2022 Fantasy Football Developer Kit

v0.11.0

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 **11**
How to Read This Chapter . . . 11
Prerequisites . . . 11
Setup, Authorization, Fantasy Math API, Accessing the Simulations . . . 12
`__name == '__main__'` . . . 12
Authorization Workflow . . . 12
GraphQL . . . 13

Included Helper Functions 14
`generate_token` 14
`get_players` 14
`master_player_lookup` 14
`get_sims` 14

4. Introduction to the Data **16**
Important Note About Your Working Directory 16
Querying the Data 16
Querying sims 18
Working with Simulations 18
Correlations 21
Conditional Probabilities 23

5. Project #1: Who Do I Start **26**
WDIS API 26
Projects Connect via APIs 27
Building the WDIS Project 27
Parameters 28
Coding Up a Who Do I Start Calculator 30
Beyond WDIS 35
Plotting 47
WDIS Wrap Up 53

6. Project #2: League Integration **55**
Working with Public APIs — General Process 55
1. Authentication 55
2. Finding an endpoint 55
3. Visit endpoint in browser 56
4. Get what you need in Python 59
5. Clean things up 59
Common Outputs 59
1. Team Data 59
2 and 3. Matchup and Team Schedule Data 60
4. Roster Data 60
ESPN Integration 62
Authentication and Setup 62
Connection to ESPN in Python 62
ESPN Endpoints 64

Roster Data . . . 65
Team Data . . . 84
Schedule Info . . . 87
Wrap Up . . . 90
Fleaflicker Integration . . . 92
Roster Data . . . 92
Team Data . . . 106
Schedule Info . . . 110
Wrap Up . . . 112
Sleeper Integration . . . 113
Roster Data . . . 113
Team Data . . . 130
Schedule Info . . . 132
Wrap Up . . . 137
Yahoo Integration . . . 139
Authentication and Setup . . . 139
Yahoo Endpoints . . . 144
Roster Data . . . 145
Team Data . . . 159
Schedule Info . . . 165
Wrap Up . . . 174
Saving League Data to a Database . . . 175
Getting the Data . . . 175
Writing it to a Database . . . 176
Other League Data . . . 178
Wrap Up - League Configuration . . . 178
Auto WDIS - Integrating WDIS with your League . . . 180
Writing to a File . . . 192
Writing WDIS Output to a File . . . 193
Wrap Up . . . 197

7. Project #3: League Analyzer **198**
Loading Rosters . . . 198
Aside: How “good” were these performances? . . . 201
Updating the sims with actual scores . . . 202
Analyzing Matchups . . . 203
Analyzing Teams . . . 212

High and Low . . . 214
Aside - walking through `totals_by_team` . . . 214
Back to analyzing teams . . . 216
Aside: high/low scores and how more data → less variance . . . 218
Back to the team analysis . . . 219
Writing to a File . . . 220
Plots . . . 222

8. Project #4: Best Ball Projector **228**
Best Ball Leagues . . . 228
Walkthrough . . . 228

Appendix A: Installing Python **247**
Python . . . 247
Editor . . . 249
Console (REPL) . . . 249
Using Spyder . . . 250

Appendix B: ini files **251**

Appendix C: Probability + Fantasy Football **253**
Distributions: Smoothed Out X's . . . 255

Appendix D: Technical Review **258**
Comprehensions . . . 259
f-strings . . . 259
Pandas Functions and the axis argument . . . 260
stack/unstack . . . 262
Review . . . 265

Appendix E: Fantasy Math Web Access **266**
License Key and Email . . . 266
Leagues . . . 267
Analyzing a Matchup . . . 269
Results . . . 270
Who Do I Start . . . 272

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).

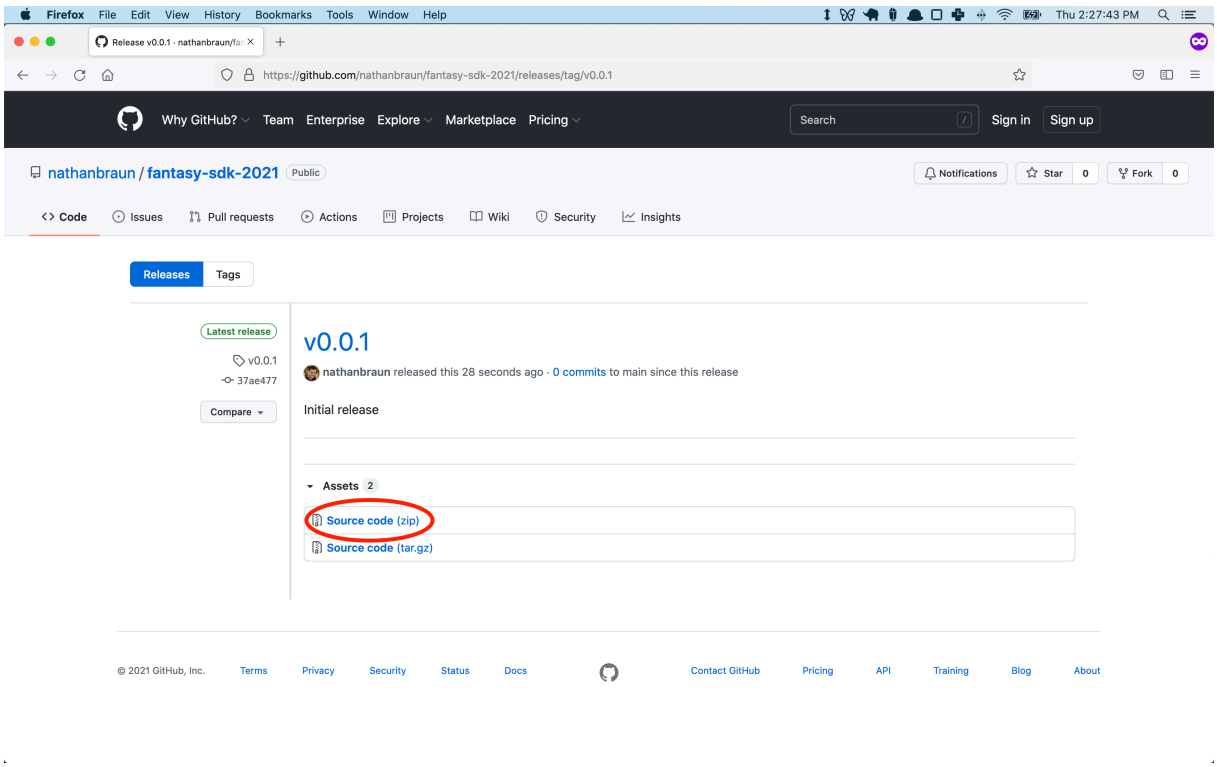

Figure 0.1: Developer Kit Files on Github

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.

(File extensions are the end part of the file name that starts with a period and gives info on file type. So .txt files for text, .xlsx for Excel, .csv for comma seperated data, etc.)

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:

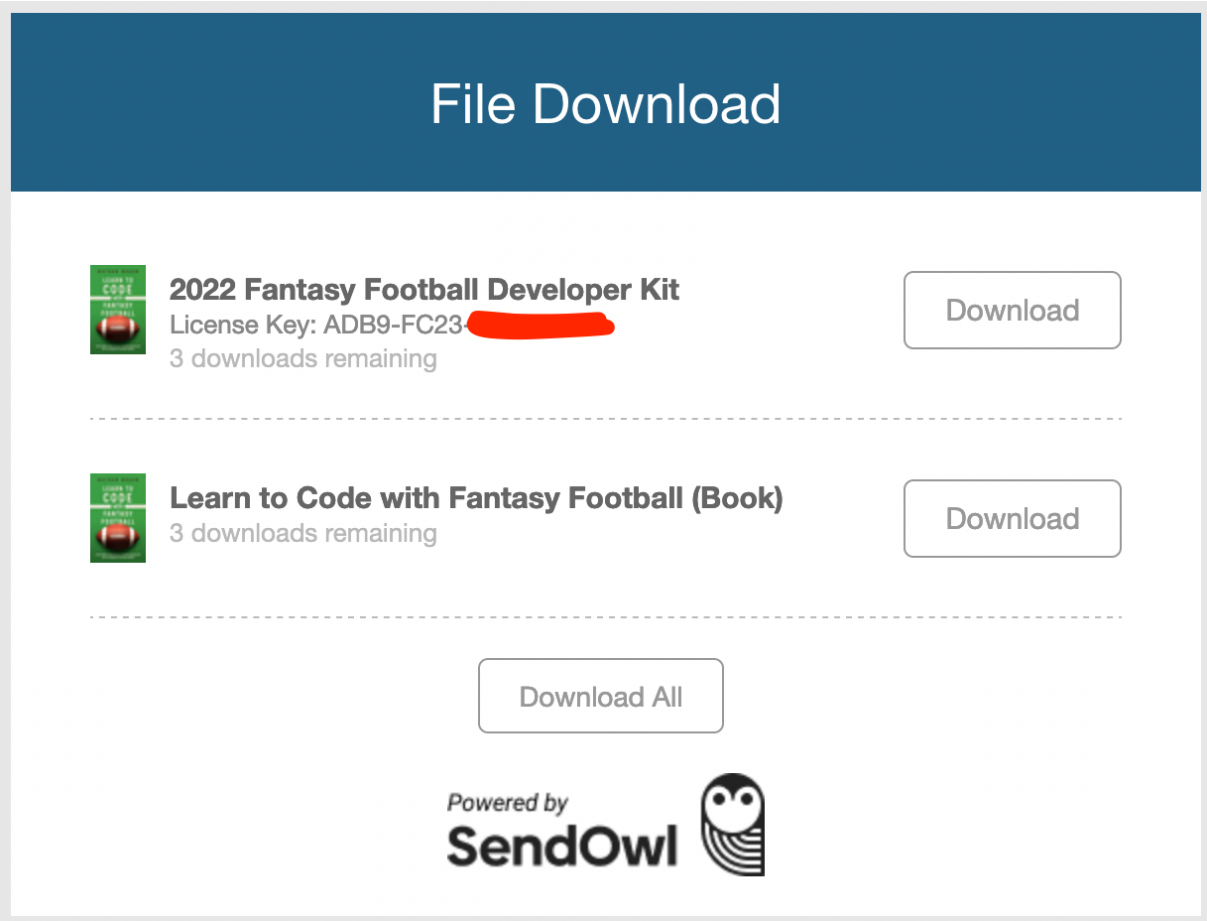

Figure 0.2: Sendowl License Key Screenshot

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 so way I could ever do that." But if I just sat down and tried to write these complete programs off the top of my head I wouldn't be able to either.

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

I 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

The tools we're building rest on the following ideas:

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 players offensive line is, etc).
3. Your job as a fantasy football player is (usually[1]) 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 is 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

[1] Later 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.

But there some major drawbacks with this approach. One is that they restrict you to certain types of curves and shapes. Not all data follows a Normal (or Gamma or Poisson, etc) distribution. But the bigger problem is that manipulating this math is tedious and difficult at best and literally impossible at worse.

Say I told you Kyler Murray and Patrick Mahomes's projections this week are both normally distributed ~N(18,6) and ~N(20.5,7) respectively.

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

Option 2 for working with distributions: simulations

Simulations are a much easier way of working with distributions, especially when you can program. 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. 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. Though 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 Monto Carlo analysis that work with the same raw, underlying simulations that power Fantasy Math.

The developer kit comes with historical access to the API (2017-2021 data, which is what we'll use when walking through the examples), as well as access for the 2022 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`.

Ideally, you would have these files open in your Spyder editor and be running the examples (highlight the line(s) you want and press F9 to send it to the REPL/console) as we go through them in the book.

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

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

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

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

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__'`

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

Authorization Workflow

Try running everything in `utilities.py` above `__name__ == '__main__'` (through line 227).

Then run:

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

You should see something like this:

```
In [2]: token
Out[2]: 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJxaWNlbnNlIjoiss
DN0M1MzMtQkM5MDQxRDctOTdDRUE2MEQtMUY2NTRENUIiLCJleHAiOjE2MDE1MzgxOTN9.
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 that you'll just have to 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 [3]: validate(token)
Out[3]: {'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 back.

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 [4]:
    QUERY_STR = """
        query {
            available(week: 1, season: 2020) {
                fantasymath_id,
                position,
                actual
            }
        }
        """
In [5]:
r = requests.post(API_URL, json={'query': QUERY_STR},
                  headers={'Authorization': f'Bearer {token}'})

In [6]: df = DataFrame(json.loads(r.text)['data']['available'])
```

This returns a `DataFrame` of every player in the Fantasy Math API for week 1, 2020.

```
In [7]: df.head()
Out[7]:
    fantasymath_id position  actual
0  patrick-mahomes       QB   28.15
1    lamar-jackson       QB   35.55
2     dak-prescott       QB   21.20
3   russell-wilson       QB   41.85
4   deshaun-watson       QB   23.80
```

This DataFrame includes the `fantasymath_id`, `position` and `actual` columns 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 (`fantasymath_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 2022 season. You can ask for eligible IDs from prior weeks by passing a season (2017-2021) and week (1-16) argument.

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

`'pass4'` or `'pass6'` for the `qb` argument.

`'ppr'` or `'ppr0'` (e.g. standard, or non-ppr) for the `skill` argument.

`'mfl'` or `'high'` for `dst` scoring. Note you should probably just use mfl 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 2017. This will be useful for the league integration project.

`get_sims`

The `get_sims` function returns the actual simulations.

It take a list of `fantasymath_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 2022 season. If you pass it a season (2017-2021) and week (1-16) argument it'll return past simulations.

It also takes an `nsims` argument for the number of simulations to return (up to 1000).

4. Introduction to the Data

Note: the following assumes you have `intro.py` open in your editor. We'll start from the top.

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

```
In [1]:
import pandas as pd
from utilities import LICENSE_KEY, generate_token, get_players, get_sims
```

Important Note 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 [1]:
from utilities import (LICENSE_KEY, generate_token, get_players, get_sims)
...
ModuleNotFoundError: No module named 'utilities'
```

You're not in the right working directory.

Since all of our projects will use these utility functions, you should always be working out of this directory, and the rest of the projects will assume you are. You can set it by editing the working directory directly or clicking on the Folder icon and finding it that way.

Querying the 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 = 2021
WEEK = 1
SCORING = {'qb': 'pass4', 'skill': 'ppr', 'dst': 'mfl'}

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('fantasymath_id')

In [7]: players.head()
Out[7]:
                position  actual
fantasymath_id
patrick-mahomes       QB   20.12
lamar-jackson         QB   26.94
dak-prescott          QB   16.76
russell-wilson        QB   30.86
deshaun-watson        QB   20.38
```

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 queried and saved 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 1, 2021) the player function returns actual scores too. We can use this to see how the model actually performed. If we wanted to get player ids for *this* week in 2022 we'd leave the `week` and `season` arguments off and do: `get_players(token, **SCORING)`.

Querying sims

Once we have a list of fantasymath 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=1000, **SCORING)


In [11]: sims.head()
Out[11]:
   patrick-mahomes  kyler-murray  josh-allen  ...   younghoe-koo
0        20.773452     23.936068   20.005827  ...      13.871794
1        18.276913     20.569428   13.617354  ...      12.683769
2        26.623091     24.136518   23.145033  ...      17.445629
3        40.953278     26.068849    7.020584  ...       5.128567
4        37.780356     30.797208   16.932572  ...       4.066573
```

Each simulation is a row (there are 1000 sims) and each player is a column (there were 294 in week 1, 2021).

```
In [12]: sims.shape
Out[12]: (1000, 294)
```

Working with Simulations

So take Kyler Murray. 1000 draws. What can we do with this?

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

```
In [13]: sims['kyler-murray'].mean()
Out[13]: 20.56205171344302
```

Or the median:

```
In [14]: sims['kyler-murray'].median()
Out[14]: 20.520401227179107
```

These are just your standard Pandas summary functions. But this is sort of boring. The real benefit of simulations is how they let you work with distributions.

Remember above, when I asked for the probability Kyler Prescott outscores Mahomes?

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

```
In [15]: sims[['kyler-murray', 'patrick-mahomes']].head()
Out[15]:
   kyler-murray  patrick-mahomes
0     23.936068        20.773452
1     20.569428        18.276913
2     24.136518        26.623091
3     26.068849        40.953278
4     30.797208        37.780356
```

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

```
In [16]: (sims['kyler-murray'] > sims['patrick-mahomes']).head()
Out[16]:
0     True
1     True
2    False
3    False
4    False
```

And taking the average of it:

```
In [17]: (sims['kyler-murray'] > sims['patrick-mahomes']).mean()
Out[17]: 0.474
```

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

So Kyler had a 47.4% of outscoring Mahomes in week 1, 2021, which is a little worse than a coinflip.

When working with simulations, you're really only limited by your imagination in terms of the types of questions you can ask. For example, what's the probability Murray outscores both Stafford AND Russell Wilson by at least 11.5 points?

Can you imagine working with the equations above to figure that out mathematically? But, again, with simulations its easy:

```
In [18]:
(sims['kyler-murray'] >
         sims[['matthew-stafford', 'russell-wilson']].max(axis=1) + 11.5).
            mean()
Out[18]: 0.103
```

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.

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

This is something that's impossible to figure out mathematically, 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 [19]: sims['bb_qb'] = sims[['kyler-murray',
                               'matthew-stafford']].max(axis=1)

In [20]: sims[['bb_qb', 'kyler-murray', 'matthew-stafford']].describe()
Out[20]:
             bb_qb  kyler-murray  matthew-stafford
count  1000.000000   1000.000000       1000.000000
mean     23.557781     20.562052         17.285181
std       6.720670      8.016098          7.723573
min       2.951515      0.299006          0.093361
25%      18.831668     15.228552         12.042626
50%      23.602900     20.520401         17.059832
75%      27.703503     25.942013         22.443702
max      43.935287     43.594528         43.935287
```

So on average, we'd expect a best ball duo of Kyler and Stafford to score 23.56 points, vs 20.56 for Dak and 17.29 for Stafford individually.

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

```
In [21]:
sims['bb_qb2'] = sims[['kyler-murray', 'matthew-stafford',
                       'kirk-cousins']].max(axis=1)

In [22]:
sims[['bb_qb2', 'bb_qb', 'kyler-murray', 'matthew-stafford',
      'kirk-cousins']].describe().round(2)
Out[22]:
        bb_qb2    bb_qb  kyler-murray  matthew-stafford  kirk-cousins
count  1000.00  1000.00       1000.00           1000.00       1000.00
mean     24.92    23.56         20.56             17.29         15.64
std       6.22     6.72          8.02              7.72          8.03
min       7.52     2.95          0.30              0.09          0.06
25%      20.59    18.83         15.23             12.04          9.74
50%      24.87    23.60         20.52             17.06         15.60
75%      28.70    27.70         25.94             22.44         21.39
max      43.94    43.94         43.59             43.94         40.18
```

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.

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 0.40. 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.

Pandas makes it easy to calculate the correlation between two columns.

```
In [23]: sims[['kyler-murray', 'deandre-hopkins']].corr()
Out[23]:
                kyler-murray  deandre-hopkins
kyler-murray        1.000000         0.333871
deandre-hopkins     0.333871         1.000000
```

You can see here that Murray and Hopkins — though the data is purely simulated — have a correlation around 0.33, just like it'd be in real life.

Same with Murray and the DST he was playing against (Tennesee in week 1, 2021) — except the correlation between a QB and the DST he's playing against is even stronger.

```
In [24]: sims[['kyler-murray', 'ten-dst']].corr()
Out[24]:
              kyler-murray   ten-dst
kyler-murray      1.000000 -0.524488
ten-dst          -0.524488  1.000000
```

Note: this definitely isn't an accident. It took a lot of work behind the scenes to (a) figure out what these historical correlations were and (b) come up with a way to simulate draws from players while preserving them.

It's turned out well. We ran these separately above, but what's great about these simulations is that they're all correlated *simultaneously*, that is — Kyler is positively correlated with Hopkins (and Chase Edmonds, Christian Kirk and his K etc) and negatively correlated with the Titans DST, *all at the same time.*

```
In [25]: sims[['kyler-murray', 'deandre-hopkins', 'ten-dst']].corr()
Out[25]:
                kyler-murray  deandre-hopkins   ten-dst
kyler-murray        1.000000         0.333871 -0.524488
deandre-hopkins     0.333871         1.000000 -0.246311
ten-dst            -0.524488        -0.246311  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 Hopkins is also negatively correlated with the Titans D, but not as much as Murray.

Let's add a few more:

```
In [26]:
(sims[['kyler-murray', 'ryan-tannehill', 'deandre-hopkins', 'aaron-rodgers
    ',
     'ten-dst']]
     .corr()
      .round(2))

Out[26]:
                k-murray  r-tannehill  d-hopkins  a-rodgers  ten-dst
kyler-murray        1.00         0.20       0.33      -0.05    -0.52
ryan-tannehill      0.20         1.00       0.02       0.04    -0.08
deandre-hopkins     0.33         0.02       1.00      -0.04    -0.25
aaron-rodgers      -0.05         0.04      -0.04       1.00     0.06
ten-dst            -0.52        -0.08      -0.25       0.06     1.00
```

We can see Murray and Tannehill — opposing QBs — are positively correlated. Many times (not ev-

ery 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 Aaron Rodgers (who was playing New Orleans) so you can see he's not correlated with anyone at all. His correlations with all the ARI-TEN players are around around 0 (they're not perfectly 0 since there's noise in the data).

This ~0 correlation 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 1, 2021, DeAndre Hopkins regular, unconditional projected points distribution looked like this:

```
In [27]: sims['deandre-hopkins'].describe()
Out[27]:
count    1000.000000
mean       17.839929
std         9.145592
min         0.083673
25%        11.254122
50%        17.609742
75%        23.956916
max        48.478362
```

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

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

```
In [28]:
pd.concat([
    sims.loc[sims['kyler-murray'] > 30, 'deandre-hopkins'].describe(),
    sims.loc[sims['kyler-murray'] < 12, 'deandre-hopkins'].describe()],
       axis=1)

Out[28]:
       deandre-hopkins  deandre-hopkins
count       117.000000       144.000000
mean         23.280697        12.408938
std           9.110483         6.933093
min           2.156757         0.083673
25%          17.615405         7.155903
50%          24.074598        12.711400
75%          28.261350        17.645547
max          46.666290        27.832841
```

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 week 1, 2021 wondering who should I start: Aaron Rodgers or Russell Wilson?

In terms of a simple, "who will score more points?" the simulations say Rodgers.

```
In [29]: (sims['aaron-rodgers'] > sims['russell-wilson']).mean()
Out[29]: 0.522
```

And that's fine. But what if it's Monday night, I'm down 60 points and I also have Tyler Lockett?

```
In [30]: (sims[['aaron-rodgers', 'tyler-lockett']].sum(axis=1) > 60).mean
    ()
Out[30]: 0.018

In [31]: (sims[['russell-wilson', 'tyler-lockett']].sum(axis=1) > 60).mean
    ()
Out[31]: 0.036
```

My chances with Russell Wilson (3.6%) and Lockett are twice as good vs my chances with Rodgers (1.8%) and Lockett, even though Rodgers usually scores more. This is what taking into account correlations allows you to do.

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. That's exactly is the first project we'll walk through.

5. 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 works[1]. 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 (as strings) 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 before working on anything. I at the outset. I'm just mentioning it here so you know how this project works even if you skip building it yourself.

[1] Remember, 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

Spelling out the API like this also let's 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.

A final version you can use with minimal tweaking is in `./code/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 week 1, 2019, 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 `./code/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 real 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 = 1000
SCORING = {'qb': 'pass6', 'skill': 'ppr', 'dst': 'high'}

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

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

bench = ['darrell-henderson', 'ronald-jones', 'tony-pollard']
```

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 everyone in `team1`, `team2` and `bench` is a *valid Fantasy Math player ID*.

We're not running any checks on the ones I have since I know they work, but if you want to fill in your own I recommend making sure they're valid 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]: list(valid_players['fantasymath_id'])[:20]
Out[5]:
['patrick-mahomes',
 'lamar-jackson',
 'kyler-murray',
 'josh-allen',
 'tom-brady',
 'aaron-rodgers',
 'russell-wilson',
 'jalen-hurts',
 'ryan-tannehill',
 'dak-prescott',
 'matthew-stafford',
 'justin-herbert',
 'matt-ryan',
 'trevor-lawrence',
 'kirk-cousins',
 'joe-burrow',
 'baker-mayfield',
 'sam-darnold',
 'ben-roethlisberger',
 'jameis-winston']
```

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 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'))
else:
    valid_players = get_players(token, season=SEASON, week=WEEK, **SCORING
        )
```

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
~~~

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

~~~ {.python}
In [9]:
if USE_SAVED_DATA:
    sims = pd.read_csv(path.join('projects', 'wdis', 'data', 'sims.csv'))
else:
    sims = get_sims(token, players, week=WEEK, season=SEASON, nsims=NSIMS,
                    **SCORING)
```

Here's what that data looks like:

```
In [10]: sims.head()
Out[10]:
   ronald-jones  clyde-edwards-helaire  ...     gb-dst     tb-dst
0     15.667348              25.095216  ...  18.109629  16.192634
1      6.839920              11.012609  ...  21.042288  16.024047
2     11.986970               3.997752  ...   7.620291   9.792464
3     12.836043              32.077903  ...  13.591491  20.961197
4     10.403136              18.911745  ...   7.605196   9.242636
```

Recall how each Pandas DataFrame includes an index. Here it's just the default, which is 0-999. 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.

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 [11]: sims[team1].head()
Out[11]:
   jalen-hurts  saquon-barkley  ...  jason-myers     tb-dst
0    39.348113        1.525245  ...     6.343894  16.192634
1    29.724425       12.182842  ...    16.075855  16.024047
2    16.245248       21.301939  ...     5.926157   9.792464
3    32.031302        5.301391  ...     7.884661  20.961197
4    31.787873        5.488627  ...    14.625293   9.242636
```

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 [9]: sims[team1].sum()
Out[9]:
jalen-hurts              23800.747985
saquon-barkley           13163.745761
clyde-edwards-helaire    14037.179182
keenan-allen             16148.635810
cooper-kupp              13881.492888
dallas-goedert            8734.796130
jason-myers               7208.671593
tb-dst                   13757.765221
```

Sums up player point totals over all the simulations, which isn't useful. What we want is to sum things up by *row*. So that we have total team1 score for simulation 0, another for sim 1, etc. We get this by passing `axis=1` to sum.

```
In [10]: sims[team1].sum(axis=1).head()
Out[10]:
0    137.002739
1    142.046462
2     96.810310
3    151.593627
4    110.225517
```

So in the first simulation, my team scored 137.00 points. In the second, 142.05. These are the first 5 values out of 1000 simulations.

Here's team2:

```
In [11]: sims[team2].sum(axis=1).head()
Out[11]:
0    131.643266
1     88.513975
2    113.298312
3    118.707099
4    141.310156
```

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 [12]: team1_beats_team2 = sims[team1].sum(axis=1) > sims[team2].sum(
    axis=1)

In [13]: team1_beats_team2.head()
Out[13]:
0     True
1     True
2    False
3     True
4    False
```

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

```
In [14]: team1_beats_team2.mean()
Out[14]: 0.338
```

Yikes. Not looking good.

But that's our probability of winning given our current lineups. Now let's build a simple function that let's us repeat this calculation for all the guys we might want to start.

```
In [15]:
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 using a loop:

```
In [16]:
team1_no_wdis = ['jalen-hurts', 'saquon-barkley', 'keenan-allen',
                 'cooper-kupp', 'dallas-goedert', 'jason-myers', 'tb-dst']

wdis = ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones',
        'tony-pollard']

for player in wdis:
    print(player)
    print(simple_wdis(sims, team1_no_wdis, team2, player))
--
clyde-edwards-helaire
0.338
darrell-henderson
0.304
ronald-jones
0.295
tony-pollard
0.259
```

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 one thing, it's sort of 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 [17]:
def simple_wdis2(sims, team1, team2, wdis):
    return {
        player: (sims[team1 + [player]].sum(axis=1) >
                 sims[team2].sum(axis=1)).mean()
        for player in wdis}
```

And calling it:

```
In [18]: simple_wdis2(sims, team1_no_wdis, team2, wdis)
Out[18]:
{'clyde-edwards-helaire': 0.338,
 'darrell-henderson': 0.304,
 'ronald-jones': 0.295,
 'tony-pollard': 0.259}
```

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 [19]:
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]

    return {
        player: (sims[team1_no_wdis + [player]].sum(axis=1) >
                 sims[team2].sum(axis=1)).mean() for player in wdis}
```

Again, running it:

```
In [20]: simple_wdis3(sims, team1, team2, wdis)
Out[20]:
{'clyde-edwards-helaire': 0.338,
 'darrell-henderson': 0.304,
 'ronald-jones': 0.295,
 'tony-pollard': 0.259}

In [21]: simple_wdis3(sims, team1, team2,
                      ['saquon-barkley', 'darrell-henderson'])
Out[21]: {'saquon-barkley': 0.338, 'darrell-henderson': 0.301}
```

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, incorrect results.

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 [22]: 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 [23]:
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) >= 0

    return Series({
        player: (sims[team1_no_wdis + [player]].sum(axis=1) >
                 sims[team2].sum(axis=1)).mean() for player in wdis})
```

So we ahve:

```
In [24]: simple_wdis4(sims, team1, team2, wdis)
Out[24]:
clyde-edwards-helaire    0.338
darrell-henderson        0.304
ronald-jones             0.295
tony-pollard             0.259
```

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', 'saquon-barkley', 'clyde-edwards-helaire',
         'keenan-allen', 'cooper-kupp', 'dallas-goedert', 'jason-myers',
         'tb-dst']

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


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 projections for our overall team score.

```
In [1]: sims[team1].sum(axis=1).describe()
Out[1]:
count    1000.000000
mean      110.733035
std        24.030553
min        50.562599
25%        94.208634
50%       109.762270
75%       126.326315
max       192.154536
```

This is our projected score starting Edwards-Helaire. It's also be 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([(sims[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    1000.000000        1000.000000   1000.000000   1000.000000
mean      110.733035         107.811707    106.215585    103.497983
std        24.030553          23.663409     23.599774     23.050019
min        50.562599          49.855563     45.789494     44.291899
25%        94.208634          91.433251     90.674087     88.432335
50%       109.762270         106.596040    105.208185    101.880457
75%       126.326315         122.626498    121.004810    118.446831
max       192.154536         195.325792    183.326129    176.332011
```

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  110.733035  24.030553  ...  109.762270  126.326315
darrell-henderson      107.811707  23.663409  ...  106.596040  122.626498
ronald-jones           106.215585  23.599774  ...  105.208185  121.004810
tony-pollard           103.497983  23.050019  ...  101.880457  118.446831
```

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]: sims[bench_options].max(axis=1).head()
Out[5]:
0    16.166280
1    17.146598
2    11.986970
3    12.836043
4    16.209262
```

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

```
In [6]:
pd.concat([sims[bench_options].max(axis=1),
           sims[current_starter]], axis=1).head()
Out[6]:
           0  clyde-edwards-helaire
0  16.166280              25.095216
1  17.146598              11.012609
2  11.986970               3.997752
3  12.836043              32.077903
4  16.209262              18.911745
```

It's another boolean column:

```
In [7]: (sims[bench_options].max(axis=1) > sims[current_starter]).mean()
Out[7]: 0.558
```

So even though we should start CEH, 56% 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 since we're

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 starting CEH instead of one of backups cause us to lose the game? 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 = sims[team1_sans_starter].sum(axis=1) + sims[
    current_starter]

team1_w_best_backup = (sims[team1_sans_starter].sum(axis=1) +
                       sims[bench_options].max(axis=1))

team2_total = sims[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]: 0.056
```

So starting CEH will only lose us our matchup less than 6% of the time.

