O'REILLY®

Third Edition
Covers Python 3

Head First Python

A Learner's Guide to the Fundamentals of Python Programming

Paul Barry

Early Release
RAW & UNEDITED





A Brain-Friendly Guide

Head First Python

THIRD EDITION

A Learner's Guide to the Fundamentals of Python Programming, A Brain-Friendly Guide

---

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Paul Barry



Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First Python

by Paul Barry

Copyright © 2023 Paul Barry. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Suzanne McQuade
- Development Editor: Melissa Potter
- Production Editor: Beth Kelly
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato

- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- August 2023: Third Edition

Revision History for the Early Release

- 2023-02-08: First Release
- 2023-03-24: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492051299 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Head First Python*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of

or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05122-0

[FILL IN]

Preface

Install the latest Python 3

What you do here depends on the platform you're running, which is assumed to be one of *Windows*, *macOS*, or *Linux*.

The good news is that all three platforms run that latest Python, release 3.10. There's no bad news.



If you are already running release 3.10 or later, move to the next page – you're ready. If you haven't already installed Python or are using an older version, select the paragraph below which applies to you, and read on.

Installing on Windows

The wonderful Python folk at *Microsoft* work hard to ensure the most-recent release of Python is always available to you via the *Windows Store* application. Open the Store, search for "Python", select the

most-recent version, then click the **Get** button. Watch patiently while the progress indicator moves from zero to 100% then, once the install completes, move to the next page – you're ready.

Installing on macOS

The latest Macs ship with older, out-of-date releases of Python. *Don't use these*. Instead, head over to Python's home on the web, https://www.python.org/, then click on the "Downloads" option. The latest release of Python 3 should being to download, as the Python site is smart enough to spot you're connecting from a Mac. Once the download completes, run the installer that's waiting for you in your Downloads folder. Click the **Next** button until there are no more **Next** buttons to click then, when the install is complete, move to the next page – you're ready.

NOTE

There's no need to remove the older preinstalled releases of Python which come with your Mac. This install will supersede them.

Installing on Linux

The *Head First Coders* are a rag-tag team of techies whose job is to keep the *Head First Authors* on the straight and narrow (no mean

feat). The coders love *Linux* and the *Ubuntu* distribution, so that's discussed here.

It should come as no surprise that the latest Ubuntu comes with Python 3 installed and up-to-date. If this is the case, cool, you're all set. If you are using a Linux distribution other than Ubuntu, use your system's package manager to install Python 10 (or later) into your Linux system. Once done, move to the next page – you're ready.

Let's complete your install with two things: a required back-end dependency, as well as a modern, Python-aware text editor.

Python on its own is not enough

In order to explore, experiment, and learn about Python, you need to install a runtime back-end called *Jupyter* into your Python. As you'll see in a moment, doing so is straightforward.

When it comes to creating Python code, you can use just about *any* programmer's editor, but we're recommending you use a specific one when working through this book's material: Microsoft's *Visual Studio Code*, known the world over as **VS Code**.


jupyter

Install the latest Jupyter Notebook back-end

NOTE

Don't worry, you'll learn all about what this is used for soon!

Regardless of the operating system you're running, make sure you're connected to the Internet, open a Terminal window, then type:

```
python3 -m pip install jupyter
```

A veritable slew of status messages whiz by on screen. If you are seeing a message near the end stating everything is "Successfully installed", then you're golden. If not, check the Jupyter docs and try again.



Install the latest release of VS Code

Grab your favorite browser and surf on over to the VS Code download page:

https://code.visualstudio.com/Download

NOTE

Their are alternatives to VS Code, but – in our view – VS Code is hard to beat when it comes to this book's material. And, no, we are *not* part of some global conspiracy to promote Microsoft products!!

Pick the download which matches your environment, then wait for the download to complete. Follow the instructions from the site to install VS Code, then flip the page to learn how to complete your VS Code setup.

Configure VS Code to your taste

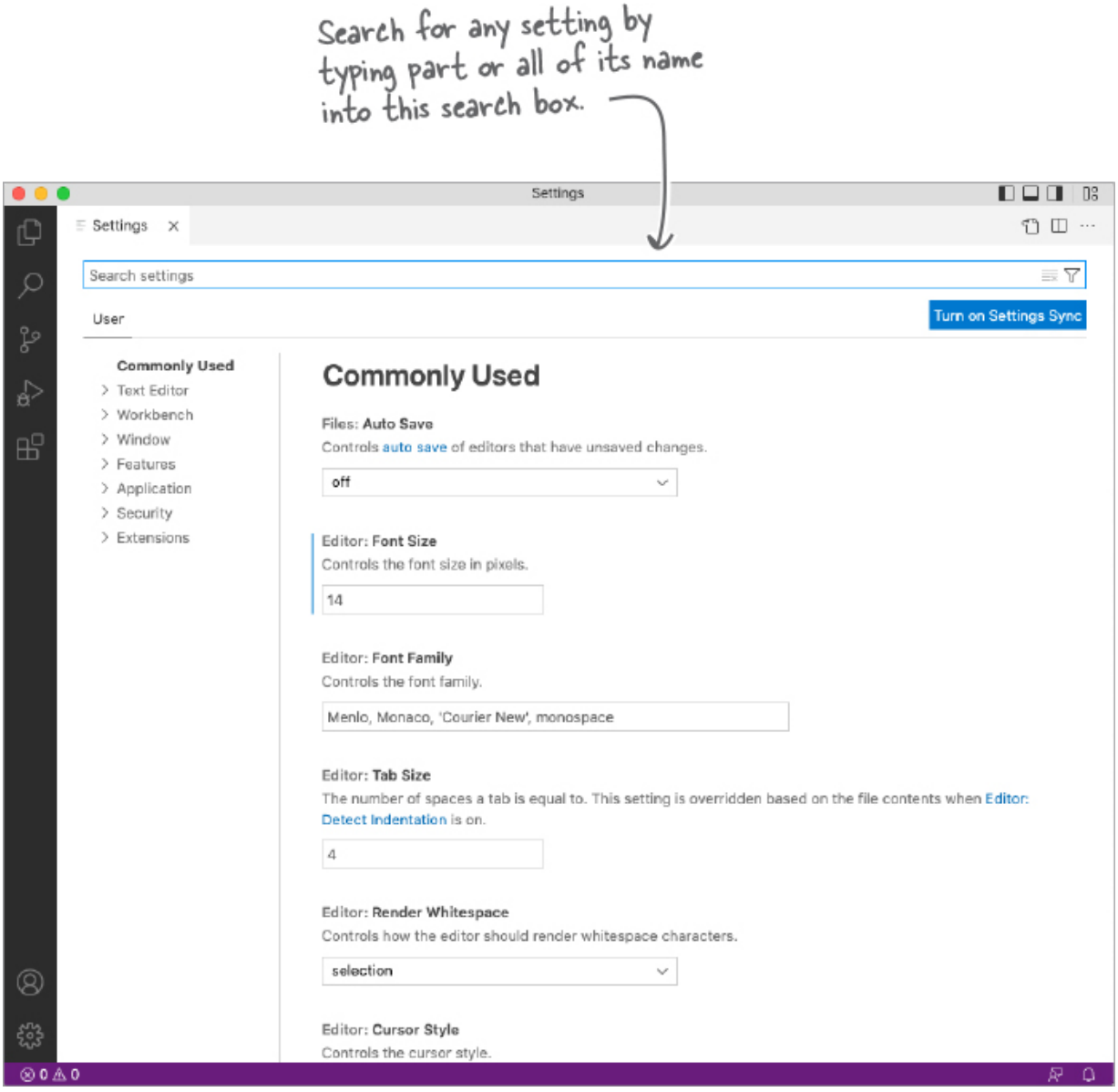


Go ahead and run VS Code for the first time. From the menu, select the **File**, then **Preferences**, then **Settings** to access the editor's settings preferences.

NOTE

On the Mac, start with the “Code” menu.

You should see something like this:



Until you become familiar with VS Code, you may wish to configure your editor to match the settings preferred by the *Head First Coders*. Here are the settings used in this book:

- The *indentation guides* are switched off.
- The editor's *appearance theme* is set to *Light*.
- The editor *minimap* is disabled.
- The editor's *occurrences highlight* is switched off.
- The editor's *render line highlight* is set to **none**.
- The terminal and text editor's *font size* is set to 14.
- The notebook's *show cell status bar* is set to **hidden**.
- The editor's *lightbulb* is disabled.

NOTE

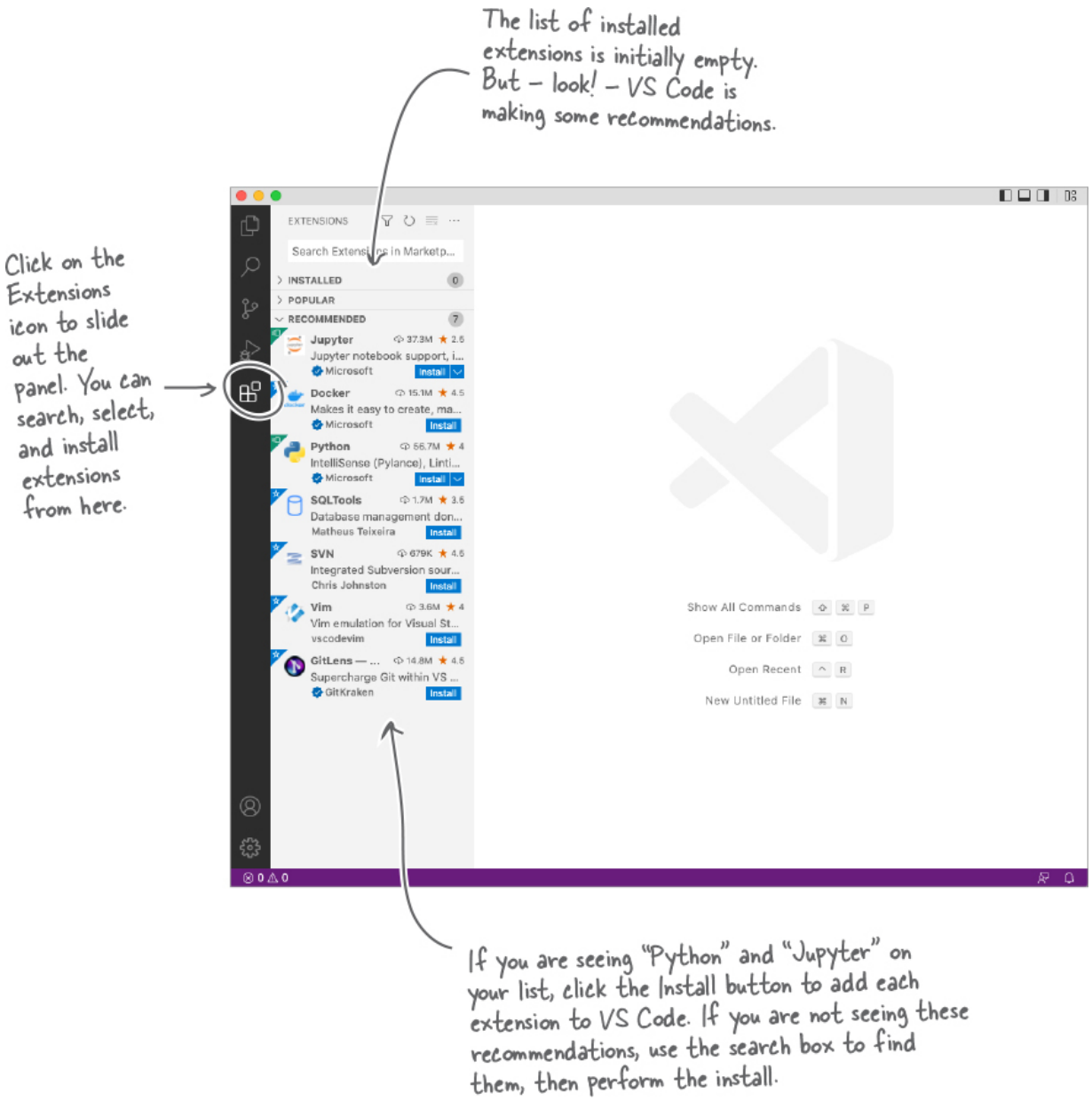
You don't have to use these settings but, if you want to match on your screen what you see in this book, then these adjustments are recommended.

Add 2 required extensions to VS Code



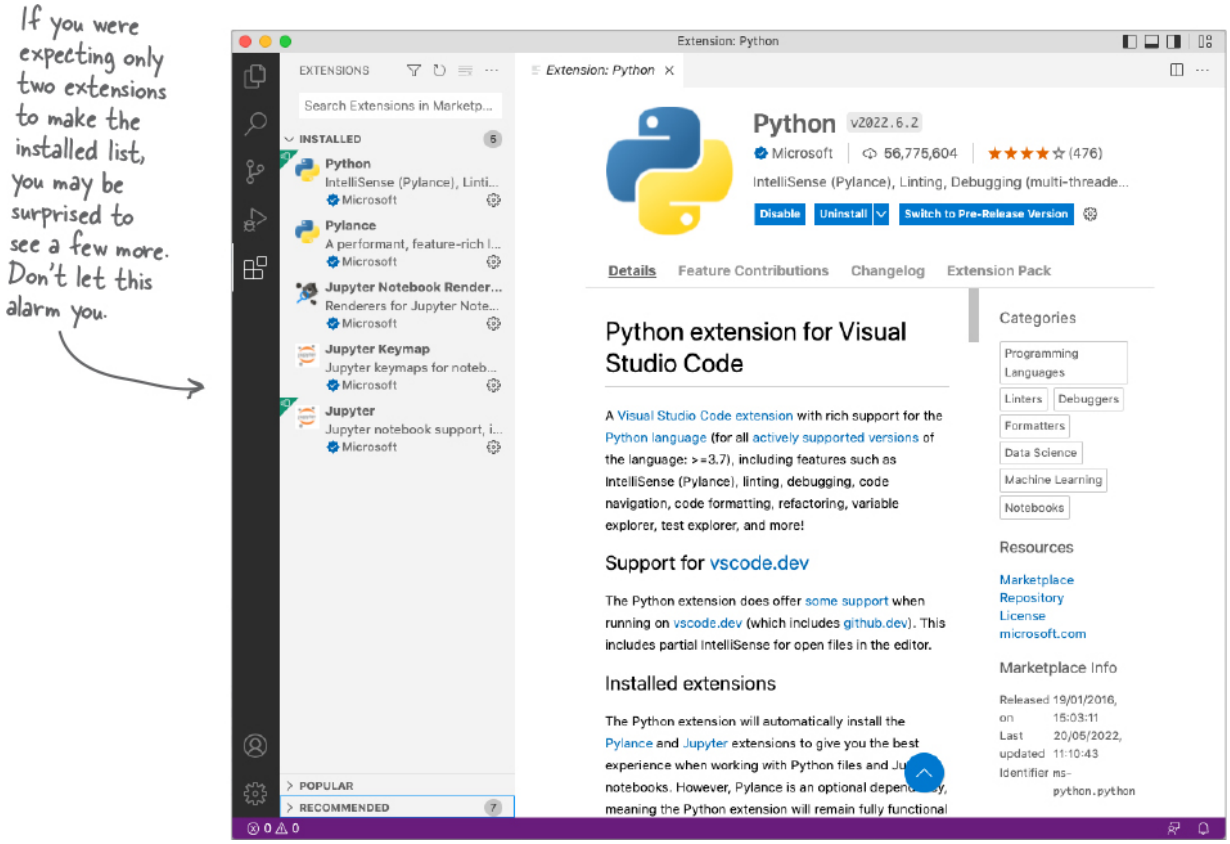
Whereas you are not obliged to copy the recommended editor setup, you absolutely have to install two VS Code extensions, namely **Python** and **Jupyter**.

When you are done adjusting your preferred editor settings, close the Settings tab by clicking the X. Then, to search for, select, and install extensions, click on the Extensions icon to the left of the main VS Code screen:



VS Code's Python support is state-of-the-art

Installing the Python and Jupyter extensions actually results in a few additional VS Code extension installations, as shown here:



These additional extensions enhance VS Code's support for Python and Jupyter over and above what's included in the standard extensions. Although you don't need to know what these extra extensions do (for now), know this: They help turn VS Code into a *supercharged* Python editor.

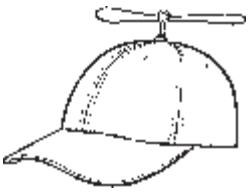






With Python 3, Jupyter, and VS Code installed, you're all set!

GEEK NOTE



Throughout this book you'll encounter technical call-out boxes like this. These *Geek Note* boxes are used to delve into a specific topic in a bit more detail than we'd normally do in the main text. Don't panic if the material in these boxes throws you off your game. They are designed to appease your curious inner nerd. You can safely (and without guilt) skip any *Geek Note* on a first-reading.

NOTE

Yes, this is a Geek Note about Geek Notes (and we'll not be having any recursion jokes, thank you).

Chapter 0: Why Python? Similar But Different

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 0 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



Figure I-1.

Python starts counting from zero, which should sound familiar. In fact, Python has a lot in **common** with other programming languages. There's **variables**, **loops**, **conditionals**, **functions**, and the like. In this, our opening chapter, we take you on a **high-level whistle-stop tour** of Python's basics, introducing the language without getting too much into the weeds. You'll learn how to **create** and **run** code within VS Code and Jupyter Notebook. You'll see how lots of

programming functionality comes **built-in** to Python, which you'll **leverage** to get your job done. You'll also learn that although Python shares a lot of the ideas with other programming languages, how they manifest in your Python code can be, well, **different**. Now, don't get the wrong idea here: we're talking different **good**, not different *bad*. Read on to learn more…

IT'S A HEAD FIRST EXCLUSIVE



Figure I-2.

Today's interview is with a very special guest: **The Python programming language**.

Head First: Let me just say what a pleasure it is to have you here today.

Python: Why, thank you.

Head First: You're regarded as one of the most popular programming languages in use today. Why do you think that is?

Python: (Blushing) My… that's high praise indeed. At a guess, I'd say lots of programmers like me.

Head First: Clearly. But why do you think that is? What makes *you* so different?

Python: I don't think I'm all the different to most other programming languages. I do variables, loops, conditionals, functions, and so on, just like all the others.

Head First: OK, but what accounts for your success?

Python: I guess it's a combination of things.

Head First: Such as…?

Python: Well… I'm told I'm easy to read.

Head First: You mean people know what you're thinking?!?

Python: Oh, that's very funny, and very droll. I am, of course, referring to the fact that my *code* is easy to read.

Head First: And why do you think that is?

Python: For starters, I don't require semi-colons at the end of my statements, and the use of parentheses around *everything* is needed less often than with other programming languages. This makes my code very clean, which aids with readability and understanding.

Head First: What else?

Python: Well, there's the indentation thing. IMHO, I get a lot of *unjustified* flak for this, as I use indentation with whitespace to signal blocks of code. This makes my code look consistent, clean, and readable. On top of this, there's no needless arguing over the "correct" placement of curly braces.

Head First: So… no curly braces in your code then?

Python: There is, actually, just not around blocks. In Python, curly braces surround data *not* code. To indicate blocks, indent with whitespace.

Head First: That's interesting… and I'd imagine programmers used to languages like C, C++, Java, C#, Go, Rust, and so on, struggle with this?

Python: Yes, sadly old habits do die hard. Now, I'm not just saying this, but after you've written code with me, you'll hardly ever notice the whitespace, as it becomes second nature very quickly. It helps, too, that modern editors are Python-savvy, doing the indentation for you. There's rarely a reason to complain.

Head First: Granted, readability is *A Good Thing*™, but there must be more than that to your popularity?

Python: There's also my Standard Library: a collection of included code libraries which help with all manner of programming problems. And, of course, there's PyPI?

Head First: Py…P… what?!?

Python: Py…P…I, which is shorthand for the *Python Package Index*, an online, globally-accessible repository of shareable third-party Python modules. These help programmers tackle every conceivable programming problem you can think of. It's a wonderful community resource.

Head First: So this PyPI thing lets you exploit other programmer's code *without* writing your own?

Python: It sounds kind of seedy when you put it like that…

Head First: Sorry, perhaps "leverage" is a better word?

Python: Yes, that's it: *leverage*. My guiding philosophy is to ensure programmers only ever write new code when then really have to.

Head First: How so?

Python: One word: *built-in*.

Head First: That's two words...

Python: Yes, but there's the hyphen... anyway, the bottom line is I've more than my Standard Library built-in...

Head First: ... I'm waiting...

Python: I'm taking about my *other* built-ins. For starters, there's the BIFs, my built-in functions. These are little powerhouses of generic functionality that work in lots of places and situations.

Head First: Provide our readers with a quick example.

Python: Consider **len**, which is shorthand for "length". Give **len** an object, and it reports how big the object is.

Head First: Thank heavens I'm sitting down.

Python: Ha ha. I know it sounds trivial, but **len**'s a bit of a wonder. Give **len** a list, and it tells you how many objects are in the list. Give it a string, and **len** tells you how many characters are in the string, and so on and so forth. If an object can report its size, **len** can tell you what it is.

Head First: So **len** is polymorphic?

Python: Oooooh, that's fancy lingo, isn't it? 😉 But, yes, like a lot of my BIFs, **len** works with many different objects of many different types. At a push, I'd agree that **len** is polymorphic.

Head First: So what else makes you popular?

Python: My engaging personality.

Head First: More like your *frivolous* personality. Come on, let's try to keep this business-like.

Python: Surely I'm allowed to have a bit of fun? After all, I am named after a 1960's British comedy troupe.

Head First: Seriously? Now you're yanking my chain.

Python: I'm telling you not a word of a lie. Look it up (see: https://docs.python.org/3/faq/general.html#why-is-it-called-python). I'm not named after a snake. I'm named after *Monty Python's Flying Circus*.

Head First: If you say so...

Python: And I'm partial to spam with *everything*.

Head First: OK, now you're just being silly.

Python: OK, I promise I'll behave. Go on, ask me another question.

Head First: What's your deal with data?

Python: I just *love* data. Whether it's working with my built-in data structures (these are really cool, BTW), or talking to databases, or

grabbing data off the web, I'm your go-to data-munging programming language.

Head First: I guess your love of data reflects your #1 position in the machine learning world, eh?

Python: I guess so. To be honest, though, I find all that ML stuff a bit highfalutin. I'm just as happy running your functions, ripping through your loops, and helping you to get your work done.

Head First: But surely the merging of Python, data, and deep learning are a match made in heaven?

Python: Yes, I guess. But, not everyone needs to do that sort of thing. Don't get me wrong, I'm thrilled I'm a mover'n'shaker in the AI field, but it's not like that's *all* I do.

Head First: Care to expand?

Python: Sure. At my core, I'm a plain-ol' general purpose programming language, which can be put to many uses. I'm equally happy running your latest webapp as I am running on the *Mars Rover*. It's all just code to me. Beautifully formatted, easy to read, *code*. That's what it's all about.

Head First: I couldn't agree more. Thanks, Python, for chatting with us today.

Python: You're very welcome. See ya!



Figure I-3.

For sure. But, that's not all.

As the interview with Python confirms, there's a bunch of reasons for Python's popularity. We've listed the takeaways we gleaned from the interview at the bottom of this page.

There are, of course, other reasons to consider Python as your next favorite programming language, but our list is enough to be going on with for now.

Let's spend some time considering these takeaways in more detail. Once you've surveyed our list, grab a pencil – yes, a *pencil* – and meet us at the top of the next page!

1. **Python code is easy to read.**

2. **Python comes with a Standard Library.**

3. **Python has practical, powerful, and generic built-in functions (BIFs).**

4. **Python comes with built-in data structures.**

5. **Python has the Python Package Index (PyPI).**

6. **Python doesn't take itself too seriously.**

NOTE

Don’t underestimate the importance of this last one.

LOOK HOW EASY IT IS TO READ PYTHON

You don't know much about Python yet, but we bet you can make some pretty good guesses about how Python code works. Take a look at each line of code below and note down what you think it does. We've done the first one for you to get you started. Don't worry if you don't understand all of this code yet – the answers are on the next page, so feel free to take a sneaky peek if you get stuck.

`suits = ["Clubs", "Spades", "Hearts", "Diamonds"]`	A variable called "suits" is assigned an array of 4 strings, representing the suits in a deck of cards.
`faces = ["Jack", "Queen", "King", "Ace"]`	
`numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]`	
`import random`	
`def draw():`	
`    which_suit = random.choice(suits)`	
`    which_type = random.choice([faces, numbered])`	
`    which_card = random.choice(which_type)`	
`    return which_card, "of", which_suit`	
`draw()`	
`draw()`	
`draw()`	

Figure I-4.

LOOK HOW EASY IT IS TO READ PYTHON

You don't know much about Python yet, but we bet you could make some pretty good guesses about how Python code works. You were to note down what you thought each line of code below did. We did the first one for you to get you started. How do your notes compare to ours?

`suits = ["Clubs", "Spades", "Hearts", "Diamonds"]`	A variable called "suits" is assigned an array of 4 strings, representing the suits in a deck of cards.
`faces = ["Jack", "Queen", "King", "Ace"]`	Another variable called "faces" assigned an array of the 4 face cards.
`numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]`	And here's another array variable called "numbered" which represents the deck's numbered cards.
`import random`	A library is included to provide support for randomness.
`def draw():`	The empty parentheses are a bit of a give away. This looks like the start of a function.
`    which_suit = random.choice(suits)`	Randomly select a suit and assign it to a new variable called "which_suit".
`    which_type = random.choice([faces, numbered])`	Assign either the "faces" or "numbered" array to the "which_type" variable. Do so randomly.
`    which_card = random.choice(which_type)`	Randomly select a card from the "which_type" array – it'll be either a face or a number value.
`    return which_card, "of", which_suit`	Return the randomly selected card, then word "of", then the randomly selected suit
`draw()`	Invoke the "draw" function to randomly generate a drawn card from the deck.
`draw()`	Ditto.
`draw()`	Ditto.

Figure I-5.

Getting ready to run some code

There's a tiny bit of house-keeping to work through *before* you get run any code.

To help keep things organized, let's create a folder on your computer called `Learning`. You can put this folder anywhere on your hard-drive, so long as you remember *where* you put it, as you are going to use it *all the time*.

With your `Learning` folder created, start VS Code.



Figure I-6.

RELAX



Figure I-7.

Don't panic if you haven't installed VS Code.

That's no biggie. Pop back to this book's Intro and work through the pages beginning at **Install the latest Python** 3. You'll find the instructions to install VS Code and some required extensions there.

Do this now, then pop back here when you're ready. We'll wait...

NOTE

Don't feel bad if you skipped the Intro. You aren't the first to do this, and won't be the last. 😉

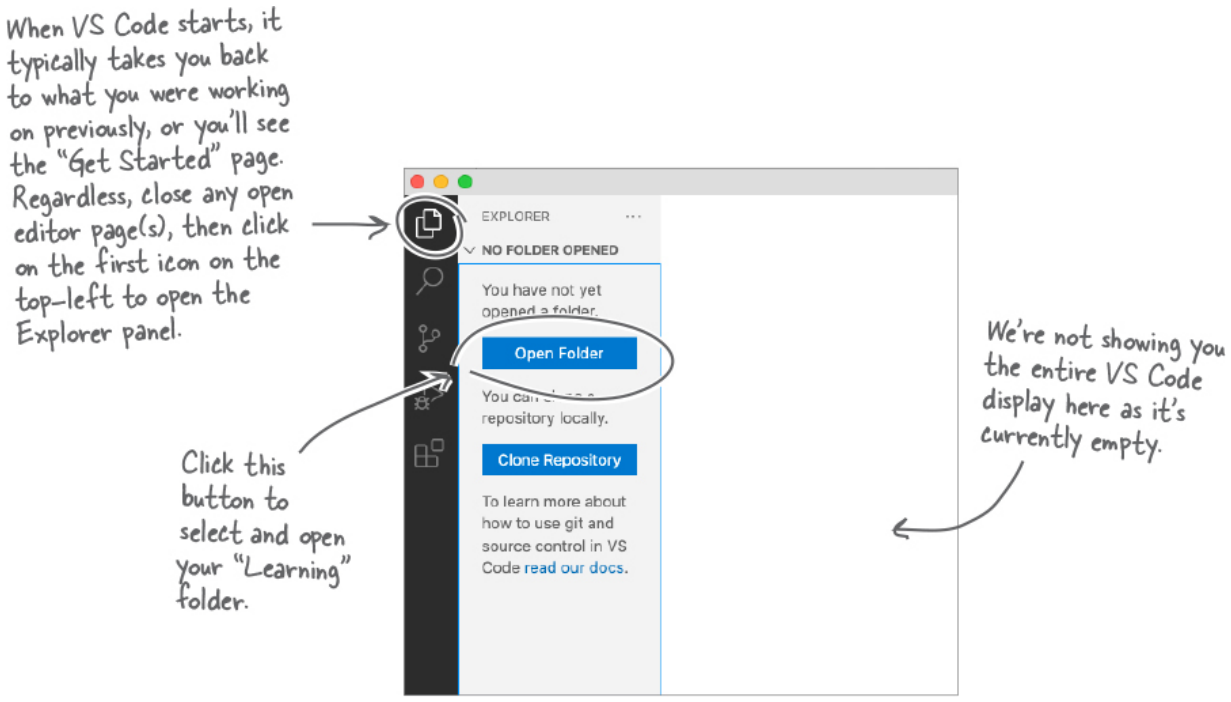


Figure I-8.

Each time you work with VS Code in this book, you'll open your `Learning` folder as needed. **Do this now before continuing**.

Preparing for your first Jupyter experience

OK. You're running VS Code, and you've opened your `Learning` folder. Let's create a new notebook by first selecting the **File** menu, then selecting the **New File…** menu option. You'll be presented with three choices:



Figure I-9.

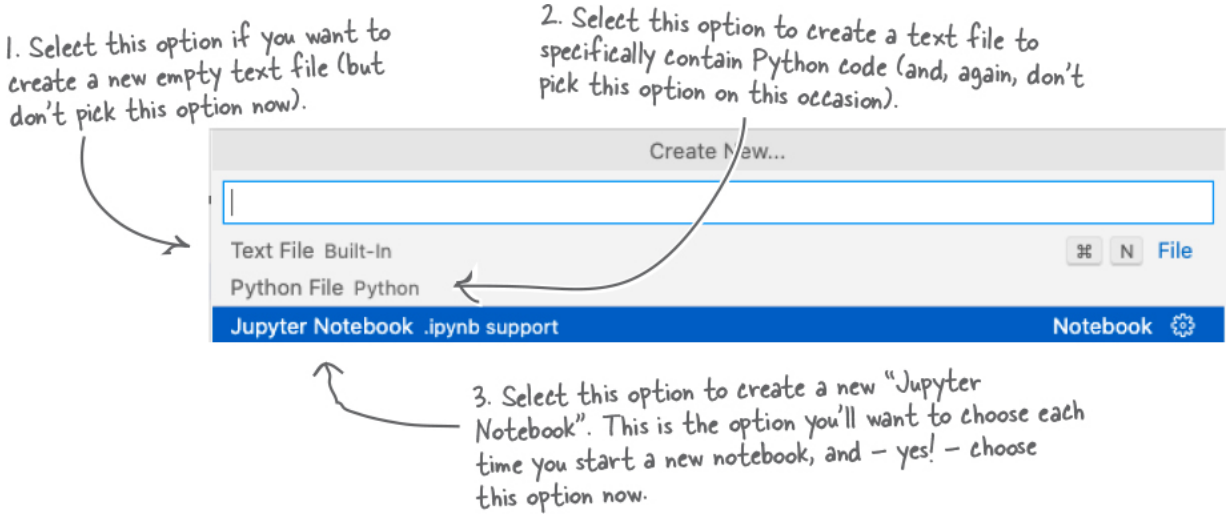


Figure I-10.

VS Code creates and opens a new, *untitled* notebook called `Untitled-1.ipynb`, which appears on screen, and looks something like this:

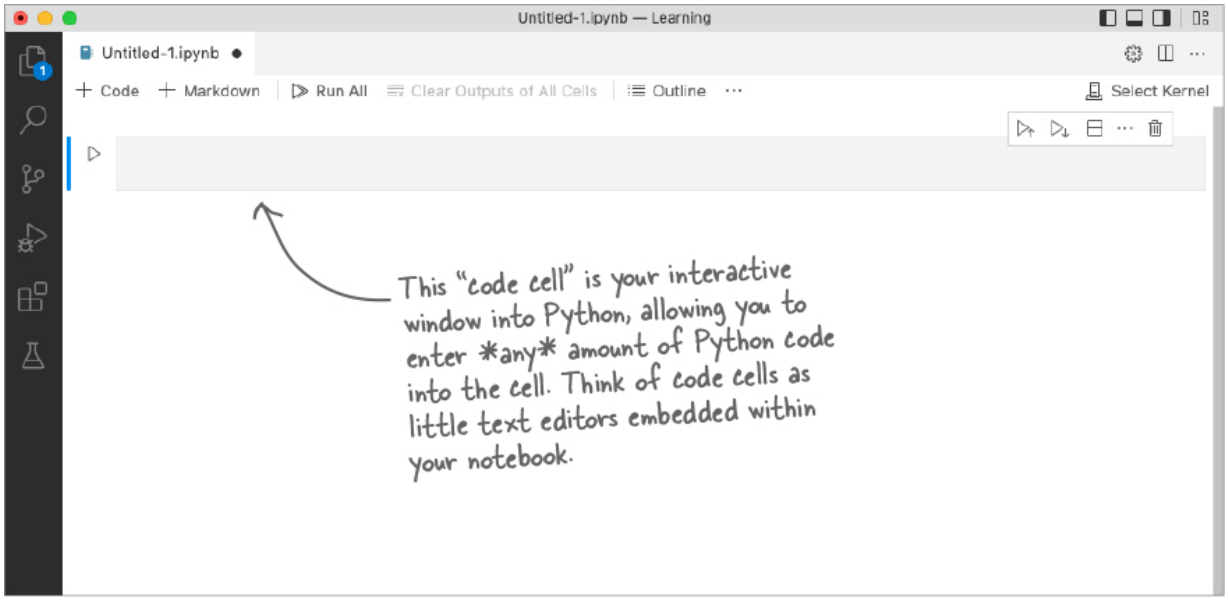


Figure I-11.

Drum roll, please. You're now ready to type in and run some Python code.

TEST DRIVE



Figure I-12.

