0.302526474,
'200': 0.095015248,
'210': 1.0}}],
'universeId': 1},
'rosterLocked': False,
'status': 'ONTEAM',
'tradeLocked': False},
'status': 'NORMAL'}],
'tradeReservedEntries': 0}]]}
```

In this case there's so much data that it's difficult to view in the console. That's fine. The "make a list" → "view one item" process is *recursive*, i.e. we keep doing it and go deeper as long as we need to.

In this case our roster is full of players. From looking at in the browser it looks like these players are in the '`entries`' field:

```
In [7]: list_of_players_on_roster0 = roster0['roster']['entries']
```

Specific example:

```
In [8]: roster0_player0 = list_of_players_on_roster0[0]
```

Even a single player entry is too much data to look at here, but we can see it's just a Python dict.

```
In [9]: roster0_player0.keys()
Out[9]: dict_keys(['acquisitionDate', 'acquisitionType', 'injuryStatus',
                  'lineupSlotId', 'pendingTransactionIds', 'playerId',
                  'playerPoolEntry', 'status'])
```

Remember what we need to get here for each player:

1. his ESPN player id
2. whether we're starting him and the position they're in if we are (e.g. if it's an RB, is he in the RB spot or flex?)
3. the player's position and name

Let's write a function to get that info from `roster0_player0`. I got the field names by looking at the data in the browser:

```
In [10]:
def process_player1(player):
    dict_to_return = {}
    dict_to_return['team_position'] = player['lineupSlotId']
    dict_to_return['espn_id'] = player['playerId']

    dict_to_return['name'] = (
        player['playerPoolEntry']['player']['fullName'])
    dict_to_return['player_position'] = (
        player['playerPoolEntry']['player']['defaultPositionId'])

    return dict_to_return
```

And trying it out:

```
In [11]: process_player1(roster0_player0)
Out[11]:
{'team_position': 4,
 'espn_id': 4262921,
 'name': 'Justin Jefferson',
 'player_position': 3}
```

So this first player is Justin Jefferson. The positions are coded as numbers (4) instead of more readable string values ('WR'). After some Googling I found a key:

```
In [12]:  
TEAM_POSITION_MAP = {  
    0: 'QB', 1: 'TQB', 2: 'RB', 3: 'RB/WR', 4: 'WR', 5: 'WR/TE',  
    6: 'TE', 7: 'OP', 8: 'DT', 9: 'DE', 10: 'LB', 11: 'DL',  
    12: 'CB', 13: 'S', 14: 'DB', 15: 'DP', 16: 'D/ST', 17: 'K',  
    18: 'P', 19: 'HC', 20: 'BE', 21: 'IR', 22: '', 23: 'RB/WR/TE',  
    24: 'ER', 25: 'Rookie', 'QB': 0, 'RB': 2, 'WR': 4, 'TE': 6,  
    'D/ST': 16, 'K': 17, 'FLEX': 23, 'DT': 8, 'DE': 9, 'LB': 10,  
    'DL': 11, 'CB': 12, 'S': 13, 'DB': 14, 'DP': 15, 'HC': 19}  
  
PLAYER_POSITION_MAP = {1: 'QB', 2: 'RB', 3: 'WR', 4: 'TE', 5: 'K',  
                      16: 'D/ST'}
```

So let's modify our function to add these more readable position values.

```
In [13]:  
def process_player2(player):  
    dict_to_return = {}  
    dict_to_return['team_position'] = (  
        TEAM_POSITION_MAP[player['lineupSlotId']])  
    dict_to_return['espn_id'] = player['playerId']  
  
    dict_to_return['name'] = (  
        player['playerPoolEntry']['player']['fullName'])  
  
    dict_to_return['player_position'] = (  
        PLAYER_POSITION_MAP[  
            player['playerPoolEntry']['player']['defaultPositionId']])  
    return dict_to_return
```

Then we can run it on every player using a list comprehension.

```
In [14]: [process_player2(x) for x in list_of_players_on_roster0]
Out[14]:
[{'team_position': 'WR',
 'espn_id': 4262921,
 'name': 'Justin Jefferson',
 'player_position': 'WR'},
 {'team_position': 'QB',
 'espn_id': 4040715,
 'name': 'Jalen Hurts',
 'player_position': 'QB'},
 {'team_position': 'RB',
 'espn_id': 4239996,
 'name': 'Travis Etienne Jr.',
 'player_position': 'RB'},
 {'team_position': 'WR',
 'espn_id': 16737,
 'name': 'Mike Evans',
 'player_position': 'WR'},
 ...
 ...]
```

Remember, when you have a list of dictionaries that have the same fields — which is what we have here (all the players have `name`, `player_position`, `team_position`, `espn_id` fields) — you should put them in a DataFrame ASAP.

```
In [15]: roster0_df = DataFrame([process_player2(x) for x in
                                list_of_players_on_roster0])
```

```
In [16]: roster0_df
Out[16]:
   team_position  espn_id          name player_position
0            WR  4262921    Justin Jefferson        WR
1            QB  4040715      Jalen Hurts        QB
2            RB  4239996  Travis Etienne Jr.       RB
3            WR   16737      Mike Evans        WR
4        RB/WR/TE  3045138      Mike Williams        WR
5            RB  4239934      AJ Dillon        RB
6            TE  3123076     David Njoku       TE
7            BE  2980453    Jamaal Williams        RB
8            BE  2576414    Raheem Mostert        RB
9            K   15683     Justin Tucker        K
10           BE  4241555    Elijah Mitchell       RB
11           BE  4569609      Evan Hull        RB
12          D/ST -16004    Bengals D/ST      D/ST
13           BE  4360078      Alec Pierce        WR
14           BE  2576623    DeVante Parker        WR
15           BE   16760     Jimmy Garoppolo       QB
```

Awesome.

This looks pretty good, but at some point we're probably going to want to refer to players on our roster by (team) position, in which case duplicate `team_position` might be a problem. It'd be better to be able to refer to our RB1, RB2 etc.

Let's add that. As always let's start with a specific example. Say, WRs:

```
In [17]: wrs = roster0_df.query("team_position == 'WR'")
```

```
In [18]: wrs
```

```
Out[18]:
```

	team_position	espn_id	name	player_position
0	WR	4262921	Justin Jefferson	WR
3	WR	16737	Mike Evans	WR

So we want `team_position` to say WR1, WR2 instead of just WR, WR.

The easiest way is probably to make a column with 1, 2 then stick them together.

```
In [19]: suffix = Series(range(1, len(wrs) + 1), index=wrs.index)
```

```
In [20]: suffix
```

```
Out[20]:
```

3	1
4	2

The key is we're making this column have the same index as `wrs`. That way we can do this:

```
In [21]: wrs['team_position'] + suffix.astype(str)
```

```
Out[21]:
```

4	WR1
5	WR2

Which is what we want. Now let's put this in a function:

```
In [22]:  
def add_pos_suffix(df_subset):  
    # only add if more than one position -- want just K, not K1  
    if len(df_subset) > 1:  
        suffix = Series(range(1, len(df_subset) + 1), index=df_subset.  
                        index)  
  
        df_subset['team_position'] = (  
            df_subset['team_position'] + suffix.astype(str))  
    return df_subset
```

and apply it to every position in our DataFrame.

```
In [23]: roster0_df2 = pd.concat([
    add_pos_suffix(roster0_df.query(f"team_position == '{x}'"))
    for x in roster0_df['team_position'].unique()])
In [24]: roster0_df2
Out[24]:
   team_position  espn_id      name player_position
0            WR1  4262921  Justin Jefferson        WR
3            WR2   16737     Mike Evans        WR
1            QB   4040715    Jalen Hurts       QB
2            RB1  4239996  Travis Etienne Jr.      RB
5            RB2  4239934      AJ Dillon        RB
4  RB/WR/TE  3045138  Mike Williams        WR
6            TE  3123076     David Njoku       TE
7            BE1  2980453  Jamaal Williams       RB
8            BE2  2576414    Raheem Mostert      RB
10           BE3  4241555  Elijah Mitchell       RB
11           BE4  4569609      Evan Hull        RB
13           BE5  4360078      Alec Pierce        WR
14           BE6  2576623  DeVante Parker        WR
15           BE7   16760  Jimmy Garoppolo       QB
9            K   15683     Justin Tucker        K
12          D/ST  -16004  Bengals D/ST        D/ST
```

Cool. Now let's identify our starters:

```
In [25]: roster0_df2['start'] = (
    ~roster0_df2['team_position'].str.startswith('BE'))
In [26]: roster0_df2
Out[26]:
   team_position  espn_id      name player_position  start
0            WR1  4262921  Justin Jefferson        WR  True
3            WR2   16737     Mike Evans        WR  True
1            QB   4040715    Jalen Hurts       QB  True
2            RB1  4239996  Travis Etienne Jr.      RB  True
5            RB2  4239934      AJ Dillon        RB  True
4  RB/WR/TE  3045138  Mike Williams        WR  True
6            TE  3123076     David Njoku       TE  True
7            BE1  2980453  Jamaal Williams       RB False
8            BE2  2576414    Raheem Mostert      RB False
10           BE3  4241555  Elijah Mitchell       RB False
11           BE4  4569609      Evan Hull        RB False
13           BE5  4360078      Alec Pierce        WR False
14           BE6  2576623  DeVante Parker        WR False
15           BE7   16760  Jimmy Garoppolo       QB False
9            K   15683     Justin Tucker        K  True
12          D/ST  -16004  Bengals D/ST        D/ST  True
```

Alright. Let's stop and put all of this into a function:

```
def process_players(entries):
    roster_df = DataFrame([process_player2(x) for x in entries])

    roster_df2 = pd.concat([
        add_pos_suffix(roster_df.query(f"team_position == '{x}'"))
        for x in roster_df['team_position'].unique()])

    roster_df2['start'] = (
        ~roster_df2['team_position'].str.startswith('BE'))

    return roster_df2
```

I'm not showing it here because I don't necessarily want to keep showing the same data over and over, but you should try this out on `list_of_players_on_roster0` to make sure it works.

Now let's think about team id. Looking at the JSON, it looks like team id is further up from `entries`, next to `roster`. Let's play around with adding it.

As always, it's easiest to start with a specific example. We've been working with `list_of_rosters[0]` so let's do the next one:

```
In [27]: roster1 = list_of_rosters[1]
```

So we need our team id:

```
In [28]: roster1['id']
Out[28]: 2
```

And the `entries` data, which we can run through our function:

```
In [29]: process_players(roster1['roster']['entries']).head()
Out[29]:
   team_position  espn_id           name player_position  start
0            WR1  4362628     JaMarr Chase             WR  True
3            WR2  3915416       DJ Moore             WR  True
1            QB  3918298      Josh Allen             QB  True
2            RB1  4241457     Najee Harris            RB  True
5            RB2  4035886    Khalil Herbert            RB  True
```

So what we need to do is build a function that wraps (i.e. calls) both these functions inside of it, adds team id, then returns it.

```
def process_roster(team):
    roster_df = process_players(team['roster']['entries'])
    roster_df['team_id'] = team['id']
    return roster_df
```

And let's try it out:

```
In [30]: roster3 = list_of_rosters[3]

In [31]: process_roster(roster3).head()
Out[31]:
   team_position  espn_id           name  player_position  start
   team_id
0          RB1    3117251 Christian McCaffrey      RB     True
4
2          RB2    3042519      Aaron Jones      RB     True
4
1          WR1    4241478     DeVonta Smith      WR     True
4
5          WR2    3895856     Christian Kirk      WR     True
4
3    RB/WR/TE  4048244 Alexander Mattison      RB     True
4
```

Nice. Now we can easily run this on every team, and stick the DataFrames together for complete rosters.

```
In [32]: all_rosters = pd.concat([process_roster(x)
                                 for x in list_of_rosters],
                                 ignore_index=True)
```

Let's look at a random sample to make sure we're getting different teams etc.

```
In [33]: all_rosters.sample(15)
Out[33]:
   team_position  espn_id           name  ...  start  team_id
62            K    4360234  Evan McPherson  ...  True     4
42           TE    2576925  Darren Waller  ...  True     3
66           WR2   3128429 Courtland Sutton  ...  True     6
165          BE4   4426485  Jonathan Mingo  ... False    12
10           BE4   4569609      Evan Hull  ... False    1
100          RB2   4373626  Tyler Allgeier  ...  True     8
180          BE2   4372414      Elijah Moore  ... False   13
7            BE1   2980453  Jamaal Williams  ... False    1
34           BE3   3051392  Ezekiel Elliott  ... False    3
175          RB2   4035538  David Montgomery  ...  True   13
8            BE2   2576414  Raheem Mostert  ... False    1
101          IR    4242335  Jonathan Taylor  ...  True     8
28           BE6   4686472  Marvin Mims Jr.  ... False    2
212          BE1   4430191      Skyy Moore  ... False   16
201          RB2   4241474 Brian Robinson Jr.  ...  True   15
```

Great. Anything else?

Well, the saved data we're looking at happens to be from Week 2, 2023, *after* PHI-MIN played Thursday

night but *before* all the games on Sunday. So some of the players played and have actual scores. We'll definitely want this info too.

It'd be nice if these actual scores were somewhere in the JSON from this `mRoster` endpoint but after looking at this I don't think it is. The good news it's available, just in a different endpoint (`mBoxscore`). So let's call it quick:

```
In [1]: boxscore_json = requests.get(boxscore_url,
                                      cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()
```

Again, you should call this and make sure it works, but for the purposes of working through this let's use my example data.

```
In [2]:
with open('./projects/integration/raw/espn/boxscore.json') as f:
    boxscore_json = json.load(f)
```

There's enough going on here that we should really look at it in the browser (again, find it on your system and open it up — it should open in Chrome or Firefox).

Here it is.

The screenshot shows a JSON viewer interface with tabs for "JSON", "Raw Data", and "Headers". The "JSON" tab is selected, displaying a hierarchical tree of data. The data structure includes fields like draftDetail, gameId, id, schedule, and player. Two specific entries under the player node are highlighted with red circles: "appliedStatTotal" and "id". The "appliedStatTotal" field is set to 28.560000000000002, and the "id" field is set to 14880. These highlighted fields correspond to the data points being analyzed in the accompanying text.

```
JSON Raw Data Headers
Save Copy Collapse All Expand All (slow) Filter JSON
draftDetail: {...}
gameId: 1
id: 395209553
schedule:
  0: {...}
  1: {...}
  2: {...}
  3: {...}
  4: {...}
  5: {...}
  6: {...}
  7:
    away:
      cumulativeScore: {...}
      rosterForCurrentScoringPeriod: {...}
      rosterForMatchupPeriod:
        appliedStatTotal: 28.560000000000002
      entries:
        0:
          lineupSlotId: 0
          playerId: 14880
          playerPoolEntry:
            appliedStatTotal: 28.560000000000002
            id: 14880
        player:
          defaultPositionId: 1
          eligibleSlots: [...]
          firstName: "Kirk"
          fullName: "Kirk Cousins"
          id: 14880
          lastName: "Cousins"
          proTeamId: 16
          stats: [...]
          universeId: 2
        rosterForMatchupPeriodDelayed:
          teamId: 8
          tiebreak: 0
          totalPoints: 0
      home:
        id: 8
        matchupPeriodId: 2
      ...
    ...
  ...
...
}
```

Figure 0.4: Kirk Cousins Week 2 ESPN Stats

A few things I noted looking at this:

- the data we want is in the '`schedule`' field
- '`schedule`' is a giant list of "matchups". Each of these contain "home" and "away" data (I think this is misleading — what does home and away mean in fantasy? — but whatever) for *all* games, *all* season
- inside each '`home`' / '`away`' field the data we want is somewhere in `rosterForMatchupPeriod`

Take the away team week 2, matchup 0. On that team, the only player who's played is Kirk Cousins, who scored 28.56 points. I circled the data we want (his actual points and his ESPN id) in red on the screenshot.

So let's start by finding the dict that has that data. First we need to limit our matchups to the week we're on:

```
In [3]: matchup_list = [x for x in boxescore_json['schedule'] if  
                      x['matchupPeriodId'] == WEEK]
```

Then we can grab a matchup:

```
In [4]: matchup0 = matchup_list[0]  
  
In [5]: matchup0_away = matchup0['away']  
  
In [6]: matchup0_away_roster = (  
          matchup0['away']['rosterForMatchupPeriod']['entries'])  
  
In [7]: cousins_dict = matchup0_away_roster[0]
```

It looks like this:

```
In [8]: cousins_dict
Out[8]:
{'lineupSlotId': 0,
 'playerId': 14880,
 'playerPoolEntry': {'appliedStatTotal': 28.560000000000002,
 'id': 14880,
 'player': {'defaultPositionId': 1,
 'eligibleSlots': [0, 7, 20, 21],
 'firstName': 'Kirk',
 'fullName': 'Kirk Cousins',
 'id': 14880,
 'lastName': 'Cousins',
 'proTeamId': 16,
 'stats': [{"appliedStats": {'3': 14.56, '4': 16.0, '72': -2.0},
 'appliedTotal': 28.560000000000002,
 'id': '01null',
 'proTeamId': 0,
 'scoringPeriodId': 0,
 'seasonId': 2023,
 'statSourceId': 0,
 'statSplitTypeId': 1,
 'stats': {'0': 44.0,
 '64': 2.0,
 '1': 31.0,
 '65': 1.0,
 '2': 13.0,
 '3': 364.0,
 '4': 4.0,
 '68': 1.0,
 '5': 72.0,
 '69': 1.0,
 '6': 36.0,
 '7': 18.0,
 '8': 14.0,
 '72': 1.0,
 '9': 7.0,
 '73': 1.0,
 '10': 3.0,
 '11': 6.0,
 '12': 3.0,
 '13': 2.0,
 '14': 1.0,
 '15': 1.0,
 '16': 1.0,
 '17': 1.0,
 '210': 1.0,
 '211': 19.0,
 '21': 0.70454545,
 '22': 364.0,
 '156': 1.0,
 '158': 24.0,
 '175': 2.0,
 '176': 1.0}]}],
 'universeId': 2}}}
```

And here's a quick function that takes this dict and returns the data we want (actual score and ESPN id):

```
In [9]:  
def proc_played_player(player):  
    dict_to_return = {}  
    dict_to_return['espn_id'] = player['playerId']  
  
    dict_to_return['actual'] = (player['playerPoolEntry'][  
        'player']['stats'][0]['appliedTotal'])  
    return dict_to_return  
  
In [10]: proc_played_player(cousins_dict)  
Out[10]: {'espn_id': 14880, 'actual': 28.56}
```

This function is doing most of the work. Now we can wrap it in a function that takes a *team* and calls it for *all* the players.

```
In [11]:  
def proc_played_team(team):  
    if 'rosterForMatchupPeriod' in team.keys():  
        return DataFrame([proc_played_player(x) for x in  
                         team['rosterForMatchupPeriod']['entries']])  
    else:  
        return DataFrame()
```

Looking at the JSON, it looks like the “home” team in matchup 2 had multiple players (Mattison and Devonta Smith). Let’s try calling it on this team and make sure it works:

```
In [12]: matchup2 = matchup_list[2]  
  
In [13]: matchup2_home = matchup2['home']  
  
In [14]: proc_played_team(matchup2_away)  
Out[14]:  
  espn_id  actual  
0  4241478    23.1  
1  4048244     4.9
```

This is working. Now we can wrap *this* in a function that operates on a matchup:

```
In [15]:  
def proc_played_matchup(matchup):  
    return pd.concat([proc_played_team(matchup['home']),  
                     proc_played_team(matchup['away'])],  
                     ignore_index=True)  
  
In [16]: proc_played_matchup(matchup0)  
Out[16]:  
    espn_id  actual  
0      14880   28.56  
  
In [17]: proc_played_matchup(matchup2)  
Out[17]:  
    espn_id  actual  
0    4241478   23.1  
1    4048244    4.9
```

So now we can run this on *all* the relevant matchups.

```
In [18]: scores = pd.concat([proc_played_matchup(x) for x in matchup_list])  
  
In [19]: scores  
Out[19]:  
    espn_id  actual  
0      14880   28.56  
0    4241478   23.10  
1    4048244   4.90  
0    4047646   6.90  
1    4259545   27.10  
2    3050478   12.00  
0    4040715   25.22  
1    4262921   24.90  
0    3121023   8.20  
0    4036133   25.60
```

So now we have the actual scores for mid-week guys. Great. Then we just have to link it up with our main roster.

```
In [20]: all_rosters_w_pts = pd.merge(
    all_rosters, scores, how='left')

In [21]: all_rosters_w_pts.head(15)
Out[21]:
   team_position  espn_id      name ...  start  team_id  actual
0            WR1  4262921  Justin Jefferson ...  True     1  24.90
1            WR2    16737      Mike Evans ...  True     1    NaN
2            QB   4040715     Jalen Hurts ...  True     1  25.22
3            RB1  4239996  Travis Etienne Jr. ...  True     1    NaN
4            RB2  4239934        AJ Dillon ...  True     1    NaN
5       RB/WR/TE  3045138     Mike Williams ...  True     1    NaN
6            TE   3123076      David Njoku ...  True     1    NaN
7            BE1  2980453     Jamaal Williams ... False     1    NaN
8            BE2  2576414      Raheem Mostert ... False     1    NaN
9            BE3  4241555     Elijah Mitchell ... False     1    NaN
10           BE4  4569609        Evan Hull ... False     1    NaN
11           BE5  4360078      Alec Pierce ... False     1    NaN
12           BE6  2576623     DeVante Parker ... False     1    NaN
13           BE7   16760      Jimmy Garoppolo ... False     1    NaN
14             K   15683      Justin Tucker ...  True     1    NaN
```

Now we just need to be able to connect it to our Fantasy Math simulation data. We could try merging on `name`, but that won't always work — players' names aren't the same on every platform + there's nicknames, jr's and sr's, duplicates etc.

So instead, we'll do something easier —

— I'll take of it for you behind the scenes. There's a utility function `master_player_lookup` that return a master lookup of everyone's player ids. Like all the other functions, it takes a `token`.

```
In [1] from utilities import (
    LICENSE_KEY, generate_token, master_player_lookup)

In [2]:
token = generate_token(LICENSE_KEY)['token']
fantasymath_players = master_player_lookup(token)
```

Just like we've been doing, let's load the saved snapshot of this table so we all see the same thing:

```
In [3]: fantasymath_players = (
    pd.read_csv('./projects/integration/raw/espn/lookup.csv')

In [4]: fantasymath_players.head()
Out[4]:
      name  player_id  pos  fleaflicker_id  espn_id  ...
0  Tanner McKee        1   QB          17761  4685201  ...
1  Bryce Young        2   QB          17602  4685720  ...
2   Will Levis        3   QB          17623  4361418  ...
3   C.J. Stroud        4   QB          17589  4432577  ...
4 Anthony Richardson       6   QB          17586  4429084  ...
```

Now we can link this up with our `all_rosters` data and we'll be set.

```
In [5]:
all_rosters_w_id = pd.merge(
    all_rosters_w_pts,
    fantasymath_players[['fantasymath_id', 'espn_id']],
    how='left')
```

Remember the fields we said we wanted at the start of this project:

- `fantasymath_id`
- `name`
- `player_position`
- `team_position`
- `start`
- `actual`
- `team_id`.

That's exactly what we have (we also have `espn_id`, which we'll drop). So:

```
In [6]: all_rosters_final = all_rosters_w_id.drop('espn_id', axis=1)

In [7]: all_rosters_final.sample(10)
Out[7]:
      team_position      name  ...  team_id  actual  player_id
51           WR2  Christian Kirk  ...        4     NaN      933
84        RB/WR/TE  Javonte Williams  ...        7     NaN      346
209          WR2  Christian Watson  ...       16     NaN      245
4            RB2      AJ Dillon  ...        1     NaN      520
170          RB2      D Andre Swift  ...       12    27.1      512
62            K    Evan McPherson  ...        4     NaN      435
202            TE      Kyle Pitts  ...       15     NaN      419
89           BE3    Russell Wilson  ...        7     NaN     2088
184          BE6  Raheem Blackshear  ...       13     NaN      289
39            QB    Patrick Mahomes  ...        3     NaN     1091
```

We also said we wanted this in a function called `get_league_rosters`. Let's think about what it needs to take. How about:

- ESPN league id
- player-ESPN id lookup table
- week

It also technically needs the ESPN `swid` and `espn_s2` cookies, but we'll save them in constants vs passing them in.

Let's do it:

```
def get_league_rosters(lookup, league_id, week):
    roster_url = (f'https://lm-api-reads.fantasy.espn.com/apis/v3/games/
                   ffl/seasons/{SEASON}' +
                  f'/segments/0/leagues/{league_id}?view=mRoster')

    roster_json = requests.get(
        roster_url,
        cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()

    all_rosters = pd.concat([process_roster(x) for x in roster_json['teams']
                           ],
                           ignore_index=True)

    # score part
    boxscore_url = (
        f'https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/seasons
          /{SEASON}' +
        f'/segments/0/leagues/{league_id}?view=mBoxscore')
    boxscore_json = requests.get(
        boxscore_url,
        cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()

    matchup_list = [x for x in boxscore_json['schedule'] if
                    x['matchupPeriodId'] == week]
    scores = pd.concat([proc_played_matchup(x) for x in matchup_list])

    all_rosters = pd.merge(all_rosters, scores, how='left')

    all_rosters_w_id = pd.merge(all_rosters,
                                lookup[['player_id', 'espn_id']],
                                how='left').drop('espn_id', axis=1)

    return all_rosters_w_id
```

Now let's make sure it works.

```
In [8]: complete_league_rosters = get_league_rosters(fantasymath_players,
LEAGUE_ID)

In [9]: complete_league_rosters.head(10)
Out[9]:
   team_position           name  ...  actual  player_id
0          WR1  Justin Jefferson  ...    24.90      551
1          WR2        Mike Evans  ...       NaN     1716
2            QB        Jalen Hurts  ...    25.22      498
3          RB1  Travis Etienne Jr.  ...       NaN     343
4          RB2        AJ Dillon  ...       NaN     520
5      RB/WR/TE      Mike Williams  ...       NaN    1129
6            TE        David Njoku  ...       NaN     1167
7          BE1      Jamaal Williams  ...       NaN     1114
8          BE2      Raheem Mostert  ...       NaN    1651
9          BE3      Elijah Mitchell  ...       NaN     353
```

Awesome. This was probably the most complicated piece of the league connection project, and we've knocked it out.

Team Data

Now we need team data, specifically:

- `team_id`
- `owner_id`
- `owner_name`
- `league_id`

It looks like `mTeam` gives us what we need:

```
In [1]:
teams_url = ('https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/
seasons/2021' +
             f'/segments/0/leagues/{LEAGUE_ID}?view=mTeam')

In [2]: teams_json = requests.get(
             teams_url,
             cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()
```

Same thing. Definitely run this and try it out, but for this walkthrough we'll use the saved JSON:

```
In [3]:
with open('./projects/integration/raw/espn/teams.json') as f:
    teams_json = json.load(f)
```

After playing around with this in the browser, it looks like what we need is in two spots: `teams` and `members`

```
In [4]: teams_list = teams_json['teams']

In [5]: members_list = teams_data['members']
```

We'll have to process them both separately, then link them up:

```
def process_team(team):
    dict_to_return = {}
    dict_to_return['team_id'] = team['id']
    dict_to_return['owner_id'] = team['owners'][0]
    return dict_to_return

def process_member(member):
    dict_to_return = {}
    dict_to_return['owner_id'] = member['id']
    dict_to_return['owner_name'] = (
        member['firstName'] + ' ' + member['lastName'][0]).title()
    return dict_to_return
```

Teams:

```
In [6]: DataFrame([process_team(team) for team in teams_list])
Out[6]:
   team_id          owner_id
0      1 {1335E3DB-3C91-49ED-8EE4-79BB6E814DED}
1      2 {7101984C-BFEA-4C83-BA81-C40148286E69}
2      3 {A7445C92-658A-450D-9E6C-B953F6AACD73}
3      4 {6499F9DA-F6CF-4880-8D1D-6B1B15FFE28D}
4      6 {CDDFB7CB-F280-431E-9660-C02982E8FE78}
5      7 {3FB29269-A700-4805-B365-A11DF3ED06E9}
6      8 {836103D1-FE2F-41A7-BAD4-D21C82D15421}
7      9 {CADD9070-6903-4D1A-A294-AF037C3E3DB7}
8     10 {66468385-C6F1-4533-93D1-778D80A23294}
9     11 {3636C950-1257-4B01-A27D-41D9974D62F7}
10    12 {5A7DB0EC-EA68-4CB7-BBA0-51DD90C06CA4}
11    13 {28B9AF9C-C701-40CD-B9AF-9CC70170CD79}
12    15 {03E01B80-2233-4830-AE15-D912ABB2F843}
13    16 {09C69E89-E8F8-4B7B-898C-44D3FCB33B7C}
```

And members:

```
In [7]: DataFrame([process_member(member) for member in members_list])
Out[7]:
   owner_id  owner_name
0 {03E01B80-2233-4830-AE15-D912ABB2F843}  David R
1 {09C69E89-E8F8-4B7B-898C-44D3FCB33B7C}  Madison C
2 {1335E3DB-3C91-49ED-8EE4-79BB6E814DED}  Isabella H
3 {28B9AF9C-C701-40CD-B9AF-9CC70170CD79}  Stacey C
4 {3636C950-1257-4B01-A27D-41D9974D62F7}  Teresa W
5 {3FB29269-A700-4805-B365-A11DF3ED06E9}  Jennifer W
6 {5A7DB0EC-EA68-4CB7-BBA0-51DD90C06CA4}  Michael L
7 {6499F9DA-F6CF-4880-8D1D-6B1B15FFE28D}  Larry W
8 {66468385-C6F1-4533-93D1-778D80A23294}  Cecil H
9 {7101984C-BFEA-4C83-BA81-C40148286E69}  Jeffrey H
10 {836103D1-FE2F-41A7-BAD4-D21C82D15421}  Misty H
11 {A7445C92-658A-450D-9E6C-B953F6AACD73}  Ashley R
12 {CADD9070-6903-4D1A-A294-AF037C3E3DB7}  Penney R
13 {CDDFB7CB-F280-431E-9660-C02982E8FE78}  Sara R
```

So here's what we need to do:

1. Hit the teams endpoint.
2. Since the `teams` (which has `team_id`, `owner_id`) and `members` (which has `owner_name`, `owner_id`) data are in separate spots, process them separately, then `merge` on `owner_id`.
3. Add league id.
4. Done.

We'll do that all in a function, remember this one needs to be called `get_teams_in_league`.

```
def get_teams_in_league(league_id):
    teams_url = (
        'https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/seasons'
        '/2021' +
        f'/segments/0/leagues/{league_id}?view=mTeam')

    # teams_json = requests.get(
    #     teams_url, cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()
    with open('./projects/integration/raw/espn/teams.json') as f:
        teams_json = json.load(f)

    teams_list = teams_json['teams']
    members_list = teams_json['members']

    teams_df = DataFrame([process_team(team) for team in teams_list])
    member_df = DataFrame([process_member(member) for member in
                           members_list])

    comb = pd.merge(teams_df, member_df)
    comb['league_id'] = league_id

    return comb
```

Let's call it and make sure it works:

In [8]: `league_teams = get_teams_in_league(LEAGUE_ID)`

In [9]: `league_teams`

Out[9]:

	team_id	owner_id	owner_name	league_id
0	1 {1335E3DB-3C91-49ED-8EE4-79BB6E814DED}	Isabella H	395209553	
1	2 {7101984C-BFEA-4C83-BA81-C40148286E69}	Jeffrey H	395209553	
2	3 {A7445C92-658A-450D-9E6C-B953F6AACD73}	Ashley R	395209553	
3	4 {6499F9DA-F6CF-4880-8D1D-6B1B15FFE28D}	Larry W	395209553	
4	6 {CDDFB7CB-F280-431E-9660-C02982E8FE78}	Sara R	395209553	
5	7 {3FB29269-A700-4805-B365-A11DF3ED06E9}	Jennifer W	395209553	
6	8 {836103D1-FE2F-41A7-BAD4-D21C82D15421}	Misty H	395209553	
7	9 {CADD9070-6903-4D1A-A294-AF037C3E3DB7}	Penney R	395209553	
8	10 {66468385-C6F1-4533-93D1-778D80A23294}	Cecil H	395209553	
9	11 {3636C950-1257-4B01-A27D-41D9974D62F7}	Teresa W	395209553	
10	12 {5A7DB0EC-EA68-4CB7-BBA0-51DD90C06CA4}	Michael L	395209553	
11	13 {28B9AF9C-C701-40CD-B9AF-9CC70170CD79}	Stacey C	395209553	
12	15 {03E01B80-2233-4830-AE15-D912ABB2F843}	David R	395209553	
13	16 {09C69E89-E8F8-4B7B-898C-44D3FCB33B7C}	Madison C	395209553	

Schedule Info

Next we need schedule. Looking at Steven Morse's list of remaining endpoints:

- mBoxscore
- mSettings
- kona_player_info
- player_wl
- mSchedule

The mSchedule endpoint is the obvious one to try. Unfortunately, often APIs are *not* obvious, and it's actually in mBoxscore.

The good news is that it's pretty straightforward.

```
In [1]:  
schedule_url = (  
    f'https://lm-api-reads.fantasy.espn.com/apis/v3/games/ffl/seasons/{  
        SEASON}' +  
    f'/segments/0/leagues/{LEAGUE_ID}?view=mBoxscore')  
  
In [2]:  
schedule_json = requests.get(schedule_url,  
                             cookies={'swid': SWID, 'espn_s2': ESPN_S2}  
                           ).json()
```

(Again, the saved version:)

```
In [3]:  
with open('./projects/integration/raw/espn/schedule.json') as f:  
    schedule_json = json.load(f)
```

As usual, let's get the list of matchups + a specific one to look at:

```
In [4]: matchup_list = schedule_json['schedule']  
  
In [5]: matchup0 = matchup_list[0]
```

This has a *lot* of data, but everything we need is basically:

```
In [6]: matchup0['id'] # matchup_id  
Out[6]: 1  
  
In [7]: matchup0['home']['teamId'] # "home" team_id  
Out[7]: 7  
  
In [8]: matchup0['away']['teamId'] # "away" team_id  
Out[8]: 8  
  
In [9]: matchup0['matchupPeriodId'] # week  
Out[9]: 1
```

So let's put that in a function.

```
def process_matchup(matchup):
    dict_to_return = {}

    dict_to_return['matchup_id'] = matchup['id'] # matchup_id
    dict_to_return['home_id'] = matchup['home']['teamId'] # "home"
        team_id
    dict_to_return['away_id'] = matchup['away']['teamId'] # "away"
        team_id
    dict_to_return['week'] = matchup['matchupPeriodId'] # week

    return dict_to_return
```

And do our usual call it on everything + put it in a DataFrame:

```
In [10]: matchup_df = DataFrame([process_matchup(matchup) for matchup in
                                matchup_list])

In [11]: matchup_df.head(10)
Out[11]:
   matchup_id  home_id  away_id  week
0            1        3       10      1
1            2        7        8      1
2            3       15       13      1
3            4        2        4      1
4            5       16       12      1
5            6       11        1      1
6            7        9        6      1
7            8       10        8      2
8            9       13        3      2
9           10        4        7      2
```

I think this is pretty much what we want, but let's double check with our list at the beginning of the chapter.

“The `schedule` table includes: `team1_id`, `team2_id`, `matchup_id`, `season`, `week` and `league_id` columns.”

So we need to add `league_id` and `season`, and rename `home` and `away` to `team1` and `team2`. Why this last part? Because while it's nice that ESPN assigns a home and away team every matchup, not every hosting platform does this. We don't want to write multiple versions of analysis code depending on whether we're analyzing ESPN or Yahoo leagues.

Making these changes and putting them in our function `get_league_schedule`:

```
def get_league_schedule(league_id):
    schedule_url = f'https://lm-api-reads.fantasy.espn.com/apis/v3/games/
        ffl/seasons/{SEASON}/segments/0/leagues/{LEAGUE_ID}?view=mBoxscore'

    schedule_json = requests.get(
        schedule_url,
        cookies={'swid': SWID, 'espn_s2': ESPN_S2}).json()

    matchup_list = schedule_json['schedule']

    matchup_df = DataFrame([process_matchup(matchup) for matchup in
                           matchup_list])
    matchup_df['league_id'] = league_id
    matchup_df['season'] = SEASON
    matchup_df.rename(columns={'home_id': 'team1_id', 'away_id': 'team2_id'},
                      inplace=True)
    return matchup_df
```

Making sure it works:

```
In [12]: league_schedule = get_league_schedule(**ESPN_PARAMETERS)
```

```
In [13]: league_schedule.head()
```

```
Out[13]:
```

	matchup_id	team1_id	team2_id	week	league_id	season
0	1	3	10	1	395209553	2023
1	2	7	8	1	395209553	2023
2	3	15	13	1	395209553	2023
3	4	2	4	1	395209553	2023
4	5	16	12	1	395209553	2023

Wrap Up

To wrap up we'll get rid of all but the final versions of the functions we wrote above. I've put my final cleaned up version in [./hosts/espn.py](#)

Like we wanted, this file has:

- `get_teams_in_league`
- `get_league_schedule`
- `get_league_rosters`

These are functions that are meant to be called outside this file.

The other functions (e.g. `process_player3`) are NOT meant to be called outside this file. They're "helper" functions, `process_player3` is used inside `get_league_rosters`, no one else should ever have to use it.

For these helper functions, I kept the final versions (`process_player3` vs `process_player2` or `1`), dropped the number (just `process_player`) then added an underscore to the front (e.g. `_process_player`).

Making helper functions start with `_` is a python convention that lets people reading your code know they shouldn't have to use this outside your module.

Feel free to explore these files or work through any other league host integrations you need. When you're ready to pick up with rest of the book head over to [Saving League Data to a Database](#).

Fleaflicker Integration

Note: this section covers how to get data from leagues hosted on Fleaflicker. Skip this and find the relevant section (Yahoo, ESPN, Sleeper) if your league is hosted on something else.

To get what we need from Fleaflicker you need to know two things. Your:

- league id
- team id

You can get these by going to your main team page:

<https://www.fleaflicker.com/nfl/leagues/316893/teams/1605156>

Here, my league id is 316893 and my team id is 1605156.

Fleaflicker has a public API, which appears to power their site and will also give us everything we need. The documentation is here: <https://www.fleaflicker.com/api-docs/index.html>

Our approach will be:

1. Think about the data we want (rosters, team information, etc).
2. Try one of the endpoints described in the documentation.
3. Figure out if (and where) it has the information we need.
4. Get that information out using Python.

Remember, all of these APIs return JSON, which are nested dicts and lists of dicts and lists. **Almost always, we'll be manipulating these data structures until we have list of Python dictionaries that all have the same keys. Then we'll pass these to DataFrame.**

Roster Data

Let's start with roster data. On the API documentation, if you look on the sidebar, under PATHS, you'll see a Team roster path, which seems like a good place to start.

The documentation says it takes a few parameters. By trial an error, I've found you really only need league and team id. So this path API endpoint would return data on my team:

<https://www.fleaflicker.com/api/FetchRoster?leagueId=316893&teamId=1605156>

If you look at this URL, you should see some data. But so that we're seeing the same thing, let's look at some specific data I saved from the Fleaflicker API between Thursday and Sunday, Week 2, 2023.

This is one of the files that came with this book:

`./projects/integration/raw/fleaflicker/roster.json`

Find it in your file explorer and double click on it. It should open up in the browser. You'll see:

2025 Fantasy Football Developer Kit

```
JSON Raw Data Headers
JSON Copy Collapse All Expand All Filter JSON

▼ groups:
  ▼ 0:
    group: "START"
    ▼ slots:
      ▼ 0:
        ▼ position:
          label: "QB"
          group: "START"
        ▼ eligibility:
          0: "QB"
          min: 1
          max: 4
          start: 1
        ▼ colors:
          0: "DRAFT_BOARD_RED"
      ▼ leaguePlayer:
        ▼ proPlayer:
          id: 15581
          nameFull: "Jalen Hurts"
          nameShort: "J. Hurts"
          proTeamAbbreviation: "PHI"
          position: "QB"
        ▼ headshotUrl: "https://d26bvpbynng29h.cloudfront.net/nfl/15581.png"
        nflByeWeek: 14
      ▼ news:
        ▶ 0: {...}
        nameFirst: "Jalen"
        nameLast: "Hurts"
      ▼ proTeam:
        abbreviation: "PHI"
        location: "Philadelphia"
        name: "Eagles"
      ▼ positionEligibility:
        0: "QB"
    ▼ requestedGames:
      ▼ 0:
        ▶ game: {...}
        ▶ stats: [...]
        ▶ statsProjected: [...]
        ▶ pointsProjected: {...}
        ▶ period: {...}
        ▶ ranks: {...}
      ▼ viewingProjectedPoints:
        value: 20.99
        formatted: "20.99"
      ▶ viewingActualStats: ...
```

Figure 0.5: Fleaflicker Roster JSON

Looking at this data (remember you should have a JSON viewer installed) we can see the roster data we want is in `groups`. Further, `groups` is subdivided into two or three parts (depending if your league has IR spots): 0 is our starters, 1 is our bench.

Both groups have a list of “slots”. In this league we start 1 QB, 2 RBs, 2 WRs, a FLEX, 1 TE, 1 K, 1 DST, so that’s 9 positions. And there are 9 slots in group 0.

Drilling deeper, each slot has: `position`, `leaguePlayer` and `swappableWith` fields. The two we care about are `leaguePlayer` (specifically the `proPlayer` field) and `position`, which tells us *where* in our lineup this player is slotted.

Finally this gets us into some real information. Here, my starting QB is Jalen Hurts, and we can see his fleaflicker id is 15581. This is what we need.

Again, this is *really* nested. There’s a lot we have to ignore. That’s fine.

But finally, now that we’ve familiarized ourselves with this endpoint in the browser, let’s get the data in Python. We’ll pick up right below our imports in:

```
./code/integration/fleaflicker_working.py.
```

As usual, I like to make constants uppercase:

```
In [1]:  
LEAGUE_ID = 316893  
TEAM_ID = 1605156  
WEEK = 2
```

We can use these to make our team url:

```
In [2]:  
roster_url = ('https://www.fleaflicker.com/api/FetchRoster?' +  
    f'leagueId={LEAGUE_ID}&teamId={TEAM_ID}' )
```

Next we’ll visit this URL via Python with the `requests` library and turn the raw data we get back into the nested dicts and lists Python can work with.

```
In [3]: roster_json = requests.get(roster_url).json()
```

Saved Data

This is how you get live data for your league and team, but for this walkthrough I recommend using the snapshot I’ve saved. We’ll set a boolean `USED_SAVED_DATA` that controls whether we do this from now on:

```
In [4]: USE_SAVED_DATA = True
```

Then (with this set to `True`) can load our saved data in Python like this:

```
In [4]:  
if USE_SAVED_DATA:  
    with open('./projects/integration/raw/fleaflicker/roster.json') as f:  
        roster_json = json.load(f)
```

To make it easier to follow along, I'll included a `USE_SAVED_DATA` option for rest of the data we get in this chapter. If you want to try it with your own league, you can set it to `False`.

Things and Collections of Things

My approach to getting data like this: get the collection of things ("slots" in this case) I want, usually in a list. Then, pick out a specific example and look at it more.

