NATHAN BRAUN

LEARN TO CODE with SOCCER

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

Preview of Learn to Code with Soccer

v0.0.3

Contents

Note About This Preview

The purpose of this preview is to help you decide if Learn to Code with Soccer is right for you. There are a *lot* of options for learning data science and how to code, many of them free.

So, the decision about whether or not to learn from *this* book will come down to:

1. How interesting and motivating soccer is to you as a topic.
2. How the book compares to alternatives, specifically whether it can save you enough time (e.g. by explaining things clearly, putting everything in one place, focusing on the right stuff) to justify paying for it vs working through some free tutorial or other book.

Don't underestimate 1. Sports are a major gateway to coding, especially for the self taught crowd (like me). There's a reason Nate Silver started out by programming baseball models on the side at his day job. The concepts you'll learn here are applicable to *any* analysis problem, not just soccer.

But this preview is to help you get a sense of 2. Unless you value it very cheaply, your time is by far the biggest cost when learning anything. It's important to make sure this book is a fit. The preview includes selections from the chapters 2 (basic Python) and 7 (modeling) to help you do that.

Though I'd recommend trying out Python, it does require some setup (see the appendix for more) and I understand people may want to get a sense of the book's teaching style at a glance. If that's you, check out modeling chapter (no code required) and learn linear regression while you're at it!

Other highlights from the preview:

- The High Level Data Analysis Process gives an overview of data science generally.
- More String Methods and How to Figure Things Out in Python explains why I'm not teaching you all 44 of Python's options for working with text (spoiler: to not bog you down) and what you should do instead.

I sincerely hope it's useful. Happy coding!

Nate

What People Are Saying

> "The book here was **really, really well done**…" - Bill Connelly, ESPN

> "This is amazingly awesome. I've recently slowly crept into data science driven by a pet passion for fantasy sport analytics. …I'm roughly 40 pages into this book and **the way the learning is framed here is 10x what you'll get someplace else.**" - u/Nick58

> "I'm like ~40 pages in and the simple intro to python chapter is much more engaging for me personally because it's info I'm interested in. I've taken automate the boring stuff, python for finance, etc and while those courses are great.. **I seem to be understanding it better because its about a subject I like**." - u/financenstuff

> "Incredible work! Bought it right away. Only 3 chapters in and **this book is already better than expected**. Worth every penny. Thank you!" - u/TheMotizzle

> "…probably the **best / most complete Pandas walk through I've seen**." - Bill S

> "I've probably **picked up more, and at a better pace**, using this than a lot of the free online tools I'd been trying the past few months. - Ryan P

> "Love the book, bought it a before last season of the nfl. **My python has come a very long way thanks to you.**" - u/TomatoHead7

> "…it helped me tremendously … **I wouldn't be where I'm at with the Python language today without this book to kick start things.**" - u/F1rstxLas7

> "I can't tell you how many times I've tried to get into programming and gave up because it was so dry. **This has been such a nice change of pace and I'm loving it**." - Paval M

> "Just picked up LTCWFF last week and I am really enjoying it so far. This is **exactly what I needed to finally get past tutorial hell and apply Python to something I love**." - Philip D

> "I have always wanted to learn a language but always seemed to get discouraged by the 'Hello World' chapters that were never ending. I like that your book cuts out the riff raff and teaches the important things! **I'm flying through the book and feel like I'm learning a ton!** Best wishes from a satisfied customer" - Jason K

> "I recently purchased LTCWFF and **could not be more satisfied with the content**. …it has been great to work through your in-depth examples learning new skills. I had a previous interest in this sort of analysis and have had intermediate programming experience, but never could tie the two together." - Owen B

> "I purchased this the other day and thus far it's been great refreshers for basic Python… **I appreciate the Anki cards** … they're helping cement the terminology and such." - u/michaelmanieri

> "…**very informative and good intro to coding**. Additionally, [Nate] would answer any questions I emailed him within 24 hours. Excellent customer service and pushed new editions to everyone who had already paid. I really appreciate [Nate]'s commitment to his product." - u/ledsdeadbaby

> "I was amazed by how you **broke down complicated concepts and made them easier to understand**." - Ryan C

1. Introduction

The Purpose of Data Analysis

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

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

Data analysis is one (hopefully) accurate and consistent way to get these insights.