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 = sims[team1_sans_starter].sum(axis=1) + sims[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 = sims[team1_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[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()
```

Note: these functions (sumstats, win_prob, etc) are using variables and data we've defined above (`sims`, `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 (`sims`, `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]: win_prob(current_starter)
Out[12]: 0.338

In [13]: wrong_prob(current_starter, bench_options)
Out[13]: 0.558

In [14]: regret_prob(current_starter, bench_options)
Out[14]: 0.056

# now with next best alternative, henderson
In [15]: sumstats('darrell-henderson')
Out[15]:
mean    107.811707
std      23.663409
5%       69.444717
25%      91.433251
50%     106.596040
75%     122.626498
95%     148.618580

In [16]: win_prob('darrell-henderson')
Out[16]: 0.304

In [17]:
wrong_prob('darrell-henderson',
           ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard'])
Out[17]: 0.729

In [18]:
regret_prob('darrell-henderson',
           ['clyde-edwards-helaire', 'ronald-jones', 'tony-pollard'])
Out[18]: 0.09
```

So, while we're going to really regret starting CEH 5.6% of the time, starting Henderson will cost us the game even more — 9% 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`) 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 [19]:
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 [20]: wdis
Out[20]: ['clyde-edwards-helaire', 'darrell-henderson', 'ronald-jones',
          'tony-pollard']

In [21]: start_bench_scenarios(wdis)
Out[21]:
[{'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 [22]: wdis
Out[22]: ['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 [23]: player = 'clyde-edwards-helaire'

In [24]: [x for x in wdis if x != player]
Out[24]: ['darrell-henderson', 'ronald-jones', 'tony-pollard']
```

And we want that in a dict:

```
In [25]: {'starter': player, 'bench': [x for x in wdis if x != player]}
Out[25]:
{'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 [26]:
[{'starter': player, 'bench': [x for x in wdis if x != player] }
    for player in wdis]
--
Out[26]:
[{'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 [27]: df = pd.concat([sumstats(player) for player in wdis], axis=1)

In [28]: df = df.T

In [29]: df.head()
Out[29]:
                           mean        std  ...         75%         95%
clyde-edwards-helaire  110.733035  24.030553  ...  126.326315  151.763279
darrell-henderson      107.811707  23.663409  ...  122.626498  148.618580
ronald-jones           106.215585  23.599774  ...  121.004810  146.338239
tony-pollard           103.497983  23.050019  ...  118.446831  143.036962
```

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

```
In [30]:
wps = [win_prob(player) for player in wdis]
df['wp'] = wps

In [31]: df.head()
Out[31]:
                            mean        std  ...         95%     wp
clyde-edwards-helaire  110.733035  24.030553  ...  151.763279  0.338
darrell-henderson      107.811707  23.663409  ...  148.618580  0.304
ronald-jones           106.215585  23.599774  ...  146.338239  0.295
tony-pollard           103.497983  23.050019  ...  143.036962  0.259
```

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

In [33]: df.head()
Out[33]:
                            mean        std  ...     wp  wrong
clyde-edwards-helaire  110.733035  24.030553  ...  0.338  0.558
darrell-henderson      107.811707  23.663409  ...  0.304  0.729
ronald-jones           106.215585  23.599774  ...  0.295  0.801
tony-pollard           103.497983  23.050019  ...  0.259  0.912
```

Finally, regret:

```
In [34]:
df['regret'] = [regret_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']` above [2].)

Our final results:

```
In [35]: df.round(2)
Out[35]:
                        mean    std     5%  ...    wp  wrong  regret
clyde-edwards-helaire  110.73  24.03  72.55  ...  0.34   0.56    0.06
darrell-henderson      107.81  23.66  69.44  ...  0.30   0.73    0.09
ronald-jones           106.22  23.60  67.88  ...  0.30   0.80    0.10
tony-pollard           103.50  23.05  66.74  ...  0.26   0.91    0.14
```

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).

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

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.

```
In [36]:
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

    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()


    # start with DataFrame of summary stats
    df = pd.concat([sumstats(player) for player in wdis], axis=1)
    df.columns = wdis
    df = df.T

    # then add prob of win, being wrong, regretting decision
    df['wp'] = [win_prob(x['starter']) for x in scenarios]
    df['wrong'] = [wrong_prob(**x) for x in scenarios]
    df['regret'] = [regret_prob(**x) for x in scenarios]

    return df.sort_values('wp', ascending=False)
```

And calling it:

```
In [37]: wdis_plus(sims, team1, team2, wdis)
Out[37]:
                            mean        std  ...  wrong  regret
clyde-edwards-helaire  110.733035  24.030553  ...  0.558   0.056
darrell-henderson      107.811707  23.663409  ...  0.729   0.090
ronald-jones           106.215585  23.599774  ...  0.801   0.099
tony-pollard           103.497983  23.050019  ...  0.912   0.135
```

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):

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

if USE_SAVED_DATA:
    k_sims = pd.read_csv(path.join('projects', 'wdis', 'data', 'k_sims.csv
        '))
else:
    k_sims = get_sims(token, fa_kickers, week=WEEK, season=SEASON, nsims
        =1000,
                          **SCORING)

sims_plus = pd.concat([sims, k_sims], axis=1)

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

And running our function:

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

In [3]: df_k.round(2)
Out[3]:
                          mean        std  ...  wrong  regret
jason-myers           110.733035  24.030553  ...  0.947   0.122
brandon-mcmanus       110.418490  24.229420  ...  0.945   0.122
jason-sanders         110.738544  24.464206  ...  0.935   0.124
josh-lambo            110.190778  24.463583  ...  0.952   0.125
nick-folk             109.845195  23.773732  ...  0.974   0.125
chase-mclaughlin      109.940836  24.175336  ...  0.961   0.125
graham-gano           110.147293  23.933109  ...  0.943   0.126
aldrick-rosas         110.031290  24.320196  ...  0.948   0.126
matt-ammendola        110.195370  24.459525  ...  0.935   0.127
quinn-nordin          109.834384  23.789037  ...  0.976   0.128
jake-elliott          109.925878  24.653654  ...  0.947   0.129
rodrigo-blankenship   110.300410  24.342567  ...  0.947   0.130
joey-slye             109.836682  24.277795  ...  0.943   0.130
ryan-santoso          110.108755  24.218905  ...  0.940   0.130
daniel-carlson        110.219441  24.327548  ...  0.932   0.130
cairo-santos          109.678733  24.386241  ...  0.959   0.133
tristan-vizcaino      109.939553  24.416617  ...  0.944   0.134
evan-mcpherson        110.067798  24.342865  ...  0.936   0.136
greg-joseph           109.840432  24.320211  ...  0.936   0.141
```

There we go. We can see Myers maximizes my chances of winning by .002 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  137.002739  131.643266
1  142.046462   88.513975
2   96.810310  113.298312
3  151.593627  118.707099
4  110.225517  141.310156
```

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  137.002739
1    0  team2  131.643266
2    1  team1  142.046462
3    1  team2   88.513975
4    2  team1   96.810310
```

The easiest way to 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    137.002739
   team2    131.643266
1  team1    142.046462
   team2     88.513975
2  team1     96.810310
```

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  137.002739
1    0  team2  131.643266
2    1  team1  142.046462
3    1  team2   88.513975
4    2  team1   96.810310
```

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', shade=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')
```

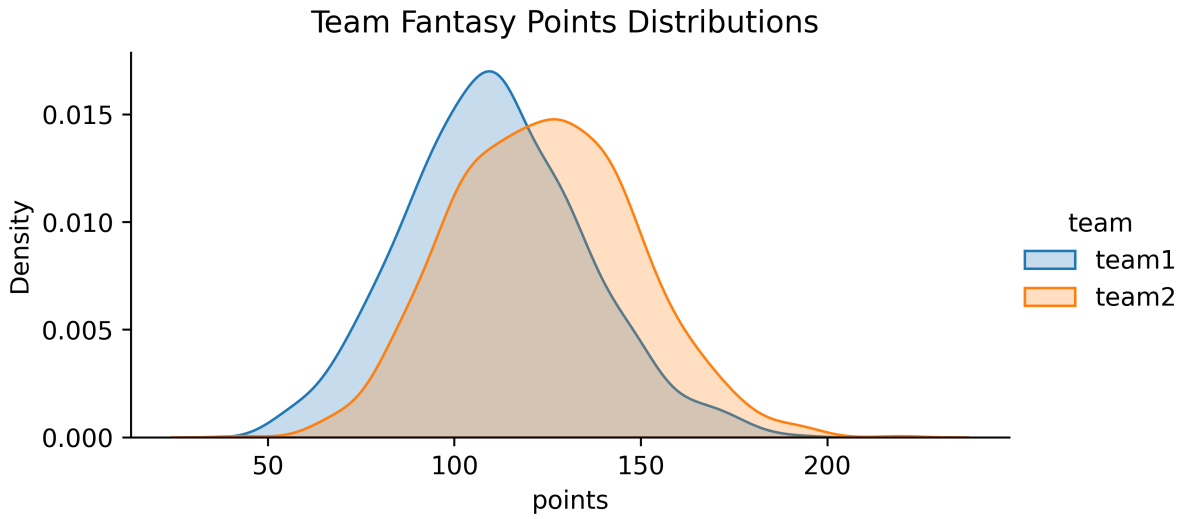

Figure 0.1: Team 1 vs Team 2

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(
    [sims[team1_sans_starter].sum(axis=1) + sims[player] for player in
       wdis],
    axis=1)
points_wide.columns = wdis
points_wide['opp'] = sims[team2].sum(axis=1)
--

In [9]: points_wide.head()
Out[9]:
   clyde-helaire  darrell-henderson  ronald-jones  tony-pollard
      opp
0     137.002739         128.073803    127.574872    124.757691
   131.643266
1     142.046462         148.180450    137.873773    132.068071
   88.513975
2      96.810310         101.719815    104.799528     93.369480
   113.298312
3     151.593627         126.214446    132.351766    125.685228
   118.707099
4     110.225517          97.477974    101.716909    107.523035
   141.310156
```

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  137.002739
1    0      darrell-henderson  128.073803
2    0           ronald-jones  127.574872
3    0           tony-pollard  124.757691
4    0                    opp  131.643266
5    1  clyde-edwards-helaire  142.046462
6    1      darrell-henderson  148.180450
7    1           ronald-jones  137.873773
8    1           tony-pollard  132.068071
9    1                    opp   88.513975
```

And the plot:

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

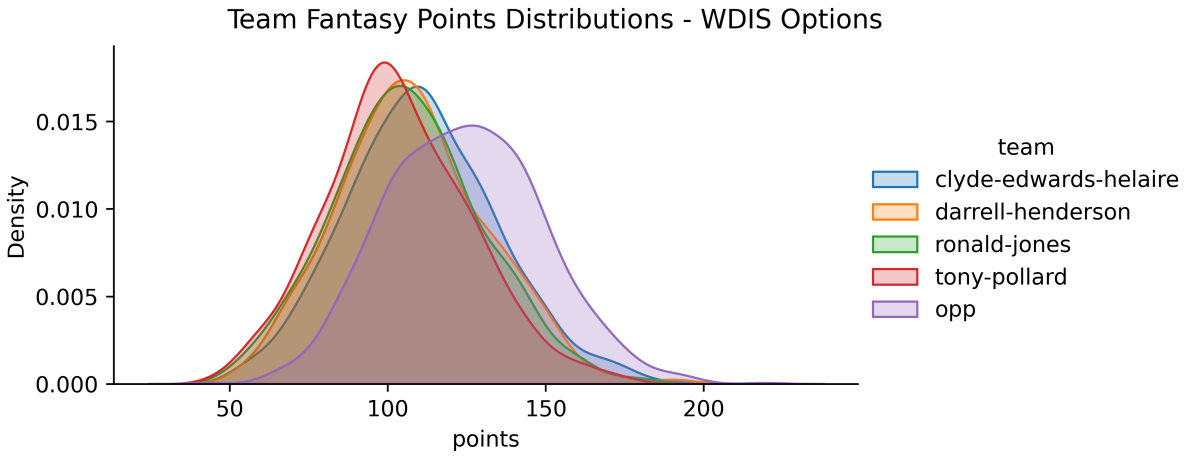

Figure 0.2: WDIS Options vs Opponent

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

```
def wdis_plot(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) >= 0

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

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

    points_wide.columns = wdis

    # add in opponent
    points_wide['opp'] = sims[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', shade=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')
```

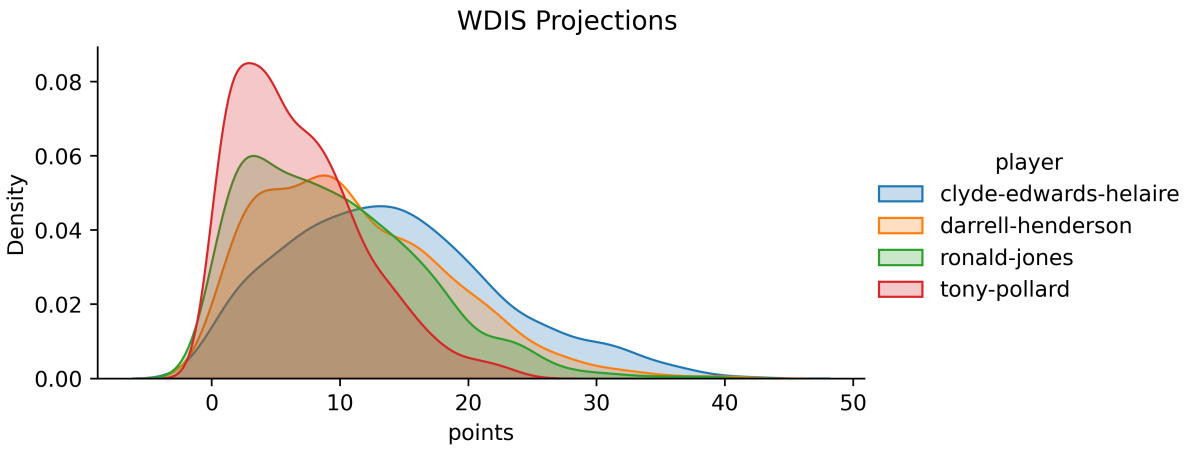

Figure 0.3: WDIS Options - Player Plots

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.

6. 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 to consume. Instead, they're what these platforms use to power their own, internal sites. 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:

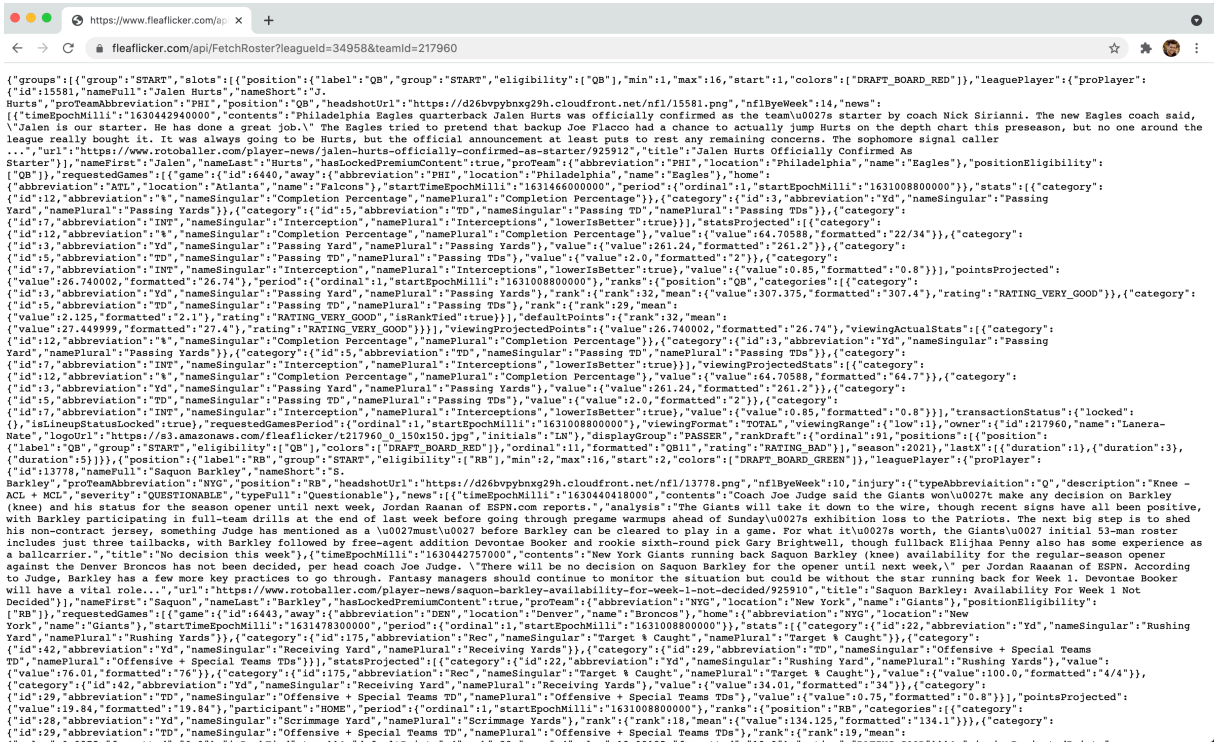

Figure 0.1: Unformatted JSON in Chrome

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.

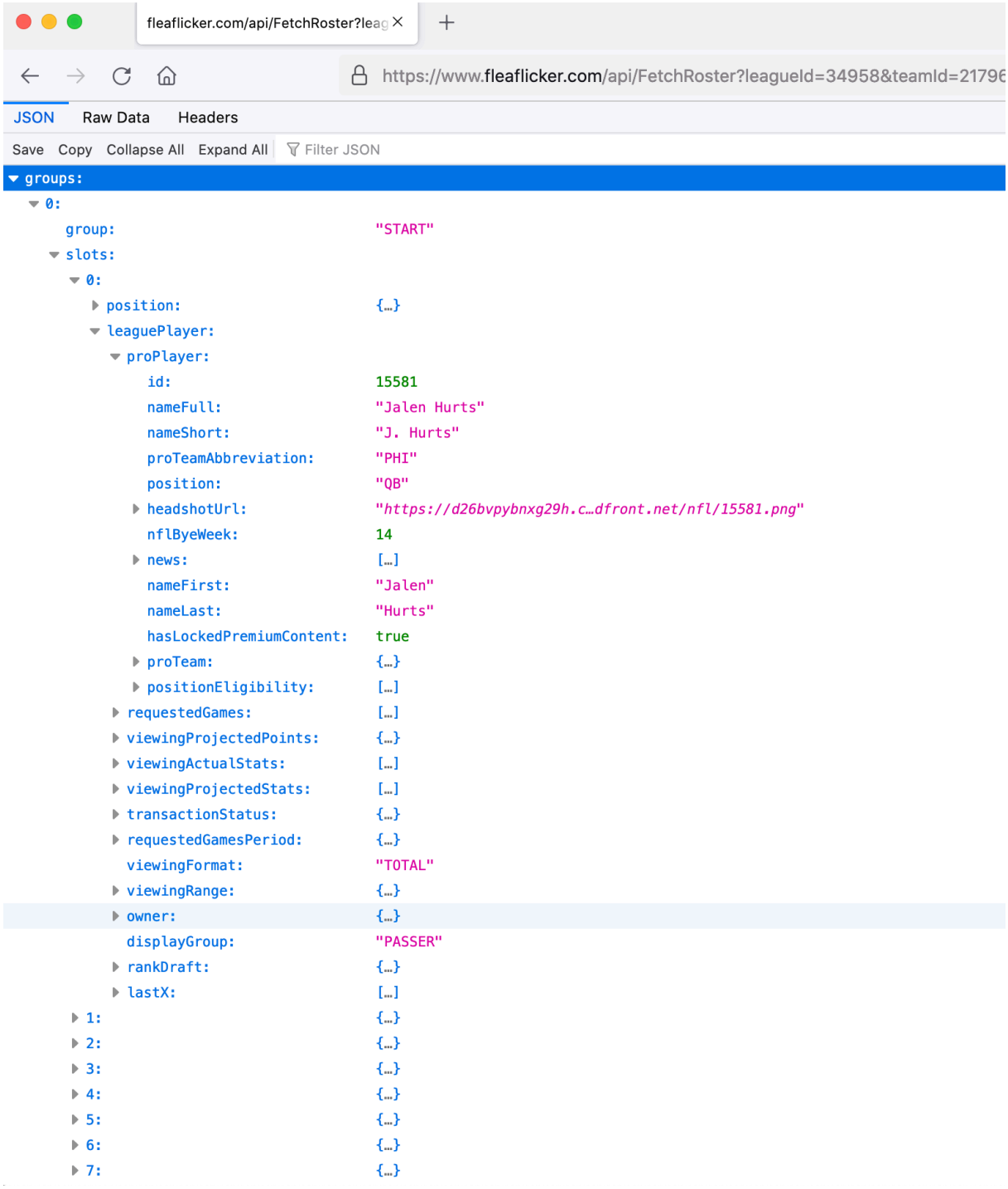

Figure 0.2: Formatted JSON in FireFox

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 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 tables we need:

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 function to reshape our schedule to that.

The `schedule` table includes columns for:

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

The function that gets 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: - `fantasymath_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 and weekly league analyzer projects.

The next sections will be 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.

Connection 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 = 242906
TEAM_ID = 11
```

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 let's 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://fantasy.espn.com/apis/v3/games/ffl/seasons/2021/segments/0/leagues/242906?view=mRoster

where you replace `242906` 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...XXXXXXXXXY8gJl3rF%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://fantasy.espn.com/apis/v3/games/ffl/seasons/2021/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 = ('https://fantasy.espn.com/apis/v3/games/ffl/seasons/2021' +
              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 1, 2021.

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:

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 to view in the console. That's fine. The "make a list" → "view one item" process is *recursive*, i.e. we keep doing it 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. their ESPN 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

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': 2,
 'espn_id': 3068267,
 'name': 'Austin Ekeler',
 'player_position': 2}
```

So this first player is Austin Ekeler. The positions are coded as numbers (2) instead of more readable string values (`'RB'`). 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': 'RB',
  'espn_id': 3068267,
  'name': 'Austin Ekeler',
  'player_position': 'RB'},
 {'team_position': 'RB',
  'espn_id': 3051392,
  'name': 'Ezekiel Elliott',
  'player_position': 'RB'},
 {'team_position': 'TE',
  'espn_id': 3040151,
  'name': 'George Kittle',
  'player_position': 'TE'},
 {'team_position': 'WR',
  'espn_id': 16460,
  'name': 'Adam Thielen',
  '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             RB  3068267        Austin Ekeler              RB
1             RB  3051392      Ezekiel Elliott              RB
2             TE  3040151        George Kittle              TE
3             WR    16460         Adam Thielen              WR
4             WR  2974858       Kenny Golladay              WR
5       RB/WR/TE  3120348  JuJu Smith-Schuster              WR
6             BE  4046692       Chase Claypool              WR
7             QB     2330            Tom Brady              QB
8             BE    15072     Marvin Jones Jr.              WR
9             BE  3916148         Tony Pollard              RB
10          D/ST   -16033          Ravens D/ST            D/ST
11            BE  2572861        J.D. McKissic              RB
12            BE  3915511           Joe Burrow              QB
13            BE  3115378         Russell Gage              WR
14            BE  4258595            Cole Kmet              TE
15             K    14993        Greg Zuerlein               K
```

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
3            WR    16460    Adam Thielen              WR
4            WR  2974858  Kenny Golladay              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 [21]: suffix = Series(range(1, len(wrs) + 1), index=wrs.index)

In [22]: suffix
Out[22]:
3    1
4    2
```

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

```
In [23]: wrs['team_position'] + suffix.astype(str)
Out[23]:
4    WR1
5    WR2
```

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

```
In [24]:
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 [25]:
roster0_df2 = pd.concat([
    add_pos_suffix(roster0_df.query(f"team_position == '{x}'"))
    for x in roster0_df['team_position'].unique()])

In [26]: roster0_df2
Out[26]:
   team_position  espn_id                name player_position
0            RB1  3068267        Austin Ekeler              RB
1            RB2  3051392      Ezekiel Elliott              RB
2             TE  3040151        George Kittle              TE
3            WR1    16460         Adam Thielen              WR
4            WR2  2974858       Kenny Golladay              WR
5       RB/WR/TE  3120348  JuJu Smith-Schuster              WR
6            BE1  4046692       Chase Claypool              WR
8            BE2    15072     Marvin Jones Jr.              WR
9            BE3  3916148         Tony Pollard              RB
11           BE4  2572861        J.D. McKissic              RB
12           BE5  3915511           Joe Burrow              QB
13           BE6  3115378         Russell Gage              WR
14           BE7  4258595            Cole Kmet              TE
7             QB     2330            Tom Brady              QB
10         D/ST   -16033          Ravens D/ST            D/ST
15            K    14993        Greg Zuerlein               K
```

Cool. Now let's identify our starters:

```
In [27]: roster0_df2['start'] = (
               ~roster0_df2['team_position'].str.startswith('BE'))


In [28]: roster0_df2
Out[28]:
   team_position  espn_id                name player_position  start
0            RB1  3068267        Austin Ekeler              RB   True
1            RB2  3051392      Ezekiel Elliott              RB   True
2             TE  3040151        George Kittle              TE   True
3            WR1    16460         Adam Thielen              WR   True
4            WR2  2974858       Kenny Golladay              WR   True
5       RB/WR/TE  3120348  JuJu Smith-Schuster              WR   True
6            BE1  4046692       Chase Claypool              WR  False
8            BE2    15072     Marvin Jones Jr.              WR  False
9            BE3  3916148         Tony Pollard              RB  False
11           BE4  2572861        J.D. McKissic              RB  False
12           BE5  3915511           Joe Burrow              QB  False
13           BE6  3115378         Russell Gage              WR  False
14           BE7  4258595            Cole Kmet              TE  False
7             QB     2330            Tom Brady              QB   True
10         D/ST   -16033          Ravens D/ST            D/ST   True
15            K    14993        Greg Zuerlein               K   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 [29]: roster1 = list_of_rosters[1]
```

So we need our team id:

```
In [30]: roster1['id']
Out[30]: 2
```

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

```
In [31]: process_players(roster1['roster']['entries']).head()
Out[31]:
  team_position  espn_id                 name player_position  start
0           WR1  2976212          Stefon Diggs              WR   True
2           WR2    15795        DeAndre Hopkins              WR   True
1           RB1  3128720             Nick Chubb              RB   True
3           RB2  4039359  Darrell Henderson Jr.              RB   True
4      RB/WR/TE    13982           Julio Jones              WR   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 [32]: roster3 = list_of_rosters[3]

In [33]: process_roster(roster3).head()
Out[33]:
  team_position  espn_id           name player_position  start  team_id
0           WR1  3121422  Terry McLaurin              WR   True        4
2           WR2    15880    Robert Woods              WR   True        4
1           RB1  3042519     Aaron Jones              RB   True        4
4           RB2  3025433      Mike Davis              RB   True        4
3      RB/WR/TE  2577327   Tyler Lockett              WR   True        4
```

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