```
In [5]: list_of_starter_slots = roster_json['groups'][0]['slots']  
  
In [6]: list_of_bench_slots = roster_json['groups'][1]['slots']  
  
In [7]: starter_slot0 = list_of_starter_slots[0]
```

In the browser, it looks like the info we wanted was mostly in the `proPlayer` field:

```
In [8]: starter_slot0['leaguePlayer']['proPlayer']  
Out[8]:  
{'id': 12894,  
 'nameFull': 'Patrick Mahomes',  
 'nameShort': 'P. Mahomes',  
 'proTeamAbbreviation': 'KC',  
 'position': 'QB',  
 'headshotUrl': 'https://d26bvpbynxg29h.cloudfront.net/nfl/12894.png',  
 'nflByeWeek': 10,  
 ...  
 'nameFirst': 'Patrick',  
 'nameLast': 'Mahomes',  
 'proTeam': {'abbreviation': 'KC',  
 'location': 'Kansas City',  
 'name': 'Chiefs'},  
 'positionEligibility': ['QB']}
```

Remember what we need to get here for each player:

1. the fleaflicker player id
2. whether we're starting them and the position they're in if we are (e.g. if it's an RB, are they in the RB spot or flex)
3. the player's position and name

Within each slot, it looks like that information is in two places: `proPlayer` and also `position`. Let's build a function to get it:

```
In [9]:  
def process_player1(slot):  
    fleaflicker_player_dict = slot['leaguePlayer']['proPlayer']  
    fleaflicker_position_dict = slot['position']  
  
    dict_to_return = {}  
    dict_to_return['name'] = fleaflicker_player_dict['nameFull']  
    dict_to_return['player_position'] = (fleaflicker_player_dict['position'])  
    dict_to_return['fleaflicker_id'] = fleaflicker_player_dict['id']  
  
    dict_to_return['team_position'] = fleaflicker_position_dict['label']  
  
    return dict_to_return
```

And try it out:

```
In [10]: process_player1(starter_slot0)  
Out[10]:  
{'name': 'Patrick Mahomes',  
 'player_position': 'QB',  
 'fleaflicker_id': 12894,  
 'team_position': 'QB'}
```

Nice. Now let's run this on every slot in our list of starters and bench.

Aside: when I first ran this I saw:

```
In [11]: [process_player1(player) for player in list_of_starter_slots]  
In [12]: [process_player1(player) for player in list_of_bench_slots]  
...  
KeyError: 'leaguePlayer'
```

It's no longer an issue with the snapshot I've included here, but when I first ran this I got this error. It turns out that sometimes — e.g. if you have an open spot on your roster — a `slot` doesn't have a `leaguePlayer` field. And so calling:

```
slot['leaguePlayer']['proPlayer']
```

inside the function throws an error. This may or may not come up for you, but either way let's tweak `process_player` to make sure keys exist before we pull data out of them:

```
def process_player2(slot):
    dict_to_return = {}

    if 'leaguePlayer' in slot.keys():
        fleaflicker_player_dict = slot['leaguePlayer']['proPlayer']

        dict_to_return['name'] = fleaflicker_player_dict['nameFull']
        dict_to_return['player_position'] = (
            fleaflicker_player_dict['position'])
        dict_to_return['fleaflicker_id'] = fleaflicker_player_dict['id']

    if 'position' in slot.keys():
        fleaflicker_position_dict = slot['position']

        dict_to_return['team_position'] = (
            fleaflicker_position_dict['label'])

    return dict_to_return
```

This works:

```
In [13]: [process_player2(x) for x in list_of_starter_slots]
Out[13]:
[{'name': 'Patrick Mahomes',
 'player_position': 'QB',
 'fleaflicker_id': 12894,
 'team_position': 'QB'},
 {'name': 'Rhamondre Stevenson',
 'player_position': 'RB',
 'fleaflicker_id': 16339,
 'team_position': 'RB'},
 ...]
```

Remember, when you have a list of dictionaries that have the same fields — which is what we have here (all the players have `name`, `player_position`, `team_position`, `fleaflicker_id` fields) — you should put them in a DataFrame ASAP.

```
In [14]: starter_df1 = DataFrame([process_player2(x) for x in
                                list_of_starter_slots])

In [15]: starter_df1
Out[15]:
```

	name	player_position	fleaflicker_id	team_position
0	Patrick Mahomes	QB	12894	QB
1	Rhamondre Stevenson	RB	16339	RB
2	Saquon Barkley	RB	13778	RB
3	James Conner	RB	12951	RB/WR/TE
4	DeVonta Smith	WR	16253	WR
5	Chris Godwin	WR	12926	WR
6	Darren Waller	TE	11261	TE
7	Younghoe Koo	K	13324	K
8	Dallas Cowboys	D/ST	2336	D/ST

Cool.

What else? Well, this snapshot of data we're using is from Friday morning, Week 2, 2023 — right after PHI-MIN played Thursday night. In that case DeVonta Smith played and has an actual scores.

Looking at the JSON, those point values appear to be in the `requestedGames` field of `leaguePlayer`. They're further down in a field called, `pointsActual`, which is NOT in the data for players who haven't played yet. Let's modify our function to return this info if it exists:

```

def process_player3(slot):
    dict_to_return = {}

    if 'leaguePlayer' in slot.keys():
        fleaflicker_player_dict = slot['leaguePlayer']['proPlayer']

        dict_to_return['name'] = fleaflicker_player_dict['nameFull']
        dict_to_return['player_position'] = (
            fleaflicker_player_dict['position'])
        dict_to_return['fleaflicker_id'] = fleaflicker_player_dict['id']

    if 'requestedGames' in slot['leaguePlayer']:
        game = slot['leaguePlayer']['requestedGames'][0]
        if 'pointsActual' in game:
            if 'value' in game['pointsActual']:
                dict_to_return['actual'] = (
                    game['pointsActual']['value'])

    if 'position' in slot.keys():
        fleaflicker_position_dict = slot['position']

        dict_to_return['team_position'] = (
            fleaflicker_position_dict['label'])

    return dict_to_return

```

Running it on everyone again:

```

In [16]: starter_df1 = DataFrame
          ([process_player3(x) for x in list_of_starter_slots])

In [17]: starter_df1
Out[17]:
      name player_position ... team_position actual
0   Patrick Mahomes     QB ...           QB   NaN
1 Rhamondre Stevenson     RB ...           RB   NaN
2   Saquon Barkley     RB ...           RB   NaN
3   James Conner       RB ...  RB/WR/TE   NaN
4   DeVonta Smith      WR ...           WR  19.1
5   Chris Godwin       WR ...           WR   NaN
6   Darren Waller      TE ...           TE   NaN
7   Younghoe Koo        K ...           K   NaN
8   Dallas Cowboys    D/ST ...  D/ST   NaN

```

Note the 19.1 for DeVonta. Awesome.

This looks pretty good, but at some point we're probably going to want to refer to players on our roster by position, in which case duplicate `team_position` might be a problem. It'd be better to be able to

refer to our RB1, RB2 etc.

Let's add that. As always let's start with a specific example, say WRs:

```
In [18]: wrs = starter_df_raw.query("team_position == 'WR'")  
  
In [19]: wrs  
Out[19]:
```

	name	player_position	fleaflicker_id	team_position	actual
4	DeVonta Smith	WR	16253	WR	19.1
5	Chris Godwin	WR	12926	WR	NaN

So we want `team_position` to say WR1, WR2 instead of just WR, WR.

The easiest way is probably to make a column with 1, 2 then stick them together.

```
In [20]: suffix = Series(range(1, len(wrs) + 1), index=wrs.index)  
  
In [21]: suffix  
Out[21]:
```

4	1
5	2

The key is we're making this column have the same index as `wrs`, that way we can do this:

```
In [22]: wrs['team_position'] + suffix.astype(str)  
Out[22]:
```

4	WR1
5	WR2

Which is what we want. Now let's put this in a function:

```
In [22]:  
def add_pos_suffix(df_subset):  
    # only add if more than one position -- want just K, not K1  
    if len(df_subset) > 1:  
        suffix = Series(range(1, len(df_subset) + 1), index=df_subset.  
                        index)  
  
        df_subset['team_position'] = df_subset['team_position'] + suffix.  
                                    astype(str)  
    return df_subset
```

and apply it to every position in the starter DataFrame.

```
In [23]:  
starter_df2 = pd.concat([  
    add_pos_suffix(starter_df1.query(f"team_position == '{x}'"))  
    for x in starter_df1['team_position'].unique()])  
  
In [24]: starter_df2  
Out[24]:
```

		name	player_position	...	team_position	actual
0		Patrick Mahomes	QB	...	QB	Nan
1	Rhamondre Stevenson		RB	...	RB1	Nan
2	Saquon Barkley		RB	...	RB2	Nan
3	James Conner		RB	...	RB/WR/TE	Nan
4	DeVonta Smith		WR	...	WR1	19.1
5	Chris Godwin		WR	...	WR2	Nan
6	Darren Waller		TE	...	TE	Nan
7	Younghoe Koo		K	...	K	Nan
8	Dallas Cowboys		D/ST	...	D/ST	Nan

Cool. Now let's make a bench version, identify both DataFrames, and stick them together.

```
In [25]: bench_df = DataFrame([process_player2(x) for x in  
                           list_of_bench_slots])  
In [26]: starter_df2['start'] = True  
In [27]: bench_df['start'] = False  
  
In [28]: roster_df = pd.concat([starter_df2, bench_df], ignore_index=True)  
  
In [29]: roster_df  
Out[29]:
```

		name	player_position	...	team_position	actual	start
0		Patrick Mahomes	QB	...	QB	NaN	True
1	Rhamondre Stevenson		RB	...	RB1	NaN	True
2	Saquon Barkley		RB	...	RB2	NaN	True
3	James Conner		RB	...	RB/WR/TE	NaN	True
4	DeVonta Smith		WR	...	WR1	19.1	True
5	Chris Godwin		WR	...	WR2	NaN	True
6	Darren Waller		TE	...	TE	NaN	True
7	Younghoe Koo		K	...	K	NaN	True
8	Dallas Cowboys		D/ST	...	D/ST	NaN	True
9	Brock Purdy		QB	...	BN	NaN	False
10	Kyren Williams		RB	...	BN	NaN	False
11	Joshua Kelley		RB	...	BN	NaN	False
12	Rashee Rice		WR	...	BN	NaN	False
13	Marvin Mims		WR	...	BN	NaN	False
14	Hunter Henry		TE	...	BN	NaN	False
15	Sam LaPorta		TE	...	BN	NaN	False

This is really close to what we want. We just need my fleaflicker team id, and the corresponding player ids for Fantasy Math.

Adding team id is easy:

```
In [30]: roster_df['team_id'] = TEAM_ID
```

Now we just need to be able to connect it to our Fantasy Math simulation data. We could try merging on `name`, but that won't always work — players' names aren't the same on every platform + there's nicknames, jr's and sr's, duplicates etc.

So instead, we'll do something easier —

— I'll take of it for you behind the scenes. There's a utility function `master_player_lookup` that return a master lookup of everyone's player ids. Like all the other functions, it takes a `token`.

```
In [31]: from utilities import (LICENSE_KEY, generate_token,
                               master_player_lookup)
```

```
In [32]: token = generate_token(LICENSE_KEY)['token']
fantasymath_players = master_player_lookup(token)
```

As with the other data, I've saved a snapshot.

Here's what it looks like:

```
In [33]: fantasymath_players.head()
Out[33]:
   player_id  pos  fleaflicker_id  espn_id  yahoo_id  sleeper_id
0          1    QB        17761  4685201      40220       9230
1          2    QB        17602  4685720      40029       9228
2          3    QB        17623  4361418      40068       9999
3          4    QB        17589  4432577      40030       9758
4          6    QB        17586  4429084      40040       9229
```

Now we can link this up with the roster from above and we'll be set.

```
In [34]: roster_df_w_id = pd.merge(
    roster_df, fantasymath_players[['player_id', 'fleaflicker_id']],
    how='left')
```

Remember the fields we said we wanted at the start of this project:

- `player_id`
- `name`
- `player_position`
- `team_position`
- `start`

- `actual`
- `team_id`.

That's exactly what we have (we also have `fleaflicker_id`, which we'll drop).

The only problem is we said we wanted rosters for *all* teams in league, not just one. And we want to be able to get all of them with a `league_id`, we don't want to have to pass the `team_id` in every time.

In the next section we'll hit another endpoint to get info on every team in a league, so this part will have to wait.

In the meantime though we can put all this in a function. To get the above we basically need the following:

- Fleaflicker league and team ids
- our player-Fleaflicker id lookup table

Let's do it:

```
def get_team_roster(team_id, league_id, lookup):
    roster_url = ('https://www.fleaflicker.com/api/FetchRoster?' +
                  f'leagueId={league_id}&teamId={team_id}')

    with open('./projects/integration/raw/fleaflicker/roster.json') as f:
        roster_json = json.load(f)

    starter_slots = roster_json['groups'][0]['slots']
    bench_slots = roster_json['groups'][1]['slots']

    starter_df = DataFrame([process_player3(x) for x in starter_slots])
    bench_df = DataFrame([process_player3(x) for x in bench_slots])

    starter_df['start'] = True
    bench_df['start'] = False

    team_df = pd.concat([starter_df, bench_df], ignore_index=True)
    team_df['team_id'] = team_id

    team_df_w_id = pd.merge(team_df,
                           lookup[['fantasymath_id', 'fleaflicker_id']],
                           how='left').drop('fleaflicker_id', axis=1)

    if 'actual' not in team_df_w_id.columns:
        team_df_w_id['actual'] = np.nan

    return team_df_w_id
```

Now let's make sure it works.

```
In [35]: my_roster = get_team_roster(TEAM_ID, LEAGUE_ID,  
fantasymath_players)
```

```
In [36]: my_roster
```

```
Out[36]:
```

	name	player_position	...	team_id	player_id
0	Patrick Mahomes	QB	...	1605156	1091
1	Rhamondre Stevenson	RB	...	1605156	349
2	Saquon Barkley	RB	...	1605156	904
3	James Conner	RB	...	1605156	1121
4	DeVonta Smith	WR	...	1605156	372
5	Chris Godwin	WR	...	1605156	1138
6	Darren Waller	TE	...	1605156	1574
7	Younghoe Koo	K	...	1605156	1228
8	Dallas Cowboys	D/ST	...	1605156	5159
9	Brock Purdy	QB	...	1605156	156
10	Kyren Williams	RB	...	1605156	167
11	Joshua Kelley	RB	...	1605156	522
12	Rashee Rice	WR	...	1605156	48
13	Marvin Mims	WR	...	1605156	43
14	Hunter Henry	TE	...	1605156	1352
15	Sam LaPorta	TE	...	1605156	68

Awesome. This was probably the most complicated piece of the league connection project, and we've knocked it out.

We're not quite finished yet though, let's grab our team data so we can get complete league rosters.

Team Data

Now we need team data, specifically:

- `team_id`
- `owner_id`
- `owner_name`
- `league_id`

Looking at the documentation, team data is available in the League standings endpoint.

```
In [1]: teams_url = (  
    'https://www.fleaflicker.com/api/FetchLeagueStandings?' +  
    f'leagueId={LEAGUE_ID}')
```

```
In [2]: teams_json = requests.get(teams_url).json()
```

Again, that's how we'd get the live data, but for this walkthrough we'll load the data I saved mid Week 1:

```
In [3]:  
with open('./projects/integration/raw/fleaflicker/teams.json') as f:  
    teams_json = json.load(f)
```

Opening it up in your browser reveals that the team info is all inside divisions. So let's start with a single division:

```
In [4]: division0 = teams_json['divisions'][0]
```

And follow up with single team from that division:

```
In [5]: team0_division0 = division0['teams'][0]  
  
In [6]: team0_division0  
Out[6]:  
{'id': 1747268,  
 'name': 'Fast chicken',  
 'logoUrl': 'https://s3.amazonaws.com/fleaflicker/t1747268_0_150x150.jpg',  
 'recordOverall': {'wins': 1,  
 'winPercentage': {'value': 1.0, 'formatted': '1.000'},  
 'rank': 1,  
 'formatted': '1-0'},  
 'recordDivision': {'wins': 1,  
 'winPercentage': {'value': 1.0, 'formatted': '1.000'},  
 'rank': 1,  
 'formatted': '1-0'},  
 'recordPostseason': {'winPercentage': {'formatted': '.000'}},  
 'rank': 1,  
 'formatted': '0-0'},  
 'pointsFor': {'value': 119.3, 'formatted': '119.3'},  
 'pointsAgainst': {'value': 57.55, 'formatted': '57.55'},  
 'streak': {'value': 1.0, 'formatted': 'W1'},  
 'waiverPosition': 5,  
 'owners': [{"id": 2075108,  
 'displayName': 'CalBrusda',  
 'lastSeen': '1694797192000',  
 'initials': 'C',  
 'lastSeenIso': '2023-09-15T16:59:52Z"}],  
 'newItemCounts': {'activityUnread': 1},  
 'initials': 'FC'}
```

I can see two parts we really care about: the team id and owner display name. Let's build a function to get them out.

```
In [7]:  
def process_team(team):  
    dict_to_return = {}  
  
    dict_to_return['team_id'] = team['id']  
    dict_to_return['owner_id'] = team['owners'][0]['id']  
    dict_to_return['owner_name'] = team['owners'][0]['displayName']  
  
    return dict_to_return  
  
In [8]: process_team(team0_division0)  
Out[8]: {'team_id': 1747268, 'owner_id': 2075108, 'owner_name': 'CalBrusda'}  

```

Let's work our way up and make a function that returns all the teams given some division:

```
In [8]:  
def teams_from_div(division):  
    return DataFrame([process_team(x) for x in division['teams']])  
  
In [9]: teams_from_div(division0)  
Out[9]:  
   team_id  owner_id  owner_name  
0  1747268    2075108  CalBrusda  
1  1605152    1328114  WesHeroux  
2  1747266    2075011    Meat11  
3  1605155     103206    Brusda
```

Nice. Now let's work our way up further to the league level.

```
In [10]:  
def divs_from_league(divisions):  
    return pd.concat([teams_from_div(division) for division in divisions],  
                    ignore_index=True)  
  
In [11]: divs_from_league(teams_json['divisions'])  
Out[11]:  
   team_id  owner_id  owner_name  
0  1747268    2075108  CalBrusda  
1  1605152    1328114  WesHeroux  
2  1747266    2075011    Meat11  
3  1605155     103206    Brusda  
4  1605147    1319336  carnsc815  
5  1605156     138510    nbraun  
6  1664196    1471474  MegRyan0113  
7  1603352    1316270    UnkleJim  
8  1605157    1324792  JBKBDomination  
9  1605148    1319991  Lzrlightshow  
10 1605151    1319571    Edmundo  
11 1605154    1319436    ccarns
```

Almost what we want, just need league id. Let's put it in a function. Remember it needs to be called `get_teams_in_league`.

```
In [12]:  
def get_teams_in_league(league_id):  
    teams_url = ('https://www.fleaflicker.com/api/FetchLeagueStandings?' +  
                 f'leagueId={league_id}')  
  
    # teams_json = requests.get(teams_url).json()  
    with open('./projects/integration/raw/fleaflicker/teams.json') as f:  
        teams_json = json.load(f)  
  
    teams_df = divs_from_league(teams_json['divisions'])  
    teams_df['league_id'] = league_id  
    return teams_df  
  
In [13]: league_teams = get_teams_in_league(LEAGUE_ID)  
  
In [14]: league_teams  
Out[14]:
```

	team_id	owner_id	owner_name	league_id
0	1747268	2075108	CalBrusda	316893
1	1605152	1328114	WesHeroux	316893
2	1747266	2075011	Meat11	316893
3	1605155	103206	Brusda	316893
4	1605147	1319336	carnsc815	316893
5	1605156	138510	nbraun	316893
6	1664196	1471474	MegRyan0113	316893
7	1603352	1316270	UnkleJim	316893
8	1605157	1324792	JBKBDomination	316893
9	1605148	1319991	Lzrlightshow	316893
10	1605151	1319571	Edmundo	316893
11	1605154	1319436	ccarns	316893

Ok. So the team portion is done. We can also combine it with our `get_team_roster` function above for `get_league_rosters`, which is what we wanted. It's straightforward:

```
def get_league_rosters(lookup, league_id):  
    teams = get_teams_in_league(league_id)  
  
    league_rosters = pd.concat(  
        [get_team_roster(x, league_id, lookup) for x in teams['team_id']],  
        ignore_index=True)  
    return league_rosters
```

Let's test it out:

```
In [15]: league_rosters = get_league_rosters(fantasymath_players,
LEAGUE_ID)

In [16]: league_rosters.sample(20)
Out[16]:
```

		name	player_position	...	team_id	player_id
9		Justin Fields	QB	...	1747268	328.0
177		Derrick Henry	RB	...	1605154	1303.0
58		Sean Tucker	RB	...	1605155	20.0
104	Cincinnati Bengals		D/ST	...	1664196	5157.0
169		Derek Carr	QB	...	1605151	1689.0
83		James Conner	RB	...	1605156	1121.0
78		Tyler Boyd	WR	...	1605147	1327.0
175		Cole Kmet	TE	...	1605151	582.0
49	Javonte Williams		RB	...	1605155	346.0
23		Matt Gay	K	...	1605152	776.0
92		Rashee Rice	WR	...	1605156	48.0
51		Breece Hall	RB	...	1605155	165.0
7	Daniel Carlson		K	...	1747268	981.0
123		Jaylen Warren	RB	...	1603352	199.0
74		Cam Akers	RB	...	1605147	514.0
16		Daniel Jones	QB	...	1605152	677.0
165		George Pickens	WR	...	1605151	219.0
159		Dan Arnold	TE	...	1605148	1273.0
99	Tyler Allgeier		RB	...	1664196	169.0
35		Jaylen Waddle	WR	...	1747266	374.0

Perfect.

Schedule Info

The last thing we need is the schedule.

Looking at the documentation, the FetchLeagueScoreboard endpoint seems promising. This endpoint takes a week argument and returns a list of 6 games (12 teams in league so makes sense).

Getting it in Python:

```
In [1]:
schedule_url = (
    'https://www.fleaflicker.com/api/FetchLeagueScoreboard?' +
    f'leagueId={LEAGUE_ID}&scoringPeriod={WEEK}&season={SEASON}')

In [2]: schedule_json = requests.get(schedule_url).json()
```

And our working snapshot from mid week 1:

```
with open('./projects/integration/raw/fleaflicker/schedule.json') as f:  
    schedule_json = json.load(f)
```

The normal start-with-a-list and pick out a single example:

```
In [3]:  
matchup_list = schedule_json['games']  
matchup0 = matchup_list[0]
```

We just need the following. Note we're renaming "home" and "away" to team1 and team2 respectively.

```
In [4]:  
def process_matchup(game):  
    return_dict = {}  
    return_dict['team1_id'] = game['home']['id']  
    return_dict['team2_id'] = game['away']['id']  
    return_dict['matchup_id'] = game['id']  
    return return_dict  
  
In [5]: process_matchup(matchup0)  
Out[5]: {'team1_id': 1605156, 'team2_id': 1605147, 'game_id': '53623240'}
```

Looks like it works, so let's apply it to every game, put it in a DataFrame, and add the week info.

```
In [6]:  
def get_schedule_by_week(league_id, week):  
    schedule_url = (  
        'https://www.fleaflicker.com/api/FetchLeagueScoreboard?' +  
        f'leagueId={LEAGUE_ID}&scoringPeriod={WEEK}&season=2021')  
  
    # schedule_json = requests.get(schedule_url).json()  
    with open('./projects/integration/raw/fleaflicker/schedule.json') as f:  
        :  
        schedule_json = json.load(f)  
  
    matchup_df = DataFrame([process_matchup(x)  
                           for x in schedule_json['games']])  
    matchup_df['season'] = SEASON  
    matchup_df['week'] = week  
    matchup_df['league_id'] = league_id  
    return matchup_df
```

This will work for any specific week, but we wanted it for the whole season, and we wanted it in a function `get_league_schedule`. All we have to do is call this function for every week and stick them together.

```
In [7]:  
def get_league_schedule(league_id):  
    return pd.concat([get_schedule_by_week(league_id, week, False) for  
                     week in  
                     range(1, 15)], ignore_index=True)  
  
In [8]: league_schedule.head()  
Out[8]:  
   team1_id  team2_id  game_id  season  week  league_id  
0    1605156   1605147  53623240    2023     1    316893  
1    1605154   1605151  53623242    2023     1    316893  
2    1747266   1605155  53623238    2023     1    316893  
3    1605148   1605157  53623241    2023     1    316893  
4    1603352   1664196  53623239    2023     1    316893
```

Wrap Up

To wrap up we'll get rid of all but the final versions of the functions we wrote above. I've put my final cleaned up version in [./hosts/fleaflicker.py](#)

Like we wanted, this file has:

- `get_teams_in_league`
- `get_league_schedule`
- `get_league_rosters`

These are functions that are meant to be called outside this file.

The other functions (e.g. `process_player3`) are NOT meant to be called outside this file. They're "helper" functions, `process_player3` is used inside `get_league_rosters`, no one else should ever have to use it.

For these helper functions, I kept the final versions (`process_player3` vs `process_player2` or 1), dropped the number (just `process_player`) then added an underscore to the front (e.g. `_process_player`).

Making helper functions start with `_` is a python convention that lets people reading your code know they shouldn't have to use this outside your module.

Feel free to explore these files or work through any other league host integrations you need. When you're ready to pick up with rest of the book head over to [Saving League Data to a Database](#).

Sleeper Integration

Note: this section covers how to get data from leagues hosted on Sleeper. Skip this and find the relevant section (Yahoo, ESPN, Fleaflicker) if your league is hosted on something else.

To get connect to Sleeper you need to know our league id. You can find it in your league URL:

<https://sleeper.app/leagues/league id is here>

For example, my league id is 1002102487509295104.

Sleeper has a public API, which will give us everything we need. The documentation is here:

<https://docs.sleeper.app/#introduction>

Our approach will be:

1. Think about the data we want (rosters, team information, etc).
2. Try one of the endpoints described in the documentation.
3. Figure out if (and where) it has the information we need.
4. Get that information out using Python.

Remember, all of these APIs return JSON, which are nested dicts and lists of dicts and lists. **Almost always, we'll be manipulating these data structures until we have list of Python dictionaries that all have the same keys. Then we'll pass these to DataFrame.**

Roster Data

Let's start by getting league rosters. Remember, we want:

- player_id
- name
- player_position
- team_position
- start
- team_id
- actual points scored so far this week.

The endpoint that works best for this is the matchup endpoint, which takes a week argument and returns full rosters + mid-week points.

Here are our matchups for week 2, 2023:

<https://api.sleeper.app/v1/league/1002102487509295104/matchups/2>

If you click on this URL, you should see some data. To make sure we're seeing the same thing, let's look at some specific data I saved from the Sleeper API between Thursday and Sunday, Week 2, 2023.

This is one of the files that came with this book:

```
./projects/integration/raw/sleeper/matchup.json
```

Find it on your computer and double click on it. It should open up in the browser. You'll see:

```
JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON

▶ 0: {...}
▶ 1: {...}
▶ 2: {...}
▶ 3: {...}
▼ 4:
  ▼ starters_points:
    0: 0
    1: 2.3
    2: 0
    3: 0
    4: 0
    5: 0
    6: 0
    7: 0
    8: 0
  ▼ starters:
    0: "4881"
    1: "4866"
    2: "4663"
    3: "6794"
    4: "4037"
    5: "5844"
    6: "6801"
    7: "17"
    8: "GB"
  roster_id: 5
  points: 2.3
  ▶ players_points: {...}
  ▶ players: [...]
  matchup_id: 1
  custom_points: null
▼ 5:
  ▶ starters_points: [...]
  ▶ starters: [...]
  roster_id: 6
  points: 0
  ▶ players_points: {...}
```

Figure 0.6: Sleeper Roster JSON

Let's load it in Python. Picking up in `./projects/integration/sleeper_working.py` right after our imports:

Setting some parameters:

```
In [1]:  
LEAGUE_ID = 1002102487509295104  
WEEK = 2
```

Normally you'd get live data this way:

```
In [2]:  
matchup_url = f'https://api.sleeper.app/v1/league/{LEAGUE_ID}/matchups/{  
    WEEK}'  
matchup_json = requests.get(matchup_url).json()
```

But for this walkthrough we'll keep using the snapshot I saved. We can load that in Python like this:

```
In [3]:  
with open('./projects/integration/raw/sleeper/matchup.json') as f:  
    matchup_json = json.load(f)
```

It's usually easiest to start with a single team:

```
In [4]: team5 = matchup_json[5]  
  
In [5]: team5['starters']  
Out[5]: ['3294', '4866', '7611', '8146', '4037', '2197', '5844', '3199', '  
SEA']
```

The `starters` field is just a list. Based on the fact the defense (Seattle DST) is at the end, I think it's safe to assume the order of `starters` denotes position. We don't have anything on positions in `matchup_json`, but let's see if we can get that info from a different endpoint.

According the Sleeper API docs, league settings are available here:

https://api.sleeper.app/v1/league/%7BLEAGUE_ID%7D

And we can get it with:

```
In [6]:  
settings_url = f'https://api.sleeper.app/v1/league/{LEAGUE_ID}'  
settings_json = requests.get(settings_url).json()
```

Though just like the other endpoints, I've saved a snapshot:

```
In [7]:  
with open('./projects/integration/raw/sleeper/settings.json') as f:  
    settings_json = json.load(f)
```

The positions are in a field called `roster_positions`:

```
In [8]: positions = settings_json['roster_positions']

In [9]: positions
Out[9]:
['QB',
 'RB',
 'RB',
 'WR',
 'WR',
 'WR',
 'WR',
 'TE',
 'FLEX',
 'DEF',
 'BN',
 'BN']
```

So, based on:

```
In [10]: team0['starters']
Out[10]: ['6797', '6130', '6945', '2133', '5859', '7553', '4068', '4227',
          'WAS']
```

My guess like this team's QB is player '`6797`', the RBs are '`6130`' and '`6945`', etc. But we should probably check this. We need a link between `sleeper_id` and name and position.

Sleeper has an API endpoint for player information, but so does the Fantasy Math API, so we'll just use that. In `utilities.py`, I've included a helper function `master_player_lookup` that links up `sleeper_id` and `fantasymath_id`. It also includes position.

```
In [11]:  
token = generate_token(LICENSE_KEY)['token']  
fantasymath_players = master_player_lookup(token)  
  
In [12]: fantasymath_players.head()  
Out[12]:  
      name  player_id  pos  ...  sleeper_id  
0  Tanner McKee       1   QB  ...    9230  
1  Bryce Young       2   QB  ...    9228  
2  Will Levis        3   QB  ...    9999  
3  C.J. Stroud       4   QB  ...    9758  
4 Anthony Richardson 6   QB  ...    9229
```

We'll use a saved snapshot of it:

```
In [13]: fantasymath_players = pd.read_csv(  
        './projects/integration/raw/sleeper/lookup.csv')
```

So let's take our Sleeper starters and attach this lookup to it. We can do this via a merge, but merge only works on two DataFrames. Right now have one DataFrame (the `fantasymath_players` lookup) and then just a list of Sleeper ids.

So we need to make that list a DataFrame. We talked about how to do this in LTCWFF; one way:

```
In [14]: starters5 = Series(team5['starters']).to_frame('sleeper_id')  
  
In [15]: starters5  
Out[15]:  
  sleeper_id  
0      3294  
1      4866  
2      7611  
3      8146  
4      4037  
5      2197  
6      5844  
7      3199  
8      SEA  
  
In [16]: type(starters5)  
Out[16]: pandas.core.frame.DataFrame
```

You could also turn the list of starters into a list of dictionaries:

```
In [17]: DataFrame([{'sleeper_id': x} for x in team9['starters']])  
Out[17]:  
   sleeper_id  
0        6797  
1        6130  
2        6945  
3        2133  
4        5859  
5        7553  
6        4068  
7        4227  
8        WAS
```

Or pass `DataFrame` the data and column name separately:

```
In [16]: DataFrame(team9['starters'], columns=['sleeper_id'])  
Out[16]:  
   sleeper_id  
0        6797  
1        6130  
2        6945  
3        2133  
4        5859  
5        7553  
6        4068  
7        4227  
8        WAS
```

It doesn't matter how we do it. But once we have it, we can merge `player_id` and `position` from our lookup:

```
In [17]:  
starters5_w_info = pd.merge(  
    starters5,  
    fantasymath_players[['sleeper_id', 'player_id', 'name', 'pos']],  
    how='left')  
  
In [18]: starters5_w_info  
Out[18]:  
   sleeper_id  player_id          name  pos  
0        3294      1297      Dak Prescott  QB  
1        4866       904     Saquon Barkley  RB  
2        7611       349  Rhamondre Stevenson  RB  
3        8146       210     Garrett Wilson  WR  
4        4037      1138     Chris Godwin  WR  
5        2197      1719    Brandin Cooks  WR  
6        5844       756     T.J. Hockenson  TE  
7        3199      1329    Michael Thomas  WR  
8        SEA       5178           SEA  DST
```

We have most of what we need for this team's roster. Next, let's add any actual points. Again, this data snapshot is from Friday morning, Week 2. PHI-MIN played Thursday. We want these points reflected in our roster, so let's add them.

```
In [19]: starters5_w_info['actual'] = team5['starters_points']
```

```
In [20]: starters5_w_info
```

```
Out[20]:
```

	sleeper_id	player_id	name	pos	actual
0	3294	1297	Dak Prescott	QB	0.0
1	4866	904	Saquon Barkley	RB	0.0
2	7611	349	Rhamondre Stevenson	RB	0.0
3	8146	210	Garrett Wilson	WR	0.0
4	4037	1138	Chris Godwin	WR	0.0
5	2197	1719	Brandin Cooks	WR	0.0
6	5844	756	T.J. Hockenson	TE	25.6
7	3199	1329	Michael Thomas	WR	0.0
8	SEA	5178	SEA	DST	0.0

We can see the one Minnesota player (Hockenson) has points. Everyone else has 0. These 0's aren't really no points; they'd be better represented by missing. Let's fill that in:

```
In [21]: starters9_w_info.loc[
    starters9_w_info['actual'] == 0, 'actual'] = np.nan
```

```
In [22]: starters9_w_info
```

```
Out[22]:
```

	sleeper_id	player_id	name	pos	actual
0	3294	1297	Dak Prescott	QB	NaN
1	4866	904	Saquon Barkley	RB	NaN
2	7611	349	Rhamondre Stevenson	RB	NaN
3	8146	210	Garrett Wilson	WR	NaN
4	4037	1138	Chris Godwin	WR	NaN
5	2197	1719	Brandin Cooks	WR	NaN
6	5844	756	T.J. Hockenson	TE	25.6
7	3199	1329	Michael Thomas	WR	NaN
8	SEA	5178	SEA	DST	NaN

The other thing is we'll want to add *team* position, vs just the player position we have now. This mostly matters for flex. Above, Michael Thomas is in our flex spot, and it'd be nice to know that. Right now all we know is he's a WR.

Remember we have a list of positions from our settings JSON above:

```
In [23]: positions
Out[23]:
['QB',
 'RB',
 'RB',
 'WR',
 'WR',
 'WR',
 'TE',
 'FLEX',
 'DEF',
 'BN',
 'BN']
```

Let's use these to make a "team position" column:

```
In [24]: starters5_w_info['team_position'] = [x for x in positions if x != 'BN']

In [25]: starters5_w_info
Out[25]:
   sleeper_id  player_id          name  pos  actual team_position
0      3294      1297    Dak Prescott  QB    NaN        QB
1      4866      904    Saquon Barkley  RB    NaN        RB
2      7611      349  Rhamondre Stevenson  RB    NaN        RB
3      8146      210    Garrett Wilson  WR    NaN        WR
4      4037     1138    Chris Godwin  WR    NaN        WR
5      2197     1719    Brandin Cooks  WR    NaN        WR
6      5844      756    T.J. Hockenson  TE  25.6        TE
7      3199     1329    Michael Thomas  WR    NaN       FLEX
8      SEA       5178         SEA     DST    NaN        DEF
```

Now we can tell Michael Thomas is in the FLEX. Awesome.

This looks good, but at some point duplicate `team_positions` might be an issue too. It'd be better to be able to refer to our RB1, RB2 etc.

Let's add that. As always let's start with a specific example, say WRs:

```
In [24]: wrs = starters9_w_info.query("team_position == 'WR'")
```

```
In [25]: wrs
```

```
Out[25]:
```

	sleeper_id	player_id	name	pos	actual	team_position
3	8146	210	Garrett Wilson	WR	Nan	WR
4	4037	1138	Chris Godwin	WR	Nan	WR
5	2197	1719	Brandin Cooks	WR	Nan	WR

So we want `team_position` to say WR1, WR2 instead of just WR, WR.

The easiest way is probably to make a column with 1, 2 then stick them together.

```
In [26]: suffix = Series(range(1, len(wrs) + 1), index=wrs.index)
```

```
In [27]: suffix
```

```
Out[27]:
```

3	1
4	2
5	3

The key is we're making this column have the same index as `wrs`, that way we can do this:

```
In [28]: wrs['team_position'] + suffix.astype(str)
```

```
Out[28]:
```

3	WR1
4	WR2
5	WR3

Which is what we want. Now let's put this in a function:

```
def add_pos_suffix(df_subset):  
    # only add if more than one position -- want just K, not K1  
    if len(df_subset) > 1:  
        suffix = Series(range(1, len(df_subset) + 1), index=df_subset.  
                        index)  
  
        df_subset['team_position'] = df_subset['team_position'] + suffix.  
                                    astype(str)  
    return df_subset
```

and apply it to every position in the starter DataFrame.

```
In [29]:  
starters5_pos = pd.concat([  
    add_pos_suffix(starters5_w_info.query(f"team_position == '{x}'"))  
    for x in starters5_w_info['team_position'].unique()])  
  
--  
  
In [30]: starters5_pos  
Out[30]:
```

	sleeper_id	player_id	name	pos	actual	team_position
0	3294	1297	Dak Prescott	QB	NaN	QB
1	4866	904	Saquon Barkley	RB	NaN	RB1
2	7611	349	Rhamondre Stevenson	RB	NaN	RB2
3	8146	210	Garrett Wilson	WR	NaN	WR1
4	4037	1138	Chris Godwin	WR	NaN	WR2
5	2197	1719	Brandin Cooks	WR	NaN	WR3
6	5844	756	T.J. Hockenson	TE	25.6	TE
7	3199	1329	Michael Thomas	WR	NaN	FLEX
8	SEA	5178	SEA	DST	NaN	DEF

Cool. Now let's attach a bench version to it. We'll basically do the same thing, but for everyone in `players` instead of `starters`.

```
In [31]:  
players5 = Series(team5['players']).to_frame('sleeper_id')  
players5_w_info = pd.merge(players5, fantasymath_players, how='left')  
  
In [32]: players5_w_info  
Out[32]:
```

	sleeper_id	name	...	espn_id	yahoo_id
0	SEA	SEA	...	-16026	100026
1	NYJ	NYJ	...	-16020	100020
2	8146	Garrett Wilson	...	4569618	33965
3	7611	Rhamondre Stevenson	...	4569173	33508
4	6804	Jordan Love	...	4036378	32696
5	5955	Hunter Renfrow	...	3135321	31981
6	5890	Damien Harris	...	3925347	31919
7	5857	Noah Fant	...	4036131	31852
8	5844	T.J. Hockenson	...	4036133	31840
9	5086	Marquez Valdes-Scantling	...	3051738	31144
10	4985	Rashaad Penny	...	3139925	30997
11	4973	Hayden Hurst	...	3924365	30995
12	4866	Saquon Barkley	...	3929630	30972
13	4080	Zay Jones	...	3059722	30150
14	4037	Chris Godwin	...	3116165	30197
15	3294	Dak Prescott	...	2577417	29369
16	3199	Michael Thomas	...	2976316	29281
17	2197	Brandin Cooks	...	16731	27548
18	2078	Odell Beckham	...	16733	27540
19	1837	Jimmy Garoppolo	...	16760	27590

The only wrinkle is — unlike starter points — player points are a *dict* of values:

```
In [33]: team5['players_points']
Out[33]:
{'SEA': 0.0,
 'NYJ': 0.0,
 '8146': 0.0,
 '7611': 0.0,
 '6804': 0.0,
 '5955': 0.0,
 '5890': 0.0,
 '5857': 0.0,
 '5844': 25.6,
 '5086': 0.0,
 '4985': 2.4,
 '4973': 0.0,
 '4866': 0.0,
 '4080': 0.0,
 '4037': 0.0,
 '3294': 0.0,
 '3199': 0.0,
 '2197': 0.0,
 '2078': 0.0,
 '1837': 0.0}
```

There are multiple ways to get this information into our DataFrame. We could turn `player_points` into a DataFrame and merge it in, or use Pandas builtin `replace` function. But let's use `apply` and an anonymous function.