Of course, that requires *data*.

What is Data?

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

You might have data about anything, but let's take a soccer game, say the 2018 World Cup Final - France vs Croatia. What would a collection of structured information about it look like?

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

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

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

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

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

It's common to refer to rows as **observations** and columns as **variables**, particularly when using the data for more advanced forms of analysis, like modeling. Other names for this rectangle-like format include tabular data or flat file (because all this info about France-Croatia is *flattened* out into one big table).

More in the full version —

Excel. CSVs. Delimited data. More example datasets. Granularity.

What is Analysis?

How many soccer balls are in the following picture?

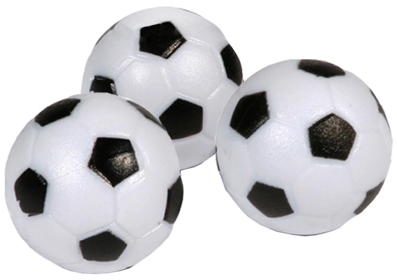


Figure 0.1: Few soccerballs

Pretty easy question right? What about this one?

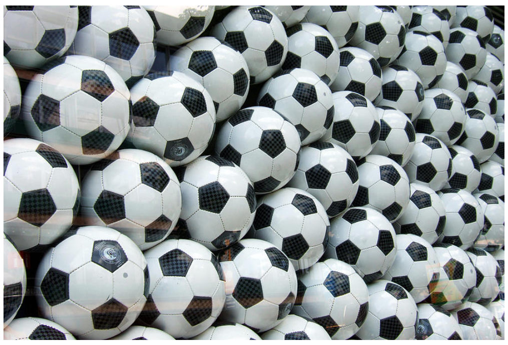


Figure 0.2: Many soccerballs

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

If you open up the shot data this book comes with, you'll notice it's 1366 rows and 24 columns.

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

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

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

Types of Data Analysis

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

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

More in the full version —

Key to using summary statitics. Moneyball. Relationship between data and models. Modeling in practice. Modeling the future.

High Level Data Analysis Process

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

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

1. Collecting Data

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

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

2. Storing Data

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

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

3. Loading Data

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

This book covers basic SQL and loading data with Python.

4. Manipulating Data

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

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

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

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

5. Analyzing Data for Insights

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

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

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

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

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

This is the forest. If you ever find yourself banging your head against a tree — either confused or wondering *why* we're talking about something — refer back here and think about where it fits in.

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

2. Python

Introduction to Python Programming

This section is an introduction to basic Python programming.

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

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

We'll write code that makes heavy use of both later in the book. But, even when using third party packages, you will also be using a core set of Python features and functionality. These features — called the **standard library** — are built-in to Python.

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

How to Read This Chapter

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

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

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

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

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

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

Let's get started.

Important Parts of the Python Standard Library

Comments

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

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

The problem with comments is it's easy for them to become out of date. This often happens when you change your code and forget to update the comment.

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

For example, this would be an unnecessary comment:

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

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

Variables

Variables are a fundamental concept in any programming language.

At their core, variables[1] are just named pieces of information. This information can be anything from a single number to an entire dataset — the point is that they let you store and recall things easily.

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

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

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

Assigning data to variables

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

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

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

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

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

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

One of the benefits of developing with a REPL is that you can type in a variable, and the REPL will evaluate (i.e. determine what it is) and print it. That's what the code above is doing. But note while `goals_scored` is 2, the assignment statement itself, `goals_scored = 2`, doesn't evaluate to anything, so the REPL doesn't print anything out.

You can update and override variables too. Going into the code below, `goals_scored` has a value of 2 (from the code we just ran above). So the right hand side, `goals_scored + 1` is evaluated first (2 + 1 = 3), and *then* the result gets (re)assigned to `goals_scored`, overwriting the 2 it held previously.