Your cursor is blinking in that empty code cell. Go ahead and type in the first three lines of code from the card deck example from the start of this chapter. Here's the first three lines of code again:

Remember: It's a little embedded editor. You can type as much (or as little) Python code as you like in here.

```
suits = ["Clubs", "Spades", "Hearts", "Diamonds"]
faces = ["Jack", "Queen", "King", "Ace"]
numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Figure I-13.



Figure I-14.

No need. Just press Shift+Enter.

Jupyter comes with keyboard shortcuts designed to make your interactions with your notebook as easy as… say… pressing the **Shift** and **Enter** keys *together*.

NOTE

When you see "Shift+Enter" in this book, press and hold down the Shift key, then tap the Enter key (before releasing both).

Pressing Shift+Enter runs the cell's code

When you press **Shift+Enter**, the code in the currently selected cell runs. The *focus* then moves to the next cell in your notebook. If no "next cell" exists, Jupyter creates a new one for you:

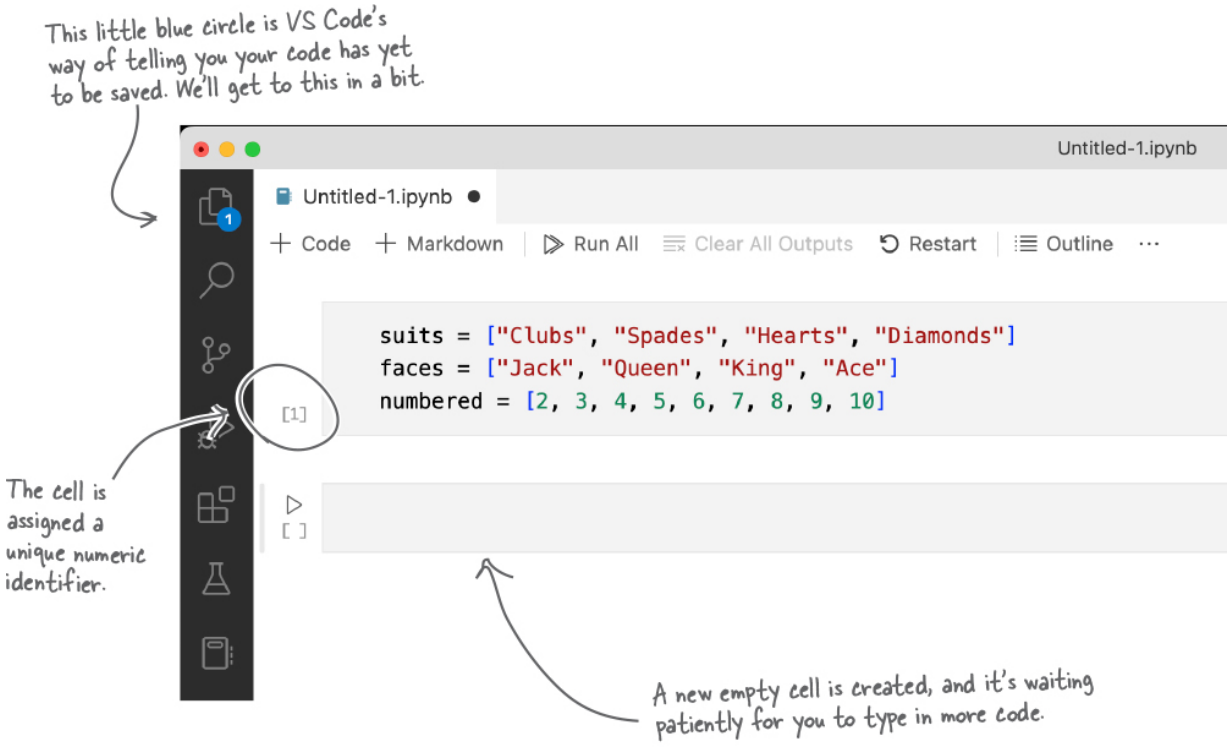


Figure I-15.



Figure I-16.

The code ran, but produced no output.

Those three lines of code are examples of Python's variable assignment. The name of the variable is on the left of the `=` operator, and its value's on the right.

Those variable's values are *defined* by this code.

EXERCISE



Figure I-17.

Your notebook is waiting for more code. Let's get in a little bit of practice using VS Code by adding the following code to your notebook:

1. Type `import random` into your waiting cell, then press **Shift+Enter**.
2. Type the code for the `draw` function into the next cell, then press **Shift+Enter** to run that cell, too. Here's a copy of the `draw` function's code from earlier:

```
def draw():
    which_suit = random.choice(suits)
    which_type = random.choice([faces, numbered])
    which_card = random.choice(which_type)
    return which_card, "of", which_suit
```

Note the use of indentation to denote the block of code associated with the function.

Do you recall what this code does?

Figure I-18.

THERE ARE NO DUMB QUESTIONS

Q: I'm really surprised those first three lines of code ran at all. Surely those three variables, `suits`, `faces`, and `numbered`, need to be predeclared with some sort of type?

A: That may be how some other, more traditional programming languages roll, but not Python. Variables come into existence the moment they are assigned a value, and that value can be of any type. There's no need to predeclare type information, as Python works this all out dynamically at run-time.

Q: Does it matter how I enter my indentation? Can I use the tab key, the space key, or can I use both?

A: We'd love to tell you that it doesn't matter, but it does. When it comes to indenting your Python code, you can use spaces or tabs, but **not both** (so, no mixing'n'matching). Why this is so is kind of technical, so we're not going to get into the weeds on this right now. Our best advice is to configure your editor to automatically replace taps of the tab key with four spaces (and we're pretty sure VS Code defaults to this behavior). And, yes, there's it a *convention* in the Python programming community to indent by four spaces, not by two, and not by eight. You can, of course, ignore this convention if you so wish, but note that most other Python programmers are hard-wired to expect indentation by four spaces. Of course, there's always the

rebels who insist on doing their own thing (we're looking at *you*, *Anvil.Works*).

Q: So, pressing Enter doesn't run my code? I have to keep using the Shift+Enter combination?

A: Yes, as you are operating within a Jupyter Notebook. Remember: each cell in a notebook is like a little embedded editor. Pressing the **Enter** key terminates the current line of code, then opens up a new line (for you to enter more code). As the **Enter** key is already taken, the Jupyter folk had to come up with another way to run a code cell, settling on **Shift+Enter**.

BTW: On the last page of this chapter, you'll find a handy cut-out cheatsheet of our most-useful Jupyter keyboard shortcuts. For now, **Shift+Enter** is all you need to know.

EXERCISE SOLUTION



Figure I-19.

You were asked to add more code to your notebook. Here's what your notebook should look like now:

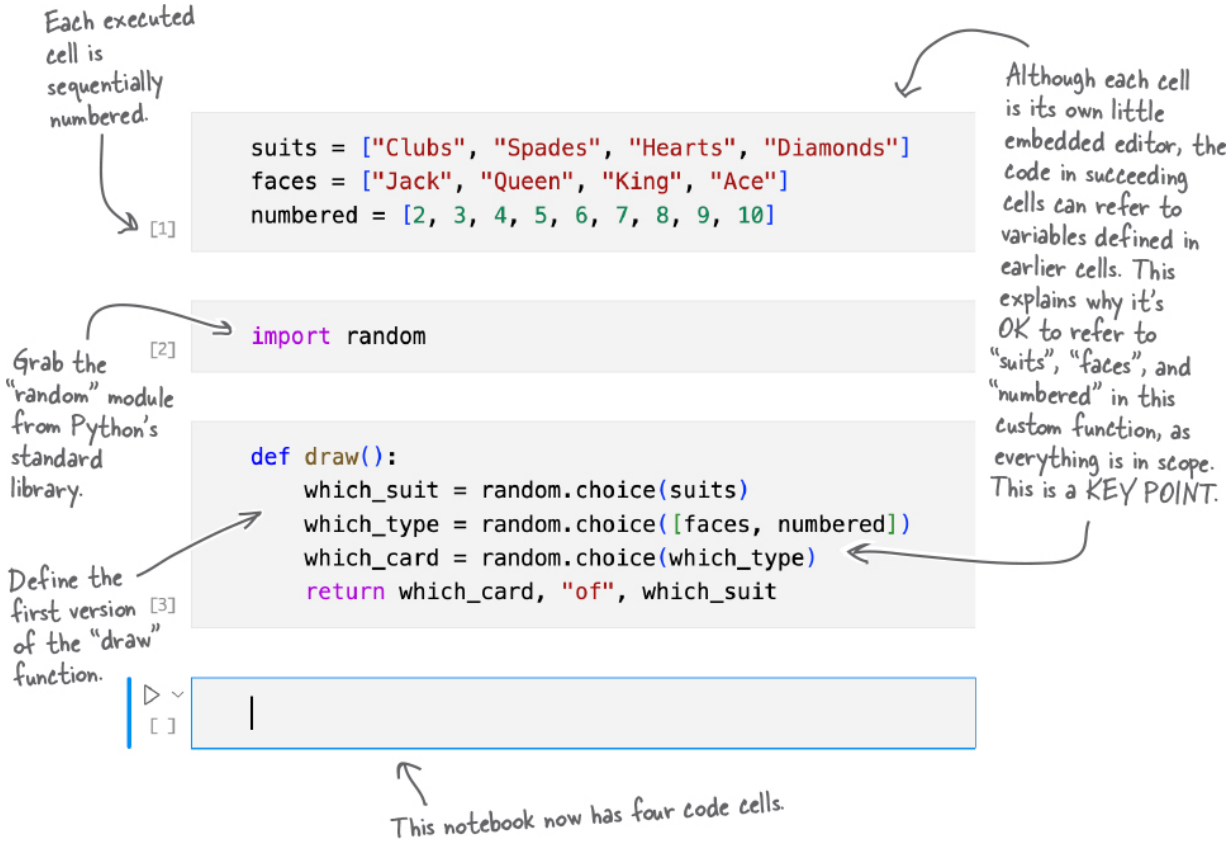


Figure I-20.



Figure I-21.

There's still no output, but this code is executing.

That **import** statement has pulled-in the `random` library, and the **def** statement has defined the `draw` function. Let's take that function for a spin to see what happens...

TEST DRIVE



Figure I-22.

Let's draw some cards from your card deck.

In your next empty code cell, type `draw()` then press **Shift+Enter**.

We did this in three cells to confirm the code is producing random cards, and here's what we saw:

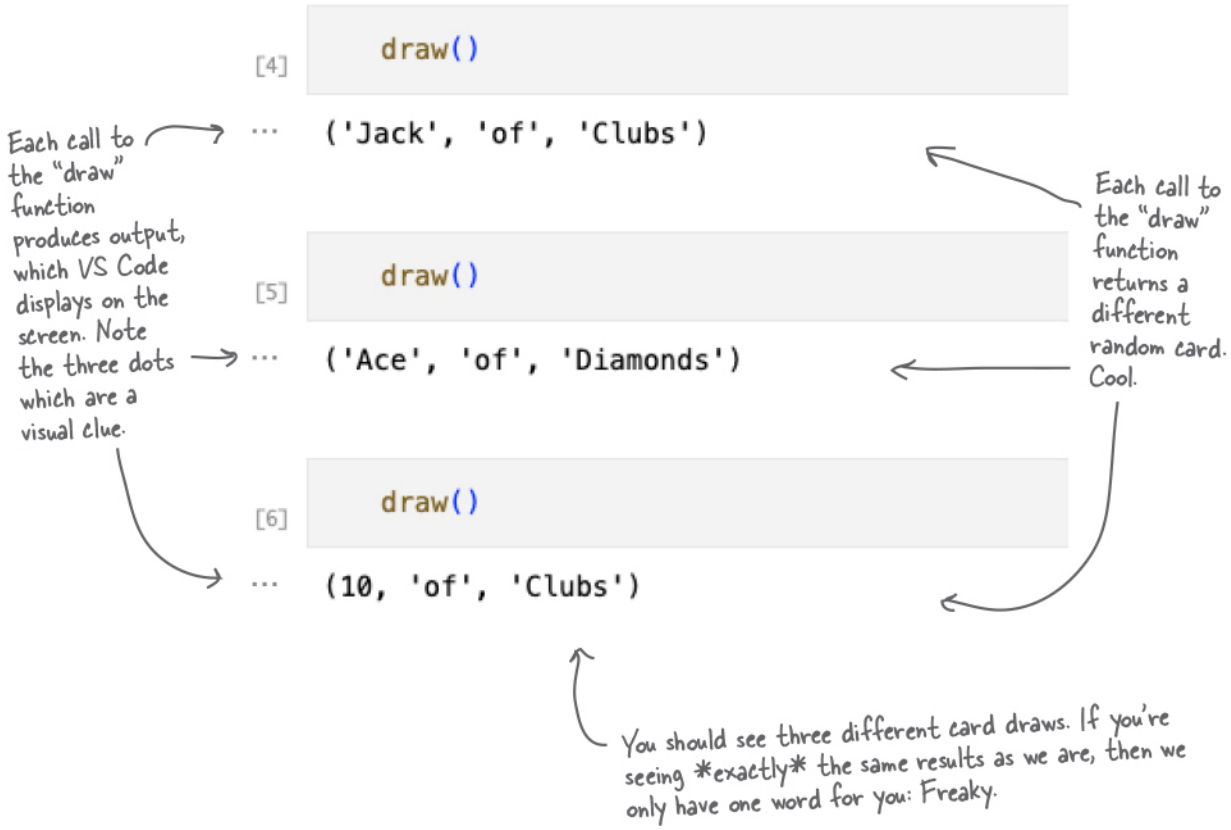


Figure I-23.

So... Python code really is easy to read... and run

Besides Jupyter, there are other ways to run Python code, and you'll learn about some of them as you work through this book. However, using VS Code with the Jupyter extension is – in our view – the *perfect* way to read, run, experiment, and play with Python code when first learning the language. So get ready to spend *a lot* of time in Jupyter and VS Code.

Before moving on, take a moment to select **File** then **Save** from the VS Code menu to save your notebook under the name `Cards.ipynb`.

NOTE

Do this now!

What if you want a bunch of cards?

Your `draw` function is a great start, drawing one card from the deck each time the function is executed. But, what if you want to draw five cards? Or ten? Or twenty? Or all fifty-two, for that matter?

Although you'd be a little off your head to suggest manually invoking your `draw` function as many times as is needed, most programmers instead reach for a loop. You'll learn more about Python's loops later in this book. For now, here's how you'd use Python's **for** loop to execute some code a fixed number of times:

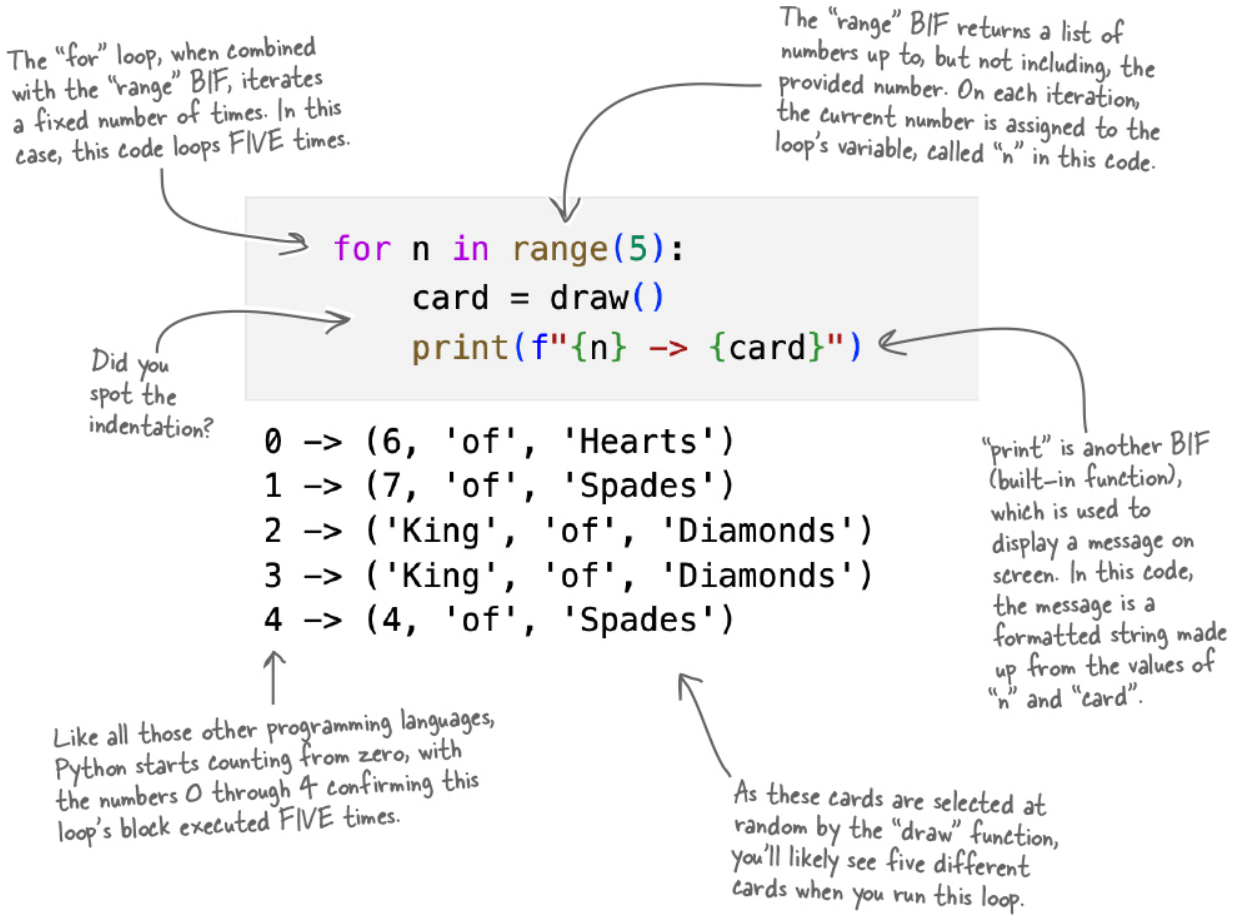


Figure I-24.

This all looks fine, in that the **for** loop's indented block ran five times. But, can you spot a problem with the output?

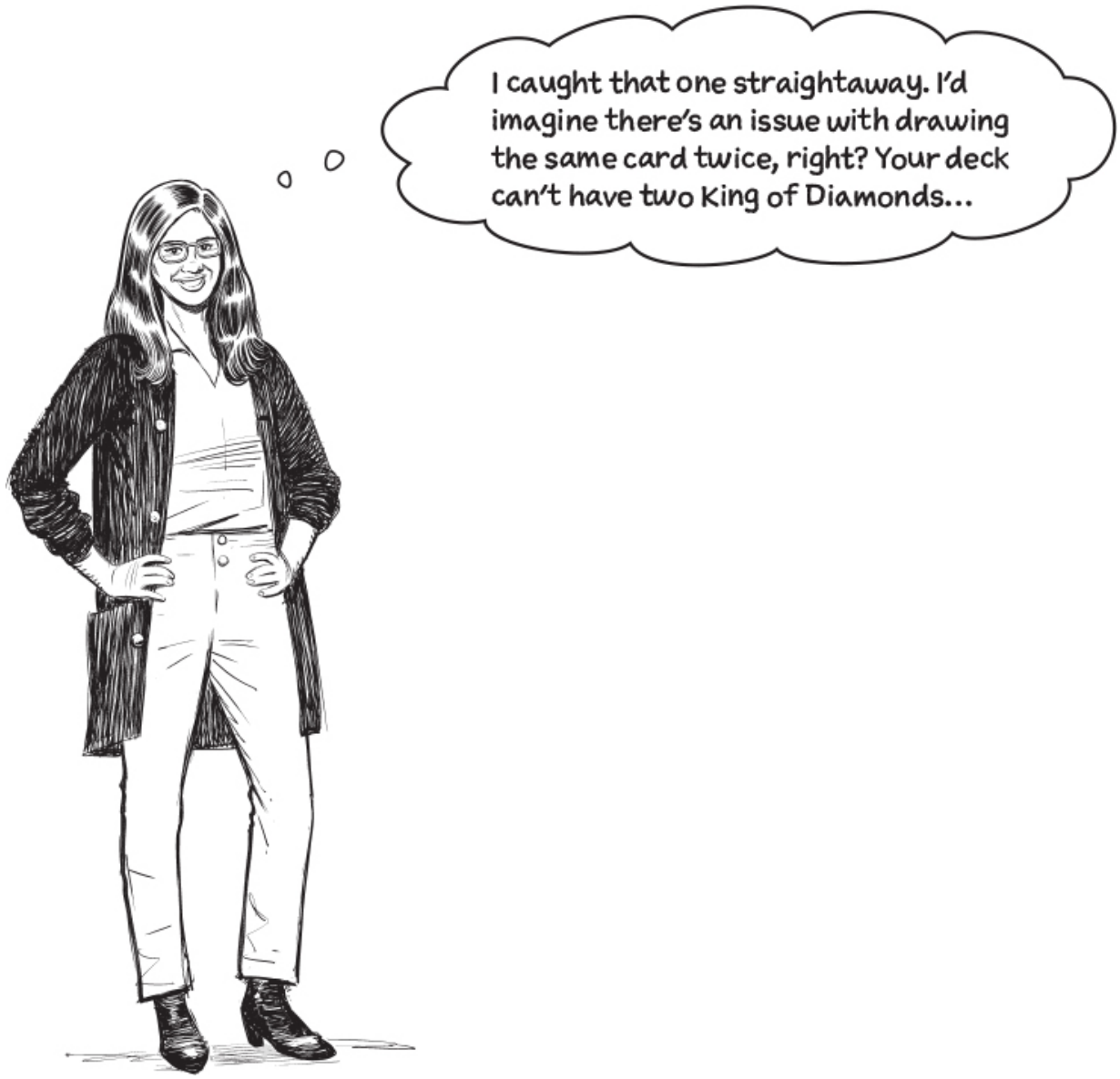


Figure I-25.

Yes, the loop code drew the same card twice.

Of course, the assumption here is that a drawn card is not put pack into the deck of cards. So... unless your playing with “magic cards” (or someone’s up to no good), the fact that this code randomly selects the same card more than once isn’t likely what’s required.

If your goal is to model a deck of cards, the current `draw` function isn't up to snuff, is it?

BRAIN POWER



Figure I-26.

If you were asked to come up with a data structure which better models a deck of cards, which data structure would you choose?

The Big 4: list, tuple, dictionary, and set

Python's excellent built-in support for data structures is legendary, and is often cited as the main reason most Python programmers *love* Python.

As this is your opening chapter, we're not going to overload you with any sort of in-depth discussion of these data structures right now. There are lots of pages (entire chapters, in fact) dedicated to *The Big 4* later in this book.

Although we haven't called out their use specifically, you have already encountered lists *and* tuples. While rather cheekily referring to these as *arrays* earlier, each of these are in fact a bona fide, honest to goodness, Python **list**:

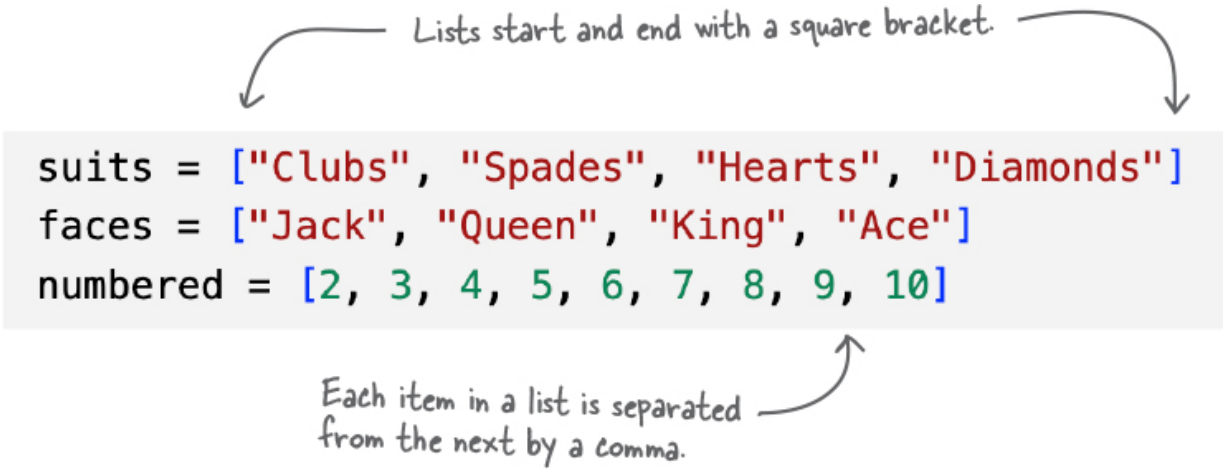


Figure I-27.

You've also seen your fair share of tuples, too. Every time you invoke the `draw` function, it gives you back a **tuple**:

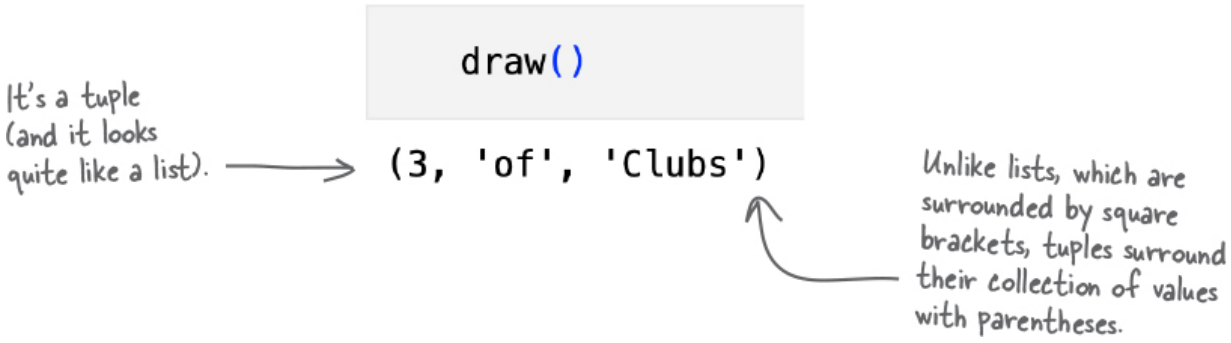


Figure I-28.

You'd be forgiven for thinking tuples look at little weird, and we'd have to agree that we think they look a little weird, too. Don't let this worry you.

You'll learn more about both lists and tuples later in this book. Although both lists and tuples have their uses, they are not a great fit when it comes to modelling a deck of cards. Some other data structure is needed here. But, which one?

NOTE

Hint: There is a big clue in the title of this page.

Model your deck of cards with a...

Although Python's **dictionary** data structure is *hugely* popular, it primarily provides for lookup functionality (which features *lots* later in

this book). Despite this, and like a list and tuple, a dictionary isn't the best fit for modelling your deck of cards.

Which leaves you with **set**.

Sets in Python are likes sets from Math class: they contain a collection of unique values where duplicates are *not* allowed. As the code snippet below shows, when you use a Python set, you inherit a method called **add** which – brace yourself – adds a item to an existing set.

Take a look at this loop which uses the three lists from the previous page to construct a deck of cards.

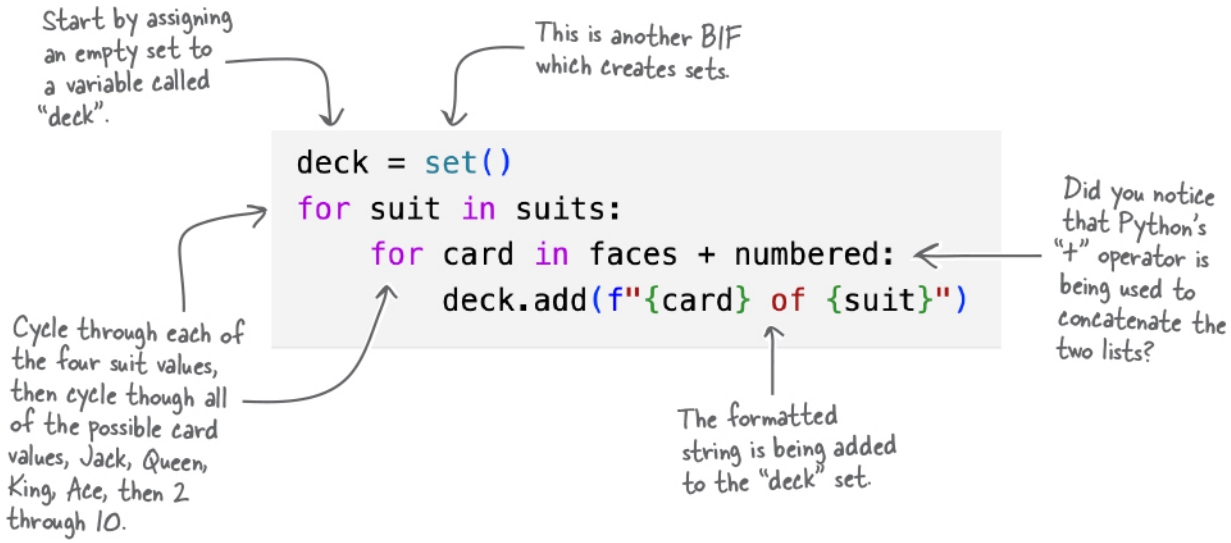


Figure I-29.

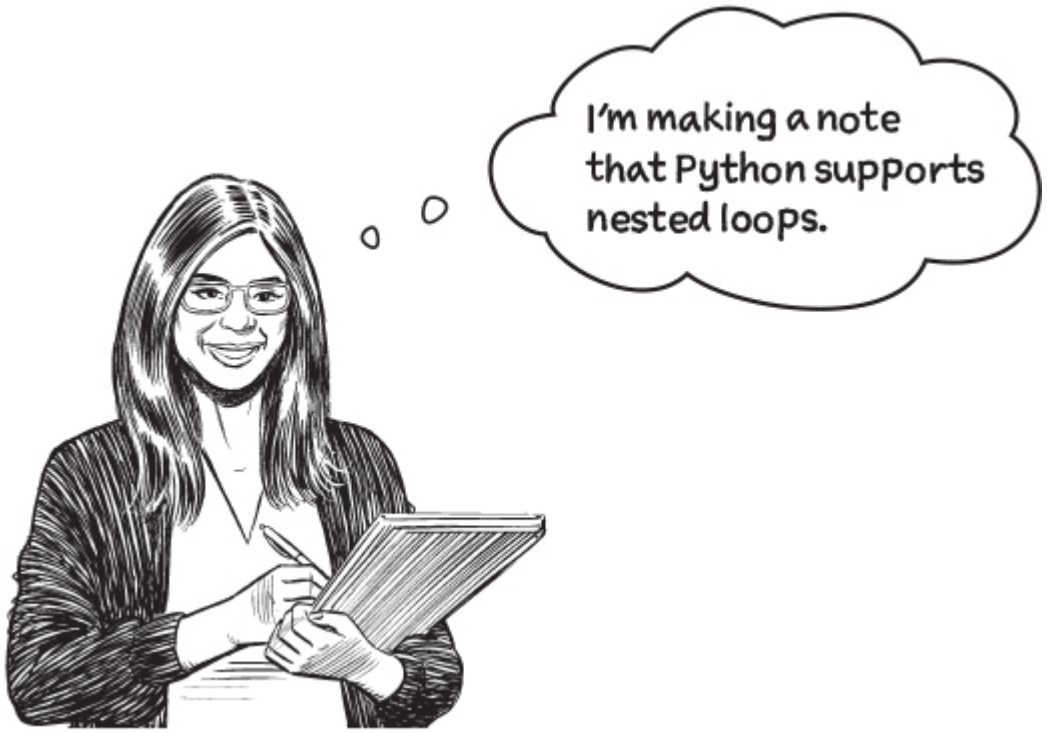


Figure I-30.

That shouldn't come as a surprise.

The block of code associated with any loop can contain any other code, including another loop. That's a given. But, did you notice how Python's use of indentation makes it easy to work out which block belongs where?

Use BIFs to learn more about your variables

Python's built-in functions have many uses. Although you're less than twenty pages into this chapter, you've already learned a little about **len**, **print**, **range**, and **set**.

We know you've been waiting patiently for this... so let's see **len** in action:



Figure I-31.

As you can no doubt imagine, **len** gets a lot of use. However, the most used BIF is very likely **print**, which – among other things – displays objects on screen. Let's see what happens when **print** is asked to display your `deck` variable:

```
print(deck)
```

```
{'Queen of Diamonds', 'Jack of Spades', '3 of Hearts', '10 of Diamonds',
'Queen of Hearts', 'Ace of Hearts', 'Jack of Diamonds', '2 of Diamonds',
'4 of Diamonds', '8 of Spades', '9 of Diamonds', '2 of Spades', '10 of
Spades', '6 of Spades', '9 of Clubs', 'King of Clubs', '8 of Diamonds', '7
of Diamonds', '2 of Clubs', '7 of Clubs', '5 of Diamonds', '10 of Clubs',
'Jack of Clubs', 'Queen of Spades', '5 of Spades', '3 of Spades', '4 of
Clubs', '4 of Spades', 'Ace of Clubs', '9 of Spades', 'Jack of Hearts',
'King of Hearts', 'King of Spades', 'King of Diamonds', 'Queen of Clubs',
'10 of Hearts', '3 of Clubs', '4 of Hearts', '5 of Hearts', 'Ace of
Diamonds', 'Ace of Spades', '3 of Diamonds', '7 of Spades', '6 of Hearts',
'7 of Hearts', '8 of Hearts', '8 of Clubs', '2 of Hearts', '6 of
Diamonds', '9 of Hearts', '6 of Clubs', '5 of Clubs'}
```

Trust us: there are 52 items in this "deck", one for each possible card in the set.

You might be somewhat disappointed to note that your "deck" set is unordered, even though your loop code added items to the set in a very specific order. Note that sets are inherently unordered, as their main function is to provide a data structure which forbids duplicates (and does set-type things).

Figure I-32.

Chaining BIFs to get more done

You can use more than one BIF at a time (should you need to). A great BIF to know about is **sorted**, which returns a sorted copy of any object's data.

The **sorted** BIF is often combined (or *chained*) with the **print** BIF to display an ordered version of some data. Care is needed, though, as the ordered data returned by **sorted** is always a list, regardless of what's being sorted. This means using **sorted** on a set does *not* produce a sorted set, it produces a *sorted list* made up from the data in

the original, *unordered* set. The original set is unchanged, in that it's still a set, which is still unordered.

Remember: "BIF" is shorthand for "built-in function".

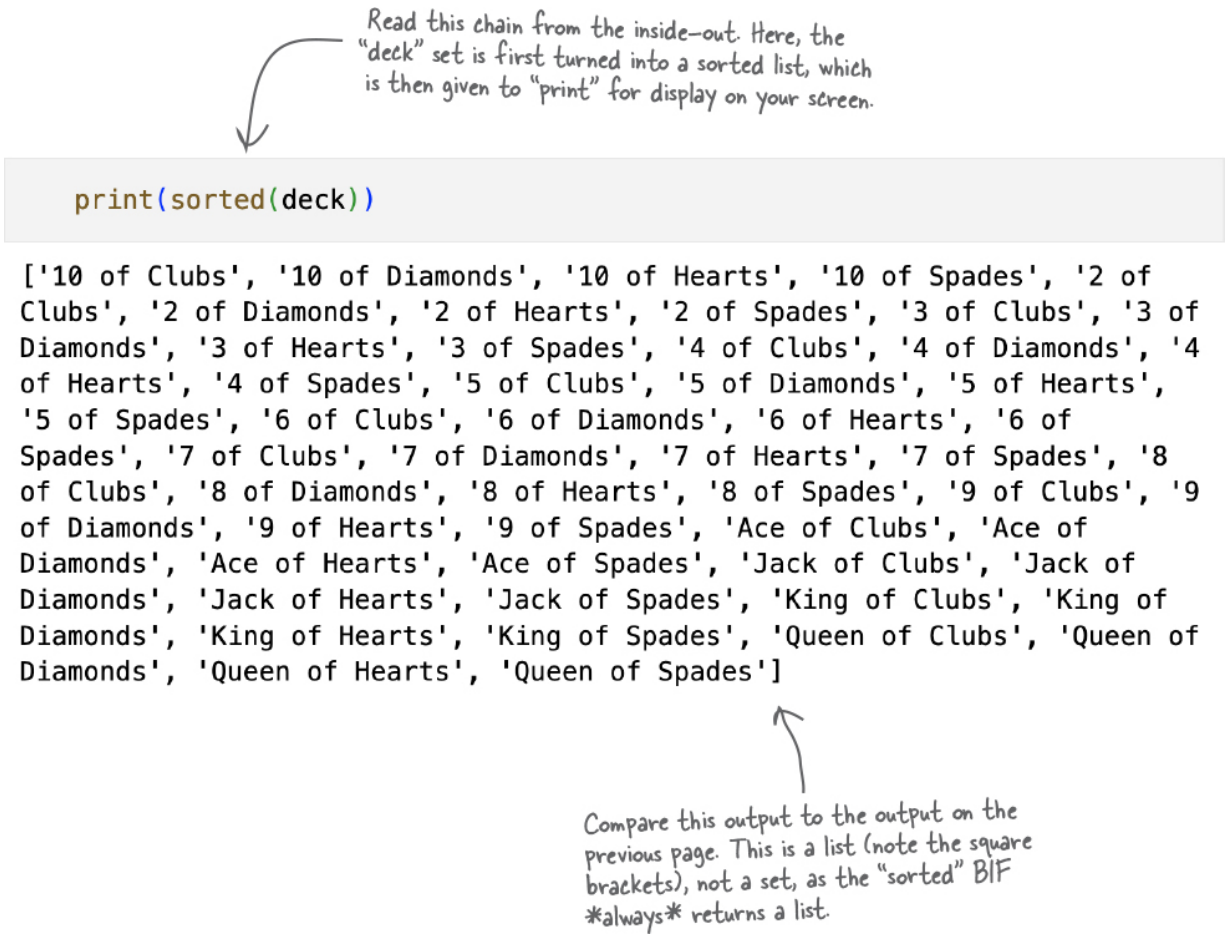


Figure I-33.

The **print sorted** chain is a great combination, but get ready to type the next chain (shown on the next page) more times that you'll likely think possible. It really is a common combination, and useful to boot.

The print dir combo mambo

When provided with the name of any Python object, the **dir** BIF returns a list of the object's attributes which, in the case of the `deck` variable, are the attributes associated with a set object.

As you can see (below), there are an awful lot of attributes associated with a Python set. Note how the output from **dir** is chained with a call to the **print** BIF, ensuring the displayed output is drawn *across* your screen as opposed to *down* your screen, which cuts down on the amount of scrolling required of your poor fingers. This may or not be something to dance about but – hey! – every little helps.

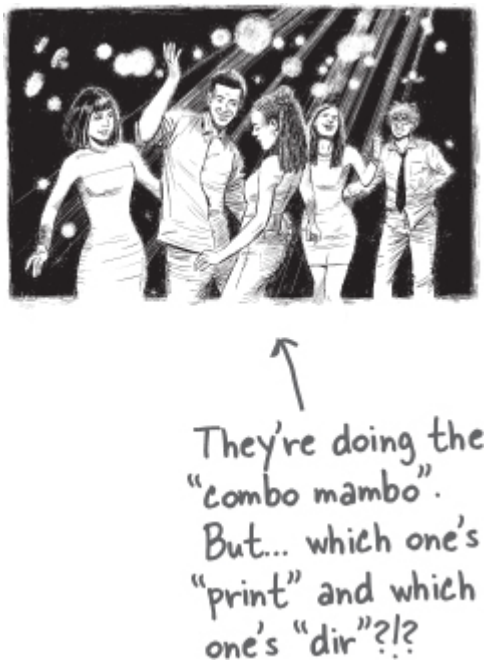


Figure I-34.

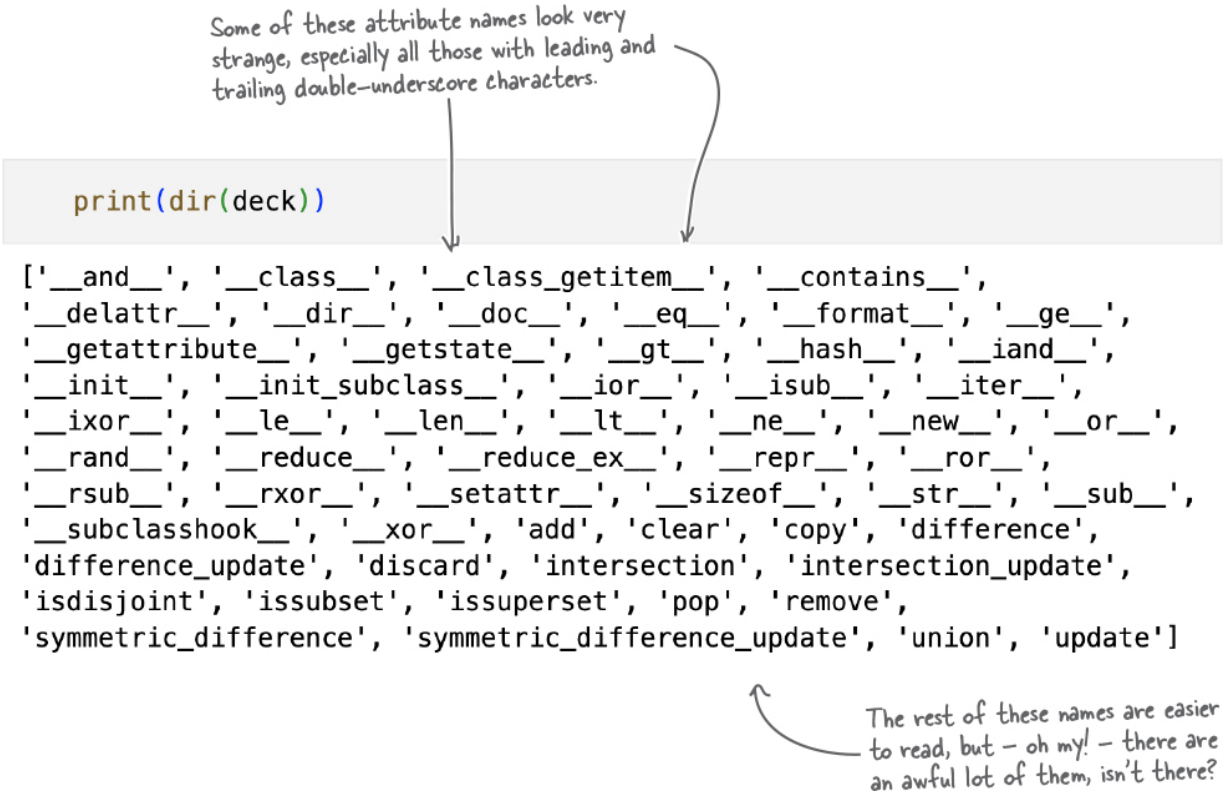


Figure I-35.

Here's a simple rule to follow when looking at the output from **print dir**: *For now, ignore the attributes which begin and end with a double-underscore*. You'll learn why they exist later in this book, but – for now – ignore, ignore, ignore!

Getting help with dir's output

You might not think this to look at it, but you'll likely use the **dir** BIF more than any other BIF when working with Python, especially when experimenting within a Jupyter notebook. This is due to **dir**'s ability to fess-up the list of attributes associated with any object. Typically,

these attributes include a list of *methods* which can be applied to the object.

Although it might be tempting (albeit a little bonkers) to randomly execute any of the methods associated with the `deck` variable to see what they do, a more sensible approach is to read the documentation associated with the method...

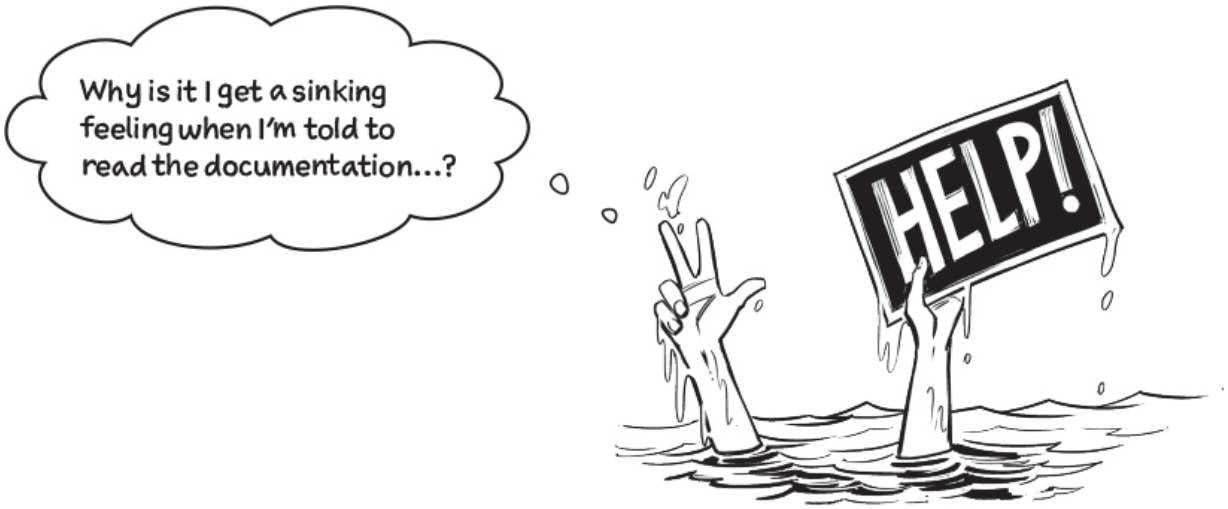


Figure I-36.

Now, don't worry: we aren't about to send you off to wade through thousands of pages of online Python documentation. That's the **help** BIF's job:

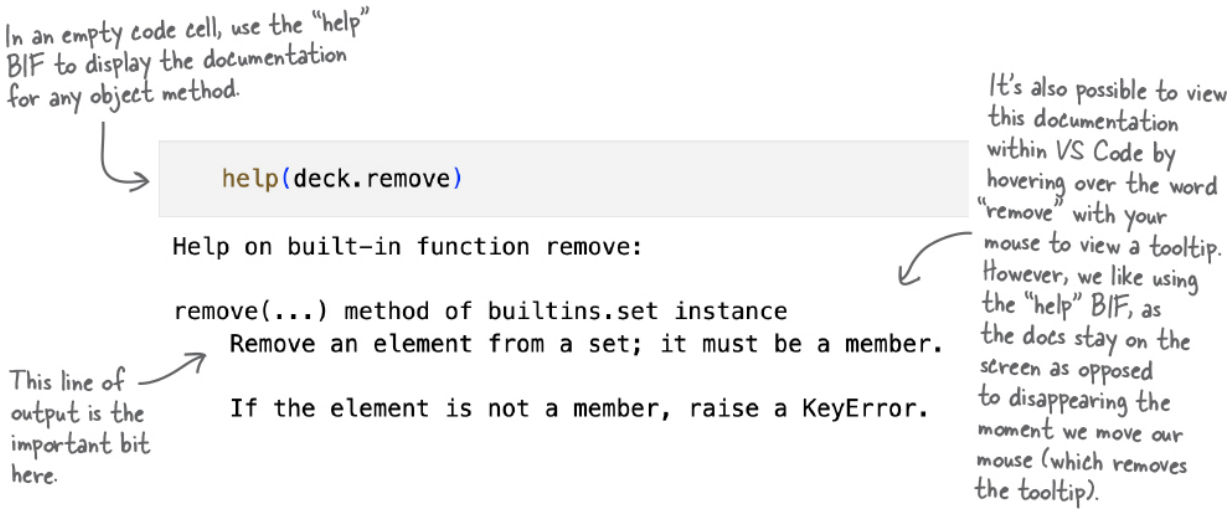


Figure I-37.

This feels like a deck of cards now

Now that your deck of cards is a set, you can better model its behavior.

Sadly, randomly selecting a card from the deck is complicated by the fact the `random. choice` technique from earlier in this chapter doesn't work with sets. This is a pity, as it would've been nice to use `random.choice(deck)` to pick a card from your deck but – alas – this won't work.

NOTE

For now, don't worry about why this is. The reason is kind of technical, which we'll revisit later in this book.

NOTE

This is to make sure our readers keep reading, right?!? [Editor].

Not to worry. A quick hack lets you first *convert* a copy of your set of cards to a list, which can then be used with `random.choice`. It couldn't be more straightforward:

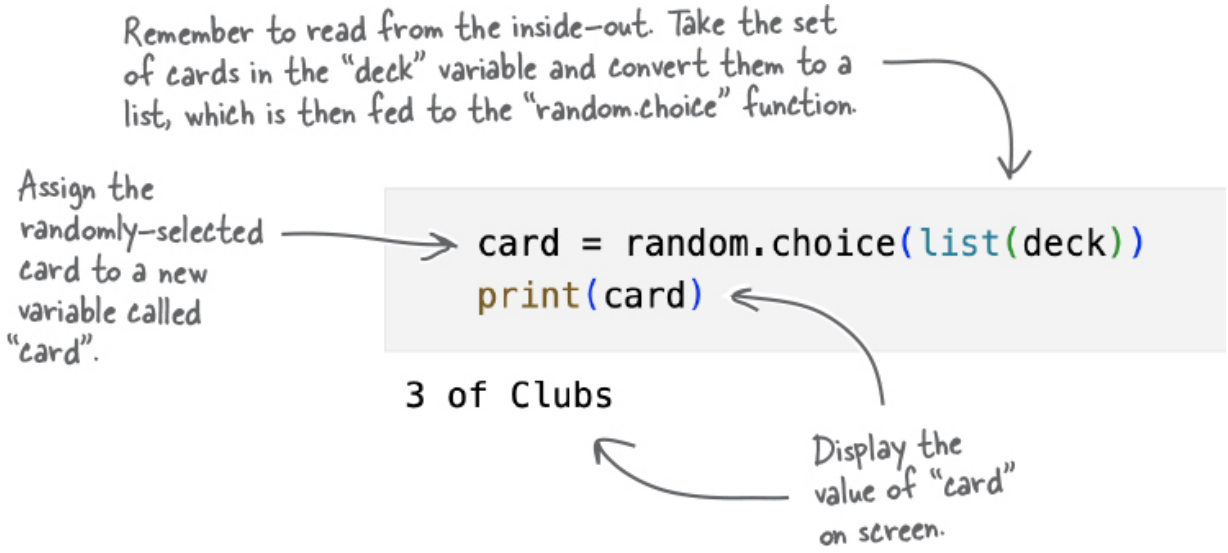


Figure I-38.

Having selected a card (it's the *three of clubs* for us, but is likely a different card if you're following along), we should really remove the card from the deck so that subsequent random choices no longer select it. There was a big clue shown at the bottom of the previous page as to which set method can help you here.

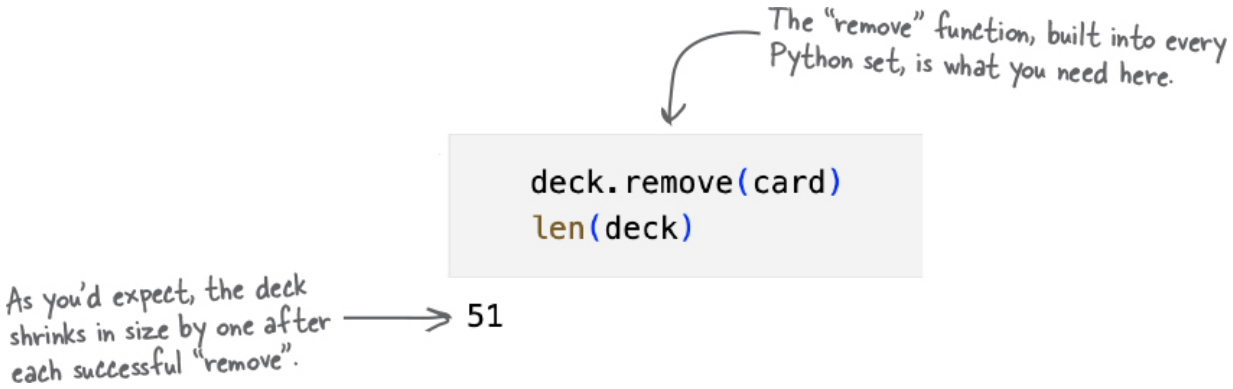


Figure I-39.

What exactly is "card"?

If you're wondering what the `card` variable is, there's another BIF, called **type**, which reports the type of the value currently assigned to `card`:



Figure I-40.



Figure I-41.

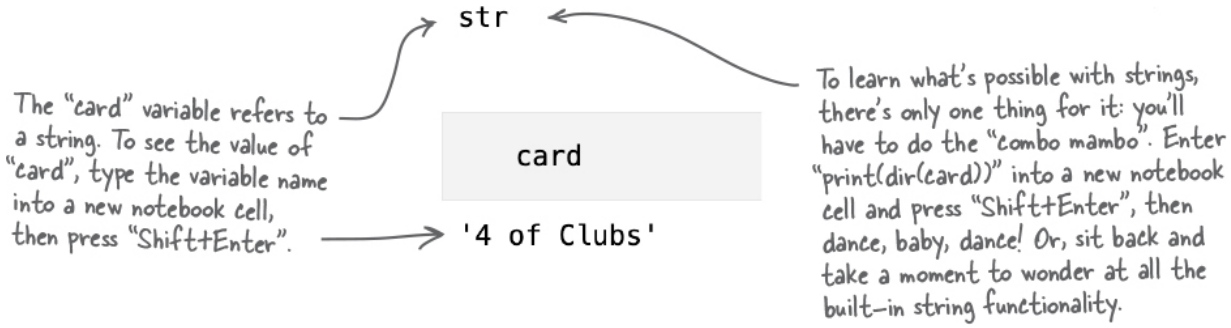

str
The "card" variable refers to a string. To see the value of "card", type the variable name into a new notebook cell, then press "Shift+Enter".
card
'4 of Clubs'
To learn what's possible with strings, there's only one thing for it: you'll have to do the "combo mambo". Enter "print(dir(card))" into a new notebook cell and press "Shift+Enter", then dance, baby, dance! Or, sit back and take a moment to wonder at all the built-in string functionality.

Figure I-42.

SHARPEN YOUR PENCIL

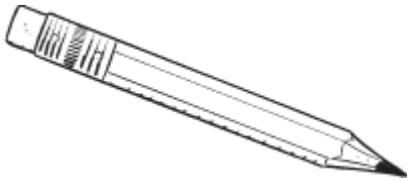


Figure I-43.

Take another look at the existing version of the `draw` function which, you'll recall, doesn't guard against returning the same card when called repeatedly:

```
def draw():
    which_suit = random.choice(suits)
    which_type = random.choice([faces, numbered])
    which_card = random.choice(which_type)
    return which_card, "of", which_suit
```

Grab your pencil and, in the space below, rewrite the `draw` function to randomly select a card from your `deck` set. In addition to returning the selected card, ensure the function's code removes the drawn card from the deck:

_____

_____

SHARPEN YOUR PENCIL SOLUTION

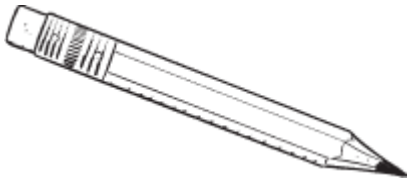


Figure I-44.

You were asked to take another look at the existing version of the `draw` function which, you'll recall, doesn't guard against returning the same card when called repeatedly:

```
def draw():
    which_suit = random.choice(suits)
    which_type = random.choice([faces, numbered])
    which_card = random.choice(which_type)
    return which_card, "of", which_suit
```

You were to grab your pencil then, in the space below, rewrite the `draw` function to randomly select a card from your `deck` set. In addition to returning the selected card, you were to ensure the function's code removes the drawn card from the deck:

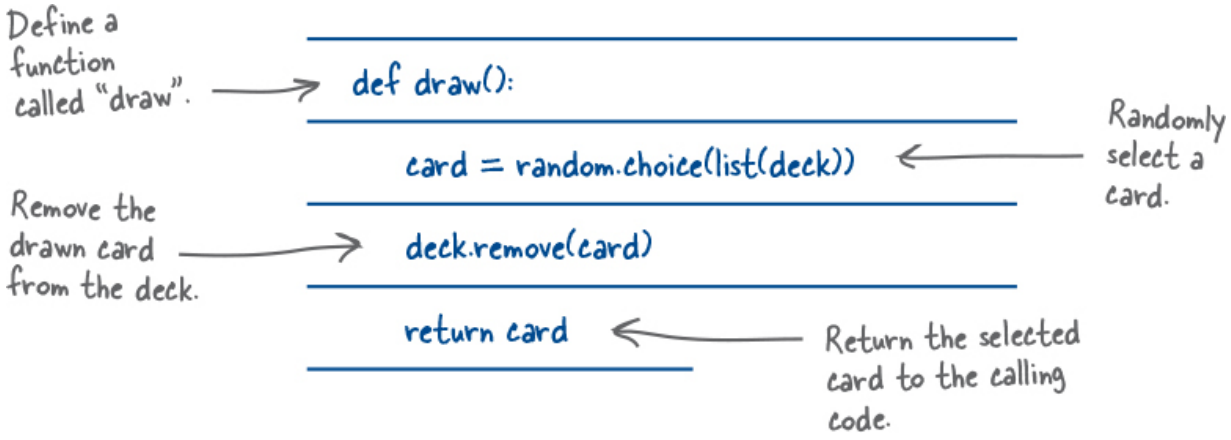

Define a function called "draw".
def draw():
card = random.choice(list(deck))
Randomly select a card.
Remove the drawn card from the deck.
deck.remove(card)
return card
Return the selected card to the calling code.

Figure I-45.

THERE ARE NO DUMB QUESTIONS

Q: I take it that colon at the end of the function's `def` line is important?

A: Yes, very. You'll get to know the colon in detail later in this book. For now, think of it as a piece of required syntax which informs everyone an indented block of code is about to start (on the next line).

Q: What happens the older definition of `draw`? Can I still get at it should I need to?

A: The new definition of `draw` replaces the previous definition. The previous definition of `draw` has disappeared into the digital ether (in a puff of smoke), never to be heard from again.

Q: This is bugging me, but why the funny spelling of Jupyter? Isn't the planet spelled with an "i"?

A: Yes, the planet is spelled with an "i", but the tool is not named after the planet. Jupyter is named after the three programming languages it initially supported, namely Julia, Python, and R. That's why the "py" letters are included in the name: That's also a reference to Python's preferred filename extension for code files, which is `.py`.

Q: The notebook extension is quite the mouthful: `.ipynb`. Does this stand for anything in particular?

A: Yes. It's shorthand for the *ipython notebook format*. Are you glad you asked?

TEST DRIVE



Figure I-46.

Let's check your new `draw` function is working as expected. Once it's defined, you can call it:

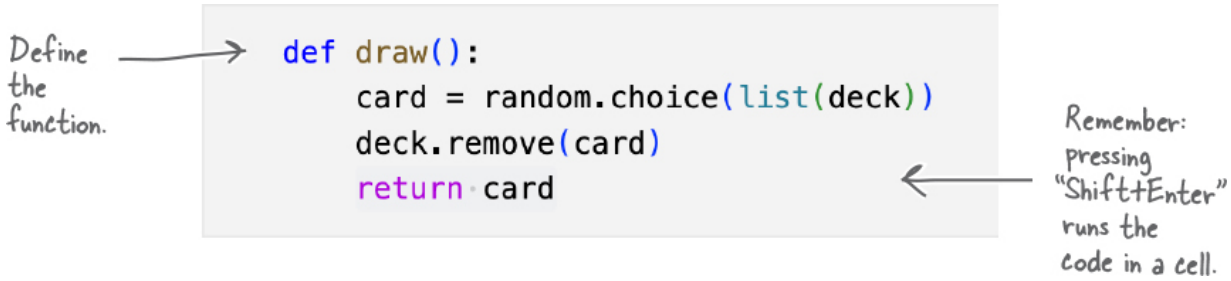


Figure I-47.

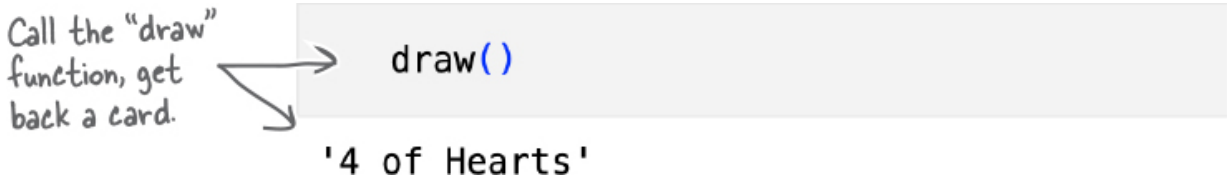


Figure I-48.

Let's recreate the deck of cards so that all 52 cards are once again available:

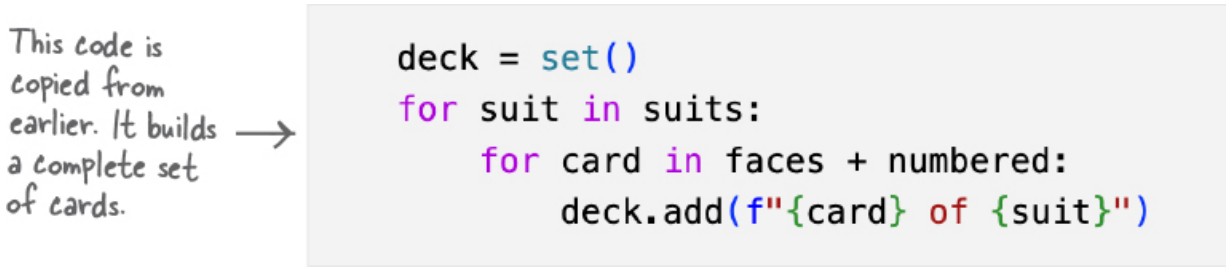


Figure I-49.

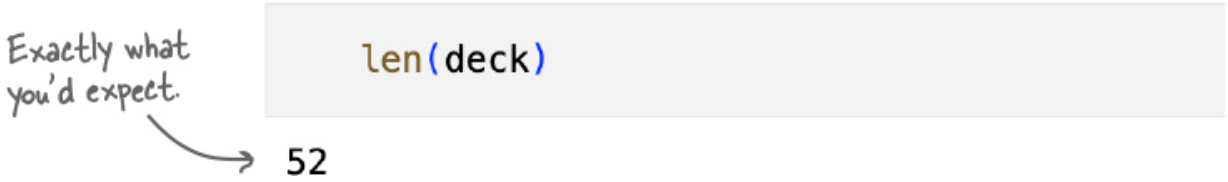


Figure I-50.

Another loop draws five cards from the deck, displaying each drawn card on screen:

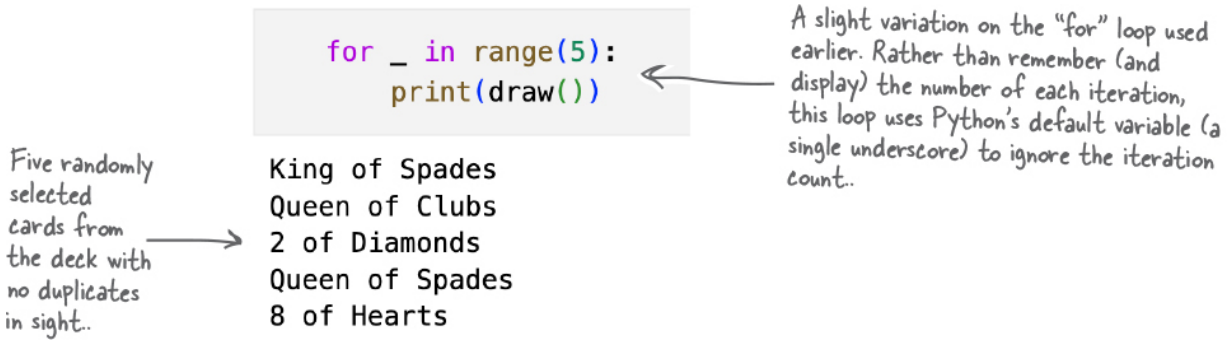


Figure I-51.

Having drawn five cards from the deck, you are left with 47 cards, which the **len** BIF confirms:



Figure I-52.

Python's wonderful **in** operator checks for membership (aka, *performs a search*):



Figure I-53.

You'll typically see the **in** operator used within the conditional-part of Python's **if** statement. Here **in** is combined with **not** to display an appropriate message based on the existence of a specific card in the deck:

You've yet to see a Python "if" statement, but look how easy it is to read this one and work out what is does.

```
if '2 of Diamonds' not in deck:
    print("Draw another card. That one's taken.")
else:
    print("Draw! Draw! Draw!")
```