```
In [34]: 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 [35]: all_rosters.sample(15)
Out[35]:
    team_position  espn_id             name ...  start  team_id
154           BE2  4362628    Ja'Marr Chase ...  False       10
20       RB/WR/TE    13982      Julio Jones ...   True        2
138           BE1  3045147     James Conner ...  False        9
157           BE5  4035004   Mecole Hardman ...  False       10
71             QB  2577417     Dak Prescott ...   True        5
100           WR2  2976499     Amari Cooper ...   True        7
58            BE4  4241401      Trey Sermon ...  False        4
169           BE3  4372016    Jaylen Waddle ...  False       11
152            IR  4241985     J.K. Dobbins ...   True       10
185           BE1  3925347    Damien Harris ...  False       12
63             TE  3043275    Austin Hooper ...   True        4
176      RB/WR/TE  3042778      Corey Davis ...   True       11
171           BE5    15826  Giovani Bernard ...  False       11
53            RB2  3025433       Mike Davis ...   True        4
164           RB1  3929630   Saquon Barkley ...   True       11
```

Great. Anything else?

Well, the saved data we're looking at happens to be from *after* TB-DAL 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 guys 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.

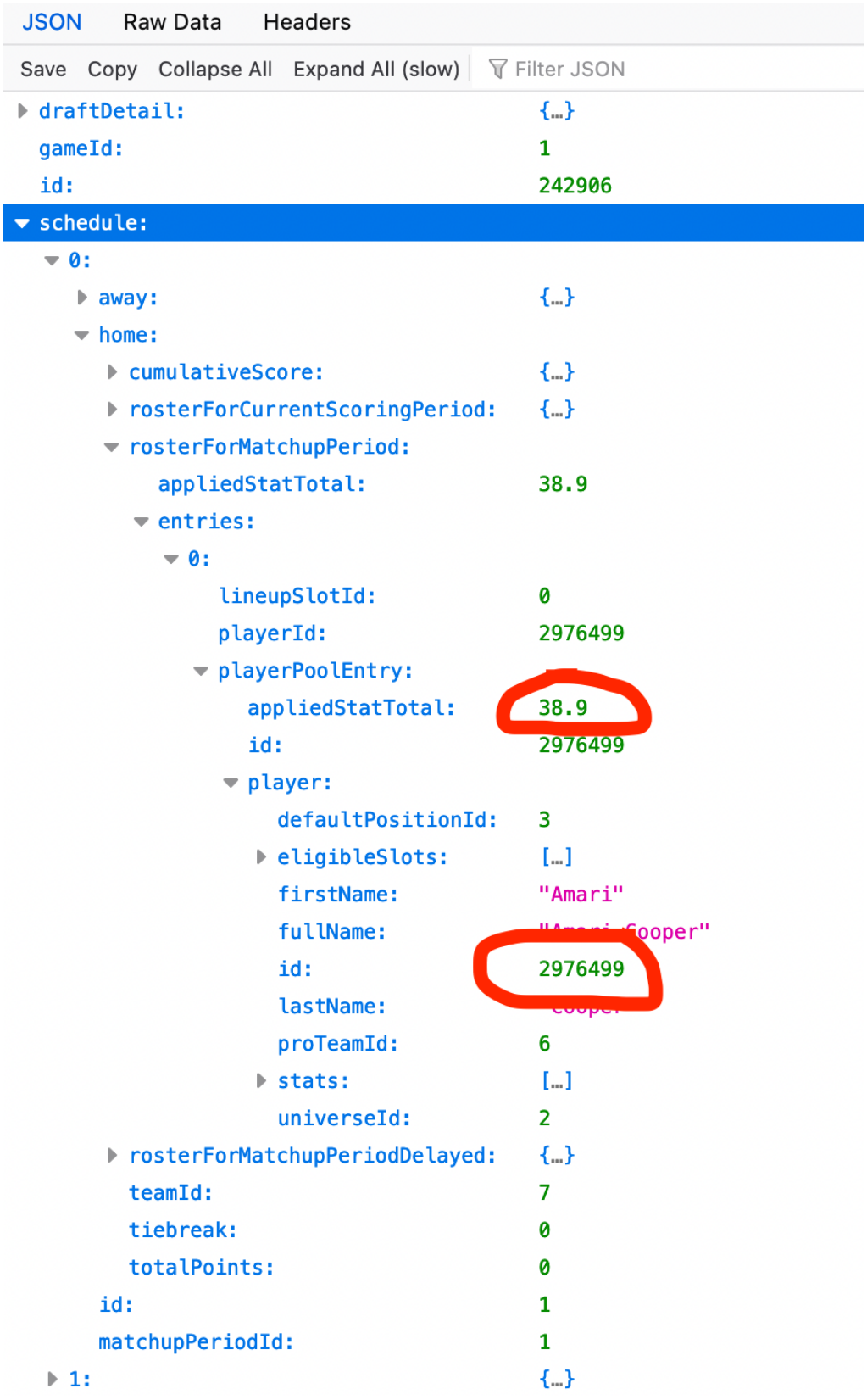

Figure 0.4: Amari Cooper Week 1 ESPN Stats

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 which 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 home team from matchup 0. On that team, the only player who's played is Amari Cooper, who scored 38.9 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:

```
In [3]: matchup_list = boxscore_json['schedule']

In [4]: matchup0 = matchup_list[0]

In [5]: matchup0_home = matchup0['home']

In [6]: matchup0_home_roster = (matchup0['home']
                                   ['rosterForMatchupPeriod']['entries'])

In [7]: amari_cooper_dict = matchup0_home_roster[0]
```

It looks like this:

```
In [8]: amari_cooper_dict
Out[8]:
{'lineupSlotId': 0,
 'playerId': 2976499,
 'playerPoolEntry': {'appliedStatTotal': 38.9,
  'id': 2976499,
  'player': {'defaultPositionId': 3,
   'eligibleSlots': [3, 4, 5, 23, 7, 20, 21],
   'firstName': 'Amari',
   'fullName': 'Amari Cooper',
   'id': 2976499,
   'lastName': 'Cooper',
   'proTeamId': 6,
   'stats': [{'appliedStats': {'53': 13.0, '42': 13.9, '43': 12.0},
     'appliedTotal': 38.9,
     'id': '01null',
     'proTeamId': 0,
     'scoringPeriodId': 0,
     'seasonId': 2021,
     'statSourceId': 0,
     'statSplitTypeId': 1,
     'stats': {'41': 13.0,
      '42': 139.0,
      '43': 2.0,
      '47': 27.0,
      '48': 13.0,
      '49': 6.0,
      '210': 1.0,
      '50': 5.0,
      '51': 2.0,
      '52': 1.0,
      '53': 13.0,
      '54': 2.0,
      '183': 1.0,
      '55': 1.0,
      '56': 1.0,
      '185': 1.0,
      '58': 16.0,
      '59': 29.0,
      '60': 10.69230769,
      '156': 1.0,
      '61': 139.0,
      '158': 12.0}}],
   '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(amari_cooper_dict)
Out[10]: {'espn_id': 2976499, 'actual': 38.9}
```

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 "away" team in matchup 2 had multiple players (Michael Gallup, Bucs D and Chris Godwin). Let's try calling it on this team and make sure it works:

```
In [12]: matchup2 = matchup_list[2]

In [13]: matchup2_away = matchup2['away']

In [14]: proc_played_team(matchup2_away)
Out[14]:
   espn_id  actual
0  4036348     7.6
1   -16027    -3.0
2  3116165    23.5
```

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  2976499    38.9
1  4241389    23.4

In [17]: proc_played_matchup(matchup2)
Out[17]:
   espn_id  actual
0  4036348     7.6
1   -16027    -3.0
2  3116165    23.5
```

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

```
In [18]: WEEK = 1

In [19]: matchup_list = [x for x in boxscore_json['schedule']
                         if x['matchupPeriodId'] == WEEK]

In [20]: scores = pd.concat([proc_played_matchup(x)
                             for x in matchup_list])

In [20]: scores
Out[20]:
   espn_id  actual
0  2976499   38.90
1  4241389   23.40
0  4036348    7.60
1   -16027   -3.00
2  3116165   23.50
0    16737    5.40
0  2577417   27.42
1    14993   10.00
2     2330   27.16
3  3051392    5.90
```

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

In [22]: all_rosters_w_pts.head(15)
Out[22]:
   team_position  espn_id                name start  team_id  actual
0             RB1  3068267        Austin Ekeler  True        1     NaN
1             RB2  3051392      Ezekiel Elliott  True        1    5.90
2              TE  3040151        George Kittle  True        1     NaN
3             WR1    16460         Adam Thielen  True        1     NaN
4             WR2  2974858       Kenny Golladay  True        1     NaN
5       RB/WR/TE  3120348  JuJu Smith-Schuster  True        1     NaN
6             BE1  4046692      Chase Claypool False        1     NaN
7             BE2    15072     Marvin Jones Jr. False        1     NaN
8             BE3  3916148         Tony Pollard False        1     NaN
9             BE4  2572861        J.D. McKissic False        1     NaN
10            BE5  3915511           Joe Burrow False        1     NaN
11            BE6  3115378         Russell Gage False        1     NaN
12            BE7  4046676       Tony Jones Jr. False        1     NaN
13             QB     2330            Tom Brady  True        1   27.16
14          D/ST   -16033          Ravens D/ST  True        1     NaN
```

Sweet. Now we have everything we want except a fantasymath id. If you look at the fantasymath ids, most of them are just the players name, but lowercase and with a dash in the middle. So this would get you 90% of the way there:

```
In [1]: all_rosters_w_pts['name'].str.lower().str.replace(' ','-').head()
Out[1]:
0      austin-ekeler
1    ezekiel-elliott
2      george-kittle
3       adam-thielen
4     kenny-golladay
```

But that doesn't always work, because there are duplicate names, nicknames, juniors and seniors 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 [2] 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/lookup.csv'))

In [4]: fantasymath_players.head()
Out[4]:
  fantasymath_id position  fleaflicker_id    espn_id  yahoo_id
0    matt-prater        K            4221    11122.0    8565.0
1   kyler-murray       QB           14664  3917315.0   31833.0
2  chase-edmonds       RB           13870  3119195.0   31104.0
3   james-conner       RB           12951  3045147.0   30218.0
4  jonathan-ward       RB           15816  4039274.0   33088.0
```

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  ... actual     fantasymath_id
190           BE6          Malcolm Brown  ...    NaN      malcolm-brown
50            WR1         Terry McLaurin  ...    NaN     terry-mclaurin
19            RB2  Darrell Henderson Jr.  ...    NaN  darrell-henderson
74            BE2           DJ Chark Jr.  ...    NaN           dj-chark
43            BE5              A.J. Green  ...    NaN           aj-green
110           BE6          Rashaad Penny  ...    NaN      rashaad-penny
60            BE6            James White  ...    NaN        james-white
90            BE3          Justin Fields  ...    NaN      justin-fields
71             QB           Dak Prescott  ...  27.42       dak-prescott
180            TE           Travis Kelce  ...    NaN       travis-kelce
```

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
- fantasymath-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 = (
    'https://fantasy.espn.com/apis/v3/games/ffl/seasons/2021' +
    f'/segments/0/leagues/{league_id}?view=mRoster')

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

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

    # score part
    boxscore_url = (
        'https://fantasy.espn.com/apis/v3/games/ffl/seasons/2021' +
            f'/segments/0/leagues/{league_id}?view=mBoxscore')

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

    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[['fantasymath_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       fantasymath_id
0           RB1        Austin Ekeler  ...    NaN        austin-ekeler
1           RB2      Ezekiel Elliott  ...    5.9      ezekiel-elliott
2            TE        George Kittle  ...    NaN        george-kittle
3           WR1         Adam Thielen  ...    NaN         adam-thielen
4           WR2       Kenny Golladay  ...    NaN       kenny-golladay
5      RB/WR/TE  JuJu Smith-Schuster  ...    NaN  juju-smith-schuster
6           BE1       Chase Claypool  ...    NaN       chase-claypool
7           BE2    Marvin Jones Jr.  ...    NaN         marvin-jones
8           BE3         Tony Pollard  ...    NaN         tony-pollard
9           BE4        J.D. McKissic  ...    NaN          jd-mckissic
```

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://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['displayName']
    return dict_to_return
```

Teams:

```
In [6]: DataFrame([process_team(team) for team in teams_list])
Out[6]:
    team_id                                 owner_id
0         1  {E3FEF9EF-006C-48DD-A8E9-A2C9FADFDFE0}
1         2  {A864A0DF-73A4-43C3-8C76-3948F6FA4ABC}
2         3  {3EE64B45-F440-4A9D-8F71-5F4E84D23833}
3         4  {CF04A824-7F01-48BB-87E4-ACE2C594F5C3}
4         5  {9A880B26-FF6F-4412-A568-E316D1D20663}
5         6  {815A7E77-6716-4812-8007-DEC3491249B3}
6         7  {C3B7CEC7-19BA-452A-BA1A-12AF6EB6FE0C}
7         8  {353D665C-8F4A-40EA-8E7A-1948F47DC0B7}
8         9  {7E5EE40A-DC21-434B-8F65-049C3F27B20E}
9        10  {9CF0C8D1-F01A-434D-BAE8-BEA61010B8AD}
10       11  {5282E960-0004-4E26-AE4E-263899C8ECCC}
11       12  {44661B21-4D10-4F09-AF41-EC513D578717}
```

And members:

```
In [7]: DataFrame([process_member(member) for member in members_list])
Out[7]:
                                  owner_id         owner_name
0   {353D665C-8F4A-40EA-8E7A-1948F47DC0B7}          Roush427
1   {3EE64B45-F440-4A9D-8F71-5F4E84D23833}      manish6077196
2   {44661B21-4D10-4F09-AF41-EC513D578717}          jrach4017
3   {5282E960-0004-4E26-AE4E-263899C8ECCC}          jmhanson2
4   {7E5EE40A-DC21-434B-8F65-049C3F27B20E}         plownstuff
5   {815A7E77-6716-4812-8007-DEC3491249B3}             r49w41
6   {9A880B26-FF6F-4412-A568-E316D1D20663}  ESPNfan2653927701
7   {9CF0C8D1-F01A-434D-BAE8-BEA61010B8AD}      jwestb7002916
8   {A864A0DF-73A4-43C3-8C76-3948F6FA4ABC}      ChuckFabulous
9   {C3B7CEC7-19BA-452A-BA1A-12AF6EB6FE0C}         heinz11247
10  {CF04A824-7F01-48BB-87E4-ACE2C594F5C3}      ashish8578623
11  {E3FEF9EF-006C-48DD-A8E9-A2C9FADFDFE0}          SwisslyBU
```

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://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(**ESPN_PARAMETERS)

In [9]: league_teams
Out[9]:
    team_id                owner_id         owner_name  league_id
0         1  {E3FEF9EF-...ADFDFE0}          SwisslyBU     242906
1         2  {A864A0DF-...6FA4ABC}      ChuckFabulous     242906
2         3  {3EE64B45-...4D23833}      manish6077196     242906
3         4  {CF04A824-...594F5C3}      ashish8578623     242906
4         5  {9A880B26-...1D20663}  ESPNfan2653927701     242906
5         6  {815A7E77-...91249B3}             r49w41     242906
6         7  {C3B7CEC7-...EB6FE0C}         heinz11247     242906
7         8  {353D665C-...47DC0B7}           Roush427     242906
8         9  {7E5EE40A-...F27B20E}         plownstuff     242906
9        10  {9CF0C8D1-...010B8AD}      jwestb7002916     242906
10       11  {5282E960-...9C8ECCC}           jmhanson2     242906
11       12  {44661B21-...D578717}           jrach4017     242906
```

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 = (
    'https://fantasy.espn.com/apis/v3/games/ffl/seasons/2021' +
        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'] = matchup0['id']  # matchup_id
    dict_to_return['home_id'] = matchup0['home']['teamId']  # "home"
        team_id
    dict_to_return['away_id'] = matchup0['away']['teamId']  # "away"
        team_id
    dict_to_return['week'] = matchup0['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        7        8     1
1           2        3        6     1
2           3       11       12     1
3           4        4        2     1
4           5        9       10     1
5           6        5        1     1
6           7        8        6     2
7           8        3        7     2
8           9       12        2     2
9          10        4       11     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 cute that ESPN assigns a home and away team every matchup, not every hosting platform does this and 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://fantasy.espn.com/apis/v3/games/ffl/seasons
       /2021/segments/0/leagues/{LEAGUE_ID}?view=mBoxscore'

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

    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'] = 2021
    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         7         8     1     242906    2021
1           2         3         6     1     242906    2021
2           3        11        12     1     242906    2021
3           4         4         2     1     242906    2021
4           5         9        10     1     242906    2021
```

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 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 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 1, 2021.

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:

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
```

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()
```

This is how you get live data for your league and team, but for this walkthrough we'll keep using the snapshot I saved. We can load that in Python like this:

```
In [4]:
with open('./projects/integration/raw/fleaflicker/roster.json') as f:
    roster_json = json.load(f)
```

My approach to these things is: 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': 15581,
 'nameFull': 'Jalen Hurts',
 'nameShort': 'J. Hurts',
 'proTeamAbbreviation': 'PHI',
 'position': 'QB',
 'headshotUrl': 'https://d26bvpybnxg29h.cloudfront.net/nfl/15581.png',
 'nflByeWeek': 14,
 'news': [{'timeEpochMilli': '1631106009000',
   'contents': "PFN's In the Mood for Fantasy Football podcast has a new
      episode and it's jam-packed with lineup advice around the latest
      fantasy news.",
   'url': 'https://www.profootballnetwork.com/in-the-mood-ff-podcast-2021-
      nflwk-1/',
   'title': 'In the Mood for Fantasy Football Podcast: Start 'em or sit
      'em for Week 1'}],
 'nameFirst': 'Jalen',
 'nameLast': 'Hurts',
 'hasLockedPremiumContent': True,
 'proTeam': {'abbreviation': 'PHI',
  'location': 'Philadelphia',
  'name': 'Eagles'},
 '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': 'Jalen Hurts',
 'player_position': 'QB',
 'fleaflicker_id': 15581,
 'team_position': 'QB'}
```

Nice. Now let's run this on every slot in our list of starters and bench:

```
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'
```

This is no longer an issue with the snapshot I've included here, but when I first ran this I got an 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': 'Jalen Hurts',
  'player_position': 'QB',
  'fleaflicker_id': 15581,
  'team_position': 'QB'},
 {'name': 'Clyde Edwards-Helaire',
  'player_position': 'RB',
  'fleaflicker_id': 15539,
  'team_position': 'RB'},
 {'name': 'Najee Harris',
  'player_position': 'RB',
  'fleaflicker_id': 16246,
  'team_position': 'RB'},
 {'name': 'Myles Gaskin',
  'player_position': 'RB',
  'fleaflicker_id': 14897,
  'team_position': 'RB/WR/TE'},
  ...
```

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            Jalen Hurts              QB           15581            QB
1  Clyde Edwards-Helaire              RB           15539            RB
2           Najee Harris              RB           16246            RB
3           Myles Gaskin              RB           14897      RB/WR/TE
4          Antonio Brown              WR            6660            WR
5          Davante Adams              WR           10295            WR
6          George Kittle              TE           13023            TE
7          Greg Zuerlein               K            8608             K
8     New Orleans Saints             D/ST            2347           D/ST
```

What else? Well, this snapshot of data we're using is from Friday morning, Week 1, 2021 — right after TB-DAL played Thursday night. In that case Antonio Brown (TB) and Greg Zuerlein (DAL) will have played and have 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_pos~  fleaflicker_id team_position  actual
0           Jalen Hurts          QB           15581            QB     NaN
1  Clyde Edwards-Helaire          RB           15539            RB     NaN
2          Najee Harris          RB           16246            RB     NaN
3          Myles Gaskin          RB           14897      RB/WR/TE     NaN
4         Antonio Brown          WR            6660            WR    18.7
5         Davante Adams          WR           10295            WR     NaN
6         George Kittle          TE           13023            TE     NaN
7         Greg Zuerlein           K            8608             K    12.0
8    New Orleans Saints         D/ST            2347           D/ST     NaN
```

Note the 18.7 and 12.0 for AB and Zuerlein respectively. 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
4   Davante Adams              WR           10295            WR
5  Kenny Golladay              WR           12963            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 [20]: suffix = Series(range(1, len(wrs) + 1), index=wrs.index)

In [21]: suffix
Out[21]:
4    1
5    2
```

They 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_pos~  fleaflicker_id team_position  actual
0           Jalen Hurts          QB           15581            QB     NaN
1  Clyde Edwards-Helaire          RB           15539           RB1     NaN
2          Najee Harris          RB           16246           RB2     NaN
3          Myles Gaskin          RB           14897      RB/WR/TE     NaN
4         Antonio Brown          WR            6660           WR1    18.7
5         Davante Adams          WR           10295           WR2     NaN
6         George Kittle          TE           13023            TE     NaN
7         Greg Zuerlein           K            8608             K    12.0
8     New Orleans Saints         D/ST            2347           D/ST     NaN
```

Cool. Not sure why Fleaflicker considers AB my WR1 over Davante Adams, but it doesn't matter. 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  start
0            Jalen Hurts              QB  ...             QB   True
1   Clyde Edwards-Helaire              RB  ...            RB1   True
2           Najee Harris              RB  ...            RB2   True
3           Myles Gaskin              RB  ...       RB/WR/TE   True
4          Antonio Brown              WR  ...            WR1   True
5          Davante Adams              WR  ...            WR2   True
6          George Kittle              TE  ...             TE   True
7          Greg Zuerlein               K  ...              K   True
8      New Orleans Saints             D/ST  ...          D/ST   True
9        Ryan Fitzpatrick              QB  ...             BN  False
10             Mac Jones              QB  ...             BN  False
11       Javonte Williams              RB  ...             BN  False
12         Kenny Golladay              WR  ...             BN  False
13          Jaylen Waddle              WR  ...             BN  False
14          Elijah Moore              WR  ...             BN  False
15                   NaN             NaN  ...             BN  False
```

This is really close to what we want. We just need my fleaflicker team id, and the corresponding fan-

tasymath ids.

Adding team id is easy:

```
In [30]: roster_df['team_id'] = TEAM_ID
```

Which leaves fantasymath id. If you look at the fantasymath ids, most of them are just the players name, but lowercase and with a dash in the middle. So this would get you 90% of the way there:

```
In [1]: roster_df['name'].str.lower().str.replace(' ','-').head()
Out[1]:
0              jalen-hurts
1    clyde-edwards-helaire
2             najee-harris
3             myles-gaskin
4            antonio-brown
```

But that doesn't always work, because there are duplicate names, nicknames, jr's and seniors 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 [2] from utilities import (LICENSE_KEY, generate_token,
                              master_player_lookup)

In [3]:
token = generate_token(LICENSE_KEY)['token']
fantasymath_players = master_player_lookup(token)
```

And, like the other data we've been working with, I've saved a snapshot we can use for this walk-through and make sure we're seeing the same thing:

```
In [4]: fantasymath_players = pd.read_csv(
                  './projects/integration/raw/lookup.csv')

In [5]: fantasymath_players.head()
Out[5]:
  fantasymath_id position  fleaflicker_id    espn_id  yahoo_id
0    matt-prater        K            4221    11122.0    8565.0
1     colt-mccoy       QB            6625    13199.0   24060.0
2   kyler-murray       QB           14664  3917315.0   31833.0
3  chase-edmonds       RB           13870  3119195.0   31104.0
4   eno-benjamin       RB           15765  4242873.0   32892.0
```

Now we can link this up with the roster from above and we'll be set.

```
In [6]:
roster_df_w_id = pd.merge(
    roster_df, fantasymath_players[['fantasymath_id', 'fleaflicker_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 `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
- fantasymath-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 [7]: my_roster = get_team_roster(TEAM_ID, LEAGUE_ID,
    fantasymath_players)

In [8]: my_roster
Out[8]:
                    name player_pos~  ...  team_id         fantasymath_id
0             Jalen Hurts          QB  ...  1605156            jalen-hurts
1   Clyde Edwards-Helaire          RB  ...  1605156  clyde-edwards-helaire
2            Najee Harris          RB  ...  1605156           najee-harris
3            Myles Gaskin          RB  ...  1605156           myles-gaskin
4           Antonio Brown          WR  ...  1605156          antonio-brown
5           Davante Adams          WR  ...  1605156          davante-adams
6           George Kittle          TE  ...  1605156          george-kittle
7           Greg Zuerlein           K  ...  1605156          greg-zuerlein
8      New Orleans Saints         D/ST  ...  1605156                no-dst
9        Ryan Fitzpatrick          QB  ...  1605156       ryan-fitzpatrick
10             Mac Jones          QB  ...  1605156              mac-jones
11       Javonte Williams          RB  ...  1605156       javonte-williams
12         Kenny Golladay          WR  ...  1605156         kenny-golladay
13          Jaylen Waddle          WR  ...  1605156          jaylen-waddle
14          Elijah Moore          WR  ...  1605156          elijah-moore
15                   NaN         NaN  ...  1605156                   NaN
```

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': 1605154,
 'name': 'Crush em',
 'recordOverall': {'winPercentage': {'formatted': '.000'}, 'formatted': '
    0-0'},
 'recordDivision': {'winPercentage': {'formatted': '.000'},
  'rank': 1,
  'formatted': '0-0'},
 'recordPostseason': {'winPercentage': {'formatted': '.000'},
  'rank': 1,
  'formatted': '0-0'},
 'pointsFor': {'formatted': '0'},
 'pointsAgainst': {'formatted': '0'},
 'streak': {'formatted': '-'},
 'waiverPosition': 10,
 'owners': [{'id': 1319436,
   'displayName': 'ccarns',
   'lastSeen': '1631447061000',
   'initials': 'C',
   'lastSeenIso': '2021-09-12T11:44:21Z'}],
 'newItemCounts': {'activityUnread': 20, 'messagesUnread': 45},
 'initials': 'CE'}
```

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': 1605154, 'owner_id': 1319436, 'owner_name': 'ccarns'}
```

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  1605154   1319436        ccarns
1  1605149   1320047  Plz-not-last
2  1605156    138510        nbraun
3  1605155    103206        BRUZDA
```

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   1605154   1319436          ccarns
1   1605149   1320047    Plz-not-last
2   1605156    138510          nbraun
3   1605155    103206          BRUZDA
4   1605147   1319336       carnsc815
5   1605151   1319571         Edmundo
6   1605153   1319578        LisaJean
7   1664196   1471474     MegRyan0113
8   1603352   1316270        UnkleJim
9   1605157   1324792  JBKBDomination
10  1605148   1319991    Lzrlightshow
11  1605152   1328114       WesHeroux
```

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   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
```

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_pos~  ...  team_id   fantasymath_id
67          Raheem Mostert          RB  ...  1605147   raheem-mostert
65            Alvin Kamara          RB  ...  1605147     alvin-kamara
148             DK Metcalf          WR  ...  1605157       dk-metcalf
114              Joe Mixon          RB  ...  1664196        joe-mixon
26      Alexander Mattison          RB  ...  1605149   alexa~mattison
120     Kansas City Chiefs         D/ST  ...  1664196           kc-dst
69            Adam Thielen          WR  ...  1605147     adam-thielen
77          Marquise Brown          WR  ...  1605147   marquise-brown
64           Lamar Jackson          QB  ...  1605147    lamar-jackson
36           Antonio Brown          WR  ...  1605156    antonio-brown
91           Rashaad Penny          RB  ...  1605151    rashaad-penny
66            Chris Carson          RB  ...  1605147     chris-carson
100           Chris Godwin          WR  ...  1605153     chris-godwin
172          Brandin Cooks          WR  ...  1605148    brandin-cooks
84              D.J. Moore          WR  ...  1605151         dj-moore
33   Clyde Edwards-Helaire          RB  ...  1605156    clyde~helaire
20          Terry McLaurin          WR  ...  1605149   terry-mclaurin
191      Green Bay Packers         D/ST  ...  1605152           gb-dst
170           Nyheim Hines          RB  ...  1605148     nyheim-hines
124              Zack Moss          RB  ...  1664196        zack-moss
```

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]:
WEEK = 1
schedule_url = (
    'https://www.fleaflicker.com/api/FetchLeagueScoreboard?' +
    f'leagueId={LEAGUE_ID}&scoringPeriod={WEEK}&season=2021')

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]: {'home_id': 1603352, 'away_id': 1605152, 'matchup_id': '49030923'}
```

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'] = 2021
    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) for week in
                      range(1, 15)], ignore_index=True)

In [8]: league_schedule.head()
Out[8]:
   team1_id  team2_id matchup_id  season  week  league_id
0   1603352   1605152   49030923    2021     1     316893
1   1605147   1605153   49030921    2021     1     316893
2   1605149   1605155   49030919    2021     1     316893
3   1605154   1605156   49030920    2021     1     316893
4   1605148   1605157   49030924    2021     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 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 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 `717250053961510912`.

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:

- `fantasymath_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, 2021:

https://api.sleeper.app/v1/league/717250053961510912/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, 2021.