[1] Note: previously we talked about how, in the language of modeling and tabular data, *variable* is another word for *column*. That's different than what we're talking about here. A variable in a dataset or model is a column; a variable in your code is named piece of information. You should usually be able to tell by the context which one you're dealing with. Unfortunately, imprecise language comes with the territory when learning new subjects, but I'll do my best to warn you about any similar pitfalls.

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

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

Types

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

```
In [6]: keeper_saves = 12  # int
In [7]: ball_speed_kmh = 96.5  # float
```

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

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

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

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

In [11]: type(keeper_saves)
Out[11]: int
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Interlude: How to Figure Things Out in Python

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

The first tool you can use to figure out your options is the REPL. In particular, the REPL's **tab completion** functionality. Type in a string like `'lionel messi'` then `.` and hit tab. You'll see all the options available to you (this is only the first page, you'll see more if you keep pressing tab).

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

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

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

```
In [16]: foo = 'lionel messi'
Out[16]: foo.
            capitalize()   encode()       format()
            isalpha()      isidentifier() isspace()
            ljust()        casefold()     endswith()
            format_map()   isascii()      islower()
```

Then, when you find something you're interested in, enter it in the REPL with a question mark after it, like `'lionel messi'.capitalize?` (or `foo.capitalize?` if you're doing it that way).

You'll see:

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

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

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

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

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

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

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

```
In [18]: '  lionel messi'
```

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

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

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

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

with a throwaway variable that doesn't matter, many programmers will name it `foo`. Second and third variables that don't matter are `bar` and `baz`. Apparently this dates back to the 1950's.

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

A quick test confirms that's what we want.

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

Stackoverflow

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

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

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

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

As programmer-blogger Umer Mansoor writes,

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

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

> A big reason to use Google is that it is hard to remember all those minor details and nuances es-

> pecially when you are programming in multiple languages… As Einstein said: 'Never memorize something that you can look up.'

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

Bools

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

Similar to variable names, bool values lack quotes. So `"True"` is a string, not a bool.

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

```
# some numbers to use in our examples
In [21]: team1_goals = 2
In [22]: team2_goals = 1

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

In [24]: team2_won = team1_goals < team2_goals

In [25]: teams_tied = team1_goals == team2_goals

In [26]: teams_did_not_tie = team1_goals != team2_goals

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

In [28]: teams_did_not_tie
Out[28]: True
```

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

```
In [29]: teams_tied = (team1_goals = team2_goals)
...
SyntaxError: invalid syntax
```

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

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

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

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

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

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

In [33]: meh = not (shootout or
                    at_least_one_good_team or
                    you_guys_are_bad)
```

if statements

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

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

In [35]: message
Out[35]: 'Nice job team 1!'
```

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

Container Types

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

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

More basic Python in the full version —

Lists. Dicts. Loops. Comprehensions. Functions. Defining your own functions. Side effects and why they're bad. Libraries. The os library.

7. Modeling

The Simplest Model

Let's say we want a model that takes in distance to the net and predicts whether a shot from there will be a goal or not. So we might have something like:

```
goal or not = model(meters to the net)
```

Terminology

First some terminology: the variable "goal or not" is our **output variable**[1]. There's always exactly one output variable.

The variable "meters to net" is our **input variable** [2]. In this case we just have one, but we could have as many as we want. For example:

```
goal or not = model(meters to the net, time left in game)
```

OK. Back to:

```
goal or not = model(meters to the net)
```

Here's a question: what is the simplest implementation for `model(...)` we might come up with?

How about:

```
model(...)= No
```

So give it any yards to go, and our model spits out: "no, it will not be a goal". Since the vast majority of shots don't actually score goals, this model will be very accurate! But since it never says anything besides no, it's not that interesting or useful.