```
In [34]:  
players5_w_info['actual'] = (  
    players5_w_info['sleeper_id'].apply(lambda x: team5['players_points'][  
        x]))  
  
In [35]: players5_w_info  
Out[35]:
```

	sleeper_id	name	...	yahoo_id	actual
0	SEA	SEA	...	100026	0.0
1	NYJ	NYJ	...	100020	0.0
2	8146	Garrett Wilson	...	33965	0.0
3	7611	Rhamondre Stevenson	...	33508	0.0
4	6804	Jordan Love	...	32696	0.0
5	5955	Hunter Renfrow	...	31981	0.0
6	5890	Damien Harris	...	31919	0.0
7	5857	Noah Fant	...	31852	0.0
8	5844	T.J. Hockenson	...	31840	25.6
9	5086	Marquez Valdes-Scantling	...	31144	0.0
10	4985	Rashaad Penny	...	30997	2.4
11	4973	Hayden Hurst	...	30995	0.0
12	4866	Saquon Barkley	...	30972	0.0
13	4080	Zay Jones	...	30150	0.0
14	4037	Chris Godwin	...	30197	0.0
15	3294	Dak Prescott	...	29369	0.0
16	3199	Michael Thomas	...	29281	0.0
17	2197	Brandin Cooks	...	27548	0.0
18	2078	Odell Beckham	...	27540	0.0
19	1837	Jimmy Garoppolo	...	27590	0.0

And getting only the bench players:

```
In [36]: bench_players = set(team5['players']) - set(team5['starters'])

In [37]: bench_players
Out[37]:
{'1837',
 '2078',
 '4080',
 '4973',
 '4985',
 '5086',
 '5857',
 '5890',
 '5955',
 '6804',
 'NYJ'}

In [38]: bench_df = players5_w_info.query(f"sleeper_id in {tuple(bench_players)}")

In [39]: bench_df['team_position'] = 'BN'

In [40]: bench_df.loc[bench_df['actual'] == 0, 'actual'] = np.nan

In [41]: bench_df
Out[41]:
   sleeper_id          name  ...  actual team_position
1           NYJ        NYJ  ...    NaN         BN
4           6804      Jordan Love  ...    NaN         BN
5           5955    Hunter Renfrow  ...    NaN         BN
6           5890     Damien Harris  ...    NaN         BN
7           5857       Noah Fant  ...    NaN         BN
9           5086 Marquez Valdes-Scantling  ...    NaN         BN
10          4985      Rashaad Penny  ...  2.4         BN
11          4973      Hayden Hurst  ...    NaN         BN
13          4080        Zay Jones  ...    NaN         BN
18          2078     Odell Beckham  ...    NaN         BN
19          1837    Jimmy Garoppolo  ...    NaN         BN
```

Now let's stick our starters and bench together, then do some light processing to make sure we have all of our other columns (name, start, etc):

```
In [42]: team5_df = pd.concat([starters5_pos, bench_df], ignore_index=True)
)

In [43]:
team5_df.drop(['yahoo_id', 'espn_id', 'fleaflicker_id', 'sleeper_id'],
    axis=1,
    inplace=True)
-- 

In [44]: team5_df.rename(columns={'position': 'player_position'}, inplace=True)

In [45]: team5_df['start'] = team5_df['team_position'] != 'BN'

In [46]: team5_df['team_id'] = team5['roster_id']

In [47]: team5_df
Out[47]:
   player_id           name  ...  start  team_id
0      1297       Dak Prescott  ...   True      6
1      904        Saquon Barkley  ...   True      6
2      349     Rhamondre Stevenson  ...   True      6
3      210        Garrett Wilson  ...   True      6
4     1138        Chris Godwin  ...   True      6
5     1719      Brandin Cooks  ...   True      6
6      756      T.J. Hockenson  ...   True      6
7     1329     Michael Thomas  ...   True      6
8      5178            SEA  ...   True      6
9      5175            NYJ  ...  False      6
10     497        Jordan Love  ...  False      6
11     741      Hunter Renfrow  ...  False      6
12     693        Damien Harris  ...  False      6
13     755         Noah Fant  ...  False      6
14     993  Marquez Valdes-Scantling  ...  False      6
15     908        Rashaad Penny  ...  False      6
16     980        Hayden Hurst  ...  False      6
17    1151         Zay Jones  ...  False      6
18    1724      Odell Beckham  ...  False      6
19    1757     Jimmy Garoppolo  ...  False      6
```

This is what we want. Now let's put all this in a function that'll work on any team:

```
def get_team_roster(team, lookup, positions):
    # starters
    starters = Series(team['starters']).to_frame('sleeper_id')

    starters_w_info = pd.merge(starters, lookup, how='left')
    starters_w_info['actual'] = team['starters_points']
    starters_w_info.loc[starters_w_info['actual'] == 0, 'actual'] = np.nan
    starters_w_info['team_position'] = [x for x in positions if x != 'BN']

    starters_pos = pd.concat([
        add_pos_suffix(starters_w_info.query(f"team_position == '{x}'"))
        for x in starters_w_info['team_position'].unique()])

    players = Series(team['players']).to_frame('sleeper_id')
    players_w_info = pd.merge(players, lookup, how='left')

    players_w_info['actual'] = (players_w_info['sleeper_id']
                                .apply(lambda x: team9['players_points'][x]))

    bench_players = set(team['players']) - set(team['starters'])

    bench_df = players_w_info.query(f"sleeper_id in {tuple(bench_players)}")
    bench_df['team_position'] = 'BN'
    bench_df.loc[bench_df['actual'] == 0, 'actual'] = np.nan

    team_df = pd.concat([starters_pos, bench_df], ignore_index=True)
    team_df.drop(['yahoo_id', 'espn_id', 'fleaflicker_id', 'sleeper_id'],
                axis=1,
                inplace=True)
    team_df.rename(columns={'position': 'player_position'}, inplace=True)
    team_df['start'] = team_df['team_position'] != 'BN'
    team_df['name'] = team_df['fantasymath_id'].str.replace('-', ' ').str.title()
    team_df['team_id'] = team['roster_id']
    return team_df
```

Once we have this, getting it for *all* teams in our league is easy:

```
In [48]:  
all_rosters = pd.concat(  
    [get_team_roster(x, fantasymath_players, positions) for x in  
     matchup_json],  
    ignore_index=True)  
--  
  
In [49]: all_rosters.sample(10)  
Out[49]:
```

		name	player_id	...	start	team_id
111	Hunter Renfrow		741	...	False	6
163	JaMarr Chase		371	...	True	9
64	DK Metcalf		720	...	True	4
220	Josh Allen		889	...	True	12
71	Sean Tucker		20	...	False	4
53	Devon Achane		21	...	False	3
144	Jerry Jeudy		548	...	True	8
192	Rondale Moore		373	...	False	10
225	JuJu Smith-Schuster		1131	...	True	12
115	Rashaad Penny		908	...	False	6

And let's put this in a function to return everyone's roster for a given week:

```
def get_league_rosters(lookup, league_id, week):  
    matchup_url = f'https://api.sleeper.app/v1/league/{league_id}/matchups  
    /{week}'  
    matchup_json = requests.get(matchup_url).json()  
  
    return pd.concat([get_team_roster(x, lookup) for x in  
                    matchup_json], ignore_index=True)
```

Awesome. This was probably the most complicated piece of the league connection project, and we've knocked it out.

Team Data

Now we need team data, specifically:

- `team_id`
- `owner_id`
- `owner_name`
- `league_id`

Looking at the documentation, team data is available in the users endpoint.

```
In [1]: teams_url = 'https://api.sleeper.app/v1/league/{LEAGUE_ID}/users'  
In [2]: teams_json = requests.get(teams_url).json()
```

Again, that's how we'd get the live data, but we'll use our snapshot for the walkthrough:

```
In [3]:  
with open('./projects/integration/raw/sleeper/teams.json') as f:  
    teams_json = json.load(f)
```

As always, let's start with a specific example:

```
In [4]: team1 = teams_json[1]
```

Looking at the data, most of what we want is straightforward to get out. We can use a function:

```
def proc_team1(team):  
    dict_to_return = {}  
  
    dict_to_return['owner_id'] = team['user_id']  
    dict_to_return['owner_name'] = team['display_name']  
    return dict_to_return
```

And calling it:

```
In [5]: proc_team1(team1)  
Out[5]: {'owner_id': '733553479255191552', 'owner_name': 'prodeezay'}
```

The only part that's *not* in there is some sort of team id. But here, as with other parts of the Sleeper API, I think order matters. If you look at the rosters section, each team includes a *roster id*, which is a number 1 through number of teams in the league. I think that corresponds to spot in this list. So let's modify our `proc_team` function to add that:

```
def proc_team2(team, team_id):
    dict_to_return = {}

    dict_to_return['owner_id'] = team['user_id']
    dict_to_return['owner_name'] = team['display_name']
    dict_to_return['team_id'] = team_id
    return dict_to_return
```

Then we can include it in our list of teams like this:

```
In [6]: proc_team2(team0, 1)
Out[6]: {'owner_id': '733553479255191552', 'owner_name': 'prodeezay',
          'team_id': 1}
```

To keep track of which spot each team is at in the list we can use the `enumerate` function. It's basically a way to get a counter when looping or using a comprehension.

```
In [7]:
for i, team in enumerate(teams_json, start=1):
    print(i)
    print(team['display_name'])
--

1
jonhanson
2
prodeezay
3
nathanbraun
4
pbecker1313
5
sporeilly
6
adeising
7
joebuckyourself0412
8
Moye
9
mkomp
10
Breger
11
CAlt99
12
GMazzzz
```

So if we wanted to stick *all* our teams together we could do it like this:

```
In [8]:
```

```
all_teams = DataFrame(  
    [proc_team2(team, i) for i, team in enumerate(teams_json, start=1)])
```

```
In [9]: all_teams
```

```
Out[9]:
```

```
          owner_id      owner_name  team_id  
0       39936457734883736       johanson      1  
1       733553479255191552     prodeezay      2  
2       744782197076164608   nathanbraun      3  
3      1002102271125192704   pbecker1313      4  
4      1002203259987210240     sporeilly      5  
5      1002241744664203264     adeising      6  
6      1002261056279900160  joebuckyourself0412      7  
7      1002293786724126720       Moye      8  
8      1002607082813050880      mkomp      9  
9      1002623786170167296      Breger     10  
10     1002663459793809408     CALt99     11  
11     1002969872512540672     GMazzzz     12
```

Almost what we want, just missing league id, which we have saved above and is easy to add as a column.

Let's put all this in a function. Remember it needs to be called `get_teams_in_league`.

```
def get_teams_in_league(league_id):  
    teams_url = f'https://api.sleeper.app/v1/league/{league_id}/users'  
  
    # teams_json = requests.get(teams_url).json()  
    with open('./projects/integration/raw/sleeper/teams.json') as f:  
        teams_json = json.load(f)  
  
    all_teams = DataFrame(  
        [proc_team2(team, i) for i, team in enumerate(teams_json, start=1)]  
    )  
    all_teams['league_id'] = league_id  
    return all_teams
```

Schedule Info

The last thing we need is the schedule.

Looking at the documentation and our data above, it looks like the matchup endpoint will do it. Again, normally you'd get it like this:

```
In [1]:  
matchup_url = f'https://api.sleeper.app/v1/league/{LEAGUE_ID}/matchups/{  
    WEEK}'  
matchup_json = requests.get(matchup_url).json()
```

But we'll use our snapshot:

```
In [2]:  
with open('./projects/integration/raw/sleeper/matchup.json') as f:  
    matchup_json = json.load(f)
```

This is all by teams, so let's look at a specific one:

```
In [3]: team0 = matchup_json[0]

In [68]: team0
Out[68]:
{'starters_points': [0.0, 0.0, 0.0, 6.9, 0.0, 0.0, 0.0, 0.0, 0.0],
 'starters': ['5870',
 '4988',
 '4018',
 '5859',
 '5872',
 '8137',
 '5001',
 '7588',
 'NE'],
 'roster_id': 1,
 'points': 6.9,
 'players_points': {'NE': 0.0,
 'JAX': 0.0,
 '9481': 0.0,
 '8676': 0.0,
 '8221': 0.0,
 '8160': 0.0,
 '8137': 0.0,
 '8119': 0.0,
 '8110': 0.0,
 '7588': 0.0,
 '6943': 0.0,
 '6920': 0.0,
 '5872': 0.0,
 '5870': 0.0,
 '5859': 6.9,
 '5248': 0.0,
 '5001': 0.0,
 '4988': 0.0,
 '4018': 0.0,
 '1373': 0.0},
 'players': ['NE',
 'JAX',
 '9481',
 '8676',
 '8221',
 '8160',
 '8137',
 '8119',
 '8110',
 '7588',
 '6943',
 '6920',
 '5872',
 '5870',
 '5859',
 '5248',
 '5001',
 '4988',
 '4018',
 '1373'],
 'matchup_id': 2,
 'custom_points': None}
```

And build a function to get what we want out of it:

```
def proc_team_schedule(team):
    dict_to_return = {}
    dict_to_return['team_id'] = team['roster_id']
    dict_to_return['matchup_id'] = team['matchup_id']
    return dict_to_return
```

Running it:

```
In [4]: proc_team_schedule(team0)
Out[4]: {'team_id': 1, 'matchup_id': 2}
```

Now let's run it on every team in our data:

```
In [5]: schedule_w2 = DataFrame([proc_team_schedule(team)
                                 for team in matchup_json])

In [6]: schedule_w2
Out[6]:
   team_id  matchup_id
0         1          2
1         2          5
2         3          4
3         4          6
4         5          1
5         6          6
6         7          4
7         8          5
8         9          3
9        10          2
10       11          3
11       12          1
```

This has the info we need, but it's at the team and game level. Let's get it to just the game level, where we have `matchup_id` as a column (no duplicates) and `team1_id` and `team2_id`. This is a nice little self contained puzzle if you want to stop and try it out for yourself.

Hint: the `drop_duplicates` method takes a column to drop duplicates by, as well as a `keep` argument where you can tell it which duplicate you want to keep (e.g. `'first'` or `'last'` are acceptable values).

My solution:

```
In [7]:  
schedule_w2_wide = pd.merge(  
    schedule_w2.drop_duplicates('matchup_id', keep='first'),  
    schedule_w2.drop_duplicates('matchup_id', keep='last'), on='matchup_id'  
)  
  
In [8]: schedule_w2_wide  
Out[8]:  
   team_id_x  matchup_id  team_id_y  
0           1          2        10  
1           2          5         8  
2           3          4         7  
3           4          6         6  
4           5          1        12  
5           9          3        11
```

The merge functions adds `_x` and `_y` suffixes automatically, let's rename them. Then add the other info we want and we'll be all set:

```
In [9]:  
schedule_w2_wide.rename(  
    columns={'team_id_x': 'team1_id', 'team_id_y': 'team2_id'},  
    inplace=True)  
  
In [10]: schedule_w2_wide['season'] = SEASON  
  
In [11]: schedule_w2_wide['week'] = WEEK  
  
In [12]: schedule_w2_wide  
Out[12]:  
   team1_id  matchup_id  team2_id  season  week  
0           1          2        10    2023     2  
1           2          5         8    2023     2  
2           3          4         7    2023     2  
3           4          6         6    2023     2  
4           5          1        12    2023     2  
5           9          3        11    2023     2
```

Great. Let's put it in a function:

```
def get_schedule_by_week(league_id, week):
    matchup_url = f'https://api.sleeper.app/v1/league/{league_id}/matchups
                  /{week}'

    matchup_json = requests.get(matchup_url).json()

    team_sched = DataFrame([proc_team_schedule(team) for team in
                           matchup_json])

    team_sched_wide = pd.merge(
        team_sched.drop_duplicates('matchup_id', keep='first'),
        team_sched.drop_duplicates('matchup_id', keep='last'), on='
                                  matchup_id')

    team_sched_wide.rename(
        columns={'team_id_x': 'team1_id', 'team_id_y': 'team2_id'},
        inplace=True)

    team_sched_wide['season'] = SEASON
    team_sched_wide['week'] = week
    return team_sched_wide
```

And try it out:

```
In [13]: sched_w3 = get_schedule_by_week(LEAGUE_ID, 3)

In [14]: sched_w3
Out[14]:
   team1_id  matchup_id  team2_id  season  week
0          1            2          5    2021    3
1          2            6          4    2021    3
2          3            3          9    2021    3
3          6            5          8    2021    3
4          7            4         11    2021    3
5         10           1         12    2021    3
```

This function returns the schedule one week at a time. For the whole season, we'd have to run it for every week, then stick them together.

We can do that, but we need to know how many weeks there are. The number of regular season games in the settings endpoint we hit earlier:

```
In [15]: settings_json['settings']['playoff_week_start']
Out[15]: 14
```

So let's build all this into a function. Given some league id, we want: (1) to hit the settings endpoint and figure out the number of regular season games, then (2) use our `get_schedule_by_week` function to get matchups for every game, (3) sticking all those together. And we want that in a function called

get_league_schedule.

How about this:

```
def get_league_schedule(league_id):
    settings_url = f'https://api.sleeper.app/v1/league/{league_id}'
    settings_json = requests.get(settings_url).json()

    n = settings_json['settings']['playoff_week_start']
    if n == 0:
        n = 19
    return pd.concat(
        [get_schedule_by_week(league_id, x) for x in range(1, n)],
        ignore_index=True)
```

And calling it:

```
In [14]: league_schedule = get_league_schedule(LEAGUE_ID)
```

```
In [15]: league_schedule
```

```
Out[15]:
```

	team1_id	matchup_id	team2_id	season	week
0	1	1	12	2023	1
1	2	6	6	2023	1
2	3	4	11	2023	1
3	4	5	8	2023	1
4	5	2	10	2023	1
..
103	2	4	10	2023	18
104	3	1	12	2023	18
105	4	5	9	2023	18
106	5	3	6	2023	18
107	7	6	8	2023	18

Wrap Up

To wrap up we'll get rid of all but the final versions of the functions we wrote above. I've put my final cleaned up version in `./hosts/sleeper.py`

Like we wanted, this file has:

- `get_teams_in_league`
- `get_league_schedule`
- `get_league_rosters`

These are functions that are meant to be called outside this file.

The other functions (e.g. `proc_team2`) are NOT meant to be called outside this file. They're "helper" functions, `proc_team2` is used inside `get_teams_in_league`, no one else should ever have to use it.

For these helper functions, I kept the final versions (`proc_team2` vs `proc_team1`), dropped the number (just `proc_team`) then added an underscore to the front (e.g. `_proc_team`).

Making helper functions start with `_` is a python convention that lets people reading your code know they shouldn't have to use this outside your module.

Feel free to explore these files or work through any other league host integrations you need. When you're ready to pick up with rest of the book head over to [Saving League Data to a Database](#).

Yahoo Integration

Note: this section covers how to get data from leagues hosted on Yahoo. Skip this and find the relevant section (Sleeper, ESPN, Fleaflicker) if your league is hosted on something else.

Authentication and Setup

Yahoo has a complicated authorization process that is a pain to set up initially.

Broadly: Yahoo doesn't let regular people like us access their APIs, only third party "applications". In practice, this just means we have to fill out a form with Yahoo where we tell them about our application, and get back some credentials we can use. This part isn't that bad. The form is automated so it's not like we have to wait for approval or anything.

So say the name of our "app" (in quotes because it's really just a handful of lines of Python) is "the yahoo fantasy football developer kit app". We've created it and it has permissions to access the fantasy football API. Great.

But that's our app, not us. It can't just log into anyone's fantasy football account and view their data. We'll need to set up a workflow similar to 'Sign in with Google' or other third party logins, where our app brings us to Yahoo, Yahoo says something like: "are you sure you want to share your data with *the yahoo fantasy football developer kit app*?" We say yes, and then we're in.

It's a pain because it's our app, under our account, that we need to link up with Yahoo just to get our own Yahoo fantasy football data. It's a lot of messing around, but the good news is we only have to do it once.

You need to be logged into your Yahoo account to do this, so let's just log into our league and grab some information we'll need later while we're there.

Setup/Registering Your Yahoo App

1. Login to your Yahoo Fantasy Football league.
2. Find your league id, which is in the top left corner of your league. Put this in `LEAGUE_ID` variable in `./code/projects/integration/yahoo_working.py`.
3. Go to <https://developer.yahoo.com/apps/create/> and create an app. Note you need to be logged into the Yahoo account.
4. You'll see this screen:

Create Application

Application Name

Description

Homepage URL

Redirect URI(s)

Please specify any additional redirect uris.

API Permissions

Select private user data APIs that your application needs to access.

Fantasy Sports

Oath Ad Platforms

Relationships (Social Directory)

OpenID Connect Permissions

By clicking Create App, you agree to be bound by the [Yahoo Developer Network Terms of Use](#).

[Create App](#) [Cancel](#)

Figure 0.7: Create Yahoo Application

Put in the following:

Application Name (Required) — I put in `the yahoo fantasy football developer kit app` but you can put in whatever you want.

Redirect URI(s) (Required) — This is required, so it needs to a valid URL, but we won't be using it. Just put in `localhost:8080`

API Permissions (Required) — Check the `Fantasy Sports` box and leave the `Read` option selected.

4. Then click [Create App](#). Yahoo will create your app, and redirect you to a page with a `Client ID` and `Client Secret`. Make note of these (or at least don't close the browser), we'll use them in a minute.

Setting up OAuth

Now we have our app, but we still need the part where it links up to your Yahoo account and asks for permissions.

We'll outsource all of this to a third party library, `yahoo_oauth`.

So far, Anaconda has come with every Python package we've needed. But to connect to Yahoo fantasy we'll need a third party package. Here's how to get it:

1. Open up Anaconda Navigator.
2. Click on `Environments` on the left side bar.
3. This will bring up a list of your environments. Click on the 'play button' style green triangle in environment you're using (usually there will just be one environment: `base (root)`, but if you know you're using a different one click that).
4. Select `Open Terminal`.

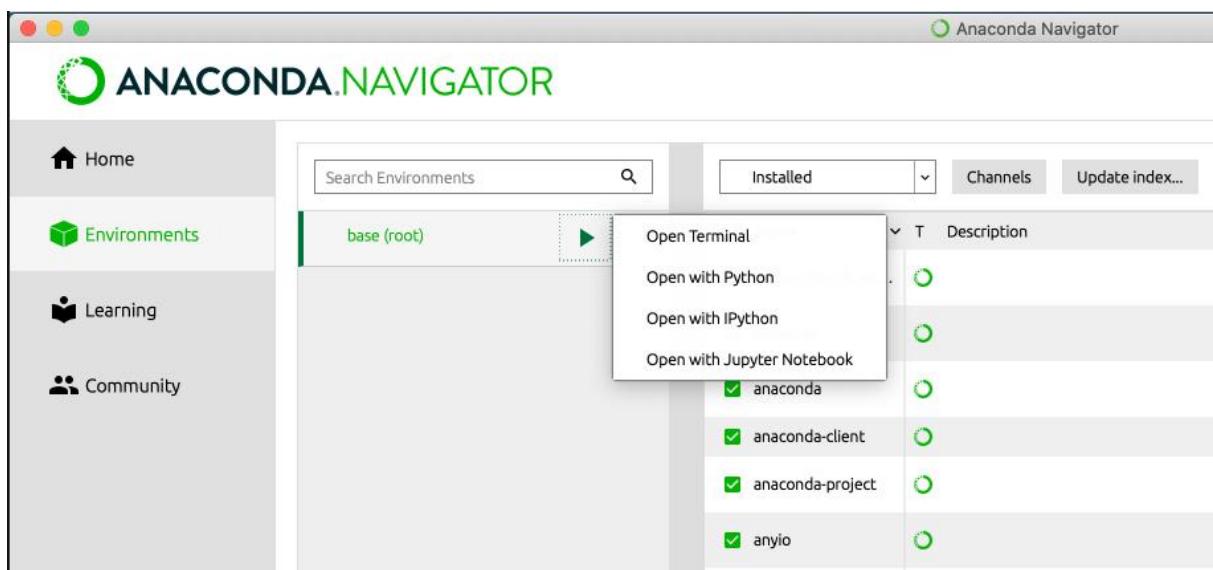


Figure 0.8: Opening Terminal in Anaconda

5. A text console will open up. Type:

```
pip install yahoo_oauth
```

6. Then close it after it installs. Then restart Spyder.

Connecting our app to our Yahoo data

To start, grab your consumer key (aka client id) and secret (aka client secret) from the Yahoo app you made a minute ago and put them in your `config.ini` file:

```
[yahoo]
CONSUMER_KEY =
    dj0yJmk9VTZXXXXXXXXXXXXXXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk
CONSUMER_SECRET = 56ec2XXXXXXXXXXXX8aa1eff00c8
```

(Note those are my keys, partially X'd out)

Yahoo Walkthrough

Now we can start with the walkthrough. Open up `./code/integration/yahoo_working.py`. We'll pick up right after the imports.

Again, we're leaving the connection between our app and our Yahoo account to the `yahoo_oauth` package. This package works by reading a JSON file on your computer with some credentials. Initially, the only credentials in there are your consumer key and secret. Once it connects to your account and you give it, permission, `yahoo_oauth` will automatically add some other data to it.

What this means is we want to make a credentials file for `yahoo_oauth`, but only if it doesn't already exist. Otherwise we'll be overwriting the extra stuff `yahoo_oauth` stored and it'll ask us for permissions again.

So let's do that. First, we'll make sure we have the path to our Yahoo credential files. That should be in `config.ini` and loaded in `utilities.py`:

```
...
from utilities import (LICENSE_KEY, generate_token, master_player_lookup,
                      YAHOO_FILE, YAHOO_KEY, YAHOO_SECRET, YAHOO_GAME_ID)
...
```

If this is the first time running this, this file doesn't exist yet and we need to create it. That's what this does:

```
In [1]:
if not Path(YAHOO_CREDENTIALS).exists():
    yahoo_credentials_dict = {
        'consumer_key': CONSUMER_KEY,
        'consumer_secret': CONSUMER_SECRET,
    }
    with open(YAHOO_FILE, 'w') as f:
        json.dump(yahoo_credentials_dict, f)
```

i.e., “if our Yahoo credentials file doesn't exist, output our consumer key and secret there as JSON”.

After you run this the first time, if you were to open up your `yahoo_credentials.json` file you'd see something like this:

```
{"consumer_key":  
"dj0yJmk9VTZXXXXXXXXXXXXXXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk",  
"consumer_secret": "56ec2XXXXXXXXXX8aa1eff00c8"}
```

Now let's run the oauth part:

```
In [2]: OAUTH = OAuth2(None, None, from_file=YAHOO_FILE)  
[2021-09-07 14:43:55,761 DEBUG] [yahoo_oauth.oauth.__init__] Checking  
...  
Enter verifier :
```

A browser window will open asking you for permission to connect to your Yahoo account. You'll also see a message pop up in the REPL: `Enter verifier`

Click `Agree` and it'll take you to a page with a code. Mine is `zj2b6hx`.

Sharing approval

Use this code to connect and share your Yahoo info with fantasy football developer kit integration

zj2b6hx

Close

Figure 0.9: Yahoo Oauth Code

Copy and paste the code into the REPL, which is still waiting with its `Enter verifier` message.

Congrats, after all that we're in. Now if you look at `yahoo_credentials.json` you'll see something like:

```
{  
    "access_token": "DA0XXXXXXXXXXXX...XXXXXXXXXgs94_IyynI-",  
    "consumer_key": "dj0yJmk9VTXXXX...XXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk",  
    "consumer_secret": "56ec2XXXXXXXXXXXX8aa1eff00c8",  
    "guid": null,  
    "refresh_token": "AG2MN2HnhjgvTzcisDnRS1XXXXXXXXXXXXXX-",  
    "token_time": 1631043860.729034,  
    "token_type": "bearer"  
}
```

Part of the deal with the `yahoo_oauth` package is that we'll need to make all of our requests through it. For example, here's how we'd query the route for some general info about Yahoo fantasy:

```
In [3]: game_url = 'https://fantasysports.yahooapis.com/fantasy/v2/game/nfl'  
  
In [4]: OAuth.session.get(game_url, params={'format': 'json'}).json()  
Out[4]:  
{'fantasy_content': {'xml:lang': 'en-US',  
    'yahoo:uri': '/fantasy/v2/game/nfl',  
    'game': [{  
        'game_key': '423',  
        'game_id': '423',  
        'name': 'Football',  
        'code': 'nfl',  
        'type': 'full',  
        'url': 'https://football.fantasysports.yahoo.com/f1',  
        'season': '2023',  
        'is_registration_over': 0,  
        'is_game_over': 0,  
        'is_offseason': 0}],  
    'time': '26.293992996216ms',  
    'copyright': 'Certain Data by Sporradar, Stats Perform and Rotowire',  
    'refresh_rate': '60'}}}
```

Yahoo Endpoints

Unfortunately, the Yahoo API doesn't have a lot of good documentation. This is what I could find:

<https://developer.yahoo.com/fantasysports/guide/>

It's basically a bunch of PHP examples that appear to have been last updated in 2012. The other problem is, because the whole OAuth2 song and dance — it's more difficult to explore the API endpoints in the browser. We'll have to: (1) hit the API via the `yahoo_oauth` package, then (2) save the results to JSON on our computer, then (3) open that JSON up (just find it in your file explorer and double click on it) and look at it in the browser.

To start, we need our league id, team id and week. Let's run it:

```
In [5]:  
LEAGUE_ID = 39252  
TEAM_ID = 1  
WEEK = 2
```

Yahoo's endpoint system is sort of quirky. Every endpoint requires a "game_id", which by that they mean Yahoo's own internal game id, i.e. how Yahoo distinguishes fantasy football from everything else they have going on. It's returned in the '`game/nfl`' route we queried above. So this year (2023) the `game_id` for fantasy football is 423.

The `game_id` goes in all the endpoints, along with `league_id` and `team_id`, like this:

```
team/423.l.{LEAGUE_ID}.t.{TEAM_ID}
```

or

```
league/423.l.{LEAGUE_ID} depending if we're querying info about a league or a team.
```

We could leave it hardcoded, but then we'd have to update it next year, so instead I put it in `utilities.py` under `YAHOO_GAME_ID`. Then next year, we can change it and our code will still work.

Roster Data

Let's start with the roster data, which is available by week under the endpoint:

```
In [6]:  
roster_url = ('https://fantasysports.yahooapis.com/fantasy/v2' +  
             f'/team/{YAHOO_GAME_ID}.l.{LEAGUE_ID}.t.{TEAM_ID}/roster;week  
             ={WEEK}')
```

Again, we have to call it with oauth:

```
In [7]:  
roster_json = OAUTH.session.get(roster_url, params={'format': 'json'}).  
            json()
```

Occasionally, if you leave a session up for a long time, your `oauth` token will expire, in which case your query won't work. If that happens just run this (line 22) again:

```
OAUTH = OAuth2(None, None, from_file=YAHOO_FILE)
```

Saved Data

This is how you get live data for your league and team, but for this walkthrough I recommend using the snapshot I saved. We'll set a boolean `USED_SAVED_DATA` that controls whether we do this:

```
In [4]: USE_SAVED_DATA = True
```

Then (with this set to `True`) can load our saved data in Python like this:

```
In [8]:  
if USE_SAVED_DATA:  
    with open('./projects/integration/raw/yahoo/roster.json') as f:  
        roster_json = json.load(f)
```

Viewing Data in the Browser

Normally I like to look the JSON in the browser to figure out what we're dealing with and want. We can't do that here directly with OAuth, but we *can* view this local json file by finding it in on our computer's file system and opening in the browser.

So open up this file (remember you should have a JSON viewer installed) and take a look. You'll see:

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```

▼ fantasy_content:
  xml:lang: "en-US"
  yahoo:uri: "/fantasy/v2/team/406.l.43886.t.11/roster;week=1"
  ▼ team:
    ▶ 0: [...]
    ▼ 1:
      ▼ roster:
        ▼ 0:
          ▼ players:
            ▼ 0:
              ▼ player:
                ▼ 0:
                  player_key: "406.p.30977"
                ▼ 1:
                  player_id: "30977"
                ▼ 2:
                  ▼ name:
                    full: "Josh Allen"
                    first: "Josh"
                    last: "Allen"
                    ascii_first: "Josh"
                    ascii_last: "Allen"
                ▶ 3: {...}
              ▼ 4:
            ▼ 5:
            ▼ 6:
            ▼ 7:
            ▼ 8:
            ▼ 9:
            ▼ 10:
            ▼ 11:
            ▼ 12:
            ▼ 13:
            ▼ 14:
            ▼ 15:
            ▼ count:
  
```

Figure 0.10: Yahoo Roster JSON

From clicking around it appears the info we want is here:

```
In [9]:  
players_dict = (  
    roster_json['fantasy_content']['team'][1]['roster'][0]['players'])
```

This is a collection of players. Normally a collection like this would be a list, but Yahoo puts this in a dictionary numbered '`'0'` to however big the roster is (note the numbers are strings).

```
In [10]: players_dict.keys()  
Out[10]: dict_keys(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9',  
                   '10', '11', '12', '13', '14', '15', 'count'])
```

Also note the '`'count'` key. This is tells how many spots there are in the dict, not including `count` itself. You'll notice when we process these using dictionary comprehensions we'll have to skip over

this.

Let's start with one player:

```
In [11]: player0 = players_dict['0']

In [12]: player0
Out[12]:
{'player': [[{'player_key': '423.p.30123'},
  {'player_id': '30123'},
  {'name': {'full': 'Patrick Mahomes',
    'first': 'Patrick',
    'last': 'Mahomes',
    'ascii_first': 'Patrick',
    'ascii_last': 'Mahomes'}},
  {'url': 'https://sports.yahoo.com/nfl/players/30123'},
  {'editorial_player_key': 'nfl.p.30123'},
  {'editorial_team_key': 'nfl.t.12'},
  {'editorial_team_full_name': 'Kansas City Chiefs'},
  {'editorial_team_abbr': 'KC'},
  {'editorial_team_url': 'https://sports.yahoo.com/nfl/teams/kansas-city/'},
  {'bye_weeks': {'week': '10'}},
  {'is_keeper': {'status': False, 'cost': False, 'kept': False}},
  {'uniform_number': '15'},
  ...
  {'is_undroppable': '1'},
  {'position_type': '0'},
  {'primary_position': 'QB'},
  {'eligible_positions': [{'position': 'QB'}, {'position': 'Q/W/R/T'}]}},
  {'has_player_notes': 1},
  {'has_recent_player_notes': 1},
  {'player_notes_last_timestamp': 1694144485}],
  {'selected_position': [{"coverage_type": "week", "week": "1"},
    {"position": "QB"}],
  {'is_flex': 0}]]}
```

It's Patrick Mahomes. Nice.

I don't want to complain too much about the Yahoo API, but this is formatted very strangely, as a list of single item `key: value` dictionaries. The very first thing I want to do is write some code to convert these to normal dictionaries.

This is what I have:

```
In [13]:  
def player_list_to_dict(player):  
    player_info = player['player'][0]  
  
    player_info_dict = {}  
    for x in player_info:  
        if (type(x) is dict) and (len(x.keys()) == 1):  
            for key in x.keys():  
                player_info_dict[key] = x[key]  
    return player_info_dict
```

It's just going through each list item and adding it to one big dictionary. Let's try it:

```
In [14]: player_list_to_dict(player0)  
Out[14]:  
{'player_key': '423.p.30123',  
 'player_id': '30123',  
 'name': {'full': 'Patrick Mahomes',  
 'first': 'Patrick',  
 'last': 'Mahomes',  
 'ascii_first': 'Patrick',  
 'ascii_last': 'Mahomes'},  
 'url': 'https://sports.yahoo.com/nfl/players/30123',  
 ...  
 'uniform_number': '15',  
 'display_position': 'QB',  
 'is_undroppable': '1',  
 'position_type': '0',  
 'primary_position': 'QB',  
 'eligible_positions': [{position: 'QB'}, {position: 'Q/W/R/T'}],  
 'has_player_notes': 1,  
 'has_recent_player_notes': 1,  
 'player_notes_last_timestamp': 1694144485}
```

That's better. OK.

Remember what we need to get here for each player:

1. the Yahoo player id
2. whether we're starting the player and the position he's in if we are (e.g. if it's an RB, is he's in the RB spot or flex)
3. the player's position and name

Let's write a function to get that. The details of this are all based on the JSON data in the browser:

```
In [15]:  
def process_player(player):  
    player_info = player_list_to_dict(player)  
    pos_info = player['player'][1]['selected_position'][1]  
  
    dict_to_return = {}  
    dict_to_return['yahoo_id'] = int(player_info['player_id'])  
    dict_to_return['name'] = player_info['name']['full']  
    dict_to_return['player_position'] = player_info['primary_position']  
    dict_to_return['team_position'] = pos_info['position']  
  
    return dict_to_return
```

And running it:

```
In [16]: process_player(player0)  
Out[16]:  
{'yahoo_id': 30123,  
 'name': 'Patrick Mahomes',  
 'player_position': 'QB',  
 'team_position': 'QB'}
```

This looks good. Let's run it on every player in our `players_dict`. Note the list comprehension, and remember `players_dict` is a dict with a random '`count`' key in it:

```
In [17]: [process_player(player) for key, player
          in players_dict.items() if key != 'count']

Out[17]:
[{'yahoo_id': 30123,
 'name': 'Patrick Mahomes',
 'player_position': 'QB',
 'team_position': 'QB'},
 {'yahoo_id': 33500,
 'name': 'Amon-Ra St. Brown',
 'player_position': 'WR',
 'team_position': 'WR'},
 {'yahoo_id': 30996,
 'name': 'Calvin Ridley',
 'player_position': 'WR',
 'team_position': 'WR'},
 {'yahoo_id': 31005,
 'name': 'Nick Chubb',
 'player_position': 'RB',
 'team_position': 'RB'},
 {'yahoo_id': 31934,
 'name': 'Alexander Mattison',
 'player_position': 'RB',
 'team_position': 'RB'},
 {'yahoo_id': 31019,
 'name': 'Dallas Goedert',
 'player_position': 'TE',
 'team_position': 'TE'},
 {'yahoo_id': 27535,
 'name': 'Mike Evans',
 'player_position': 'WR',
 'team_position': 'W/R/T'},
 {'yahoo_id': 32675,
 'name': 'Tua Tagovailoa',
 'player_position': 'QB',
 'team_position': 'Q/W/R/T'},
 {'yahoo_id': 33408,
 'name': 'Kadarius Toney',
 'player_position': 'WR',
 'team_position': 'BN'},
 {'yahoo_id': 34019,
 'name': 'James Cook',
 'player_position': 'RB',
 'team_position': 'BN'},
 ...]
```

Now that we have a list of dictionaries with the same fields (`name`, `player_position`, `team_position`, `yahoo_id`) we should put them in a DataFrame ASAP.

```
In [18]:  
players_df = DataFrame(  
    [process_player(player) for key, player in players_dict.items() if key  
     !=  
     'count' ])  
  
In [19]: players_df  
Out[19]:  
      yahoo_id           name  player_position  team_position  
0       30123   Patrick Mahomes          QB          QB  
1       33500  Amon-Ra St. Brown          WR          WR  
2       30996    Calvin Ridley          WR          WR  
3       31005    Nick Chubb          RB          RB  
4       31934 Alexander Mattison          RB          RB  
5       31019  Dallas Goedert          TE          TE  
6       27535     Mike Evans          WR  W/R/T  
7       32675  Tua Tagovailoa          QB  Q/W/R/T  
8       33408   Kadarius Toney          WR          BN  
9       34019     James Cook          RB          BN  
10      31017  Christian Kirk          WR          BN  
11      30971   Baker Mayfield          QB          BN  
12      31268    Allen Lazard          WR          BN  
13      33422    Elijah Moore          WR          BN  
14      30182    Cooper Kupp          WR          IR
```

This looks pretty good, but at some point we're probably going to want to refer to players on our roster by position, e.g. our RB1, RB2, etc

Let's add that. As always let's start with a specific example, say WRs:

```
In [20]: wrs = players_df.query("team_position == 'WR'")  
  
In [21]: wrs  
Out[21]:  
      yahoo_id           name  player_position  team_position  
1       33500  Amon-Ra St. Brown          WR          WR  
2       30996    Calvin Ridley          WR          WR
```

So we want `team_position` to say WR1, WR2 instead of just WR, WR.

The easiest way is probably to make a column with 1, 2 then stick them together.

```
In [22]: suffix = Series(range(1, len(wrs) + 1), index=wrs.index)  
  
In [23]: suffix  
Out[23]:  
1    1  
2    2
```

The key is we're making this column have the same index as `wrs`, that way we can do this:

```
In [24]: wrs['team_position'] + suffix.astype(str)
Out[24]:
1    WR1
2    WR2
dtype: object
```

Which is what we want. Now let's put this in a function:

```
In [25]:
def add_pos_suffix(df_subset):
    if len(df_subset) > 1:
        suffix = Series(range(1, len(df_subset) + 1), index=df_subset.
            index)

        df_subset['team_position'] = (
            df_subset['team_position'] + suffix.astype(str))
    return df_subset
```

and apply it to every position in our DataFrame.

```
In [26]:
players_df2 = pd.concat([
    add_pos_suffix(players_df.query(f"team_position == '{x}'"))
    for x in players_df['team_position'].unique()])

In [27]: players_df2
Out[27]:
   yahoo_id           name player_position team_position
0     30123      Patrick Mahomes         QB          QB
1     33500  Amon-Ra St. Brown         WR         WR1
2     30996      Calvin Ridley         WR         WR2
3     31005      Nick Chubb          RB         RB1
4     31934  Alexander Mattison         RB         RB2
5     31019    Dallas Goedert          TE          TE
6     27535      Mike Evans          WR      W/R/T
7     32675    Tua Tagovailoa         QB      Q/W/R/T
8     33408    Kadarius Toney          WR         BN1
9     34019      James Cook          RB         BN2
10    31017    Christian Kirk          WR         BN3
11    30971    Baker Mayfield         QB         BN4
12    31268      Allen Lazard          WR         BN5
13    33422      Elijah Moore          WR         BN6
14    30182      Cooper Kupp          WR          IR
```

Cool, there's our player DataFrame. Now let's identify our starters:

```
In [28]:  
players_df2['start'] = ~(players_df2['team_position'].str.startswith('BN')  
|  
    players_df2['team_position'].str.startswith('IR'))  
In [29]: players_df2  
Out[29]:  
   yahoo_id          name player_position team_position  start  
0     30123  Patrick Mahomes           QB            QB  True  
1     33500  Amon-Ra St. Brown          WR           WR1  True  
2     30996    Calvin Ridley           WR           WR2  True  
3     31005      Nick Chubb           RB           RB1  True  
4     31934 Alexander Mattison          RB           RB2  True  
5     31019    Dallas Goedert           TE            TE  True  
6     27535      Mike Evans           WR           W/R/T  True  
7     32675    Tua Tagovailoa          QB           Q/W/R/T  True  
8     33408    Kadarius Toney          WR           BN1 False  
9     34019      James Cook           RB           BN2 False  
10    31017    Christian Kirk          WR           BN3 False  
11    30971    Baker Mayfield          QB           BN4 False  
12    31268      Allen Lazard           WR           BN5 False  
13    33422      Elijah Moore           WR           BN6 False  
14    30182      Cooper Kupp           WR            IR False
```

This is really close to what we want. We just need a team id, and the corresponding fantasymath player ids.

Let's stop and put everything we've done so far into a function.

```
def process_players(players):  
    players_raw = DataFrame(  
        [process_player(player) for key, player in players.items() if key  
         !=  
         'count'])  
  
    players_df = pd.concat([  
        add_pos_suffix(players_raw.query(f"team_position == '{x}'"))  
        for x in players_raw['team_position'].unique()])  
  
    players_df['start'] = ~(players_df['team_position'].str.startswith('BN')  
    |  
        players_df['team_position'].str.startswith('IR')))  
  
    return players_df
```

I'm not showing it here because I don't think we need to show the same data over and over, but you should try this out on `players_dict` to make sure it works.

Now let's think about team id. Looking at the JSON, it looks like team id is further up from

`players_dict`. Specifically, it's here:

```
In [30]: team_id = roster_json['fantasy_content']['team'][0][1]['team_id']

In [31]: team_id
Out[31]: '1'
```

So what we need to do is build a function that wraps (i.e. calls) both these functions inside of it, adds team id, then returns it.

Something like this:

```
In [32]:
def process_roster(team):
    team_df = process_players(team[1]['roster']['0']['players'])
    team_id = team[0][1]['team_id']

    team_df['team_id'] = team_id
    return team_df

In [33]: roster_df = process_roster(roster_json['fantasy_content']['team'])

In [34]: roster_df
Out[34]:
   yahoo_id           name  ...  start  team_id
0     30123  Patrick Mahomes  ...   True      1
1     33500  Amon-Ra St. Brown  ...   True      1
2     30996      Calvin Ridley  ...   True      1
3     31005       Nick Chubb  ...   True      1
4     31934  Alexander Mattison  ...   True      1
5     31019        Dallas Goedert  ...   True      1
6     27535        Mike Evans  ...   True      1
7     32675      Tua Tagovailoa  ...   True      1
8     33408      Kadarius Toney  ...  False      1
9     34019        James Cook  ...  False      1
10    31017  Christian Kirk  ...  False      1
11    30971      Baker Mayfield  ...  False      1
12    31268      Allen Lazard  ...  False      1
13    33422      Elijah Moore  ...  False      1
14    30182      Cooper Kupp  ...  False      1
```

What else? Well, this example snapshot of data we're using is from Week 2, Friday morning; right after PHI-MIN played Thursday night. In that case Mattison (Vikings) and Goedert (Eagles) will have played and have actual scores. We want to get these too.