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:

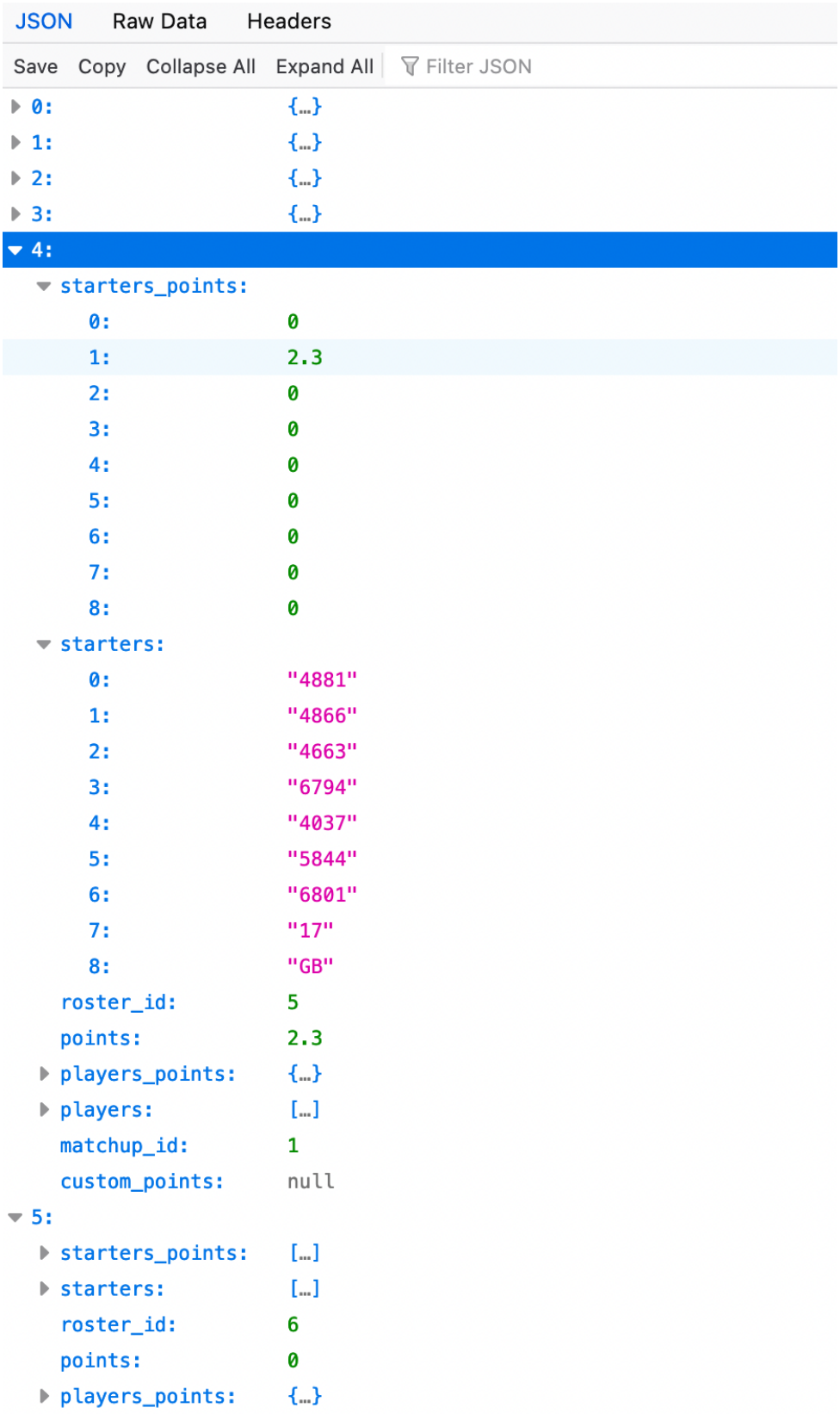

Figure 0.6: Sleeper Roster JSON

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 = 717250053961510912
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]: team9 = matchup_json[9]

In [5]: team9['starters']
Out[5]: ['6797', '6130', '6945', '2133', '5859', '7553', '4068',
         '4227', 'WAS']
```

The `starters` field is just a list. Based on the fact the defense (Washington 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',
 'TE',
 'FLEX',
 'K',
 'DEF',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN',
 '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]:
    fantasymath_id position  ~ sleeper_id
0      matt-prater        K  ~         17
1     kyler-murray       QB  ~       5849
2    chase-edmonds       RB  ~       5000
3     james-conner       RB  ~       4137
4  darrell-daniels       TE  ~       4323
```

We'll use a saved snapshot of it:

```
In [65]: 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 [12]: starters9 = Series(team9['starters']).to_frame('sleeper_id')

In [13]: starters9
Out[13]:
  sleeper_id
0       6797
1       6130
2       6945
3       2133
4       5859
5       7553
6       4068
7       4227
8        WAS

In [14]: type(starters9)
Out[14]: pandas.core.frame.DataFrame
```

You could also turn the list of starters into a list of dictionaries:

```
In [15]: DataFrame([{'sleeper_id': x} for x in team9['starters']])
Out[15]:
  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 `fantasymath_id` and `position` from our lookup:

```
In [17]: starters9_w_info = pd.merge(starters9, fantasymath_players,
                                     how='left')

In [18]: starters9_w_info
Out[18]:
  sleeper_id    fantasymath_id position  ...  yahoo_id
0       6797    justin-herbert       QB  ...   32676.0
1       6130  devin-singletary       RB  ...   31906.0
2       6945    antonio-gibson       RB  ...   32736.0
3       2133     davante-adams       WR  ...   27581.0
4       5859          aj-brown       WR  ...   31883.0
5       7553        kyle-pitts       TE  ...   33392.0
6       4068  mike-williams-wr       WR  ...   30120.0
7       4227   harrison-butker        K  ...   30346.0
8        WAS           was-dst      DST  ...  100028.0
```

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. NYG-WAS played Thursday. We want these points reflected in our roster, so let's add them.

```
In [19]: starters9_w_info['actual'] = team9['starters_points']

In [20]: starters9_w_info
Out[20]:
  sleeper_id    fantasymath_id position  ...  actual
0       6797    justin-herbert       QB  ...    0.00
1       6130  devin-singletary       RB  ...    0.00
2       6945    antonio-gibson       RB  ...    2.43
3       2133     davante-adams       WR  ...    0.00
4       5859          aj-brown       WR  ...    0.00
5       7553        kyle-pitts       TE  ...    0.00
6       4068  mike-williams-wr       WR  ...    0.00
7       4227   harrison-butker        K  ...    0.00
8        WAS           was-dst      DST  ...    4.00
```

We can see the two WAS-NYG players (Gibson and WAS defense) have 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    fantasymath_id position  ...  actual
0       6797    justin-herbert       QB  ...     NaN
1       6130  devin-singletary       RB  ...     NaN
2       6945    antonio-gibson       RB  ...    2.43
3       2133     davante-adams       WR  ...     NaN
4       5859          aj-brown       WR  ...     NaN
5       7553        kyle-pitts       TE  ...     NaN
6       4068  mike-williams-wr       WR  ...     NaN
7       4227   harrison-butker        K  ...     NaN
8        WAS           was-dst      DST  ...    4.00
```

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, Mike Williams 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 [29]: positions
Out[29]:
['QB',
 'RB',
 'RB',
 'WR',
 'WR',
 'TE',
 'FLEX',
 'K',
 'DEF',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN',
 'BN']
```

Let's use these to make a "team position" column:

```
In [22]: starters9_w_info['team_position'] = [x for x in positions
                                               if x != 'BN']

In [23]: starters9_w_info
Out[23]:
  sleeper_id    fantasymath_id position  ...  actual team_position
0       6797    justin-herbert       QB  ...     NaN            QB
1       6130  devin-singletary       RB  ...     NaN            RB
2       6945    antonio-gibson       RB  ...    2.43            RB
3       2133     davante-adams       WR  ...     NaN            WR
4       5859         aj-brown       WR  ...     NaN            WR
5       7553        kyle-pitts       TE  ...     NaN            TE
6       4068  mike-williams-wr       WR  ...     NaN          FLEX
7       4227   harrison-butker        K  ...     NaN             K
8        WAS           was-dst      DST  ...    4.00           DEF
```

Now we can tell Mike Williams is in the FLEX. Awesome.

This looks good, but at some point duplicate `team_position`s 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 fantasymath_id position  ...  actual team_position
3       2133  davante-adams       WR  ...     NaN            WR
4       5859       aj-brown       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
```

They 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
dtype: object
```

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]:
starters9_pos = pd.concat([
    add_pos_suffix(starters9_w_info.query(f"team_position == '{x}'"))
    for x in starters9_w_info['team_position'].unique()])

--

In [30]: starters9_pos
Out[30]:
  sleeper_id    fantasymath_id position  ...  actual team_position
0       6797    justin-herbert       QB  ...     NaN            QB
1       6130  devin-singletary       RB  ...     NaN           RB1
2       6945    antonio-gibson       RB  ...    2.43           RB2
3       2133     davante-adams       WR  ...     NaN           WR1
4       5859          aj-brown       WR  ...     NaN           WR2
5       7553        kyle-pitts       TE  ...     NaN            TE
6       4068  mike-williams-wr       WR  ...     NaN          FLEX
7       4227   harrison-butker        K  ...     NaN             K
8        WAS           was-dst      DST  ...    4.00           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]:
players9 = Series(team9['players']).to_frame('sleeper_id')
players9_w_info = pd.merge(players9, fantasymath_players, how='left')

In [32]: players9_w_info
Out[32]:
   sleeper_id    fantasymath_id position  ...  yahoo_id
0         WAS           was-dst      DST  ...  100028.0
1        7588  javonte-williams       RB  ...   33423.0
2        7553        kyle-pitts       TE  ...   33392.0
3        7527         mac-jones       QB  ...   33403.0
4        6945    antonio-gibson       RB  ...   32736.0
5        6798      jalen-reagor       WR  ...   32691.0
6        6797    justin-herbert       QB  ...   32676.0
7        6783               NaN      NaN  ...       NaN
8        6130  devin-singletary       RB  ...   31906.0
9        5980      myles-gaskin       RB  ...   32066.0
10       5947     jakobi-meyers       WR  ...   32231.0
11       5859          aj-brown       WR  ...   31883.0
12       4962       sony-michel       RB  ...   31001.0
13       4227   harrison-butker        K  ...   30346.0
14       4144       jonnu-smith       TE  ...   30213.0
15       4068  mike-williams-wr       WR  ...   30120.0
16         24         matt-ryan       QB  ...    8780.0
17       2133     davante-adams       WR  ...   27581.0
```

The only wrinkle is — unlike starter points — player points are a *dict* of values:

```
In [33]: team9['players_points']
Out[33]:
{'WAS': 4.0,
 '7588': 0.0,
 '7553': 0.0,
 '7527': 0.0,
 '6945': 2.43,
 '6798': 0.0,
 '6797': 0.0,
 '6783': 0.0,
 '6130': 0.0,
 '5980': 0.0,
 '5947': 0.0,
 '5859': 0.0,
 '4962': 0.0,
 '4227': 0.0,
 '4144': 0.0,
 '4068': 0.0,
 '24': 0.0,
 '2133': 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]:
players9_w_info['actual'] = (players9_w_info['sleeper_id'].apply(
                             lambda x: team9['players_points'][x]))

In [35]: players9_w_info
Out[35]:
   sleeper_id    fantasymath_id position  ...  actual
0         WAS           was-dst      DST  ...    4.00
1        7588  javonte-williams       RB  ...    0.00
2        7553       kyle-pitts       TE  ...    0.00
3        7527        mac-jones       QB  ...    0.00
4        6945   antonio-gibson       RB  ...    2.43
5        6798     jalen-reagor       WR  ...    0.00
6        6797   justin-herbert       QB  ...    0.00
7        6783              NaN      NaN  ...    0.00
8        6130  devin-singletary       RB  ...    0.00
9        5980     myles-gaskin       RB  ...    0.00
10       5947    jakobi-meyers       WR  ...    0.00
11       5859         aj-brown       WR  ...    0.00
12       4962      sony-michel       RB  ...    0.00
13       4227   harrison-butker        K  ...    0.00
14       4144      jonnu-smith       TE  ...    0.00
15       4068  mike-williams-wr       WR  ...    0.00
16         24        matt-ryan       QB  ...    0.00
17       2133    davante-adams       WR  ...    0.00
```

And getting only the bench players:

```
In [36]: bench_players = set(team9['players']) - set(team9['starters'])

In [37]: bench_players
Out[37]: {'24', '4144', '4962', '5947', '5980', '6783', '6798', '7527',
          '7588'}

In [38]:
bench_df = players9_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

In [39]: bench_df
Out[39]:
   sleeper_id    fantasymath_id position  ...  actual  team_position
1        7588  javonte-williams       RB  ...     NaN             BN
3        7527         mac-jones       QB  ...     NaN             BN
5        6798      jalen-reagor       WR  ...     NaN             BN
7        6783               NaN      NaN  ...     NaN             BN
9        5980      myles-gaskin       RB  ...     NaN             BN
10       5947     jakobi-meyers       WR  ...     NaN             BN
12       4962       sony-michel       RB  ...     NaN             BN
14       4144       jonnu-smith       TE  ...     NaN             BN
16         24         matt-ryan       QB  ...     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 [40]: team9_df = pd.concat([starters9_pos, bench_df],
                              ignore_index=True)

In [41]: team9_df.drop(['yahoo_id', 'espn_id', 'fleaflicker_id',
                        'sleeper_id'], axis=1, inplace=True)

In [42]: team9_df.rename(columns={'position': 'player_position'},
                         inplace=True)

In [43]: team9_df['start'] = team9_df['team_position'] != 'BN'

In [44]: team9_df['name'] = (team9_df['fantasymath_id']
                             .str.replace('-', ' ')
                             .str.title())

In [45]: team9_df['team_id'] = team9['roster_id']

In [46]: team9_df
Out[46]:
      fantasymath_id player_position  ...              name  team_id
0     justin-herbert              QB  ...    Justin Herbert       10
1   devin-singletary              RB  ...  Devin Singletary       10
2     antonio-gibson              RB  ...    Antonio Gibson       10
3      davante-adams              WR  ...     Davante Adams       10
4           aj-brown              WR  ...          Aj Brown       10
5         kyle-pitts              TE  ...        Kyle Pitts       10
6   mike-williams-wr              WR  ...  Mike Williams Wr       10
7    harrison-butker               K  ...   Harrison Butker       10
8            was-dst             DST  ...           Was Dst       10
9   javonte-williams              RB  ...  Javonte Williams       10
10         mac-jones              QB  ...         Mac Jones       10
11      jalen-reagor              WR  ...      Jalen Reagor       10
12               NaN             NaN  ...               NaN       10
13      myles-gaskin              RB  ...      Myles Gaskin       10
14     jakobi-meyers              WR  ...     Jakobi Meyers       10
15       sony-michel              RB  ...       Sony Michel       10
16       jonnu-smith              TE  ...       Jonnu Smith       10
17         matt-ryan              QB  ...         Matt Ryan       10
```

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 [47]:
all_rosters = pd.concat(
  [get_team_roster(x, fantasymath_players, positions) for x in
     matchup_json],
  ignore_index=True)
--

In [48]: all_rosters.sample(10)
Out[48]:
            fantasymath_id player_pos~  ...                  name team_id
131           robert-woods          WR  ...          Robert Woods       8
119             joe-burrow          QB  ...            Joe Burrow       7
45                 mia-dst         DST  ...               Mia Dst       3
97            stefon-diggs          WR  ...          Stefon Diggs       6
39           calvin-ridley          WR  ...         Calvin Ridley       3
64                 ind-dst         DST  ...               Ind Dst       4
49           damien-harris          RB  ...         Damien Harris       3
92   clyde-edwards-helaire          RB  ...  Clyde Edwards Helaire       6
71         latavius-murray          RB  ...       Latavius Murray       4
104          antonio-brown          WR  ...         Antonio Brown       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()
    with open('./projects/integration/raw/sleeper/matchup.json') as f:
        matchup_json = json.load(f)

    settings_url = f'https://api.sleeper.app/v1/league/{LEAGUE_ID}'
    # settings_json = requests.get(settings_url).json()
    with open('./projects/integration/raw/sleeper/settings.json') as f:
        settings_json = json.load(f)

    return pd.concat([
      get_team_roster(x, lookup, settings_json['roster_positions'])
      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]: team0 = teams_json[0]
```

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(team0)
Out[5]: {'owner_id': '336962490602622976', 'owner_name': 'Centurions'}
```

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': '336962490602622976',
         'owner_name': 'Centurions',
         '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
Centurions
2
Plutolife
3
KingKane
4
MattBurtt
5
DangerZone22
6
SpartanLife
7
stefense
8
Sprinter
9
ChokingHawks
10
75SteelCurtain
```

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  336962490602622976      Centurions        1
1  338853659477540864       Plutolife        2
2  338854151683321856        KingKane        3
3  339268993724469248       MattBurtt        4
4  339274968888004608    DangerZone22        5
5  339276558738935808     SpartanLife        6
6  339536653859172352        stefense        7
7  339613463590469632        Sprinter        8
8  447142474310217728    ChokingHawks        9
9  605632280605581312  75SteelCurtain       10
```

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, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 'starters': ['96',
  '4035',
  '6151',
  '1426',
  '1992',
  '4217',
  '2374',
  '7062',
  'LAC'],
 'roster_id': 1,
 'points': 0.0,
 ...
 'players': ['LAC',
  'CLE',
  '96',
  '7062',
  '6290',
  '6151',
  '5937',
  '59',
  '4993',
  '4217',
  '4149',
  '4035',
  '3157',
  '2374',
  '2320',
  '1992',
  '1426',
  '1166'],
 'matchup_id': 5,
 '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['game_id'] = team['matchup_id']
    return dict_to_return
```

Running it:

```
In [4]: proc_team_schedule(team0)
Out[4]: {'team_id': 1, 'game_id': 5}
```

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  game_id
0        1        5
1        2        2
2        3        1
3        4        3
4        5        1
5        6        5
6        7        4
7        8        4
8        9        3
9       10        2
```

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 `game_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('game_id', keep='first'),
    schedule_w2.drop_duplicates('game_id', keep='last'), on='game_id')

In [8]: schedule_w2_wide
Out[8]:
   team_id_x  game_id  team_id_y
0          1        5          6
1          2        2         10
2          3        1          5
3          4        3          9
4          7        4          8
```

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'] = 2021
schedule_w2_wide['week'] = WEEK

In [11]: schedule_w2_wide
Out[11]:
   team1_id  game_id  team2_id  season  week
0         1        5         6    2021     2
1         2        2        10    2021     2
2         3        1         5    2021     2
3         4        3         9    2021     2
4         7        4         8    2021     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('game_id', keep='first'),
        team_sched.drop_duplicates('game_id', keep='last'), on='game_id')

    team_sched_wide.rename(
        columns={'team_id_x': 'team1_id', 'team_id_y': 'team2_id'},
        inplace=True)

    team_sched_wide['season'] = 2021
    team_sched_wide['week'] = week
    return team_sched_wide
```

And try it out:

```
In [12]: get_schedule_by_week(LEAGUE_ID, 3)
Out[12]:
   team1_id  game_id  team2_id  season  week
0         1        4         8    2021     3
1         2        1         3    2021     3
2         4        2         5    2021     3
3         6        3        10    2021     3
4         7        5         9    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 [13]: settings_json['settings']['playoff_week_start']
Out[13]: 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']
    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  game_id  team2_id  season  week
0          1        5         7    2021     1
1          2        2         5    2021     1
2          3        3         8    2021     1
3          4        1        10    2021     1
4          6        4         9    2021     1
..       ...      ...       ...     ...   ...
60         1        5         9    2021    13
61         2        2         4    2021    13
62         3        1        10    2021    13
63         5        3         7    2021    13
64         6        4         8    2021    13
```

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 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 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 app etc, 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 social 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.

Registering Your Yahoo App

1. Login to your Yahoo league and 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`.
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:

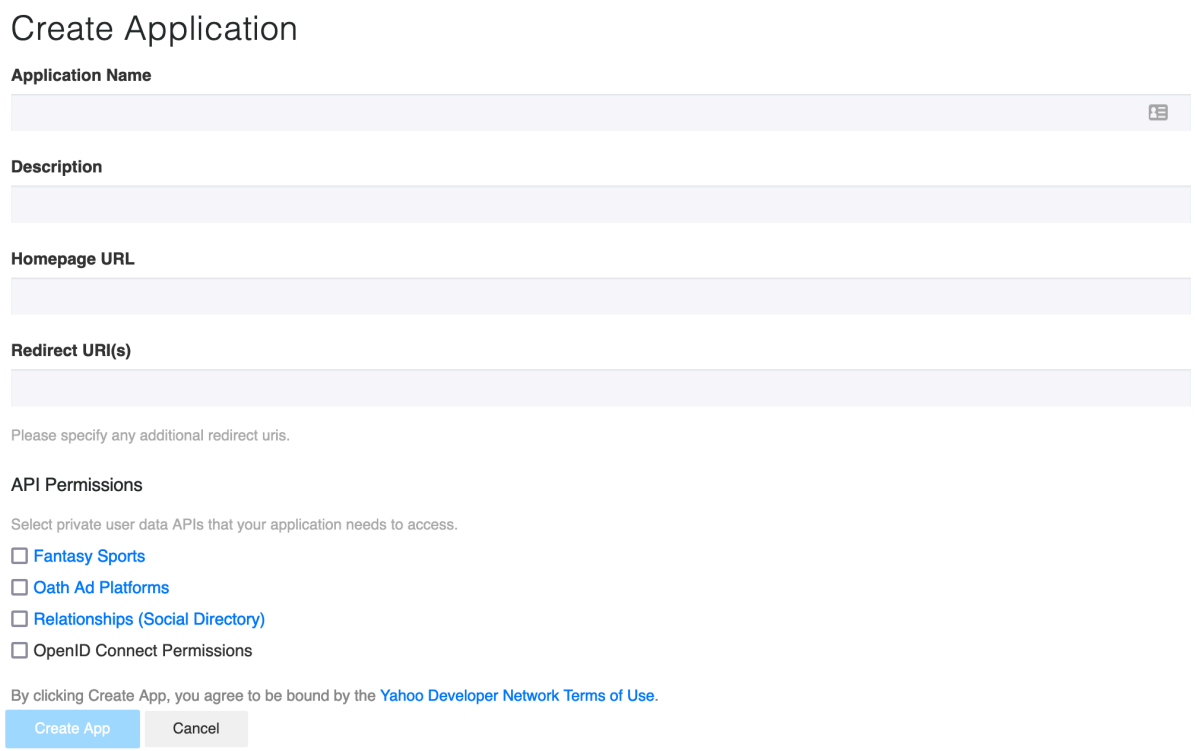

Figure 0.7: Create Yahoo Application

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`.

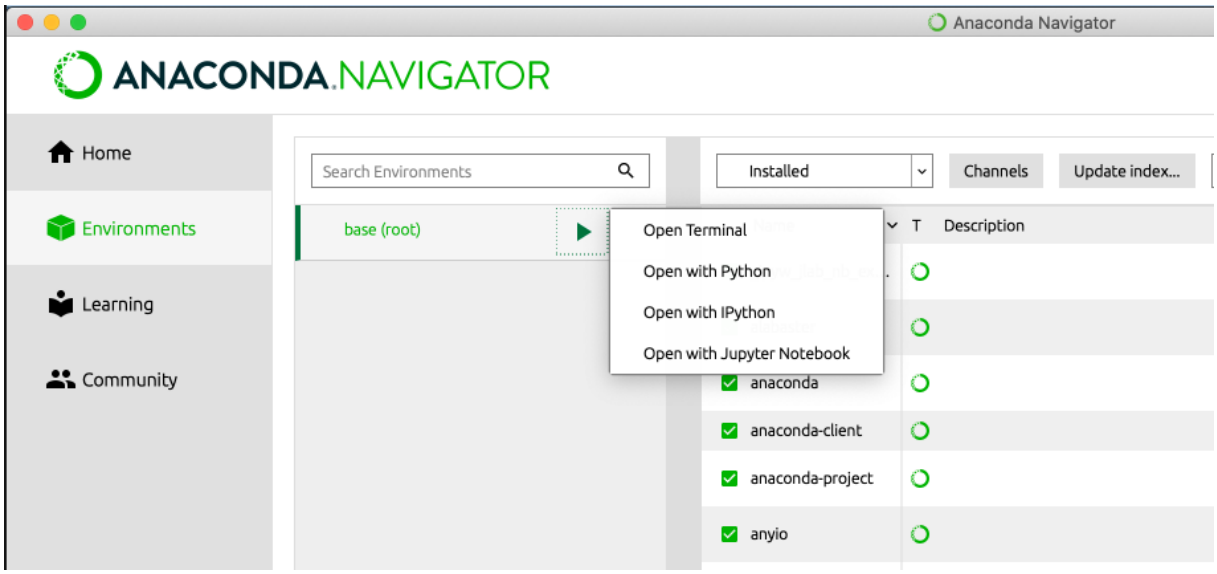

Figure 0.8: Opening Terminal in Anaconda

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 =
    dj0yJmk9VTZXXXXXXXXXXXXXXXXXXXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk
CONSUMER_SECRET = 56ec2XXXXXXXXXXX8aa1eff00c8
```

Now open up `./code/integration/yahoo_working.py`. We'll pick up right after the imports.

Those are my keys, partially X'd out so no one goes in and drops all my players.

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)
...
```

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":
"dj0yJmk9VTZXXXXXXXXXXXXXXXXXXXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk",
"consumer_secret": "56ec2XXXXXXXXXXX8aa1eff00c8"}
```

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`.

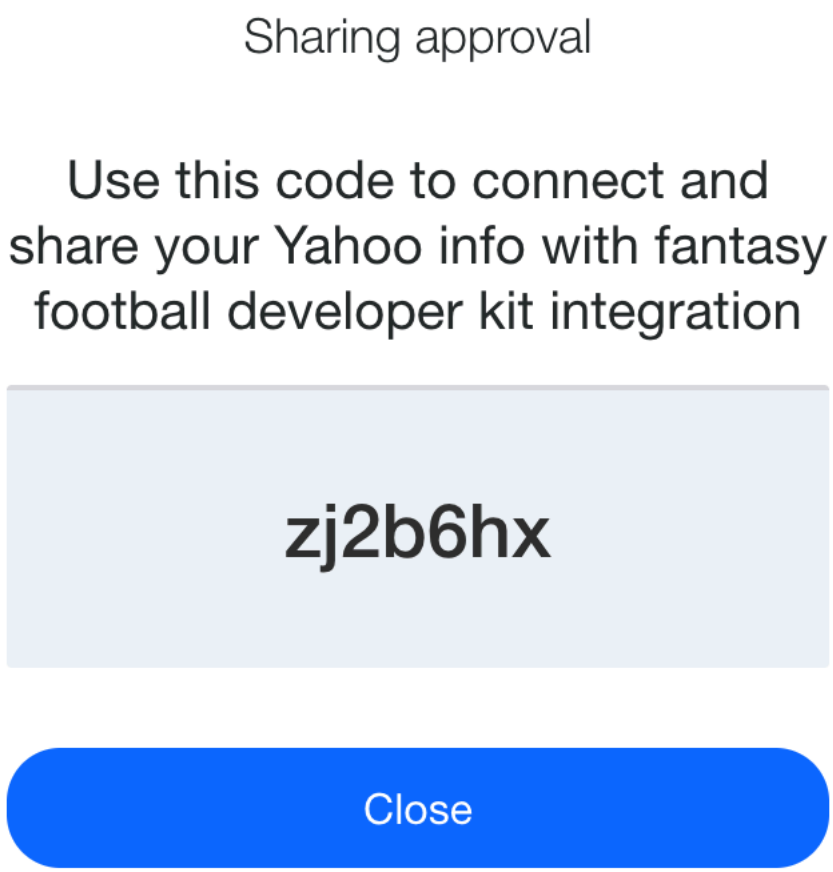

Figure 0.9: Yahoo Oauth Code

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": "DA0XXXXXXXXXXXXXXXXXXXXXXXXXXXgs94_IyynI-",
    "consumer_key": "
       dj0yJmk9VTZXXXXXXXXXXXXXXXXXXXXXXXXXXtZXJzZWNyZXQmc3Y9MCZ4PWVk",
    "consumer_secret": "56ec2XXXXXXXXXXX8aa1eff00c8",
    "guid": null,
    "refresh_token": "AG2MN2HnhjgvTzcisDnRSlXXXXXXXXXXXXXXX-",
    "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': '414',
    'game_id': '414',
    'name': 'Football',
    'code': 'nfl',
    'type': 'full',
    'url': 'https://football.fantasysports.yahoo.com/f1',
    'season': '2022',
    'is_registration_over': 0,
    'is_game_over': 0,
    'is_offseason': 0,
    'is_live_draft_lobby_active': 1}],
  'time': '23.262023925781ms',
  'copyright': 'Data provided by Yahoo! and STATS, LLC',
  '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 unfortunate part of this 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 = 43886
TEAM_ID = 11
WEEK = 1
```

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 (2022) the `game_id` for fantasy football is 414. We'll be going through some saved data from last year (2021), when the game id was 406 so that's what you'll see below.

The game_id goes in all the endpoints, along with `league_id` and `team_id`, like this:

`team/406.l.{LEAGUE_ID}.t.{TEAM_ID}`

or

`league/406.l.{LEAGUE_ID}` depending if we're querying info about a league or a team.