What about:

```
prob goal = 1 + -0.01*distance in m + -0.0000001*distance in m ^ 2
```

[1] Other terms for this variable include: *left hand side variable* (it's to the left of the equals sign); *dependent* variable (its value *depends* on the value of distance to the goal), or *y* variable (traditionally output variables are denoted with y, inputs with x's).

[2] Other words for input variables include: *right hand side*, *independent*, *explanatory*, or *x* variables.

So from 1 meter out we'd get a probability of 0.99, 3 meters 0.97, 10 meters 0.90 and 99 meters 0.000002. This is more interesting. I made the numbers up, so it isn't a *good* model (for 50 meters it gives about a 0.50 probability of a goal, which is way too high). But it shows how a model transforms inputs to an output using some mathematical function.

Linear regression

This type of model format:

```
output variable =
    some number + another number*data + yet another number*other data
```

is called **linear regression**. It's *linear* because when you have one piece of data (input variable), the equation is a line on a set of x, y coordinates, like this:

`y = m*x + b`

If you recall math class, `m` is the slope, `b` the intercept, and `x` and `y` the horizontal and vertical axes.

Notice instead of saying *some number*, *another number* and *input* and *output data* we use `b`, `m`, `x` and `y`. This shortens things and gives you an easier way to refer back to parts of the equation. The particular letters don't matter (though people have settled on conventions). The point is to provide an abstract way of thinking about and referring to parts of our model.

A linear equation can have more than one data term in it, which is why statisticians use `b0` and `b1` instead of `b` and `m`. So we can have:

`y = b0 + b1*x1 + b2*x2 + ... + ... bn*xn`

Up to any number n you can think of. As long as it's a bunch of `x*b` terms added together it's a linear equation. Don't get tripped up by the notation: `b0`, `b1`, and `b2` are different numbers, and `x1` and `x2` are different columns of data. The notation just ensures you can include as many variables as you need to (just add another number).

In our probability-of-a-goal model that I made up, `x1` was meters from the net, and `x2` was meters from the net squared. We had:

`prob goal = b0 + b1*(distance in meters)+ b2*(distance in meters ^ 2)`

Let's try running this model in Python and see if we can get better values for `b0`, `b1`, and `b2`.

Remember: the first step in modeling is making a dataset where the columns are your input variables and one output variable. So we need a three column DataFrame with distance, distance squared, and goal scored or not. We need it at the shot level. Let's do it.

Note: the code for this section is in the file `07_01_ols.py`. We'll pick up from the top of the file.

```
import pandas as pd
import statsmodels.formula.api as smf
from os import path

DATA_DIR = './data'

df = pd.read_csv(path.join(DATA_DIR, 'shots.csv'))

df['dist_m_sq'] = df['dist_m']**2
df['goal'] = df['goal'].astype(int)
```

Besides loading our libraries and the data, this first section of the code also does some minor processing:

First, we had to make `dist_m_sq` by squaring our `dist` variable. Note exponents in Pandas use the `**` operator. I don't necessarily expect you to have known that off the top of your head. However, I *do* expect you to figure it out via Google after trying `^` and getting an error message.

This initial `goal` variable is a column of booleans (`True` if the shot went in, `False` otherwise). That's fine except our model can only operate on actual numbers. We need to convert this boolean column into its numeric equivalent.

The way to do this while making everything easy to interpret is by transforming `goal` into a **dummy variable**. Like a column of booleans, a dummy variable only has two values. Instead of `True` and `False` it's just `1` and `0`. Calling `astype(int)` on a boolean column will automatically do that conversion (line 13) [3].

Now we have our data. In many ways, getting everything to this point is the whole reason we've learned Pandas, SQL, scraping data and everything else. All for this:

```
In [1]: df[['goal', 'dist', 'dist_m_sq']].head()
Out[1]:
   goal     dist_m   dist_m_sq
0     0  12.987566  168.676860
1     0  16.559476  274.216235
2     0  17.013624  289.463402
3     1   8.506812   72.365850
4     0  15.975528  255.217498
```

Once we have our table, we just need to pass it to our modeling function, which we get from the third

[3] Notice even though we have two outcomes — goal or not — we just have the one column. There'd be no benefit to including an extra column `missed` because it'd be the complete opposite of `goal`; it doesn't add any information. In fact, including two perfectly correlated variables like this in your input variables breaks the model, and most statistical programs will drop the unnecessary variable automatically.

party library statsmodels. There are different ways to use it, but I've found the easiest is via the formula API.