After playing around with the API, I found these point values in another endpoint:

```
In [1]:  
points_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +  
             f'team/{YAHOO_GAME_ID}.l.{LEAGUE_ID}.t.{TEAM_ID}' +  
             f'/players/stats?type=week;week={WEEK};out=stats')
```

This is how we'd get it with our same `OAUTH` object (you can keep using the saved data so that we're using the same thing):

```
In [2]:  
points_json = OAUTH.session.get(points_url, params={'format': 'json'}).  
            json()
```

Looking at this in the browser, it appears the collection of players and scores is here:

```
In [3]:  
player_dict = points_json['fantasy_content']['team'][1]['players']
```

This is the usual dict of numbers as strings. And Goedert (one of the guys who already played) is this one:

```
In [4]: goedert = player_dict['7']
```

This function will get what we need:

```
def process_player_stats(player):  
    dict_to_return = {}  
    dict_to_return['yahoo_id'] = int(player['player'][0][1]['player_id'])  
    dict_to_return['actual'] = float(  
        player['player'][1]['player_points']['total'])  
    return dict_to_return
```

Running it on Goedert (who had 6 catches for 22 yards), which is 8.2 points in our PPR league.

```
In [5]: process_player_stats(goedert)  
Out[5]: {'yahoo_id': 31019, 'actual': 8.2}
```

Great. Now let's put it in a function to run it on every player:

```
def process_team_stats(team):  
    stats = DataFrame([process_player_stats(player) for key, player in  
                     team.items() if key != 'count'])  
    stats.loc[stats['actual'] == 0, 'actual'] = np.nan  
    return stats
```

Calling it (Mattison is the other guy who played):

```
In [6]: stats
Out[6]:
    yahoo_id  actual
0      27535    NaN
1      30123    NaN
2      30182    NaN
3      30971    NaN
4      30996    NaN
5      31005    NaN
6      31017    NaN
7      31019    8.2
8      31268    NaN
9      31934    4.9
10     32675    NaN
11     33408    NaN
12     33422    NaN
13     33500    NaN
14     34019    NaN
```

Finally we just have to merge it back into roster:

```
In [7]: roster_df_w_stats = pd.merge(roster_df, stats)

In [8]: roster_df_w_stats.head(10)
Out[8]:
    yahoo_id          name  ...  team_id  actual
0      30123  Patrick Mahomes  ...      1    NaN
1      33500  Amon-Ra St. Brown  ...      1    NaN
2      30996    Calvin Ridley  ...      1    NaN
3      31005    Nick Chubb  ...      1    NaN
4      31934  Alexander Mattison  ...      1    4.9
5      31019    Dallas Goedert  ...      1    8.2
6      27535    Mike Evans  ...      1    NaN
7      32675   Tua Tagovailoa  ...      1    NaN
8      33408   Kadarius Toney  ...      1    NaN
9      34019    James Cook  ...      1    NaN
```

Now we just need to be able to connect it to our Fantasy Math simulation data. We could try merging on `name`, but that won't always work — players' names aren't the same on every platform + there's nicknames, jr's and sr's, duplicates etc.

So instead, we'll do something easier —

— I'll take of it for you behind the scenes. There's a utility function `master_player_lookup` that return a master lookup of everyone's player ids. Like all the other functions, it takes a `token`.

```
In [9]: from utilities import (
    LICENSE_KEY, generate_token, master_player_lookup)

In [10]:
token = generate_token(LICENSE_KEY)['token']
fantasymath_players = master_player_lookup(token)
```

As with the other data, I've saved a snapshot.

Here's what it looks like:

```
In [11]: fantasymath_players.head()
Out[11]:
   player_id  pos  fleaflicker_id  espn_id  yahoo_id  sleeper_id
0           1   QB        17761  4685201      40220       9230
1           2   QB        17602  4685720      40029       9228
2           3   QB        17623  4361418      40068       9999
3           4   QB        17589  4432577      40030       9758
4           6   QB        17586  4429084      40040       9229
```

Now we can link this up with the roster from above and we'll be set.

```
In [12]:
roster_df_w_id = pd.merge(roster_df_w_stats,
                           fantasymath_players[['player_id', 'yahoo_id']],
                           how='left')
```

Remember the fields we said we wanted at the start of this project:

- `player_id`
- `name`
- `player_position`
- `team_position`
- `start`
- `actual`
- `team_id`

That's exactly what we have (we also have `yahoo_id`, which we'll drop).

The only problem is we said we wanted rosters for *all* teams in league, not just one. And we want to be able to get them with just a `league_id`, we don't necessarily want to have to pass the `team_id` in every time.

In the next section we'll hit another endpoint to get info on every team in a league, so this part will have to wait.

In the meantime though we can put all this in a function. To get the above we basically need the following:

- Yahoo league and team ids
- our player-Yahoo id lookup table

Let's do it:

```
def get_team_roster(team_id, league_id, week, lookup):  
    roster_url = ('https://fantasysports.yahooapis.com/fantasy/v2' +  
                  f'/team/{YAHOO_GAME_ID}.l.{league_id}.t.{team_id}/roster  
                  ;week={week}')  
  
    if USE_SAVED_DATA:  
        with open('./projects/integration/raw/yahoo/roster.json') as f:  
            roster_json = json.load(f)  
    else:  
        roster_json = OAUTH.session.get(roster_url, params={'format': 'json'}).json()  
  
    roster_df = process_roster(roster_json['fantasy_content']['team'])  
  
    # stats  
    points_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +  
                  f'team/{YAHOO_GAME_ID}.l.{league_id}.t.{team_id}' +  
                  f'/players/stats?type=week;week={week};out=stats')  
  
    if USE_SAVED_DATA:  
        with open('./projects/integration/raw/yahoo/points.json') as f:  
            points_json = json.load(f)  
    else:  
        points_json = OAUTH.session.get(points_url, params={'format': 'json'}).json()  
  
    player_dict = points_json['fantasy_content']['team'][1]['players']  
    stats = process_team_stats(player_dict)  
    roster_df_w_stats = pd.merge(roster_df, stats)  
  
    roster_df_w_id = pd.merge(roster_df_w_stats,  
                             lookup[['player_id', 'yahoo_id']],  
                             how='left').drop('yahoo_id', axis=1)  
  
    return roster_df_w_id
```

Looking at it:

```
In [13]: my_roster
my_roster = get_team_roster(TEAM_ID, LEAGUE_ID, 1, fantasymath_players)

In [14]: my_roster
Out[14]:
          name player_position ... actual player_id
0    Patrick Mahomes        QB ...   NaN      1091
1  Amon-Ra St. Brown       WR ...   NaN      377
2    Calvin Ridley        WR ...   NaN      929
3     Nick Chubb         RB ...   NaN      910
4  Alexander Mattison      RB ...  4.9      772
5    Dallas Goedert        TE ...  8.2      974
6     Mike Evans          WR ...   NaN     1716
7    Tua Tagovailoa        QB ...   NaN      493
8    Kadarius Toney        WR ...   NaN      379
9     James Cook          RB ...   NaN      172
10   Christian Kirk        WR ...   NaN      933
11   Baker Mayfield        QB ...   NaN      890
12    Allen Lazard         WR ...   NaN      932
13   Elijah Moore          WR ...   NaN      378
14   Cooper Kupp          WR ...   NaN     1139
```

Awesome. This was the most complicated piece of the league connection project, and we've knocked it out.

We're not quite finished yet though, let's grab our team data so we can get complete league rosters.

Team Data

Now we need team data, specifically:

- `team_id`
- `owner_id`
- `owner_name`
- `league_id`

Team data is available in the standings endpoint.

```
In [1]:
teams_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +
             f'league/{YAHOO_GAME_ID}.l.{LEAGUE_ID}' +
             '?out=metadata,settings,standings,scoreboard,teams,players,' +
             'draftresults,transactions')
```

Let's get it using our same `oauth` object.

```
In [2]: teams_json = OAuth.session.get(teams_url,
                                         params={'format': 'json'}).json()
```

Again, that's how we'd get the live data, but to load the example data and follow along:

```
In [3]:
with open('./projects/integration/raw/yahoo/teams.json') as f:
    teams_json = json.load(f)
```

Opening this in the browser and looking at it, our team data is here:

```
In [4]: teams_dict = (
    teams_json['fantasy_content']['league'][2]['standings'][0]['teams'])
```

Just like last time, this is a dict with keys ranging from 0 to 12 as strings (plus 'count'):

```
In [5]: teams_dict.keys()
Out[5]: dict_keys(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'count'])
```

So this is a single team:

```
In [6]: team0 = teams_dict['0']

In [7]: team0
Out[7]:
{'team': [[{'team_key': '423.l.39252.t.1'},
  {'team_id': '1'},
  {'name': 'Bus Riders in the Sky'},
  [],
  {'url': 'https://football.fantasysports.yahoo.com/f1/39252/1'},
  {'team_logos': [{"team_logo": {"size": 'large',
    'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/t_s192sq/fantasy-logos/66f51a2b5c9abff155afc056a184a283a2cec3bf7fb964500f7b0ab23590327.jpg"}}],
  [],
  {'waiver_priority': 6},
  [],
  {'number_of_moves': 1},
  {'number_of_trades': 0},
  {'roster_adds': {'coverage_type': 'week',
    'coverage_value': 1,
    'value': '1'}},
  [],
  {'league_scoring_type': 'head'},
  [],
  [],
  {'has_draft_grade': 1,
    'draft_grade': 'A+',
    'draft_recap_url': 'https://football.fantasysports.yahoo.com/f1/39252/1/draftrecap'},
  [],
  [],
  {'managers': [{"manager": {'manager_id': '1',
      'nickname': 'Tyler',
      'guid': '7MIKYOTUTFOVHTULSLCSDCD6CE',
      'is_commissioner': '1',
      'image_url': 'https://s.yimg.com/ag/images/4561/37418383823_724def_64sq.jpg',
      'felo_score': '922',
      'felo_tier': 'diamond'}}]}],
  {'team_points': {'coverage_type': 'season', 'season': '2023', 'total': '0'}},
  {'team_standings': {'rank': '',
    'outcome_totals': {'wins': 0, 'losses': 0, 'ties': 0, 'percentage': ''}},
  'points_for': '0',
  'points_against': '0'}]]
```

Again, just like with players, each team is structured as a list of single item dicts. Further up we built a helpful function to turn this into something more logical:

```
def player_list_to_dict(player):
    player_info = player['player'][0]

    player_info_dict = {}
    for x in player_info:
        if (type(x) is dict) and (len(x.keys()) == 1):
            for key in x.keys(): # tricky way to get access to key
                player_info_dict[key] = x[key]
    return player_info_dict
```

Let's modify this function to make it more general — able to take a player *or* team:

```
In [8]:
def yahoo_list_to_dict(yahoo_list, key):
    return_dict = {}
    for x in yahoo_list[key][0]:
        if (type(x) is dict) and (len(x.keys()) == 1):
            for key_ in x.keys(): # tricky way to get access to key
                return_dict[key_] = x[key_]
    return return_dict
```

And trying it out:

```
In [9]: yahoo_list_to_dict(team_0, 'team')
Out[9]:
{'team_key': '423.l.39252.t.1',
 'team_id': '1',
 'name': 'Bus Riders in the Sky',
 'url': 'https://football.fantasysports.yahoo.com/f1/39252/1',
 'team_logos': [{team_logo: {'size': 'large',
                            'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/t_s192sq/fantasy-logos/66f51a2b5c9abff155afc056a184a283a2cec3bf7bfb964500f7b0ab23590327.jpg'}}],
 'waiver_priority': 6,
 'number_of_moves': 1,
 'number_of_trades': 0,
 'roster_adds': {'coverage_type': 'week', 'coverage_value': 1, 'value': '1'},
 'league_scoring_type': 'head',
 'managers': [{"manager": {"manager_id": '1',
                           'nickname': 'Tyler',
                           'guid': '7MIKYOTUTF0VHTULSLCSDCD6CE',
                           'is_commissioner': '1',
                           'image_url': 'https://s.yimg.com/ag/images/4561/37418383823_724def_64sq.jpg',
                           'felo_score': '922',
                           'felo_tier': 'diamond'}}]}
```

Nice. Ok, now let's use this and create a function to return just the fields we want from a given team.

```
In [10]:  
def process_team(team):  
    team_dict = yahoo_list_to_dict(team, 'team')  
    owner_dict = team_dict['managers'][0]['manager']  
  
    dict_to_return = {}  
    dict_to_return['team_id'] = team_dict['team_id']  
    dict_to_return['owner_id'] = owner_dict['guid']  
    dict_to_return['owner_name'] = owner_dict['nickname']  
    return dict_to_return  
  
In [11]: process_team(team0)  
Out[11]:  
{'team_id': '1',  
 'owner_id': '7MIKYOTUTFOVHTULSLCSDCD6CE',  
 'owner_name': 'Tyler'}
```

Now let's write a function where we call this for every team, and also add in `league_id` while we're at it. Remember, we said this function would be called `get_teams_in_league`.

```
In [12]:
def get_teams_in_league(league_id):
    teams_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +
                 f'league/{YAHOO_GAME_ID}.l.{league_id}' + 
                 ';out=metadata,settings,standings,scoreboard,teams,players
                 ,draftresults,transactions')

    if USE_SAVED_DATA:
        with open('./projects/integration/raw/yahoo/teams.json') as f:
            teams_json = json.load(f)
    else:
        teams_json = (OAUTH
                      .session
                      .get(teams_url, params={'format': 'json'})
                      .json())

    teams_dict = (teams_json['fantasy_content']
                  ['league'][2]['standings'][0]['teams'])

    teams_df = DataFrame([process_team(team) for key, team in
                           teams_dict.items() if key != 'count'])

    teams_df['league_id'] = league_id
    return teams_df

In [13]: league_teams = get_teams_in_league(LEAGUE_ID)

In [14]: league_teams
Out[14]:
   team_id          owner_id owner_name  league_id
0      1  7MIKYOTUTFOVHTULSLCSDCD6CE      Tyler  39252
1      2  4FOCKGEBSXA6SH5NN3VFLNYRR4      Paul  39252
2      3  CNHW7F5KWZHTDOPC3KQJLLCS43     Ross  39252
3      4  XWDHEODK4B6DE5KRQ4FNZW37DE    Oscar  39252
4      5  77UWSX5MXPDQ7PBKR6BQDCZFI4     Reed  39252
5      6  727PP4JCVHEKP6COKH35Q0QVRY      Paul  39252
6      7  IA4UIFKQF2VCY3GISWAL7TPDDI    Craig  39252
7      8  HDIL7ZMDK4GQEOPKX5JB5ELNEI  M-BABBS  39252
8      9  BZ7HBQDAAYYINXANARENS7D7GY    Patrick  39252
9     10  LLZK43EGZ3BEYYK6534WUIF4I4      Will  39252
```

Ok. So the team portion is done. We can also combine it with our `get_team_roster` function above to get `get_league_rosters`, which is what we wanted. It's straightforward:

```
def get_league_rosters(lookup, league_id, week):
    teams = get_teams_in_league(league_id)

    league_rosters = pd.concat(
        [get_team_roster(x, league_id, week, lookup, use_saved_data=False)
         for x in
            teams['team_id']]], ignore_index=True)
    league_rosters['team_position'].replace({'W/R/T': 'WR/RB/TE'}, inplace
        =True)
    return league_rosters
```

Let's test it out:

```
In [15]: league_rosters = get_league_rosters(fantasymath_players, LEAGUE_ID, WEEK)

In [16]: league_rosters.sample(20)
Out[16]:
```

		name	player_position	...	actual	player_id
124	Ezekiel Elliott		RB	...	NaN	1302
35	DeAndre Hopkins		WR	...	NaN	1964
94	Diontae Johnson		WR	...	NaN	770
130	Derrick Henry		RB	...	NaN	1303
82	Jerry Jeudy		WR	...	NaN	548
58	Tee Higgins		WR	...	NaN	550
36	Jared Goff		QB	...	NaN	1288
15	Kirk Cousins		QB	...	40.2	2085
69	Brandin Cooks		WR	...	NaN	1719
79	Rachaad White		RB	...	NaN	178
139	Joshua Kelley		RB	...	NaN	522
99	DeVonta Smith		WR	...	23.1	372
56	Antonio Gibson		RB	...	NaN	567
11	Baker Mayfield		QB	...	NaN	890
122	AJ Dillon		RB	...	NaN	520
16	Brandon Aiyuk		WR	...	NaN	555
29	Justin Herbert		QB	...	NaN	494
126	Joe Burrow		QB	...	NaN	492
19	David Montgomery		RB	...	NaN	689
78	Trevor Lawrence		QB	...	NaN	327

Perfect.

Schedule Info

The last thing we need is the schedule.

This endpoint returns every matchup for a given team id:

```
In [1]:  
# schedule info  
schedule_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +  
                 f'team/{YAHOO_GAME_ID}.l.{LEAGUE_ID}.t.{TEAM_ID}' +  
                 ';out=matchups')
```

Let's get it using our same `oauth` object.

```
In [2]:  
schedule_json = OAUTH.session.get(schedule_url, params={'format': 'json'})  
.json()
```

And again, working with the example data for walk through purposes:

```
In [3]:  
with open('./projects/integration/raw/yahoo/schedule.json') as f:  
    schedule_json = json.load(f)
```

Opening this up in the browser and looking at it, our schedule data is here:

```
In [4]:  
matchups_dict = schedule_json['fantasy_content']['team'][1]['matchups']
```

This is our usual dict of items, so let's look at one:

```
In [5]: matchup0 = matchups_dict['0']

In [6]: matchup0
Out[6]:
{'matchup': {'week': '1',
'week_start': '2023-09-07',
'week_end': '2023-09-11',
'status': 'midevent',
'is_playoffs': '0',
'is_consolation': '0',
'is_matchup_recap_available': 0,
'0': {'teams': {'0': {'team': [[{'team_key': '423.l.39252.t.1'},
{'team_id': '1'},
{'name': 'Bus Riders in the Sky'},
[], {'url': 'https://football.fantasysports.yahoo.com/f1/39252/1'},
{'team_logos': [{team_logo: {'size': 'large',
'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/t_s192sq/fantasy-logos/66f51a2b5c9abff155afc056a184a283a2cec3bf7fb964500f7b0ab23590327.jpg'}}]}],
[], {'waiver_priority': 6},
[], {'number_of_moves': 1},
{'number_of_trades': 0},
...
{'managers': [{'manager': {'manager_id': '8',
'nickname': 'M-BABBS',
'guid': 'HDIL7ZMDK4GQEOPKX5JB5ELNEI',
'image_url': 'https://s.yimg.com/ag/images/4554/37880806840_bf04df_64sq.jpg',
'felo_score': '704',
'felo_tier': 'gold'}]}],
{'win_probability': 0.4,
'team_points': {'coverage_type': 'week', 'week': '1', 'total': '0.00'},
'team_projected_points': {'coverage_type': 'week',
'week': '1',
'total': '120.16'}]}},
'count': 2}}}
```

Playing around with a bit, information on the two teams in this matchup is available in the '0' and '1' in `matchup_0['matchup']['0']['teams']`.

```
In [7]: matchup_0['matchup']['0']['teams']['0']
Out[7]:
{'team': [{{'team_key': '423.l.39252.t.1'},
  {'team_id': '1'},
  {'name': 'Bus Riders in the Sky'},
  [],
  {'url': 'https://football.fantasysports.yahoo.com/f1/39252/1'},
  {'team_logos': [{team_logo: {'size': 'large',
    'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/t_s192sq/fantasy-logos/66f51a2b5c9abff155afc056a184a283a2cec3bf7bfb964500f7b0ab23590327.jpg'}}}],
  [],
  {'waiver_priority': 6},
  [],
  {'number_of_moves': 1},
  {'number_of_trades': 0},
  {'roster_adds': {'coverage_type': 'week',
    'coverage_value': 1,
    'value': '1'}},
  [],
  {'league_scoring_type': 'head'},
  [],
  [],
  {'has_draft_grade': 1,
    'draft_grade': 'A+',
    'draft_recap_url': 'https://football.fantasysports.yahoo.com/f1/39252/1/draftrecap'},
  [],
  [],
  {'managers': [{manager: {'manager_id': '1',
    'nickname': 'Tyler',
    'guid': '7MIKYOTUTFOVHTULSLCSDCD6CE',
    'is_commissioner': '1',
    'image_url': 'https://s.yimg.com/ag/images/4561/37418383823_724def_64sq.jpg',
    'feo_score': '922',
    'feo_tier': 'diamond'}}}],
  {'win_probability': 0.6,
    'team_points': {'coverage_type': 'week', 'week': '1', 'total': '43.90'},
    'team_projected_points': {'coverage_type': 'week',
      'week': '1',
      'total': '132.01'}}}]}
```

This is the usual list of single item dicts, so we can use our `yahoo_list_to_dict` function on it, then extract the `team_id`. That and week should be all we need.

```
In [8]:  
matchup0_team0 = yahoo_list_to_dict(  
    matchup0['matchup']['0']['teams']['0'], 'team')  
  
In [9]:  
matchup0_team1 = yahoo_list_to_dict(  
    matchup0['matchup']['0']['teams']['1'], 'team')  
  
In [16]: matchup0_team0['team_id']  
Out[16]: '1'  
  
In [17]: matchup0_team1['team_id']  
Out[17]: '8'  
  
In [18]: matchup0['matchup']['week']  
Out[18]: '1'
```

As always, let's put this in a function:

```
In [13]:  
def process_matchup(matchup):  
    team0 = yahoo_list_to_dict(matchup['matchup'][0]['teams'][0], 'team')  
    team1 = yahoo_list_to_dict(matchup['matchup'][0]['teams'][1], 'team')  
  
    dict_to_return = {}  
    dict_to_return['team_id'] = team0['team_id']  
    dict_to_return['opp_id'] = team1['team_id']  
    dict_to_return['week'] = matchup['matchup']['week']  
  
    return dict_to_return  
  
In [14]: process_matchup(matchup_0)  
Out[14]: {'team_id': '1', 'opp_id': '8', 'week': '1'}
```

Now let's run it on every matchup:

```
In [15]:  
DataFrame([process_matchup(matchup)  
          for key, matchup  
          in matchup_dict.items()  
          if key != 'count'])  
  
--  
Out[15]:  
   team_id  opp_id  week  
0         1       8     1  
1         1      10     2  
2         1       9     3  
3         1       2     4  
4         1       6     5  
5         1       3     6  
6         1       4     7  
7         1       5     8  
8         1       7     9  
9         1       8    10  
10        1      10    11  
11        1       9    12  
12        1       2    13  
13        1       6    14  
14        1       3    15
```

That's pretty good. The only thing Yahoo didn't include anything like a matchup id, which is something we said we wanted.

So let's make it. What uniquely identifies a matchup? How about: season, week, and the two team ids, sorted in ascending order. Something like:

```
def make_matchup_id(season, week, team1, team2):  
    teams = [team1, team2]  
    teams.sort()  
  
    return int(str(season) + str(week).zfill(2) +  
               str(teams[0]).zfill(2) + str(teams[1]).zfill(2))
```

So for 2023, week 1, team 1 vs team 8 we'd have:

```
In [16]: make_matchup_id(2023, 1, 1, 8)  
Out[16]: 2023010108
```

And let's make sure switching the team ids around gives us the same thing:

```
In [17]: make_matchup_id(2023, 1, 8, 1)  
Out[17]: 2023010108
```

Perfect. Let's add it to our `process_matchup` function:

```
def process_matchup2(matchup):
    team0 = yahoo_list_to_dict(matchup['matchup']['0']['teams']['0'],
        'team')
    team1 = yahoo_list_to_dict(matchup['matchup']['0']['teams']['1'],
        'team')

    dict_to_return = {}
    dict_to_return['team_id'] = team0['team_id']
    dict_to_return['opp_id'] = team1['team_id']
    dict_to_return['week'] = matchup['matchup']['week']
    dict_to_return['matchup_id'] = make_matchup_id(
        SEASON, matchup['matchup']['week'], team0['team_id'], team1['team_id'])

    return dict_to_return
```

Now let's write a function to get the schedule for any team:

```
def get_schedule_by_team(team_id, league_id):
    schedule_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +
        f'team/{YAHOO_GAME_ID}.l.{league_id}.t.{team_id}' +
        ';out=matchups')

    schedule_raw = OAUTH.session.get(schedule_url, params={'format': 'json'}).json()
    matchup_dict = schedule_raw['fantasy_content']['team'][1]['matchups']
    df = DataFrame([process_matchup2(matchup)
        for key, matchup
        in matchup_dict.items()
        if key != 'count'])
    df['season'] = SEASON
    return df
```

And run it for every team:

```
In [18]:  
all_team_schedules = pd.concat([get_schedule_by_team(x, LEAGUE_ID) for x  
    in  
        league_teams['team_id']], ignore_index=  
        True)  
  
In [19]: full_team_schedules.head(20)  
Out[19]:  
   team_id  opp_id  week  matchup_id  season  
0         1       8     1  2021010108  2023  
1         1      10     2  2021020110  2023  
2         1       9     3  2021030109  2023  
3         1       2     4  2021040102  2023  
4         1       6     5  2021050106  2023  
5         1       3     6  2021060103  2023  
6         1       4     7  2021070104  2023  
7         1       5     8  2021080105  2023  
8         1       7     9  2021090107  2023  
9         1       8    10  2021100108  2023  
10        1      10    11  2021110110  2023  
11        1       9    12  2021120109  2023  
12        1       2    13  2021130102  2023  
13        1       6    14  2021140106  2023  
14        1       3    15  2021150103  2023  
15        2       4     1  2021010204  2023  
16        2       9     2  2021020209  2023  
17        2       5     3  2021030205  2023  
18        2       1     4  2021040102  2023  
19        2       7     5  2021050207  2023
```

This is close to what we want, but not exactly. Remember we wanted the `schedule` table to include: `team1_id`, `team2_id`, `matchup_id`, `season`, `week` and `league_id` columns.

The main thing is this is by team *and* week, when really we just want it by week. Since it doesn't matter which team is which, we can just do it like this:

```
In [20]: schedule_by_week = all_team_schedules.drop_duplicates('matchup_id')
        )

In [21]: schedule_by_week.columns = ['team1_id', 'team2_id', 'week',
        'matchup_id', 'season']

In [22]: schedule_by_week.head(10)
Out[22]:
   team1_id  team2_id  week  matchup_id  season
0          1         1     8  2021010108    2023
1          1         1    10  2021020110    2023
2          1         1     9  2021030109    2023
3          1         2     4  2021040102    2023
4          1         1     6     5  2021050106    2023
5          1         1     3     6  2021060103    2023
6          1         1     4     7  2021070104    2023
7          1         1     5     8  2021080105    2023
8          1         1     7     9  2021090107    2023
9          1         1    10  2021100108    2023
```

That's what we want, and we want a function `get_league_schedule` to return it. It will: (1) get our team data, then (2) call `get_schedule_by_team` on every team, (3) reshape that data so each line is a game.

```
def get_league_schedule(league_id):
    league_teams = get_teams_in_league(league_id)

    schedule_by_team = pd.concat([get_schedule_by_team(x, league_id) for
                                  x in league_teams['team_id']],
                                  ignore_index=True)

    schedule_by_week = schedule_by_team.drop_duplicates('matchup_id')

    schedule_by_week.columns = ['team1_id', 'team2_id', 'week',
                               'matchup_id',
                               'season']
    return schedule_by_week
```

```
In [23]: league_schedule = get_league_schedule(LEAGUE_ID)
```

```
In [24]: league_schedule.head(10)
```

```
Out[24]:
```

	team1_id	team2_id	week	matchup_id	season
0	1	8	1	2021010108	2023
1	1	10	2	2021020110	2023
2	1	9	3	2021030109	2023
3	1	2	4	2021040102	2023
4	1	6	5	2021050106	2023
5	1	3	6	2021060103	2023
6	1	4	7	2021070104	2023
7	1	5	8	2021080105	2023
8	1	7	9	2021090107	2023
9	1	8	10	2021100108	2023

Wrap Up

To wrap up we'll get rid of all but the final versions of the functions we wrote above. I've put my final cleaned up version in `./hosts/yahoo.py`

Like we wanted, this file has:

- `get_teams_in_league`
- `get_league_schedule`
- `get_league_rosters`

The other functions (e.g. `process_player3`) are NOT meant to be called outside this file. They're "helper" functions, `process_player3` is used inside `get_league_rosters`, no one else should ever have to use it.

For these helper functions, I kept the final versions (`process_player3` vs `process_player2` or 1), dropped the number (just `process_player`) then added an underscore to the front (e.g. `_process_player`).

Making helper functions start with `_` is a python convention that let's people reading your code know they shouldn't have to use this outside your module.

Feel free to explore these files or work through any other league host integrations you need. When you're ready to pick up with rest of the book head over to the next section, [Saving League Data to a Database](#).

Saving League Data to a Database

This section applies to all hosts — ESPN, Fleaflicker, Yahoo, Sleeper.

For each host, we ended up with three pieces of data: rosters, team data, and schedule. The team and schedule data isn't going to change throughout the season, so it's more polite to save it to a database and avoid continuously hitting these sites' APIs for the same data over and over.

Like we did in Learn to Code with Fantasy Football, we'll use sqlite, which comes with Anaconda.

So let's set that up now. This code is in `./projects/integration/db_working.py`. And we'll pick up right after the imports. The only thing to note. My example has:

```
import hosts.fleaflicker as site
```

because fleaflicker is the site I use, but you can swap it out with any of the other code we used, e.g.

```
import hosts.espn as site
```

This is why we purposely made each league integration end up with the same exact functions and made everything return the same data, regardless of how it came originally. It lets us use whichever site we want — and keep the rest of our analysis code the same - just by changing one line.

To start we just need our league and team ids:

```
In [2]:  
LEAGUE_ID = 316893  
TEAM_ID = 1605156
```

Getting the Data

We have two tables that won't be changing and which would be good to write to our fantasy database. These are the `teams` and `schedule` tables. Let's grab those:

```
In [3]:  
teams = site.get_teams_in_league(LEAGUE_ID, example=True)  
schedule = site.get_league_schedule(LEAGUE_ID, example=True)
```

Note the `example=True` argument. Including this will use the saved JSON and csv snapshots I've included with the book. I'd recommend leaving them on when following along here, though obviously you should leave them off (they default to `False`) when analyzing your own team.

Writing it to a Database

Once we have the `teams` and `schedule` DataFrames we can write them to SQL. To do that we first we have to connect to the database:

```
In [4]: conn = sqlite3.connect(DB_PATH)
```

And to write our `team` table to SQL:

```
In [5]: teams.to_sql('teams', conn, index=False, if_exists='replace')
```

Setting `if_exists` to replace will completely any existing `teams` table. If you're just in one league that'd be OK. But with multiple leagues, you'd want to add on your team info at the end, not replace it. Setting `if_exists='append'` does this, but it also will add on duplicate team data if it's already in there.

To avoid this, we can write some SQL that (1) deletes your existing teams data for the league you're working with, then (2) appends the new data to the end. This updates the data for league you're working with while leaving everything else alone.

We can do that like this:

```
In [6]: from textwrap import dedent
```

```
In [7]: conn.execute(dedent(f"""
    DELETE FROM teams
    WHERE league_id = {LEAGUE_ID};"""))
```

```
In [8]: teams.to_sql('teams', conn, index=False, if_exists='append')
```

Note: `dedent` is a function that let's you use f strings with multi-line strings. I'd would tell you more, but that's literally all I know about it. Something to do with (the opposite of?) indenting. I would recommend just memorizing it. f strings + multi line strings = `textwrap.dedent`.

This delete then write process could be useful, so let's put it in a function that takes the table name, connection and league_id and deletes the data we want from it.

```
def clear_league_from_table1(league_id, table, conn):
    conn.execute(dedent(f"""
        DELETE FROM {table}
        WHERE league_id = {league_id};"""))
```

Then let's write our schedule, both the game/week and team/game/week versions. First we'll run our `clear_league_from_table` function, then append the data:

```
In [9]: clear_league_from_table(LEAGUE_ID, 'schedule', conn)
...
OperationalError: no such table: schedule
```

Uh oh, getting an error. The problem is we haven't written any data to a `schedule` table yet. So our delete-then-add code runs into an error on the delete step (this was a real error I got while writing this chapter).

After some stack overflow I settled on this solution:

```
In [10]:
def clear_league_from_table2(league_id, table, conn):
    # get list of tables in db
    tables_in_db = [x[0] for x in list(conn.execute(
        "SELECT name FROM sqlite_master WHERE type='table';"))]

    # check if table is in list -- if it's NOT then we don't have to
    # delete any data
    if table in tables_in_db:
        conn.execute(dedent(f"""
            DELETE FROM {table}
            WHERE league_id = {league_id};"""))
```

This works:

```
In [11]: clear_league_from_table2(LEAGUE_ID, 'schedule', conn)

In [12]: schedule.to_sql('schedule', conn, index=False,
                        if_exists='append')
```

I think even this delete → write step could be simplified. How about:

```
def overwrite_league(df, name, conn, league_id):
    clear_league_from_table2(league_id, name, conn)
    df.to_sql(name, conn, index=False, if_exists='append')
```

Finally, let's add a function to get data *out* for a specific league.

```
def read_league(name, league_id, conn):
    return pd.read_sql(dedent(
        f"""
        SELECT *
        FROM {name}
        WHERE league_id = {league_id}
        """), conn)
```

Trying it:

```
In [13]: read_league('teams', LEAGUE_ID, conn)
Out[13]:
   team_id  owner_id      owner_name  division_id  league_id
0  1605154  1319436        ccarns       936221    316893
1  1605149  1320047  Plz-not-last       936221    316893
2  1605156  138510        nbraun       936221    316893
3  1605155  103206        BRUZDA       936221    316893
4  1605147  1319336      carnsc815       936222    316893
5  1605151  1319571      Edmundo       936222    316893
6  1605153  1319578      LisaJean       936222    316893
7  1664196  1471474      MegRyan0113       936222    316893
8  1603352  1316270      UnkleJim       936223    316893
9  1605157  1324792  JBKBDomination       936223    316893
10 1605148  1319991      Lzrlightshow       936223    316893
11 1605152  1328114      WesHeroux       936223    316893
```

Other League Data

So that's teams and schedule info. What about rosters? Should we save them too?

I don't think so. We know they'll always be changing — fantasy teams pick up and start or sit different guys all the time. So rather than worrying about it let's just scrape it every time.

The only thing left is it'd be good to store some data on our league too. Just some basics: `league_id`, our `team_id`, host, scoring settings, etc.

We don't even need to scrape this, we can just make a quick DataFrame by hand (i.e. by writing out a list of dicts), then write it to the db the same way.

```
In [13]:
league = DataFrame([{'league_id': LEAGUE_ID, 'team_id': TEAM_ID, 'host':
                     'fleaflicker', 'name': LEAGUE_NAME, 'qb_scoring':
                     'pass4', 'skill_scoring': 'ppr0',
                     'dst_scoring': 'mfl'}])

In [14]: overwrite_league(league, 'league', conn, LEAGUE_ID)
-- 

In [15]: overwrite_league(league, 'league', conn, LEAGUE_ID)
```

Perfect. This should be everything we need right now.

Wrap Up - League Configuration

In polishing this section up, I moved the `overwrite_league` and `read_league` functions to `./hosts/db.py`

I also turned some of this into template to connect to a new league for the first time. It gets the one time stuff (team list, schedule, league and scoring info) and writes it to our database.

This template is in `./hosts/league_setup.py`.

It's straightforward. We're just using the functions we wrote above to scrape team list, schedule and write everything to a database.

Auto WDIS - Integrating WDIS with your League

Now that we've written code to (A) get data from our league and (B) calculate who do I start win probabilities, let's work through how to tie these together.

Note: I'll use the final, cleaned up versions of both the wdis and integration code. So that we're on the same page, and it's easier to follow along, I've included saved versions of everything we'll be working with in [./projects/integration/raw/wdis/](#).

If you want to look at your own team and league you can, there just has to be data there (i.e. if you're working through this pre-season 2022 and haven't drafted yet, you're not going to be able to get your team roster).

You don't have to do it with this walk through, but before you run this with your own league, make sure you run [./hosts/league_setup.py](#) with your league info. This file gets things that don't change (the list of teams, schedule, etc) and writes them to a database.

The file we're working through now is [./projects/integration/auto_wdis_working.py](#).

Open it up and we'll pick up right after the imports. The only thing to note. My example has:

```
import hosts.fleaflicker as site
```

because fleaflicker is the site I use, but — when working with your own team — you can swap it out with any of the other code we used, e.g.

```
import hosts.espn as site
```

This is why we purposely made each league integration end up with the same exact functions and made everything return the same data, regardless of how it came originally. It lets us use whichever site we want — and keep the rest of our analysis code the same - just by changing one line.

Then we'll set our parameters. Really the only thing we need is league id and week:

```
In [1]:  
LEAGUE_ID = 316893  
WEEK = 2
```

Normally, we'd get everything else out of our own league database. That's what the [./hosts/league_setup.py](#) file does. To get it out we'd just need to connect to it:

```
In [3]: conn = sqlite3.connect(DB_PATH)
```

However, so that we're seeing the same thing, I've saved an example database with the data we're looking at in this chapter.

```
In [4]:  
conn = sqlite3.connect('./projects/integration/raw/wdis/fantasy.sqlite')
```

No matter which database we use, the first step is getting all the relevant data out of it using the handy `read_league` function we wrote.

```
In [5]:  
teams = db.read_league('teams', LEAGUE_ID, conn)  
schedule = db.read_league('schedule', LEAGUE_ID, conn)  
league = db.read_league('league', LEAGUE_ID, conn)  
host = league.iloc[0]['host']
```

Our `league` table has a few other parameters we'll need:

```
In [6]: league  
Out[6]:  
league_id  team_id  ... qb_scoring skill_scoring dst_scoring  
0         316893   1605156  ...           pass4             ppr0            mfl
```

Let's grab these quick:

```
In [7]:  
TEAM_ID = league.iloc[0]['team_id']  
SCORING = {}  
SCORING['qb'] = league.iloc[0]['qb_scoring']  
SCORING['skill'] = league.iloc[0]['skill_scoring']  
SCORING['dst'] = league.iloc[0]['dst_scoring']  
  
In [8]: TEAM_ID  
Out[8]: 1605156  
  
In [9]: SCORING  
Out[9]: {'qb': 'pass4', 'skill': 'ppr0', 'dst_scoring': 'mfl'}
```

Perfect.

Now let's get everyone's rosters. Again, these are always changing, and when you're doing this with your own league you'll want to grab a current snapshot with the `site.get_league_rosters` function:

```
In [10]:  
# need players from FM API  
token = generate_token(LICENSE_KEY)['token']  
player_lookup = master_player_lookup(token).query("fleaflicker_id.notnull()  
)  
  
rosters = site.get_league_rosters(player_lookup, LEAGUE_ID, WEEK)
```

But again, *here*, we'll use the data I've saved so you can follow along:

```
In [11]: player_lookup = pd.read_csv(  
        './projects/integration/raw/wdis/player_lookup.csv')  
  
In [12]: rosters = pd.read_csv(  
        './projects/integration/raw/wdis/rosters.csv')
```

Remember how the WDIS code we wrote works. We need:

- our starters
- opponent's starters
- WDIS options
- raw simulations

Let's start with *our* roster:

```
In [13]: roster = rosters.query(f"team_id == {TEAM_ID}")  
  
In [14]: roster.head()  
Out[14]:  
       name    ... team_id      fantasymath_id  
16   Jalen Hurts  ... 1605156  jalen-hurts  
17 Clyde Edwards-Helaire  ... 1605156 clyde-edwards-helaire  
18   Najee Harris  ... 1605156  najee-harris  
19     Myles Gaskin  ... 1605156  myles-gaskin  
20 Antonio Brown  ... 1605156  antonio-brown
```

Ok. So we need a list of our current starters:

```
In [14]:  
current_starters = list(roster.loc[roster['start'] &  
                                roster['fantasymath_id'].notnull(),  
                                'fantasymath_id'])  
  
--  
  
In [15]: current_starters  
Out[15]:  
['jalen-hurts',  
 'clyde-edwards-helaire',  
 'najee-harris',  
 'myles-gaskin',  
 'antonio-brown',  
 'davante-adams',  
 'george-kittle',  
 'greg-zuerlein',  
 'no-dst']
```

And our opponents starters. This is a two step process: we need to find our opponent id from the schedule, then our opponent's starters from rosters.

Currently our schedule is in wide format, where every row is a game. Let's write a quick function to put it into wide format:

```
In [16]:  
def schedule_long(sched):  
    sched1 = sched.rename(columns={'team1_id': 'team_id', 'team2_id':  
                                  'opp_id'})  
    sched2 = sched.rename(columns={'team2_id': 'team_id', 'team1_id':  
                                  'opp_id'})  
    return pd.concat([sched1, sched2], ignore_index=True)  
  
In [17]:  
schedule_team = schedule_long(schedule)
```

Then we can use it to find our opponent's `team_id` this week.

```
In [18]:  
# first: use schedule to find our opponent this week  
opponent_id = schedule_team.loc[  
    (schedule_team['team'] == TEAM_ID) & (schedule_team['week'] == WEEK),  
    'opp'].values[0]
```

And their starters and how many points they've scored. Note, I took this snapshot of data Friday morning, after NYG-WAS played Thursday night. So Sterling Shepard, who my opponent had, already played, and scored 8.5 points. We'll incorporate Sterling's actual score into our projection in a bit.

```
In [19]:  
opponent_starters = rosters.loc[  
    (rosters['team_id'] == opponent_id) & rosters['start'] &  
    rosters['fantasymath_id'].notnull(), ['fantasymath_id', 'actual']]
```

How it looks:

```
In [20]: opponent_starters  
Out[20]:  
    fantasymath_id  actual  
0      kyler-murray    NaN  
1      jamaal-williams    NaN  
2      jonathan-taylor    NaN  
3      mike-williams-wr    NaN  
4      julio-jones    NaN  
5  sterling-shepard     8.5  
6      tj-hockenson    NaN  
7      justin-tucker    NaN  
8        ne-dst    NaN
```

The only other thing we need are the raw simulations, which we get from the fantasymath API. We need sims for all of our players, plus our opponents starters.

```
In [21]:  
players_to_sim = pd.concat([  
    roster[['fantasymath_id', 'actual']],  
    opponent_starters])
```

In order to avoid errors, I recommend ensuring you're querying players with simulations this week (vs injured guys etc). To get a list of available player ids for any given season and week we can use the `get_players` function:

```
In [22]: available_players = get_players(token, season=2021,  
                                         week=WEEK, **SCORING)  
  
In [23]: available_players.head()  
Out[23]:  
   fantasymath_id  position  fleaflicker_id  
0      matt-prater        k          4221.0  
1     kyler-murray       qb          14664.0  
2   chase-edmonds       rb          13870.0  
3    james-conner       rb          12951.0  
4   maxx-williams      te          11191.0
```

Now we can get the sims:

```
In [24]:  
sims = get_sims(token, set(players_to_sim['fantasymath_id']) &  
                 set(available_players['fantasymath_id']), season=2021,  
                 week=WEEK, nsims=1000, **SCORING)  
--  
  
In [25]: sims.head()  
Out[25]:  
   kyler-murray  jalen-hurts  ...  justin-tucker  no-dst  
0      15.361354  15.628621  ...      17.941534  9.180006  
1      12.318237  14.276543  ...      1.004135  11.840859  
2      18.559885  15.616962  ...      0.362156  8.255119  
3      19.214451  10.953816  ...      1.737689  5.054510  
4      12.645511  30.179172  ...      10.539921  10.287236
```

These are correlated, simulated projections built on top of Fantasy Pros Expert Consensus Rankings. They're really good. But for this analysis — which remember we're doing Friday morning, Week 2, 2021, after WAS-NYG played Thursday night — they're not as good as having actual scores.