Roster Data

Let's start with the roster data, which is available by week under the endpoint:

```
In [6]:
team_url = ('https://fantasysports.yahooapis.com/fantasy/v2' +
            f'/team/406.l.{LEAGUE_ID}.t.{TEAM_ID}/roster;week={WEEK}')
```

Again, we have to call it with oauth:

```
In [7]: team_data = OAUTH.session.get(team_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 again:

```
OAUTH = OAuth2(None, None, from_file=YAHOO_FILE)
```

This is how you get live data for the current week. It's how you'll analyze your own team (make sure to use 414 instead of 406 for this year), 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 Yahoo API between Thursday and Sunday, Week 1, 2021.

We can load that data like this:

```
In [8]:
with open('./projects/integration/raw/yahoo/roster.json') as f:
    roster_json = json.load(f)
```

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:

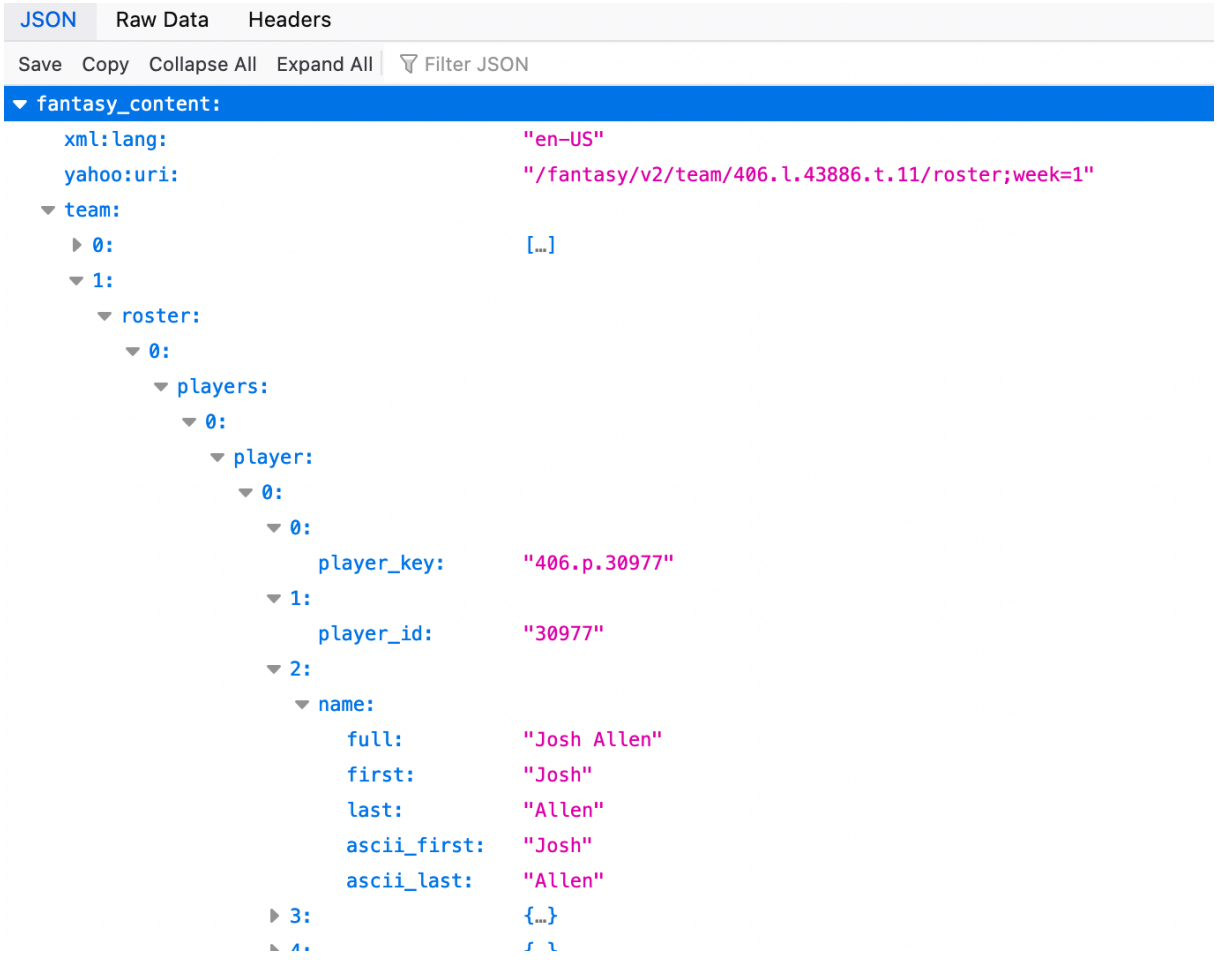

Figure 0.10: Yahoo Roster JSON

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': '406.p.30977'},
   {'player_id': '30977'},
   {'name': {'full': 'Josh Allen',
     'first': 'Josh',
     'last': 'Allen',
     'ascii_first': 'Josh',
     'ascii_last': 'Allen'}},
   {'editorial_player_key': 'nfl.p.30977'},
   {'editorial_team_key': 'nfl.t.2'},
   {'editorial_team_full_name': 'Buffalo Bills'},
   {'editorial_team_abbr': 'Buf'},
   {'bye_weeks': {'week': '7'}},
   {'uniform_number': '17'},
   {'display_position': 'QB'},
   ...
   {'is_undroppable': '0'},
   {'position_type': 'O'},
   {'primary_position': 'QB'},
   {'eligible_positions': [{'position': 'QB'}]},
   [],
   [],
   []],
  {'selected_position': [{'coverage_type': 'week', 'week': '1'},
    {'position': 'QB'},
    {'is_flex': 0}]},
  {'is_editable': 1}]}
```

It's Josh Allen. Nice.

I don't want to constantly complain about the Yahoo API, but these players are formatted strangely, as lists 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': '406.p.30977',
 'player_id': '30977',
 'name': {'full': 'Josh Allen',
  'first': 'Josh',
  'last': 'Allen',
  'ascii_first': 'Josh',
  'ascii_last': 'Allen'},
 'editorial_player_key': 'nfl.p.30977',
 'editorial_team_key': 'nfl.t.2',
 'editorial_team_full_name': 'Buffalo Bills',
 'editorial_team_abbr': 'Buf',
 'bye_weeks': {'week': '7'},
 'uniform_number': '17',
 'display_position': 'QB',
 'is_undroppable': '0',
 'position_type': 'O',
 'primary_position': 'QB',
 'eligible_positions': [{'position': 'QB'}]}
```

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': 30977,
 'name': 'Josh Allen',
 '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': '30977',
  'name': 'Josh Allen',
  'player_position': 'QB',
  'team_position': 'QB'},
 {'yahoo_id': '26650',
  'name': 'DeAndre Hopkins',
  'player_position': 'WR',
  'team_position': 'WR'},
 {'yahoo_id': '32719',
  'name': 'Chase Claypool',
  'player_position': 'WR',
  'team_position': 'WR'},
 {'yahoo_id': '30121',
  'name': 'Christian McCaffrey',
  'player_position': 'RB',
  'team_position': 'RB'},
 {'yahoo_id': '28514',
  'name': 'Mike Davis',
  'player_position': 'RB',
  'team_position': 'RB'},
 {'yahoo_id': '26658',
  'name': 'Zach Ertz',
  'player_position': 'TE',
  'team_position': 'TE'},
 {'yahoo_id': '33394',
  'name': 'Jaylen Waddle',
  'player_position': 'WR',
  'team_position': 'W/R/T'},
```

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      30977            Josh Allen              QB            QB
1      26650       DeAndre Hopkins              WR            WR
2      32719        Chase Claypool              WR            WR
3      30121   Christian McCaffrey              RB            RB
4      28514            Mike Davis              RB            RB
5      26658             Zach Ertz              TE            TE
6      33394         Jaylen Waddle              WR         W/R/T
7      24017        Rob Gronkowski              TE            BN
8      33423      Javonte Williams              RB            BN
9      31074          Nyheim Hines              RB            BN
10     33508   Rhamondre Stevenson              RB            BN
11     26060          Cole Beasley              WR            BN
12     33447  Terrace Marshall Jr.              WR            BN
13      7520          Robbie Gould               K             K
14    100011          Indianapolis             DEF           DEF
15     29281        Michael Thomas              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     26650  DeAndre Hopkins              WR            WR
2     32719   Chase Claypool              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
```

They 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      30977           Josh Allen              QB            QB
1      26650      DeAndre Hopkins              WR           WR1
2      32719       Chase Claypool              WR           WR2
3      30121  Christian McCaffrey              RB           RB1
4      28514          Mike Davis              RB           RB2
5      26658           Zach Ertz              TE            TE
6      33394        Jaylen Waddle              WR         W/R/T
7      24017       Rob Gronkowski              TE           BN1
8      33423     Javonte Williams              RB           BN2
9      31074         Nyheim Hines              RB           BN3
10     33508  Rhamondre Stevenson              RB           BN4
11     26060         Cole Beasley              WR           BN5
12     33447  Terrace Marshall Jr.             WR           BN6
13      7520         Robbie Gould               K             K
14    100011         Indianapolis             DEF           DEF
15     29281       Michael Thomas              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      30977            Josh Allen              QB            QB   True
1      26650       DeAndre Hopkins              WR           WR1   True
2      32719        Chase Claypool              WR           WR2   True
3      30121   Christian McCaffrey              RB           RB1   True
4      28514            Mike Davis              RB           RB2   True
5      26658             Zach Ertz              TE            TE   True
6      33394         Jaylen Waddle              WR         W/R/T   True
7      24017        Rob Gronkowski              TE           BN1  False
8      33423      Javonte Williams              RB           BN2  False
9      31074          Nyheim Hines              RB           BN3  False
10     33508   Rhamondre Stevenson              RB           BN4  False
11     26060          Cole Beasley              WR           BN5  False
12     33447  Terrace Marshall Jr.              WR           BN6  False
13      7520          Robbie Gould               K             K   True
14    100011          Indianapolis             DEF           DEF   True
15     29281        Michael Thomas              WR            IR  False
```

This is really close to what we want. We just need a team id, and the corresponding fantasymath 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 necessarily want to keep showing 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]: '11'
```

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.

```
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 ... team_position  start team_id
0      30977            Josh Allen ...            QB   True      11
1      26650       DeAndre Hopkins ...           WR1   True      11
2      32719        Chase Claypool ...           WR2   True      11
3      30121   Christian McCaffrey ...           RB1   True      11
4      28514           Mike Davis ...           RB2   True      11
5      26658            Zach Ertz ...            TE   True      11
6      33394         Jaylen Waddle ...         W/R/T   True      11
7      24017        Rob Gronkowski ...           BN1  False      11
8      33423      Javonte Williams ...           BN2  False      11
9      31074          Nyheim Hines ...           BN3  False      11
10     33508   Rhamondre Stevenson ...           BN4  False      11
11     26060          Cole Beasley ...           BN5  False      11
12     33447  Terrace Marshall Jr. ...           BN6  False      11
13      7520          Robbie Gould ...             K   True      11
14    100011          Indianapolis ...           DEF   True      11
15     29281        Michael Thomas ...            IR  False      11
```

What else? Well, this example snapshot of data we're using is from Friday morning, right after TB-DAL played Thursday night. In that case Gronk (TB) will have played and have an actual score. We want to get this data too.

After playing around with the API, those point values are in another endpoint.

```
In [1]:
points_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +
                f'team/406.l.{LEAGUE_ID}.t.{TEAM_ID}' +
            "/players;out=metadata,stats,ownership,percent_owned,
               draft_analysis")
```

We can get that data with our same `OAUTH` object:

```
In [2]:
points_json = OAUTH.session.get(points_url, params={'format': 'json'}).
    json()
```

Again, try this out and make sure it works. But we'll keep using the saved snapshot for this walk-through:

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 Gronk (the only player here who had played so far this week) is this one:

```
In [4]:
gronk = player_dict['1']
```

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 Gronk:

```
In [5]: process_player_stats(gronk)
Out[5]: {'yahoo_id': 24017, 'actual': 29.0}
```

And putting 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 (sort of anti-climactic since Gronk is the only one who played):

```
In [6]: stats
Out[6]:
    yahoo_id  actual
0       7520     NaN
1      24017    29.0
2      26060     NaN
3      26650     NaN
4      26658     NaN
5      28514     NaN
6      29281     NaN
7      30121     NaN
8      30977     NaN
9      31074     NaN
10     32719     NaN
11     33394     NaN
12     33423     NaN
13     33447     NaN
14     33508     NaN
15    100011     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_position  start team_id  actual
0     30977           Josh Allen ...            QB   True      11     NaN
1     26650      DeAndre Hopkins ...           WR1   True      11     NaN
2     32719       Chase Claypool ...           WR2   True      11     NaN
3     30121  Christian McCaffrey ...           RB1   True      11     NaN
4     28514          Mike Davis ...           RB2   True      11     NaN
5     26658           Zach Ertz ...            TE   True      11     NaN
6     33394        Jaylen Waddle ...         W/R/T   True      11     NaN
7     24017       Rob Gronkowski ...           BN1  False      11    29.0
8     33423     Javonte Williams ...           BN2  False      11     NaN
9     31074         Nyheim Hines ...           BN3  False      11     NaN
```

Sweet. So we have actual, mid week points. That leaves a fantasymath id. If you look at the fantasymath ids, most of them are just the players name, but lowercase and with a dash in the middle. So this would get you 90% of the way there:

```
In [1]: roster_df['name'].str.lower().str.replace(' ','-').head()
Out[1]:
0             josh-allen
1        deandre-hopkins
2         chase-claypool
3    christian-mccaffrey
4             mike-davis
```

But that doesn't always work, because there are duplicate names, nicknames, juniors and seniors 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 [2] from utilities import (
          LICENSE_KEY, generate_token, master_player_lookup)

In [3]:
token = generate_token(LICENSE_KEY)['token']
fantasymath_players = master_player_lookup(token)
```

As with the other data, I've saved a snapshot of this lookup here:

```
In [4]: fantasymath_players.head()
Out[4]:
    fantasymath_id position  fleaflicker_id    espn_id
0      matt-prater        K            4221    11122.0
1       colt-mccoy       QB            6625    13199.0
2     kyler-murray       QB           14664  3917315.0
3  chris-streveler       QB           15510  3040206.0
4    chase-edmonds       RB           13870  3119195.0
```

Now we can link this up with the roster from above and we'll be set.

```
In [5]:
roster_df_w_id = pd.merge(roster_df,
                          fantasymath_players[['fantasymath_id', 'yahoo_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 `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
- fantasymath-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/406.l.{league_id}.t.{team_id}/roster;week={week}
                      ')

    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/406.l.{LEAGUE_ID}.t.{TEAM_ID}' +
                "/players;out=metadata,stats,ownership,percent_owned,
                   draft_analysis")
    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[['fantasymath_id', 'yahoo_id']],
                              how='left').drop('yahoo_id', axis=1)

    return roster_df_w_id
```

Note: this function is actually calling `OAUTH.session.get` — not loading example JSON from a file

like we were doing above — and so returns live data for the current week. That's what we ultimately want (we're not writing these functions to work with example data) but just a heads up that your output may look different.

```
In [6]: my_roster
my_roster = get_team_roster(TEAM_ID, LEAGUE_ID, 1, fantasymath_players)

In [7]: my_roster
Out[7]:
                   name player_position  ... team_id       fantasymath_id
0             Josh Allen              QB  ...      11           josh-allen
1        DeAndre Hopkins              WR  ...      11      deandre-hopkins
2         Chase Claypool              WR  ...      11       chase-claypool
3    Christian McCaffrey              RB  ...      11  christian-mccaffrey
4             Mike Davis              RB  ...      11           mike-davis
5              Zach Ertz              TE  ...      11            zach-ertz
6          Jaylen Waddle              WR  ...      11        jaylen-waddle
7         Rob Gronkowski              TE  ...      11       rob-gronkowski
8       Javonte Williams              RB  ...      11    javonte-williams
9           Nyheim Hines              RB  ...      11         nyheim-hines
10   Rhamondre Stevenson              RB  ...      11  rhamondre-stevenson
11          Cole Beasley              WR  ...      11         cole-beasley
12  Terrace Marshall Jr.              WR  ...      11    terrace-marshall
13          Robbie Gould               K  ...      11         robbie-gould
14          Indianapolis             DEF  ...      11              ind-dst
15        Michael Thomas              WR  ...      11                  NaN
```

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`

Team data is available in the standings endpoint.

```
In [1]:
teams_url = ('https://fantasysports.yahooapis.com/fantasy/v2/' +
             f'league/406.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 in 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', '10',
          '11', 'count'])
```

So this is a single team:

```
In [6]: team0 = teams_dict['0']

In [7]: team0
Out[7]:
{'team': [[{'team_key': '406.l.43886.t.1'},
   {'team_id': '1'},
   {'name': 'Make It Rain'},
   [],
   {'url': 'https://football.fantasysports.yahoo.com/f1/43886/1'},
   {'team_logos': [{'team_logo': {'size': 'large',
       'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/
          t_s192sq/fantasy-logos/25185524702_4264e9b5ef.jpg'}}]},
   {'division_id': '3'},
   {'waiver_priority': 8},
   [],
   {'number_of_moves': 0},
   {'number_of_trades': 0},
   {'roster_adds': {'coverage_type': 'week',
     'coverage_value': '1',
     'value': '0'}},
   [],
   {'league_scoring_type': 'head'},
   [],
   [],
   {'has_draft_grade': 1,
    'draft_grade': 'C-',
    'draft_recap_url': 'https://football.fantasysports.yahoo.com/f1
       /43886/1/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': '719',
       'felo_tier': 'gold'}}]}],
  {'team_points': {'coverage_type': 'season', 'season': '2021', 'total': '
     '}},
  {'team_standings': {'rank': '',
    'outcome_totals': {'wins': 0, 'losses': 0, 'ties': 0, 'percentage': ''
       },
    'divisional_outcome_totals': {'wins': 0, 'losses': 0, 'ties': 0},
    'points_for': '0',
    'points_against': 0}}]}
```

Again, just like with players, each team is structured as a list of single item dicts. Last time 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': '406.l.43886.t.1',
 'team_id': '1',
 'name': 'Make It Rain',
 'url': 'https://football.fantasysports.yahoo.com/f1/43886/1',
 'team_logos': [{'team_logo': {'size': 'large',
    'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/
       t_s192sq/fantasy-logos/25185524702_4264e9b5ef.jpg'}}],
 'division_id': '3',
 'waiver_priority': 8,
 'number_of_moves': 0,
 'number_of_trades': 0,
 'roster_adds': {'coverage_type': 'week', 'coverage_value': '1', 'value':
    '0'},
 'league_scoring_type': 'head',
 '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': '719',
    'felo_tier': 'gold'}}]}
```

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']
    dict_to_return['division_id'] = team_dict['division_id']
    return dict_to_return

In [11]: process_team(team0)
Out[11]:
{'team_id': '1',
 'owner_id': '7MIKYOTUTFOVHTULSLCSDCD6CE',
 'owner_name': 'Tyler',
 'division_id': '3'}
```

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/406.l.{league_id}' +
                 ';out=metadata,settings,standings,scoreboard,teams,players
                    ,draftresults,transactions')

    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 division_id
      league_id
0        1  7MIKYOTUTFOVHTULSLCSDCD6CE          Tyler           3
   43886
1        2  IA4UIFKQF2VCY3GISWAL7TPDDI          Craig           3
   43886
2        3  UNHRXNC6ZOS7SIOXOCWF45T77Q      Hackstock           3
   43886
3        4  LRNTX6K2RRXDH5YZ2R5ULV54LU   Jeff Harding           2
   43886
4        5  NXNIOTHGRZ24RTJNSBUNUFD6XM     Matt Lyons           3
   43886
5        6  77UWSX5MXPDQ7PBKR6BQDCZFI4           Reed           2
   43886
6        7  Z4WTKHLRPMPBNFKAO3JAPB4BTA           Ryan           2
   43886
7        8  Y4DULOALVEDHOXTPRUBK74XEBA  Brandon Vonck           1
   43886
8        9  UMT6C4DM6K2QO56CWSYNOAQDEQ           Alex           1
   43886
9       10  UHU7HBFPLMCYSVQ56YRRXOBZ4Y           Dane           1
   43886
10      11  4FOCKGEBSXA6SH5NN3VFLNYRR4           Paul           1
   43886
11      12  TXDSERMBBTB5UV7IOI3EMWGQVY           Tony           2
   43886
```

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) 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, WEEK)

In [16]: league_rosters.sample(20)
Out[16]:
                     name   ... team_id         fantasymath_id
129          Tony Pollard   ...       9          tony-pollard
182         Jason Sanders   ...      12         jason-sanders
79       David Montgomery   ...       6       david-montgomery
5          T.J. Hockenson   ...       1           tj-hockenson
176     Leonard Fournette   ...      12      leonard-fournette
66            Jerry Jeudy   ...       5            jerry-jeudy
75          Lamar Jackson   ...       6          lamar-jackson
84          Mike Williams   ...       6       mike-williams-wr
154       DeAndre Hopkins   ...      11        deandre-hopkins
93           Alvin Kamara   ...       7           alvin-kamara
50         Gerald Everett   ...       4         gerald-everett
1           Davante Adams   ...       1          davante-adams
18           Najee Harris   ...       2           najee-harris
82    JuJu Smith-Schuster   ...       6    juju-smith-schuster
19         Antonio Gibson   ...       2         antonio-gibson
62           Stefon Diggs   ...       5           stefon-diggs
141  Clyde Edwards-Helaire   ...      10  clyde-edwards-helaire
156   Christian McCaffrey   ...      11    christian-mccaffrey
57        Ty'Son Williams   ...       4         tyson-williams
17          Ja'Marr Chase   ...       2           jamarr-chase
```

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/406.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': '2021-09-09',
  'week_end': '2021-09-13',
  'status': 'preevent',
  'is_playoffs': '0',
  'is_consolation': '0',
  'is_matchup_recap_available': 0,
  '0': {'teams': {'0': {'team': [[{'team_key': '406.l.43886.t.1'},
       {'team_id': '1'},
       {'name': 'Make It Rain'},
       [],
       {'url': 'https://football.fantasysports.yahoo.com/f1/43886/1'},
       {'team_logos': [{'team_logo': {'size': 'large',
           'url': 'https://yahoofantasysports-res.cloudinary.com/image/
              upload/t_s192sq/fantasy-logos/25185524702_4264e9b5ef.jpg'
              }}]},
       {'division_id': '3'},
       {'waiver_priority': 8},
       [],
       ...
       {'managers': [{'manager': {'manager_id': '2',
           'nickname': 'Craig',
           'guid': 'IA4UIFKQF2VCY3GISWAL7TPDDI',
           'image_url': 'https://s.yimg.com/ag/images/
              default_user_profile_pic_64sq.jpg',
           'felo_score': '858',
           'felo_tier': 'platinum'}}]}],
      {'win_probability': 0.47,
       'team_points': {'coverage_type': 'week', 'week': '1', 'total': '
          0.00'},
       'team_projected_points': {'coverage_type': 'week',
        'week': '1',
        'total': '128.04'}}]},
    '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': '406.l.43886.t.1'},
   {'team_id': '1'},
   {'name': 'Make It Rain'},
   [],
   {'url': 'https://football.fantasysports.yahoo.com/f1/43886/1'},
   {'team_logos': [{'team_logo': {'size': 'large',
       'url': 'https://yahoofantasysports-res.cloudinary.com/image/upload/
          t_s192sq/fantasy-logos/25185524702_4264e9b5ef.jpg'}}]},
   {'division_id': '3'},
   {'waiver_priority': 8},
   [],
   {'number_of_moves': 0},
   {'number_of_trades': 0},
   {'roster_adds': {'coverage_type': 'week',
     'coverage_value': '1',
     'value': '0'}},
   [],
   {'league_scoring_type': 'head'},
   [],
   [],
   {'has_draft_grade': 1,
    'draft_grade': 'C-',
    'draft_recap_url': 'https://football.fantasysports.yahoo.com/f1
       /43886/1/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': '719',
       'felo_tier': 'gold'}}]}],
  {'win_probability': 0.53,
   'team_points': {'coverage_type': 'week', 'week': '1', 'total': '0.00'},
   'team_projected_points': {'coverage_type': 'week',
    'week': '1',
    'total': '132.07'}}]}
```

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 [10]: matchup0_team0['team_id']
Out[10]: '11'

In [11]: matchup0_team1['team_id']
Out[11]: '9'

In [12]: matchup0['matchup']['week']
Out[12]: '1'
```

As always, let's put this in a function:

```
In [13]:
def process_matchup(matchup):
    team0 = yahoo_list_to_dict(matchup_0['matchup']['0']['teams']['0'], '
       team')
    team1 = yahoo_list_to_dict(matchup_0['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': '2', '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      2    1
1        1      5    2
2        1      3    3
3        1     10    4
4        1     12    5
5        1      9    6
6        1      8    7
7        1     11    8
8        1      7    9
9        1      4   10
10       1      6   11
11       1      8   12
12       1      2   13
13       1      5   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 over. 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 2021, week 1, team 2 vs team 7 we'd have:

```
In [16]: make_matchup_id(2021, 1, 2, 7)
Out[16]: 2021010207
```

And let's make sure switching the team ids around gives us the same thing:

```
In [17]: make_matchup_id(2021, 1, 7, 2)
Out[17]: 2021010207
```

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(
        2021, 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/406.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'] = 2021
    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      2    1  2021010102    2021
1         1      5    2  2021020105    2021
2         1      3    3  2021030103    2021
3         1     10    4  2021040110    2021
4         1     12    5  2021050112    2021
5         1      9    6  2021060109    2021
6         1      8    7  2021070108    2021
7         1     11    8  2021080111    2021
8         1      7    9  2021090107    2021
9         1      4   10  2021100104    2021
10        1      6   11  2021110106    2021
11        1      8   12  2021120108    2021
12        1      2   13  2021130102    2021
13        1      5   14  2021140105    2021
14        1      3   15  2021150103    2021
15        2      1    1  2021010102    2021
16        2      3    2  2021020203    2021
17        2      5    3  2021030205    2021
18        2      6    4  2021040206    2021
19        2      9    5  2021050209    2021
```

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        2    1  2021010102    2021
1        1        5    2  2021020105    2021
2        1        3    3  2021030103    2021
3        1       10    4  2021040110    2021
4        1       12    5  2021050112    2021
5        1        9    6  2021060109    2021
6        1        8    7  2021070108    2021
7        1       11    8  2021080111    2021
8        1        7    9  2021090107    2021
9        1        4   10  2021100104    2021
```

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        2    1  2021010102    2021
1        1        5    2  2021020105    2021
2        1        3    3  2021030103    2021
3        1       10    4  2021040110    2021
4        1       12    5  2021050112    2021
5        1        9    6  2021060109    2021
6        1        8    7  2021070108    2021
7        1       11    8  2021080111    2021
8        1        7    9  2021090107    2021
9        1        4   10  2021100104    2021
```

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 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 overflowing 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 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[
        rosters['start'] &
        rosters['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_ 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[
        rosters['start'] &
        rosters['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  ...  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 a 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:
                                    mean     wp  wrong  regret
pos      player
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.

7. 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.

Loading Rosters

The very first step is getting everyone's rosters.

Note: the league integration makes this *way* easier. That's why this project comes after it. Even if you haven't worked through it yet, you can still use the cleaned up versions of the code I provide to get your own team.

Our code integrates with Fleaflicker, Yahoo, ESPN and Sleeper. If you're on another league hosting platform, you'll have to get all the rosters and matchups yourself. This will be annoying, but I'm working on a template to make it a bit easier.

This section picks up up in `./code/projects/league_working.py`, right after the imports.

Re: the imports, note I'm using:

```
import hosts.fleaflicker as site
```

because my league is in Fleaflicker, but change this if you're using a different site. E.g.:

```
import hosts.espn as site
```

Before you run this with your own team, make sure you add your league using `./hosts/league_setup.py`. This file gets the data that doesn't change – teams, schedule, scoring settings – and writing them to a database. We'll be calling this database, so it's a requirement.

Let's start with our parameters. Because everything else is saved in the database, the only thing we need to know is our league id and week. For this example we'll work through analyzing my Fleaflicker league, and we'll pretend we're doing it on the Friday of Week 2, 2021 (I've saved some data snapshots from then that we'll use).

We're doing it mid-week because then we can work through handling the Thursday night results (WAS-NYG played Thursday in Week 2, 2021). Then we can update our analysis to take into account how these guys actually did.

(Note: we'll make it flexible re: handling guys who've already played. If we're running this on Tuesday or Wednesday, no one will have played and our analysis will be pure projection. We'll also be able to run it late Sunday afternoon after the day games have played, or Monday morning when everyone except MNF guys have played. Whatever.)

So here's what we're starting with:

```
In [1]:
LEAGUE_ID = 316893
WEEK = 2
```

We'll use that to get everyone out of our league info database (the `./hosts/league_setup.py` file, which you should run before analyzing your own teams, adds data to this DB). Normally you'd connect to it like this:

```
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')
```

Once we have our connection, we can get all the relevant data out of it using the handy `read_league` function we wrote earlier:

```
In [5]: teams = site.read_league('teams', LEAGUE_ID, conn)

In [6]: schedule = site.read_league('schedule', LEAGUE_ID, conn)

In [7]: league = site.read_league('league', LEAGUE_ID, conn)
```

Our `league` table has a few other parameters we'll need:

```
In [6]:
TEAM_ID = league.iloc[0]['team_id']
HOST = league.iloc[0]['host']
SCORING = {}
SCORING['qb'] = league.iloc[0]['qb_scoring']
SCORING['skill'] = league.iloc[0]['skill_scoring']
SCORING['dst'] = league.iloc[0]['dst_scoring']
```

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 [7]:
token = generate_token(LICENSE_KEY)['token']
player_lookup = master_player_lookup(token)

rosters = (site.get_league_rosters(player_lookup, LEAGUE_ID)
           .query("start"))
```

But again, *here*, I've saved the data so that we're seeing and working with the same thing:

```
In [8]: player_lookup = pd.read_csv(
              './projects/league/raw/player_lookup.csv')

In [9]: rosters = pd.read_csv(
              './projects/league/raw/rosters.csv').query("start")
```

We'll we're at it let's also grab the simulations for every starter. This is how you'd normally do it:

```
In [10]: available_players = get_players(token, **SCORING)

In [11]:
sims = get_sims(token, (set(rosters['fantasymath_id']) &
                set(available_players['fantasymath_id'])),
                nsims=1000, **SCORING)
```

But I've saved a copy of these too.