We imported this at the top:

```
import statsmodels.formula.api as smf
```

We'll be using the ols function. OLS stands for Ordinary Least Squares, and is another term for basic, standard linear regression.

Compared to getting the data in the fight format, actually running the model in Python is trivial. We just have to tell smf.ols which output variable and which are the inputs, then run it.

We do that in two steps like this:

```
In [3]: model = smf.ols(formula='goal ~ dist_m + dist_m_sq', data=df)

In [4]: results = model.fit()
```

Once we've done that, we can look at the results:

```
In [5]: results.summary2()
Out[5]:
"""
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.068
Dependent Variable: goal             AIC:                366.9415
Date:               2022-07-08 13:04 BIC:                382.6004
No. Observations:   1366             Log-Likelihood:     -180.47
Df Model:           2                F-statistic:        50.75
Df Residuals:       1363             Prob (F-statistic): 5.51e-22
R-squared:          0.069            Scale:              0.076425
------------------------------------------------------------------
               Coef.   Std.Err.     t     P>|t|    [0.025   0.975]
------------------------------------------------------------------
Intercept      0.2985    0.0229  13.0560  0.0000   0.2537   0.3434
dist_m        -0.0148    0.0017  -8.5768  0.0000  -0.0182  -0.0114
dist_m_sq      0.0001    0.0000   5.0956  0.0000   0.0001   0.0002
-----------------------------------------------------------------
Omnibus:              699.924      Durbin-Watson:         1.997
Prob(Omnibus):        0.000        Jarque-Bera (JB):      3093.164
Skew:                 2.559        Prob(JB):              0.000
Kurtosis:             8.305        Condition No.:         2122
=================================================================
* The condition number is large (2e+03). This might indicate
strong multicollinearity or other numerical problems.
"""
```

We get back a lot of information from this regression. The part we're interested in — the values for b0, b1, b2 — are under Coef (for coefficients). They're also available in results.params.

Remember the intercept is another word for b0. It's the value of y when all the data is 0. In this case, we can interpret as the probability of making a goasl when you're right next to — 0 feet away — the net. The other coefficients are next to dist and dist_m_sq.

So instead of my made up formula from earlier, the formula that best fits this data is:

`0.2985 + -0.0148*meters + 0.0001*(meters ^ 2).`

Let's test it out with some values:

```
In [6]:
def prob_of_goal(meters):
    b0, b1, b2 = results.params
    return (b0 + b1*meters + b2*(meters**2))

In [7]: prob_of_goal(1)
Out[7]: 0.2838389299586142

In [8]: prob_of_goal(15)
Out[8]: 0.1088113102073702

In [9]: prob_of_goal(25)
Out[9]: 0.018905161012311905
```

Seems reasonable. Let's use the results.predict method to predict it for every value of our data.

```
In [10]: df['goal_hat'] = results.predict(df)

In [11]: df[['goal', 'goal_hat']].head(5)
Out[11]:
   goal  goal_hat
0     0  0.068017
1     0  0.023154
2     0  0.040137
3     1  0.178944
4     0  0.100453
```

We can see, for the first five observations, our goal_hat was the highest on the shot that was actually a goal. Not bad.

It's common in linear regression to predict a newly trained model on your input data. The convention is to write this variable with a ^ over it, which is why it's often suffixed with "hat".

The difference between the predicted values and what actually happened is called the **residual**. The math of linear regression is beyond the scope of this book, but basically the computer is picking out `b0`, `b1`, `b2` to make the residuals as small as possible[4].