So let's use our `actual` points field and overwrite the simulations for the NYG-WAS guys who've already played.

The easiest way to do that is to: (1) find everyone who's already played, and (2) loop through each of them, overwriting their simulated values with actual scores. Like this:

```
In [26]: players_w_pts = players_to_sim.query("actual.notnull()")  
  
In [27]:  
for player, pts in zip(players_w_pts['fantasymath_id'], players_w_pts['actual']):  
    sims[player] = pts
```

Sterling Shepard is the only guy in our matchup who's played, but now his "sims" look like this (vs Jalen Hurts, who hasn't played yet):

```
In [28]: sims[['sterling-shepard', 'jalen-hurts']]  
Out[28]:  
      sterling-shepard  jalen-hurts  
0              8.5     15.316021  
1              8.5     20.805039  
2              8.5     24.841830  
3              8.5     25.095962  
4              8.5     15.114346  
..             ...       ...  
995             8.5     5.953876  
996             8.5     19.154578  
997             8.5     17.908486  
998             8.5     9.893287  
999             8.5     25.133482
```

Great. Now, finally we have everything we need to calculate WDIS.

Remember, besides the sims and starting lineups our WDIS function needs the wdis bench options (current starter + bench candidates).

At this point we're just hard coding them, like this:

```
In [29]:  
# wdis options + current starter  
wdis_options = ['antonio-brown', 'jaylen-waddle', 'christian-kirk']
```

The results:

```
In [29]: wdis.calculate(sims, current_starters, opponent_starters,  
                      wdis_options)  
Out[29]:  
      mean        std        5%    ...      wp    wrong   regret  
antonio-brown  94.523771  19.798203  62.806859  ...  0.535  0.589  0.051  
jaylen-waddle  93.128948  20.064397  60.504378  ...  0.507  0.680  0.079  
christian-kirk  92.557072  19.507654  62.079994  ...  0.502  0.731  0.084
```

So the model says to start Antonio Brown, who gives me a 0.535 probability of winning.

We can swap it out for a different position too:

```
In [30]:  
wdis.calculate(sims, current_starters, opponent_starters['fantasymath_id'  
],  
                ['jalen-hurts', 'mac-jones'])  
--  
Out[30]:
```

	mean	std	5%	...	wp	wrong	regret
jalen-hurts	94.523771	19.798203	62.806859	...	0.535	0.36	0.035
mac-jones	90.594189	19.386102	58.005401	...	0.461	0.64	0.109

This is cool, and it's nice we haven't had to manually enter in all our starters + our opponents players, but it's still kind of annoying to have to put in the WDIS options.

Let's write some code that takes a position + our roster (starters + bench) and automatically analyzes all the eligible bench players.

It's always easiest to start with a specific example, so let's do WR1

```
In [31]: pos = 'WR1'
```

We need a list of fantasymath_ids for: our current starter at WR1 (Antonio Brown), and all the eligible bench players. Identifying the starter will be easy (his `team_position` is `WR1`), but what about the bench players?

I think the easiest way is to check if the `player_position` column is "in" our position, e.g. '`WR1`'.

In python you can do that with `in`:

```
In [32]: 'WR' in 'WR1'  
Out[32]: True  
  
In [33]: 'RB' in 'WR1'  
Out[33]: False
```

It'll work with flex positions too:

```
In [33]: 'WR' in 'RB/WR/TE'  
Out[33]: True  
  
In [34]: 'QB' in 'RB/WR/TE'  
Out[34]: False
```

Note, this only works because our flex position is '`RB/WR/TE`'. If it was '`FLEX`' or '`W/R/T`' (which is what it originally was in Yahoo) it'd be a different story.

We can use `in` on every `player_position` with `apply` like this:

```
In [35]: pos_in_wr1 = roster['player_position'].astype(str).apply(  
    lambda x: x in pos)  
  
In [36]: pos_in_wr1  
Out[36]:  
16    False  
17    False  
18    False  
19    False  
20    True  
21    True  
22    False  
23    False  
24    False  
25    False  
26    False  
27    False  
28    False  
29    True  
30    True  
31    False
```

This returns a list of bools, let's filter our data on these to see who we're dealing with:

```
In [37]: roster.loc[pos_in_wr1]  
Out[37]:  
          name player_position ... start team_id fantasymath_id  
20 Antonio Brown      WR ...  True  1605156 antonio-brown  
21 Davante Adams      WR ...  True  1605156 davante-adams  
29 Christian Kirk     WR ... False  1605156 christian-kirk  
30 Jaylen Waddle     WR ... False  1605156 jaylen-waddle
```

This is *almost* what we want, but not quite. We want starter + bench options, but *not* our other starters. In other words, we want to look at our WR1 + all the bench options. We *don't* want to include our WR2 (Davante Adams), because he's already starting. We can clarify with this:

```
In [38]:  
bench_wr1_elig = ((roster['player_position']  
    .astype(str)  
    .apply(lambda x: x in pos) & ~roster['start']) |  
    (roster['team_position'] == pos))
```

That's another column of bools (the right ones this time), passing it to `loc` and keeping just the `fantasymath_id` gets us what we want:

```
In [39]: wdis_ids = list(roster.loc[bench_wr1_elig, 'fantasymath_id'])

In [40]: wdis_ids
Out[40]: ['antonio-brown', 'christian-kirk', 'jaylen-waddle']
```

As always, let's put this in a function:

```
def wdis_options_by_pos(roster, team_pos):
    is_wdis_elig = ((roster['player_position']
                     .astype(str)
                     .apply(lambda x: x in team_pos) & ~roster['start']) |
                     (roster['team_position'] == team_pos))

    return list(roster.loc[is_wdis_elig, 'fantasymath_id'])
```

And try it out:

```
In [41]: wdis_players_flex = wdis_options_by_pos(roster, 'RB/WR/TE')

In [42]: wdis_players_flex
Out[42]:
['myles-gaskin',
 'elijah-mitchell-rb',
 'tyson-williams',
 'javonte-williams',
 'christian-kirk',
 'jaylen-waddle']
```

This is easy to plug into our `wdis.calculate` function:

```
In [43]:
df_flex = wdis.calculate(sims, current_starters,
                         opponent_starters['fantasymath_id'],
                         wdis_players_flex)
--
```



```
In [44]: df_flex
Out[44]:
      mean      std    ...      wp    wrong   regret
myles-gaskin  94.523771  19.798203    ...  0.535  0.766  0.095
elijah-mitchell-rb  93.596292  19.879092    ...  0.523  0.813  0.107
tyson-williams  93.648150  19.886028    ...  0.519  0.825  0.111
javonte-williams  93.957985  19.885288    ...  0.518  0.810  0.112
christian-kirk   91.749487  19.321485    ...  0.495  0.902  0.135
jaylen-waddle    92.321363  19.824063    ...  0.492  0.884  0.138
```

Let's put this calculate part in a function too:

```
def wdis_by_pos1(pos, sims, roster, opp_starters):
    wdis_options = wdis_options_by_pos(roster, pos)

    starters = list(roster.loc[
        roster['start'] &
        roster['fantasymath_id'].notnull(), 'fantasymath_id'])

    return wdis.calculate(sims, starters, opp_starters,
                          set(wdis_options) & set(sims.columns))
```

Calling it:

```
In [45]: wdis_by_pos1('QB', sims, roster,
                      opponent_starters['fantasymath_id'])
Out[45]:
      mean      std      5%  ...      wp      wrong      regret
jalen-hurts  94.523771  19.798203  62.806859  ...  0.535  0.36  0.035
mac-jones     90.594189  19.386102  58.005401  ...  0.461  0.64  0.109

In [46]: wdis_by_pos1('RB/WR/TE', sims, roster,
                      opponent_starters['fantasymath_id'])
Out[46]:
      mean      std      5%  ...      wp      wrong      regret
myles-gaskin  94.523771  19.798203  62.806859  ...  0.535  0.766
  0.095
elijah-mitchell-rb  93.596292  19.879092  62.027842  ...  0.523  0.813
  0.107
tyson-williams  93.648150  19.886028  61.740988  ...  0.519  0.825
  0.111
javonte-williams  93.957985  19.885288  61.751722  ...  0.518  0.810
  0.112
christian-kirk    91.749487  19.321485  61.286738  ...  0.495  0.902
  0.135
jaylen-waddle     92.321363  19.824063  61.162144  ...  0.492  0.884
  0.138
```

We could also get a list of our positions:

```
In [47]: positions = list(
    roster.loc[roster['start'] & roster['fantasymath_id'].notnull(),
               'team_position'])

In [48]: positions
Out[48]: ['QB', 'RB1', 'RB2', 'RB/WR/TE', 'WR1', 'WR2', 'TE', 'K', 'D/ST']
```

Quick aside: this is one line of code that gets us a list of of positions. It's useful to be able to get these, and this line works. But it's fairly gnarly. Let's put it in a function with a reasonable name so we don't think about this line every time we see it.

```
In [48]:  
def positions_from_roster(roster):  
    return list(roster.loc[roster['start'] &  
                           roster['fantasymath_id'].notnull(),  
                           'team_position'])  
  
--  
  
In [49]: positions_from_roster(roster)  
Out[49]: ['QB', 'RB1', 'RB2', 'RB/WR/TE', 'WR1', 'WR2', 'TE', 'K', 'D/ST']
```

That's better.

Ok, now that we have our list of positions, we can run our `wdis_by_pos1` function over them.

```
In [50]:  
for pos in positions:  
    print(wdis_by_pos1(pos, sims, roster, opponent_starters))
```

Not printing out the results because it's pretty long, but this works. But it might be easier to stick these on top of each other in one giant table.

If we do that it'd be helpful to add position our return DataFrame. Let's do it as an index with some `reset_index` and `set_index` functions.

```
def wdis_by_pos2(pos, sims, roster, opp_starters):  
    wdis_options = wdis_options_by_pos(roster, pos)  
  
    starters = list(roster.loc[  
        roster['start'] &  
        roster['fantasymath_id'].notnull(), 'fantasymath_id'])  
  
    df = wdis.calculate(sims, starters, opp_starters,  
                        set(wdis_options) & set(sims.columns))  
  
    rec_start_id = df['wp'].idxmax()  
  
    df['pos'] = pos  
    df.index.name = 'player'  
    df.reset_index(inplace=True)  
    df.set_index(['pos', 'player'], inplace=True)  
  
    return df
```

And calling it:

```
In [51]: wdis_by_pos2('QB', sims, roster, opponent_starters['fantasymath_id'])
Out[51]:
          mean      std    ...      wp    wrong   regret
pos player
QB  jalen-hurts  94.523771  19.798203    ...  0.535    0.36    0.035
     mac-jones    90.594189  19.386102    ...  0.461    0.64    0.109
```

We can see our DataFrame now has a multi index, where each row is identified by position + player. The reason we did that is so we can stick these sub-wdis position DataFrames together, like this:

```
In [52]:
df_start = pd.concat(
    [wdis_by_pos2(pos, sims, roster, opponent_starters) for pos in
     positions])

In [53]: df_start.head(10)
Out[53]:
          mean      std    ...      wp    wrong   regret
pos player
QB  jalen-hurts  94.523771  19.798203    ...  0.360    0.035
     mac-jones    90.594189  19.386102    ...  0.640    0.109
RB1 clyde-edwards-helaire  94.523771  19.798203    ...  0.634    0.078
     elijah-mitchell-rb  91.969530  19.976872    ...  0.767    0.120
     tyson-williams    92.021388  19.663684    ...  0.818    0.128
     javonte-williams  92.331223  19.792092    ...  0.781    0.135
RB2 najee-harris    94.523771  19.798203    ...  0.492    0.055
     javonte-williams  89.234584  19.570730    ...  0.825    0.128
     tyson-williams    88.924749  19.458267    ...  0.847    0.146
     elijah-mitchell-rb  88.872892  19.377049    ...  0.836    0.160
```

One interesting things about multi indexed DataFrames is the `xs` function. It returns a subset of the DataFrame. For example:

```
In [54]: df_start.xs('WR1')
Out[54]:
          mean      std      5%    ...      wp    wrong   regret
player
antonio-brown  94.523771  19.798203  62.806859    ...  0.535    0.589    0.051
jaylen-waddle  93.128948  20.064397  60.504378    ...  0.507    0.680    0.079
christian-kirk 92.557072  19.507654  62.079994    ...  0.502    0.731    0.084
```

Note how this DataFrame no longer is multi indexed. Everything here was in WR1, so it's just the player.

This is a DataFrame, and we can use regular python functions on it. One cool function is `idxmax` for “index max”. It returns the *index* of the max. So if we want to see which WR1 maximizes our win probability:

```
In [55]: df_start.xs('WR1')['wp'].idxmax()  
Out[55]: 'antonio-brown'
```

We can run it on every position to see our whole best starting lineup:

```
In [56]: rec_starters = [df_start.xs(pos)['wp'].idxmax() for pos in  
positions]  
  
In [57]: rec_starters  
Out[57]:  
['jalen-hurts',  
'clyde-edwards-helaire',  
'najee-harris',  
'myles-gaskin',  
'antonio-brown',  
'davante-adams',  
'george-kittle',  
'greg-zuerlein',  
'no-dst']
```

These correspond to our positions:

```
In [58]: positions  
Out[58]: ['QB', 'RB1', 'RB2', 'RB/WR/TE', 'WR1', 'WR2', 'TE', 'K', 'D/ST']
```

One interesting computer science concept is *zipping*, where we match up two similar lists. So we could do this:

```
In [59]:  
for pos, starter in zip(positions, rec_starters):  
    print(f"at {pos}, start {starter}")  
  
at QB, start jalen-hurts  
at RB1, start clyde-edwards-helaire  
at RB2, start najee-harris  
at RB/WR/TE, start myles-gaskin  
at WR1, start antonio-brown  
at WR2, start davante-adams  
at TE, start george-kittle  
at K, start greg-zuerlein  
at D/ST, start no-dst
```

Writing to a File

Once it's in functions, this code is all pretty clean, and we could stop there. But sometimes it's also nice to output things outside of Python. For example, maybe we want to generate some text output that we could email or paste on our leagues message board or whatever.

The easiest way to do that is with the `print` function.

`print` takes an optional `file` argument where you can pass a Python file object.

Note this object isn't quite the same as a file path. First you have to open it, then you can write to it.

Say we want to print some stuff to a file named `league_info.txt`. We would do:

```
In [1]: my_file = open('league_info.txt', 'w')
```

And print to it like:

```
In [2]: print(f'Your league is {LEAGUE_ID}!', file=my_file)
```

Then we can open up `league_info.txt` and see:

```
Your league is 316893!
```

The usual way of writing to files is to put them inside a `with` statement, like this:

```
with open('league_info.txt', 'w') as my_file:  
    print(f'Your league is {LEAGUE_ID}!', file=my_file)
```

This just has the benefit that — if something goes wrong with your `print` statement and you get an error — the `with` statement will gracefully close your file.

In my experience, `with` statements hardly ever come up — working with raw files one of the few exceptions, so I'd recommend not thinking about it too much and just getting in the habit of doing it for that.

Writing WDIS Output to a File

Let's print our WDIS advice and recommended starters to a file.

We need to think for a second about where we might want to print. We could put our league and week info in a file, maybe something like:

```
In [3]: f'fleaflicker_{LEAGUE_ID}_2021-{str(WEEK).zfill(2)}-wdis.txt'  
Out[3]: 'fleaflicker_316893_2021-02-wdis.txt'
```

This would be fine. *But*, we are going to have other analysis output (e.g. plots, a league analysis) for this specific league and week too. So it might be easier to make this a *directory*, then put the WDIS text in there. So something like:

```
In [4]: f'./output/fleaflicker_{LEAGUE_ID}_2021-{str(WEEK).zfill(2)}/wdis.txt'
Out[4]: './output/fleaflicker_316893_2021-02/wdis.txt'
```

Let's do that. But this directory might not exist yet, let's make it if it doesn't. In Python you do that with [Path](#).

```
In [5]:
from pathlib import Path
from os import path

In [6]:
league_wk_output_dir = path.join(
    OUTPUT_PATH, f'fleaflicker_{LEAGUE_ID}_2021-{str(WEEK).zfill(2)}')

In [7]:
Path(league_wk_output_dir).mkdir(exist_ok=True, parents=True)
```

Now let's print our results it out to a file. Note I'm printing:

```
print("", file=f)
```

Anytime I want a blank line. We'll print out suggested starters, a few columns from our summary DataFrame as well as a note about whether we're currently starting the optimal or lineup.

```
In [8]:  
with open(path.join(league_wk_output_dir, 'wdis.txt'), 'w') as f:  
    print(f"WDIS Analysis, Fleaflicker League {LEAGUE_ID}, Week {WEEK}",  
          file=f)  
    print("", file=f)  
    print(f"Run at {dt.datetime.now()}", file=f)  
    print("", file=f)  
    print("Recommended Starters:", file=f)  
    for starter, pos in zip(starters, positions):  
        print(f"{pos}: {starter}", file=f)  
  
    print("", file=f)  
    print("Detailed Projections and Win Probability:", file=f)  
    print(df_start[['mean', 'wp', 'wrong', 'regret']], file=f)  
    print("", file=f)  
  
    if set(team_starters) == set(starters):  
        print("Current starters maximize probability of winning.", file=f)  
    else:  
        print("Not maximizing probability of winning.", file=f)  
        print("", file=f)  
        print("Start:", file=f)  
        print(set(starters) - set(team_starters), file=f)  
        print("", file=f)  
        print("Instead of:", file=f)  
        print(set(team_starters) - set(starters), file=f)
```

And the output it produces in `'./output/fleaflicker_316893_2021-01/wdis.txt'`:

WDIS Analysis, Fleaflicker League 316893, Week 2

Run at 2021-09-17 10:04:27.761777

Recommended Starters:

QB: jalen-hurts
 RB1: clyde-edwards-helaire
 RB2: najee-harris
 RB/WR/TE: myles-gaskin
 WR1: antonio-brown
 WR2: davante-adams
 TE: george-kittle
 K: greg-zuerlein
 D/ST: no-dst

Detailed Projections and Win Probability:

pos	player	mean	wp	wrong	regret
QB	jalen-hurts	94.377542	0.554	0.358	0.032
	mac-jones	90.375185	0.491	0.642	0.095
RB1	clyde-edwards-helaire	94.377542	0.554	0.630	0.068
	tyson-williams	92.330186	0.523	0.764	0.099
	javonte-williams	91.572212	0.511	0.813	0.111
	elijah-mitchell-rb	91.290203	0.499	0.793	0.123
RB2	najee-harris	94.377542	0.554	0.502	0.055
	tyson-williams	89.190119	0.485	0.802	0.124
	javonte-williams	88.432145	0.470	0.839	0.139
	elijah-mitchell-rb	88.150137	0.468	0.857	0.141
RB/WR/TE	myles-gaskin	94.377542	0.554	0.786	0.091
	tyson-williams	94.229789	0.551	0.793	0.094
	javonte-williams	93.471814	0.532	0.835	0.113
	elijah-mitchell-rb	93.189806	0.530	0.833	0.115
	christian-kirk	92.200422	0.522	0.876	0.123
	jaylen-waddle	92.375368	0.513	0.877	0.132
WR1	antonio-brown	94.377542	0.554	0.564	0.051
	jaylen-waddle	92.720904	0.523	0.712	0.082
	christian-kirk	92.545959	0.520	0.724	0.085
WR2	davante-adams	94.377542	0.554	0.398	0.030
	christian-kirk	88.518732	0.468	0.798	0.116
	jaylen-waddle	88.693678	0.465	0.804	0.119
TE	george-kittle	94.377542	0.554	0.000	0.000
K	greg-zuerlein	94.377542	0.554	0.000	0.000
D/ST	no-dst	94.377542	0.554	0.456	0.041
	ari-dst	93.703521	0.539	0.544	0.056

Current starters maximize probability of winning.

Wrap Up

To wrap up, we'll get rid of all but the final versions of the functions we wrote above. I've put my final cleaned up version in: `./wdis.py`

I also moved the `schedule_long` function to `utilities.py` to use later.

8. Project #3: League Analyzer

In this section, we'll build a tool to analyze our league, getting projections, over-unders and betting lines for each matchup, as well as team-specific stats like probability everyone scores the high or the low.

Open up `./projects/league/league_working.py` and we can get started.

Prerequisites

Note: the league integration project makes this *way* easier. That's why this project comes after it.

No worries if you haven't worked through it yet — we'll walk through it using some example data from my own league.

If you do want to follow along with your own league during this walk through, just set `USE_SAVED_DATA` (line 15) to `False` and update `LEAGUE` and `WEEK` (lines 22-23) with your league values. Note you have to add your league to `./hosts/league_setup.py` and run that first.

Input Data

To build this analysis we'll need four datasets. These are:

1. A list of **teams** in your league.
2. Your league **schedule** — i.e. who plays who each week.
3. Your league rosters — plus any actual points scored so far in the week (e.g. if we're running this analysis on a Friday, when some players have scores from the Thursday night game).
4. The Fantasy Math simulations.

We wrote the code to get 1-3 last chapter. Here, we'll take them as given and assume they're in the right format. The simulations (4) come with the kit.

The first part of `./projects/league/league_working.py` loads all this data — either the example I have saved (which is what I'd recommend using to start) or data from your own league and week.

To start, run that (through line 68) and let's pick up at line 74.

Here's what we have. Our DataFrame of teams:

	In [1]:	teams
	Out[1]:	
0	team_id	owner_id
1	1747268	2075108
2	1605152	1328114
3	1747266	2075011
4	1605155	103206
5	1605147	1319336
6	1605156	138510
7	1664196	1471474
8	1603352	1316270
9	1605157	1324792
10	1605148	1319991
11	1605151	1319571
		JBKBDomination
		Lzrlightshow
		Edmundo
		ccarns
		316893

And schedule:

	In [2]:	schedule.head(10)
	Out[2]:	
0	team1_id	team2_id
1	1605148	1605154
2	1603352	1605156
3	1605151	1605157
4	1605147	1664196
5	1605152	1747266
6	1605155	1747268
7	1605156	1605147
8	1605154	1605151
9	1747266	1605155
		53623235
		53623233
		53623236
		53623234
		53623231
		53623232
		53623240
		53623242
		53623238
		53623241
		2023
		2023
		2023
		2023
		2023
		2023
		1
		1
		1
		1
		1
		316893
		316893
		316893
		316893
		316893
		316893
		316893
		316893
		316893
		316893
		316893

We're going to pretend we're writing this league analysis code on Friday of Week 2, during the 2023 season. So "last night" the Eagles beat the Vikings on Thursday night football. That's why when we look at our rosters:

	In [6]:	rosters.head()
	Out[6]:	
0	name	...
1	Anthony Richardson	...
2	David Montgomery	...
3	AJ Dillon	...
4	Justin Jefferson	...
		team_position
		QB
		RB
		RB
		WR
		WR
		actual
		NaN
		NaN
		NaN
		13.9
		NaN
		True
		1747268
		1747268
		1747268
		1747268
		1747268
		6
		689
		520
		551
		377
		player_id

We can see we have an actual score for Justin Jefferson.

Finally we have the simulation data. Note, for someone like Jefferson, who already played, *all* his sims are what he got, i.e. 13.9:

So the first sims for Anthony Richardson, David Montgomery and JJ (ids 6, 689, 551 respectively):

```
In [20]: sims[[6, 689, 551]].head()
Out[20]:
      6          689      551
0  4.301246  9.360650  13.9
1 14.850512 10.853158  13.9
2 26.540597 21.061450  13.9
3  9.579856 12.966101  13.9
4 15.152442 21.622380  13.9
```

Now we should be set to do some analysis.

Analyzing Matchups

Let's start by thinking about what might be interesting to look at in any particular matchup.

We have the raw simulations, so there are a bunch of different things we could do.

How about:

- Who's favored?
- What's the probability they win?
- What's the line (how much do we project them to win by)?
- What's the total over under?

Let's pick an actual matchup to start. We have our full schedule, let's get the matchups for this week.

```
In [1]: schedule_this_week = schedule.query(f"week == {WEEK}")
In [2]: schedule_this_week
Out[2]:
   team1_id  team2_id  matchup_id  season  week  league_id
6     1605156    1605147  53623240    2023     2      316893
7     1605154    1605151  53623242    2023     2      316893
8     1747266    1605155  53623238    2023     2      316893
9     1605148    1605157  53623241    2023     2      316893
10    1603352   1664196  53623239    2023     2      316893
11    1605152   1747268  53623237    2023     2      316893
```

This is pretty impersonal, but we can add some owner/team name info later.

Let's start with the this matchup, 1605156 (team 1) vs 1605147 (team 2).

```
In [3]:  
team1 = 1605156  
team2 = 1605147
```

We can get a list of team 1's player ids with:

```
In [4]: rosters.query(f"start & (team_id == {team1})")['player_id']  
Out[4]:  
79    1091  
80    349  
81    904  
82   1121  
83    372  
84   1138  
85   1574  
86   1228  
87    5159
```

That's short and not that hard to type, but seeing the syntax is a little jarring (the brackets [] next to the () throws me off). So let's put it in a function.

```
def lineup_by_team(team_id):  
    return rosters.query(  
        f"start & (team_id == {team_id} & player_id.notnull())")['player_id']
```

Recall the two benefits of functions: DRY and allowing you to think clearer. This is a pretty clear example of the latter.

From there, getting the simulations of team 1's lineup is easy:

```
In [5]: sims[team1_roster].head()  
Out[5]:  
      1091      349      904     ...      1574      1228      5159  
0  15.257107  6.547626  22.071970  ...  4.843042  5.756176  4.976292  
1  24.679233  11.237680  16.238548  ...  5.186847  6.271533  14.421049  
2   7.397607  3.893144  10.603857  ... 11.132498  5.179903  6.249771  
3  25.821673  12.554041  14.428419  ...  3.911991  14.406736  6.953320  
4  41.633112  3.024962  13.784381  ...  8.936479  0.950506  13.464130
```

For now, we're interested in point totals, so let's `sum(axis=1)` it. Then do it for team 2 too.

```
In [6]: team1_pts = sims[team1_roster].sum(axis=1)  
In [7]: team2_roster = lineup_by_team(team2)  
In [8]: team2_pts = sims[team2_roster].sum(axis=1)
```

These points look like:

```
In [9]: pd.concat([team1_pts, team2_pts], axis=1).head(10)
Out[9]:
      0          1
0  88.913796  105.280698
1  101.462538  95.911795
2   74.157969  108.624296
3  116.929091  134.744941
4  119.253753  112.089568
5  118.788815  98.740174
6  130.281946  126.707327
7  140.199710  81.394420
8  107.975816  116.495831
9  118.390521  123.878607
```

Now answering our questions is straightforward. For example, what's team 1's probability of winning?

```
In [10]: team1_wp = (team1_pts > team2_pts).mean()
In [11]: team1_wp
Out[11]: 0.504
```

Team 1 has a 50.4% of winning.

What's the spread (how much does team 1 usually win by)?

```
In [12]: line = (team1_pts - team2_pts).median()
In [13]: line
Out[13]: 0.32514973574635775
```

Betting lines are usually rounded to the nearest 0.5, so let's do that to make it look more official (note: I Googled this to figure out how to do it):

```
In [14]: line = round(line*2)/2
In [15]: line
Out[15]: 0.5
```

What about the moneyline?

(Note: a moneyline is just another way to express probability; for the favorite, it's how much you'd have to bet to get \$100 if you won. For the underdog, it's how much you'd win if you bet \$100. If this is confusing, don't worry about it; I personally find normal probabilities more intuitive.)

This will do it:

```
def wp_to_ml(wp):
    if wp > 0.5:
        return int(round(-1*(100/((1 - wp)) - 100), 0))
    else:
        return int(round((100/((wp))) - 100), 0))

In [16]: wp_to_ml(team1_wp)
Out[16]: -102
```

The over-under is also easy:

```
In [17]: over_under = (team1_pts + team2_pts).median()

In [18]: over_under
Out[18]: 221.97816339980017
```

Great. So we've already written (in an iterative, easy-to-get-feedback sort of way) some basic code to get the information we wanted about a given matchup.

We're tunneling through a giant rock wall and now have a bit of light shining through. Let's expand it.

First let's move some of the work to functions to make it more reusable. If we put the stats we want (win probability, line etc) in a dictionary, it'll be easy to turn them into a DataFrame.

```
def summarize_matchup(sims_a, sims_b):
    """
    Given two teams of sims (A and B), summarize a matchup with win
    probability, over-under, betting line, etc
    """

    # start by getting team totals
    total_a = sims_a.sum(axis=1)
    total_b = sims_b.sum(axis=1)

    # get win prob
    winprob_a = (total_a > total_b).mean()
    winprob_b = 1 - winprob_a

    # get over-under
    over_under = (total_a + total_b).median()

    # spread
    spread = (total_a - total_b).median().round(2)
    spread = round(spread*2)/2

    # moneyline
    ml_a = wp_to_ml(winprob_a)

    return {'wp_a': round(winprob_a, 2), 'wp_b': round(winprob_b, 2),
            'over_under': round(over_under, 2), 'spread': spread, 'ml': ml_a}
```

A quick check to make sure it's working as expected on the matchup we've been looking at:

```
In [19]: summarize_matchup(sims[team1_roster], sims[team2_roster])
Out[19]: {'wp_a': 0.5, 'wp_b': 0.5, 'over_under': 221.98, 'spread': 0.5,
          'ml': -102}
```

Then it's a matter of applying this function to all of the matchups. There are multiple ways to do that. Probably the easiest way is zipping up each home and away column, then going through them with a loop (note how we're unpacking each matchup into `a` and `b`):

```
In [20]: matchup_list = []

In [21]:
for a, b in zip(schedule_this_week['team1_id'],
                 schedule_this_week['team2_id']):

    # gives us Series of starting lineups for each team in matchup
    lineup_a = lineup_by_team(a)
    lineup_b = lineup_by_team(b)

    # use lineups to grab right sims, feed into summarize_matchup function
    working_matchup_dict = summarize_matchup(
        sims[lineup_a], sims[lineup_b])

    # add some other info to working_matchup_dict
    working_matchup_dict['team_a'] = a
    working_matchup_dict['team_b'] = b

    # add working dict to list of matchups, then loop around to next
    # matchup
    matchup_list.append(working_matchup_dict)
```

The end result, `matchup_list`, is a list of dicts that all have the same fields. That's good because its really easy to turn that into a DataFrame.

```
In [21]: matchup_df = DataFrame(matchup_list)

In [22]: matchup_df
Out[22]:
   wp_a  wp_b  over_under  spread   ml  team_a  team_b
0  0.50  0.50     221.98    0.5 -102  1605156  1605147
1  0.40  0.60     185.82   -6.0  149  1605154  1605151
2  0.58  0.42     170.38    5.5 -138  1747266  1605155
3  0.90  0.10     198.01   27.5 -931  1605148  1605157
4  0.20  0.80     202.50   -19.0  390  1603352  1664196
5  0.62  0.38     184.20    8.0 -163  1605152  1747268
```

This is cool, but it'd be better with names instead of just team ids. We can connect `team_id` to something more personal (`owner_name`) with our `teams` data:

```
In [23]: teams
Out[23]:
   team_id  owner_id    owner_name  league_id
0  1605154  1319436      ccarns     316893
1  1605149  1320047  Plz-not-last     316893
2  1605156  138510       nbraun     316893
3  1605155  103206      BRUZDA     316893
4  1605147  1319336    carnsc815     316893
5  1605151  1319571      Edmundo     316893
6  1605153  1319578     LisaJean     316893
7  1664196  1471474    MegRyan0113     316893
8  1603352  1316270     UnkleJim     316893
9  1605157  1324792  JBKBDomination     316893
10 1605148  1319991    Lzrlightshow     316893
11 1605152  1328114     WesHeroux     316893
```

So let's add them. Note we have team ids in two columns: `team_a`, and `team_b`. Using the `teams` DataFrame to merge in `owner_name` would be straightforward, if tedious. It'd be something like this (note, step through and run these steps one at a time and look at the results in the REPL if you're having trouble following):

```
In [24]:
# for a
matchup_df = pd.merge(matchup_df, teams[['team_id', 'owner_name']],
                      left_on='team_a', right_on='team_id')
matchup_df['team_a'] = matchup_df['owner_name']
matchup_df.drop(['team_id', 'owner_name'], axis=1, inplace=True)
```

Then for team b:

```
In [25]:
# for b
matchup_df = pd.merge(matchup_df, teams[['team_id', 'owner_name']],
                      left_on='team_b', right_on='team_id')
matchup_df['team_b'] = matchup_df['owner_name']
matchup_df.drop(['team_id', 'owner_name'], axis=1, inplace=True)
```

This works:

```
In [26]: matchup_df
Out[26]:
   wp_a  wp_b  over_under  spread  ml      team_a      team_b
0  0.50  0.50    221.98    0.5 -102      nbraun  carnsc815
1  0.40  0.60    185.82   -6.0  149      ccarns   Edmundo
2  0.58  0.42    170.38    5.5 -138      Meat11    Brusda
3  0.90  0.10    198.01   27.5 -931  Lzrlightshow  JBKBDomination
4  0.20  0.80    202.50   -19.0  390      UnkleJim  MegRyan0113
5  0.62  0.38    184.20    8.0 -163     WesHeroux   CalBrusda
```

and doesn't use anything we didn't cover in LTCWFF.

If I was coding this myself I'd probably take advantage of Pandas `replace` function, which takes a dictionary of current, replacement values.

We can create our `team_to_owner` dict with comprehensions + zips, then use `.replace` to swap these values in one line.

```
In [27]:  
team_to_owner = {team: owner for team, owner in zip(teams['team_id'],  
                                                    teams['owner_name'])}  
  
In [28]: team_to_owner  
Out[28]:  
{1747268: 'CalBrusda',  
 1605152: 'WesHeroux',  
 1747266: 'Meat11',  
 1605155: 'Brusda',  
 1605147: 'carnsc815',  
 1605156: 'nbraun',  
 1664196: 'MegRyan0113',  
 1603352: 'UnkleJim',  
 1605157: 'JBKBDomination',  
 1605148: 'Lzrlightshow',  
 1605151: 'Edmundo',  
 1605154: 'ccarns'}  
  
In [29]: matchup_df[['team_a', 'team_b']] = (  
    matchup_df[['team_a', 'team_b']].replace(team_to_owner))  
  
In [30]: matchup_df  
Out[30]:  
      wp_a  wp_b  over_under  spread   ml       team_a        team_b  
0  0.50  0.50     221.98    0.5 -102      nbraun      carnsc815  
1  0.40  0.60     185.82   -6.0  149      ccarns      Edmundo  
2  0.58  0.42     170.38    5.5 -138      Meat11       Brusda  
3  0.90  0.10     198.01   27.5 -931  Lzrlightshow  JBKBDomination  
4  0.20  0.80     202.50   -19.0  390      UnkleJim    MegRyan0113  
5  0.62  0.38     184.20    8.0 -163      WesHeroux    CalBrusda
```

Nice.

Already, I think this is something my league might be interested in.

We could also write a few functions that take in this `matchup_df`, and pick out certain matchups. Maybe a “lock of the week” for team with the highest win probability or something.

Let's try it. Note this DataFrame is by matchup, so each team's win probability could be in one of two columns. Let's rearrange so each row a team, with just one column for win probability.

```
In [31]: wp_a = matchup_df[['team_a', 'wp_a', 'team_b']]
In [32]: wp_a.columns = ['team', 'wp', 'opp']

In [33]: wp_b = matchup_df[['team_b', 'wp_b', 'team_a']]
In [34]: wp_b.columns = ['team', 'wp', 'opp']

In [35]: stacked = pd.concat([wp_a, wp_b], ignore_index=True)

In [36]: stacked
Out[36]:
      team      wp        opp
0      nbraun  0.50    carnsc815
1      ccarns   0.40     Edmundo
2      Meat11   0.58     Brusda
3  Lzrlightshow  0.90  JBKBDomination
4      UnkleJim  0.20    MegRyan0113
5      WesHeroux  0.62    CalBrusda
6      carnsc815  0.50      nbraun
7      Edmundo   0.60      ccarns
8      Brusda    0.42     Meat11
9  JBKBDomination  0.10    Lzrlightshow
10     MegRyan0113  0.80    UnkleJim
11     CalBrusda  0.38    WesHeroux
```

Now we just need to pick out the row with the highest win probability.

```
In [37]: lock = stacked.sort_values('wp', ascending=False).iloc[0]

In [38]: lock.to_dict()
Out[38]: {'team': 'Lzrlightshow', 'wp': 0.9, 'opp': 'JBKBDomination'}
```

Looks good, let's put it in a function.

```
def lock_of_week(df):
    # team a
    wp_a = df[['team_a', 'wp_a', 'team_b']]
    wp_a.columns = ['team', 'wp', 'opp']

    # team b
    wp_b = df[['team_b', 'wp_b', 'team_a']]
    wp_b.columns = ['team', 'wp', 'opp']

    # combine
    stacked = pd.concat([wp_a, wp_b], ignore_index=True)

    # sort highest to low, pick out top
    lock = stacked.sort_values('wp', ascending=False).iloc[0]
    return lock.to_dict()
```

Any other interesting named matchups? Maybe the closest? The “photo finish” of the week.

I’ll just put this one in a function right away, but know if I was writing it for the first time myself, I’d prob write it outside of a function first in my main Python file (sometimes I’ll even make a file called `working.py` or `scratch.py` where I can write code like this), then put it in a function when I was confident it worked.

```
In [39]:  
def photo_finish(df):  
    # get the std dev of win probs, lowest will be closest matchup  
    wp_std = df[['wp_a', 'wp_b']].std(axis=1)  
  
    # idxmin "index min" returns the index of the lowest value  
    closest_matchup_id = wp_std.idxmin()  
  
    return df.loc[closest_matchup_id].to_dict()  
  
In [40]: photo_finish(matchup_df)  
Out[40]:  
{'wp_a': 0.5,  
 'wp_b': 0.5,  
 'over_under': 221.98,  
 'spread': 0.5,  
 'ml': -102,  
 'team_a': 'nbraun',  
 'team_b': 'carnsc815'}
```

These named results don’t necessarily have to be functions. Often we can get interesting things out of it with just one line. For example, maybe we want to call out the matchup with the lowest over under for the “Lowest firepower of the week”.

```
In [47]: matchup_df.sort_values('over_under').iloc[0].to_dict()  
Out[47]:  
{'wp_a': 0.58,  
 'wp_b': 0.42,  
 'over_under': 170.38,  
 'spread': 5.5,  
 'ml': -138,  
 'team_a': 'Meat11',  
 'team_b': 'Brusda'}
```

Nice. This is definitely something my league would be interested in (especially the guys who are favored).

Analyzing Teams

Another way to look at the league any given week is by team. As always, let's start with one particular team, then put it in a function once we know it works.

We'll stick with "team 1".

```
In [1]: team1_roster = lineup_by_team(team1)

In [2]: team1_sims = sims[team1_roster]

In [3]: team1_sims.head()
Out[3]:
   1091      349    ...     1228      5159
0  15.257107  6.547626    ...  5.756176  4.976292
1  24.679233 11.237680    ...  6.271533 14.421049
2   7.397607  3.893144    ...  5.179903  6.249771
3  25.821673 12.554041    ... 14.406736  6.953320
4  41.633112  3.024962    ...  0.950506 13.464130
```

Let's add everything up for total points and look at some summary stats.

```
In [4]: team1_total = joe_sims.sum(axis=1)

In [5]: team1_total.describe(percentiles=[.05, .25, .5, .75, .95])
Out[5]:
   count      1000.000000
   mean       111.689785
   std        17.439268
   min        65.859225
   5%         84.375465
   25%        100.080499
   50%        110.853531
   75%        122.609125
   95%        142.772065
   max        181.268537
```

There are a few other things that might be interesting (maybe share of points from each position?) but let's call it good for now.

Like we did for the matchup analysis, let's put all these in functions.

```
def summarize_team(sims):
    """
    Calculate summary stats on one set of teams.
    """
    totals = sims.sum(axis=1)
    # note: dropping count, min, max since those aren't that useful
    stats = (totals.describe(percentiles=[.05, .25, .5, .75, .95])
              [['mean', 'std', '5%', '25%', '50%', '75%', '95%']].to_dict())

    # maybe share of points by each pos? commented out now but could look
    # if
    # interesting

    # stats['qb'] = sims.iloc[:,0].mean()
    # stats['rb'] = sims.iloc[:,1:3].sum(axis=1).mean()
    # stats['flex'] = sims.iloc[:,3].mean()
    # stats['wr'] = sims.iloc[:,4:6].sum(axis=1).mean()
    # stats['te'] = sims.iloc[:,6].mean()
    # stats['k'] = sims.iloc[:,7].mean()
    # stats['dst'] = sims.iloc[:,8].mean()

    return stats
```

And — just like we did for matchups — let's loop over all teams, apply it and turn it into a DataFrame.

```
In [6]: team_list = []

In [7]:
for team_id in teams['team_id']:
    team_lineup = lineup_by_team(team_id)
    working_team_dict = summarize_team(sims[team_lineup])
    working_team_dict['team_id'] = team_id

    team_list.append(working_team_dict)

In [8]: team_df = DataFrame(team_list).set_index('team_id')

In [9]: team_df.round(2)
Out[9]:
   mean    std    5%    25%    50%    75%    95%
team_id
1747268  89.00  15.67  64.71  78.24  88.17  99.03  116.27
1605152  96.15  17.51  67.40  83.71  95.98  108.03  125.86
1747266  88.50  18.98  58.85  74.54  87.57  101.38  120.55
1605155  83.17  15.83  57.32  72.24  83.40  93.43  109.48
1605147  111.04  19.74  79.64  97.18  110.90  124.47  143.68
1605156  111.69  17.44  84.38  100.08  110.85  122.61  142.77
1664196  110.66  15.56  85.26  99.98  110.31  120.73  136.69
1603352  91.74  16.63  64.66  80.65  91.26  103.04  118.55
1605157  85.81  14.11  63.17  76.08  85.91  94.68  109.85
1605148  113.34  15.54  88.75  102.85  112.41  122.76  140.78
1605151  96.82  18.48  67.66  83.75  95.97  109.36  128.92
1605154  89.56  17.79  59.76  78.10  89.86  100.90  117.26
```

High and Low

Like many leagues, our league has a small payout and penalty for whoever gets the high and low score. Simulations make it easy to calculate each team's probability of getting either.

The first step is getting a DataFrame where we figure out each team's total for each simulation.

```
In [10]:
totals_by_team = pd.concat(
    [(sims[lineup_by_team(team_id)].sum(axis=1)
      .to_frame(team_id)) for team_id in teams['team_id']], axis=1)
```

Aside - walking through totals_by_team

Is the code above confusing? It's all things we went over in LTCWFF, but — just this once — let's walk through it. (If it's not confusing feel free to skip ahead.)

Let's start with the outer-most function, `pd.concat`. Remember, `concat` is how you stick DataFrames together — either stacked on top (like a snowman) when `axis=0` (the default), or side by side (like crayons in a box) when `axis=1`. Here's it's `axis=1`, so these are going side by side.

The first argument to `concat` is always a list with *what* you want to stick together.