```
In [12]: sims = pd.read_csv('./projects/league/raw/sims.csv')
```

So it's Friday morning, Week 2, and NYG-WAS played the night before (WAS eaked out a victory 30-29). Here are the first few original, projected simulations of some NYG-WAS players:

```
In [13]:
sims[['terry-mclaurin', 'sterling-shepard', 'antonio-gibson', 'was-dst',
    'logan-thomas']].head()
Out[13]:
   terry-mclaurin  s-shepard  antonio-gibson    was-dst  logan-thomas
0        2.984904   1.916947       11.763663  10.940769      2.747043
1       10.872171   1.747126        9.898059  -1.124291      3.567560
2        3.852918  15.288476        5.610022   7.441676      5.591404
3       12.976616   5.124510       24.920291  16.489956      0.514987
4       10.915590   5.062061       10.026187  -0.461466      6.527563
```

And some descriptive stats on these:

```
In [14]:
sims[['terry-mclaurin', 'sterling-shepard', 'antonio-gibson', 'was-dst',
    'logan-thomas']].describe().round(2)
Out[14]:
       terry-mclaurin  s~shepard  a~gibson  was-dst  logan-thomas
count         1000.00    1000.00   1000.00  1000.00       1000.00
mean             8.77       6.72     12.98     9.47          5.93
std              6.46       5.19      7.95     5.45          4.54
min              0.00       0.01      0.00    -2.87          0.01
25%              3.72       2.62      6.93     5.63          2.36
50%              7.48       5.72     11.95     9.12          5.06
75%             12.37       9.76     17.75    13.29          8.54
max             39.53      31.93     40.47    26.04         24.31
```

So going into the week, these were the projections; we had simulations of their distributions. But since they actually played, we now *know* how they did — we actually got the real draw from the distribution:

```
In [15]: players_w_pts = rosters.loc[rosters['actual'].notnull(),
                                     ['fantasymath_id', 'actual']]

In [16]: players_w_pts
Out[16]:
       fantasymath_id  actual
5    sterling-shepard     8.5
33     antonio-gibson     7.3
37     terry-mclaurin    16.7
56            was-dst     8.0
102     logan-thomas     4.5
```

Aside: How "good" were these performances?

Feel free to skip this, but one thing we can do with the actual results + original distributions is see how "good" of a draw each player got as a percentile. We just check how many of the simulations were below the actual score:

So for Sterling Shepard it'd be:

```
In [17]: (sims['sterling-shepard'] <= 8.5).mean()
Out[17]: 0.691
```

A 69% percentile draw. Nice. It might be fun to calculate all of these quick:

```
In [18]:
percentiles = [(sims[player] <= pts).mean() for player, pts in
    zip(players_w_pts['fantasymath_id'], players_w_pts['actual'])]

In [19]: players_w_pts['pctile'] = percentiles

In [20]: players_w_pts
Out[20]:
       fantasymath_id  actual  pctile
5    sterling-shepard     8.5   0.691
33     antonio-gibson     7.3   0.266
37     terry-mclaurin    16.7   0.875
56           was-dst     8.0   0.411
102      logan-thomas     4.5   0.450
```

It's a small sample size, but we can see the average percentile is close to 0.50, which is the sign of good projections.

```
In [21]: players_w_pts['pctile'].mean()
Out[21]: 0.5386
```

Updating the sims with actual scores

Back to our league analyzer code. Once we know actual scores, we don't really care about projections. And we can update our sims with actual values:

```
In [22]:
for player, pts in zip(players_w_pts['fantasymath_id'],
                       players_w_pts['actual']):
    sims[player] = pts
```

Now we have sims for the guys that haven't played yet, actual scores for the guys that have:

```
In [23]: (sims[['patrick-mahomes', 'davante-adams', 'sterling-shepard']]
          .head(5))
Out[23]:
   patrick-mahomes  davante-adams  sterling-shepard
0        12.198036      12.929617               8.5
1         6.976300      41.534314               8.5
2        20.131652      21.784214               8.5
3        37.128334      17.841323               8.5
4         4.274901      18.453803               8.5
```

So now we have what we need. Now let's get into building out the 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?
- Whats 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    1605148   1603352    49030929    2021     2     316893
7    1605151   1605147    49030927    2021     2     316893
8    1605154   1605149    49030925    2021     2     316893
9    1605155   1605156    49030926    2021     2     316893
10   1605152   1605157    49030930    2021     2     316893
11   1605153   1664196    49030928    2021     2     316893
```

This is pretty impersonal, but we can add some owner/team name info later.

Let's start with the this matchup, 1603356 (team 1) vs 1605155 (team 2).

```
In [3]:
team1 = 1605156
team2 = 1605155
```

We can get a list of team 1's players with

```
In [4]: rosters.query(f"team_id == {team1}")['fantasymath_id']
Out[4]:
16              jalen-hurts
17    clyde-edwards-helaire
18             najee-harris
19             myles-gaskin
20            antonio-brown
21            davante-adams
22            george-kittle
23            greg-zuerlein
24                   no-dst
```

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"team_id == {team_id}")['fantasymath_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]:
   jalen-hurts  clyde-edwards-helaire    ...  greg-zuerlein     no-dst
0     8.175596              20.357816    ...      11.700000  18.367046
1    27.380946              20.299723    ...       3.213830   7.266714
2    18.985869               6.694172    ...       9.467722   0.824209
3     5.505847              16.468139    ...       5.931216   6.964089
4    10.629754               5.429916    ...       9.913185   3.307983
```

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  123.097888   73.369188
1  132.628800   97.836765
2   85.785571  107.543537
3   97.006581  107.132176
4   78.964914   95.545747
5   96.756349   81.297984
6   71.958795   68.891494
7   96.745198   80.460234
8   85.796623   84.520045
9  113.278066   92.034510
```

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]: 0.553
```

Team 1 has a 55.3% of winning.

What's the betting line (how much does team 1 usually win by)?

```
In [12]: line = (team1_pts - team2_pts).median()

In [13]: line
Out[13]: 3.6738088222868583
```

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]: 3.5
```

The over-under is also easy:

```
In [16]: over_under = (team1_pts + team2_pts).median()

In [17]: over_under
Out[17]: 184.12988574498107
```

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().round(2)
    winprob_b = 1 - winprob_a.round(2)

    # get over-under
    over_under = (total_a + total_b).median().round(2)

    # line
    line = (total_a - total_b).median().round(2)
    line = round(line*2)/2

    return {'wp_a': winprob_a, 'wp_b': winprob_b,
            'over_under': over_under,
            'line': line}
```

A quick check to make sure it's working as expected on the matchup we've been looking at:

```
In [18]: summarize_matchup(sims[team1_roster], sims[team2_roster])
Out[18]: {'wp_a': 0.55, 'wp_b': 0.45, 'over_under': 184.13, 'line': 3.5}
```

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 [19]: matchup_list = []

In [20]:
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  line   team_a   team_b
0  0.46  0.54      182.52  -3.5  1605148  1603352
1  0.35  0.65      185.32 -12.5  1605151  1605147
2  0.44  0.56      187.13  -3.5  1605154  1605149
3  0.45  0.55      184.13  -3.5  1605155  1605156
4  0.51  0.49      187.26   1.0  1605152  1605157
5  0.54  0.46      181.44   2.5  1605153  1664196
```

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  line         team_a          team_b
0  0.46  0.54      182.52  -3.5   Lzrlightshow        UnkleJim
1  0.35  0.65      185.32 -12.5        Edmundo       carnsc815
2  0.44  0.56      187.13  -3.5         ccarns    Plz-not-last
3  0.45  0.55      184.13  -3.5         BRUZDA           nbraun
4  0.51  0.49      187.26   1.0      WesHeroux  JBKBDomination
5  0.54  0.46      181.44   2.5       LisaJean     MegRyan0113
```

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]:
{1605154: 'ccarns',
 1605149: 'Plz-not-last',
 1605156: 'nbraun',
 1605155: 'BRUZDA',
 1605147: 'carnsc815',
 1605151: 'Edmundo',
 1605153: 'LisaJean',
 1664196: 'MegRyan0113',
 1603352: 'UnkleJim',
 1605157: 'JBKBDomination',
 1605148: 'Lzrlightshow',
 1605152: 'WesHeroux'}

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  line        team_a          team_b
0  0.46  0.54      182.52  -3.5  Lzrlightshow        UnkleJim
1  0.35  0.65      185.32 -12.5       Edmundo       carnsc815
2  0.44  0.56      187.13  -3.5         ccarns    Plz-not-last
3  0.45  0.55      184.13  -3.5         BRUZDA           nbraun
4  0.51  0.49      187.26   1.0     WesHeroux  JBKBDomination
5  0.54  0.46      181.44   2.5      LisaJean     MegRyan0113
```

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 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     Lzrlightshow  0.46        UnkleJim
1          Edmundo  0.35       carnsc815
2           ccarns  0.44    Plz-not-last
3           BRUZDA  0.45          nbraun
4        WesHeroux  0.51  JBKBDomination
5         LisaJean  0.54     MegRyan0113
6         UnkleJim  0.54    Lzrlightshow
7        carnsc815  0.65         Edmundo
8     Plz-not-last  0.56          ccarns
9           nbraun  0.55          BRUZDA
10  JBKBDomination  0.49       WesHeroux
11     MegRyan0113  0.46        LisaJean
```

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': 'carnsc815', 'wp': 0.65, 'opp': 'Edmundo'}
```

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': 0.51,
 'wp_b': 0.49,
 'over_under': 187.26,
 'line': 1.0,
 'team_a': 'WesHeroux',
 'team_b': 'JBKBDomination'}
```

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': 0.54,
 'wp_b': 0.46,
 'over_under': 181.44,
 'line': 2.5,
 'team_a': 'LisaJean',
 'team_b': 'MegRyan0113'}
```

Nice. This is definitely something my league would be interested in (especially the guys who are favored).

Maybe if we're lucky, we'll even get offers to bet from people who feel like the model is disrespecting their chances.

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]:
   jalen-hurts  clyde~helaire  ...  greg-zuerlein     no-dst
0     8.175596      20.357816  ...      11.700000  18.367046
1    27.380946      20.299723  ...       3.213830   7.266714
2    18.985869       6.694172  ...       9.467722   0.824209
3     5.505847      16.468139  ...       5.931216   6.964089
4    10.629754       5.429916  ...       9.913185   3.307983
```

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       94.358462
std        21.429908
min        32.450198
5%         59.809317
25%        79.736586
50%        92.985388
75%       108.377577
95%       132.451397
max       177.682559
```

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
1605154  92.29  18.46  63.30  79.84  91.22  104.57  123.03
1605149  95.86  16.71  70.10  84.35  95.35  106.66  122.99
1605156  94.36  21.43  59.81  79.74  92.99  108.38  132.45
1605155  90.64  17.60  62.85  78.77  89.78  101.95  121.50
1605147  98.88  20.46  67.70  85.12  98.01  112.10  133.32
1605151  87.29  18.69  57.11  74.20  85.98   99.66  118.68
1605153  91.99  19.60  61.34  78.79  91.85  104.11  124.60
1664196  89.16  19.42  58.54  76.37  88.24  101.56  121.38
1603352  93.79  20.01  62.09  79.88  92.36  107.31  127.44
1605157  93.21  19.45  62.71  79.47  92.13  105.64  123.92
1605148  89.75  19.73  58.29  76.70  89.24  103.22  123.38
1605152  94.97  21.11  62.98  79.97  94.18  107.75  131.62
```

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      90.106325
1      97.377634
2     100.799167
3      99.320932
4      68.293707
..           ...
995    95.011760
996   107.251532
997    69.680794
998   115.732468
999   101.827744
```

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]:
      1605154     1605149   ...    1605157     1605148    1605152
0  116.137038   96.448165   ...  98.296715   50.480420  84.379955
1   97.064857   82.703776   ...  72.011108  113.430565  76.420759
2   89.911590   86.380492   ...  98.900584  108.227834  93.844186
3  103.628871   81.962210   ...  84.197711   84.200290  90.066321
4   95.385004  111.911831   ...  79.612978  106.749071  94.641096
```

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    134.313368
1    132.628800
2    118.438516
3    125.180078
4    108.881072
```

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    1605147
1    1605156
2    1664196
3    1605147
4    1605147
```

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    0.133
1605152    0.121
1605156    0.110
1603352    0.093
1605157    0.089
1605149    0.075
1605153    0.074
1605154    0.065
1664196    0.063
1605148    0.063
1605151    0.062
1605155    0.052
```

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
1605154  92.294145   0.065  0.078
1605149  95.860803   0.075  0.032
1605156  94.358462   0.110  0.095
1605155  90.644152   0.052  0.081
1605147  98.880858   0.133  0.050
1605151  87.285438   0.062  0.121
1605153  91.986236   0.074  0.092
1664196  89.160391   0.063  0.105
1603352  93.793750   0.093  0.064
1605157  93.212979   0.089  0.091
1605148  89.749512   0.063  0.123
1605152  94.971370   0.121  0.068
```

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    125.919563    61.998382
std      13.053718    10.130569
min      93.789422    26.266730
5%      107.395360    44.493004
25%     117.015151    55.412900
50%     124.776572    62.714562
75%     134.348385    68.765591
95%     147.965967    78.439357
max     186.309797    92.425998
```

About 125 and 62, with a standard deviation of 13 and 10.

Our final team_df:

```
In [26]: team_df.round(2)
Out[26]:
          mean    std     5%    25%    50%     75%     95%  p_high  p_low
team_id
1605154  90.90  18.56  61.05  78.42  91.07  102.77  121.68    0.06   0.09
1605149  94.41  17.09  66.61  82.97  94.46  105.32  122.78    0.07   0.04
1605156  93.53  21.36  60.05  78.27  93.22  107.14  129.61    0.11   0.09
1605155  90.81  17.75  62.39  79.46  89.96  101.84  120.97    0.06   0.08
1605147  99.95  20.98  67.09  85.23  99.16  113.24  135.31    0.14   0.04
1605151  87.66  18.54  59.45  74.39  86.91  100.69  119.28    0.05   0.11
1605153  90.88  22.00  58.26  74.24  88.93  105.64  127.92    0.08   0.11
1664196  90.54  19.69  57.50  76.82  89.93  104.82  123.61    0.08   0.10
1603352  93.49  19.39  63.02  80.66  92.31  106.84  126.00    0.09   0.07
1605157  92.37  19.64  61.91  78.41  91.37  105.51  125.68    0.08   0.08
1605148  89.97  20.15  58.33  75.43  89.25  102.61  124.27    0.07   0.11
1605152  94.35  21.15  60.43  79.70  93.95  107.99  131.63    0.10   0.08
```

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 14 points.

But for each team, that 50% probability range is bigger:

```
In [27]: team_df['75%'] - team_df['25%']
Out[27]:
team_id
1605154    24.736821
1605149    22.302939
1605156    28.640991
1605155    23.174908
1605147    26.981580
1605151    25.455981
1605153    25.319253
1664196    25.188301
1603352    27.426213
1605157    26.165086
1605148    26.518842
1605152    27.778166
```

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
ccarns          92.29  18.46  63.30  ... 123.03    0.06   0.08
Plz-not-last    95.86  16.71  70.10  ... 122.99    0.08   0.03
nbraun          94.36  21.43  59.81  ... 132.45    0.11   0.10
BRUZDA          90.64  17.60  62.85  ... 121.50    0.05   0.08
carnsc815       98.88  20.46  67.70  ... 133.32    0.13   0.05
Edmundo         87.29  18.69  57.11  ... 118.68    0.06   0.12
LisaJean        91.99  19.60  61.34  ... 124.60    0.07   0.09
MegRyan0113     89.16  19.42  58.54  ... 121.38    0.06   0.10
UnkleJim        93.79  20.01  62.09  ... 127.44    0.09   0.06
JBKBDomination  93.21  19.45  62.71  ... 123.92    0.09   0.09
Lzrlightshow    89.75  19.73  58.29  ... 123.38    0.06   0.12
WesHeroux       94.97  21.11  62.98  ... 131.62    0.12   0.07
```

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'{HOST}_{LEAGUE_ID}_2021-{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 - 2021
**********************************

   wp_a  wp_b  over_under  line        team_a          team_b
0  0.46  0.54      182.52  -3.5  Lzrlightshow        UnkleJim
1  0.35  0.65      185.32 -12.5       Edmundo       carnsc815
2  0.44  0.56      187.13  -3.5         ccarns    Plz-not-last
3  0.45  0.55      184.13  -3.5         BRUZDA          nbraun
4  0.51  0.49      187.26   1.0     WesHeroux  JBKBDomination
5  0.54  0.46      181.44   2.5       LisaJean     MegRyan0113

*******************************
Team Projections, Week 2 - 2021
*******************************

                 mean    std     5%    25%    50%     75%     95%  ...
owner_name                                                         ...
carnsc815       98.88  20.46  67.70  85.12  98.01  112.10  133.32  ...
Plz-not-last    95.86  16.71  70.10  84.35  95.35  106.66  122.99  ...
WesHeroux       94.97  21.11  62.98  79.97  94.18  107.75  131.62  ...
nbraun          94.36  21.43  59.81  79.74  92.99  108.38  132.45  ...
UnkleJim        93.79  20.01  62.09  79.88  92.36  107.31  127.44  ...
JBKBDomination  93.21  19.45  62.71  79.47  92.13  105.64  123.92  ...
ccarns          92.29  18.46  63.30  79.84  91.22  104.57  123.03  ...
LisaJean        91.99  19.60  61.34  78.79  91.85  104.11  124.60  ...
BRUZDA          90.64  17.60  62.85  78.77  89.78  101.95  121.50  ...
Lzrlightshow    89.75  19.73  58.29  76.70  89.24  103.22  123.38  ...
MegRyan0113     89.16  19.42  58.54  76.37  88.24  101.56  121.38  ...
Edmundo         87.29  18.69  57.11  74.20  85.98   99.66  118.68  ...

Lock of the week:
carnsc815 over Edmundo — 0.65

Photo-finish of the week::
WesHeroux vs JBKBDomination, 0.51-0.49

Most unexciting game of the week:
LisaJean vs MegRyan0113, 181.44
```

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]:
      1605154     1605149   ...     1605157     1605148     1605152
0   59.221007  106.051605   ...   93.949385   66.647801  106.640664
1   76.839322   93.105304   ...   94.272992   56.566394   77.990600
2   85.482300  100.411518   ...   83.474652   82.413038   80.897909
3   96.384913  103.148027   ...  117.189595  122.391445   86.194904
4  100.850229   93.183648   ...   95.249319   98.004800   76.431007
```

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  1605154    100.890141
   1605149    119.204156
   1605156     77.762378
   1605155     72.182029
   1605147     97.827775
dtype: float64
```

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()
Out[4]:
   sim  team_id         pts
0    0  1605154   79.839403
1    0  1605149  118.940070
2    0  1605156   79.758579
3    0  1605155  104.255020
4    0  1605147   94.963861
```

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}.png'),
              bbox_inches='tight', dpi=500)
```

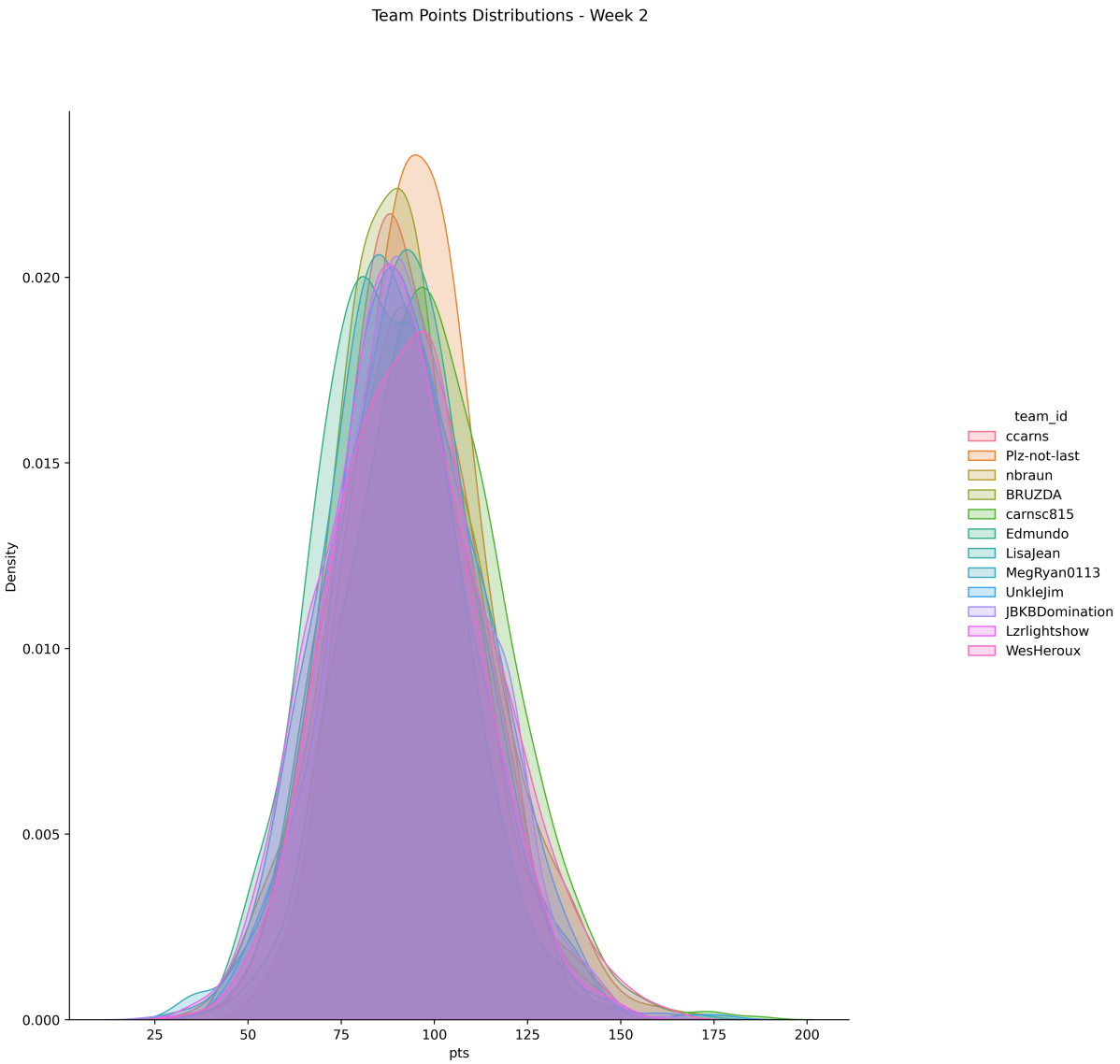

Figure 0.1: Team Distributions, Week 2 2021

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   1605148  1603352    49030929    2021     2     316893
7   1605151  1605147    49030927    2021     2     316893
8   1605154  1605149    49030925    2021     2     316893
9   1605155  1605156    49030926    2021     2     316893
10  1605152  1605157    49030930    2021     2     316893
11  1605153  1664196    49030928    2021     2     316893
90  1603352  1605148    49030929    2021     2     316893
91  1605147  1605151    49030927    2021     2     316893
92  1605149  1605154    49030925    2021     2     316893
93  1605156  1605155    49030926    2021     2     316893
94  1605157  1605152    49030930    2021     2     316893
95  1664196  1605153    49030928    2021     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  1605154   59.221007    49030925
1    1  1605154   76.839322    49030925
2    2  1605154   85.482300    49030925
3    3  1605154   96.384913    49030925
4    4  1605154  100.850229    49030925
```

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.png'),
               bbox_inches='tight', dpi=500)
```

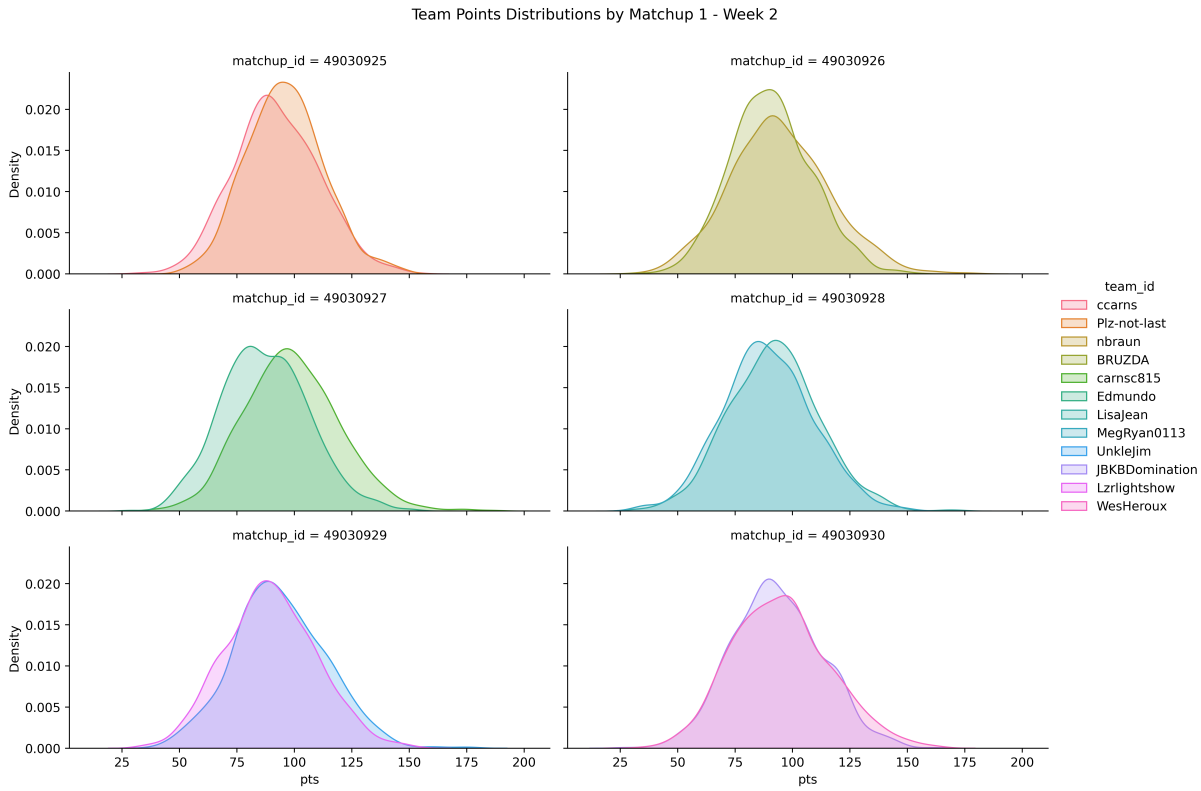

Figure 0.2: Team Distributions by Matchup, Week 2 2021

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)
```

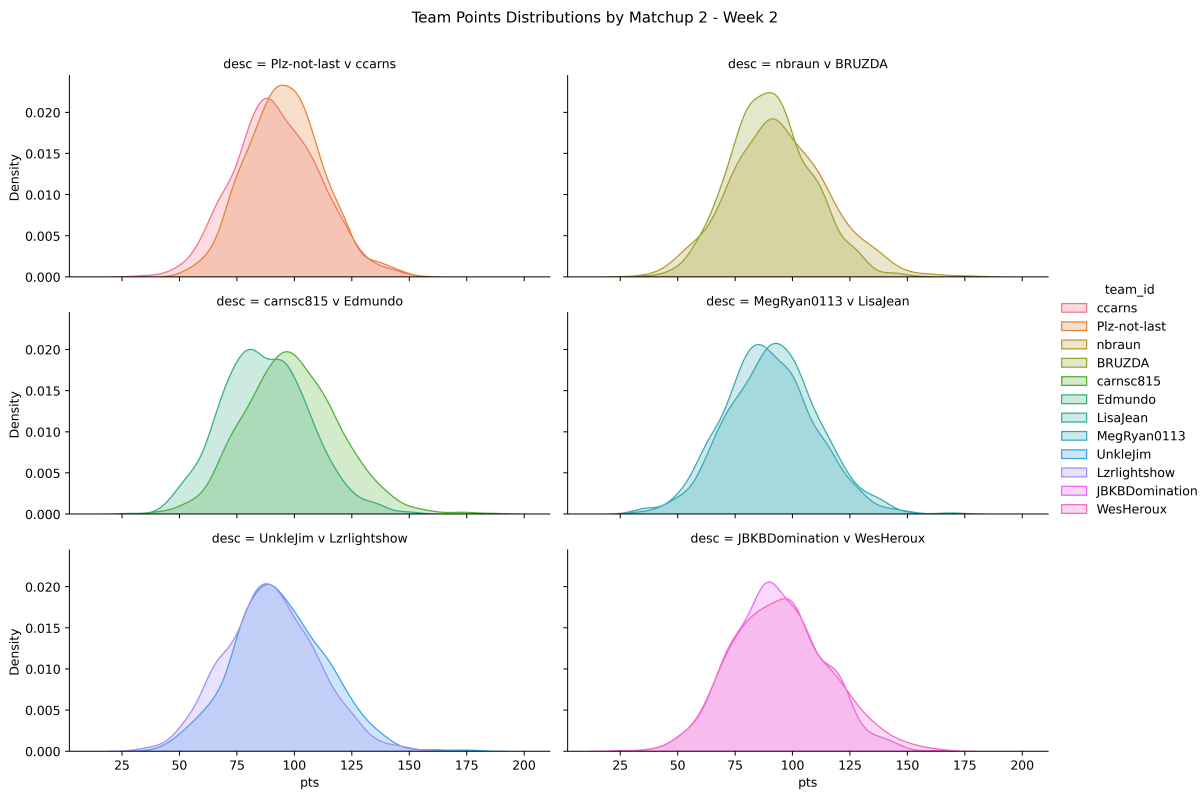

Figure 0.3: Team Distributions by Matchup, with Description, Week 2 2021

And there we go. See `./league.py` for the cleaned up version.

8. Project #4: Best Ball Projector

Best Ball Leagues

Best ball leagues have become more 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 computer 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 was in last year was 19 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 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`. A final version you can use with minimal tweaking is in `./projects/bestball/bb_final.py`.

We'll pick up right after the imports.

Let's start with some hardcoded data. The season, week and roster come from a bestball league I did a few years ago:

```
In [1]:
WEEK = 1
SEASON = 2019

SCORING = {}

roster = Series(['drew-brees', 'baker-mayfield', 'malcolm-brown',
                 'ezekiel-elliott', 'nyheim-hines', 'marlon-mack',
                 'devin-singletary', 'james-white', 'mike-evans',
                 'marquise-goodwin', 'mecole-hardman', 'christian-kirk',
                 'anthony-miller', 'dj-moore', 'curtis-samuel', 'noah-fant
                    ',
                 'george-kittle', 'cin-dst', 'den-dst'])
                 .to_frame('fantasymath_id')
```

We'll start by getting our token and querying simulations for everyone.