```
Draw another card. That one's taken.
```

The "else" part of the "if" executes when the conditional statement returns "False", whereas the "That one's taken" message is shown when the conditional returns "True".

Figure I-54.



Figure I-55.

The parentheses are optional.

You *can* wrap your conditionals in parentheses, but doing so makes the code harder to read. Most Python programmers only use parentheses when it helps to clarify what their code is doing.

But, wait! There's more...

You may already be sold on Python now you've seen how easy it is to read as well as run your Python code. But, you're not done yet.



Figure I-56.

For the remainder of this chapter, you continuing your whistle-stop tour of some of Python's other standout features.

As this is a *Head First* book, it's not enough we tell you what these are, we want you to *experience* them. So, in VS Code, close your `Cards` notebook, then create a new notebook called `WhyPython.ipynb`. You'll work in this new notebook for the rest of this chapter.

NOTE

To create a new notebook in VS Code, select File, then New File... from the menu. Choose the third option to create a new, untitled notebook. Perform a File, Save to change the untitled name to "WhyPython.ipynb".



Figure I-57.

Yes. Every… single…word.

Oh, we're only joking. 😉

The goal for this chapter is to expose you to enough Python to get a feel for why Python is so popular. As such, the coverage in this chapter is very *high-level.*

But, don't worry: You'll be returning to all of this material *in detail* later in this book. For now, concentrate on understanding the gist of what you're seeing.

With your new notebook ready in VS Code, get ready to dig in.

Python ships with a rich standard library

The *Python Standard Library* (PSL) is the name used to refer to a large collection of Python functions, types, modules, classes, and packages bundled with Python. These are guaranteed to be there once Python is installed.

When you hear programmers refer to Python as coming with "batteries included", they are referring in part to the PSL. There's a lot to it: https://docs.python.org/3/library/index.html.



Figure I-58.

In this book, “PSL” is short-hand for the “Python Standard Library”.

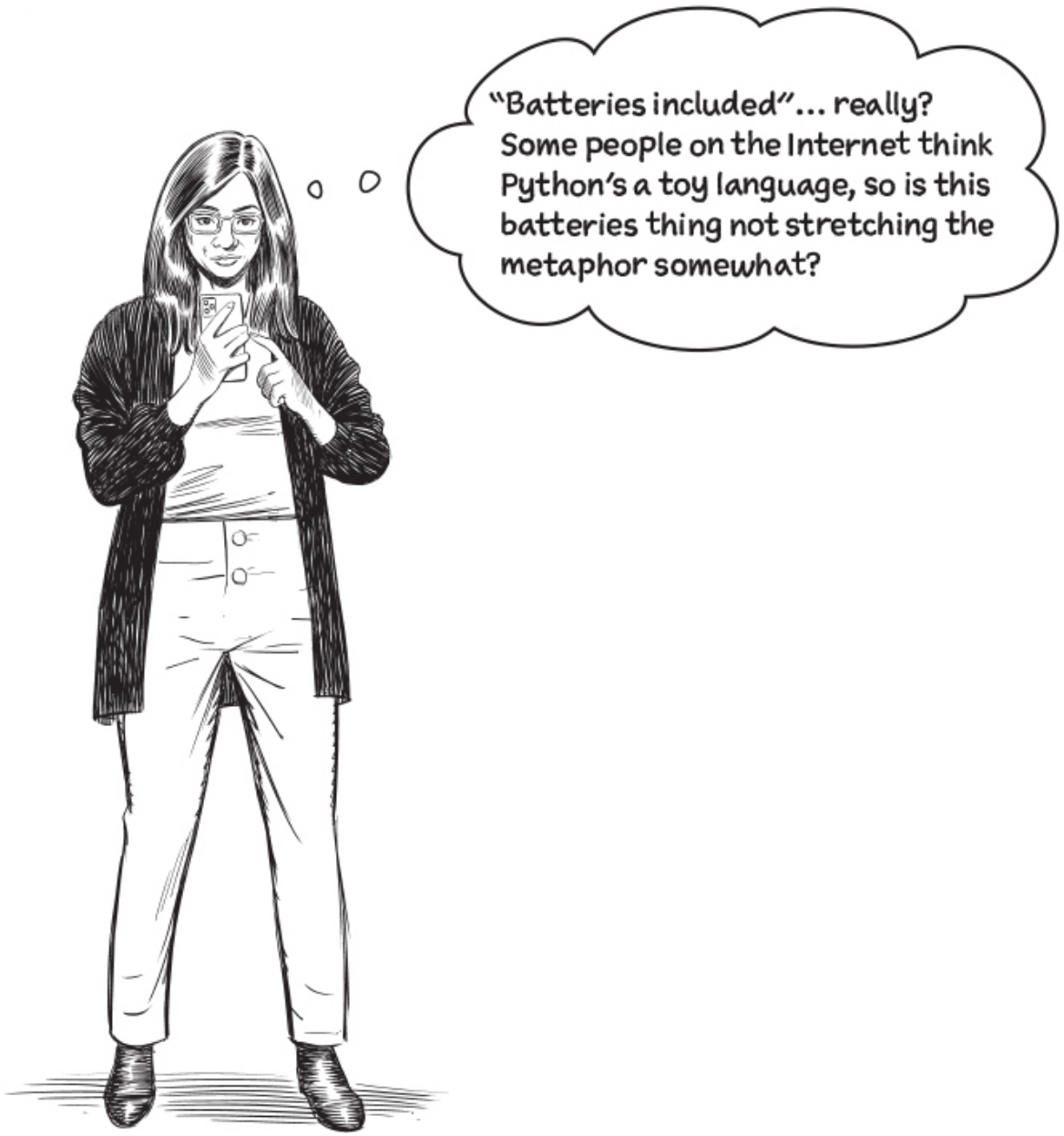


Figure I-59.

It's quite an apt description.

The Python install includes the PSL, which is complete to the point where, more times than not, you can rely on the features it provides to get a lot of work done. The thinking is that Python alone is all you'll

need to get going, which means the standard install of Python works "right out to the box" without the need for anything extra. Hence, *batteries included.*



Figure I-60.

BTW: Python is not a "toy language"

This is a common criticism levelled at Python, in that it is somehow not a "real" programming language, or some sort of "toy". If either of these observations were even remotely true, you wouldn't expect anyone *anywhere* to be using Python for anything useful, let alone relying on Python to power their business.

Python may indeed look *different*, but this does not mean it can't get the job done. Python is fun to use, but this doesn't mean it's a toy. Far from it.

WHO DOES WHAT?

We know you've yet to look at the PSL in any great detail but, to give you a taste of what's included, we've devised a little test. Without taking a peek at the documentation referred to on the last page, consider the names of some of the modules from the PSL shown on the left of this page. Grab your pencil and draw an arrow connecting the module name to what you think is the correct description on the right. To get you started, the first one has been done for you. Let's see how you do with the rest. Our answers are on the next page.

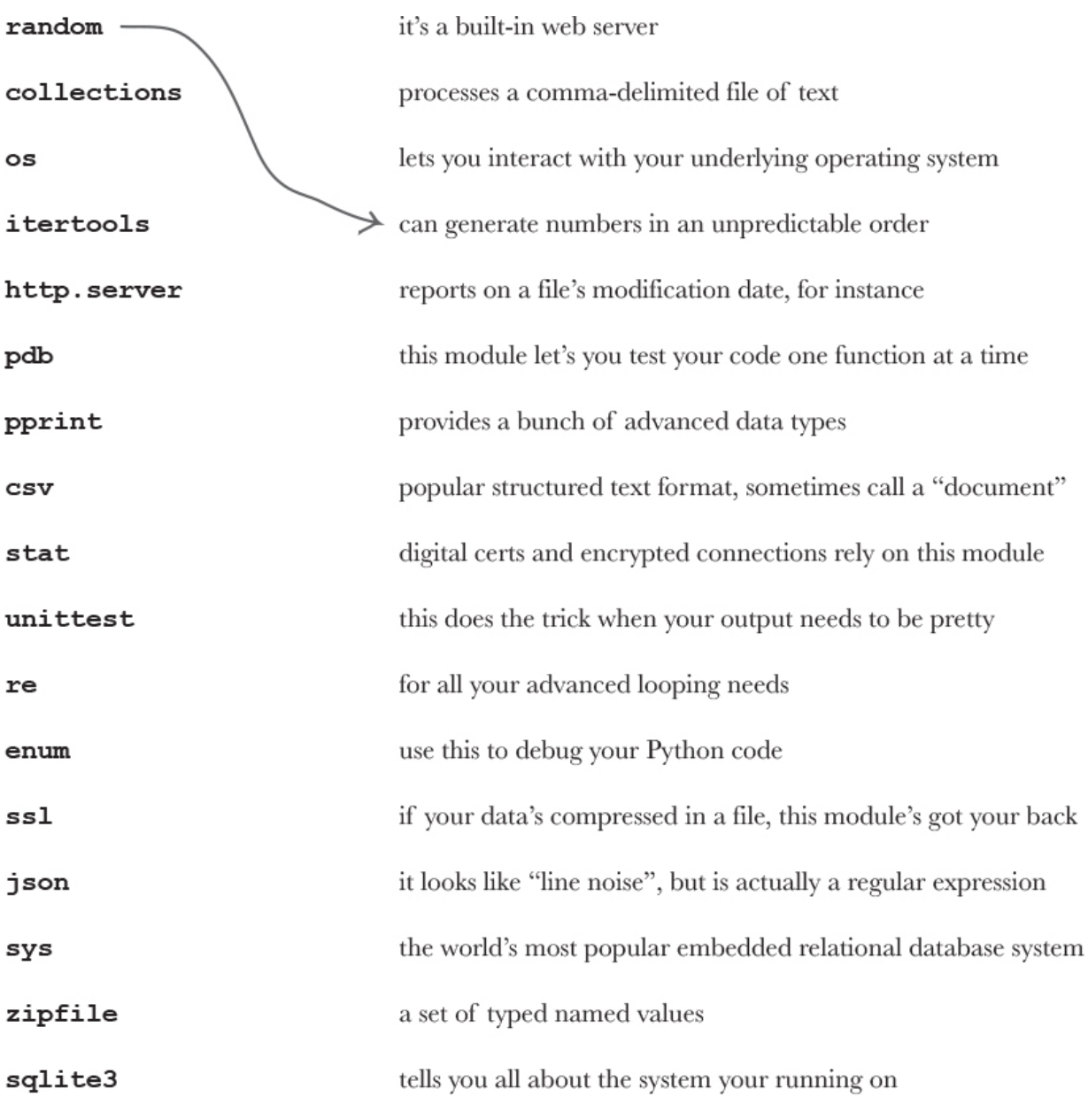

random
collections
os
itertools
http.server
pdb
pprint
csv
stat
unittest
re
enum
ssl
json
sys
zipfile
sqlite3
it's a built-in web server
processes a comma-delimited file of text
lets you interact with your underlying operating system
can generate numbers in an unpredictable order
reports on a file's modification date, for instance
this module let's you test your code one function at a time
provides a bunch of advanced data types
popular structured text format, sometimes call a "document"
digital certs and encrypted connections rely on this module
this does the trick when your output needs to be pretty
for all your advanced looping needs
use this to debug your Python code
if your data's compressed in a file, this module's got your back
it looks like "line noise", but is actually a regular expression
the world's most popular embedded relational database system
a set of typed named values
tells you all about the system your running on

Figure I-61.

WHO DOES WHAT? SOLUTION

We know you've yet to look at the PSL in any great detail but, to give you a taste of what's included, we've devised a little test. Without taking a peek at the documentation referred to earlier, your were to consider the names of some of the modules from the PSL shown on the left of this page. Grabbing your pencil, you were to draw an arrow connecting the module name to what you think is the correct description on the right. The first one was done for you. Now you can see our arrows, how did you do?

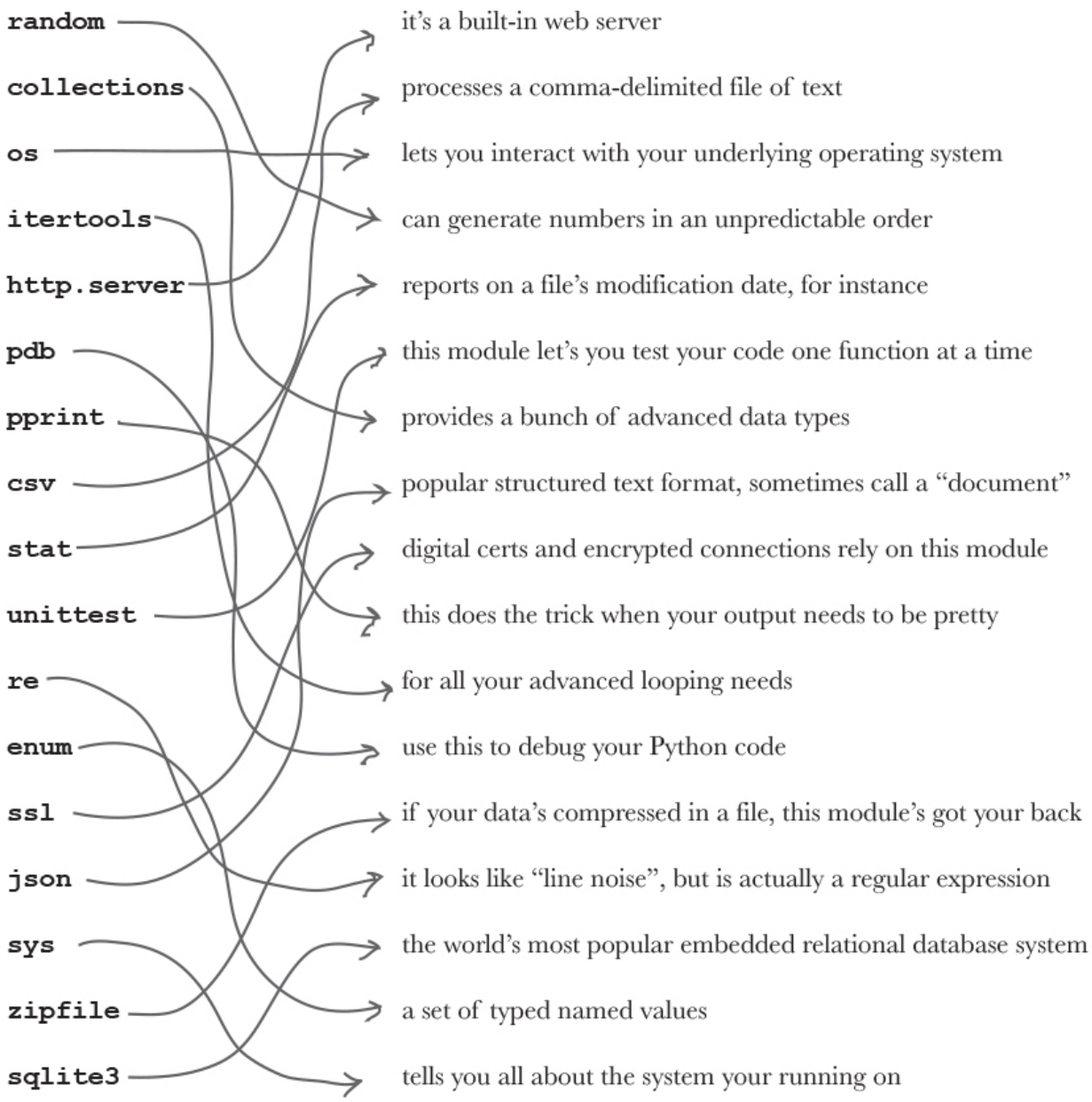

random
collections
os
itertools
http.server
pdb
pprint
csv
stat
unittest
re
enum
ssl
json
sys
zipfile
sqlite3
it's a built-in web server
processes a comma-delimited file of text
lets you interact with your underlying operating system
can generate numbers in an unpredictable order
reports on a file's modification date, for instance
this module let's you test your code one function at a time
provides a bunch of advanced data types
popular structured text format, sometimes call a "document"
digital certs and encrypted connections rely on this module
this does the trick when your output needs to be pretty
for all your advanced looping needs
use this to debug your Python code
if your data's compressed in a file, this module's got your back
it looks like "line noise", but is actually a regular expression
the world's most popular embedded relational database system
a set of typed named values
tells you all about the system your running on

Figure I-62.



Figure I-63.

There sure is a lot going on there.

Our goal is to give you a flavor of what's in the PSL, not for you to explore it in any great detail.

You are not expected to know all of this, nor remember what's on the last page, although there are three points you should consider.

NOTE

To be clear: we're not talking about coffee…

---

1 You've only scratched the surface

The PSL has a lot in it, and what's on the previous two pages provides the briefest of glimpses. As you work through this book, we'll call out uses of the PSL so you don't miss any (and you'll also find resources in the appendices for further exploring the PSL on your own).

2 The PSL represents a large body of tested code which you don't have to write, just use As the PSL has existed for decades now, the modules it contains have been tested to destruction by legions of Python programmers *all over the globe*. Consequently, you can use PSL modules with confidence.

3 The PSL is guaranteed to be there, so you can rely on its modules being available

Other than for some very specific edge cases (such as a tiny embedded micro-controller providing a minimal Python environment), you can be sure your code which uses any PSL module will be portable to other systems which also support the PSL.

Let's use your latest notebook to take a quick look at two modules from the PSL.

TEST DRIVE



Figure I-64.

Let's see two modules from your recent *Who Does What?* exercise in action. In your `WhyPython` notebook, type the code below into code cells, remembering to press **Shift+Enter** to execute the cells one-at-a-time. First up is a bit of randomness. Let's face it, everyone loves random numbers, and the PSL's **random** module makes generating them super easy:



Figure I-65.

The PSL's **collections** module is also very popular, and contains a bunch of containerised data types. A particular favorite is the

Counter class which automates frequency counting:

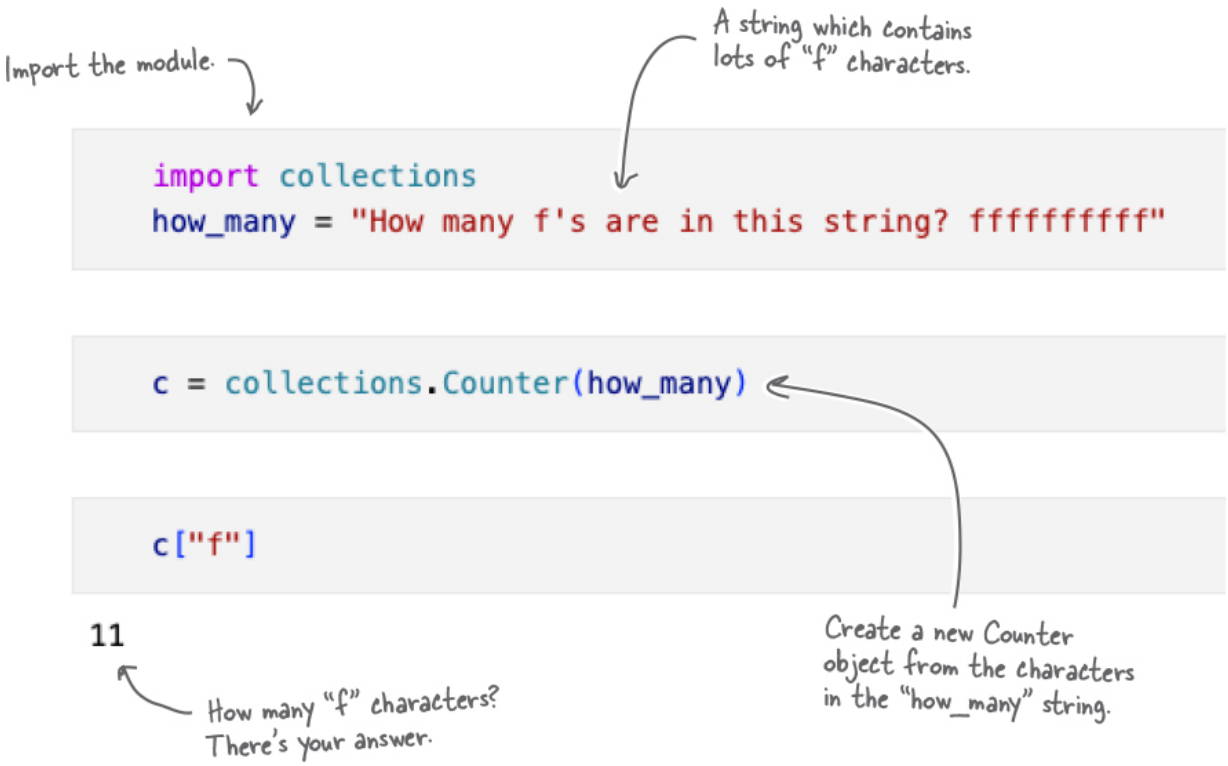


Figure I-66.

With Python you'll only write the code you need

The PSL is an prime example of Python working hard to ensure you only write new code when you absolutely have to. If a module in the PSL solves your problem, use it: *Resist the urge to code everything from scratch*.

And when it comes to reusing code, there's more than the PSL to mine.

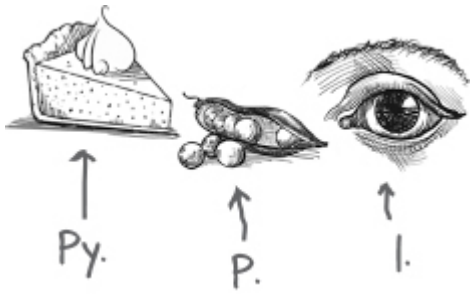


Figure I-67.

Python's package ecosystem is to die for

Not being content with what's already included in the PSL, the Python community supports an online centralised repository of third-party modules, classes, and packages. It's called the *The Python Package Index* and lives here: https://pypi.org/.

Known as *PyPI* (and pronounced "pie-pea-eye"), the index is a huge collection of software. Once you find what you're looking for, installing is a breeze, and you'll get lots of practice installing from PyPI as this book progresses.

For now, take ten minutes to visit the PyPI site (shown below) and take a look around.

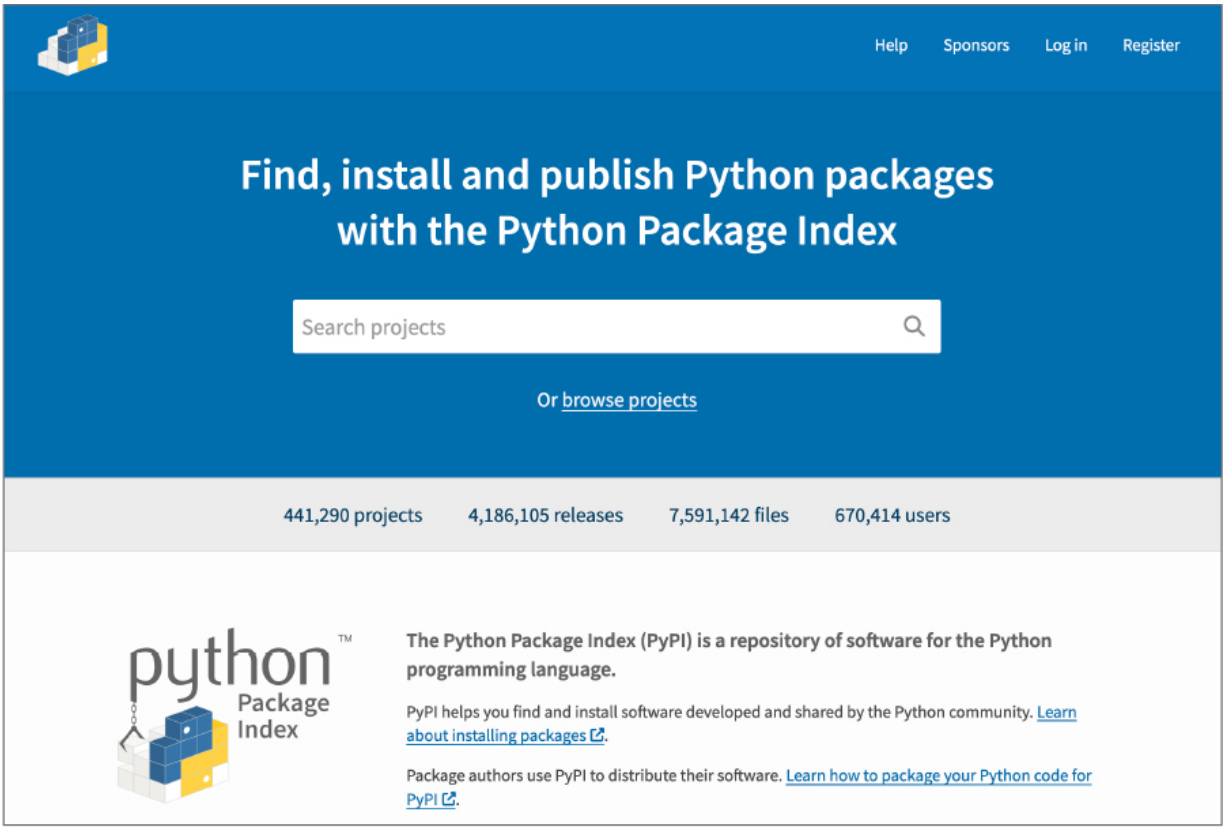

Help
Sponsors
Log in
Register
Find, install and publish Python packages with the Python Package Index
Search projects
Or browse projects
441,290 projects
4,186,105 releases
7,591,142 files
670,414 users
python
Package Index
The Python Package Index (PyPI) is a repository of software for the Python programming language.
PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.
Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.

Figure I-68.

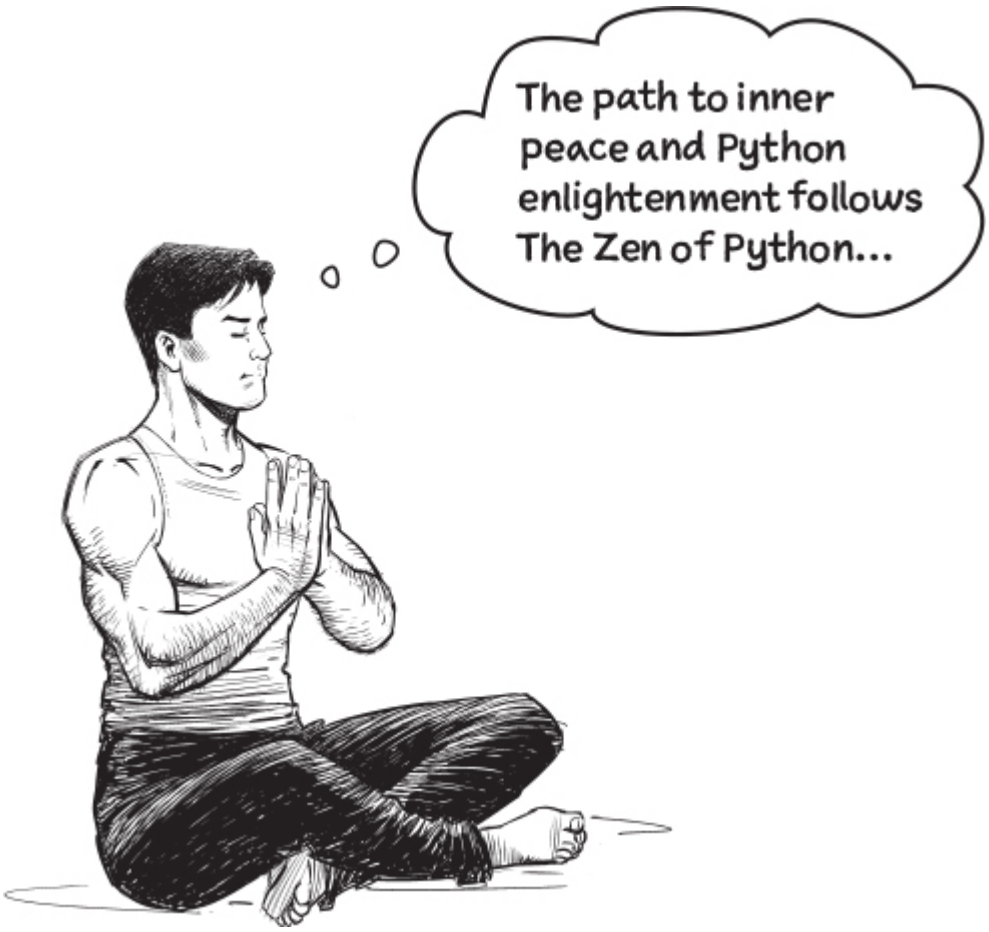


Figure I-69.

OK. If it works for you, sure, follow your Zen!

Seriously, though, when a programming language is named in honor of a bunch of comedians, it should come as no surprise that things get a little silly sometimes. This is not a bad thing.

The Python documentation is literally littered (sorry) with references to *Monty Python*. Where other documentation favors *foo* and *bar*, the Python docs favor *parrots, spam* and *eggs*. Or is it *eggs* and *spam*?

Anyway, as the documentation states: you don't have to like *Monty Python* to use Python, but it helps.



Figure I-70.



Figure I-71.

Python comes with two *Easter eggs* which demonstrate how Python programmers sometimes don't take themselves too seriously, and also don't mind when other folk have a bit of fun at their expense. To see what we mean, return to your WhyPython notebook one last time, and, in two new code cells, run each of the following lines of code. Enjoy!

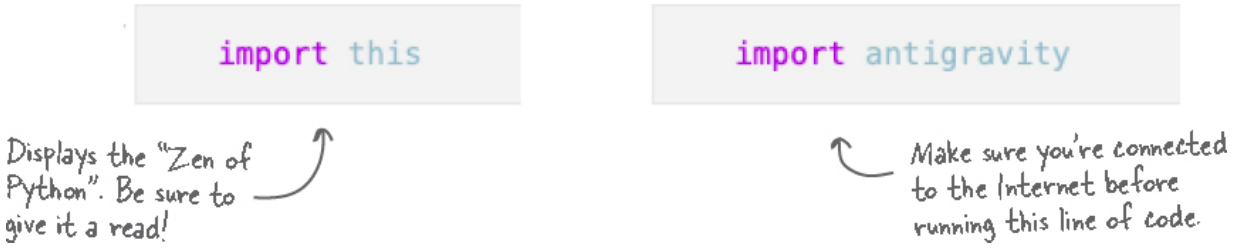

import this
import antigravity
Displays the "Zen of Python". Be sure to give it a read!
Make sure you're connected to the Internet before running this line of code.

Figure I-72.


That high-level overview has really helped. All I need now is a "biggish project" to sink my teeth into...

Figure I-73.

And we can help you with that.

In the next chapter we introduce – and start *immediately* working on – a real-world problem which you'll solve with Python code. Working *together*, we'll build a solution while learning more Python, revisiting the material from this chapter in more detail when needed, and as this book progresses.

Before getting to all that, though, take some time to review the chapter summary on the next page before testing your retention skills with this chapter's crossword puzzle.

See you in the next chapter, Chapter 1, which is actually your *second* chapter as we starting counting from zero (just like Python).

CHAPTER REVIEW BULLET POINTS

- Python is, out of the gate, designed to support the creation of code which is easy to **read**.
- Python code is also easy to **run**. Although a number of ways exist to allow you to do this, in this book **VS Code** together with **Jupyter Notebook** are your go-to tools when experimenting and running your Python code.
- To get going and be productive with Jupyter Notebook, you need to learn a single keyboard combination: **Shift+Enter**.
- In order to ensure you only every write new code when absolutely necessary, Python comes chock-full of **built-in** technology.
- The built-in functions (**BIFs**) are always available, and provide a lots of **generic** functionality.
- The **len** BIF reports the size of an object.
- The `def` keyword is used to define a **function**.
- The **range** BIF produces a fixed-size list of numbers (and is really useful with loops when you need to iterate a specific number of times).
- Talking of loops, Python provides the **for** loop which iterates a specific number of times.
- The **set** BIF creates a set. Sets are one of Python's Big 4 built-in data structures.
- The **print** BIF displays an object's value on screen. When a collection of objects are printed, the **print** BIF displays horizontally

across the screen (which often comes in handy).

- The **sorted** BIF returns an ordered copy of some data.
- The **dir** BIF returns a list of any object's attributes.
- A common idiom is to combine (or **chain**) the **dir** BIF with the **print** BIF creating (what we like to refer to as) the **print dir combo mambo**.
- Some of the attributes shown by the combo mambo refer to **methods** which can be applied to the object, for example `deck.remove`.
- Attributes with leading and trailing double-underscores are *special*, so special in fact that you can ignore them for now.
- A **list** is made up from a collection of object's surrounded by square brackets, and is the second of Python's built-in Big 4 data structures.
- A **tuple** is made up from a collection of object's surrounded by parentheses, and is the third of the Big 4.
- The final built-in data structure is the **dictionary**, which wasn't used in this chapter (only mentioned). This doesn't mean dictionaries aren't cool. They are.
- The **type** BIF reports any object's type.
- The **in** keyword was shown in two places in this chapter. Once within a **for** loop where it identified the collection to be iterated over, and again on it's own when it was used to determine if one object is contained within another (aka *search*).

- The **in** keyword is often used within the conditional part of Python's **if** statement.
- When you need a variable but either can't think of a decent name for your variable or don't need to remember a value by name, use Python's **default variable**: a single underscore character (i.e., `_`). You'll often see the default variable used with loop code.
- Python's **if** statement can also have an optional **else** part.
- Python has two built-in **boolean** values: `True` and `False`.
- The **PSL** has nothing to do with coffee, but everything to do with the **Python Standard Library**. The PSL is a large collection of built-in modules (which come with Python) and can be used all over the place to do many useful things.
- If the PSL isn't enough for you, check out **PyPI**, the **Python Package Index**, an online repository of shareable Python modules. It's often the case some of the code you need has already been written and uploaded to PyPI as a shareable module. Feel free to "leverage" as needed.
- There are other useful **keyboard shortcuts** which you can use when working within Jupyter. Our nine essential shortcuts are coming up after this chapter's crossword solution (in three pages time).

The Card Deck Crossword



Figure I-74.

Congratulations on making it to the end of your opening chapter, numbered zero in honor of the fact that Python, like a lot of other programming languages, starts counting from zero. Before you dive into your next chapter, take a few minutes to try this crossword puzzle. All of the answers to the clues are found in this chapter's pages, and the solution is on the next page. Enjoy!

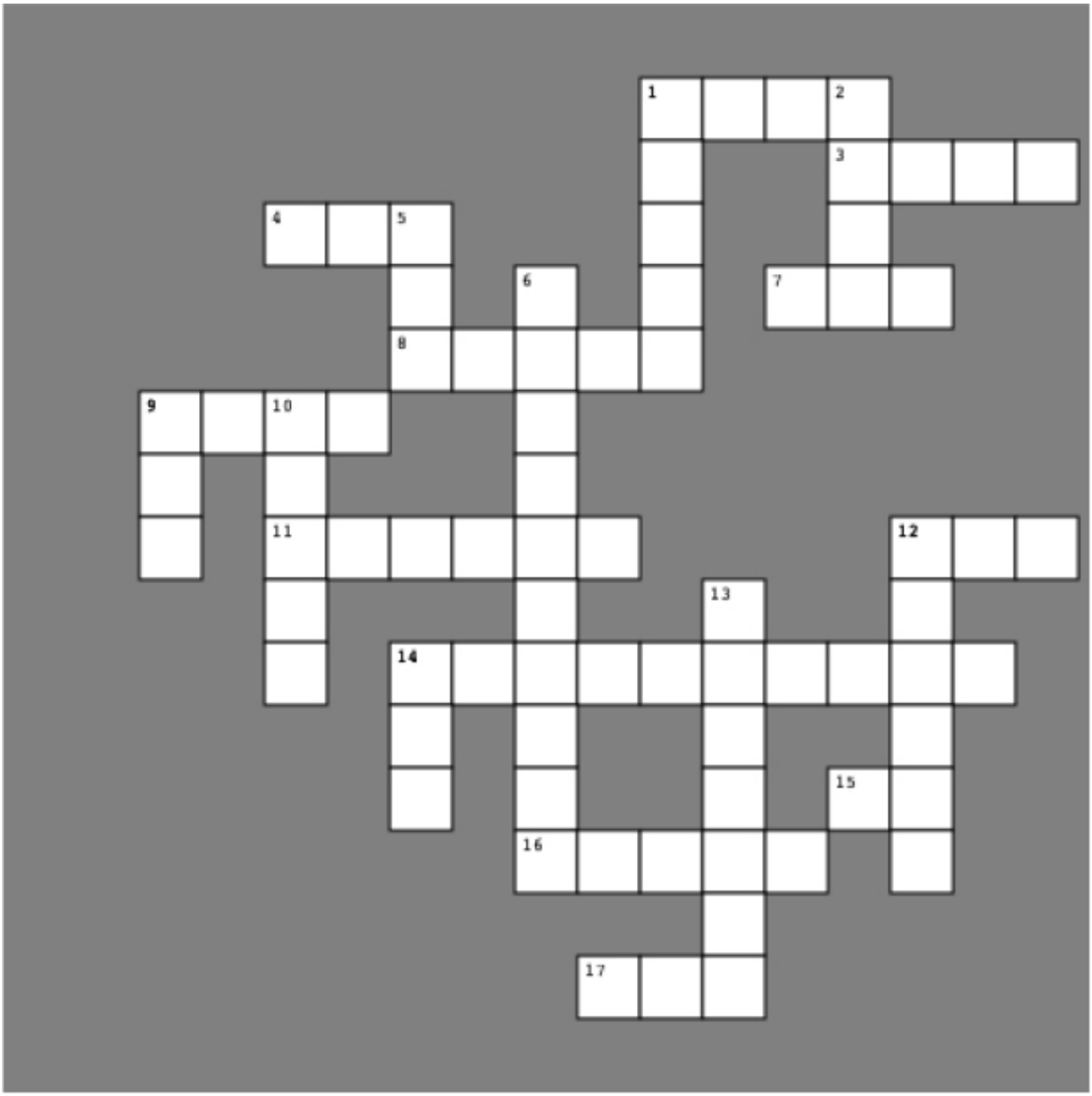


Figure I-75.

Across

2. A built-in function which tells you what something is.

3. Objects surrounded by `[` and `]`.

4. Shorthand for built-in function.

7. Reports on an object's size.

8. Generates collection of numbers.

9. The Python Package Index.

11. Includes a module in your code.

12. Objects surrounded by `{` and `}`.

14. This chapter's missing Big 4.

15. This operator can find things.

16. Use together with Shift to run.

17. Enlightenment, Python-style.

Down

1. Objects surrounded by `(` and `)`, and it is one of the Big 4, too.

2. The optional part of an `if` statement.

5. Loops a specific number of times.

6. The name given to Python's default variable.

9. It's not a *Pumpkin Skinny Latte*, but shares the same acronym

10. Displays to screen.

12. It's what the `card` variable is.

13. It's either `True` or `False`.

14. Makes up the *combo mambo*.

The Card Deck Crossword Solution



Figure I-76.

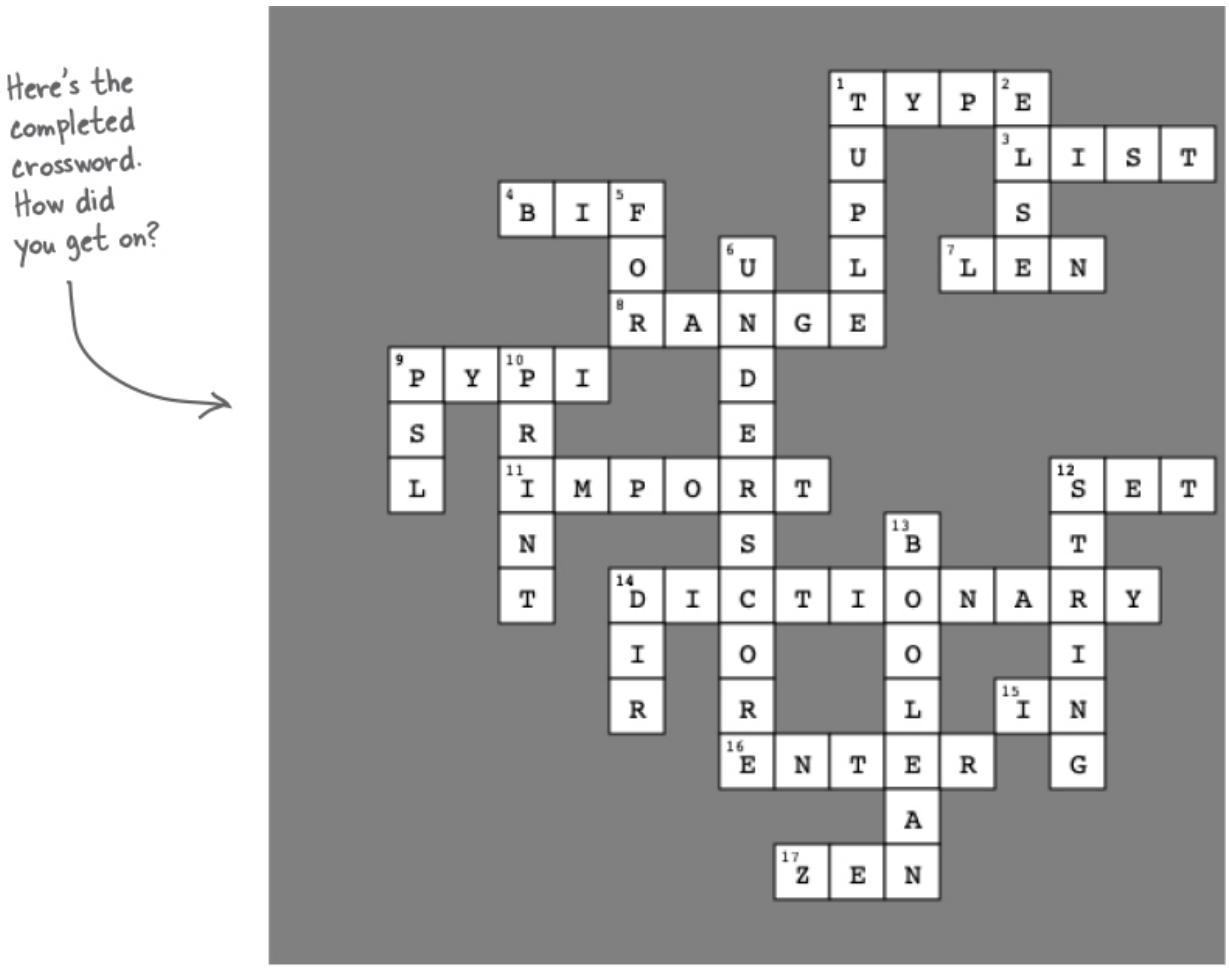


Figure I-77.

Across

2. A built-in function which tells you what something is.

3. Objects surrounded by `[` and `]`.

4. Shorthand for built-in function.

7. Reports on an object's size.

8. Generates collection of numbers.

9. The Python Package Index.

11. Includes a module in your code.

12. Objects surrounded by `{` and `}`.

14. This chapter's missing Big 4.

15. This operator can find things.

16. Use together with Shift to run.

17. Enlightenment, Python-style.

Down

1. Objects surrounded by `(` and `)`, and it is one of the Big 4, too.

2. The optional part of an `if` statement.

5. Loops a specific number of times.

6. The name given to Python's default variable.

9. It's not a *Pumpkin Skinny Latte*, but shares the same acronym

10. Displays to screen.

12. It's what the `card` variable is.

13. It's either `True` or `False`.

14. Makes up the *combo mambo*.

Just when you thought you were done...

Go grab your scissors, as here's a handy cut-out chart of the Jupyter Notebook keyboard shortcuts we view as *essential*. You'll get to use all of these are you learn more about Jupyter. For now, **Shift+Enter** remains the most important combination:

Notebook key combinations:

Shift+Enter	Execute the current code cell, then move the focus to the next cell (creating a new empty cell when at the bottom of the notebook).
Ctrl+Enter	Execute the current code cell, but don't move the focus.
Alt+Enter	Execute the current code cell, then insert a new empty cell below the executed one. Move the focus to the new cell.

Notebook key sequences:

Esc then A	Insert a new empty cell above the current cell. Move the focus to the new cell.
Esc then B	Insert a new empty cell below the current cell. Move the focus to the new cell.

Other useful notebook key sequences:

Esc then C	Take a copy of the cell which currently has the focus.
Esc then V	Paste a previously copied (or cut) cell below the currently focused cell.
Esc then X	Cut the currently focused cell from the notebook.
Z	Undo the last cut (there's no need to press the ESC key here).

NOTE

Just as well, as we asked you to take your scissors to what's on the flip-side!

Chapter 1. Diving in: *Let's Make a Splash*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

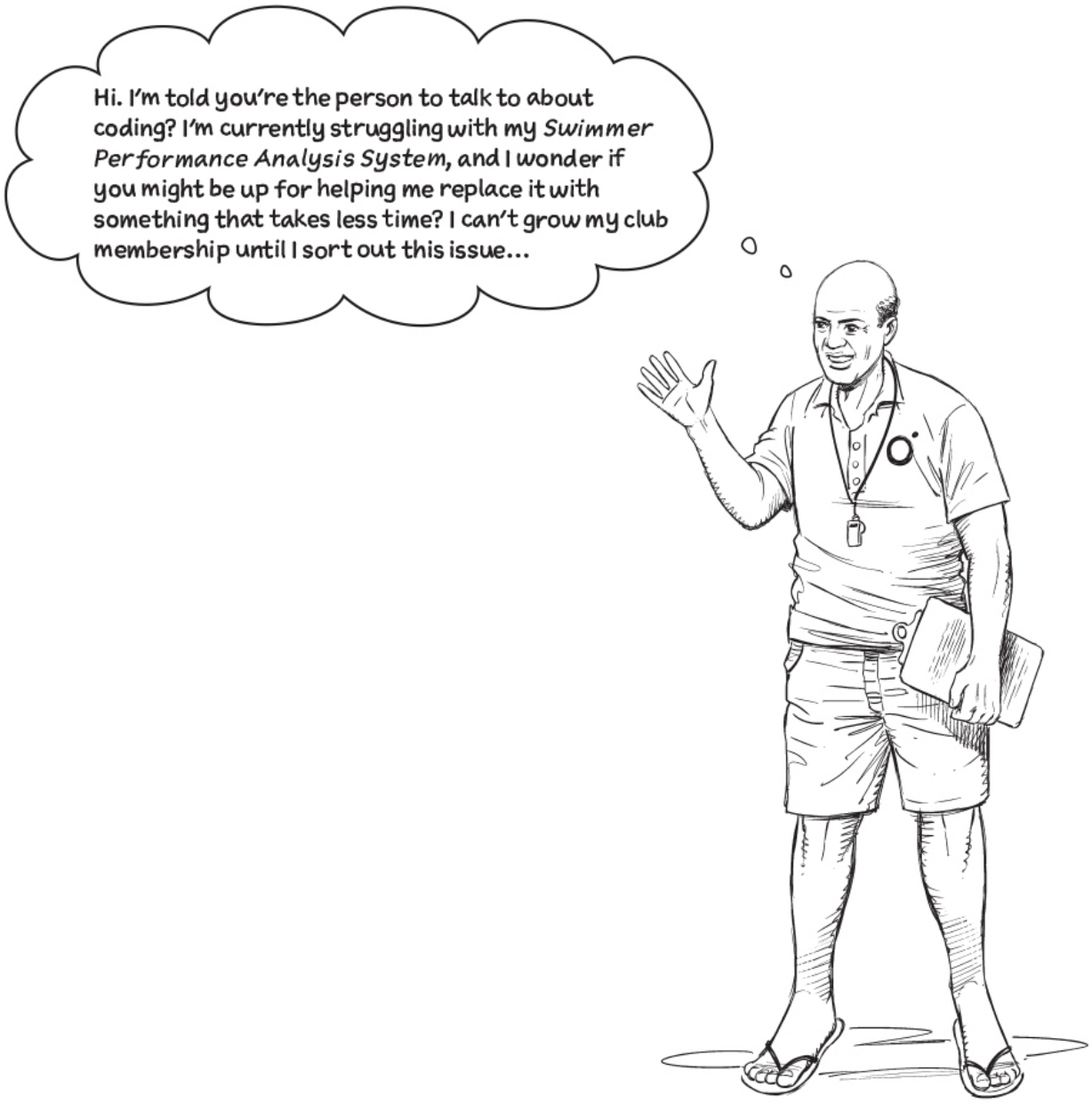
This will be Chapter 1 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



The best way to learn a new language is to write some code. And if you're going to write some code, you'll need a **real** problem. As luck would have it, we have one of those. In this chapter, you'll start your Python application development journey by making a splash with our friendly, neighborhood, **Swim Coach**. You'll begin with Python **strings**, learning how to **manipulate** them to your heart's desire, all the while working your way towards producing a Python-based solution to the Coach's problem. You'll also see more of Python's built-in **list** data structure, learn how **variables** work, as well as discover how to read Python's **error messages** without going off the deep-end, all

while solving a *real* problem with *real* Python code. Let's dive in (head first)...



This sounds interesting.

Your subsequent chat with the Coach starts to tease out some of the details...

When it comes to monitoring the progress of his young swimmers, the Coach does everything *manually*. During a training session, he records each swimmer's training times on his trusty clipboard then, later at home, *manually* types each swimmer's times into his favorite spreadsheet program in order to perform a simple performance analysis.

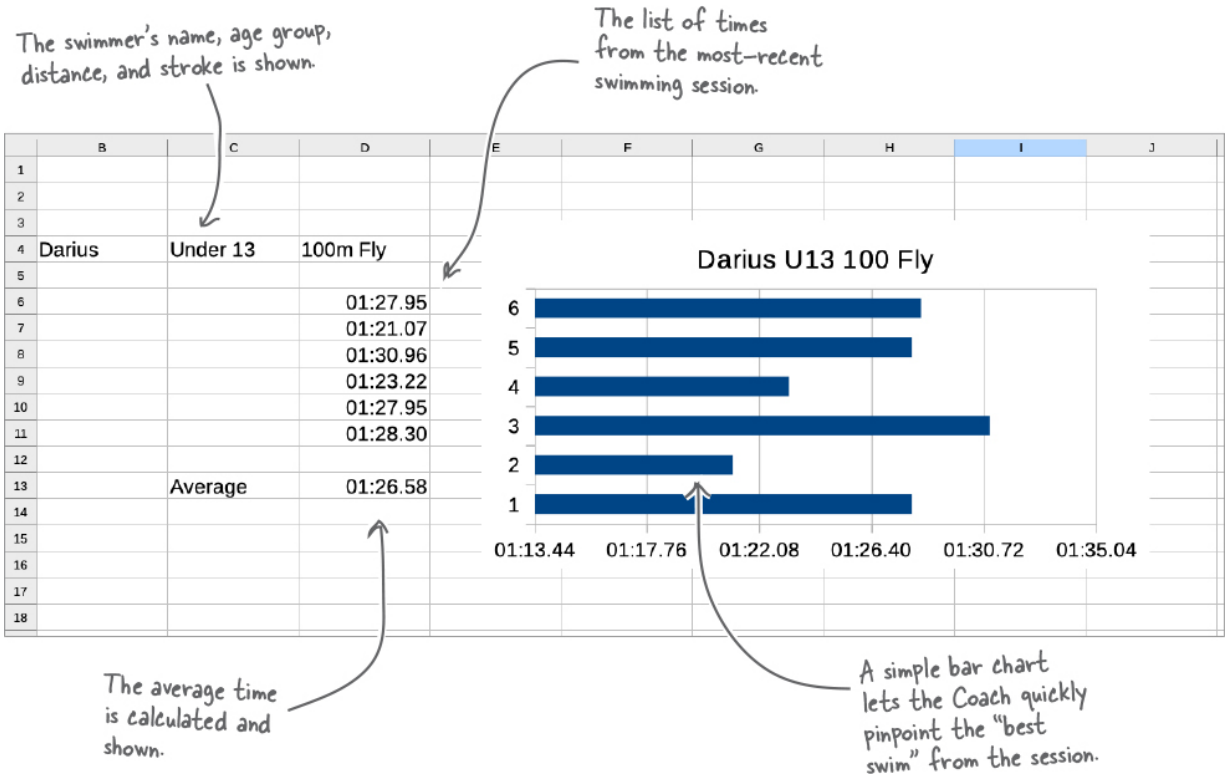
For now, this analysis is straightforward. The times for the session are averaged, and a simple bar chart allows for a quick visual check of the swimmer's performance. The Coach readily admits to being a computer neophyte, stating that he's much better at Coaching young swimmers than "mucking about with spreadsheets".



How do things work right now?

At the moment, the Coach can't expand his swim club membership, as his administrative overhead is too burdensome.

To get a feel for why this is, take a look at one of the Coach's spreadsheets for *Darius*, who is currently 12 years old, so swims in the Under 13 age group:



On the face of things, this doesn't look all that bad. Until, that is, you consider the Coach has over *sixty* such spreadsheets to create after *each* training session.

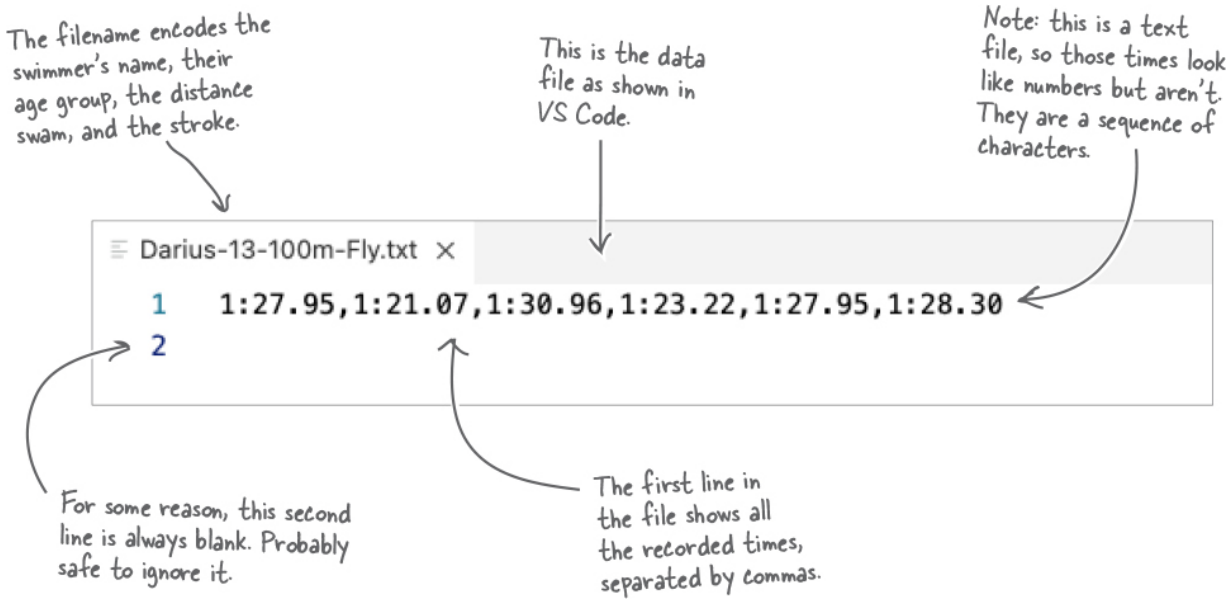
There are only twenty-two swimmers enrolled in the club, but as each can swim different distance/stroke combinations, twenty-two swimmers turns into sixty-plus spreadsheets very easily. That's a lot of manual spreadsheet work.

As you can imagine, this whole process is *painfully* slow...

Fortune decides to smile on you

A quick search of the world's largest online shopping site uncovers a new device described as an *internet-connected digital smart stopwatch*. As product names go, that's a bit of a mouthful, but the smart stopwatch let's the Coach record swim times for an identified swimmer, which are then transferred to the cloud as a CSV file (with a `.txt` extension). As an example of one of these files, here's the contents of the file which contains matching data to the spreadsheet page you saw for Darius a few pages back:





WATCH IT!



Take a moment to carefully review the data on this page.

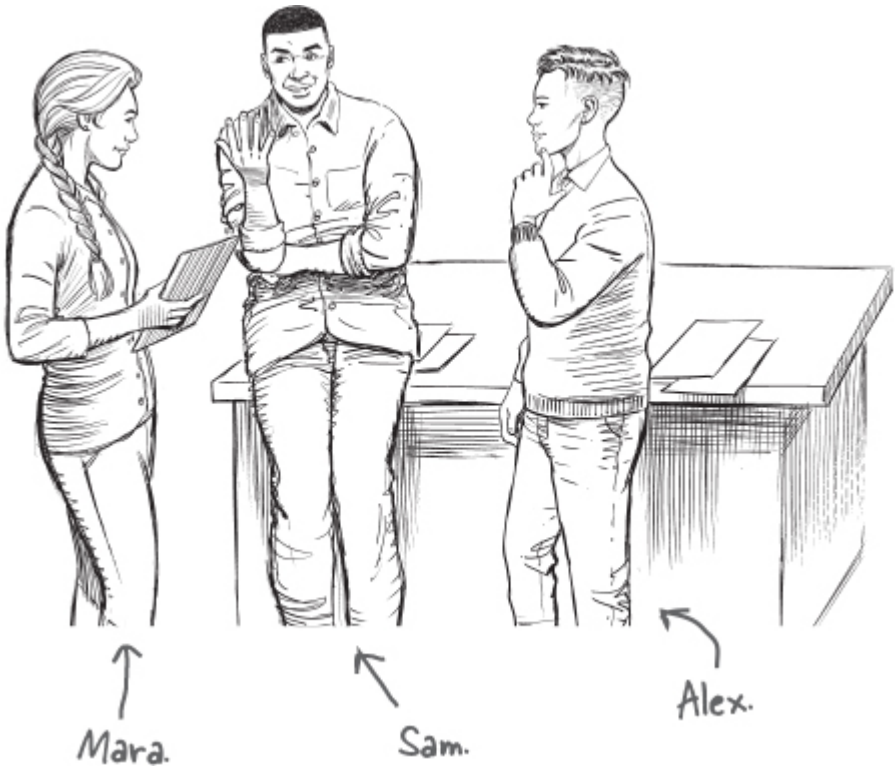
Specifically, note that information about the swimmer is encoded in the filename, whereas the contents of the file are a list of swim times for the swimmer. Both data are important, and both need to be processed.



The previous page's data looks promising.

If you can work out how to process one file, you can then repeat for any number of similarly formatted files.

The big question is: Where do you start?



Cubicle Conversation

Mara: OK, folks, let's offer some suggestions on how best to process this data file.

Sam: I guess there are two parts to this, right?

Alex: How so?

Sam: Well, firstly, I think there's some useful data embedded in the filename, so that needs to be processed. And, secondly, there's the timing data in the file itself, which needs to be extracted, converted, and processed, too.

Mara: What do you mean by "converted"?

Alex: That was my question, too.

Sam: I checked with the Coach, and "1:27.95" represents one minute, 27 seconds, and 95 one-hundredths of a second. That needs to be taken into consideration when working with these values, especially when calculating averages. So, some sort of value conversion is needed here. Remember, too, that the data in the file is *textual.*

Alex: I'll add "conversion" to the to-do list.

Mara: And I guess the filename needs to be somehow broken apart to get at the swimmer's details?

Sam: Yes. The "Darius-13-100m-Fly" prefix can be broken apart on the "-" character, giving us the swimmer's name (Darius), their age group (under 13), the distance swam (100m), and the stroke (Fly).

Alex: That's assuming we can read the filename?

Sam: Isn't that a given?

Mara: Not really, so we'll still have to code for it, although I'm pretty sure the PSL can help here.

Alex: This is getting a little complex…

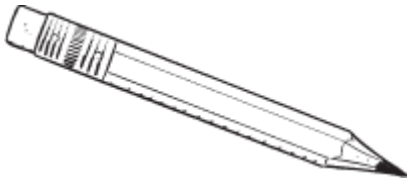
Sam: Not if we take things bit-by-bit.

Mara: We just need a plan of action.

Alex: If we're going to do all this work in Python, we'll also have a bit more learning to do.

Sam: I can recommend a great book… 😉

SHARPEN YOUR PENCIL



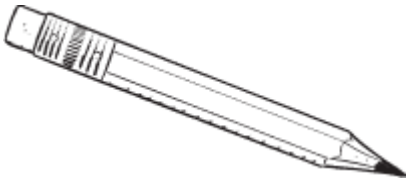
From the conversation on the last page, it looks like there are two main tasks identified at this stage: (1) extract data from the filename, and (2) process the swim times data in the file.

Grab your pencil and, for each of the identified tasks, write down what you think are the required sub-tasks for both (in the spaces provided). Our lists of sub-tasks can be found on the next page.

1 Extract data from the file's name

2 Process the data in the file

SHARPEN YOUR PENCIL SOLUTION



From the recent conversation, it looks like there are two main tasks identified at this stage: (1) extract data from the filename, and (2) process the swim times data in the file.

You were to grab your pencil and, for each of the identified tasks, write down what you thought the required sub-tasks are for both (in the spaces provided). Here's what we came up with. How did you do?.

1 Extract data from the file's name

a. Read the filename

b. Break the filename apart by the "-" character

c. Put the swimmer's name, age group, distance , and stroke into variables (so they can be used later)

2 Process the data in the file

a. Read the lines from the file

b. Ignore the second line

c. Break the first line apart by "," to produce a list of times

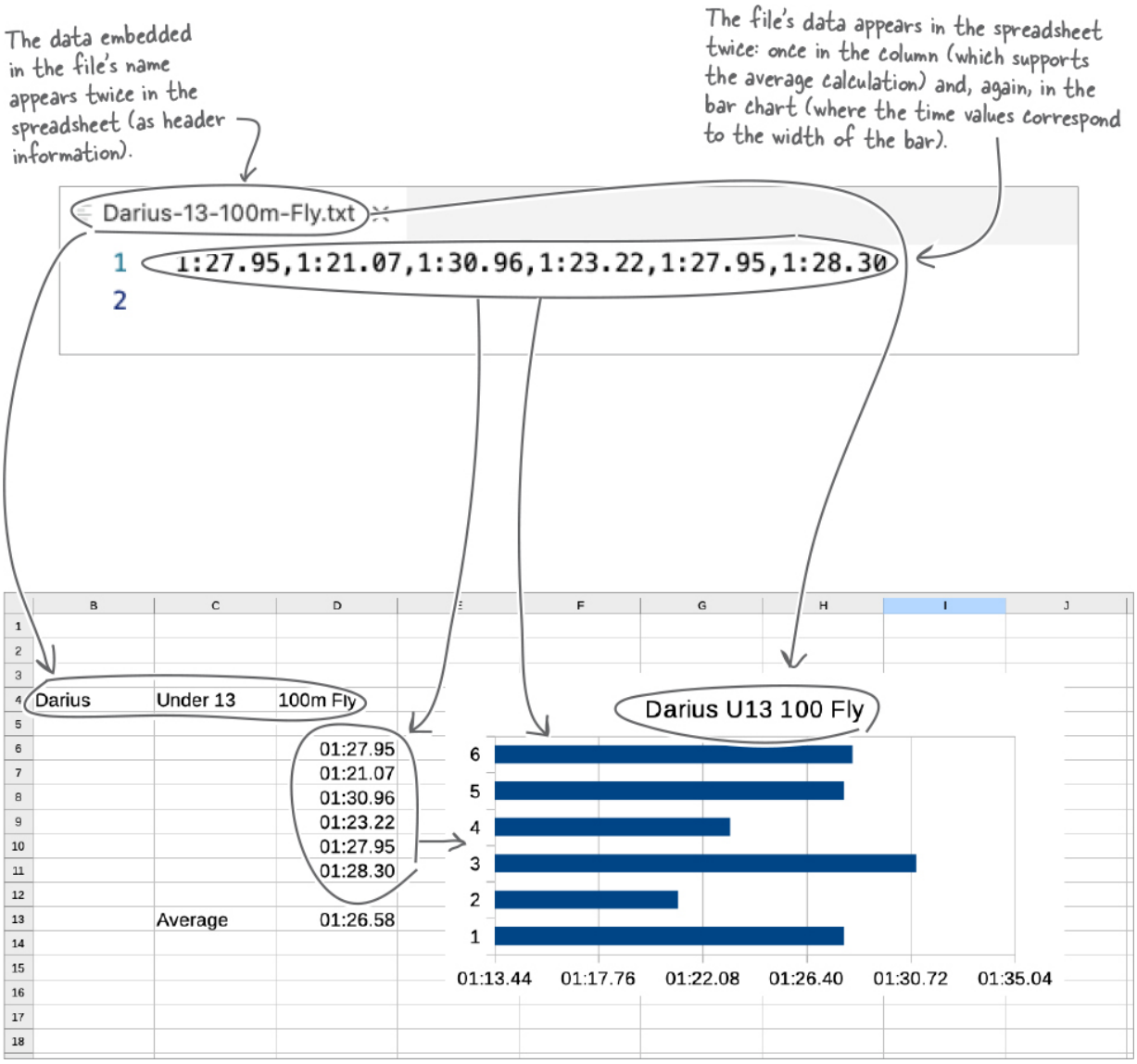
d. Take each of the times and convert them to a number from the "mins:secs.hundredths" format

e. Calculate the average time, then convert it back to the "mins:secs.hundredths" format (for display purposes)

f. Display the variables from Task #1, then the list of times and the calculated average from Task #2

The file and the spreadsheet are "related"

Before we dive into Task #1, take a moment to review how the data embedded in the file's name, together with the file's data, relates to the Coach's spreadsheet:



Our first task: Extract the filename's data

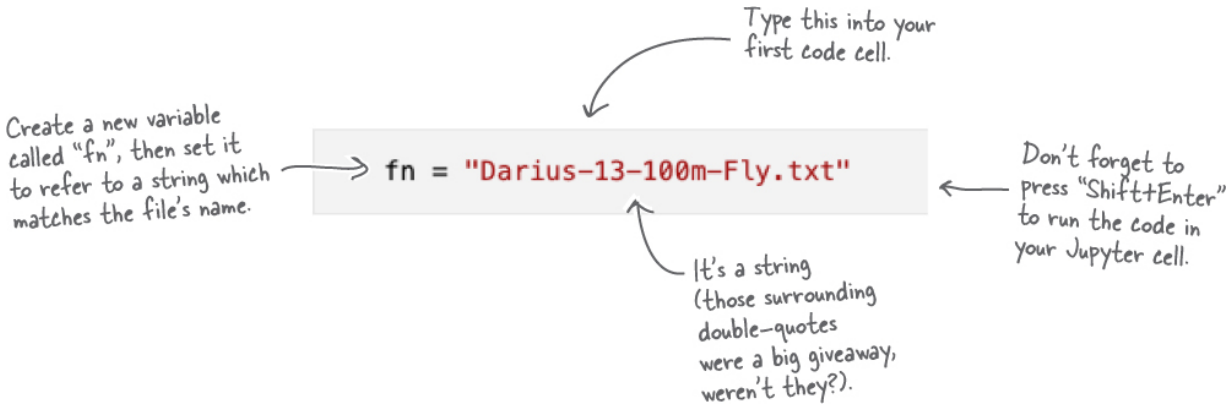
For now, the plan is to concentrate on one file, specifically the file which contains the data for the 100m Fly times for Darius, who is

swimming in the under 13 age group.

Recall the file containing the data you need is called `Darius-13-100m-Fly.txt`.

Let's create a Jupyter Notebook using VS Code called `Darius.ipynb`, which you can create in your **Learning** folder. Follow along in your notebook as, together, we work through Task #1.

Remember: To create a new notebook in VS Code, select **File** then **New File...** from the main menu, then select the **Notebook option.**



A string is not really a string…

The value to the right of the assignment operator (=) in the above line of code certainly *looks* like a string. After all, it's a sequence of characters enclosed within quotes which, in most other programming lan-

guages, is the very *definition* of a string. This is not the case in Python.

In Python, the value to the right of assignment operator is a **string object**, which isn't really that far off what a string is. However, the difference is important, as objects in Python are not just data values.

To see what we're on about here, you'll need to resort to the combo mambo.

In Python, everything is an object.



A string is an object with attributes

You already know how to list any object's attributes: use the **print dir** combo mambo.



Take a moment to appreciate what you're looking at here

If you are looking at that long list of attributes and thinking there's an awful lot to learn here, consider this instead: there's an awful lot of functionality built into strings that you *don't* have to write code for.


That sure is a big list. You want me to remember all of this?

RELAX



The **print dir** invocation produced a big list, but you only need to worry about half of it.

You can safely ignore all of the methods which start and end with the double underscore character, such as **__add__** and **__ne__**. These are this object's "magic methods" and they do serve a purpose, but it's far too early in your Python journey to worry about what they do and how you can use them. Instead, concentrate on the rest of the methods on this list.

THERE ARE NO DUMB QUESTIONS

Q: If those double-underscore methods are not important, why are they on the list returned by dir?

A: It's not that they aren't important, it's more a case that you don't need to concern yourself with what they do at this stage. Trust us, when you need to understand what the double-underscore methods do, we'll tell you. Pinky-promise.

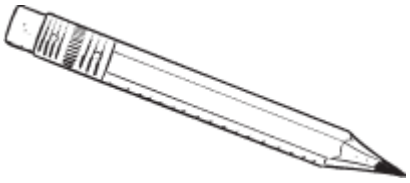
Q: Is there a way I can learn more about what a particular method does?

A: Yes, and we'll show you how in a page or two. What's cool is that using Jupyter makes this an especially easy thing to do.

Q: It's all a bit of a mouthful, all this double-underscore stuff, isn't it?

A: Yes, it is. Most Python programmers shorten "double-underscore add double-underscore" to simply "dunder add". So, if you hear someone refer to an method as "dunder exit", what they are actually referring to is **__exit__**. All of these (as a group) are called "the dunders". Further, any method which starts with a single-underscore is known as a "wonder" (and – yes – it is a perfectly acceptable reaction to groan at all of this).

SHARPEN YOUR PENCIL



Let's run two of the methods provided with strings. Take each of the lines of code shown below and enter them into a new, empty code cell. Execute each, then make a note (in the space provided) of what you think each function attribute does.