That list is a comprehension:

```
[(sims[lineup_by_team(team_id)].sum(axis=1)
.to_frame(team_id)) for team_id in teams['team_id']]
```

Remember, the last part of the comprehension (`teams['team_id']`) is what we're starting with:

```
In [11]: teams['team_id']
Out[11]:
0    1605154
1    1605149
2    1605156
3    1605155
4    1605147
5    1605151
6    1605153
7    1664196
8    1603352
9    1605157
10   1605148
11   1605152
```

So we're going over each `team_id` and doing the following to it

```
(sims[lineup_by_team(team_id)].sum(axis=1).to_frame(team_id))
```

If you're not sure what that means just look at a specific team. Here's my team:

```
In [12]: team_id = 1605156

In [13]: (sims[lineup_by_team(team_id)].sum(axis=1).to_frame(team_id))
Out[13]:
1605156
0    88.913796
1    101.462538
2    74.157969
3    116.929091
4    119.253753
..
995   89.493737
996   138.884594
997   114.450031
998   157.443222
999   105.784600
```

So it's finding my players (`lineup_by_owner(1605156)`, finding their simulations (passing them to `sims[...]`, then summing them up by row). Finally we have `to_frame` which is a way to turn Series into a one column DataFrame with whatever name you want.

The final result:

```
In [14]: totals_by_team.head()
Out[14]:
    1747268      1605152     ...      1605151      1605154
0    87.699129   96.920501   ...    98.200156   75.195873
1    79.054700   69.065687   ...  110.436149  92.364231
2   108.847134   83.234173   ...   97.523246  94.472240
3    76.245210   64.730995   ...  93.732583  70.515056
4   109.133325  113.390178   ... 104.979163  69.460524
```

Back to analyzing teams

OK. So we have our `totals_by_team` data, where the columns are total points for each team, and the rows are separate simulations. With 12 teams and 1000 sims that's:

```
In [15]: totals_by_team.shape
Out[15]: (1000, 12)
```

We want to figure out which team is most likely to get the high score, so what should we try? Maybe `max`?

```
In [16]: totals_by_team.max(axis=1).head()
Out[16]:
0    124.729784
1    127.568335
2    108.847134
3    137.684795
4    119.253753
```

That's almost what we want, but not quite. The Pandas function `max` returns the actual *value* of the high score, but we want to know *which team* scored the max.

What we need is another function, `idxmax`. This returns the “index” (the name of the column when applied to each row) where the max was located. It's exactly what we need:

```
In [18]: totals_by_team.idxmax(axis=1).head()
Out[18]:
0    1605148
1    1605148
2    1747268
3    1664196
4    1605156
```

Now we can apply it to the data, and check frequencies with `value_counts`.

```
In [19]: totals_by_team.idxmax(axis=1).value_counts(normalize=True)
Out[19]:
1605147    0.226
1605148    0.214
1605156    0.203
1664196    0.148
1605151    0.069
1605152    0.044
1747266    0.030
1605154    0.021
1747268    0.019
1603352    0.019
1605157    0.004
1605155    0.003
```

We might as well add it to our team summary DataFrame, along with the probability of getting the low, which is just `idxmin`.

```
In [20]:
team_df['p_high'] = (totals_by_team.idxmax(axis=1)
                      .value_counts(normalize=True))

In [21]:
team_df['p_low'] = (totals_by_team.idxmin(axis=1)
                     .value_counts(normalize=True))

In [22]: team_df[['mean', 'p_high', 'p_low']]
Out[22]:
          mean   p_high   p_low
team_id
1747268  88.999290  0.019  0.115
1605152  96.152922  0.044  0.066
1747266  88.502169  0.030  0.173
1605155  83.166455  0.003  0.204
1605147  111.041666  0.226  0.018
1605156  111.689785  0.203  0.004
1664196  110.656653  0.148  0.002
1603352  91.736516  0.019  0.090
1605157  85.808259  0.004  0.135
1605148  113.339037  0.214  0.001
1605151  96.822379  0.069  0.067
1605154  89.559860  0.021  0.125
```

And what are the values of the high and low, on average?

```
In [23]: high_score = totals_by_team.max(axis=1)
In [23]: low_score = totals_by_team.min(axis=1)

In [24]:
pd.concat([
    high_score.describe(percentiles=[.05, .25, .5, .75, .95]),
    low_score.describe(percentiles=[.05, .25, .5, .75, .95])], axis=1)

Out[25]:
   0          1
count 1000.000000 1000.000000
mean 131.560932 65.892413
std 12.156450 9.421779
min 97.514115 36.953144
5% 114.050015 49.859302
25% 122.889788 59.830563
50% 130.379274 66.273816
75% 138.908043 72.658449
95% 152.328206 81.040505
max 183.563316 91.939920
```

About 130 and 65, with a standard deviation of 12 and 9.

Our final team_df:

```
In [26]: team_df.round(2)
Out[26]:
   mean   std    5%   25%   50%   75%   95%
team_id
1747268  89.00  15.67  64.71  78.24  88.17  99.03  116.27
1605152  96.15  17.51  67.40  83.71  95.98  108.03  125.86
1747266  88.50  18.98  58.85  74.54  87.57  101.38  120.55
1605155  83.17  15.83  57.32  72.24  83.40  93.43  109.48
1605147 111.04  19.74  79.64  97.18 110.90 124.47 143.68
1605156 111.69  17.44  84.38 100.08 110.85 122.61 142.77
1664196 110.66  15.56  85.26  99.98 110.31 120.73 136.69
1603352  91.74  16.63  64.66  80.65  91.26 103.04 118.55
1605157  85.81  14.11  63.17  76.08  85.91  94.68 109.85
1605148 113.34  15.54  88.75 102.85 112.41 122.76 140.78
1605151  96.82  18.48  67.66  83.75  95.97 109.36 128.92
1605154  89.56  17.79  59.76  78.10  89.86 100.90 117.26
```

Aside: high/low scores and how more data → less variance

One interesting thing about these projected high and low scores is how the projection is tighter around the mean compared to any individual team. This is a good example of how more data → less variance. Any one team might get lucky or unlucky and have a good or bad score, but the league wide high

and lows — which are a function of all 12 teams — involve more data and take more “luck” to move around.

This is reflected in both the standard deviations — 13, 10 for high, low scores (vs 18-22 for individual teams) — and the ranges. According to the model, there’s a 50% chance the low is between 55 and 69 points (the 25% and 75% percentiles), which is a range of 13 points.

But for each team, that 50% probability range is bigger:

```
In [27]: team_df['75%'] - team_df['25%']
Out[27]:
team_id
1747268    20.789890
1605152    24.315746
1747266    26.842237
1605155    21.192433
1605147    27.284283
1605156    22.528626
1664196    20.747860
1603352    22.388996
1605157    18.600425
1605148    19.912877
1605151    25.604735
1605154    22.794823
```

Back to the team analysis

Like the matchup_df, it’d be nicer to view actual names vs just team ids. Let’s do that.

```
In [28]:
# add owner
team_df = (pd.merge(team_df, teams[['team_id', 'owner_name']],
                     left_index=True, right_on = 'team_id')
            .set_index('owner_name')
            .drop('team_id', axis=1))
```

And the result:

```
In [29]: team_df.round(2)
Out[29]:
   mean    std     5% ...    95%  p_high  p_low
owner_name
CalBrusda      89.00  15.67  64.71 ...  116.27    0.02   0.12
WesHeroux       96.15  17.51  67.40 ...  125.86    0.04   0.07
Meat11          88.50  18.98  58.85 ...  120.55    0.03   0.17
Brusda          83.17  15.83  57.32 ...  109.48    0.00   0.20
carnsc815       111.04 19.74  79.64 ...  143.68    0.23   0.02
nbraun          111.69 17.44  84.38 ...  142.77    0.20   0.00
MegRyan0113     110.66 15.56  85.26 ...  136.69    0.15   0.00
UnkleJim         91.74  16.63  64.66 ...  118.55    0.02   0.09
JBKBDomination  85.81  14.11  63.17 ...  109.85    0.00   0.14
Lzrlightshow     113.34 15.54  88.75 ...  140.78    0.21   0.00
Edmundo          96.82  18.48  67.66 ...  128.92    0.07   0.07
ccarns           89.56  17.79  59.76 ...  117.26    0.02   0.12
```

Writing to a File

Just like we did in the auto who do I start section, let's write this out to a file `league_analysis.txt` file in our `HOST_LEAGUE_ID_2021-WK` directory.

```
In [1]:
league_wk_output_dir = path.join(
    OUTPUT_PATH, f'{league["host"]}_{LEAGUE_ID}_{SEASON}-{str(WEEK).zfill(2)}')

In [2]:
Path(league_wk_output_dir).mkdir(exist_ok=True, parents=True)

In [3]: output_file = path.join(league_wk_output_dir, 'league_analysis.txt')
```

And writing it:

```
with open(output_file, 'w') as f:
    print(dedent(
        f"""
        ****
        Matchup Projections, Week {WEEK} - {SEASON}
        ****
    """), file=f)
    print(matchup_df, file=f)

    print(dedent(
        f"""
        ****
        Team Projections, Week {WEEK} - {SEASON}
        ****
    """), file=f)

    print(team_df.round(2).sort_values('mean', ascending=False),
          file=f)

lock = lock_of_week(matchup_df)
close = photo_finish(matchup_df)
meh = matchup_df.sort_values('over_under').iloc[0]

print(dedent("""
    Lock of the week:"""),
      file=f)
print(f"{lock['team']} over {lock['opp']} - {lock['wp']}", file=f)

print(dedent("""
    Photo-finish of the week::"""),
      file=f)
print(f"{close['team_a']} vs {close['team_b']}, {close['wp_a']}-{close['wp_b']}", file=f)

print(dedent("""
    Most unexciting game of the week:"""),
      file=f)
print(f"{meh['team_a']} vs {meh['team_b']}, {meh['over_under']}", file=f)
```

And the text file output we created as a result:

***** Matchup Projections, Week 2 - 2023 *****								
0	wp_a	wp_b	over_under	spread	ml	team_a	team_b	
0	0.50	0.50	221.98	0.5	-102	nbraun	carnsc815	
1	0.40	0.60	185.82	-6.0	149	ccarns	Edmundo	
2	0.58	0.42	170.38	5.5	-138	Meat11	Brusda	
3	0.90	0.10	198.01	27.5	-931	Lzrlightshow	JBKBDomination	
4	0.20	0.80	202.50	-19.0	390	UnkleJim	MegRyan0113	
5	0.62	0.38	184.20	8.0	-163	WesHeroux	CalBrusda	
***** Team Projections, Week 2 - 2023 *****								
owner_name	mean	std	5%	25%	50%	75%	95%	...
Lzrlightshow	113.34	15.54	88.75	102.85	112.41	122.76	140.78	...
nbraun	111.69	17.44	84.38	100.08	110.85	122.61	142.77	...
carnsc815	111.04	19.74	79.64	97.18	110.90	124.47	143.68	...
MegRyan0113	110.66	15.56	85.26	99.98	110.31	120.73	136.69	...
Edmundo	96.82	18.48	67.66	83.75	95.97	109.36	128.92	...
WesHeroux	96.15	17.51	67.40	83.71	95.98	108.03	125.86	...
UnkleJim	91.74	16.63	64.66	80.65	91.26	103.04	118.55	...
ccarns	89.56	17.79	59.76	78.10	89.86	100.90	117.26	...
CalBrusda	89.00	15.67	64.71	78.24	88.17	99.03	116.27	...
Meat11	88.50	18.98	58.85	74.54	87.57	101.38	120.55	...
JBKBDomination	85.81	14.11	63.17	76.08	85.91	94.68	109.85	...
Brusda	83.17	15.83	57.32	72.24	83.40	93.43	109.48	...
<i>Lock of the week:</i> Lzrlightshow over JBKBDomination - 0.9								
<i>Photo-finish of the week::</i> nbraun vs carnsc815, 0.5-0.5								
<i>Most unexciting game of the week:</i> Meat11 vs Brusda, 170.38								

Plots

Now let's do some plots. Let's start by looking at the distributions of everyone's scores.

This (which we created above as part of our high/low score analysis) is a good place to start:

```
In [1]: totals_by_team.head()
Out[1]:
    1747268      1605152 ...      1605151      1605154
0   87.699129  96.920501 ...   98.200156  75.195873
1   79.054700  69.065687 ...  110.436149  92.364231
2  108.847134  83.234173 ...  97.523246  94.472240
3   76.245210  64.730995 ...  93.732583  70.515056
4  109.133325 113.390178 ... 104.979163  69.460524
```

For plotting, we'll be using seaborn, which means our data needs to be in long form.

Moving from wide to long form should make you think of `stack`:

```
In [2]: totals_by_team.stack().head()
Out[2]:
0  1747268      87.699129
1605152      96.920501
1747266      69.335147
1605155      120.613551
1605147      105.280698
```

Looks like what we want. We just have to `reset_index` and rename the columns.

```
In [3]: teams_long = totals_by_team.stack().reset_index()
In [3]: teams_long.columns = ['sim', 'team_id', 'pts']

In [4]: teams_long.head(15)
Out[4]:
    sim  team_id      pts
0     0  1747268  87.699129
1     0  1605152  96.920501
2     0  1747266  69.335147
3     0  1605155  120.613551
4     0  1605147  105.280698
5     0  1605156  88.913796
6     0  1664196  115.101735
7     0  1603352  94.453438
8     0  1605157  101.186519
9     0  1605148  124.729784
10    0  1605151  98.200156
11    0  1605154  75.195873
12    1  1747268  79.054700
13    1  1605152  69.065687
14    1  1747266  90.759712
```

And now plotting is easy. Note we're replacing `team_id` with `owner_name` on the plot. We'll save it in our `HOST_LEAGUE_ID_2021-WK` directory.

```
g = sns.FacetGrid(teams_long.replace(team_to_owner), hue='team', aspect=2)
g = g.map(sns.kdeplot, 'pts', shade=True)
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle(f'Team Points Distributions - Week {WEEK}')
g.fig.savefig(path.join(LEAGUE_PATH, f'team_dist_{WEEK}.jpg'),
bbox_inches='tight', dpi=500)
```

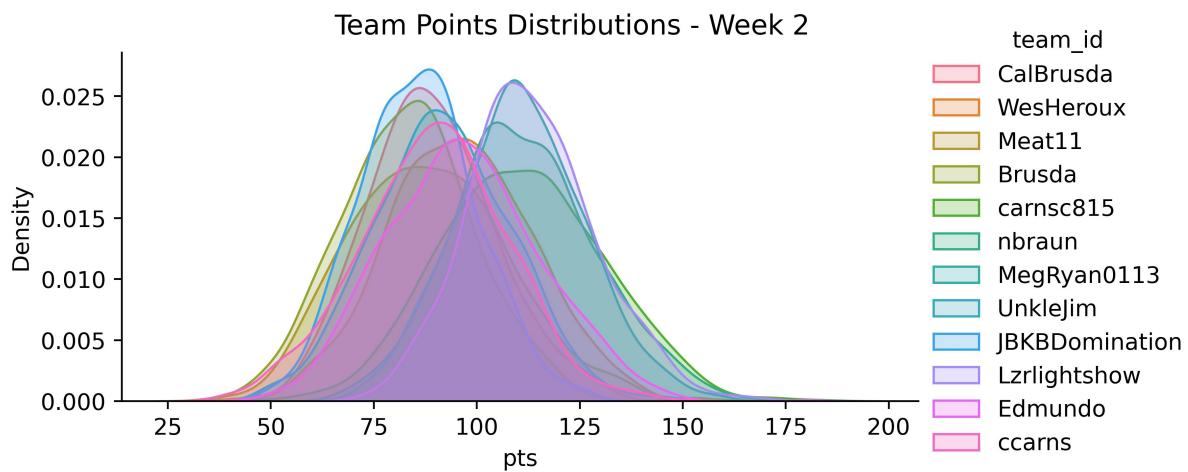


Figure 0.1: Team Distributions, Week 2 2023

This is cool, but it's a bit crowded. Let's try splitting out by matchup.

First we need to add in — for each team — info about the matchup it's playing in. We can convert our schedule to that with the `schedule_long` function we wrote earlier (and saved to `utilities.py`).

```
In [5]: schedule_team = schedule_long(schedule).query(f"week == {WEEK}")

In [6]: schedule_team
Out[6]:
   team_id  opp_id  matchup_id  season  week  league_id
6    1605156  1605147  53623240  2023     2    316893
7    1605154  1605151  53623242  2023     2    316893
8    1747266  1605155  53623238  2023     2    316893
9    1605148  1605157  53623241  2023     2    316893
10   1603352  1664196  53623239  2023     2    316893
11   1605152  1747268  53623237  2023     2    316893
90   1605147  1605156  53623240  2023     2    316893
91   1605151  1605154  53623242  2023     2    316893
92   1605155  1747266  53623238  2023     2    316893
93   1605157  1605148  53623241  2023     2    316893
94   1664196  1603352  53623239  2023     2    316893
95   1747268  1605152  53623237  2023     2    316893
```

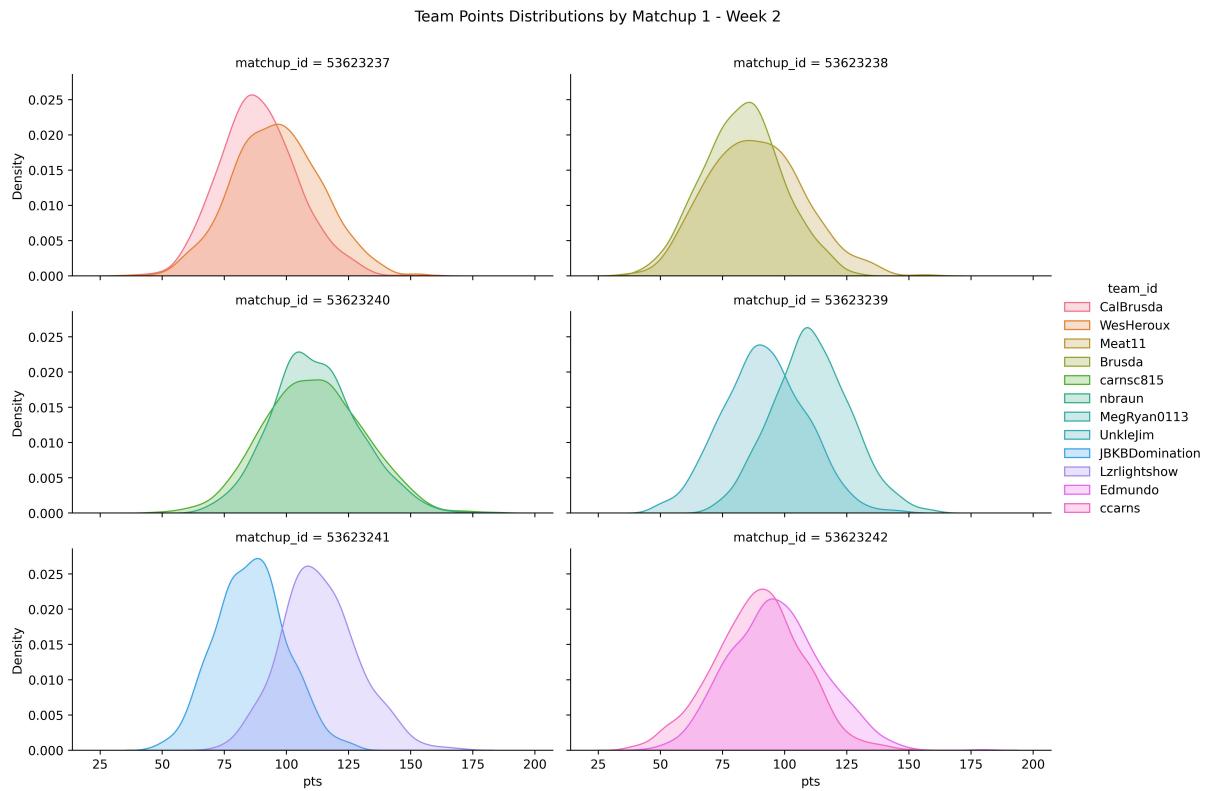
Merging the matchup info is straightforward:

```
In [7]: teams_long_w_matchup = pd.merge(teams_long, schedule_team[['team',
                                                               'matchup_id']])

In [8]: teams_long_w_matchup.head()
Out[8]:
   sim  team_id      pts  matchup_id
0    0  1747268  87.699129  53623237
1    1  1747268  79.054700  53623237
2    2  1747268  108.847134  53623237
3    3  1747268  76.245210  53623237
4    4  1747268  109.133325  53623237
```

Now we can do separate plots for each matchup_id.

```
g = sns.FacetGrid(teams_long_w_matchup.replace(team_to_owner), hue='team',
                  col='matchup_id', col_wrap=2, aspect=2)
g = g.map(sns.kdeplot, 'pts', shade=True)
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle(f'Team Points Distributions by Matchup - Week {WEEK}')
g.fig.savefig(path.join(league_wk_output_dir,
                      'team_dist_by_matchup1.jpg'),
              bbox_inches='tight', dpi=500)
```

**Figure 0.2:** Team Distributions by Matchup, Week 2 2023

This is fine but having labeling each plot with `matchup_id` is kind of lame, let's get a description. We can generate one from the schedule.

```
In [9]:  
schedule_this_week['desc'] = (  
    schedule_this_week['team2_id'].replace(team_to_owner) +  
    ' v ' +  
    schedule_this_week['team1_id'].replace(team_to_owner))  
  
In [10]: schedule_this_week[['matchup_id', 'desc']]  
Out[10]:  
   matchup_id          desc  
6      49030929  UnkleJim v Lzrlightshow  
7      49030927      carnsc815 v Edmundo  
8      49030925  Plz-not-last v ccarns  
9      49030926      nbraun v BRUZDA  
10     49030930  JBKBDomination v WesHeroux  
11     49030928  MegRyan0113 v LisaJean
```

Then just merge it in:

```
In [11]:  
teams_long_w_desc = pd.merge(teams_long_w_matchup,  
                           schedule_this_week[['matchup_id', 'desc']])
```

And tweak this part of our plot:

```
g = sns.FacetGrid(teams_long_w_desc.replace(team_to_owner), hue='team',  
                  col='desc', col_wrap=2, aspect=2)
```

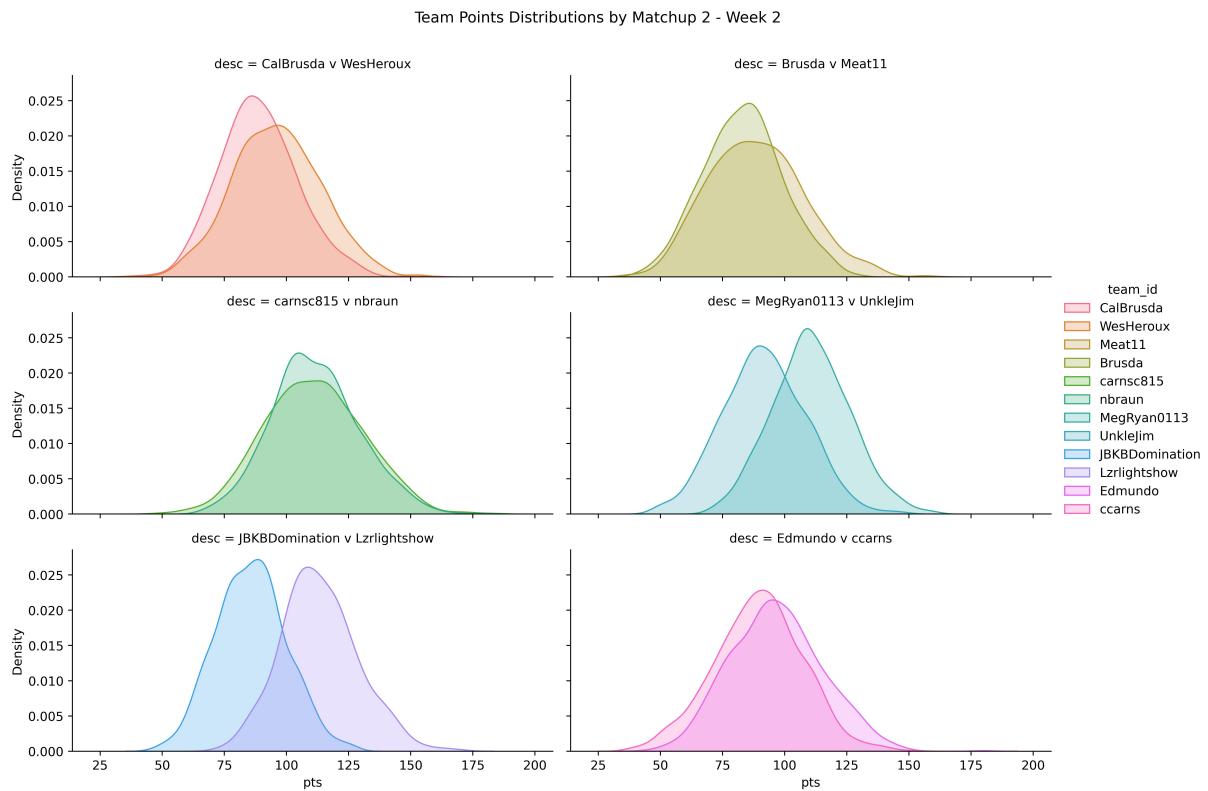


Figure 0.3: Team Distributions by Matchup, with Description, Week 2 2021

And there we go. See the [quickstart instructions](#) for more on the cleaned up version.

9. Project #4: Best Ball Projector

Best Ball Leagues

Best ball leagues have become pretty popular in recent years.

How it works: you draft your team at the beginning of the year.

That's it.

There's no in season management, etc. Each week, everyone on your roster plays, and the site calculates your highest score (using your best QB, two best RBs etc). At the end of the season, the team with highest points wins.

The one best-ball league I'm in is 20 spots and a starting lineup of 1 QB, 2 RB, 3 WR, 1 TE, 1 FLEX, 1 DST.

Considering the main draw of best-ball leagues is that you just draft without worrying about the regular season, a weekly-best ball analyzer and projector is a bit of an oxymoron.

We'll still do it because it's a good fantasy related opportunity to work on our Python and Pandas and provides some genuinely interesting (and accurate) results.

It also shows off the power of these simulations. Projecting the output of a best ball team (and seeing how often different guys are part of your lineup) is very easy to do using simulations and almost impossible to do accurately any other way.

And it's not like it isn't worthwhile. Some people may be invested enough in particular leagues (especially as they come down to the end), that forecasting results would be interesting.

The other thing building a tool like this does is let us simulate how substituting positions at the margin — a third QB vs a 7th WR say — impacts your score for a particular week.

Walkthrough

Let's get started. The code for this is in `./projects/bestball/bb_working.py`.

Let's pick up on line 39 (you should run the code above it). This loads the roster data from an actual bestball league I'm in (as of 2023). It also replaces numeric player ids with cleaned up player names to make it more interesting to follow along.

So we have my team:

```
In [1]: my_roster = rosters.query("team_id == 4")
```

```
In [2]: my_roster
```

```
Out[2]:
```

	name	player_id	player_position	...	team_id	
	name_id					
60	Justin Herbert herbert	494	QB	...	4	justin-
61	Tony Pollard pollard	703	RB	...	4	tony-
62	Isiah Pacheco pacheco	206	RB	...	4	isiah-
63	Cooper Kupp kupp	1139	WR	...	4	cooper-
64	DK Metcalf metcalf	720	WR	...	4	dk-
65	Brandon Aiyuk aiyuk	555	WR	...	4	brandon-
66	Dallas Goedert goedert	974	TE	...	4	dallas-
67	Antonio Gibson gibson	567	RB	...	4	antonio-
68	LAR	5168	DST	...	4	
69	DAL	5159	DST	...	4	
70	Tyjae Spears spears	19	RB	...	4	tyjae-
71	Sean Tucker tucker	20	RB	...	4	sean-
72	Puka Nacua nacua	65	WR	...	4	puka-
73	Brock Purdy purdy	156	QB	...	4	brock-
74	Skyy Moore moore	222	WR	...	4	skyy-
75	Brian Robinson robinson	173	RB	...	4	brian-
76	Romeo Doubs doubs	235	WR	...	4	romeo-
77	Nico Collins collins	380	WR	...	4	nico-
78	Juwani Johnson johnson	576	TE	...	4	juwan-
79	Michael Gallup gallup	946	WR	...	4	michael-

Along with the named sims (for just my team):

```
In [3]: my_sims.head()
Out[3]:
   justin-herbert  tony-pollard ... juwan-johnson michael-gallup
0      19.644986    18.239836 ...      0.379260    14.585192
1      44.899493    24.059125 ...      0.826263    2.042065
2      30.891677    26.115587 ...     17.451308    2.180005
3      45.127626     8.017181 ...      9.724999    3.820237
4      31.768032    28.349999 ...     15.882400    7.448767
```

We know we're going to be working positions and lists of players at each position a lot. It'd be nice to easily get a list of players for any position. Let's build a dictionary.

```
In [4]:
pos_dict = {pos: list(roster
                      .query(f"position == '{pos.upper()}'")['fm_id']
                      .values) for pos in ['qb', 'rb', 'wr', 'te', 'dst']}
--
```



```
In [5]: pos_dict
Out[5]:
{'qb': ['justin-herbert', 'brock-purdy'],
 'rb': ['tony-pollard',
        'isiah-pacheco',
        'antonio-gibson',
        'tyjae-spears',
        'sean-tucker',
        'brian-robinson'],
 'wr': ['cooper-kupp',
        'dk-metcalf',
        'brandon-aiyuk',
        'puka-nacua',
        'skyy-moore',
        'romeo-doubs',
        'nico-collins',
        'michael-gallup'],
 'te': ['dallas-goedert', 'juwan-johnson'],
 'dst': ['lar', 'dal']}
```

That way we can quickly work with, say, our running back sims with:

```
In [6]: nsims[pos_dict['rb']].head()
Out[6]:
   tony-pollard  isiah-pacheco ... sean-tucker  brian-robinson
0      12.689639    15.350473 ...      12.896203    21.516283
1      18.094261    20.211971 ...      13.351053    10.389005
2      26.461849     1.748790 ...      12.696113     9.659679
3      10.204148    24.465364 ...      3.297151    16.057333
4      26.341993    16.309214 ...     14.191757    11.406996
```

Using the simulations to find the max score

OK. The goal is to project our best ball roster. That means looking at the simulated fantasy scores from each position and picking out the highest one.

In the one position, non-flex eligible case (e.g. QB) this is simple. It's just a matter of applying the `max` function.

```
In [7]: nsims[pos_dict['qb']].max(axis=1).head(10)
Out[7]:
0    19.644986
1    44.899493
2    30.891677
3    45.127626
4    31.768032
5    12.078713
6    32.396756
7    23.558696
8    26.499252
9    20.376479
```

And, if we're interested in who we're starting in each simulation, the `idxmax`, which when `axis=1` returns the name of the column with the highest score.

```
In [8]: nsims[pos_dict['qb']].idxmax(axis=1).head(10)
Out[8]:
0    justin-herbert
1    justin-herbert
2    justin-herbert
3    justin-herbert
4    justin-herbert
5    justin-herbert
6    justin-herbert
7    brock-purdy
8    brock-purdy
9    justin-herbert
```

Interesting. That's for QBs, where we start one player. DST and TE will be similar. But what about running backs, where we start two of them?

The first RB is easy (just `max` and `idxmax` again). The problem is after that: there's no built in “second_max” or “second idx max” function. We're going to have to build something ourselves.

Let's think: given one row (simulation), how would you go about figuring out who the two highest scoring RBs were?

As always, it's easiest to just dive into specifics. This gives us the first sim for my team:

```
In [9]: sim = my_sims.iloc[0]
```

```
In [10]: sim
Out[10]:
justin-herbert      19.644986
tony-pollard        18.239836
isiah-pacheco       9.853029
cooper-kupp         0.000000
dk-metcalf          2.644397
brandon-aiyuk        9.117376
dallas-goedert      16.688684
antonio-gibson      3.712600
lar                  9.255857
dal                  9.922289
tyjae-spears         8.291674
sean-tucker          1.467464
puka-nacua          21.009071
brock-purdy          12.343357
skyy-moore           6.752244
brian-robinson       10.338995
romeo-doubs          7.806143
nico-collins          10.701553
juwan-johnson        0.379260
michael-gallup        14.585192
```

Let's figure out the top RBs on this, then apply it to the rest of them.

First let's limit our analysis to RBs.

```
In [11]: sim.loc[pos_dict['rb']]
Out[11]:
tony-pollard      18.239836
isiah-pacheco      9.853029
antonio-gibson      3.712600
tyjae-spears        8.291674
sean-tucker          1.467464
brian-robinson       10.338995
```

Then we want to take the two highest. So we need to sort in descending order and take the top two.

```
In [12]: sim.loc[pos_dict['rb']].sort_values(ascending=False).iloc[:2]
Out[12]:
tony-pollard      18.239836
brian-robinson      10.338995
Name: 0, dtype: float64
```

This is something we're going to have to do again (with the WRs at the very least), so let's put it in a function.

```
def n_highest_scores_from_sim1(sim, players, n):
    return sim.loc[players].sort_values(ascending=False).iloc[:n]
```

And test it out:

```
In [13]: n_highest_scores_from_sim1(sim, pos_dict['rb'], 2)
Out[13]:
tony-pollard      18.239836
brian-robinson    10.338995
Name: 0, dtype: float64

In [14]: n_highest_scores_from_sim1(sim, pos_dict['wr'], 3)
Out[14]:
puka-nacua       21.009071
michael-gallup   14.585192
nico-collins     10.701553
Name: 0, dtype: float64
```

Now, we can go through all our simulations, call this function on every single one and stick everything together.

You iterate over rows with the `iterrows()` function. It returns a row along with its index, as you can see here.

```
In [15]:
for i, row in my_sims.head().iterrows():
    print(i)
    print(row)
--
0
justin-herbert    19.644986
tony-pollard      18.239836
isiah-pacheco    9.853029
cooper-kupp       0.000000
dk-metcalf        2.644397
...
1
justin-herbert    44.899493
tony-pollard      24.059125
isiah-pacheco    12.165260
cooper-kupp       0.000000
dk-metcalf        19.207177
...
```

Aside: I'm not sure we've covered `iterrows` in LTCWFF. For me personally, it's always something on the edge of my toolkit. I definitely use it sometimes, but I still usually have to look it up.

But we have a function and do know generally what we want to do (apply it to every row), so let's

Google it:

The first link that comes up if we Google, “how to loop over rows pandas” mentions `iterrows`:

<https://stackoverflow.com/questions/16476924/how-to-iterate-over-rows-in-a-dataframe-in-pandas>

After we get the top 2 RBs for each row (simulation), we want to stick them together, which means `concat`.

Let's try it on the first five:

```
In [16]:  
pd.concat([n_highest_scores_from_sim1(row, pos_dict['rb'], 2) for _, row  
          in sims.head().iterrows()], ignore_index=True)  
--  
Out[16]:  
0    18.239836  
1    10.338995  
2    24.059125  
3    12.165260  
4    26.115587  
5    17.262072  
6    23.981187  
7    11.609040  
8    28.349999  
9    13.820688
```

It's doing what we want (note the first two numbers, 18.24 and 10.34 are what we got when we ran it on just our one row), it's just that everything is jumbled together, and we have no way to tell which a particular row represents.

Let's modify our function to add the rank of each player and also the simulation we're working on.

```
def n_highest_scores_from_sim(sim, players, n):  
    df_ = sim.loc[players].sort_values(ascending=False).iloc[:n].  
        reset_index()  
    df_.columns = ['name', 'points']  
    df_['rank'] = range(1, n+1)  
    df_['sim'] = sim.name # note: name of each row is its index value  
    return df_
```

That gives us:

```
In [17]: n_highest_scores_from_sim(sim, pos_dict['rb'], 2)  
Out[17]:  
      name    points  rank  sim  
0  tony-pollard  18.239836    1    0  
1  brian-robinson  10.338995    2    0
```

This is good. Now let's apply it across all of our simulations.

```
rbs = pd.concat([n_highest_scores_from_sim(row, pos_dict['rb'], 2) for _,
row
in sims.iterrows()], ignore_index=True)
```

That give us:

```
In [18]: rbs.head(8)
Out[18]:
      name    points  rank  sim
0  tony-pollard  18.239836    1    0
1  brian-robinson  10.338995    2    0
2  tony-pollard  24.059125    1    1
3  isiah-pacheco  12.165260    2    1
4  tony-pollard  26.115587    1    2
5  brian-robinson  17.262072    2    2
6  isiah-pacheco  23.981187    1    3
7  brian-robinson  11.609040    2    3
```

So these are our first three simulations. Tony Pollard is RB1 on 2 of them, with Isiah Pacheco RB1 on the last. Robinson and Pacheco trade off being RB2.

Having everything in long form — where each line is a player/sim — might be useful, but it also might be useful to look at this in wide form too.

Let's unstack it (which is how you go from long → wide) it so each row is one sim. Remember both `stack` and `unstack` work with multi-indexes, so before we call it we have to set `['sim', 'rank']` equal to our index.

```
In [19]: rbs_wide = rbs.set_index(['sim', 'rank']).unstack()
In [20]: rbs_wide.head()
Out[20]:
      name          points
      rank        1           2
      sim
0  tony-pollard  brian-robinson  18.239836  10.338995
1  tony-pollard  isiah-pacheco  24.059125  12.165260
2  tony-pollard  brian-robinson  26.115587  17.262072
3  isiah-pacheco  brian-robinson  23.981187  11.609040
4  tony-pollard  antonio-gibson  28.349999  13.820688
```

This is nice, but personally find it confusing to work with multi level column names like this, which are created automatically as part of `unstack`.

Luckily we can get rid of them just by renaming them:

```
In [21]:  
rbs_wide.columns = ['rb1_name', 'rb2_name', 'rb1_points', 'rb2_points']
```

That gives us:

```
In [22]: rbs_wide.head()  
Out[22]:  
      rb1_name      rb2_name  rb1_points  rb2_points  
sim  
0    tony-pollard  brian-robinson   18.239836  10.338995  
1    tony-pollard  isiah-pacheco   24.059125  12.165260  
2    tony-pollard  brian-robinson   26.115587  17.262072  
3  isiah-pacheco  brian-robinson   23.981187  11.609040  
4    tony-pollard  antonio-gibson   28.349999  13.820688
```

And now we can easily analyze the scores:

```
In [23]: points_from_rbs = (  
                 rbs_wide[['rb1_points', 'rb2_points']].sum(axis=1))  
  
In [24]: points_from_rbs.mean()  
Out[24]: 38.28931281088912
```

Or answer questions like, how often is Pollard our RB1?

```
In [25]: rbs_wide['rb1_name'].value_counts(normalize=True)  
Out[25]:  
  
rb1_name  
tony-pollard      0.635  
isiah-pacheco     0.154  
brian-robinson    0.138  
tyjae-spears      0.027  
antonio-gibson    0.027  
sean-tucker        0.019  
Name: proportion, dtype: float64
```

We have all the pieces there to be able to project a score for best ball lineup. We have points from our top two RBs and can easily extend this to points from top 3 WRs.

Let's put it in a function and do it:

```
def top_n_by_pos(sims, pos, n):
    df_long = pd.concat([n_highest_scores_from_sim(row, pos_dict[pos], n)
        for
            _, row in sims.iterrows()], ignore_index=True)
    df_wide = df_long.set_index(['sim', 'rank']).unstack()
    df_wide.columns = ([f'{pos}{x}_name' for x in range(1, n+1)] +
        [f'{pos}{x}_points' for x in range(1, n+1)])
    return df_wide
```

```
In [26]: wrs_wide = top_n_by_pos(my_sims, 'wr', pos_dict['wr'], 3)

In [27]: wrs_wide.head()
Out[27]:
      wr1_name      wr2_name      wr3_name  wr1_points  wr2_points
sim
0      puka-nacua  michael-gallup  nico-collins   21.009071   14.585192
...
1      skyy-moore  nico-collins  puka-nacua   23.105118   19.979987
...
2      nico-collins  puka-nacua  brandon-aiyuk   27.684911   24.804430
...
3      dk-metcalf  brandon-aiyuk  puka-nacua   42.026594   22.402949
...
4      dk-metcalf  puka-nacua  romeo-doubs   21.612601   15.406181
...
```

Let's do it with other positions while we're at it, just so everything is in the same format:

```
In [28]:
qb_wide = top_n_by_pos(my_sims, 'qb', pos_dict['qb'], 1)
te_wide = top_n_by_pos(my_sims, 'te', pos_dict['te'], 1)
dst_wide = top_n_by_pos(my_sims, 'dst', pos_dict['dst'], 1)
--
```

Now we can sum up and analyze our points.

```
In [29]:  
# now sum up points  
proj_points = (  
    qb_wide['qb1_points'] +  
    te_wide['te1_points'] +  
    dst_wide['dst1_points'] +  
    rbs_wide[['rb1_points', 'rb2_points']].sum(axis=1) +  
    wrs_wide[['wr1_points', 'wr2_points', 'wr3_points']].sum(axis=1))  
  
# and analyze  
In [30]: proj_points.describe()  
Out[30]:  
count    1000.000000  
mean     151.550394  
std      20.904410  
min      90.031310  
25%     137.619935  
50%     151.267534  
75%     165.661940  
max     219.258801  
dtype: float64
```

Awesome! Except... most best ball leagues also include a flex spot.

Including a flex spot would mean what what we have above, *plus* the highest scoring RB, WR or TE who is NOT in the above.

There are a couple ways you could do this, and in real life it's very possible you start your way down a dead-end before stumbling on something that works.

After playing around with this, here's what I came up with:

What about modifying our `n_highest_scores_from_sim` function to return basically the opposite, the leftover guys.

```
def leftover_from_sim(sim, players, n):  
    df = sim.loc[players].sort_values(ascending=False).iloc[n:].[  
        reset_index()  
    df.columns = ['name', 'points']  
    df['sim'] = sim.name  
    return df
```

The key is in the `iloc[n:]` part. Now we're taking the guys from there on. Since there's only 1 flex spot, we don't need rank anymore either.

Now let's apply it to all the flex eligible players (RB/WR/TE):

```
In [31]:  
rbs_leftover = pd.concat([leftover_from_sim(row, pos_dict['rb']), 2)  
                           for _, row in my_sims.iterrows()], ignore_index=True)  
wrs_leftover = pd.concat([leftover_from_sim(row, pos_dict['wr']), 3)  
                           for _, row in my_sims.iterrows()], ignore_index=True)  
tes_leftover = pd.concat([leftover_from_sim(row, pos_dict['te']), 1)  
                           for _, row in my_sims.iterrows()], ignore_index=True)
```

Then stick them together:

```
In [32]:  
leftovers = pd.concat([rbs_leftover, wrs_leftover, tes_leftover],  
                      ignore_index=True)
```

So for sim 0, the left over guys are:

```
In [33]: leftovers.query("sim == 0")  
Out[33]:  
      name    points  sim  
0    isiah-pacheco  9.853029  0  
1    tyjae-spears  8.291674  0  
2    antonio-gibson  3.712600  0  
3    sean-tucker  1.467464  0  
4000  brandon-aiyuk  9.117376  0  
4001  romeo-doubs  7.806143  0  
4002  skyy-moore  6.752244  0  
4003  dk-metcalf  2.644397  0  
4004  cooper-kupp  0.000000  0  
9000  juwan-johnson  0.379260  0
```

Here our flex would be Pacheco, with 9.85 points. So we need to figure out the best scoring guy from each sim.

Once we're working with more than one sim, we'll need to use a `groupby` by sim.

If we were just interested in the points, we could do something like `max`:

```
In [34]: leftovers.groupby('sim').max()['points'].head()  
Out[34]:  
sim  
0    9.853029  
1    19.207177  
2    11.114427  
3    10.429868  
4    15.882400  
Name: points, dtype: float64
```

(Note there's Pacheco's 9.85 for sim 0). This is part of what we want. But it'd also be nice to know WHO got the max too.