```
In [2]:
token = generate_token(LICENSE_KEY)['token']
sims = query_sims(token, list(roster['fm_id']), week=WEEK, season=SEASON,
                      nsims=1000, **SCORING)
```

It'd also be helpful to have the positions of these guys, which we can get from the `query_players` function.

```
In [3]:
players = query_players(token, week=WEEK, season=SEASON, **SCORING)
```

As usual, for walkthrough purposes I've saved snapshots of the data as CSVs. Let's load them so we're on the same page:

```
In [4]: sims = pd.read_csv('./projects/bestball/sims.csv')

In [5]: players = pd.read_csv('./projects/bestball/players.csv')
```

Great. Now we can use `players` to attach position to our roster:

```
In [6]: roster = pd.merge(roster, players)

In [7]: roster.head()
Out[7]:
     fantasymath_id position  actual
0    baker-mayfield       QB    7.76
1   ezekiel-elliott       RB   13.30
2      nyheim-hines       RB    5.70
3  devin-singletary       RB   14.80
4        mike-evans       WR    4.80
```

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 [8]:
pos_dict = {pos: list(roster
                      .query(f"position == '{pos.upper()}'")['fm_id']
                      .values) for pos in ['qb', 'rb', 'wr', 'te', 'dst']}
--

In [9]: pos_dict
Out[9]:
{'qb': ['drew-brees', 'baker-mayfield'],
 'rb': ['malcolm-brown',
  'ezekiel-elliott',
  'nyheim-hines',
  'marlon-mack',
  'devin-singletary',
  'james-white'],
 'wr': ['mike-evans',
  'marquise-goodwin',
  'mecole-hardman',
  'christian-kirk',
  'anthony-miller',
  'dj-moore',
  'curtis-samuel'],
 'te': ['noah-fant', 'george-kittle'],
 'dst': ['cin-dst', 'den-dst']}
```

OK. The goal is to project our best ball roster. That means looking at the projected 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 [10]: sims[pos_dict['qb']].max(axis=1).head()
Out[10]:
0    17.000284
1    27.386521
2    15.652178
3    23.152408
4    16.573342
```

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]: sims[pos_dict['qb']].idxmax(axis=1).head()
Out[8]:
0        drew-brees
1        drew-brees
2    baker-mayfield
3    baker-mayfield
4        drew-brees
```

Not bad. 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, so 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:

```
In [9]: sim = sims.iloc[0]

In [10]: sim
Out[10]:
drew-brees           17.000284
baker-mayfield       14.058540
malcolm-brown         6.983456
ezekiel-elliott      19.747401
james-white           8.245084
devin-singletary     13.145362
marlon-mack           7.203041
nyheim-hines          4.116764
dj-moore             11.721506
curtis-samuel        11.495201
christian-kirk       10.825759
mecole-hardman       11.022353
anthony-miller        3.958863
mike-evans           30.508368
marquise-goodwin      2.839749
noah-fant             8.616039
george-kittle        15.412129
cin-dst               8.334028
den-dst               9.854859
```

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]:
malcolm-brown         6.983456
ezekiel-elliott      19.747401
nyheim-hines          4.116764
marlon-mack           7.203041
devin-singletary     13.145362
james-white           8.245084
```

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]:
ezekiel-elliott      19.747401
devin-singletary     13.145362
```

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]:
ezekiel-elliott     19.747401
devin-singletary    13.145362

In [14]: n_highest_scores_from_sim1(sim, pos_dict['wr'], 3)
Out[14]:
mike-evans        30.508368
dj-moore          11.721506
curtis-samuel     11.495201
```

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 it's index, as you can see here.

```
for i, row in sims.head().iterrows():
    print(i)
    print(row)
```

Note: 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 [15]:
pd.concat([n_highest_scores_from_sim1(row, pos_dict['rb'], 2) for _, row
                in sims.head().iterrows()], ignore_index=True)
--
Out[15]:
0    19.747401
1    13.145362
2    15.993180
3    12.047454
4    23.104044
5    21.245833
6    23.931641
7    15.451015
8    19.578672
9    18.775539
```

It's doing what we want (note the first two numbers, 19.74 and 13.14 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 it's 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   ezekiel-elliott  19.747401     1    0
1  devin-singletary  13.145362     2    0
```

Which would be a lot more informative if we 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 takes a while to run, but when it's done it gives us:

```
In [18]: rbs.head()
Out[18]:
               name     points  rank  sim
0   ezekiel-elliott  19.747401     1    0
1  devin-singletary  13.145362     2    0
2   ezekiel-elliott  15.993180     1    1
3       james-white  12.047454     2    1
4       marlon-mack  23.104044     1    2
```

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          1          2
sim
0     ezekiel-elliott  devin-singletary  19.747401  13.145362
1     ezekiel-elliott       james-white  15.993180  12.047454
2         marlon-mack   ezekiel-elliott  23.104044  21.245833
3     ezekiel-elliott       marlon-mack  23.931641  15.451015
4     ezekiel-elliott       james-white  19.578672  18.775539
```

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    ezekiel-elliott  devin-singletary   19.747401   13.145362
1    ezekiel-elliott       james-white   15.993180   12.047454
2        marlon-mack   ezekiel-elliott   23.104044   21.245833
3    ezekiel-elliott       marlon-mack   23.931641   15.451015
4    ezekiel-elliott       james-white   19.578672   18.775539
```

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]: 34.186059191820235
```

Or answer questions like, how often is Zeke our RB1?

```
In [25]: rbs_wide['rb1_name'].value_counts(normalize=True)

Out[25]:
ezekiel-elliott     0.501
james-white         0.202
marlon-mack         0.155
devin-singletary    0.100
malcolm-brown       0.036
nyheim-hines        0.006
```

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(sims, 'wr', 3)

In [27]: wrs_wide.head()
Out[27]:
           wr1_name        wr2_name ... wr2_points  wr3_points
sim
0        mike-evans        dj-moore ...   11.721506   11.495201
1        mike-evans  anthony-miller ...   11.799630    5.867464
2          dj-moore   curtis-samuel ...   19.800464   14.641024
3    anthony-miller      mike-evans ...   24.447368   10.562063
4          dj-moore      mike-evans ...   20.463594   20.046970
```

Let's do it with other positions while we're at it, just so everything is in the same format:

```
qb_wide = top_n_by_pos(sims, 'qb', 1)
te_wide = top_n_by_pos(sims, 'te', 1)
dst_wide = top_n_by_pos(sims, 'dst', 1)
```

Now we can sum up and analyze our points.

```
In [28]:
# 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 [29]: proj_points.describe()
Out[29]:
count    1000.000000
mean      132.226784
std        21.117756
min        71.780262
25%       117.672590
50%       131.504131
75%       145.231923
max       221.842670
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 player NOT in the above.

There are a couple ways you could do this, and in real life it's entirely possible you start your way down a dead-end before stumbling on something that works.

But 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. We don't need rank anymore either.

Now let's apply it to all the flex eligible players (RB/WR/TE):

```
In [30]:
rbs_leftover = pd.concat([leftover_from_sim(row, pos_dict['rb'], 2)
                  for _, row in sims.iterrows()], ignore_index=True)
wrs_leftover = pd.concat([leftover_from_sim(row, pos_dict['wr'], 3)
                  for _, row in sims.iterrows()], ignore_index=True)
tes_leftover = pd.concat([leftover_from_sim(row, pos_dict['te'], 1)
                  for _, row in sims.iterrows()], ignore_index=True)
```

Then stick them together:

```
In [31]:
leftovers = pd.concat([rbs_leftover, wrs_leftover, tes_leftover],
                      ignore_index=True)
```

So for sim `0`, the left over guys are:

```
In [32]: leftovers.query("sim == 0")
Out[32]:
                  name     points  sim
0          james-white   8.245084    0
1          marlon-mack   7.203041    0
2        malcolm-brown   6.983456    0
3         nyheim-hines   4.116764    0
4000    mecole-hardman  11.022353    0
4001    christian-kirk  10.825759    0
4002    anthony-miller   3.958863    0
4003  marquise-goodwin   2.839749    0
8000         noah-fant   8.616039    0
```

And now we need to get the top scoring guy from each sim.

This calls for doing a `groupby` by sim.

If we were just interested in the points, we could do something like `max`:

```
In [33]: leftovers.groupby('sim').max()['points'].head()
Out[33]:
sim
0    11.022353
1     6.507203
2    18.966331
3    11.511227
4     9.872176
```

We can see the highest point value for sim 0 is Mecole Hardman's 11.02 points, which is 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 [34]: max_points_index = leftovers.groupby('sim').idxmax()['points']

In [34]: max_points_index.head()
sim
0    4000
1       4
2       8
3      12
4    4016
```

By itself, this index isn't useful, but we can pass it to our `leftovers` data with `loc` like this:

```
In [35]: leftovers.loc[max_points_index].head()
Out[35]:
                name     points  sim
4000  mecole-hardman  11.022353    0
4      malcolm-brown   6.507203    1
8       nyheim-hines  18.966331    2
12       james-white  11.511227    3
4016  christian-kirk   9.872176    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 best-ball data (sans flex). Remember, that looks like this:

```
In [36]: qb_wide.head()
Out[36]:
           qb1_name  qb1_points
sim
0        drew-brees   17.000284
1        drew-brees   27.386521
2    baker-mayfield   15.652178
3    baker-mayfield   23.152408
4        drew-brees   16.573342
```

That is, `sim` is the index, and we have name and points columns.

So let's make `flex_wide` that matches that format:

```
In [37]: flex_wide = leftovers.loc[max_points_index].set_index('sim')

In [38]: flex_wide.columns = ['flex_name', 'flex_points']

In [39]: flex_wide.head()
Out[39]:
          flex_name  flex_points
sim
0    mecole-hardman    11.022353
1     malcolm-brown     6.507203
2      nyheim-hines    18.966331
3       james-white    11.511227
4    christian-kirk     9.872176
```

Now finally, we can put everything together.

```
team_wide = pd.concat([qb_wide, rbs_wide, wrs_wide, te_wide, flex_wide,
```

```
In [40]: team_wide.head()
Out[40]:
           qb1_name  qb1_points  ... flex_points  dst1_name  dst1_points
sim                              ...
0        drew-brees   17.000284  ...   11.022353    den-dst     9.854859
1        drew-brees   27.386521  ...    6.507203    den-dst    12.141752
2    baker-mayfield   15.652178  ...   18.966331    den-dst    10.496414
3    baker-mayfield   23.152408  ...   11.511227    den-dst     8.622358
4        drew-brees   16.573342  ...    9.872176    den-dst    14.810502
```

Looking at this, one thing that might be useful is splitting this out into separate player name and point value DataFrames.

```
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 lets us do things like summarize the range of outcomes:

```
In [41]: points.sum(axis=1).describe()
Out[41]:
count    1000.000000
mean      144.617435
std        22.801801
min        81.506150
25%       129.824128
50%       143.934287
75%       159.143670
max       234.898321
dtype: float64
```

Or check how often a player will appear in our lineup.

```
In [42]: names['qb'].value_counts(normalize=True)
Out[42]:
drew-brees        0.547
baker-mayfield    0.453
Name: qb, dtype: float64

In [43]: names['wr2'].value_counts(normalize=True)
Out[43]:
dj-moore            0.217
mike-evans          0.176
curtis-samuel       0.166
anthony-miller      0.138
christian-kirk      0.133
marquise-goodwin    0.099
mecole-hardman      0.071
Name: wr2, dtype: float64

In [44]: names['flex'].value_counts(normalize=True)
Out[44]:
nyheim-hines        0.102
marlon-mack         0.100
noah-fant           0.092
curtis-samuel       0.091
james-white         0.084
devin-singletary    0.069
anthony-miller      0.068
christian-kirk      0.066
marquise-goodwin    0.062
ezekiel-elliott     0.060
malcolm-brown       0.052
mike-evans          0.049
dj-moore            0.041
mecole-hardman      0.038
george-kittle       0.026
Name: flex, dtype: float64
```

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.

```
usage = pd.concat([names[x].value_counts(normalize=True) for x in
                   pos],
                  axis=1, join='outer').fillna(0)
```

```
In [45]: usage
Out[45]:
             qb    rb1    rb2    wr1    wr2    wr3     te   flex   dst
brees     0.547  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.00
mayfield  0.453  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.00
elliott   0.000  0.501  0.196  0.000  0.000  0.000  0.000  0.060  0.00
white     0.000  0.202  0.241  0.000  0.000  0.000  0.000  0.084  0.00
mack      0.000  0.155  0.230  0.000  0.000  0.000  0.000  0.100  0.00
singlet~  0.000  0.100  0.158  0.000  0.000  0.000  0.000  0.069  0.00
brown     0.000  0.036  0.082  0.000  0.000  0.000  0.000  0.052  0.00
hines     0.000  0.006  0.093  0.000  0.000  0.000  0.000  0.102  0.00
evans     0.000  0.000  0.000  0.428  0.176  0.113  0.000  0.049  0.00
moore     0.000  0.000  0.000  0.172  0.217  0.203  0.000  0.041  0.00
kirk      0.000  0.000  0.000  0.137  0.133  0.142  0.000  0.066  0.00
miller    0.000  0.000  0.000  0.101  0.138  0.112  0.000  0.068  0.00
goodwin   0.000  0.000  0.000  0.093  0.099  0.148  0.000  0.062  0.00
samuel    0.000  0.000  0.000  0.043  0.166  0.175  0.000  0.091  0.00
hardman   0.000  0.000  0.000  0.026  0.071  0.107  0.000  0.038  0.00
kittle    0.000  0.000  0.000  0.000  0.000  0.000  0.794  0.026  0.00
fant      0.000  0.000  0.000  0.000  0.000  0.000  0.206  0.092  0.00
den       0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.73
cin       0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.27
```

Each column should sum to 1, and each row sums to how often the player in question was used *anywhere*. So Zeke was used as RB1 50%, RB2 20% and FLEX 6%, which means he shows up in 76% of our lineups.

We can add that.

```
usage.columns = [x.upper() for x in usage.columns]
usage['ALL'] = usage.sum(axis=1)
```

```
In [46]: usage.round(2)
Out[46]:
             QB   RB1   RB2   WR1   WR2   WR3    TE  FLEX   DST   ALL
brees      0.55  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.55
mayfield   0.45  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.45
elliott    0.00  0.50  0.20  0.00  0.00  0.00  0.00  0.06  0.00  0.76
white      0.00  0.20  0.24  0.00  0.00  0.00  0.00  0.08  0.00  0.53
mack       0.00  0.16  0.23  0.00  0.00  0.00  0.00  0.10  0.00  0.48
singlet~   0.00  0.10  0.16  0.00  0.00  0.00  0.00  0.07  0.00  0.33
brown      0.00  0.04  0.08  0.00  0.00  0.00  0.00  0.05  0.00  0.17
hines      0.00  0.01  0.09  0.00  0.00  0.00  0.00  0.10  0.00  0.20
evans      0.00  0.00  0.00  0.43  0.18  0.11  0.00  0.05  0.00  0.77
moore      0.00  0.00  0.00  0.17  0.22  0.20  0.00  0.04  0.00  0.63
kirk       0.00  0.00  0.00  0.14  0.13  0.14  0.00  0.07  0.00  0.48
miller     0.00  0.00  0.00  0.10  0.14  0.11  0.00  0.07  0.00  0.42
goodwin    0.00  0.00  0.00  0.09  0.10  0.15  0.00  0.06  0.00  0.40
samuel     0.00  0.00  0.00  0.04  0.17  0.18  0.00  0.09  0.00  0.47
hardman    0.00  0.00  0.00  0.03  0.07  0.11  0.00  0.04  0.00  0.24
kittle     0.00  0.00  0.00  0.00  0.00  0.00  0.79  0.03  0.00  0.82
fant       0.00  0.00  0.00  0.00  0.00  0.00  0.21  0.09  0.00  0.30
den        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.73  0.73
cin        0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.27  0.27
```

Really, we don't need all the zeros (the only people eligible to be used at QB are Brees and Baker, that's why everyone else has a 0.00) so let's get rid of them so this looks a little better

```
In [47]: usage_clean = usage.round(2).astype(str)

In [47]:
for x in usage_clean.columns:
    usage_clean[x] = (usage_clean[x]
                      .str.pad(4, fillchar='0', side='right')
                      .str.replace('^0.00$',''))
--

In [48]: usage_clean
Out[48]:
               QB   RB1   RB2   WR1   WR2   WR3    TE  FLEX   DST   ALL
brees        0.55                                                    0.55
mayfield     0.45                                                    0.45
elliott            0.50  0.20                        0.06          0.76
white              0.20  0.24                        0.08          0.53
mack               0.16  0.23                        0.10          0.48
singletary         0.10  0.16                        0.07          0.33
brown              0.04  0.08                        0.05          0.17
hines              0.01  0.09                        0.10          0.20
evans                          0.43  0.18  0.11        0.05          0.77
moore                          0.17  0.22  0.20        0.04          0.63
kirk                           0.14  0.13  0.14        0.07          0.48
miller                         0.10  0.14  0.11        0.07          0.42
goodwin                        0.09  0.10  0.15        0.06          0.40
samuel                         0.04  0.17  0.18        0.09          0.47
hardman                        0.03  0.07  0.11        0.04          0.24
kittle                                           0.79  0.03          0.82
fant                                             0.21  0.09          0.30
st                                                           0.73  0.73
st                                                           0.27  0.27
```

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 = 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()
```