The proportion of variation in the output variable that your model "explains" (the rest of variation is in the residuals) is called R^2 ("R squared", often written R2). It's always between 0-1. An R2 of 0 means your model explains nothing. An R2 of 1 means your model is perfect: your `yhat` always equals `y`; every residual is 0.

[4]Technically, OLS regression finds the coefficients that make the total sum of each *squared* residual as small as possible. Squaring a residual makes sure it's positive. Otherwise the model could just "minimize" the sum of residuals by underpredicting everything to get a bunch of negative numbers. Note "squares as small as possible" is partly the name, ordinary "least squares" regression.

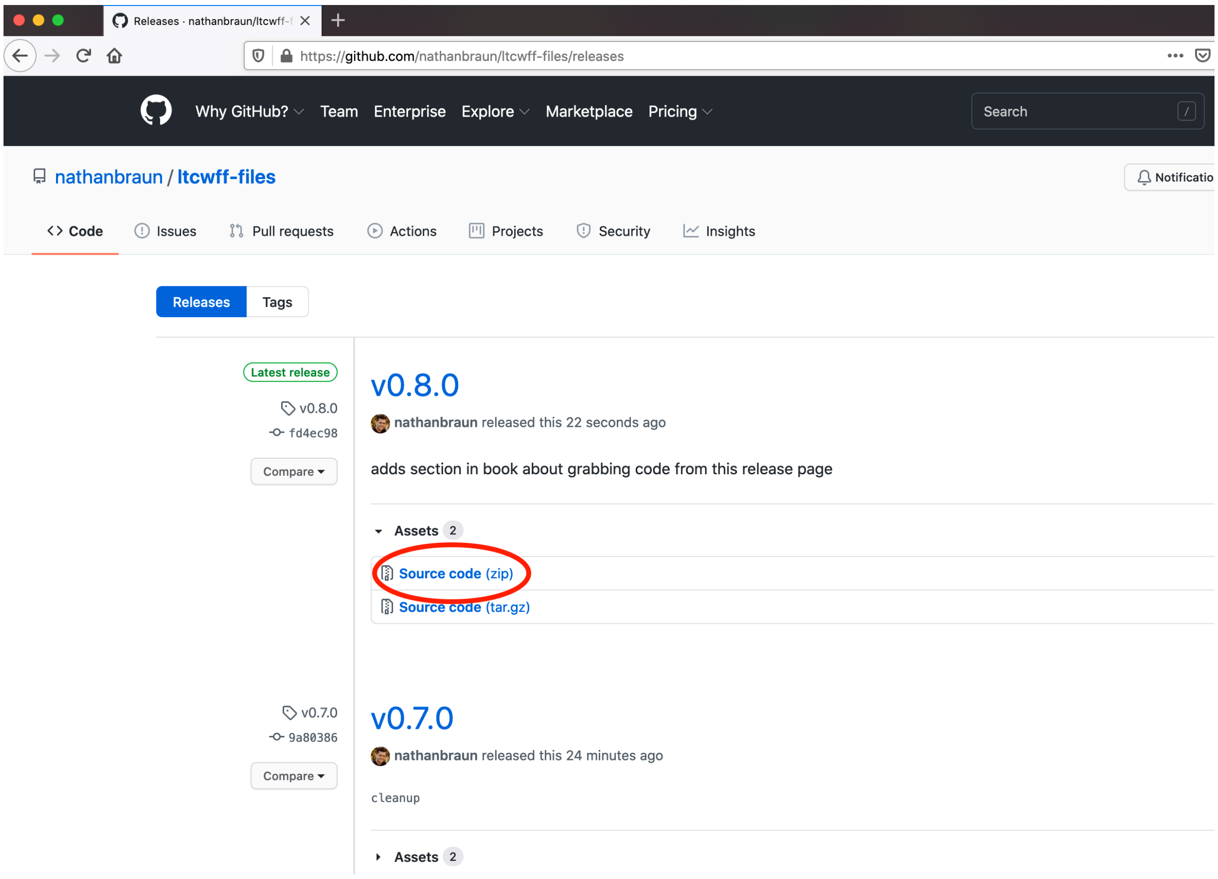
Appendix A: Prerequisites: Tooling

Files Included with this Book

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

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

https://github.com/nathanbraun/code-soccer-files/releases



v0.8.0

nathanbraun released this 22 seconds ago

adds section in book about grabbing code from this release page