In that case we need `idxmax`, which remember returns the value of the index of the max.

`idxmax` is a normal, Pandas function, which means we can use it with a groupby.

```
In [35]: max_points_index = leftovers.groupby('sim').idxmax()['points']

In [35]: max_points_index.head()
sim
0      0
1    4005
2    4010
3    4015
4    9004
```

By itself, this index isn't useful, but we can pass it to our `leftovers` data with `loc` like this:

```
In [36]: leftovers.loc[max_points_index].head()
Out[36]:
      name  points  sim
0  isiah-pacheco  9.853029  0
4005  dk-metcalf  19.207177  1
4010  romeo-doubs  11.114427  2
4015  romeo-doubs  10.429868  3
9004  juwan-johnson  15.882400  4
```

Perfect. That gives us the name and point value of the flex spot for each sim.

Now we want to link it up to our other DataFrame we made before (which had everything but the flex). Remember, that's in this format:

```
In [37]: pd.concat([qb_wide, rbs_wide], axis=1).head()
Out[37]:
      qb1_name  qb1_points  ...  rb1_points  rb2_points
sim
0  justin-herbert  19.644986  ...  18.239836  10.338995
1  justin-herbert  44.899493  ...  24.059125  12.165260
2  justin-herbert  30.891677  ...  26.115587  17.262072
3  justin-herbert  45.127626  ...  23.981187  11.609040
4  justin-herbert  31.768032  ...  28.349999  13.820688
```

That is, `sim` is the index, and we have name and points columns.

So let's make `flex_wide` that matches that format:

```
In [38]: flex_wide = leftovers.loc[max_points_index].set_index('sim')

In [39]: flex_wide.columns = ['flex_name', 'flex_points']

In [40]: flex_wide.head()
Out[40]:
      flex_name  flex_points
sim
0    isiah-pacheco     9.853029
1        dk-metcalf    19.207177
2    romeo-doubs     11.114427
3    romeo-doubs     10.429868
4   juwan-johnson    15.882400
```

Now finally, we can put everything together.

```
In [41]:
team_wide = pd.concat([qb_wide, rbs_wide, wrs_wide, te_wide, flex_wide,
                      dst_wide], axis=1)
--
```

```
In [42]: team_wide.tail()
Out[42]:
      qb1_name  qb1_points  ...  dst1_name  dst1_points
sim
995  justin-herbert    15.513885  ...        dal    15.897174
996  justin-herbert    21.485090  ...        dal    16.046387
997  justin-herbert    39.840548  ...        dal    22.882806
998  justin-herbert    28.326051  ...        lar     9.483803
999    brock-purdy     57.146211  ...        dal    17.192886
```

Looking at this, one thing that might be useful is splitting this out into separate player name and point value DataFrames.

```
In [43]:
pos = ['qb', 'rb1', 'rb2', 'wr1', 'wr2', 'wr3', 'te', 'flex', 'dst']

names = team_wide[[x for x in team_wide.columns if x.endswith('_name')]]
names.columns = pos

points = team_wide[[x for x in team_wide.columns if x.endswith('_points')]
                  ]]
points.columns = pos
```

That gives us:

```
In [44]: names.head()
Out[44]:
      qb          rb1 ...      flex  dst
sim
0  justin-herbert  tony-pollard ...  isiah-pacheco  dal
1  justin-herbert  tony-pollard ...  dk-metcalf  dal
2  justin-herbert  tony-pollard ...  romeo-doubs  dal
3  justin-herbert  isiah-pacheco ...  romeo-doubs  dal
4  justin-herbert  tony-pollard ...  juwan-johnson  dal
```

And

	qb	rb1	...	flex	dst
sim			...		
0	19.644986	18.239836	...	9.853029	9.922289
1	44.899493	24.059125	...	19.207177	24.022887
2	30.891677	26.115587	...	11.114427	16.761482
3	45.127626	23.981187	...	10.429868	12.436354
4	31.768032	28.349999	...	15.882400	36.821260

And lets us do things like summarize the range of outcomes:

```
In [45]: points.sum(axis=1).describe(percentiles=[.05, .25, .5, .75, .95])
Out[45]:
count    1000.000000
mean     164.731766
std      22.176191
min      100.544876
5%       128.318486
25%      150.024237
50%      164.133718
75%      180.346697
95%      201.242842
max      237.808111
```

Or check how often a player will appear at a position in our lineup.

```
In [46]: names['qb'].value_counts(normalize=True)
Out[46]:
qb
justin-herbert    0.643
brock-purdy      0.357
Name: proportion, dtype: float64

In [47]: names['wr2'].value_counts(normalize=True)
Out[47]:
wr2
dk-metcalf      0.196
puka-nacua      0.195
brandon-aiyuk    0.174
nico-collins    0.144
romeo-doubs     0.109
skyy-moore       0.094
michael-gallup   0.088
Name: proportion, dtype: float64

In [48]: names['flex'].value_counts(normalize=True)
Out[48]:
flex
isiah-pacheco   0.123
brian-robinson   0.107
brandon-aiyuk    0.081
tyjae-spears     0.075
puka-nacua       0.071
nico-collins     0.066
antonio-gibson   0.066
michael-gallup   0.066
romeo-doubs      0.062
dk-metcalf        0.060
skyy-moore        0.060
sean-tucker       0.052
tony-pollard      0.051
juwan-johnson    0.040
dallas-goedert    0.020
Name: proportion, dtype: float64
```

Presentation and formatting

One thing it might be interesting to do is stick all of these columns together, so we can run down a list and see how often each player ended up in each spot.

```
In [49]:  
usage = pd.concat([names[x].value_counts(normalize=True) for x in pos],  
                  axis=1, join='outer').fillna(0)  
usage.columns = [x.upper() for x in pos]  
--
```

```
In [50]: usage  
Out[50]:
```

	QB	RB1	RB2	WR1	WR2	WR3	TE	FLEX
	DST							
justin-herbert 0.000	0.643	0.000	0.000	0.000	0.000	0.000	0.000	0.000
brock-purdy 0.000	0.357	0.000	0.000	0.000	0.000	0.000	0.000	0.000
tony-pollard 0.000	0.000	0.635	0.202	0.000	0.000	0.000	0.000	0.051
isiah-pacheco 0.000	0.000	0.154	0.256	0.000	0.000	0.000	0.000	0.123
brian-robinson 0.000	0.000	0.138	0.254	0.000	0.000	0.000	0.000	0.107
tyjae-spears 0.000	0.000	0.027	0.101	0.000	0.000	0.000	0.000	0.075
antonio-gibson 0.000	0.000	0.027	0.103	0.000	0.000	0.000	0.000	0.066
sean-tucker 0.000	0.000	0.019	0.084	0.000	0.000	0.000	0.000	0.052
dk-metcalf 0.000	0.000	0.000	0.000	0.275	0.196	0.172	0.000	0.060
puka-nacua 0.000	0.000	0.000	0.000	0.264	0.195	0.149	0.000	0.071
brandon-aiyuk 0.000	0.000	0.000	0.000	0.168	0.174	0.149	0.000	0.081
nico-collins 0.000	0.000	0.000	0.000	0.139	0.144	0.177	0.000	0.066
romeo-doubs 0.000	0.000	0.000	0.000	0.062	0.109	0.123	0.000	0.062
skyy-moore 0.000	0.000	0.000	0.000	0.058	0.094	0.139	0.000	0.060
michael-gallup 0.000	0.000	0.000	0.000	0.034	0.088	0.091	0.000	0.066
dallas-goedert 0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.756	0.020
juwan-johnson 0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.244	0.040
dal 0.665	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
lar 0.335	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Each column should sum to 1, and each row sums to how often the player in question was used *anywhere*. So Pollard was used as RB1 64%, RB2 20% and FLEX 5%, which means he shows up in 89% of our lineups.

We can add that.

```
usage.columns = [x.upper() for x in usage.columns]
usage['ALL'] = usage.sum(axis=1)
```

	QB	RB1	RB2	WR1	WR2	WR3	TE	FLEX	DST	ALL
justin-herbert	0.64	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.64
brock-purdy	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.36
tony-pollard	0.00	0.64	0.20	0.00	0.00	0.00	0.00	0.05	0.00	0.89
isiah-pacheco	0.00	0.15	0.26	0.00	0.00	0.00	0.00	0.12	0.00	0.53
brian-robinson	0.00	0.14	0.25	0.00	0.00	0.00	0.00	0.11	0.00	0.50
tyjae-spears	0.00	0.03	0.10	0.00	0.00	0.00	0.00	0.08	0.00	0.20
antonio-gibson	0.00	0.03	0.10	0.00	0.00	0.00	0.00	0.07	0.00	0.20
sean-tucker	0.00	0.02	0.08	0.00	0.00	0.00	0.00	0.05	0.00	0.16
dk-metcalf	0.00	0.00	0.00	0.28	0.20	0.17	0.00	0.06	0.00	0.70
puka-nacua	0.00	0.00	0.00	0.26	0.20	0.15	0.00	0.07	0.00	0.68
brandon-aiyuk	0.00	0.00	0.00	0.17	0.17	0.15	0.00	0.08	0.00	0.57
nico-collins	0.00	0.00	0.00	0.14	0.14	0.18	0.00	0.07	0.00	0.53
romeo-doubs	0.00	0.00	0.00	0.06	0.11	0.12	0.00	0.06	0.00	0.36
skyy-moore	0.00	0.00	0.00	0.06	0.09	0.14	0.00	0.06	0.00	0.35
michael-gallup	0.00	0.00	0.00	0.03	0.09	0.09	0.00	0.07	0.00	0.28
dallas-goedert	0.00	0.00	0.00	0.00	0.00	0.00	0.76	0.02	0.00	0.78
juwan-johnson	0.00	0.00	0.00	0.00	0.00	0.00	0.24	0.04	0.00	0.28
dal	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.66	0.66
lar	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.34	0.34

Really, we don't need all the zeros (the only people eligible to be used at QB are Herbert and Purdy, that's why everyone else has a 0.00) so let's get rid of them so this looks a little better

```
In [52]: usage_clean = usage.round(2).astype(str)

In [52]:
for x in usage_clean.columns:
    usage_clean[x] = (usage_clean[x]
        .str.pad(4, fillchar='0', side='right')
        .str.replace('^0.00$', '', regex=True))
-- 

In [52]: usage_clean
Out[52]:
      QB   RB1   RB2   WR1   WR2   WR3    TE   FLEX   DST    ALL
justin-herbert  0.64
brock-purdy     0.36
tony-pollard     0.64  0.20
isiah-pacheco    0.15  0.26
brian-robinson   0.14  0.25
tyjae-spears     0.03  0.10
antonio-gibson   0.03  0.10
sean-tucker      0.02  0.08
dk-metcalf          0.28  0.20  0.17
puka-nacua        0.26  0.20  0.15
brandon-aiyuk      0.17  0.17  0.15
nico-collins       0.14  0.14  0.18
romeo-doubs        0.06  0.11  0.12
skyy-moore         0.06  0.09  0.14
michael-gallup     0.03  0.09  0.09
dallas-goedert      0.76  0.02
juwan-johnson      0.24  0.04
dal
lar

```

Perfect.

As always, let's make some plots too.

It might be interesting to look at the distributions of our players individually, as well as the distributions guys that end up being best too.

Let's try that with RBs.

By now the wide to long, stack → reset_index → rename columns pattern should be familiar.

```
players_long = my_sims[pos_dict['rb']].stack().reset_index()
players_long.columns = ['sim', 'player', 'points']
```

And the plot of just the players:

```

g = sns.FacetGrid(players_long, hue='player', aspect=2)
g = g.map(sns.kdeplot, 'points', shade=True)
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle(f'RB Projections - Best Ball')
plt.show()

```

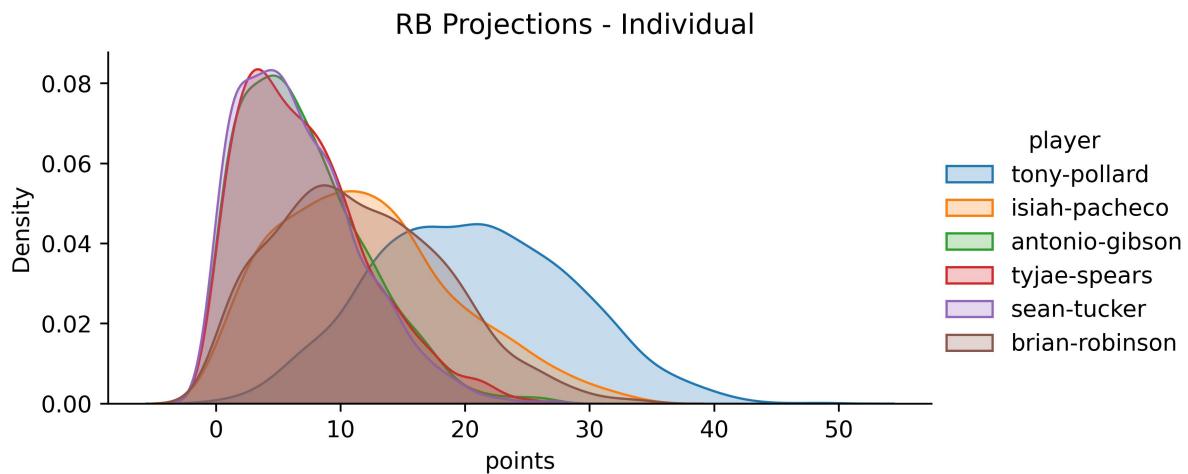


Figure 0.1: RB Individual Player Projections

Now let's tack on our RB1 and RB2 best ball projections to the end of it and plot it again.

Again, we need this in the long format seaborn likes.

```

# add in best ball rbs
bb_rb_long = points[['rb1', 'rb2']].stack().reset_index()
bb_rb_long.columns = ['sim', 'player', 'points']
players_long = pd.concat([players_long, bb_rb_long], ignore_index=True)
plot_data = pd.concat([plot_data, bb_rb_long], ignore_index=True)

```

Exact same code as before (just change the title):

```

# plot
g = sns.FacetGrid(plot_data, hue='player', aspect=2)
g = g.map(sns.kdeplot, 'points', shade=True)
g.add_legend()
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle(f'RB Projections w/ Best Ball')
plt.show()

```

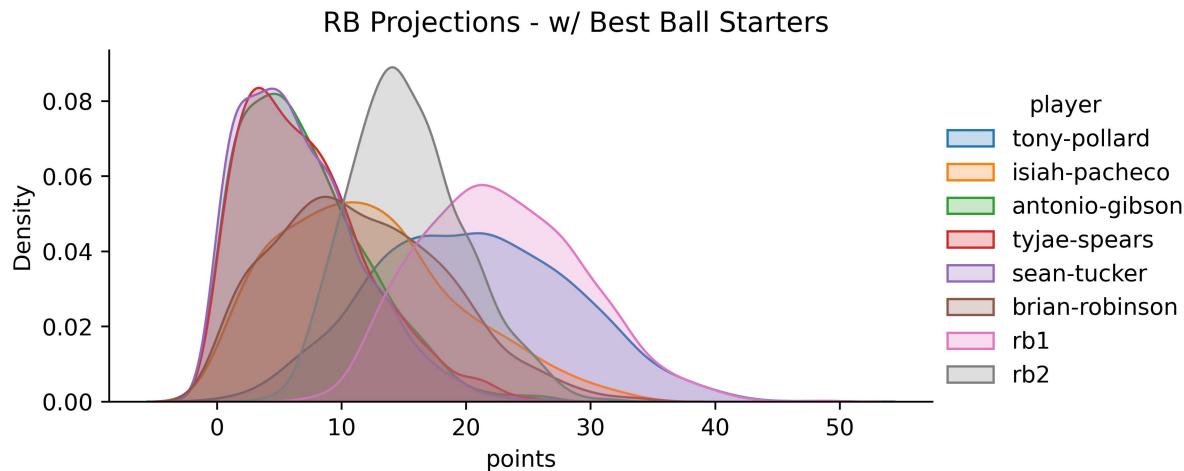


Figure 0.2: RB w/ Best Ball Starters

And there you go! Your own accurate best ball projector with plots.

Sleeper Best Ball Leagues

Of the four platforms we cover in this guide, to only Sleeper to my knowledge supports best ball leagues (it's what my best ball league uses).

So in the file `./projects/bestball/bb_sleeper.py` I've combined our Sleeper integration + the functions we wrote here (+ a bit of the league analyzer project) to analyze everyone's projections, probability of getting the high and low, etc. It also outputs everyone's usage as a csv file.

Enjoy!

Appendix A: Installing Python

Note: these are the same instructions as in LTCWFF, but I'm including these for readers who haven't read that.

Python

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

The book is project based, and you should be running code and exploring as you go through it. To do so, you need the ability to run Python 3 code and install packages. If you can do that and have a setup that works for you, great. If you do not, the easiest way to get one is from Anaconda.

Go to:

<https://www.anaconda.com/download>

And click on the right, green Download button to download the 3.x version (3.9 at time of this writing) for your operating system.

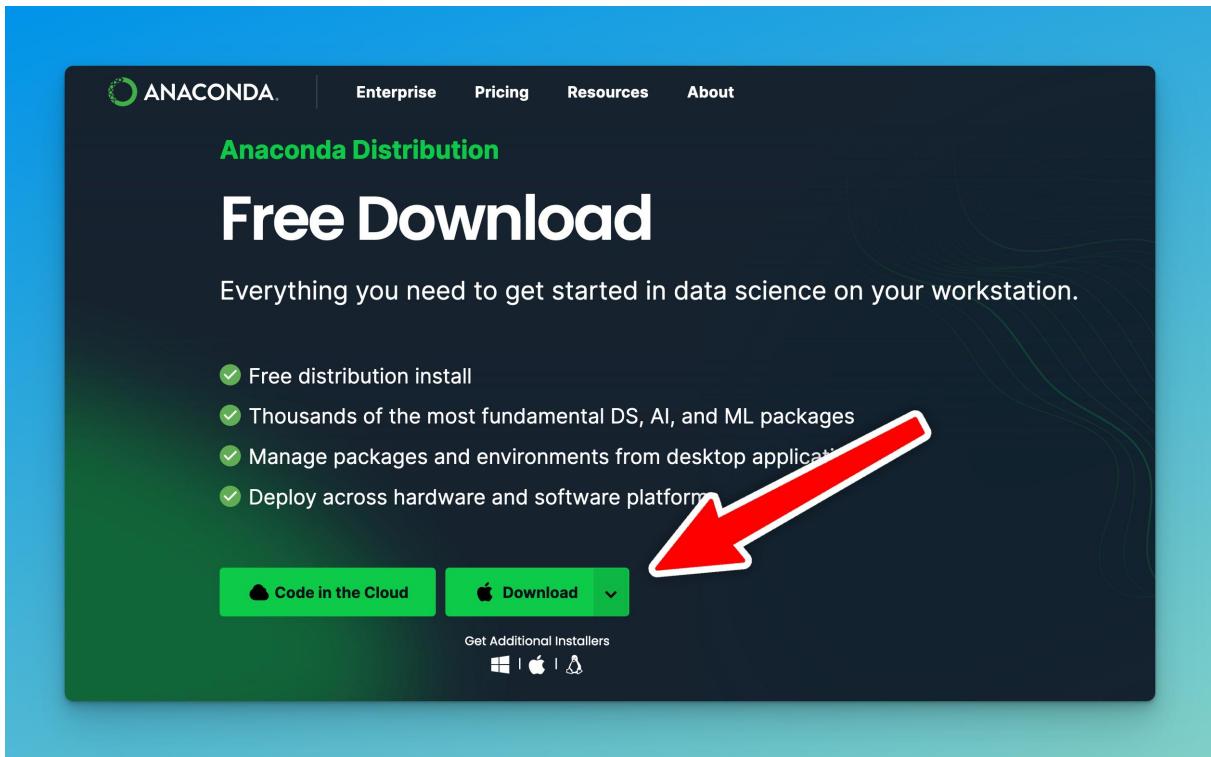


Figure 0.1: The Anaconda site

When you Download it, Anaconda may try to get you to sign up for an account. You don't have to do this, just download the package.

Then install Anaconda.

Once you have Anaconda installed, open up Anaconda navigator and launch Spyder. You may see a note about updating to a new version of Spyder and the terminal commands required to do so. You can ignore these too.

Then go to View → Window layouts and click on Horizontal split.

Make sure pane selected on the bottom right side is 'IPython console'

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

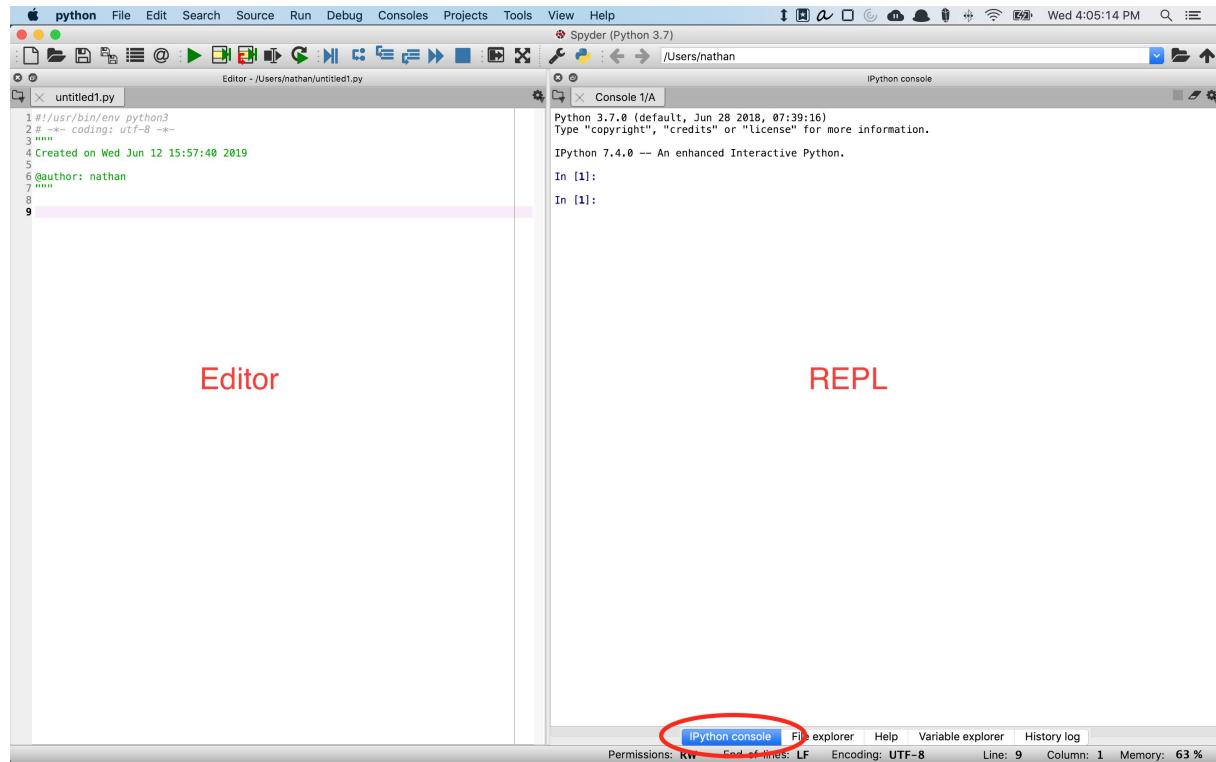


Figure 0.2: Editor and REPL in Spyder

Editor

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.

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

Console (REPL)

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

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

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

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

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

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

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

And then later:

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

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

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

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

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

Try:

```
In [2]: x = 9/0
...
ZeroDivisionError: division by zero
```

Since dividing by 0 is against the laws of math¹, Python won’t let you do it and will throw (raise) an error. No big deal – your computer didn’t crash and your data is still there. If you type `x` in the REPL again you will see it’s still 1.

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

¹See <https://www.math.toronto.edu/mathnet/questionCorner/nineoverzero.html>

Using Spyder

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.

control + shift + e moves you to the editor (e.g. if you're in the REPL). On a Mac, it's command + shift + e.

control + shift + i moves you to the REPL (e.g. if you're in the editor). On a Mac, it's command + shift + i.

Appendix B: ini files

As we work through these projects, we'll need to access various, sensitive configuration information, including your `LICENSE_KEY` for the API and authentication data for your Yahoo and ESPN leagues.

A file with an `.ini` extension (for “initialization”) is an easy way to keep track of that data. It's basically a series of key value pairs separated by headings. For example say we have a config file named `my_config.ini`:

```
[section1]
MY_SENSITIVE_DATA = this is top secret
PASSWORD = 12346

[section2]
AUTH_CODE = abcxyz
```

Headings (`section1`) need to be surrounded in brackets. Note the data to the right of the equals sign (`this is top secret`) is just text. It doesn't need to in quotation marks.

Then in Python:

```
In [1]:
from configparser import ConfigParser

config = ConfigParser(interpolation=None)
config.read('my_config.ini')

In [2]: config['section1']['MY_SENSITIVE_DATA']
Out[2]: 'this is top secret'

In [3]: config['section2']['AUTH_CODE']
Out[3]: 'abcxyz'
```

Note even purely numeric data comes out formatted as strings:

```
In [4]: config['section1']['PASSWORD']
Out[4]: '12346'
```

So if we wanted use them as numbers we'd have to convert them.

```
In [5]: int(config['section1']['PASSWORD'])
Out[5]: 12346
```

Appendix C: Probability + Fantasy Football

Here's a quick set of numbers –

17... 24... 8... 4... 13... 12... 0... 15... 14... 13... 11... 7... 0... 8... 17... 23... 2... 15... 9... 4... 28... 5...
9... 5... 17...

These are 25 randomly selected weekly RB performances from 2019. The first is Todd Gurley's 17 point performance in week 16, the second is Derrick Henry's 24 point performance week 14, etc.

Ok. Now let's arrange these from smallest (0) to largest (28), marking each with an **x** and stacking **x**'s when a score shows up multiple times. Make sense? That gives us something like this:

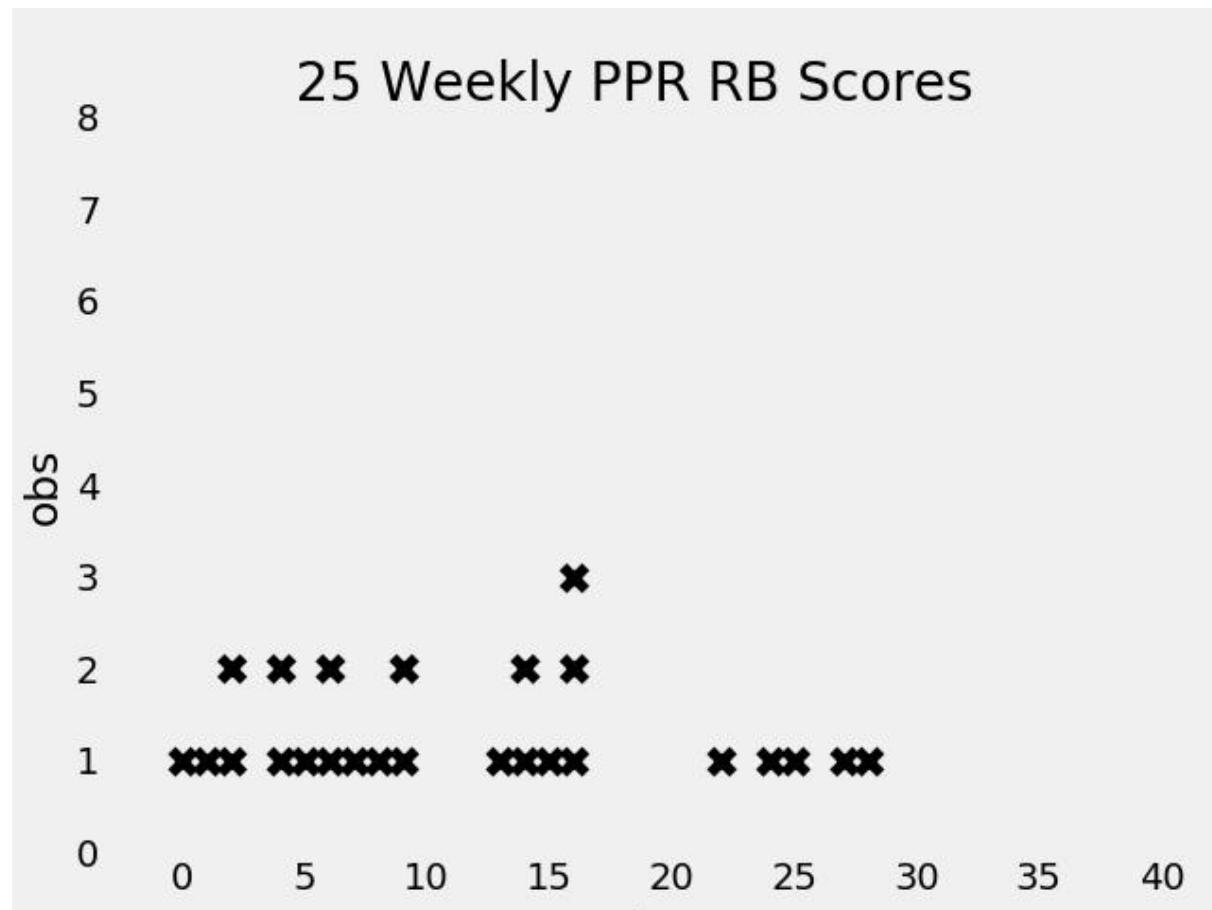


Figure 0.1: 25 random RB performances in 2019

Interesting, but why should we limit ourselves to just 25 games? Let's see what it looks like when we plot ALL the RB's over ALL their games:

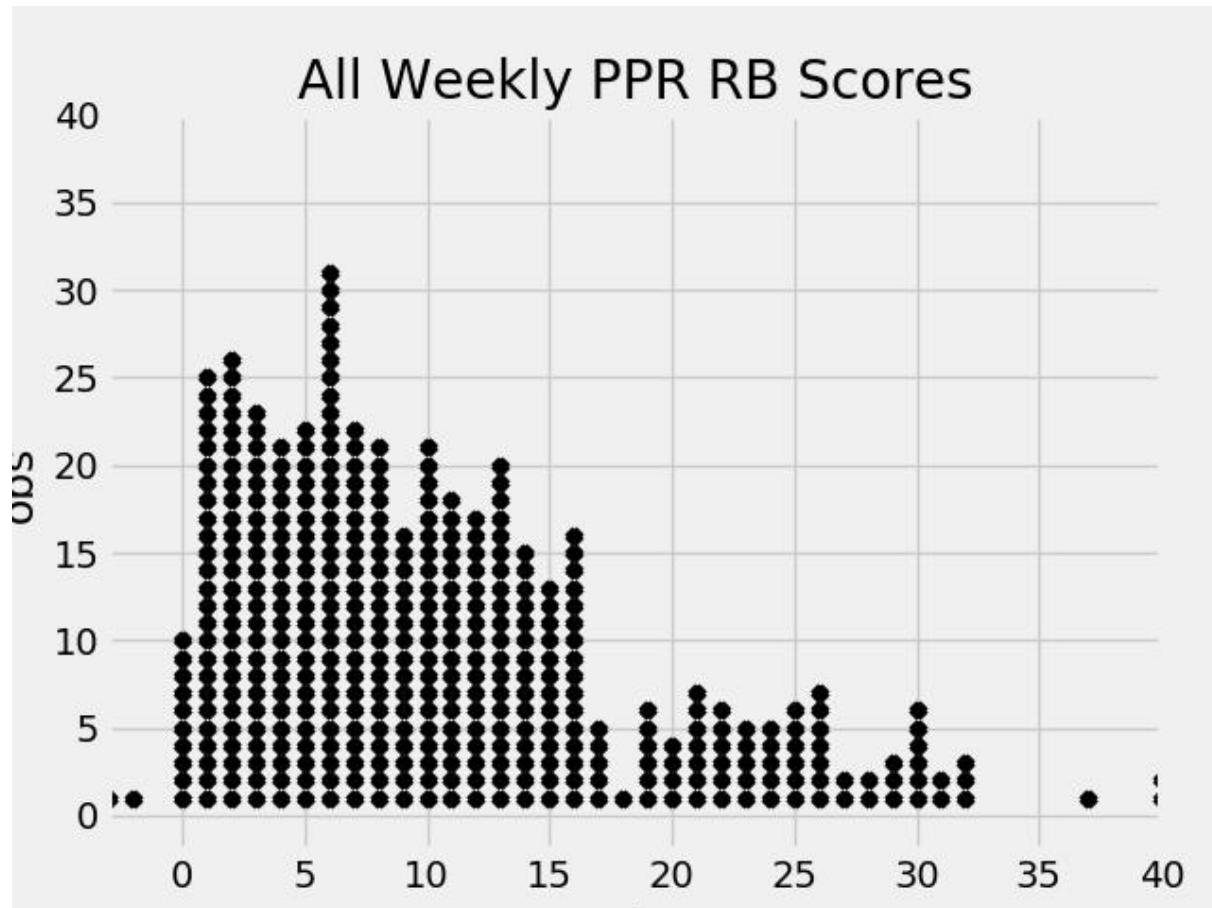


Figure 0.2: Stacked X's for all RB performances in 2019

This is the first key to thinking probabilistically about fantasy football:

For any given week, when you start a player you're picking out one of these little x's at random.

Each **x** is equally likely to get picked. Each score, however, is not. There are a lot more **x**'s between 0-10 points than there are between 20 and 30.

Distributions: Smoothed Out X's

Now let's switch from stacked x's to curves, which are easier to deal with mathematically.

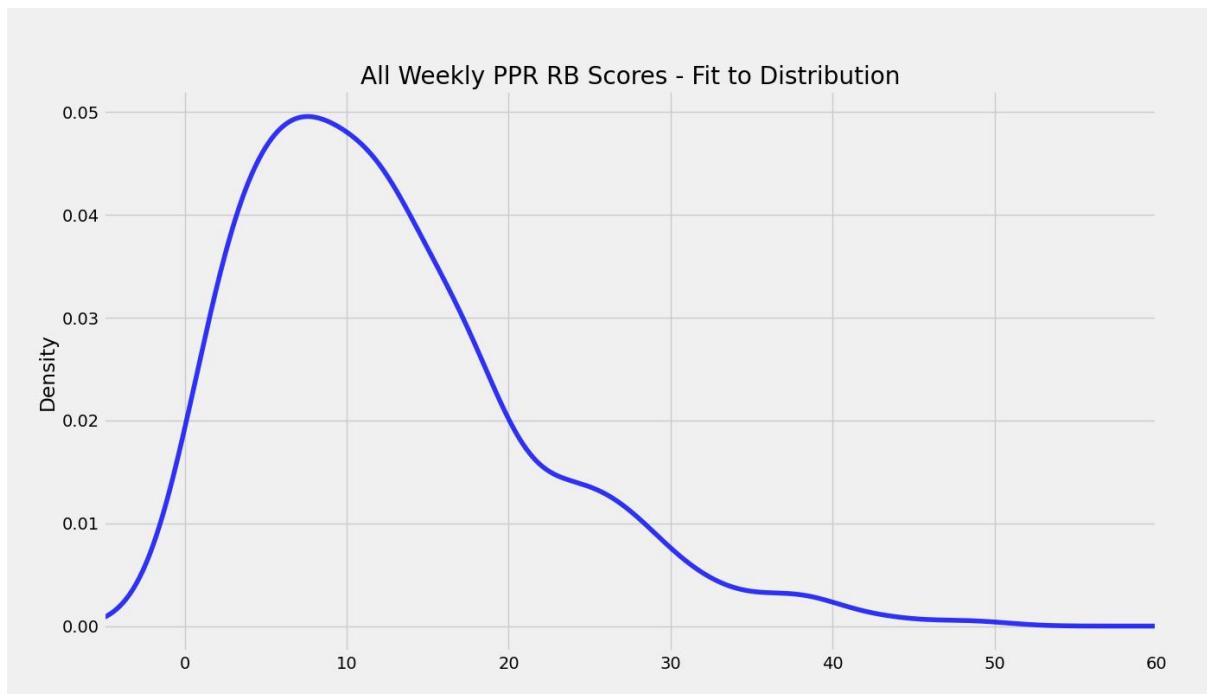


Figure 0.3: All RB performances in 2019 - Curves

The curve above is called a *distribution*. For our purposes, they're a way to smooth things out. We're replacing the stacked x's with a fuzzier, smoothed out curve. You can continue to imagine the area under these curves as filled up with little, stacked up x's if you want.

The horizontal axis on this curve still represents fantasy points. The vertical axis is less intuitive, but it ensures the area under the curve equals 1. Don't worry about it this if it's confusing.

So you're heading into the Monday night game up by 10 points. Your team is done and the guy you're playing only has one RB left — how likely is it you come out with a win?

The answer — assuming your opponent is starting a random RB — is about 50/50. Half of the area under the distribution (half of the little x's in Figure 2) is at 11 points or higher, half is below.

That's if your opponent is starting a random running back. What if he's starting Ezekiel Elliot?

Zeke is not some random RB. His score is, on average, higher than average, which means his distribution is further to the right. Something like the red one below.

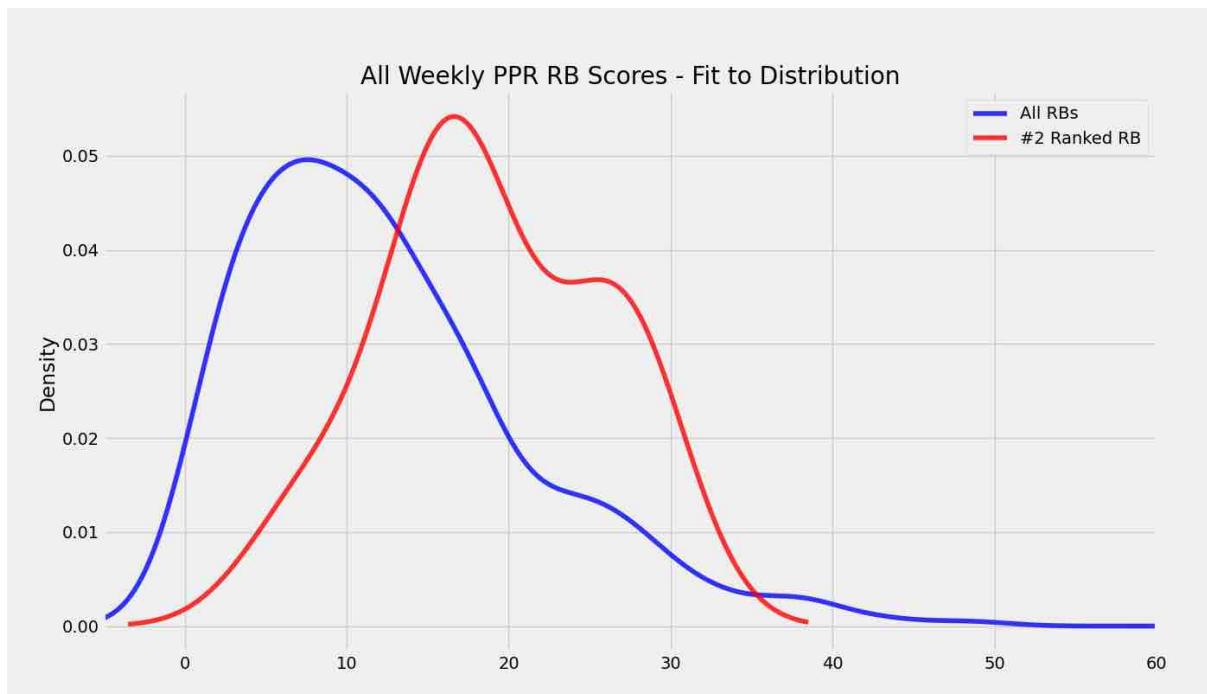


Figure 0.4: All RBs plus Zeke

In that case, Zeke will score more than 10 points 90% of the time and things aren't looking good for Monday night.

This is important: **the shape of probability distribution changes for any given player and week.**

A lot goes into these distributions: offensive line, quality of the defense, number of touches, whether the RB is in a committee, etc.

Your job as a fantasy player: assemble a lineup of guys with distributions far to the right.

This maximizes your chances of winning. Does this mean you'll always win or start the right guy?

Of course not. Even above, a draw from a random RB's distribution (the blue line) will be higher than a draw from Zeke's about 20% of the time. But it will put you in the best position to maximize your odds.

The key is changing how you view player-performances. They're not taken out of thin air, or something to be rationalized and fit into a narrative (I KNEW this [guy who performed poorly] was bad, never again!). That's a recipe for frustration and chasing points.

Instead, view performances as random draws from some range of outcomes. This captures the randomness inherent in Fantasy Football, is a good model for what's actually happening behind the scenes, and is a way to stay sane.

Appendix D: Technical Review

If you work through all these projects you'll see common themes and techniques. It's worth covering these a bit now.

Neither this nor Learn to Code with Fantasy Football is going to be able to cover everything you'll ever want to do, but Python, Pandas and data science in general very much follows the 80/20 rule: you'll be able to do 80% of the things you want to do by mastering 20% of the language's features.

However, there are just some concepts that show up so often in these projects that it's going to be way easier to make sure you understand them going in.

Now lets dive into a few more Python and Pandas specific technical things that are worth reviewing before we dive in.

Having just coded up all of these projects and detailed walk-throughs — as well as the code to create and run the models powering the API — this section contains a few of what I would say are common themes.

They are:

- comprehensions
- f strings
- Pandas functions and the 0/1 axis
- stacking/unstacking

All of these where covered in LTCWFF, and shouldn't really be anything new.

Note: this is definitely NOT to say these are only Python and Pandas concepts we'll use. There are definitely others, and if you're feeling unsure about your Python or Pandas skills generally you should revisit chapters two and three in the book.

But the above concepts do come up relatively more frequently in the data we'll be working with. These projects will be much easier for you're comfortable with these concepts.

Comprehensions

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.

It is 100% worth mastering basic, single level list comprehensions. Dictionary comprehensions show up a bit less often, but you should learn these too, along with the `.items()` syntax.

The two computer science concepts that comprehensions make easy:

Map

Change every item in a collection (list or a dict) by running it through a function.

Filter

Keep (flipside: drop) items in a collection based on some criteria.

Comprehensions let you do both (at once) in one line of a code.

I'm stressing them because they show up frequently and are unique to Python, not because they're difficult to understand or you should be intimidated by them.

In practice, comprehensions are an easy way to turn one list into another list (either by changing — mapping, or dropping — filtering, some of the items in it).

See chapter two of LTCWFF for more on list and dictionary comprehensions.

f-strings

We went over f-strings in LTCWFF, but they seem to be getting more popular and show up a lot in these projects too.

f-strings are just a way to put some Python data (e.g. data in a variable) into a string.

```
In [23]: team_name = 'Fresh Prince of Helaire'  
In [24]: print(f'Your team is {team_name} - nice!')  
Out[24]: 'Your team is Fresh Prince of Helaire - nice!'
```

You can do normal Python inside the curly brackets:

```
In [27]: f'You scored {100 + 34} points!'
Out[27]: 'You scored 134 points!'
```

If you want to use f-strings inside of multi-line strings (which are surrounded by three quotes, i.e. ““““) you need to use the `textwrap.dedent` function.