```
fn.upper()
```

_____

_____

```
fn.lower()
```

_____

_____



No problem. Great question, by the way.

This is Python's **dot** operator, which allows you to invoke a method on an object. This means `fn.upper()` calls the **upper** method on the string referenced by the `fn` variable.

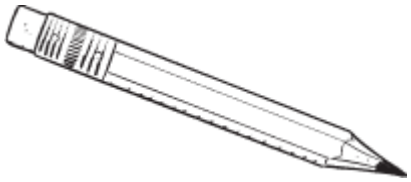
This is a little different from the BIFs which are invoked like functions. For instance, `len(fn)` returns the size of the object referred to by

the `fn` variable.

It's an error to invoke `fn.len()` (as there's no such method), just as it's an error to try `upper(fn)` (as there's no such BIF).

Think of things this way: The methods are object-specific, whereas the BIFs provide *generic* functionality which can be applied to objects of *any* type.

SHARPEN YOUR PENCIL SOLUTION



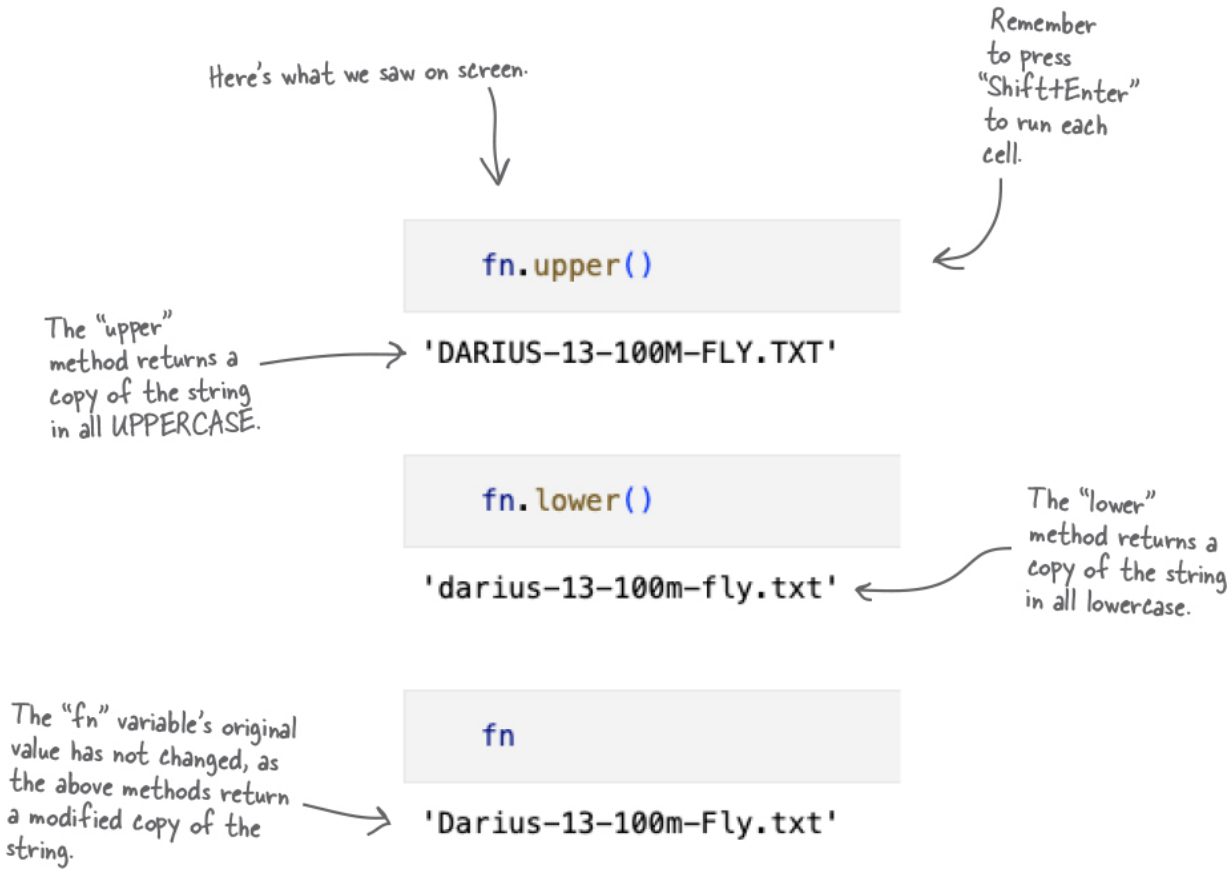
You were asked you run two of the methods provided. You were to take each of the lines of code shown below and enter them into a new, empty code cell. You were then to execute each, then make a note (in the space provided) of what you thought each method did. Here's what we think happens here:

```
fn.upper()
```

Return a copy of the value of what "fn" refers to in all UPPERCASE lettering.

```
fn.lower()
```

Return a copy of the value of what "fn" refers to in all lowercase lettering.


Here's what we saw on screen.
Remember to press "Shift+Enter" to run each cell.
fn.upper()
The "upper" method returns a copy of the string in all UPPERCASE.
'DARIUS-13-100M-FLY.TXT'
fn.lower()
The "lower" method returns a copy of the string in all lowercase.
'darius-13-100m-fly.txt'
The "fn" variable's original value has not changed, as the above methods return a modified copy of the string.
fn
'Darius-13-100m-Fly.txt'



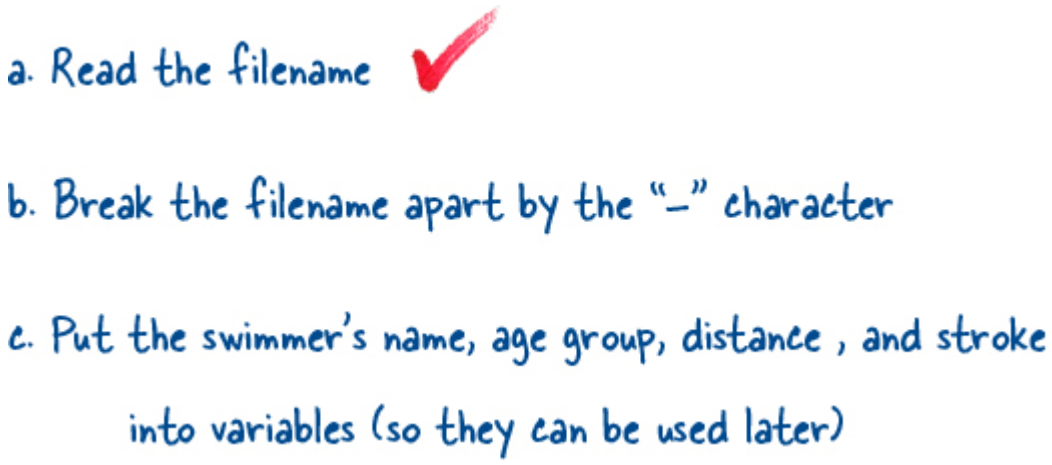
Yes, that's right.

The values returned by the **upper** and **lower** methods are both *new* string objects, which have their own a value-part and a methods-part. Although created from the data referred to by the `fn` variable, the new strings are *copies* of the string `fn` refers to.

This is all by design: Python is supposed to work this way.

Extract the swimmer's data from the filename

Recall the three sub-tasks identified earlier for Task #1:



As you've already got the filename in the `fn` variable, let's take it as given that sub-task (a) is done for now.

Breaking the filename apart by the "`-`" character is sub-task (b), and you'd be right to guess one of the string methods might help. But, which one? There's 47 of them!