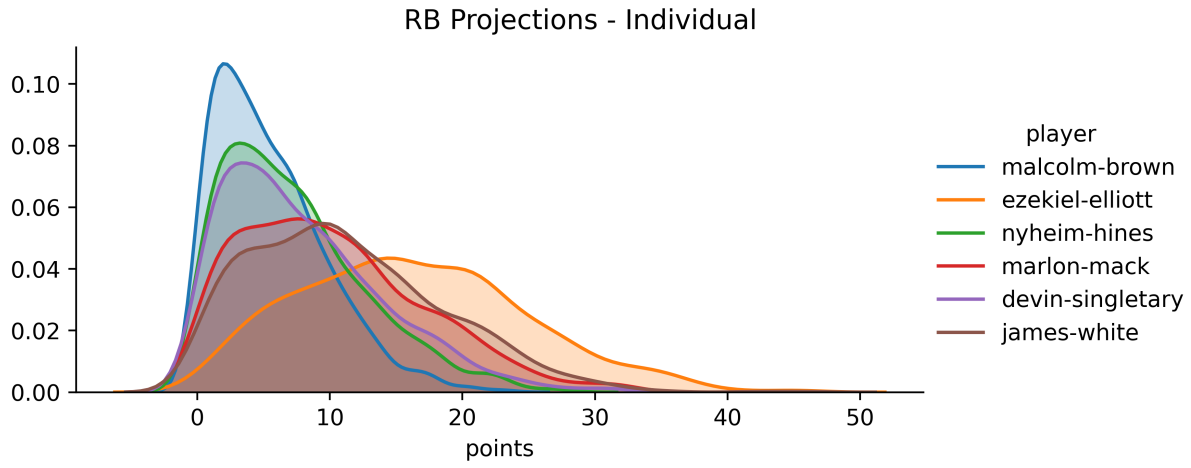

Figure 0.1: RB Individual Player Projections

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()
```

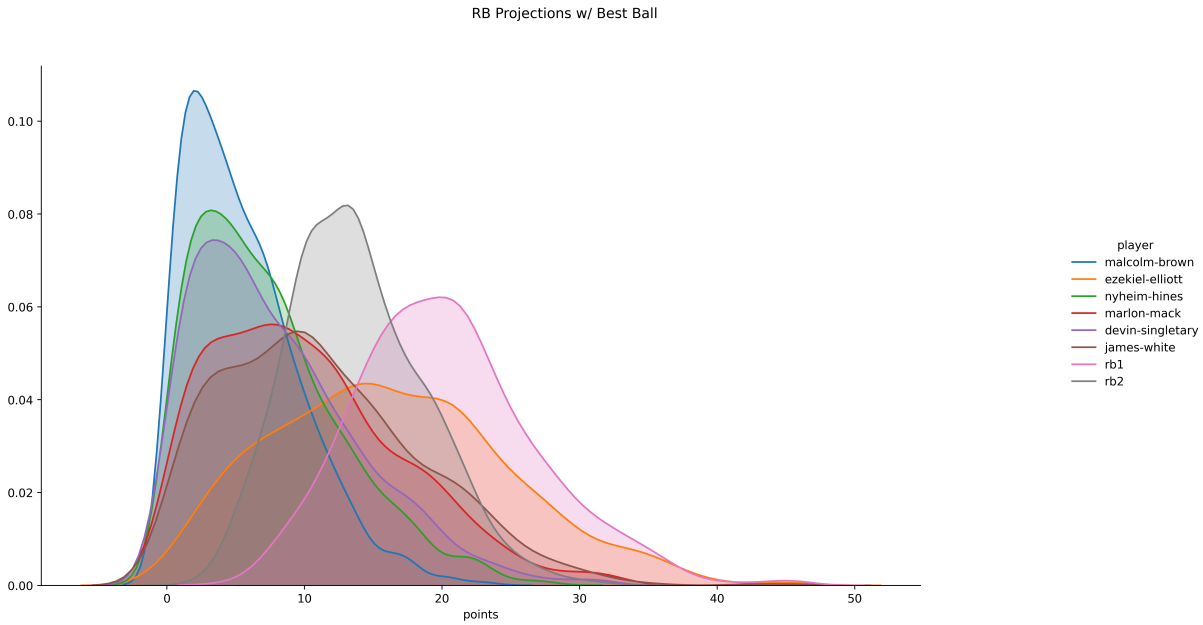

Figure 0.2: RB w/ Best Ball Starters

And there you go! Your own accurate best ball projector with plots.

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/distribution/

And click on the green button to download the 3.x version (3.8 at time of this writing) for your operating system.

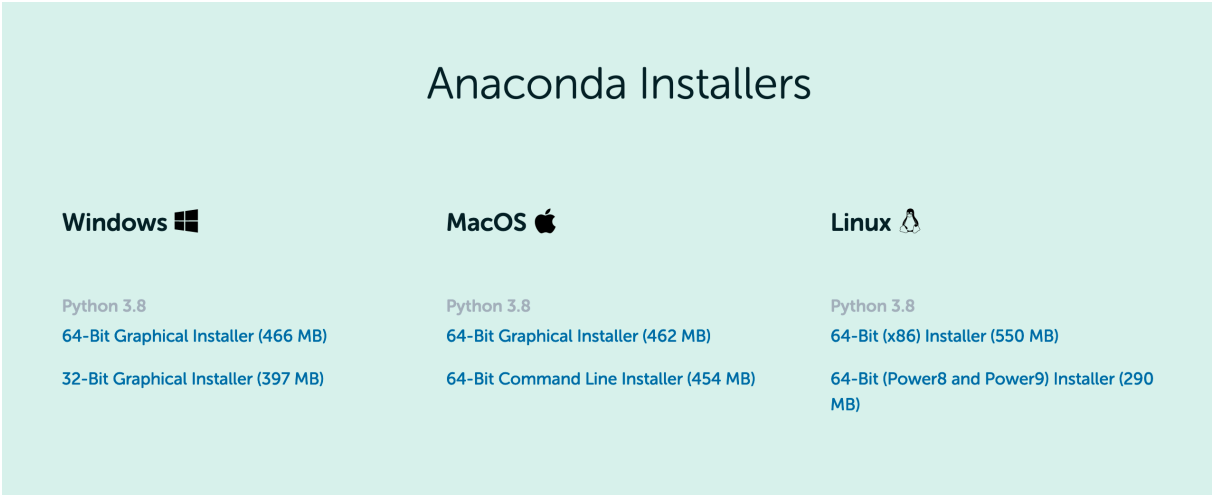

Figure 0.1: Python 3.x on the Anaconda site

When you Download it, Anaconda may try to get you to sign up for "Anaconda Nucleus". I'm not sure what this is. It's definitely NOT required to use Anaconda or Python, and — despite all the good work Anaconda is doing making it easier for people to get started with Python — it's annoying that they try to make you sign up for it. Just close it.

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.

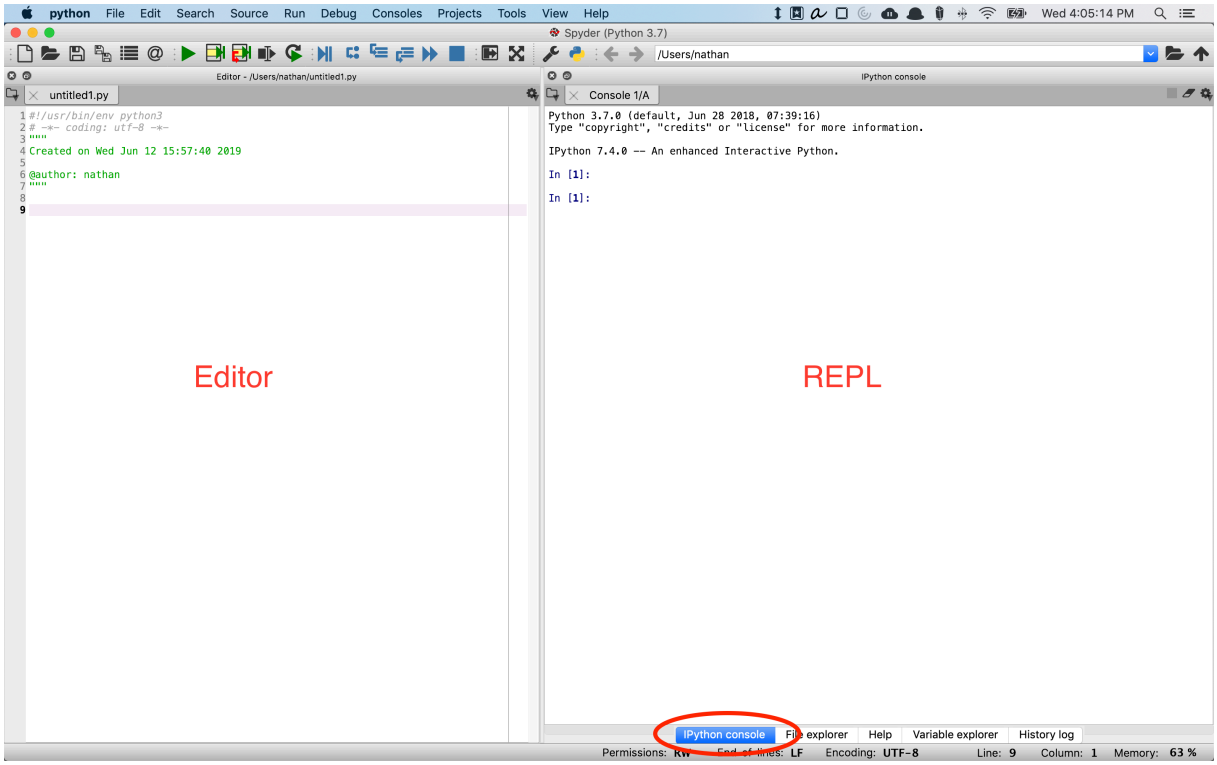

Figure 0.2: Editor and REPL in Spyder

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 math[1], 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.

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.

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

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 last year. 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:

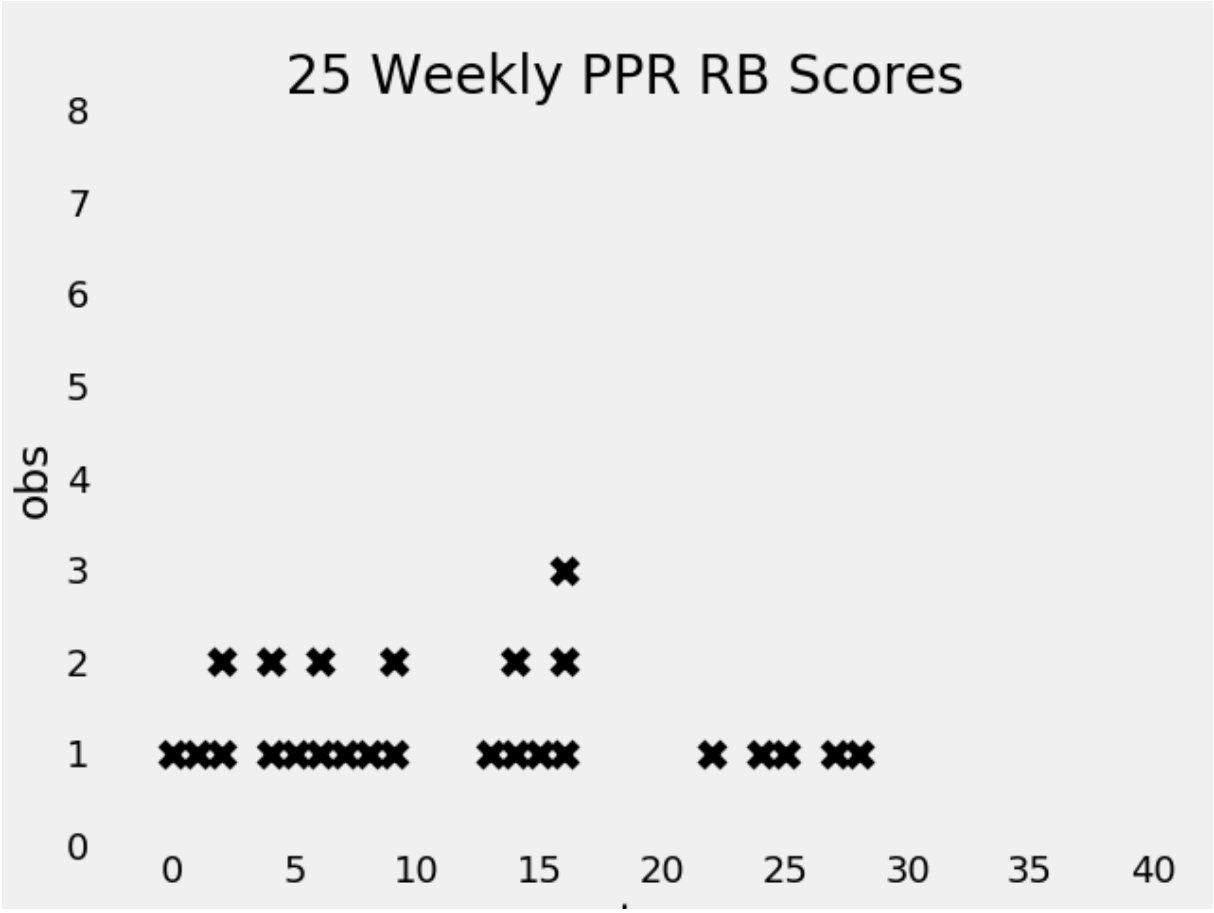

Figure 0.1: 25 random RB performances in 2019

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:

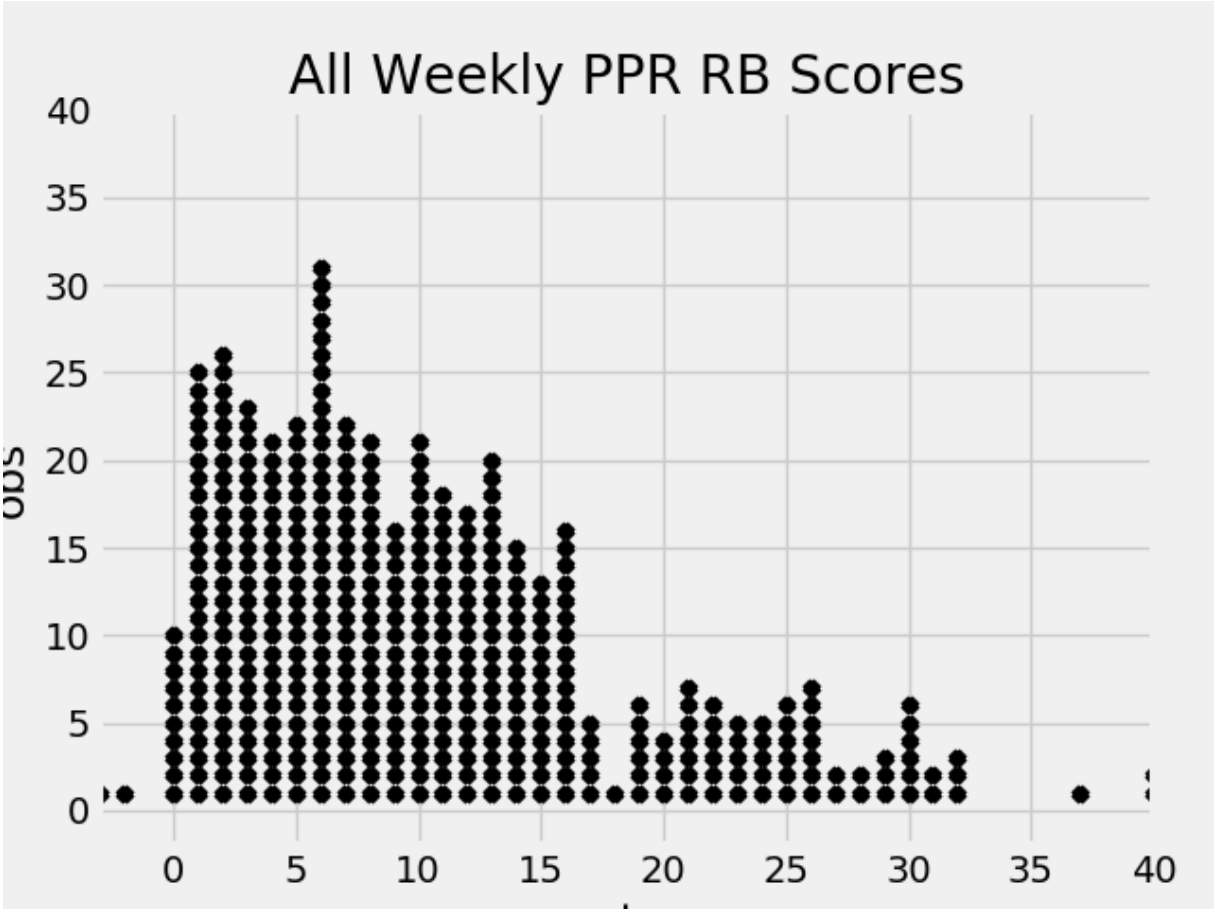

Figure 0.2: Stacked X's for all RB performances in 2019

This is the first key to thinking probabalisticly 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.

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.

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 to 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]:
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:

[1] This 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

License Key and Email

To get started you'll need your Fantasy Math license key. You can find link you recieved by email to download this guide:

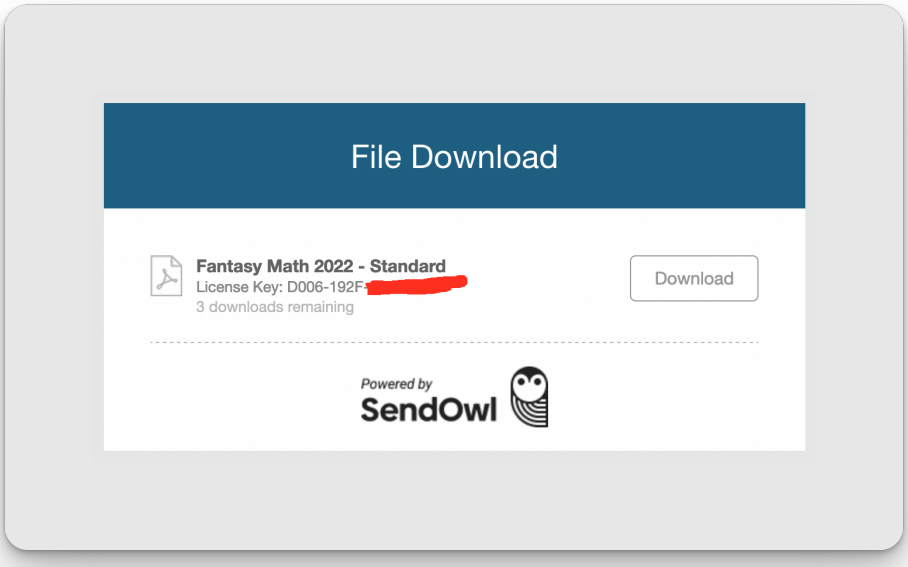

Figure 0.1: Sendowl License Key Screenshot

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:

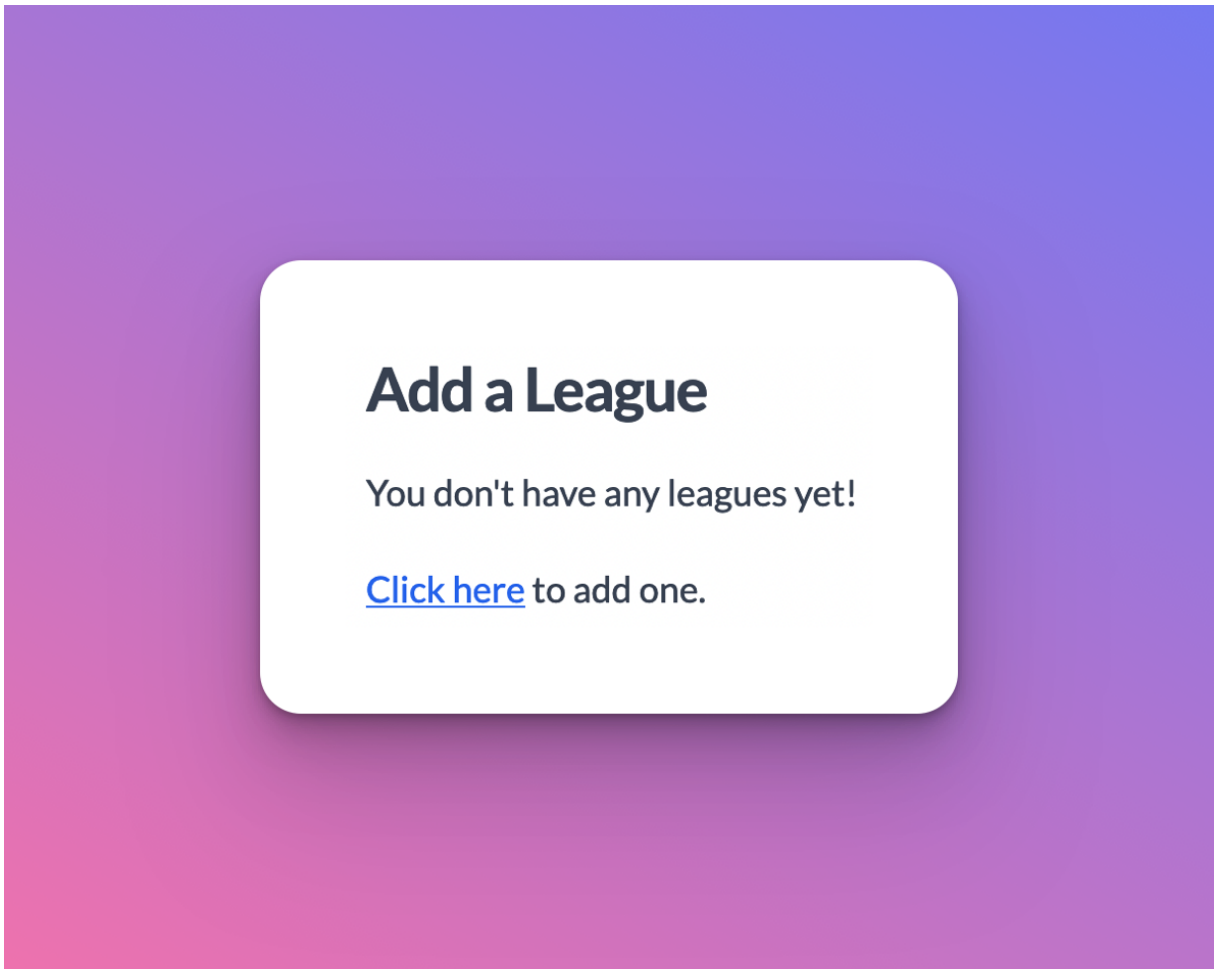

Figure 0.2: Add a League

Fantasy Math sorts your results and scoring settings by league. So let's add one, say "Family League".

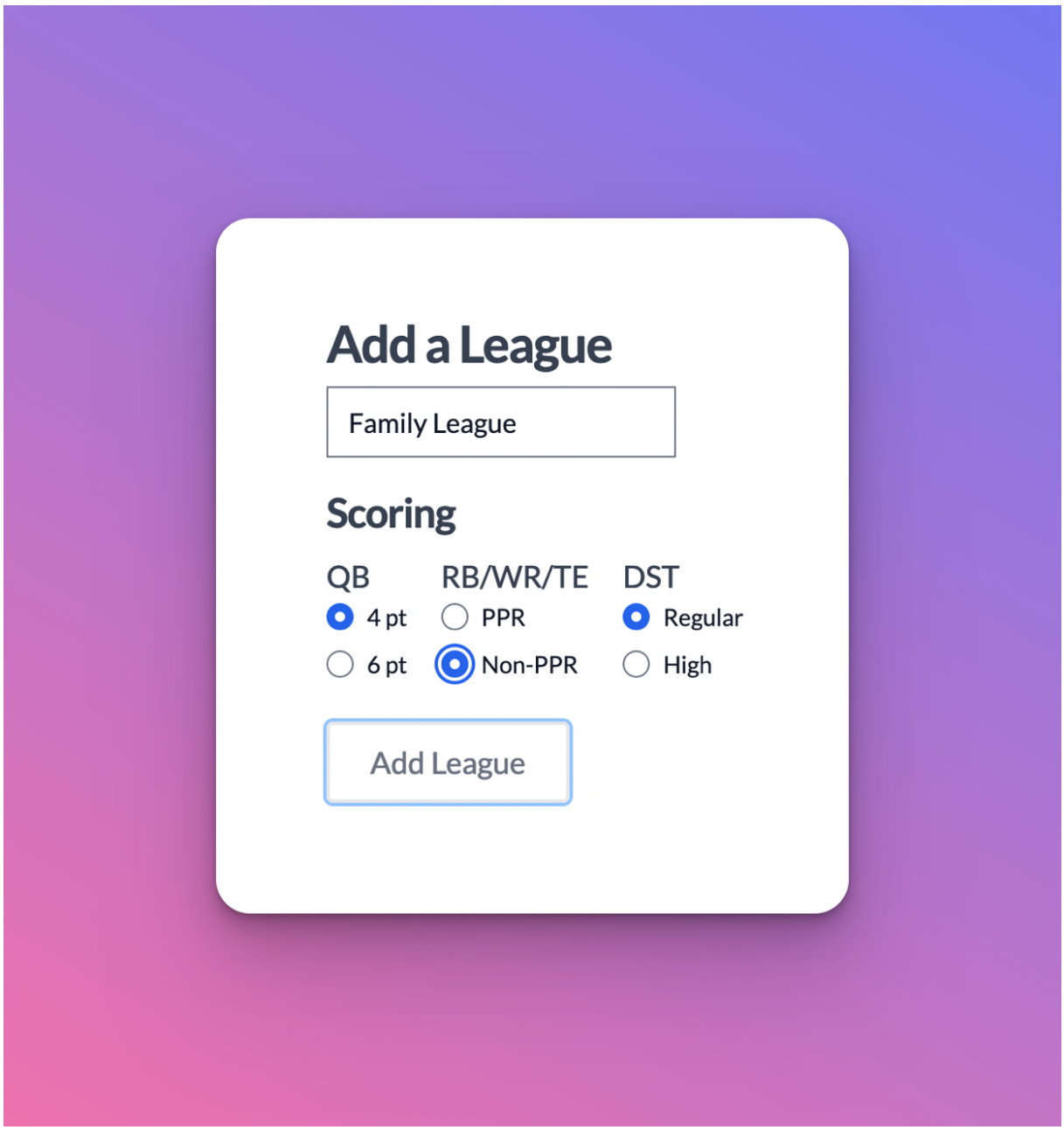

Figure 0.3: Family League

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:

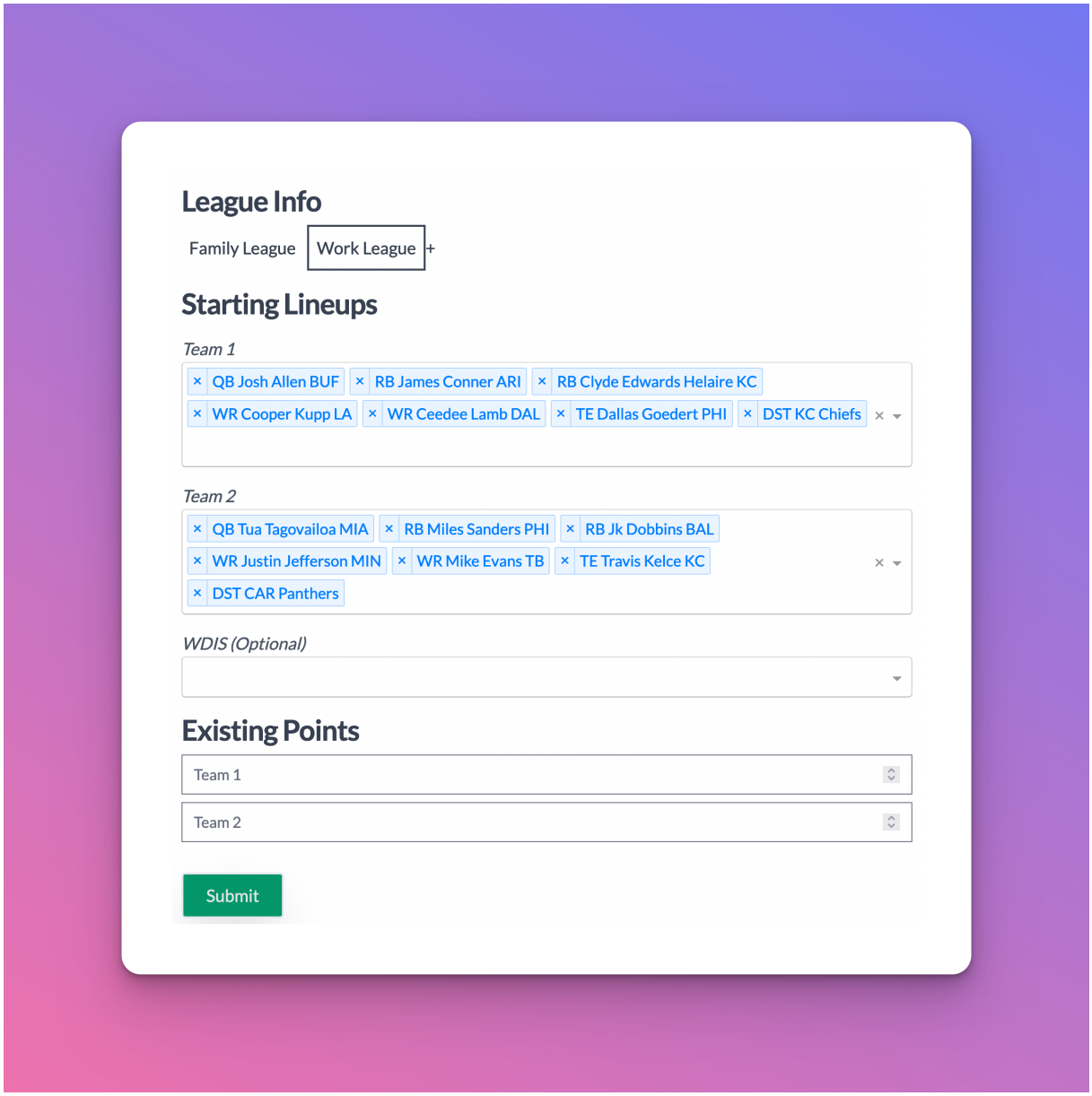

Figure 0.4: Week 1

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.

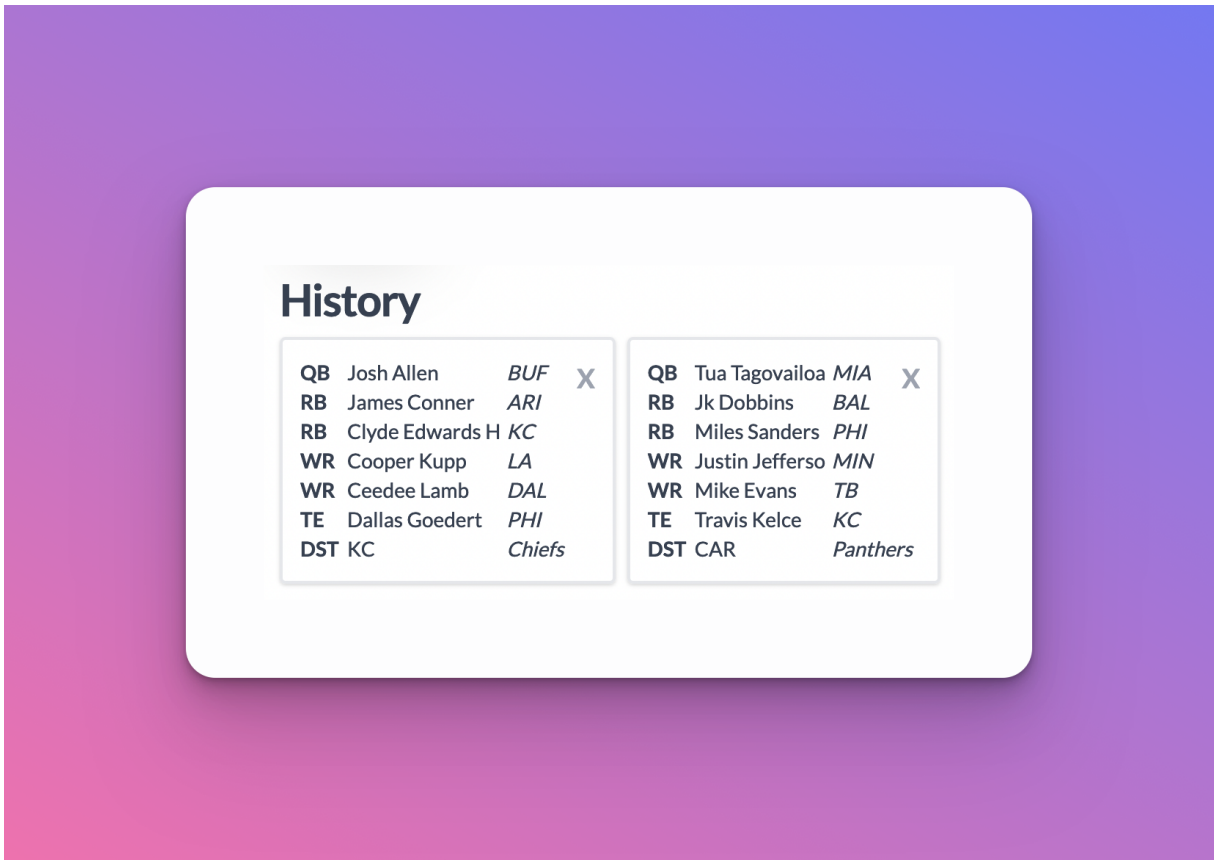

Figure 0.5: History

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 distrribution. I find this helpful to see the strengths/weaknesses of my matchup.

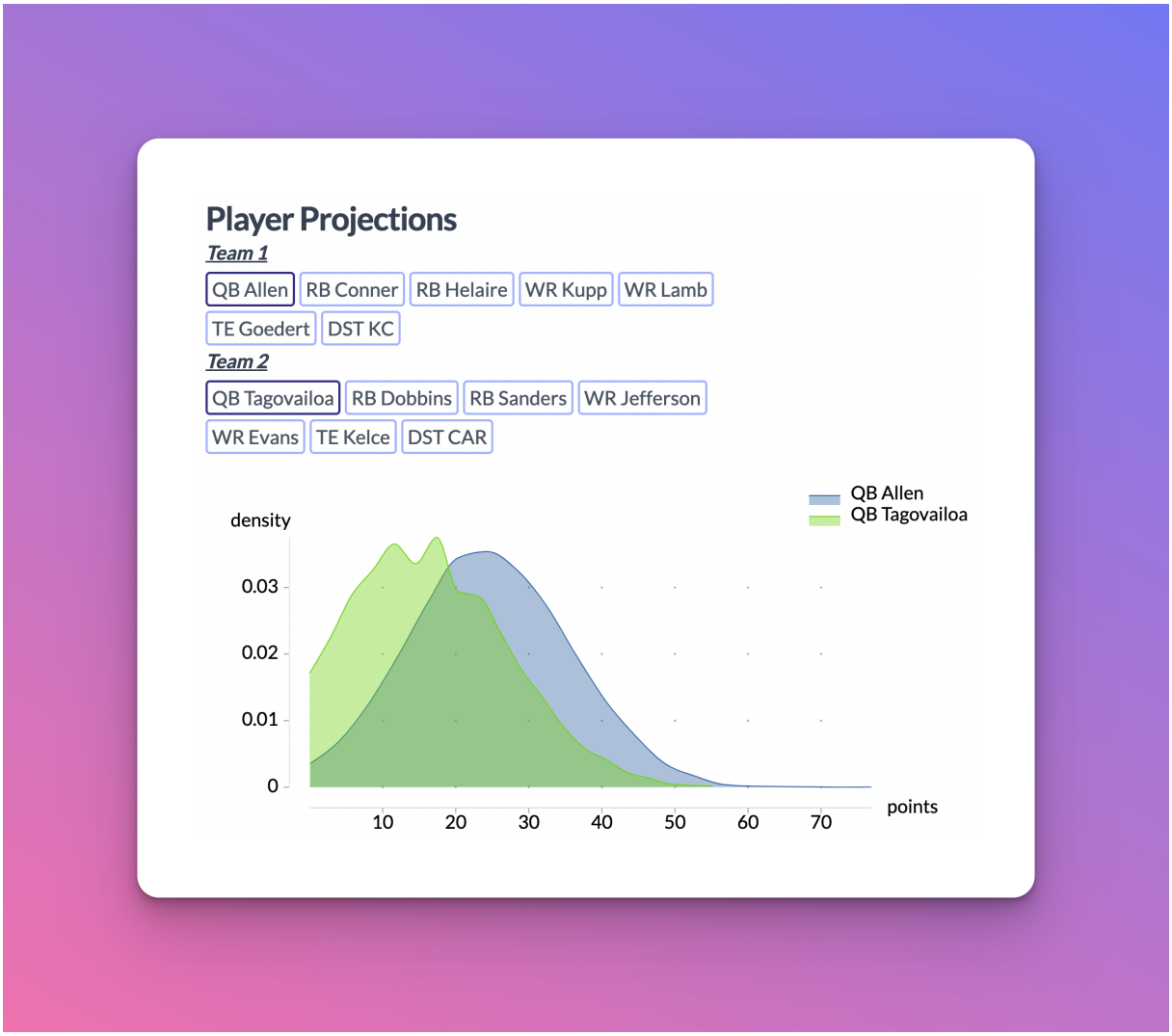

Figure 0.7: Player Projections

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.

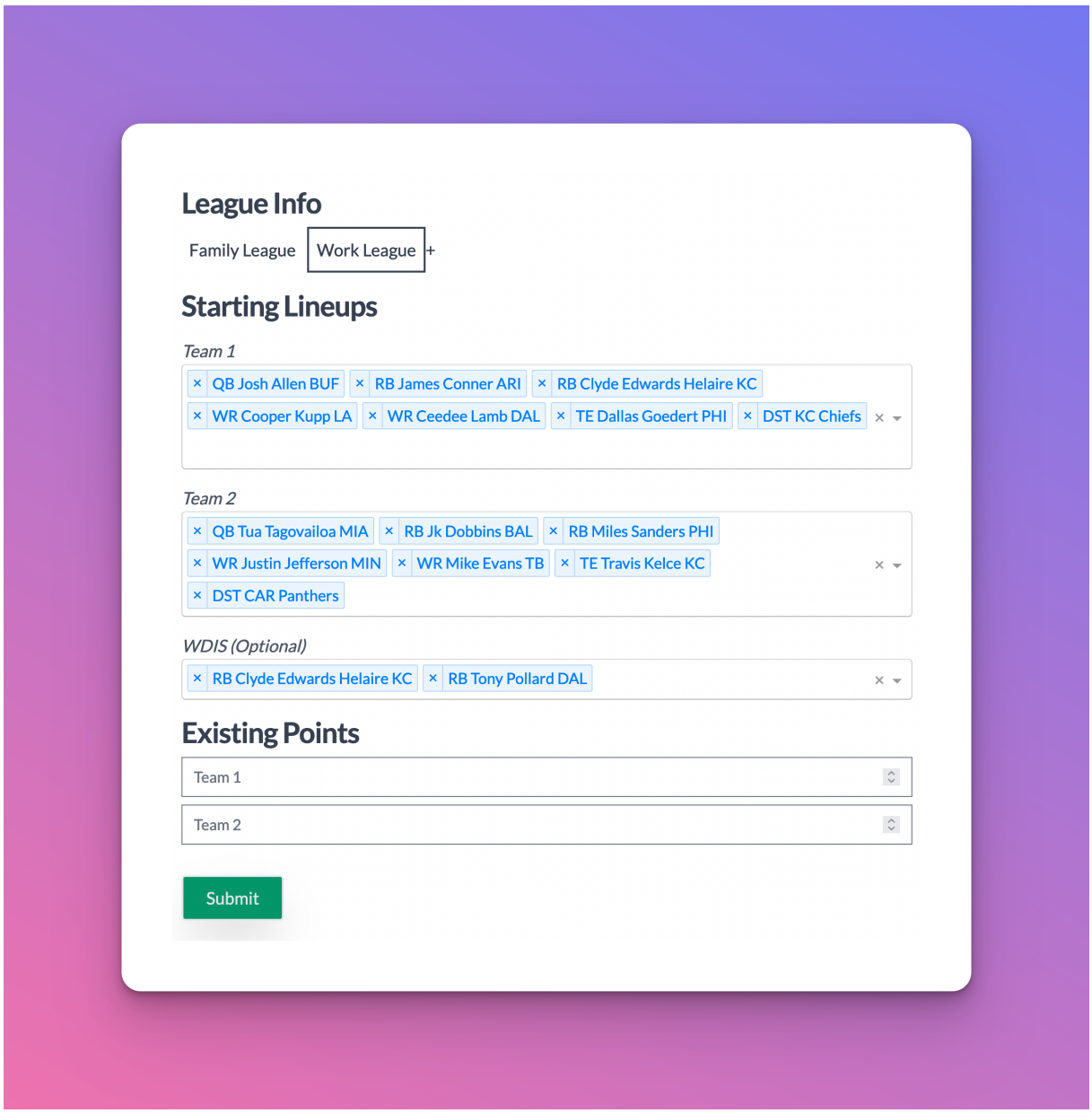

Figure 0.8: Who Do I Start

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