This is the only place I've ever run across `dedent`, so I'd just try to accept this as something you have to do and with multiline f-strings and leave it at that.

```
from textwrap import dedent

my_db_table = 'players'
my_sql_query = dedent(
    f"""
        SELECT *
        FROM {my_db_table}
    """)
```

See the relevant section in LTCWFF (and the practice problems on it) for more.

Pandas Functions and the `axis` argument

The concept of built-in functions and how you can apply them to different `axis` is fundamental in Pandas. It shows up in most programs.

But it's even more prevalent here because of the nature of the data we're working with. Therefore we'll do a quick review.

Note: there are plenty of other basic Pandas concepts (e.g. `indexing`, `loc`, `pd.concat`) that also show up in these projects. But these concepts show up almost every Pandas programs, so I'm not going to do a separate review here. Review the Pandas chapter of LTCWFF as needed.

Pandas includes a bunch of functions that calculate descriptive statistics on your data, e.g. `mean`, `sum`, etc.

You can call these functions on either or `columns axis=0` (the default) or your rows (`axis=1`).

For example, we'll be working with simulation data here, which looks like this:

	cam-newton	baker-mayfield	josh-allen
0	9.489878	14.058540	0.348651
1	5.519853	11.624132	2.822340
2	14.784496	15.652178	21.763491
3	10.526069	23.152408	21.643099
4	4.446957	10.057039	6.017969
5	6.136221	14.870359	19.187099
6	4.086974	21.521453	15.960591
7	29.266968	15.805350	1.298063
8	25.763714	16.908681	16.492491
9	1.014356	10.493941	8.459999

Every column is a player. Every row is a (correlated) simulation.

Let's say we want to take the mean of this data. When `axis=0`, we'll get the mean of each column, and our data will be three numbers:

```
In [22]: sims[['cam-newton', 'baker-mayfield', 'josh-allen']].mean(axis=0)
Out[22]:
cam-newton      11.103549
baker-mayfield  15.414408
josh-allen     11.399379
```

`axis=0` is the default, so these two lines are exactly same:

```
sims[['cam-newton', 'baker-mayfield', 'josh-allen']].mean(axis=0)
sims[['cam-newton', 'baker-mayfield', 'josh-allen']].mean()
```

That's the average of each column. We can also get the average for each simulation, i.e. the average by row. To do it by row like that we need to make `axis=1`.

In this case we'll have a number for every row, i.e. a new column of data:

```
In [26]: (sims[['cam-newton', 'baker-mayfield', 'josh-allen']]
         .head(10)
         .mean(axis=1))
Out[26]:
0      7.965690
1      6.655442
2     17.400055
3     18.440525
4      6.840655
5     13.397893
6     13.856339
7     15.456794
8     19.721629
9      6.656099
```

It works the same with all functions, not just `mean`.

```
In [27]: sims[['cam-newton', 'baker-mayfield', 'josh-allen']].head(10).max()
()
Out[27]:
cam-newton      29.266968
baker-mayfield  23.152408
josh-allen     21.763491
```

```
In [28]: (sims[['cam-newton', 'baker-mayfield', 'josh-allen']]
         .head(10)
         .max(axis=1))
Out[28]:
0    14.058540
1    11.624132
2    21.763491
3    23.152408
4    10.057039
5    19.187099
6    21.521453
7    29.266968
8    25.763714
9    10.493941
```

stack/unstack

Calling stack (unstack) is a way to transform your data from wide to long (stack) or long to wide (unstack).

Here we'll mostly be stacking (going from wide to long) to we'll focus on that.

We covered this in LTCWFF, but I said it didn't come up that often and you could skip it if you want.

It comes up often here, mostly because the simulations we're working with are in "wide" (every in their own column) form. Like this:

	cam-newton	baker-mayfield	josh-allen
0	9.489878	14.058540	0.348651
1	5.519853	11.624132	2.822340
2	14.784496	15.652178	21.763491
3	10.526069	23.152408	21.643099
4	4.446957	10.057039	6.017969

But seaborn, the plotting library we'll be working with, usually requires data to be in long form. Instead of the above it needs to be:

	sim	player	points
0	0	cam-newton	9.489878
1	0	baker-mayfield	14.058540
2	0	josh-allen	0.348651
3	1	cam-newton	5.519853
4	1	baker-mayfield	11.624132
5	1	josh-allen	2.822340
6	2	cam-newton	14.784496
7	2	baker-mayfield	15.652178
8	2	josh-allen	21.763491
9	3	cam-newton	10.526069
10	3	baker-mayfield	23.152408
11	3	josh-allen	21.643099
12	4	cam-newton	4.446957
13	4	baker-mayfield	10.057039
14	4	josh-allen	6.017969

Note these two tables have exactly the same information (sim, player, points), they're just formatted differently.

The way you go from the first to the second is using the `stack` function.

After calling `stack`, the new DataFrame always has a *multindex*. For every number of the original index (sim 0, 1, ... here) we now have a line for each column (cam-newton, baker-mayfield, josh-allen).

In [44]:	qb5 = sims[['cam-newton', 'baker-mayfield', 'josh-allen']].head(5)																																													
In [45]:	qb5.stack()																																													
Out[45]:	<table border="1"> <tbody> <tr> <td>0</td> <td>cam-newton</td> <td>9.489878</td> </tr> <tr> <td></td> <td>baker-mayfield</td> <td>14.058540</td> </tr> <tr> <td></td> <td>josh-allen</td> <td>0.348651</td> </tr> <tr> <td>1</td> <td>cam-newton</td> <td>5.519853</td> </tr> <tr> <td></td> <td>baker-mayfield</td> <td>11.624132</td> </tr> <tr> <td></td> <td>josh-allen</td> <td>2.822340</td> </tr> <tr> <td>2</td> <td>cam-newton</td> <td>14.784496</td> </tr> <tr> <td></td> <td>baker-mayfield</td> <td>15.652178</td> </tr> <tr> <td></td> <td>josh-allen</td> <td>21.763491</td> </tr> <tr> <td>3</td> <td>cam-newton</td> <td>10.526069</td> </tr> <tr> <td></td> <td>baker-mayfield</td> <td>23.152408</td> </tr> <tr> <td></td> <td>josh-allen</td> <td>21.643099</td> </tr> <tr> <td>4</td> <td>cam-newton</td> <td>4.446957</td> </tr> <tr> <td></td> <td>baker-mayfield</td> <td>10.057039</td> </tr> <tr> <td></td> <td>josh-allen</td> <td>6.017969</td> </tr> </tbody> </table>	0	cam-newton	9.489878		baker-mayfield	14.058540		josh-allen	0.348651	1	cam-newton	5.519853		baker-mayfield	11.624132		josh-allen	2.822340	2	cam-newton	14.784496		baker-mayfield	15.652178		josh-allen	21.763491	3	cam-newton	10.526069		baker-mayfield	23.152408		josh-allen	21.643099	4	cam-newton	4.446957		baker-mayfield	10.057039		josh-allen	6.017969
0	cam-newton	9.489878																																												
	baker-mayfield	14.058540																																												
	josh-allen	0.348651																																												
1	cam-newton	5.519853																																												
	baker-mayfield	11.624132																																												
	josh-allen	2.822340																																												
2	cam-newton	14.784496																																												
	baker-mayfield	15.652178																																												
	josh-allen	21.763491																																												
3	cam-newton	10.526069																																												
	baker-mayfield	23.152408																																												
	josh-allen	21.643099																																												
4	cam-newton	4.446957																																												
	baker-mayfield	10.057039																																												
	josh-allen	6.017969																																												

Technically (from Pandas' perspective), this is one column of data (points) with a multi-level index (sim and player).

I find multi-indexes confusing though ¹, so the first thing I always do with them is turn them into regular columns by immediately calling `reset_index`

```
In [44]: qb5.stack().reset_index()
Out[44]:
   level_0      level_1      0
0         0    cam-newton  9.489878
1         0  baker-mayfield 14.058540
2         0    josh-allen  0.348651
3         1    cam-newton  5.519853
4         1  baker-mayfield 11.624132
5         1    josh-allen  2.822340
6         2    cam-newton 14.784496
7         2  baker-mayfield 15.652178
8         2    josh-allen 21.763491
9         3    cam-newton 10.526069
10        3  baker-mayfield 23.152408
11        3    josh-allen 21.643099
12        4    cam-newton  4.446957
13        4  baker-mayfield 10.057039
14        4    josh-allen  6.017969
```

Note, resetting the index messes up the column names too. If you think about it, this makes sense. Pandas had no way of knowing the columns in our wide data were players. But that means we have to rename them:

¹This is a good example of what I was talking about earlier, how at different points in your coding journey, certain things will be beyond your skill set, but if you're learning and progressing you'll eventually pick them up. Multi-indexes are that way for me right now. Maybe eventually I'll grow to appreciate them and look back on this code, "Oh I must have written this in 2020, while I was on quarantine for a global pandemic but before I really got the hang of multi-indexes"

```
In [45]: qb5_long = qb5.stack().reset_index()
In [46]: qb5_long.columns = ['sim', 'player', 'points']

In [47]: qb5_long
Out[47]:
   sim      player    points
0     0  cam-newton  9.489878
1     0  baker-mayfield  14.058540
2     0  josh-allen  0.348651
3     1  cam-newton  5.519853
4     1  baker-mayfield  11.624132
5     1  josh-allen  2.822340
6     2  cam-newton  14.784496
7     2  baker-mayfield  15.652178
8     2  josh-allen  21.763491
9     3  cam-newton  10.526069
10    3  baker-mayfield  23.152408
11    3  josh-allen  21.643099
12    4  cam-newton  4.446957
13    4  baker-mayfield  10.057039
14    4  josh-allen  6.017969
```

This stack → reset_index → rename columns pattern shows up a lot in these projects, almost always when we're getting our data ready for seaborn to plot.

Review

In this section, we touched a few technical Python and Pandas topics that either are absolutely critical to understanding the code we'll be going over (comprehensions, f-strings) or show up relatively more often here than they do in other Pandas projects (axis, stacking and unstacking).

Again, there are plenty of other basic Pandas concepts ([indexing](#), [loc](#), [pd.concat](#)) that show up in these projects and which we did NOT review. But these concepts show up almost every Pandas program, and this walk-through assumes you're familiar with the basics. Review the Pandas chapter of LTCWFF as needed.

Appendix E: Fantasy Math Web Access

The web interface to these simulations is available at <https://app.fantasymath.com>. These are the instructions for accessing it.

License Key and Email

To get started you'll need your Fantasy Math license key. You can find link you received by email to download this guide:

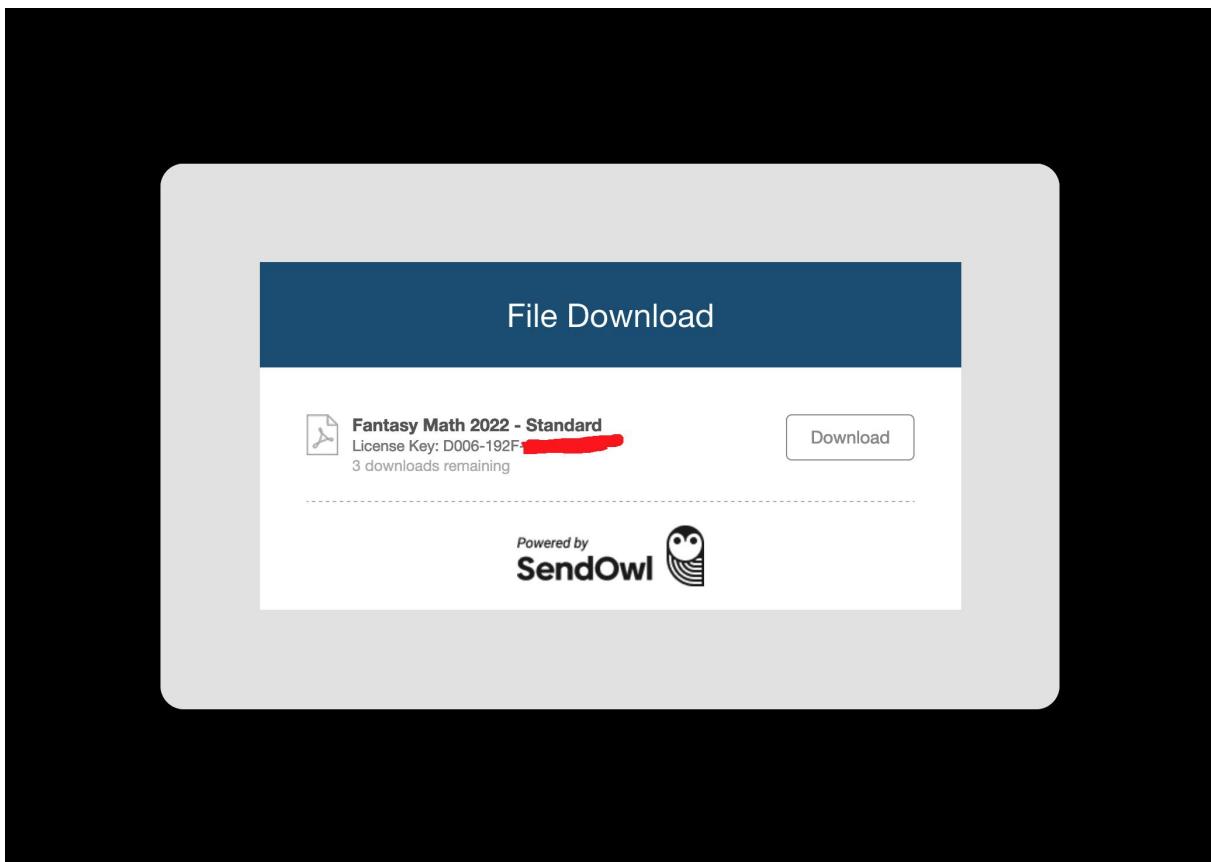


Figure 0.1: Sendowl License Key Screenshot

When you have that head over to:

<https://app.fantasymath.com/new-user>

Enter your email and license key then click submit.

Congratulations! You're in. You should get an email within a few minutes with a login link. If you don't see it or have any issues email me at nate@nathanbraun.com.

Leagues

When you login to Fantasy Math for the first time you'll see this:

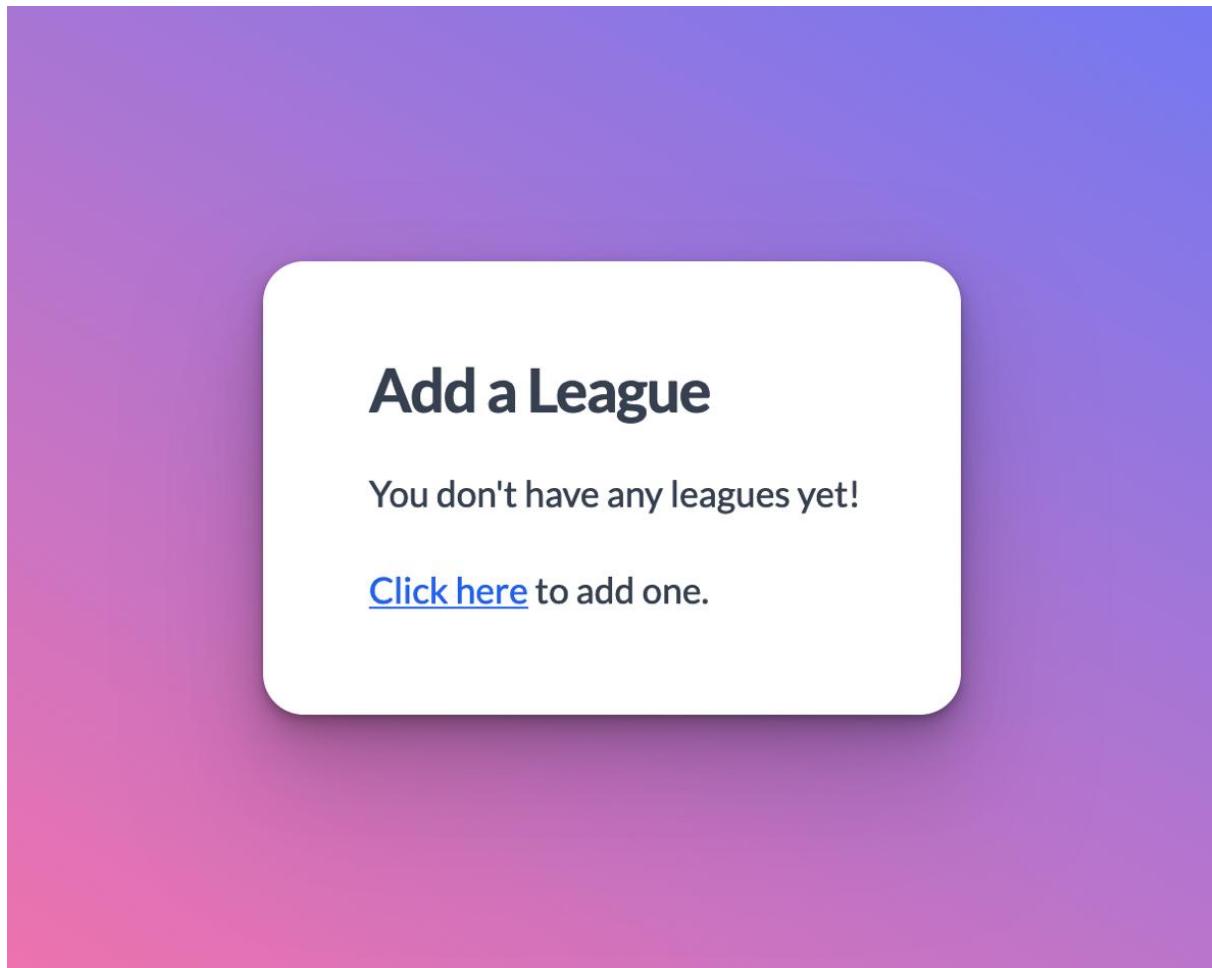


Figure 0.2: Add a League

Fantasy Math sorts your results and scoring settings by league. So let's add one, say "Family

League”.

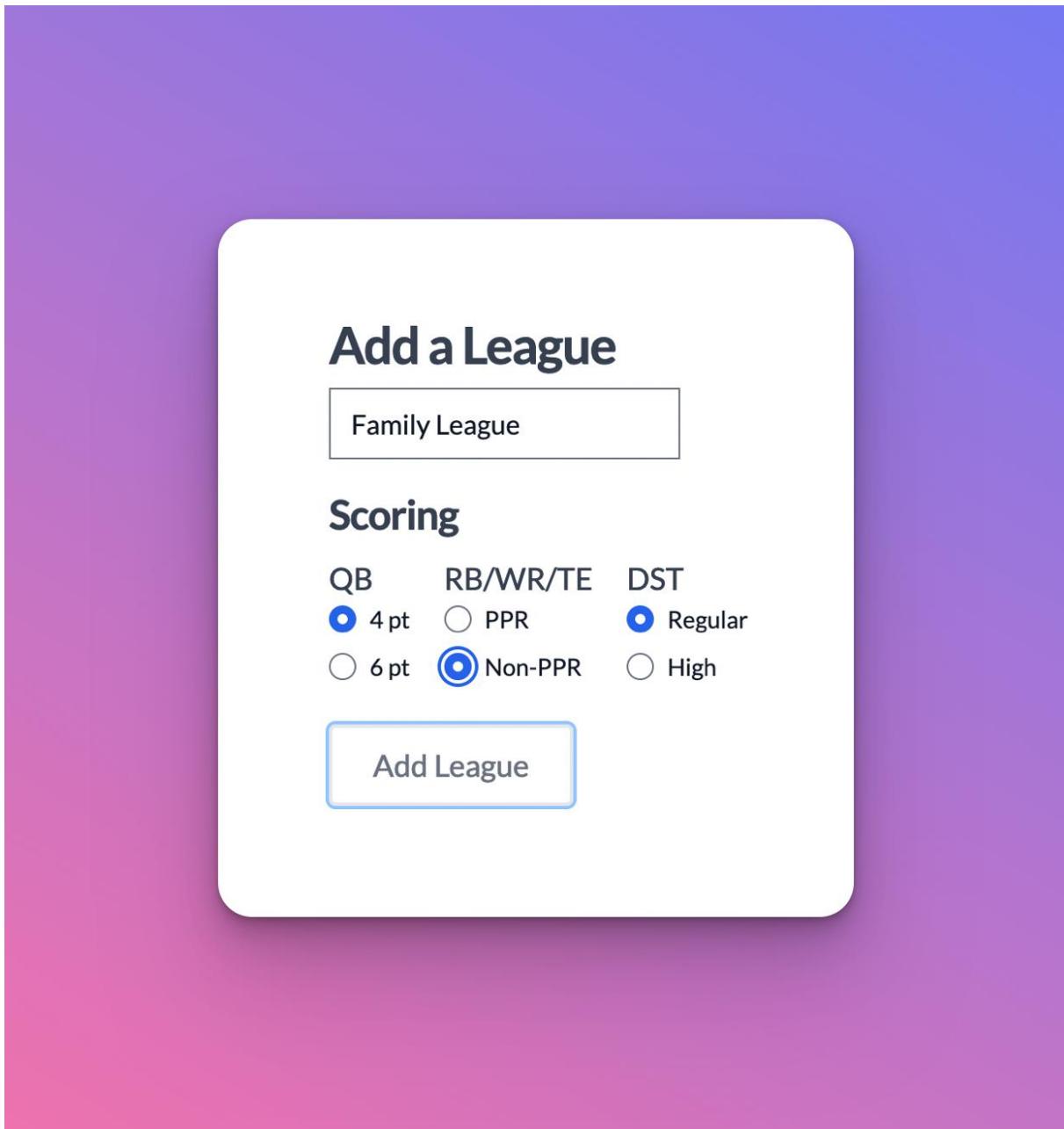


Figure 0.3: Family League

When you add it. It'll bring you back to the main page. You can click the + to add another league if you want. I'll add another one called "Work League".

Analyzing a Matchup

Once you've added at least league, go back to the main <https://app.fantasymath.com> page. You'll see the league info at the top with your active league outlined.

Let's enter in my week 1 matchup. It looks like this:

The screenshot shows the Fantasy Math app interface for analyzing a Week 1 matchup. At the top, there is a navigation bar with tabs for "Family League" and "Work League" (which is selected), followed by a "+" button. Below the navigation bar, the section title "Starting Lineups" is displayed. Under "Starting Lineups", there are two sections: "Team 1" and "Team 2". Each section contains a list of players with their positions and team abbreviations. For Team 1, the players listed are: QB Josh Allen (BUF), RB James Conner (ARI), RB Clyde Edwards Helaire (KC), WR Cooper Kupp (LA), WR Ceedee Lamb (DAL), TE Dallas Goedert (PHI), and DST KC Chiefs. For Team 2, the players listed are: QB Tua Tagovailoa (MIA), RB Miles Sanders (PHI), RB Jk Dobbins (BAL), WR Justin Jefferson (MIN), WR Mike Evans (TB), TE Travis Kelce (KC), and DST CAR Panthers. Below the lineups, there is a section titled "WDIS (Optional)" with a dropdown menu. At the bottom of the form, there is a section titled "Existing Points" with dropdown menus for "Team 1" and "Team 2", and a green "Submit" button.

Team	Player	Position	Team Abbreviation
Team 1	QB Josh Allen	QB	BUF
	RB James Conner	RB	ARI
	RB Clyde Edwards Helaire	RB	KC
	WR Cooper Kupp	WR	LA
	WR Ceedee Lamb	WR	DAL
	TE Dallas Goedert	TE	PHI
	DST KC Chiefs	DST	
Team 2	QB Tua Tagovailoa	QB	MIA
	RB Miles Sanders	RB	PHI
	RB Jk Dobbins	RB	BAL
	WR Justin Jefferson	WR	MIN
	WR Mike Evans	WR	TB
	TE Travis Kelce	TE	KC
	DST CAR Panthers	DST	

Figure 0.4: Week 1

Note: the most annoying part of using Fantasy Math is typing in all the players. Luckily though, you

only have to do this once per team. Once it analyzes a matchup, it'll add a team to the History section below.

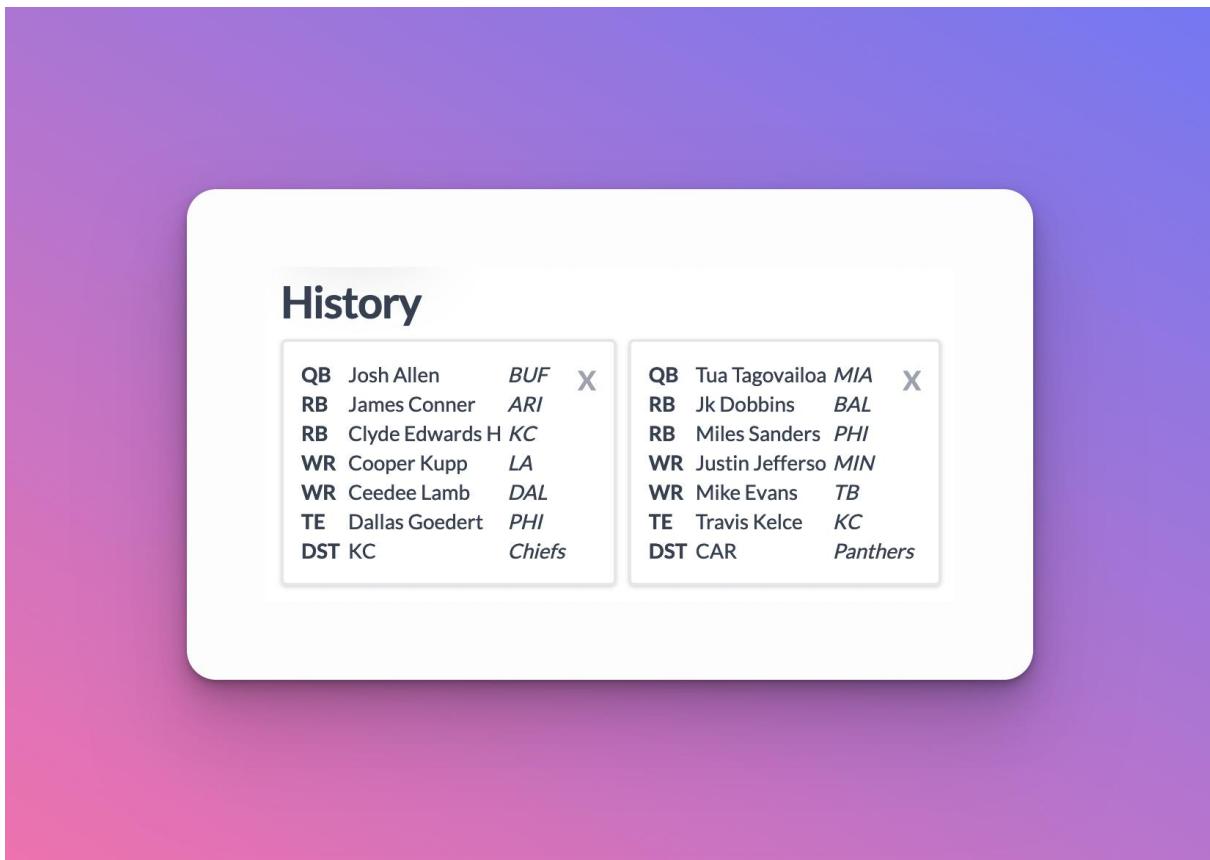


Figure 0.5: History

There you can click on a team to have it pre-fill (then edit) the selection. Note history (along with scoring settings) are saved by *league*.

Note: if it's mid-week (e.g. Friday) and a player has already played (Thursday night), leave them out of Team 1 and Team 2 and put total points in Pts 1 and Pts 2 instead.

Results

When you click submit you'll see the probability you win + projected team score distributions.

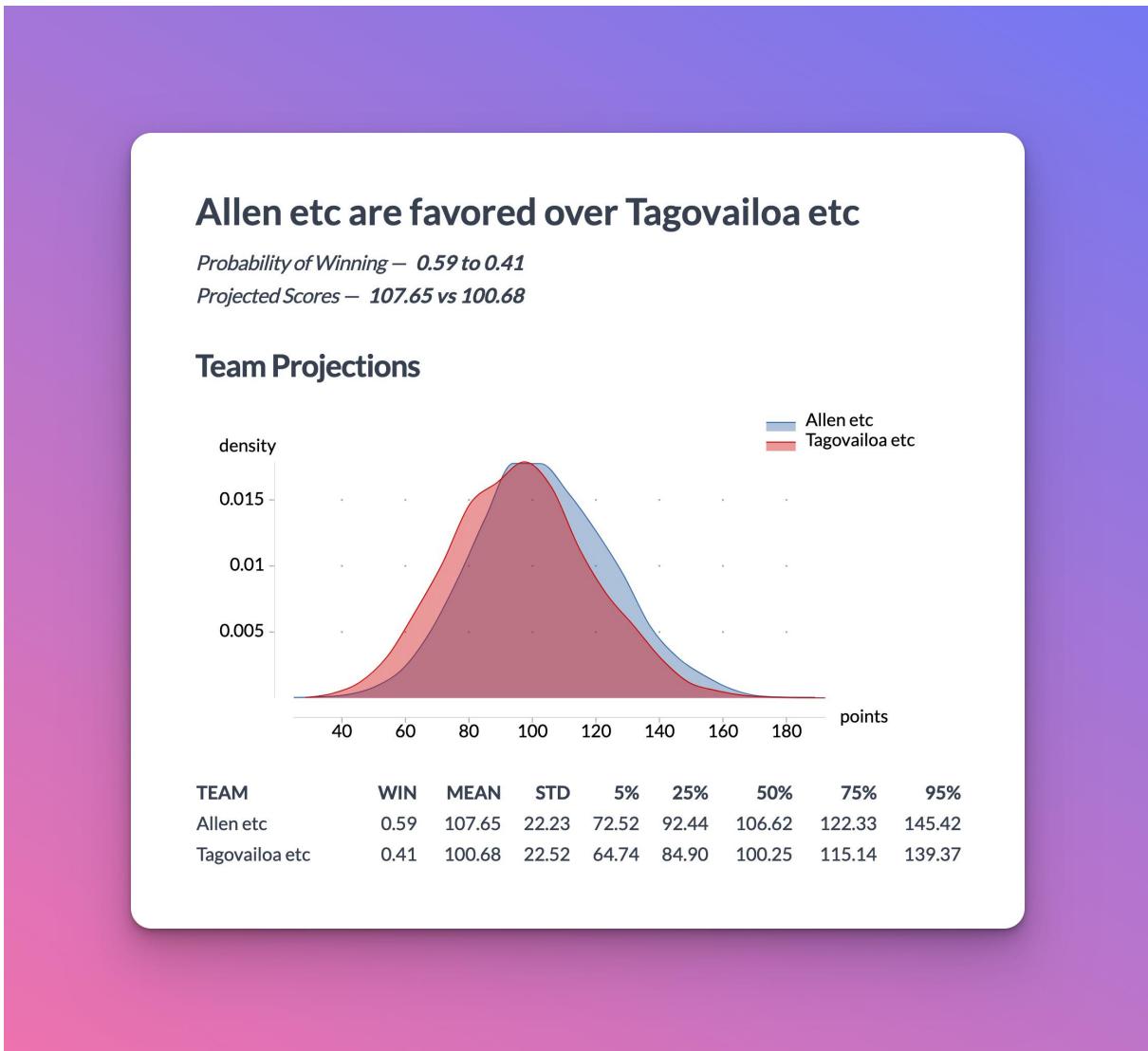
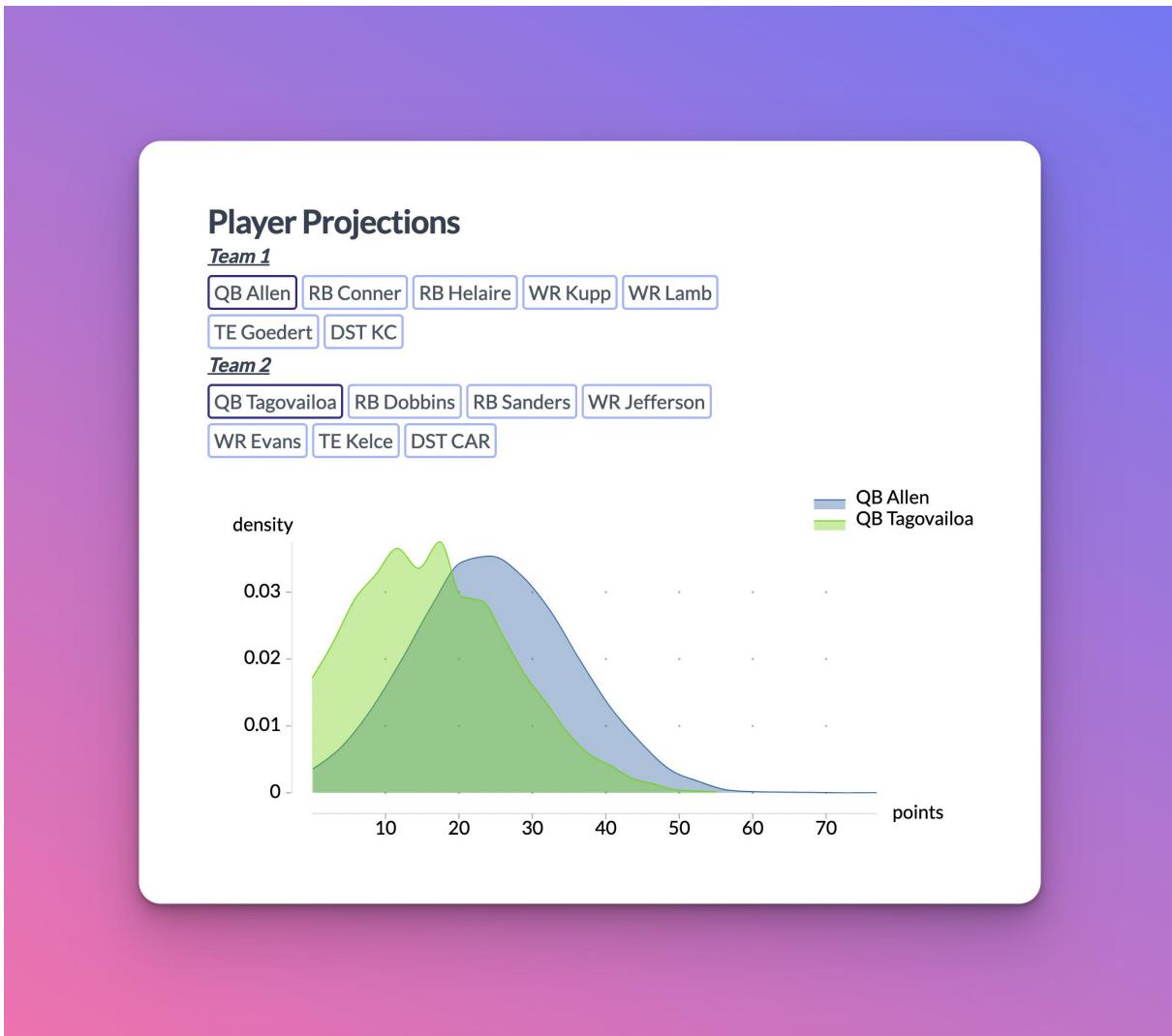


Figure 0.6: Team Projections

You'll also see distributions for each player. You can click the player names at the top to toggle each player's distribution. I find this helpful to see the strengths/weaknesses of my matchup.

**Figure 0.7:** Player Projections

Who Do I Start

Above, I'm not sure if I should start Clyde Edwards Helaire or Tony Pollard.

I can ask Fantasy Math by filling in Team 1 and Team 2 like before. Then, I can put Tony Pollard and Clyde Edwards Helaire in the **WDIS** field.

Note: one (and only one) of the players in the WDIS has to be in Team 1 or Team 2.

This is how Fantasy Math keeps track of who you're talking about.

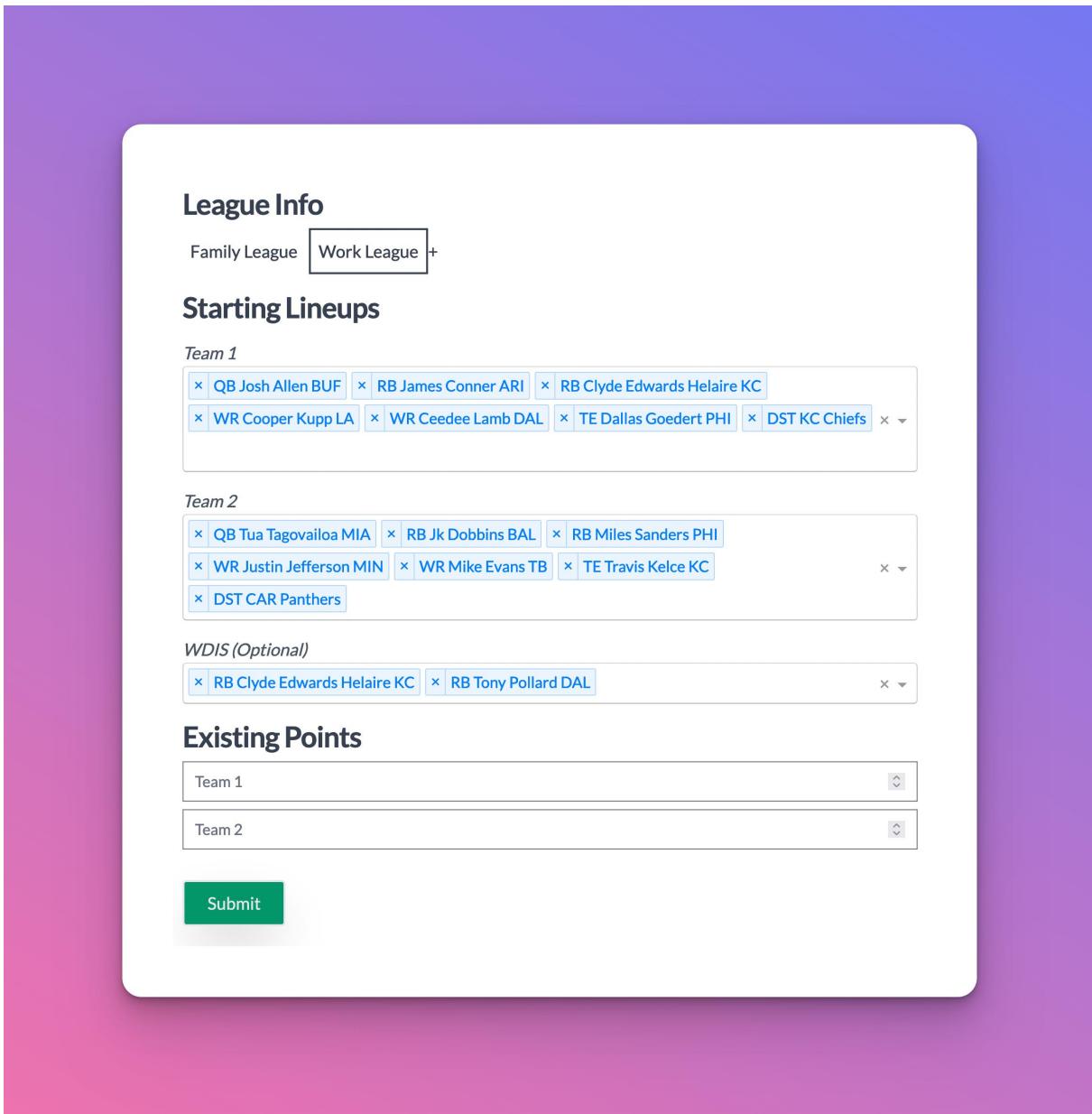


Figure 0.8: Who Do I Start

When you click submit Fantasy Math will tell you who maximizes your probability of winning. Here (Week 1, 2022) we see it's CEH:

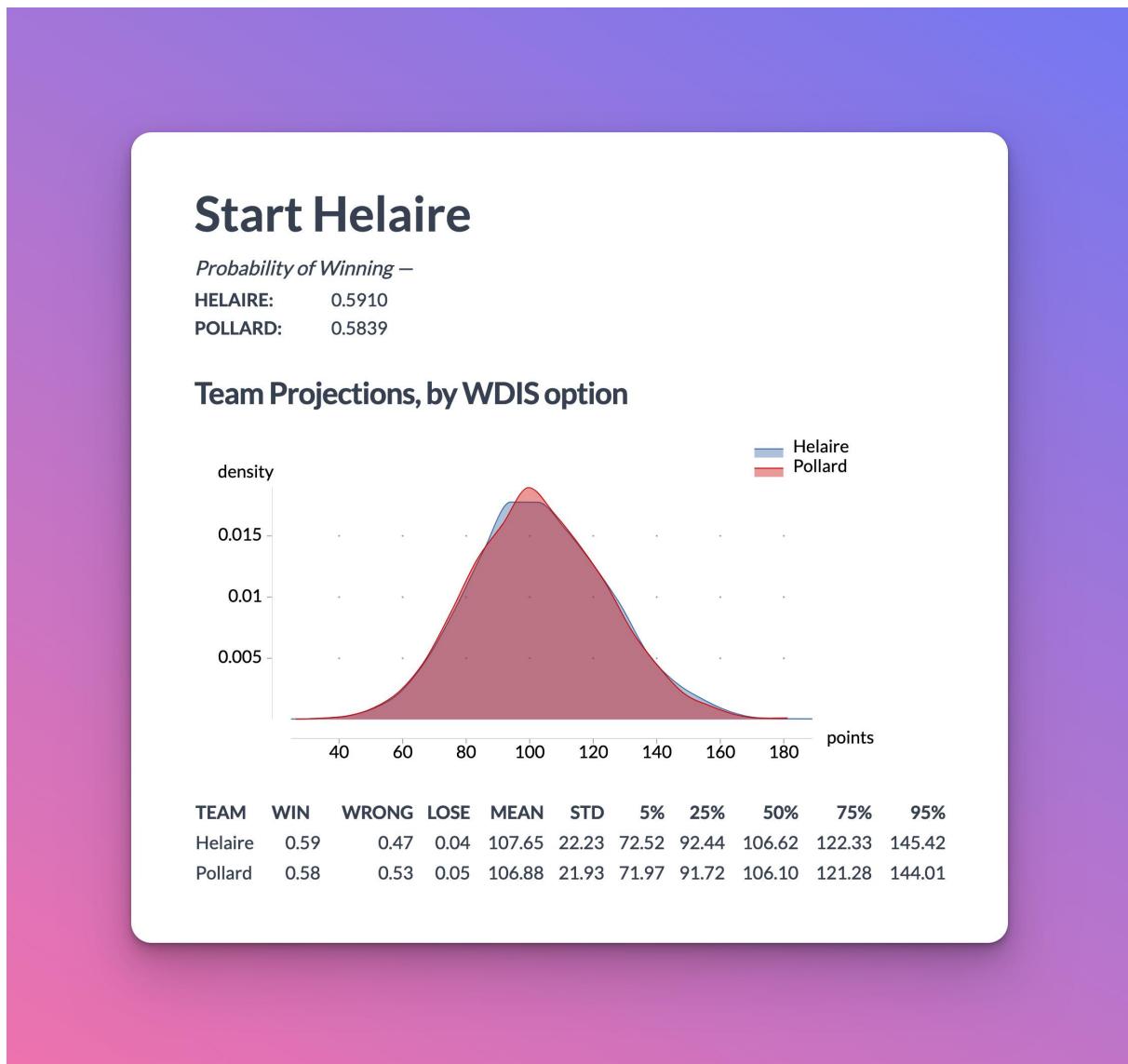


Figure 0.9: Who Do I Start Results