```
'capitalize', 'casefold', 'center', 'count', 'enc
'endswith', 'expandtabs', 'find', 'format', 'forr
'index', 'isalnum', 'isalpha', 'isascii', 'isdeci
'isdigit', 'isidentifier', 'islower', 'isnumeric
```

```
 'isprintable', 'isspace', 'istitle', 'isupper',
 'ljust', 'lower', 'lstrip', 'maketrans', 'partit
 'removeprefix', 'removesuffix', 'replace', 'rfin
 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstr
 'split', 'splitlines', 'startswith', 'strip', 'sw
 'title', 'translate', 'upper', 'zfill'
```

NOTE

This is a big list of string methods. Whereas it's easy to guess what "upper" and "lower" do, it's not so clear what some of the others do, such as "case-fold", "format_map", or "zfill". What you're looking for is a method to help with sub-task (b).



Sounds interesting.

Let's see what the **split** method does to strings. You have a choice: You can run **split** and see what happens, or you can read **split**'s documentation.

Don't try to guess what a method does...

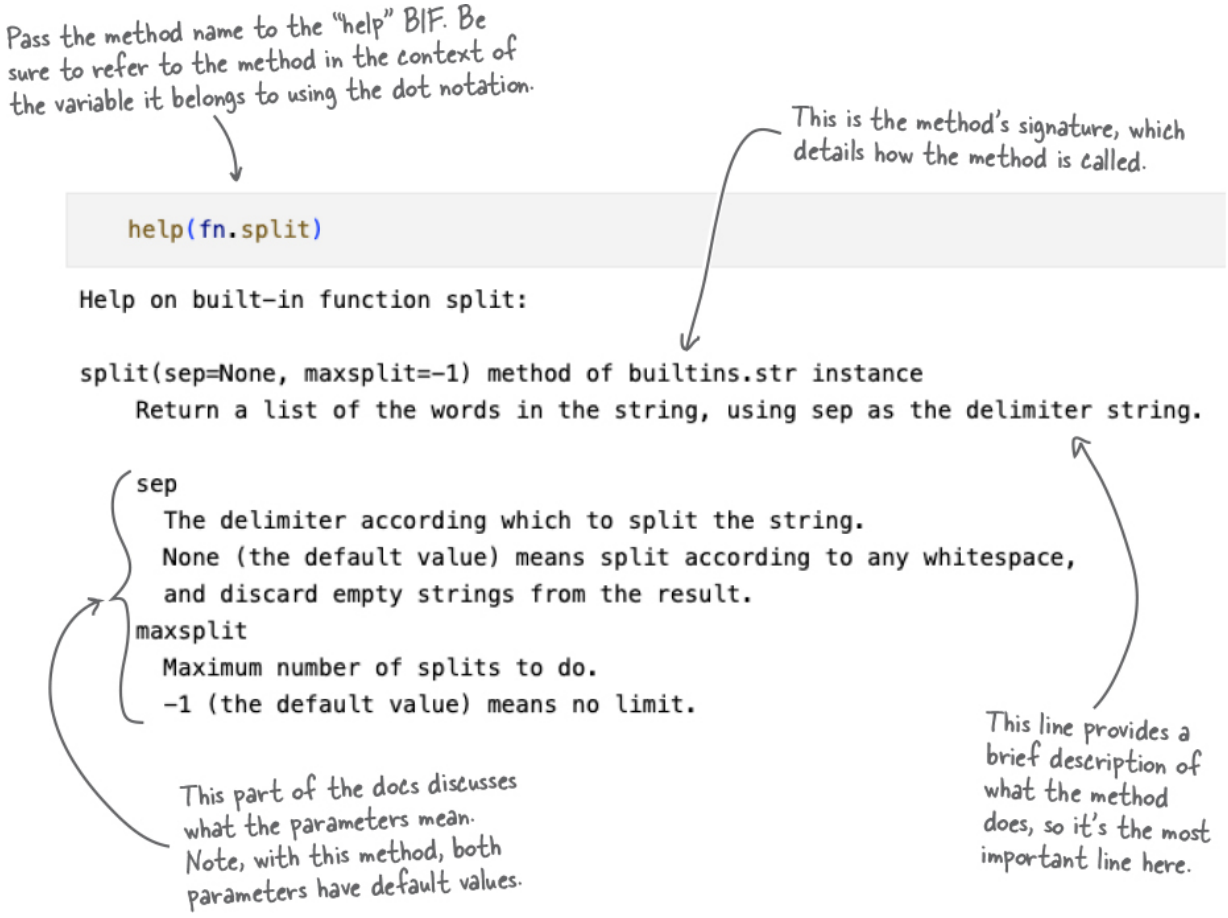
Read the method's documentation!

Now, granted, most programmers would rather eat glass than look-up and read documentation, claiming life is too short especially when there's code to be written. Typically, the big annoyance with this activity is the looking-up part. So, Python makes finding and displaying relevant documentation easy thanks to the **help** BIF.

Regrettably, Python can't read the documentation for you, so you'll still have to do that bit yourself. But the **help** BIF let's you avoid the context-shift of leaving VS Code, opening up your web browser, then searching for the docs.

To view the documentation for any method use the **help** BIF, like this:





Based on a quick read of this documentation, it sounds like the **split** method is what you need here. Let's take it for a quick spin.

TEST DRIVE



Let's continue to work within your `Darius.ipynb` notebook to explore what's possible with the **split** method. Be sure to follow along.

As an opening gambit, let's see what happens when we call **split** without supplying any arguments:

Using the dot operator, invoke the "fn" variable's "split" method (without arguments).

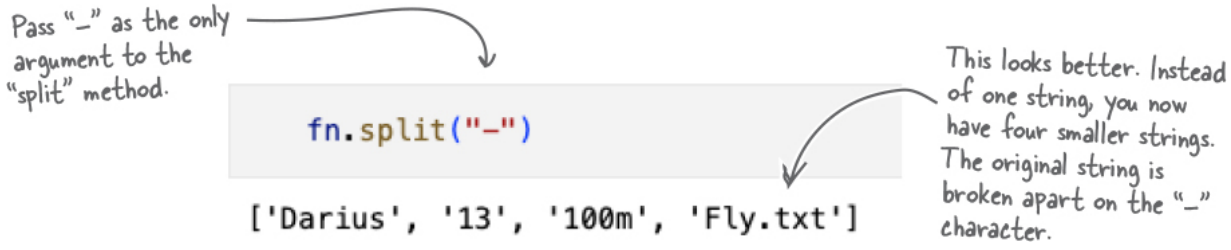
```
fn.split()
```

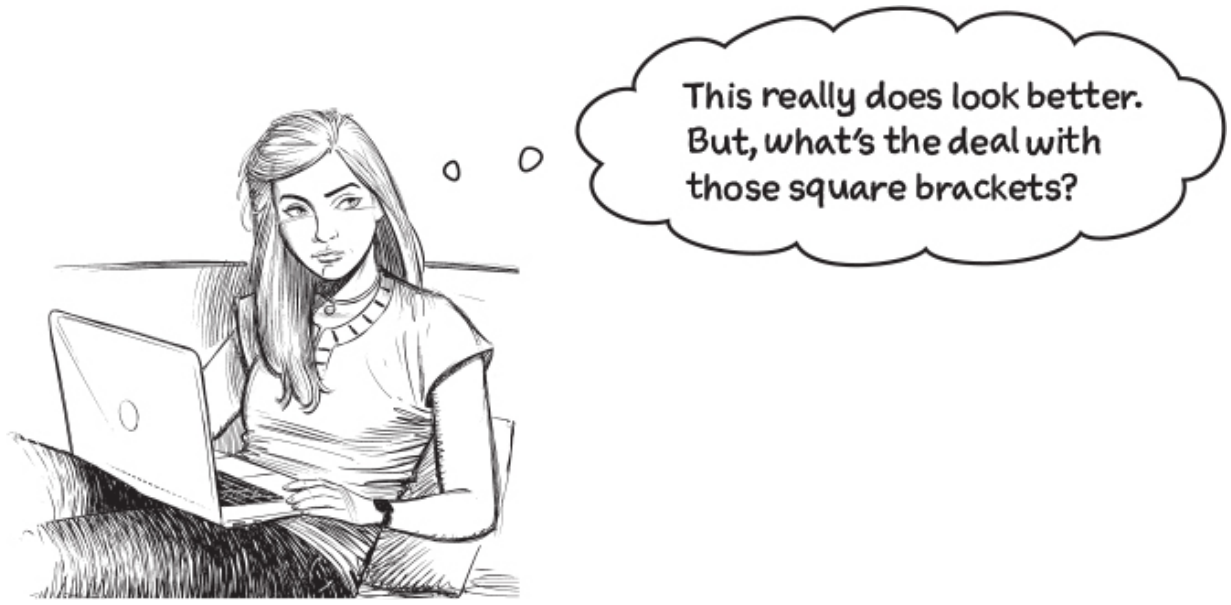
```
['Darius-13-100m-Fly.txt']
```

Ah, rats! That didn't do much other than surround the string with square brackets. This isn't very useful, is it?

If you flip back one page and re-read the **split** method's documentation, you'll learn the default behavior is to split on whitespace (e.g., space, tab, newline, carriage-return, formfeed, or vertical-tab). This is not what you want here, as you want to break the `fn` string apart on the "`-`" character.

Let's try again, this time specifying " - " as the delimiter. Doing so is easy, as all you do is pass the dash character as an argument to the **split** method call:





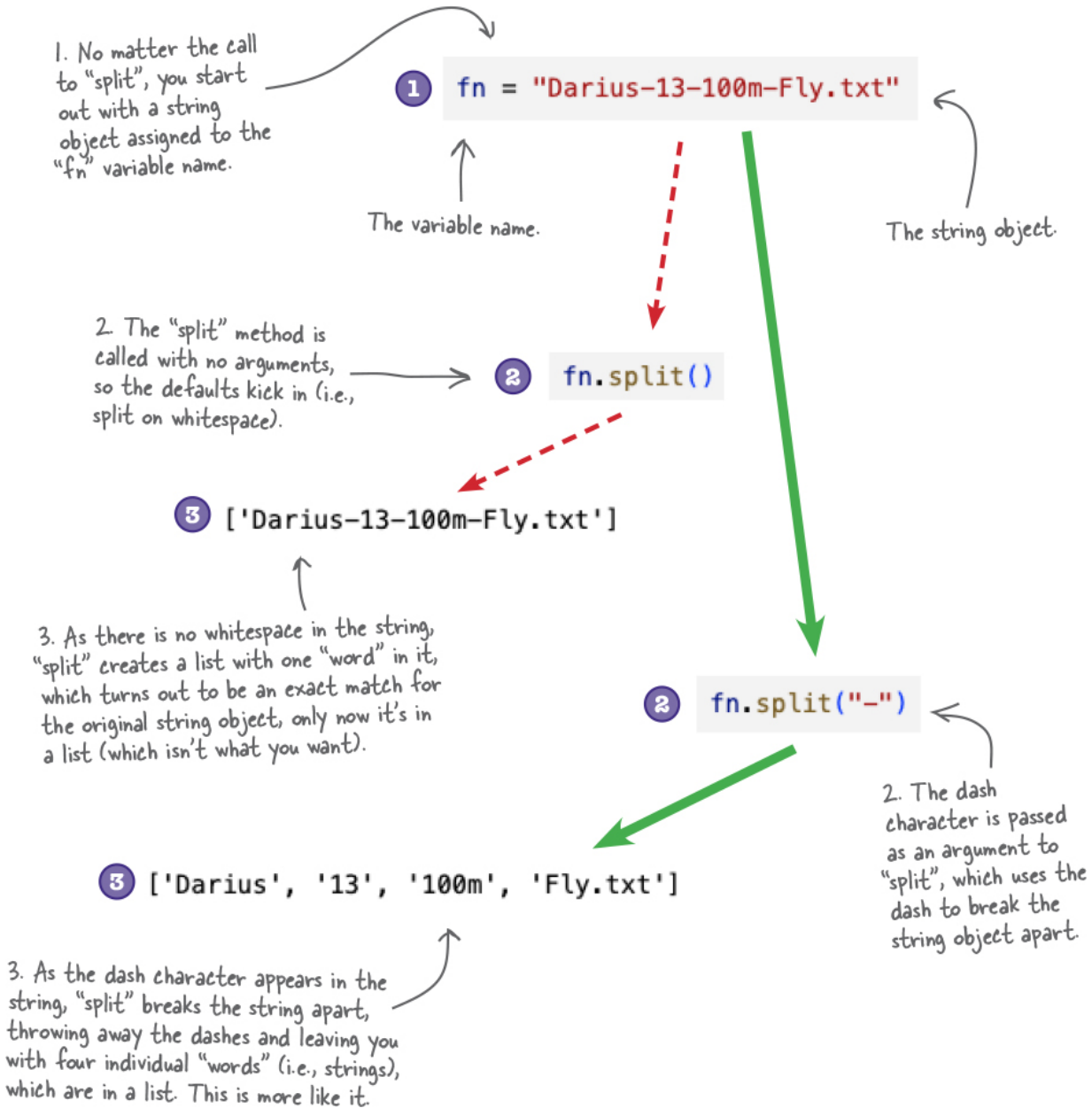
You did read the "split" method's documentation, didn't you? The answer's right there...

The split method returns a list of words...

In this context, you can think of a “word” as being a synonym for “string object”.

Lists in Python are data *enclosed* in **square brackets**.

Let’s review what just happened with the two calls to **split** shown in your most-recent *Test Drive*. We just can resist suggesting that doing so is as easy as 1-2-3:



Is it time for another checkmark?

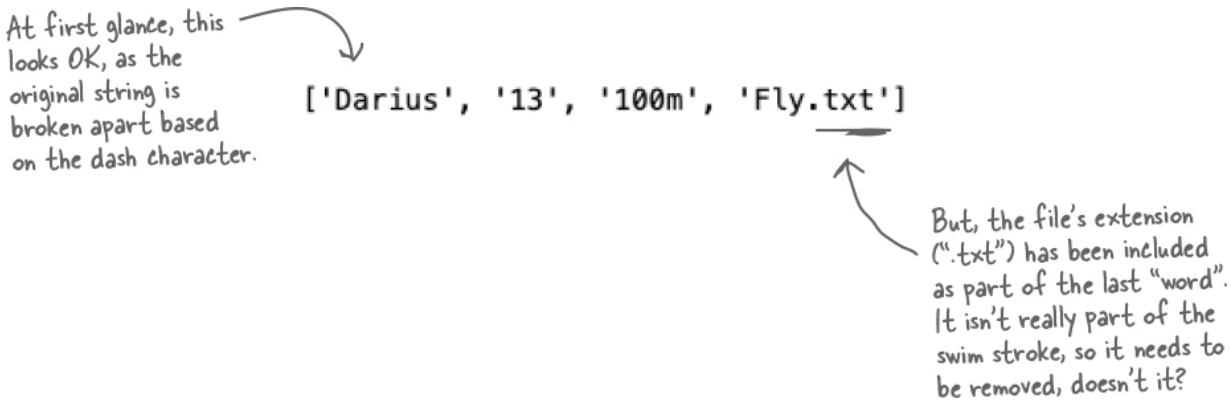
It's tempting to look at your list of sub-tasks, grab your pen, then put a satisfying checkmark beside sub-task (b), isn't it?

a. Read the filename ✔

b. Break the filename apart by the "-" character

c. Put the swimmer's name, age group, distance , and stroke into variables (so they can be used later)

But doing so would be *premature*. Take a closer look at the list produced by your call to the **split** method:





Emmm, maybe...

Let's spend a moment or two with **split** to ensure you understand how it works its magic.

EXERCISE



Let's take a moment to solidify your understanding of how **split** works. Here's the string assignment once more:

```
fn = "Darius-13-100m-Fly.txt"
```

Without first running these program statements in your notebook, see if you can describe what each of the statements do, noting down your answers in the spaces provided. If you get stuck, don't worry: Our answers start on the next page. And, BTW, once you've tried to work out what each statement does "in your head", feel free to double-check your work using VS Code (just don't start there).

1. `fn.split("13")`

2. `fn.split("-1")`

3. `fn.split(".")`

4. `fn.split("-.")`

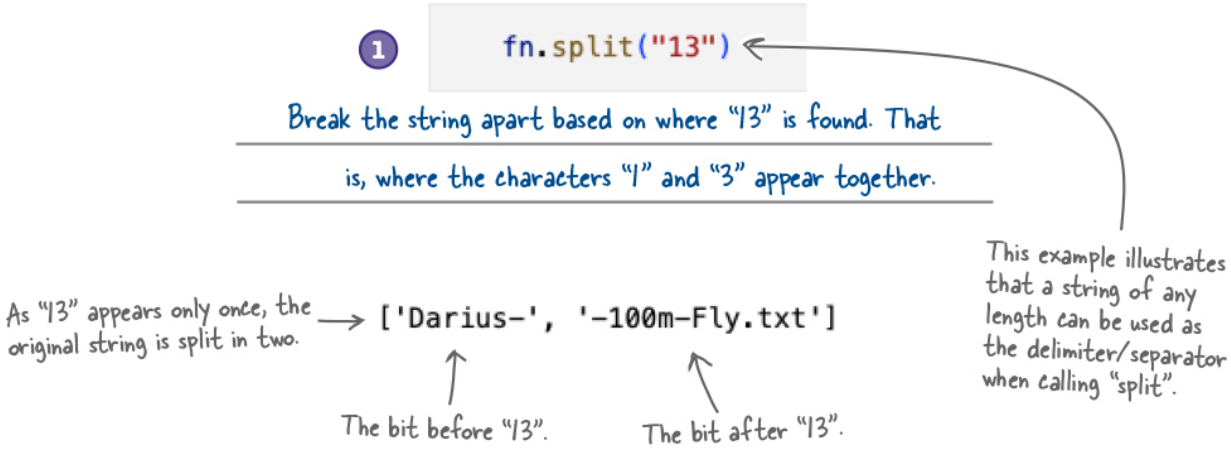
5. `fn.split("-").split(".")`

EXERCISE SOLUTION



You were to take a moment to solidify your understanding of how **split** works.

Without first running these program statements in your notebook, you were to see if you could describe what each of the statements do, noting down your answers in the spaces provided. Our answers are on this page and the next, together with what we see in VS Code when we execute each statement. How closely do your answers match?



2

```
fn.split("-1")
```

Break the string apart based on where "-1" is found. That is, where the character "-" and "1" appear together.

The string is split in three this time, as "-1" appears twice in the original string.

```
['Darius', '3', '00m-Fly.txt']
```

As with all calls to "split", the output is a list (note those square brackets).

Note that the delimiter has been removed by "split" from the returned results.

3

```
fn.split(".")
```

Break the string apart based on where "." is found, i.e., the dot character..

The string contains a single dot, so the produced list has two objects made up from the strings before and after the dot character.

```
['Darius-13-100m-Fly', 'txt']
```

4

```
fn.split("-.")
```

Break the string apart based on where "-." is found, i.e, where the character "-" and the "." appear together.

```
['Darius-13-100m-Fly.txt']
```

Although the "-" and the "." appear in the string, they do not appear together, so no splitting occurs. This is not an error. The "split" method returns the original string in a list (as per the default behavior).

5

```
fn.split("-").split(".")
```

Break the string apart based on where "-" is found, then do another split on the "." character (i.e., dot)...??

Nope. It doesn't do this!

Yikes!

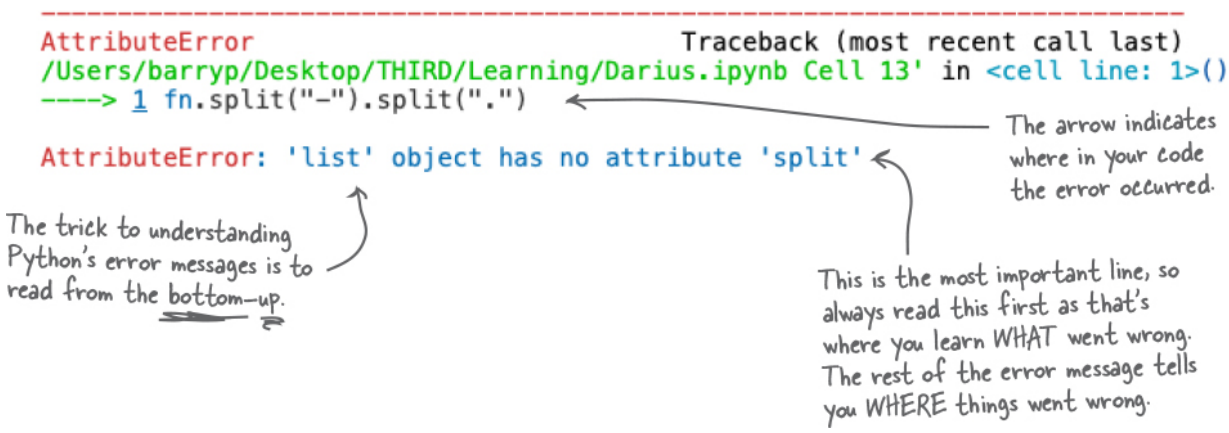
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/Users/barryp/Desktop/THIRD/Learning/Darius.ipynb Cell 13' in <cell line: 1>()
----> 1 fn.split("-").split(".")

AttributeError: 'list' object has no attribute 'split'
```

The last example was an error, producing a hairy, scary run-time error message. Flip the page to learn more about what's going on here.

Read error message from the bottom-up

The last example in your most-recent *Exercise* produced a run-time error message which likely has you scratching your head:





Err... Okay.

Whatever wets your whistle.

Just remember to always read Python's error messages from the *bot-tom-up*, and you'll be fine (magic potions, notwithstanding). Also, note that Python refers to its error messages by the name **traceback**.

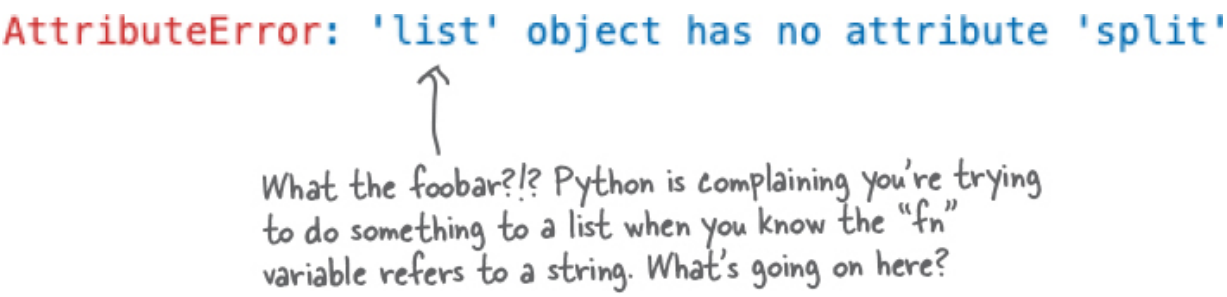
Now... just what is this particular traceback trying to tell you?

Be careful when chaining method calls

The idea behind that last example is solid: specify a *chain* of calls to **split** to break the string object on " `-` " then again on " `.`



Of course, this line of code failed, which is a bummer because the idea was sound, in that you want to split your string *twice* in an attempt to break the strings " `Fly` " and " `txt` " apart. But, look at the error message you're getting:





Yes, that's exactly what's happening.

The first **split** works fine, breaking the string object using " – ", producing a list. This list is then passed onto the next method in the chain which is *also* **split**. The trouble is lists do *not* have a **split** method, so trying to invoke **split** on a list makes no sense, resulting in Python throwing it's hands up in the air with an `AttributeError`.

Now that you know this, how do you fix it?

Fixing broken chains

Let's see what the *Head First Coders* think your options are.



The Head First Coders:
Alex, Mara, and Sam.

Cubicle Conversation

Alex: I think we should pause development while we learn all about lists...

Sam: Well, lists are important in Python, but I'm not so sure this is a list issue, despite what that traceback is telling us.

Mara: I agree. We're trying to manipulate the `fn` *string*, not a list. The fact the traceback references a list is a little confusing, and it's

happening because **split** always returns a list.

Alex: And we can't call **split** on a list, right?

Sam: No, you can't, due to the fact that lists don't include a built-in **split** method, whereas strings so. That's what the traceback is telling you: there's no such thing as a list **split** method.

Alex: So, we're out of luck?

Mara: No, we just need to rethink things a little...

Sam: Remember: `fn` is a string. So start by asking yourself if there's anything else built into strings which might help.

Alex: How do I do that again?

Sam: With your old friend, the **print dir** combo mambo.

BRAIN POWER



You're trying to get rid of that "`.txt`" bit at the end of the original string. Here's the list of string methods. Do any of these method names jump out at you?

```
'capitalize', 'casefold', 'center', 'count', 'enc
'find', 'format', 'format_map', 'index', 'isalnur
'isdecimal', 'isdigit', 'isidentifier', 'islower
'isspace', 'istitle', 'isupper', 'join', 'ljust',
'maketrans', 'partition', 'removeprefix', 'remove
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstri
'startswith', 'strip', 'swapcase', 'title', 'tran
```

Strings can do more than just split

As Python has trained us to always read from the bottom-up, the first method to catch our eye is **rstrip**. Use the **help** BIF to learn a bit about **rstrip** right from within your VS Code notebook:




```
help(fn.rstrip)
```

```
Help on built-in function rstrip:

rstrip(chars=None, /) method of builtins.str instance
    Return a copy of the string with trailing whitespace removed.

    If chars is given and not None, remove characters in chars instead.
```

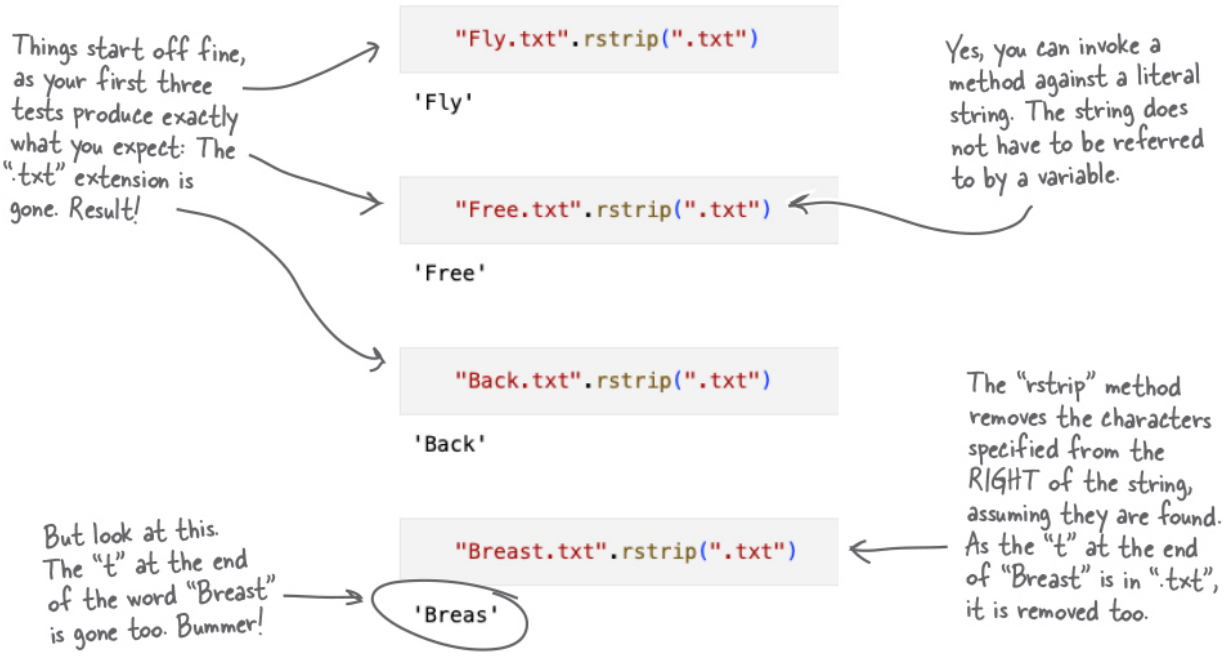
This isn't really what you want.

But, this statement shows a bit more promise.

TEST DRIVE



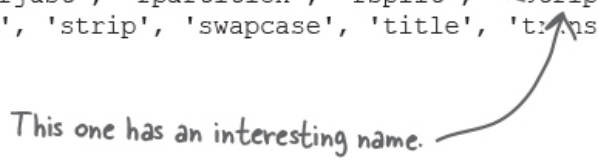
Follow along in VS Code while you test the **rstrip** method against some test values similar to those you'll encounter.



Let's try another string method

Undaunted, let's return to the list of string methods to continue your search:

```
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill'
```



As with the **rstrip** method, ask the **help** BIF for details on what **removesuffix** does:

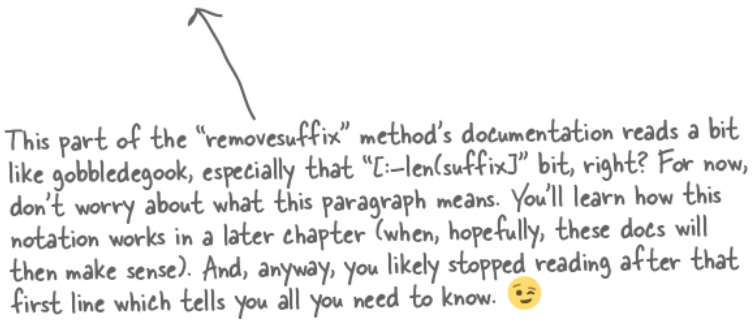
```
help(fn.removesuffix)
```

```
Help on built-in function removesuffix:

removesuffix(suffix, /) method of builtins.str instance
    Return a str with the given suffix string removed if present.

    If the string ends with the suffix string and that suffix is not empty,
    return string[:-len(suffix)]. Otherwise, return a copy of the original
    string.
```





Let's take this method for a spin in your notebook.

TEST DRIVE



As when you tested **rstrip**, let's throw some test data at the **removesuffix** method, too:

```
"Fly.txt".removesuffix(".txt")
'Fly'

"Free.txt".removesuffix(".txt")
'Free'

"Back.txt".removesuffix(".txt")
'Back'

"Breast.txt".removesuffix(".txt")
'Breast'
```

Feel free to bask in the glory, here. There are no issues this time, and all four test cases perform as you'd expect them too. It looks like "removesuffix" is the method you're looking for.

Did you notice that "rstrip" and "removesuffix" both returned strings (not lists)? This has an important implication as you can chain a method which returns a string with another string method (and avoid that nasty AttributeError).

Now that you've identified your required methods, create a method chain which extracts the data you need from the `fn` string object:

Invoke the "removesuffix" method on the original string object then, having removed the ".txt" bit, invoke "split" to break what's left on the "-" character, creating a list of the data you require.

```
fn.removesuffix(".txt").split("-")
```

```
['Darius', '13', '100m', 'Fly']
```

This is looking good!

It's time for a checkmark.

a. Read the filename ✔

b. Break the filename apart by the "-" character ✔

We don't know about you, but this one felt good!

c. Put the swimmer's name, age group, distance , and stroke into variables (so they can be used later)

THERE ARE NO DUMB QUESTIONS

Q: Can method chains be of any length?

A: Yes. Although you do need to think about code readability. The examples seen thus far have chained two methods together, which is not hard to get your head around. But imagine if a programmer decides to chain a dozen methods together? Python let's the programmer do this, but you'll likely pull your hair out trying to decipher what such a large chain does... or maybe you'll want to hunt down the programmer so you can pull *their* hair (not that we're condoning such behavior... it's just that we understand the urge).

Q: When I run the code in this chapter the original string in my `fn` variable never changes. What if I want to apply the changes to the original string. Can I do this?

A: The short answer is no. The longer answer is also no, in that you cannot change a string object in Python once it's defined. Strings in Python are immutable (they cannot mutate or change once they exist). This behavior is by design, and sometimes strikes programmers from other languages as strange. Python treats strings as a basic data type, like numbers. Numbers are also immutable. Once `42` exists in your code it cannot be mutated nor changed. It's always `42`. Same thing with strings. If a string has the value "`Marvin`", it'll re-

main that value until one of two things happen: (1) your code terminates or (2) the Universe ends.

Q: Is the fact that strings are immutable not a huge disadvantage?

A: Not really. Knowing that strings can never change frees you from having to worry about a whole host of nasty side-effects.

Q: Let me get this straight: When I assign the string "`Galaxy`" to a variable called `place`, I can never change `place`'s value later in my code due to strings being immutable?!?

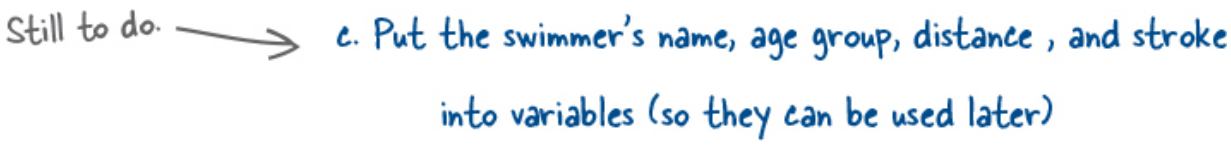
A: No, that's not what this means, as it's not the same thing. The string "`Galaxy`" once defined can never change. However, when the string is assigned to your `place` variable, the string's *object reference* is assigned to `place`, not the actual string "`Galaxy`". It's perfectly legal, later in your code, to assign a *different* object reference to `place` (that's what variables are in Python: somewhere to store an object reference).But, the string object which contains "`Galaxy`" can *never* change. The string object "`Galaxy`" and the variable name `place` are two *different* things.

Q: So, what happens to a string ("`Galaxy`" for example) which is assigned to a variable in some code then, later, the variable is assigned some other value? Does "`Galaxy`" just hang around?

A: Doesn't *every* galaxy just hang around? [Apologies to all the Astronomers reading this]. Joking aside, once any previously created object in Python gets to the stage where it is no longer referred to by any variable, it is garbage collected by Python's memory management system. You don't need to do anything to make this happen, as Python takes care of all the details for you.

All that remains is to create some variables

There's one last sub-task to complete, namely part (c):

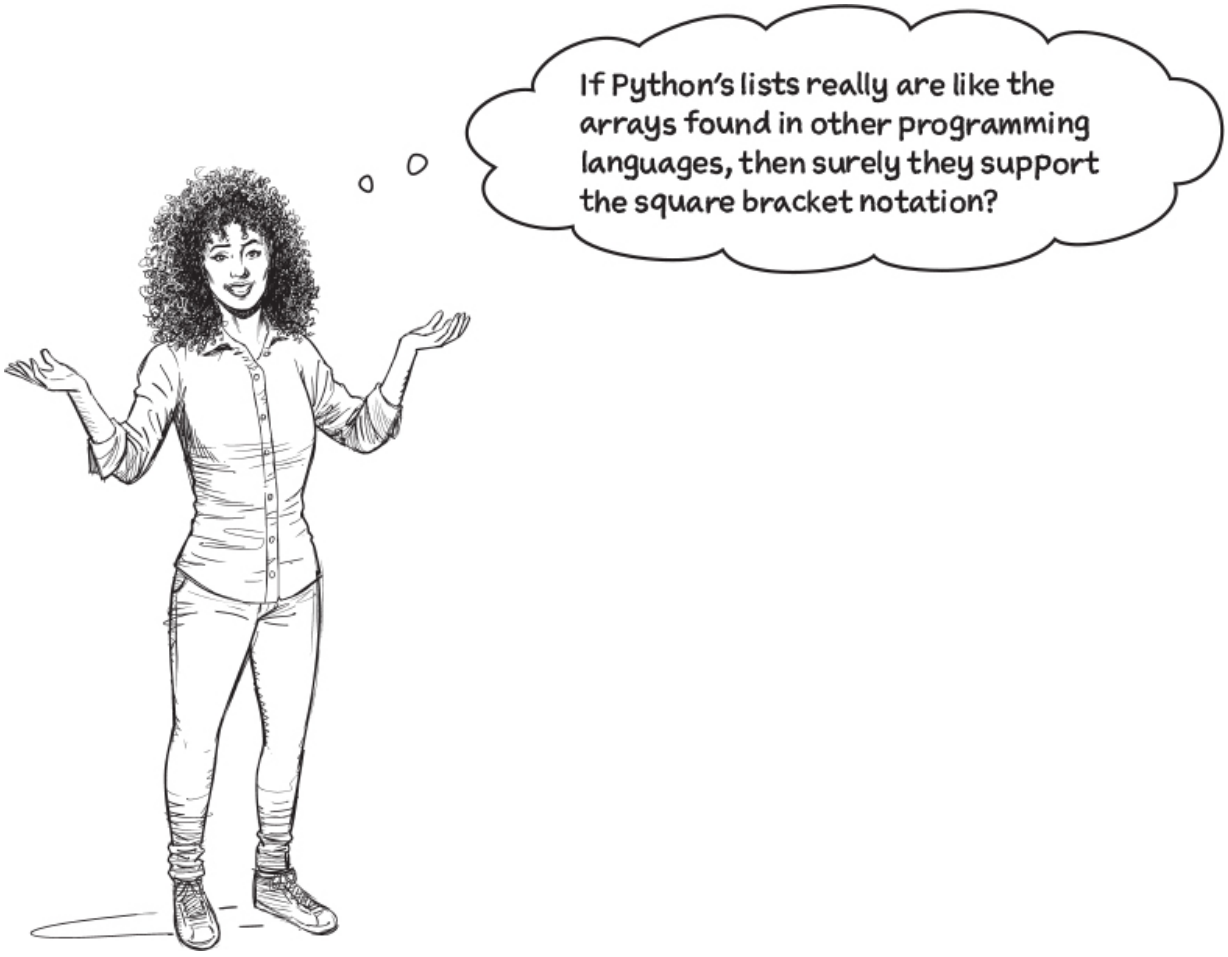


Your last line of code produced a list with the four values you need, but how do you assign each of these four values to individual variables?

```
fn.removesuffix(".txt").split("-")
['Darius', '13', '100m', 'Fly']
```

The use of square brackets is your clue that this is a list.

The four data items are in the list, ready to be used.



Indeed they do.

When working with lists, it is possible to use the familiar square bracket notation. And, as in most other programming languages,

Python starts counting from zero, so `[0]` refers to the first element in the list, `[1]` the second, `[2]` the third, and so on.

Let's put this new-found list knowledge to immediate use.

NOTE

Don't forget to follow along!

TEST DRIVE



The line of code from the last page produces a list of four data values:

Take what "fn" refers to, remove the ".txt" suffix, then break apart the resultant string on the "-" character, producing a list of data values.

```
fn.removesuffix(".txt").split("-")
['Darius', '13', '100m', 'Fly']
```

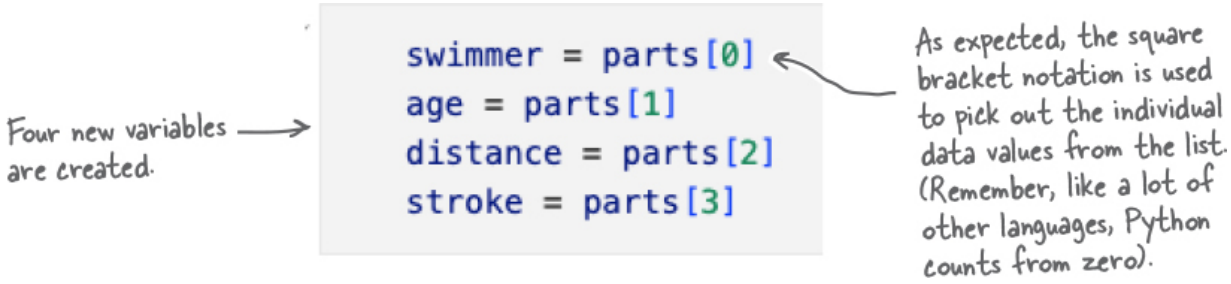
Let's assign this generated list to a variable called `parts`, remembering that `parts` is *not* a list, it's a variable name which contains an object reference to the generated list:

The "parts" variable is assigned the list's object reference. When you refer to "parts", Python does the internal double-lookup and gives you back your data.