v0.7.0

nathanbraun released this 24 minutes ago

```
cleanup
```

Figure 0.1: LTCWFF Files on GitHub

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

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

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

`/Users/nathanbraun/code-soccer-files`

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

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

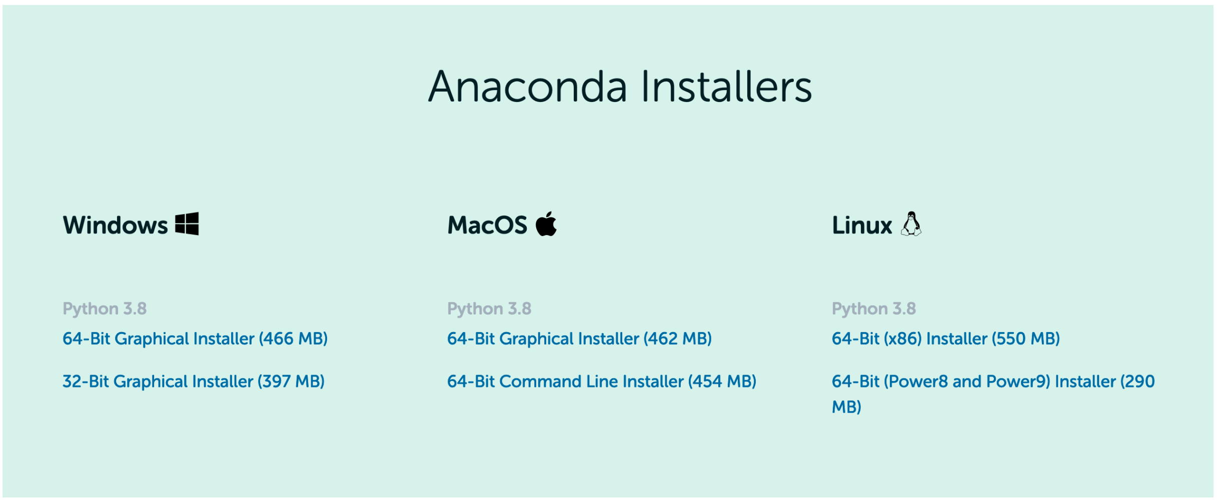
Set these aside for now and we'll pick them up in chapter 2.

Python

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

This book is hands on, and you'll need the ability to run Python 3 code and install packages. If you can do that and have a setup that works for you, great. If you do not, the easiest way to get one is from Anaconda.

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


Anaconda Installers

Windows

Python 3.8
64-Bit Graphical Installer (466 MB)
32-Bit Graphical Installer (397 MB)

MacOS

Python 3.8
64-Bit Graphical Installer (462 MB)
64-Bit Command Line Installer (454 MB)