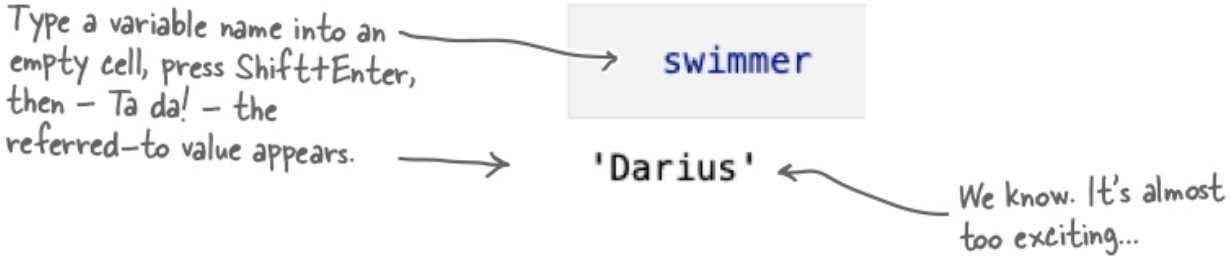
```
parts = fn.removesuffix(".txt").split("-")
parts
['Darius', '13', '100m', 'Fly']
```

Although it's a bit of an oversimplification, you can often think of Python lists as being like arrays... so, knowing this, you can use the

square bracket notation to assign each individual data value to its own variable:

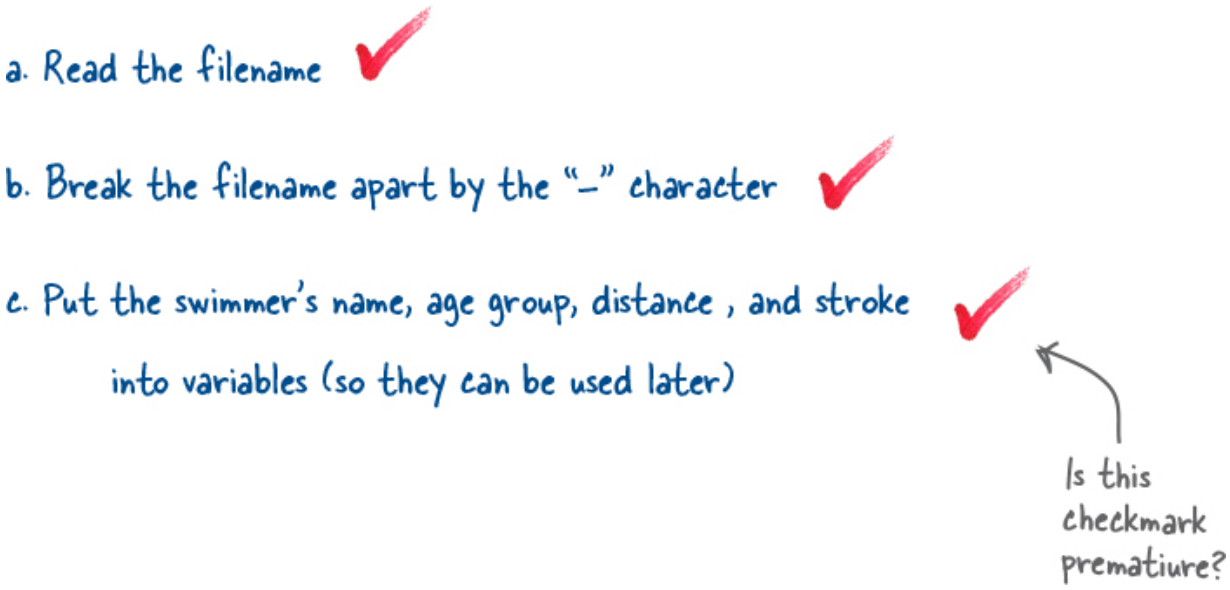


And sure enough, each individual variable name refers to an individual data value. Take `swimmer`, for instance:



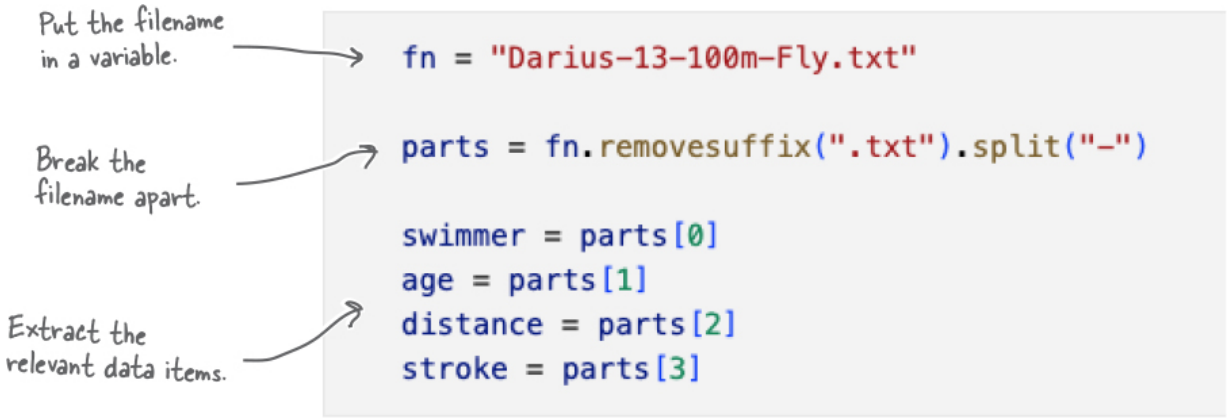
It looks like Task #1 is complete. But is it?

Let's remind ourselves of the sub-tasks for Task #1, which was to extract the data you need from the file's name:



Considering the code from your last *Test Drive*, you're likely done with Task #1.

Having used VS Code and Jupyter to work out the code you need, it's a simple task to copy'n'paste the code for Task #1 into a new cell. And the amount of code you need to copy isn't overwhelming, is it?



We were all set to begin celebrating getting to this point, but it looks like someone has a question...



The `parts` variable feels kind of integral.

That said, it's a valid question, in that `parts` is created to *temporarily* hold the list of data items, which is then combined with the square bracket notation to extract the individual data items. Once that's done, the `parts` list is no longer needed.

So, can you do without `parts`?

NOTE

And if the "parts" variable is not needed, does this mean it's spare?!? (Sorry).

Can you do without the parts list?

Short answer: yes.

Of course, getting to the point where you understand the short answer is a bit more involved. But, don't worry, it'll all make sense in a bit.

The first thing to remember is that the **split** method at the end of the chain of calls on the `fn` variable produces a list:

```
fn.removesuffix(".txt").split("-")
```

This call to "split" at the end of the chain produces a list.

The list is then assigned to the `parts` variable name, allowing you to use the square bracket notation to access the data you need:

Assign the list to the "parts" name.

```
parts = fn.removesuffix(".txt").split("-")

swimmer = parts[0]
age = parts[1]
distance = parts[2]
stroke = parts[3]
```

There's nothing new here, with the square bracket notation working just as you'd expect it to.

The **split** method always produces a list, so you could do what's shown below to achieve the same thing as what's shown above, removing the `parts` variable from the code:

```
swimmer = fn.removesuffix(".txt").split("-")[0]
age = fn.removesuffix(".txt").split("-")[1]
distance = fn.removesuffix(".txt").split("-")[2]
stroke = fn.removesuffix(".txt").split("-")[3]
```

Each call to "split" generates the list, then the square brackets pick out the required data.

Although the `parts` variable is no more, can you think of a reason why this version of your code may not be optimal?

We can think of three reasons

The latest code *does* work, but at a cost.

1 The latest code is slow

The original code generated the list *once*, assigned it to the parts list, then used the list as needed. This is efficient. The latest code generates the list *four times*, which is hugely inefficient.

2 The latest code is harder to read

It's clear what the original code is doing, but the same can't be said for the latest code, which – despite being a clever bit of Python – does require a bit of mental gymnastics to work out what's going on. The code looks (and is) more complex as a result.

3 The latest code is a maintenance nightmare

If you're asked to change the suffix from `".txt"` to say, `".py"`, it's an easy change when working with the original code (as it's a single

edit). With the latest code, you have to apply the edit four times (multiple edits) which can be fraught with danger.



Let's not be too hasty.

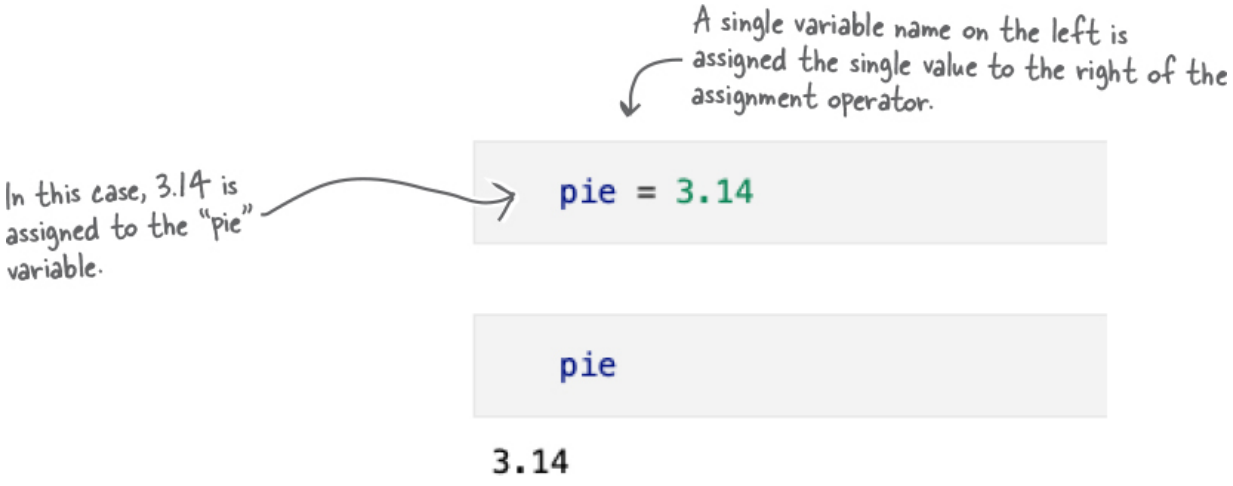
Yes, the latest code is more trouble than it's worth, but the idea of removing the `parts` variable from your code still has merit, as it's of no use once the assignment to `swimmer`, `age`, `distance`, and `stroke` have occurred.

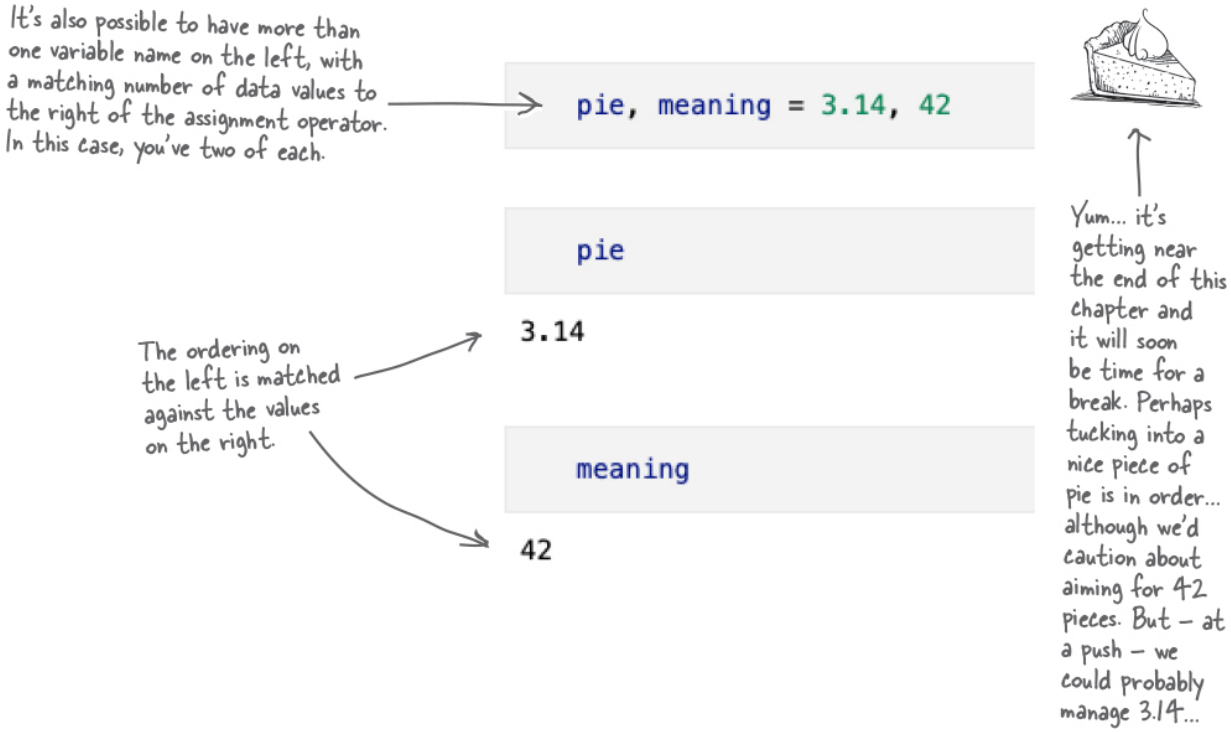
There may be another way to do this...

Multiple assignment (aka unpacking)

Although the concept is not unique to Python, the notion of *multiple assignment* is a powerful feature of the language. Also know as **unpacking** within the Python world, this feature lets you assign to more than one variable on the left of the assignment operator with a matching number of data values to the right of the assignment operator.

Here's some example code which demonstrates how this works:





Note the following: You can match *any number* of variable names against values (as long as the number of each on both sides of the assignment operator match). Python treats the data values on the right as if they are a list, but does not require you to enclose the list within square brackets.

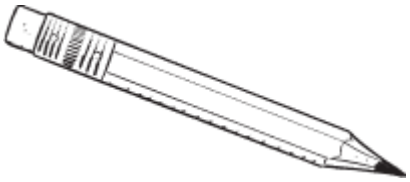


Yes, that's what this means.

Python programmers describe the list as being “unpacked” prior to the assignment, which is their way of saying the list’s data values are taken one-by-one and assigned to the variable names one-by-one.

The single list is *unpacked* and its values are *assigned* to *multiple* variable names, one at a time.

SHARPEN YOUR PENCIL



Grab your pencil and see if you can fill in the blanks below. Based on what you now know about multiple assignment (unpacking), provide the individual lines of code which assign the correct unpacked values to the individual variable names. Provide the printed output, too.

There's a line of code missing from here. Add it into the space provided.

```
fn = "Darius-13-100m-Fly.txt"

_____________________________________________________

print(swimmer)
print(age)
print(distance)
print(stroke)
```

_____

_____

_____

_____

Write down the four values you'd expect to see displayed on screen once the above code cell runs.

_____

_____

_____

values you'd expect to see displayed on screen once the above code cell runs.

There's a line of code missing from here. Add it into the space provided.

```
fn = "Abi-10-50m-Back.txt"


_____________________________________________________

print(swimmer)
print(age)
print(distance)
print(stroke)
```

_____

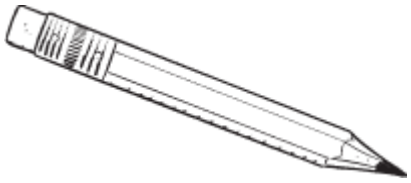
_____

_____

_____

Write down the four values you'd expect to see displayed on screen once the above code cell runs.

SHARPEN YOUR PENCIL SOLUTION



You were to grab your pencil and see if you could fill in the blanks. Based on what you knew about multiple assignment (unpacking), you were to provide the individual lines of code which assign the correct unpacked values to the individual variable names. You were to provide the printed output, too. Here's our code, below. Is your code the same?

```
fn = "Darius-13-100m-Fly.txt"

swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")

print(swimmer)
print(age)
print(distance)
print(stroke)
```

The list created by the "split" call is unpacked and used to assign values to multiple variables.

```
Darius
13
100m
Fly
```

This is looking good! Each individual variable has been assigned the correct value extracted from the file's name.

```
fn = "Abi-10-50m-Back.txt"

swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")

print(swimmer)
print(age)
print(distance)
print(stroke)
```

The file's name has changed, but not the code.

```
Abi
10
50m
Back
```

As expected, you're seeing different output values due to the use of a different filename. And, did you notice there's not a "parts" list in sight?

Task #1 is done!

Recall the list of sub-tasks once more:

a. Read the filename ✔

b. Break the filename apart by the "-" character ✔

c. Put the swimmer's name, age group, distance , and stroke into variables (so they can be used later) ✔

Task #1: extract data from the file's name.

Once the file's name is in the `fn` variable, a *single line* of Python code does all the heavy lifting:

```
swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
```

That's a pretty powerful line of code. We've told the Coach that work is progressing.



Yes, they are. We are on our way!

Of course, we've neglected to tell the Coach that there's still a bit of work to do. Let's remind everyone what Task #2 entails (shown on the next page).

Task #2: Process the data in the file

At first glance, it looks like there's a bit of work here. But, don't fret: we'll work though all of these sub-tasks *together* in the next chapter:

1. Read the lines from the file
2. Ignore the second line
3. Break the first line apart by "," to produce a list of times
4. Take each of the times and convert them to a number from the "mins:secs.hundredths" format
5. Calculate the average time, then convert it back to the "mins:secs.hundredths" format (for display purposes)
6. Display the variables from Task #1, then the list of times and the calculated average from Task #2

You've just enough time to do a few things before diving into your third chapter. Go make yourself a cup of your favorite brew, grab a piece of pie, read through the chapter summary, then work through this chapter's crossword *at your leisure*.



CHAPTER REVIEW BULLET POINTS

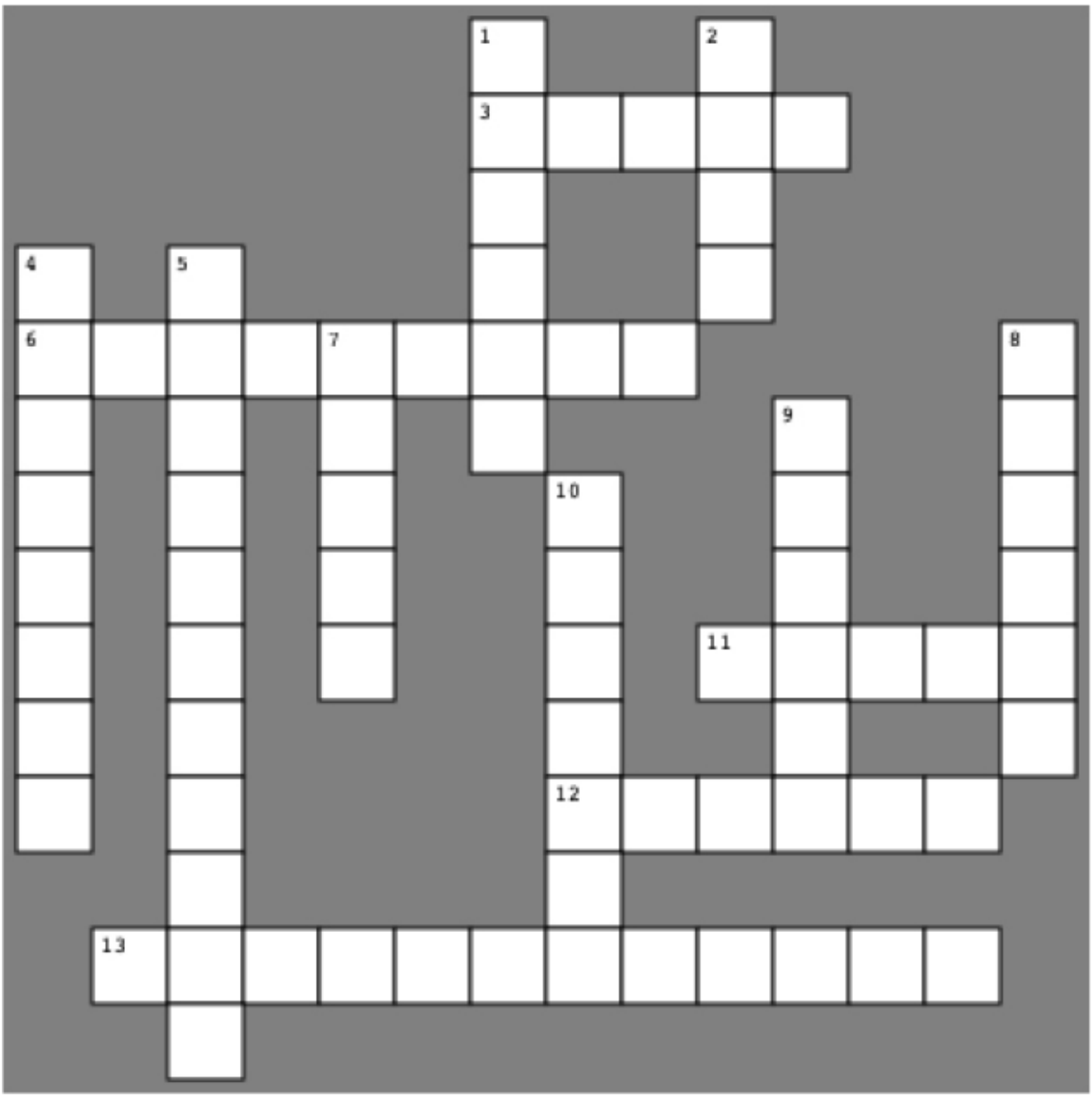
- Python's strings aren't just strings, they're **string objects**, which come jam-packed with built-in functionality.
- An object's attributes (which include a list of methods which act on the object's value) can be listed using the **print dir** combo mambo.
- Given the name of an object, the familiar **dot notation** accesses any method, e.g., `fn.upper()`.
- There are lots of string methods, but everyone's favorite has to be **split** which, although a string method, returns a list when invoked.
- Object methods can be **chained**, but care is needed to avoid an **AttributeError** (see the previous point).
- When Python error messages do appear, always read them from the **bottom-up**.
- The chain of methods made by **removesuffix** and **split** is a powerful combination.
- Although we've only briefly touched on **lists** (so far), it was a nice surprise to learn that lists understand the standard **square bracket notation** (which most other programming languages associate with arrays).
- Multiple assignment (aka **unpacking**) is a powerful technique which allows you to assign to as many variables as needed with

a single assignment statement. This feels like a Python **super-power**, and it is.

The String Crossword



All of the answers to the clues are found in this chapter's pages, and the solution is on the next page. Have fun!



Across

3. Comes in handy when breaking apart strings.

6. Another name for multiple assignment.

11. To be a variable, it has to have one of these.

12. Everything is one of these.

13. Can be used to get rid of a filename's extension.

Down

1. Removes a set of characters from the end of a string.

2. Like an array in other languages.

4. More than one.

5. The swim coach mucks about with one of these.

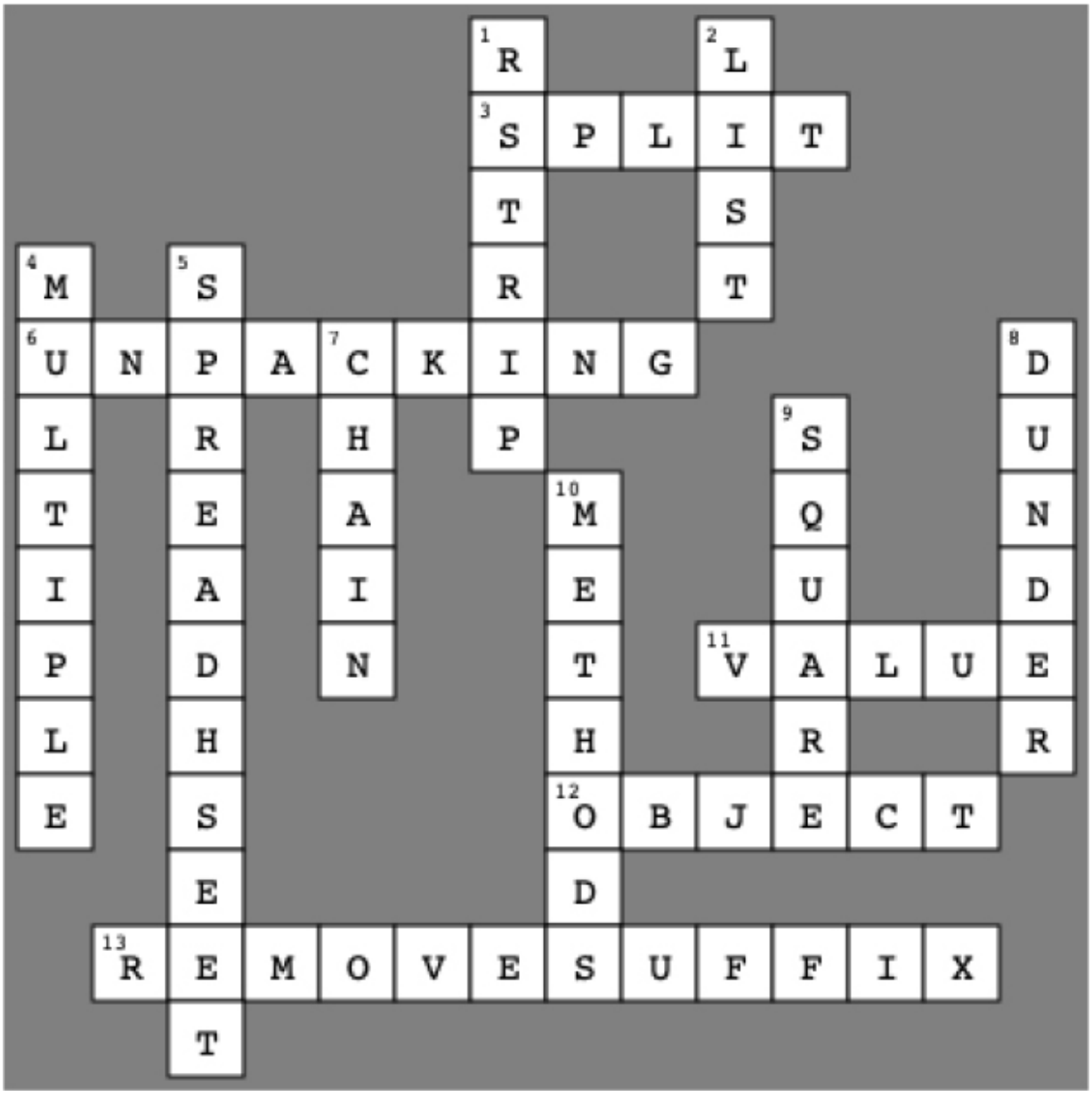
7. Ball and _____.

8. Short for "double underscore".

9. Our favorite brackets.

10. The plural name given to an object's built-in functions.

The String Crossword Solution


1 R
2 L
3 S P L I T
T S
4 M
5 S
R T
6 U N P A
7 C K I N G
8 D
L R H P
9 S U
T E A
10 M Q N
I A I E U D
P D N T
11 V A L U E
L H H R R
E S
12 O B J E C T
E D
13 R E M O V E S U F F I X
T

Across

3. Comes in handy when breaking apart strings.

6. Another name for multiple assignment.

11. To be a variable, it has to have one of these.

12. Everything is one of these.

13. Can be used to get rid of a filename's extension.

Down

1. Removes a set of characters from the end of a string.

2. Like an array in other languages.

4. More than one.

5. The swim coach mucks about with one of these.

7. Ball and _____.

8. Short for "double underscore".

9. Our favorite brackets.

10. The plural name given to an object's built-in functions.

Chapter 2. Lists of Numbers: *Processing List Data*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 2 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



The more code you write, the better you get. It's that simple. In this chapter, you continue to create Python code to help the Coach. You learn how to **read** data from a Coach-supplied data **file**, slurping its lines into a **list**, one of Python's most-powerful built-in **data structures**. As well as creating lists from the file's data, you'll also learn how to create lists from scratch, **growing** your list **dynamically** as needs be. And you'll process lists using one of Python's most popular

looping constructs: the **for** loop. You'll **convert** values from one data format to another, and you'll even make a new best friend (your very own Python **BFF**). You've had your fill of coffee and pie, so it's time to roll up your sleeves and get back to work.

Task #2: Process the data in the file

With Task #1 complete, it's time to move onto Task #2. There's a bit of work to do but, as with the previous chapter's activities, you can approach things bit-by-bit as detailed in the last chapter:

1. Read the lines from the file
2. Ignore the second line
3. Break the first line apart by "," to produce a list of times
4. Take each of the times and convert them to a number from the "mins:secs.hundredths" format
5. Calculate the average time, then convert it back to the "mins:secs.hundredths" format (for display purposes)
6. Display the variables from Task #1, then the list of times and the calculated average from Task #2



It's time to get to work.

You won't be done by the time the Coach is finished his warm-up, but you'll definitely make progress by the end of today's swim session.

Let's get started by grabbing a copy of this chapter's data before learning how Python reads data from a file.

Grab a copy of the Coach's data

There's no point learning how to read data from a file if you have no data to work with. So, head on over to this book's support website

and grab the latest copy of the Coach's data files. There are 61 individual data files packaged as a ZIP archive. Grab a copy from here::

https://python.itcarlow.ie/ed3/Learning/swimdata.zip

Don't worry, those 61 data files are small so it only takes a few seconds for the ZIP to download.

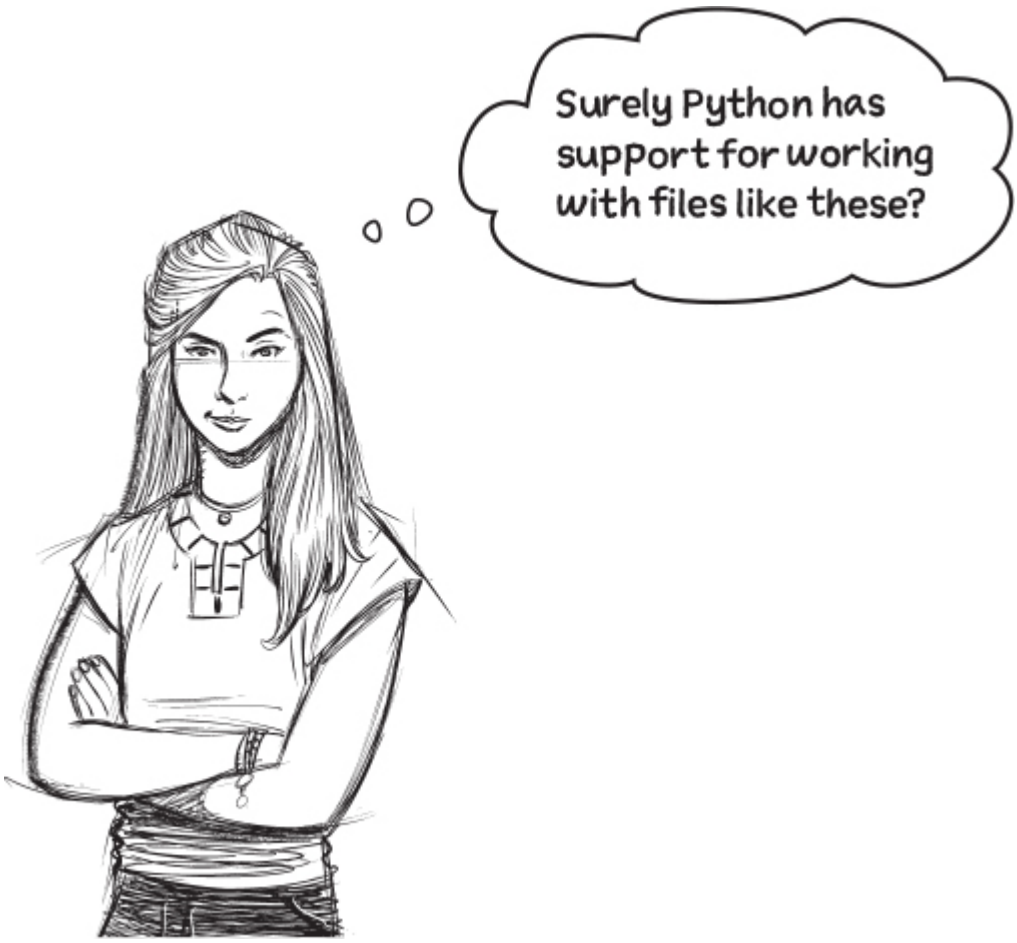
Once your download completes, unzip the file then copy the resulting swimdata folder into your `Learning` folder. This ensures the code which follows can find the data as it'll be in a known place.

Each file in the `swimdata` folder contains the recorded times for one swimmer's attempts at a specific youth distance/stroke pairing. Recall the data file from the start of the previous chapter which shows Darius's under-13 times for the 100m fly:

This is the data you're interested in.

Darius-13-100m-Fly.txt

```
1  1:27.95,1:21.07,1:30.96,1:23.22,1:27.95,1:28.30
2
```



Yes, it does.

There's a BIF called **open** which can work with files, opening them for reading, writing, appending, or any combination of the above.

The **open** BIF is powerful on it's own, but it shines when combined with Python's… em, eh… **with** statement.

The open BIF works with files

Whether your file contains textual or binary data, the **open** BIF can open the file to read, write, or append data to/from it. By default, **open** reads from a text file, which is perfect as that's what you want to do with the Darius's data file.



You can call **open** directly in your code, opening a named file, processing its data, then closing the file when you're done. This open-process-close pattern is very common, regardless of the programming language you use. In fact, Python has a language statement which makes working with the open-process-close pattern especially convenient: the **with** statement.

Although there's a bit more to the **with** statement than initially meets the eye, there's only one thing that you need to know about it right now: If you open your file with **with**, Python arranges to *automatically* close your file when you're done, regardless of what happens during whatever processing you perform on the file.



NOTE

As always, follow along.

TEST DRIVE

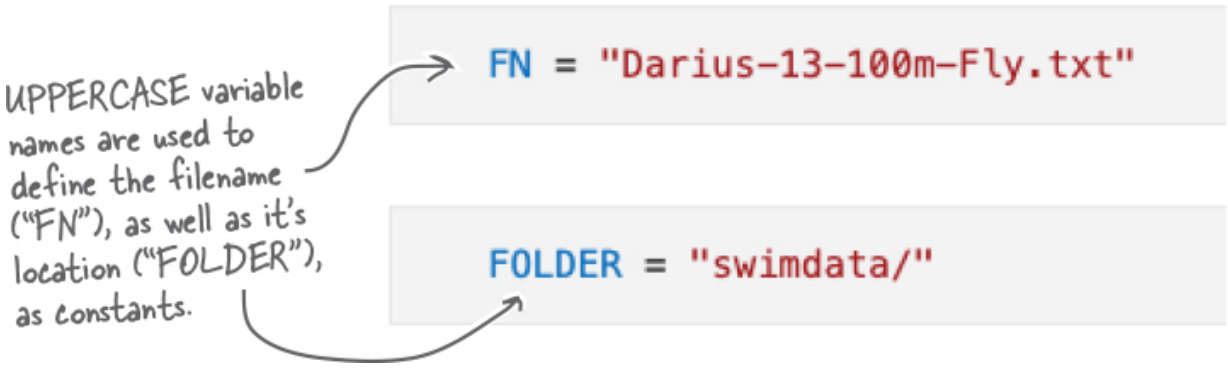


Let's see the **with** statement together with the **open** BIF at work, so you can get in on the action.

For the code that follows to work, the assumption is you've already downloaded the Coach's data, unzipping the file into a folder called `swimdata` within your `Learning` folder. Do that now if you forgot (and skip back two pages for the URL to use).

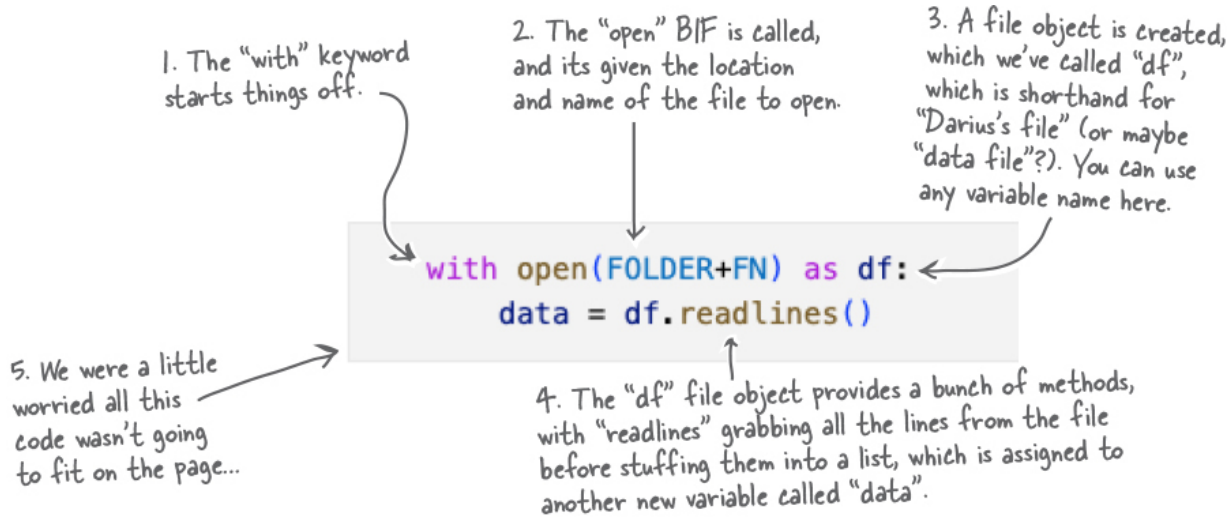
To get going, create another new notebook in VS Code, and give it the name `Average.ipynb`, saving this latest notebook in your `Learning` folder.

To identify the file you plan to work with you need two things: the file's name and the location it's to be found in. Here's how Python programmers would define constants for these values:



Although referred to as "constants", Python doesn't support the notion of constant values, so it's a convention within the Python programming community to use UPPERCASE variable names to signal to other programmers that the values are constant (and should not be changed). And, yes, eagle-eyed readers will have spotted that we – rather blatantly – disregarded (!!) this convention in the previous chapter with "`fn`". This is a *shocking* use of a lowercase variable name when an uppercase constant name should've been used. We got away with this because Python doesn't enforce this naming convention, as it's not a language rule. That said, from now on, we'll be sure to follow it.