Linux

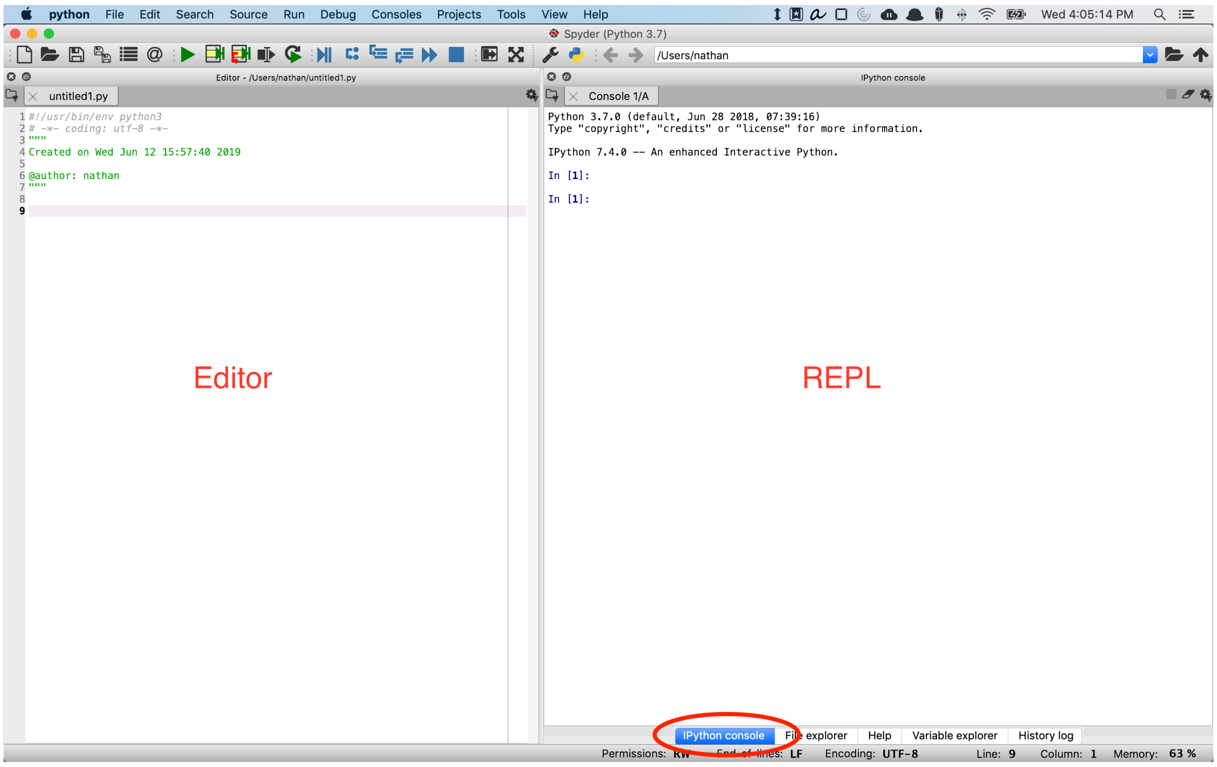
Python 3.8
64-Bit (x86) Installer (550 MB)
64-Bit (Power8 and Power9) Installer (290 MB)

Figure 0.2: Python 3.x on the Anaconda site

3. Then install it[1]. It might ask whether you want to install it for everyone on your computer or just you. Installing it for just yourself is fine.
4. Once you have Anaconda installed, open up Anaconda Navigator and launch Spyder.
5. Then, in Spyder, go to View -> Window layouts and click on Horizontal split. Make sure pane selected on the right side is 'IPython console'.

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

[1]One thing about Anaconda is that it takes up a lot of disk space. This shouldn't be a big deal. Most computers have much more hard disk space than they need and using it will not slow down your computer. Once you are more familiar with Python, you may want to explore other, more minimalistic ways of installing it.


python File Edit Search Source Run Debug Consoles Projects Tools View Help
Wed 4:05:14 PM
Spyder (Python 3.7)
/Users/nathan
Editor - /Users/nathan/untitled1.py
untitled1.py
IPython console
Console 1/A
Editor
REPL
IPython console
File explorer
Help
Variable explorer
History log
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 9 Column: 1 Memory: 63 %

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 12 15:57:40 2019

@author: nathan
"""

```

```
Python 3.7.0 (default, Jun 28 2018, 07:39:16)
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]:

In [1]:
```

Figure 0.3: Editor and REPL in Spyder

Editor

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

What are the differences?

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

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

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

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

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

Console (REPL)

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

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

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

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

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

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

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

And then later:

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

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

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

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

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

Try:

```
In [2]: x = 1

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

ZeroDivisionError: division by zero
```

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

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

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

Using Spyder and Keyboard Shortcuts

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

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

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

F5 will send the entire file to the REPL.

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

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

On a Mac, it's command instead of control:

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