With your constants defined, here's the Python code which opens the file, reads all its data into a list called (exploiting unprecedented imagination here) `data`, then automatically closes the file:



Not much code, but there's lots happening...

The code is not very long, but – as the annotations at the bottom of the last page indicate – there's a lot going on:

```
with open(FOLDER+FN) as df:
    data = df.readlines()
```

This code's a thing of beauty, isn't it?

Let's highlight three important take-aways:

1. **The with statement opens the file** *before* **its code block runs.**

You may well be asking “Which code block?”, and you’d be right to. We haven’t told you yet, but the **with** statement’s code block is all the code indented under it. In this case, the code block is only one line long and that’s OK (code blocks can be of any length).

NOTE

If you are coming to Python from one of those programming languages which uses curly-braces to delimit blocks of code, using indentation in this way may unnerve you. Don’t let it, as it’s really not that big a deal.



It not that we don’t want to talk about indentation.

It's just we feel there's much more to Python than its use of indentation (or, more correctly, *whitespace*) to delimit code blocks. Yes, it's an important aspect of the language, but it's something most Python newbies get used to quickly. When we need to, we'll call it out, otherwise we'll just get on with things. And with that said, let's get back to the take-aways.

2 **The with statement closes the file *after* its code block runs.**

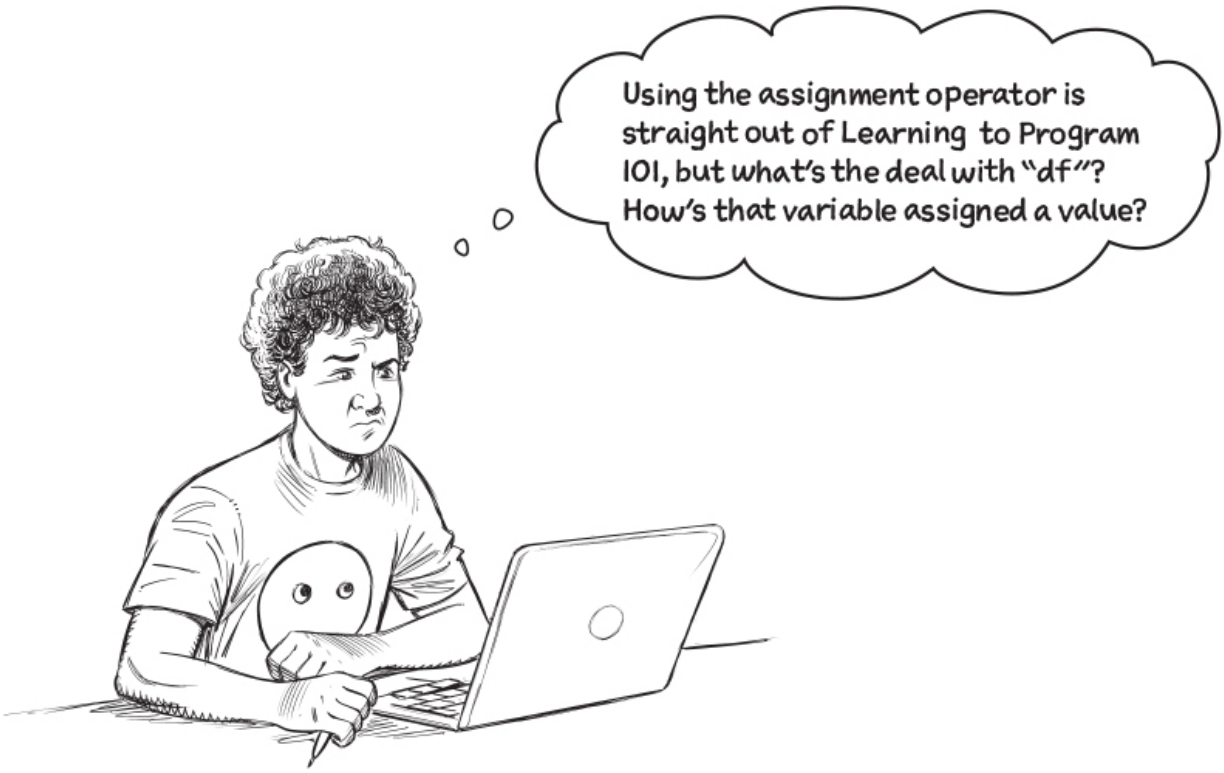
This is a cool feature, as we'd forgotten to do this. It's nice to know the **with** statement has your back, tidying up after your code block executes.

3 **Two variables are created by the code: `df` and `data`.**

The `df` variable refers to a *file object* created by the successful execution of the **open** BIF. The `data` variable refers to the list of lines read from the `df` file object by the **readlines** method. Both variables continue to exist after the code block ends, although the `df` variable now refers to a *closed* file object.

Variables are created dynamically, as needed

The `df` and `data` variables were created as a result of assignment. Although it's easy to see how `data` came into being, thanks to the use of the assignment operator (`=`), it's less clear what's going on with `df` .



The key word is "as".

Thanks to that **with**, the **as** keyword takes the **open** BIF's return value and assigns it to the identified variable name, which is `df` in your code. It's as if this code ran:

```
df = open(FOLDER+FN)
```

The **as** keyword, together with **with**, does the same thing (and looks nicer, too).

Let's take a closer look at what `df` is, as well as learn a bit about what it can do:



It's not that file objects aren't exciting...

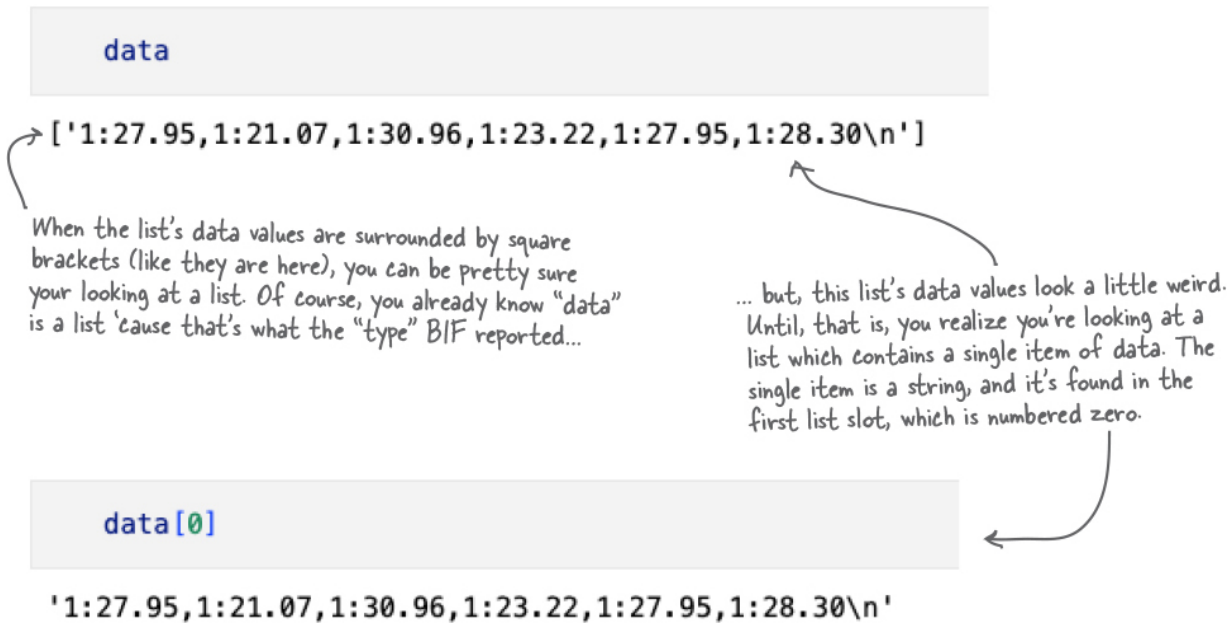
It's just, in this case, the file object is merely a means to an end: loading the file's lines into the `data` variable. So, what's `data` and what can you do with it?



You already know (from the previous chapter) that Python lists understand the square bracket notation. Before you get to that, let's take a look at what `data` contains:

NOTE

You may have thought about executing the combo mambo on "data" here, but you don't need to just yet. For now, all you need is the square bracket notation.



```
type(data[0])
```

```
str
```

Confirmation that the data value in the first slot of the "data" list is a string.

Don't forget to press Shift+Enter to execute code cells.



Yes, with some help from "with".

Despite being a one-line code block, a lot's happening here.

Not only has your list been created, assigned to your `data` variable *and* populated with the data contained within the swimmer's file, but that `with` statement has managed to complete the first two sub-tasks for Task #2. How *cool* is that?

Take a look (on the next page).

Work has started on Task #2

Those two lines of code pack a punch. Here they are again:

```
with open(FOLDER+FN) as df:
    data = df.readlines()
```

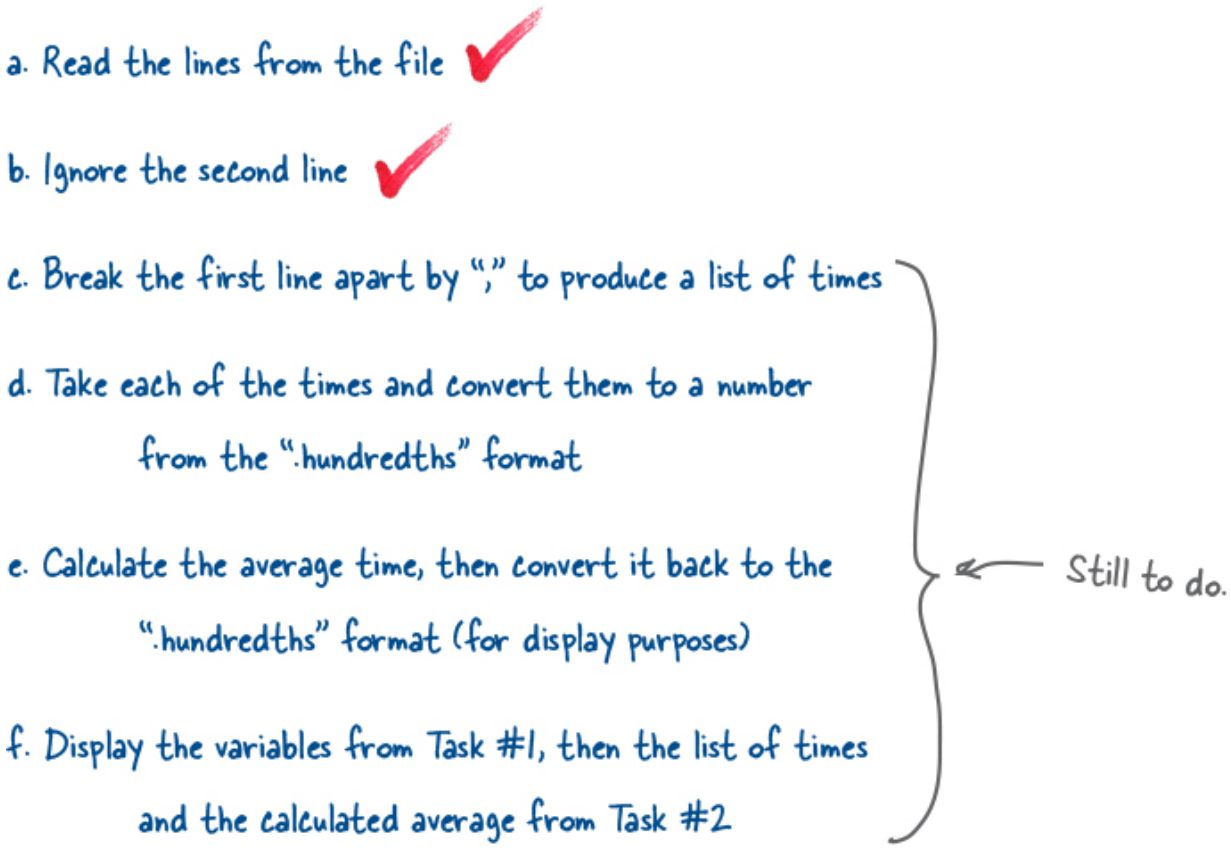
← The code.

The data value in the first slot in the `data` list is a string representing the swimmer's times:

```
'1:27.95,1:21.07,1:30.96,1:23.22,1:27.95,1:28.30\n'
```

← The data.

You can safely ignore anything else in the file, as the data you need is in the above string. It's time for a couple checkmarks to indicate your progress with Task #2:



The third sub-task should not be hard for anyone who has spent any amount of time working with Python's string technology. As luck would have it, you've just worked through the string material in the previous chapter, so you're all set to have a go. But before you get to that sub-task, we need to talk a little about one specific part of that **with** statement: the **colon**.

Your new best friend, Python's colon

The colon (`:`) indicates a code block is about to *begin*.

Unlike a lot of other programming languages, Python does not use curly braces (`{` and `}`) to delimit blocks. Instead Python uses **indentation** (or, to be more accurate, *whitespace*). In fact, in Python, curly braces delimited data, *not* code.

A code block in Python ends when the indentation ends.

In your **with** statement, the block contains only one line of code, but it could potentially contain any number of lines of code. Code indented to the same level as the immediately preceding line of code belongs to the *same* block.

The use of the colon is critical here (which is why it's your new best friend). Like in real life, if you forget your best friend, bad things happen. If you forget the colon at the end of that line, Python refuses to run your code!

Think of the colon and indentation as *going together*: you can't have one without the other.

> **FYI: the Python docs refer to a "code block" as a "suite".**

NOTE

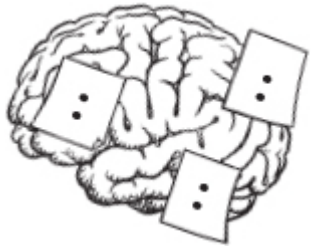
We think this is weird, too.

Your new BFF.



We think the colon is so important that we're giving the colon its own half-page. A common slip-up is to forgot to add it to the end of the line when introducing a new indented block. Python complains *LOUDLY* when you do, as it's never (ever!) nice to forget your BFF.

MAKE IT STICK



Python's use of indented whitespace is an important visual clue that a new block of code is starting. However, indentation without the colon confuses Python, so the colon is required at the end of the line of code immediately preceding the block.

Think of the use of the colon as your way of confirming to Python that the next line is the start of a new block, which is also indented (of course).

THERE ARE NO DUMB QUESTIONS

Q: When it comes to blocks of code within blocks of code, do I just need to increase the level of indentation?

A: Yes. If you are used to using curly braces to indicate the start and end of a block of code, you have likely had a need to embed a block of code within another block of code, ending up with curly braces *inside* curly braces. It's the same thing with Python, except the level of indentation increases. Visually, it is easy to work out where the block of code starts and ends. It starts on the line right after the colon, and it ends with the block's indentation ends. For an example, recall the short example of a nested **for** loop from Chapter 0: Why Python? Similar But Different.

Q: Is there a standard indentation spacing value? Do most people use four spaces to indicate indentation, or can two spaces be used? Does it matter?

A: To be honest, it doesn't really matter whether you use four spaces, two spaces, or any number of spaces to indicate indentation. The tab character can also be used. What does matter is that whatever you do, your usage needs to be **consistent**. If you use four spaces on one line of code, you can't use two spaces on the next, nor can you mix'n'match spaces with the tab character. The Python parser gets confused when you do mix'n'match and raises an **IndentationError** or **TabError**. Note: if you *consistently* use four spaces, two spaces, or

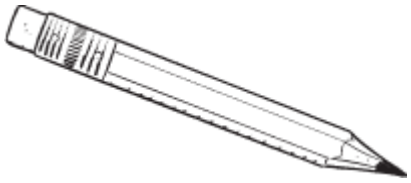
the tab character to indicate your level of indentation, you'll never see either of those errors.

Now, having said all that, there is a *convention* in the Python programming community which strongly suggests using four spaces consistently to indent your code. Many Python programmers configure their text editors to automatically replace a press of the tab key with four spaces. Also, be warned: if you are sharing code with other Python programmers and you haven't used four spaces to consistently indent your code, you better be ready to explain why.

Consistency is good.

You may not have noticed, but VS Code consistently uses four spaces to indent your Python code, automatically indenting to the correct level whenever you indicate a new block is about to begin by correctly putting a colon at the end of the line.

SHARPEN YOUR PENCIL



In the previous chapter you took a string then applied the **split** and **removesuffix** methods to it to produce the data values you needed from the file's name.

A similar strategy can be applied to your next sub-task, although you are unlikely to need to use **removesuffix**. The string you're working with has a newline character (`\n`) at the end you don't need. Find a string method to use in place of **removesuffix** to enable you to remove the newline character from the string. Combine the call to the new method in a chain which includes **split** to break the string apart by "," producing a new list, which you can assign to a new variable called `times` .

Experiment in your VS Code-hosted notebook until you've written the code you need, then write the code which create the `times` variable in the space provided below (and our code is on the next page):

_____

_____



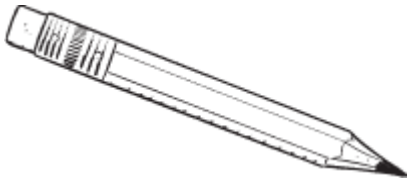
Yes, to both questions.

Yes, we did indeed introduce strings in the previous chapter and, yes, we're concentrating on lists in this one.

Recall the **split** method produces a list from a string, which is precisely why you need to use it now. If your `times` variable, above, isn't a list, you're likely doing something wrong.

When you're ready, flip the page to see the code we came up with.

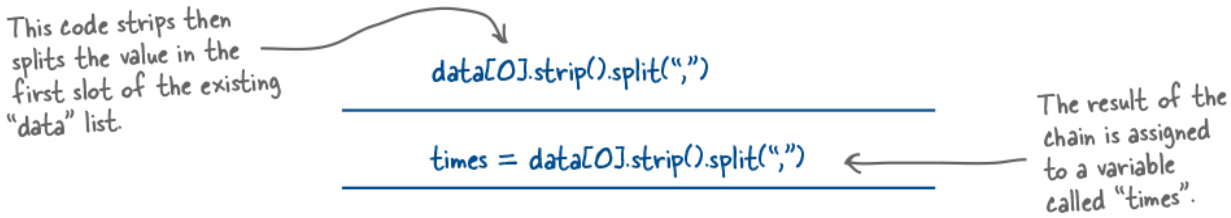
SHARPEN YOUR PENCIL SOLUTION

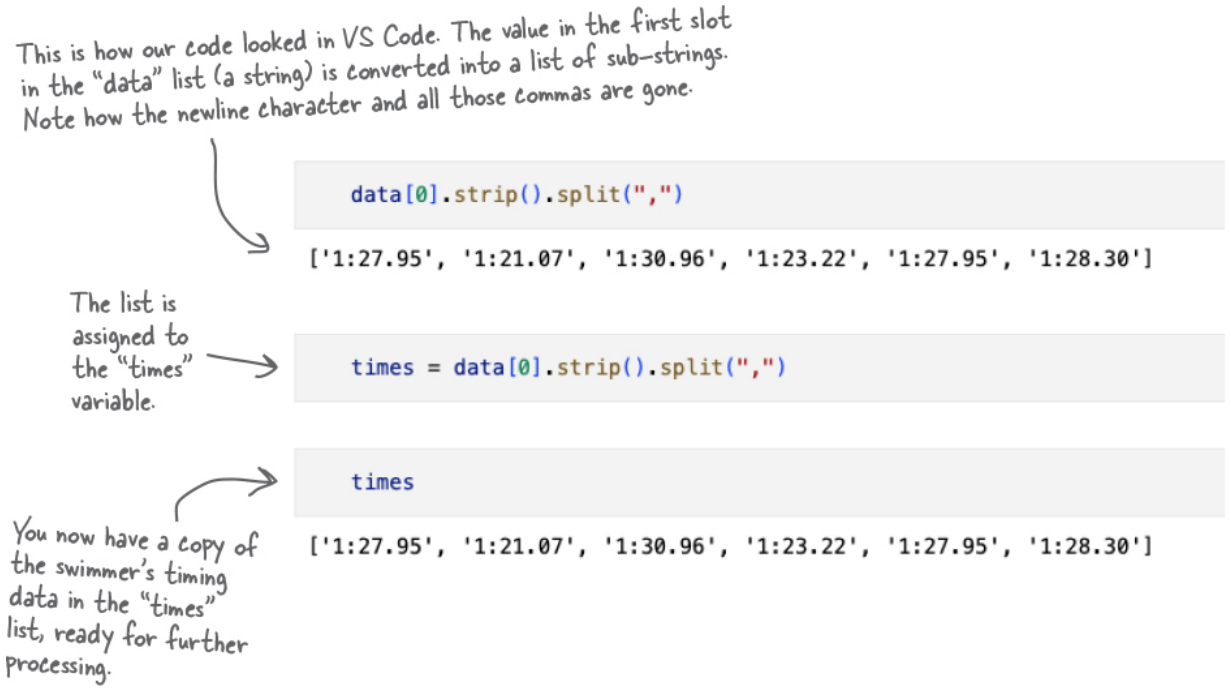


In the previous chapter you took a string then applied the **split** and **removesuffix** methods to it to produce the data values you needed from the file's name.

A similar strategy can be applied to your next sub-task, although you are unlikely to need to use **removesuffix**. The string you're working with has a newline character (`\n`) at the end you don't need. Knowing this, you were asked to find a string method to use in place of **removesuffix** to enable you to remove the newline character from the string. You were to combine the call to the new method in a chain which was to include **split** to break the string apart by "," producing a new list to be assigned to a new variable called `times` .

It was suggested you experiment in your VS Code-hosted notebook until you've written the code you need, then you were to write the code to create the `times` variable in the space provided below. Here's what we came up with:



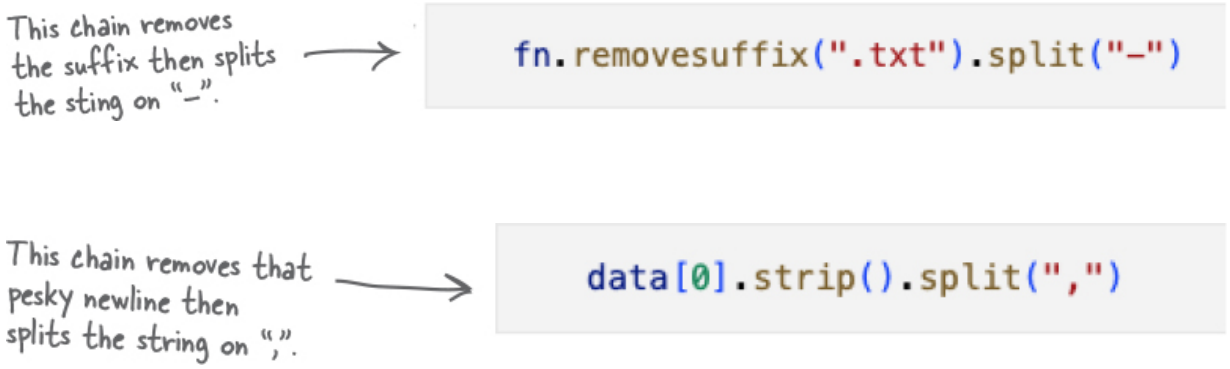


That was almost too easy

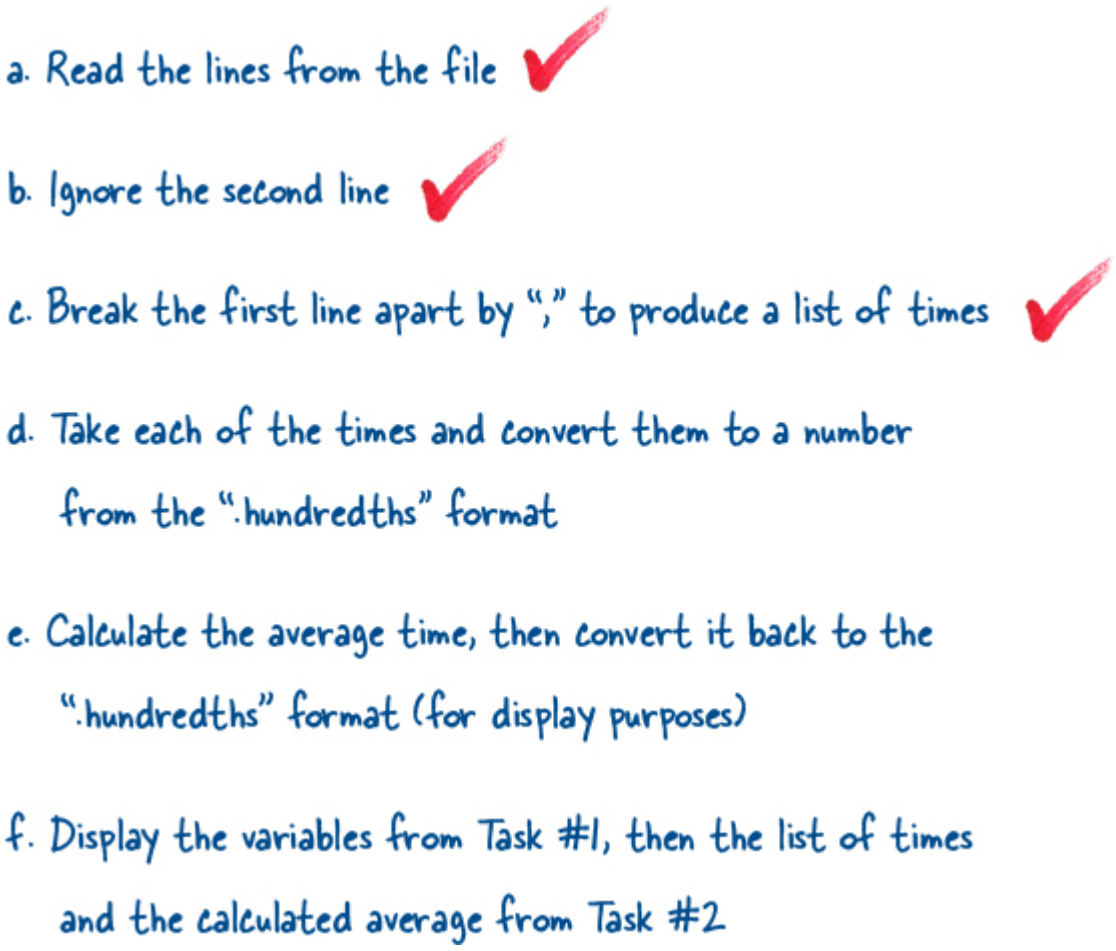
With your prior experience of working with strings from the previous chapter, we're hoping that most recent *Sharpen* wasn't too taxing.

It is important to call **strip** *before* **split**, producing a new list from the data value in the `data`'s first slot (`data[0]`). In fact, your latest

chain code is very similar to the code from the previous chapter:

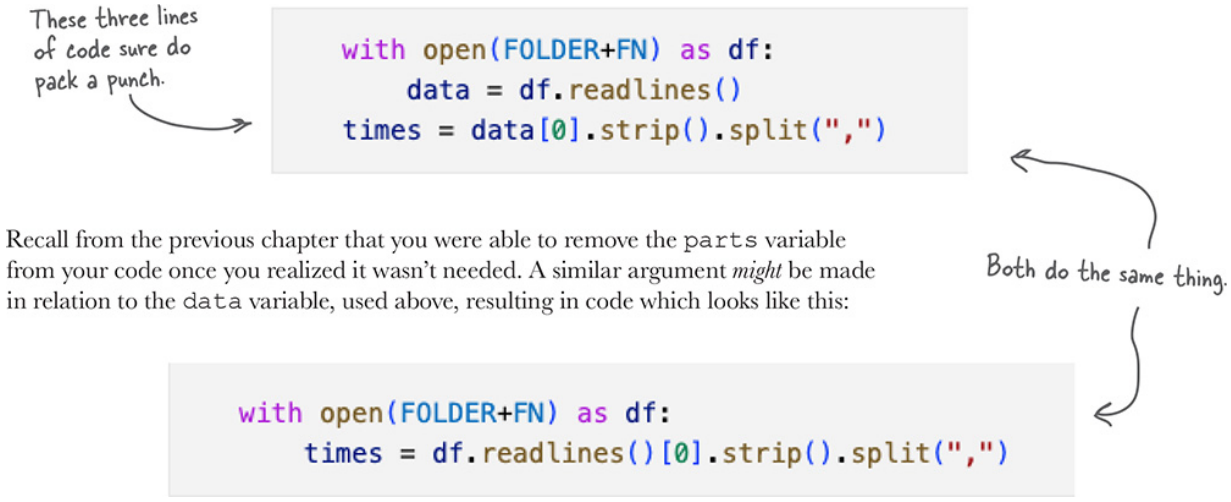


With the result of your latest chain assigned to the `times` variable, you've completed sub-task (c). It's time for another checkmark.



Pause to review this task's code

Here's how you can combine the code so far in a single code cell within VS Code:



Recall from the previous chapter that you were able to remove the `parts` variable from your code once you realized it wasn't needed. A similar argument *might* be made in relation to the `data` variable, used above, resulting in code which looks like this:

```
with open(FOLDER+FN) as df:
    times = df.readlines()[0].strip().split(",")
```

If you go ahead and try both of these `with` statements in your notebook you'll learn that both populate the list `times` refers to with the same collection of strings. So, why not use the two-line version of the code as opposed to the three-line version? After all, just like with the `parts` list in the previous chapter, the `data` list is no longer needed once it's been used that one time...



No, it's not hard to read. It's a nightmare.

Three methods are chained here, with the first one creating a list, from which you take the first slot's data (using the square bracket notation), then you strip it before splitting it... but, what does "it" refer to again?!?

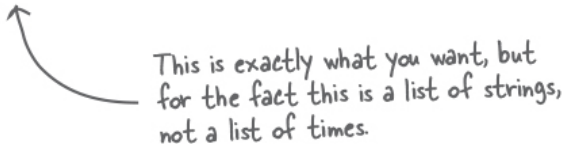
This single line of code is hard to read, understand, explain, *and* maintain. We pity the poor programmer asked to "fix" this code at some point in the future (who, most likely, will be *you*).

Converting a time string into a time value

After the code from the previous page runs, the `times` variable refers to a list of strings:

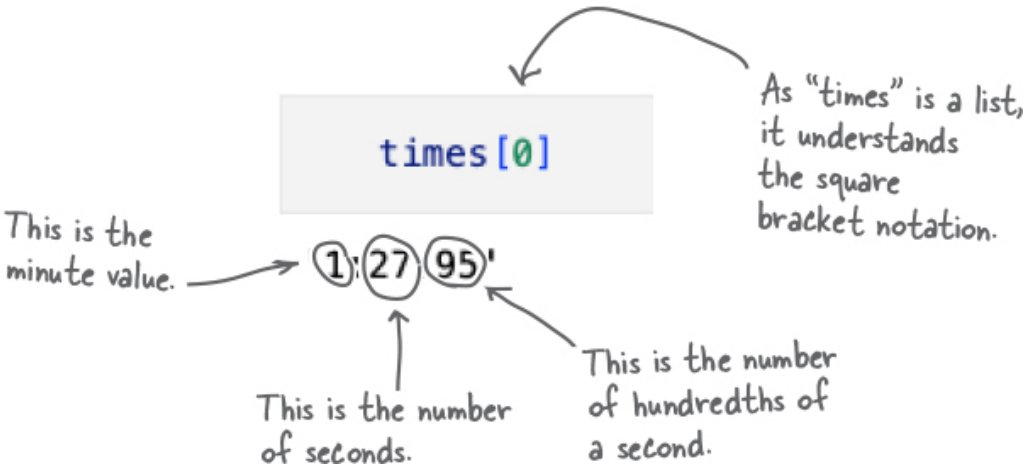
```
times
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```



The values in each of the slots in the `times` list certainly look like swim times, but they are not. They are strings. To perform any numeric calculation on this list, such as working out an average, these strings need to be converted into numeric values.

Let's take a closer look at just on value (the first). If you can come up with a strategy for converting this first time, you can then apply it to the rest of the list.



BRAIN POWER



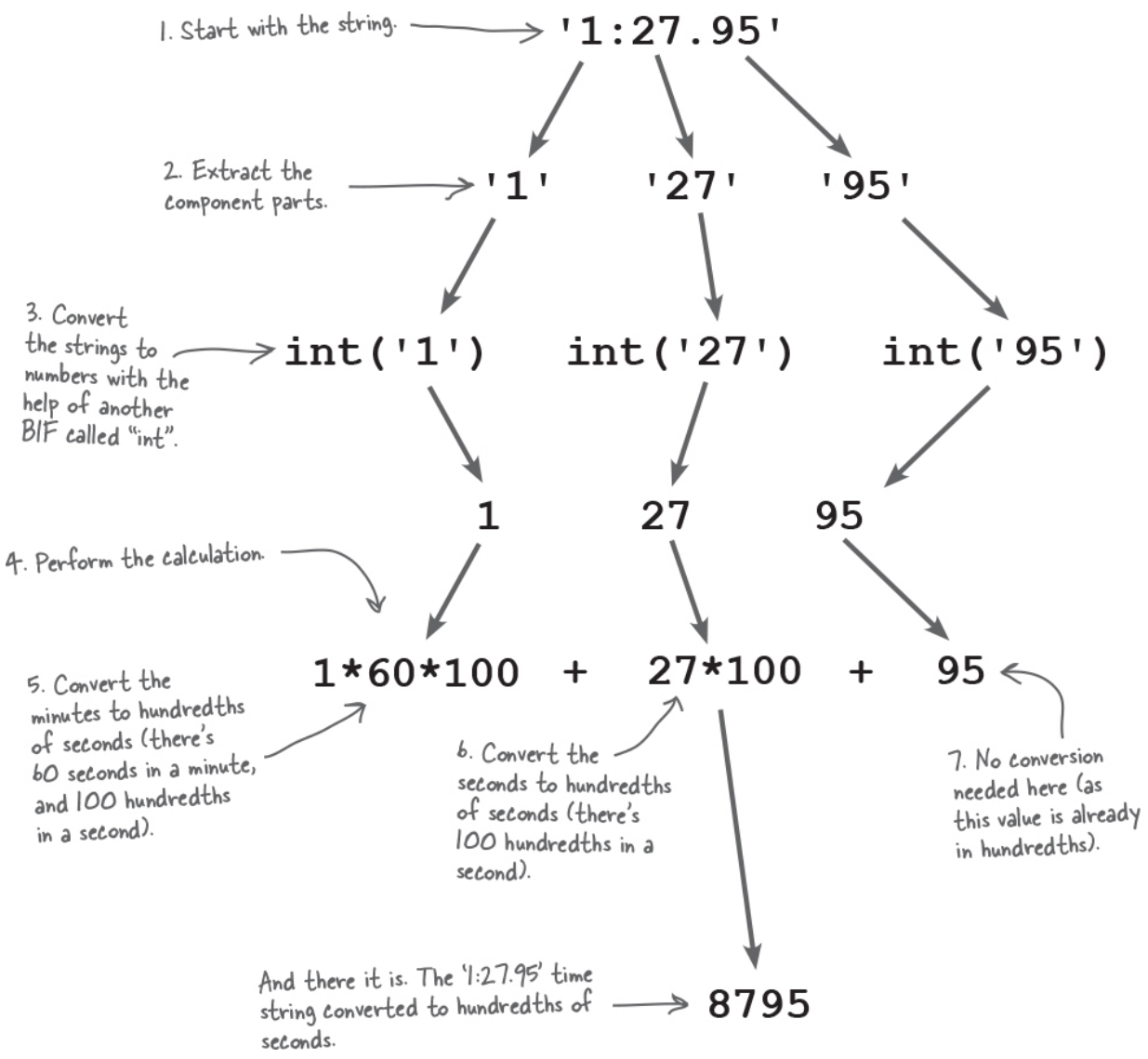
Assuming you can extract the three numbers you need from the string, can you think of a calculation which converts the string into a numeric value?

NOTE

There's more than one way to do this, so don't worry if what you think up isn't the same method as ours (which is detailed over the page).

Convert the times to hundredths of seconds

At the moment, all the swim times are *strings* even though our brains see them as *times*. Not so for our computers, though (let alone Python). Our digital buddies need a little help with the conversion, and here's how we'd suggest it can be done:



Turning this conversion strategy into Python code is *remarkably* straightforward. Let's take a look.

To hundredths of seconds with Python

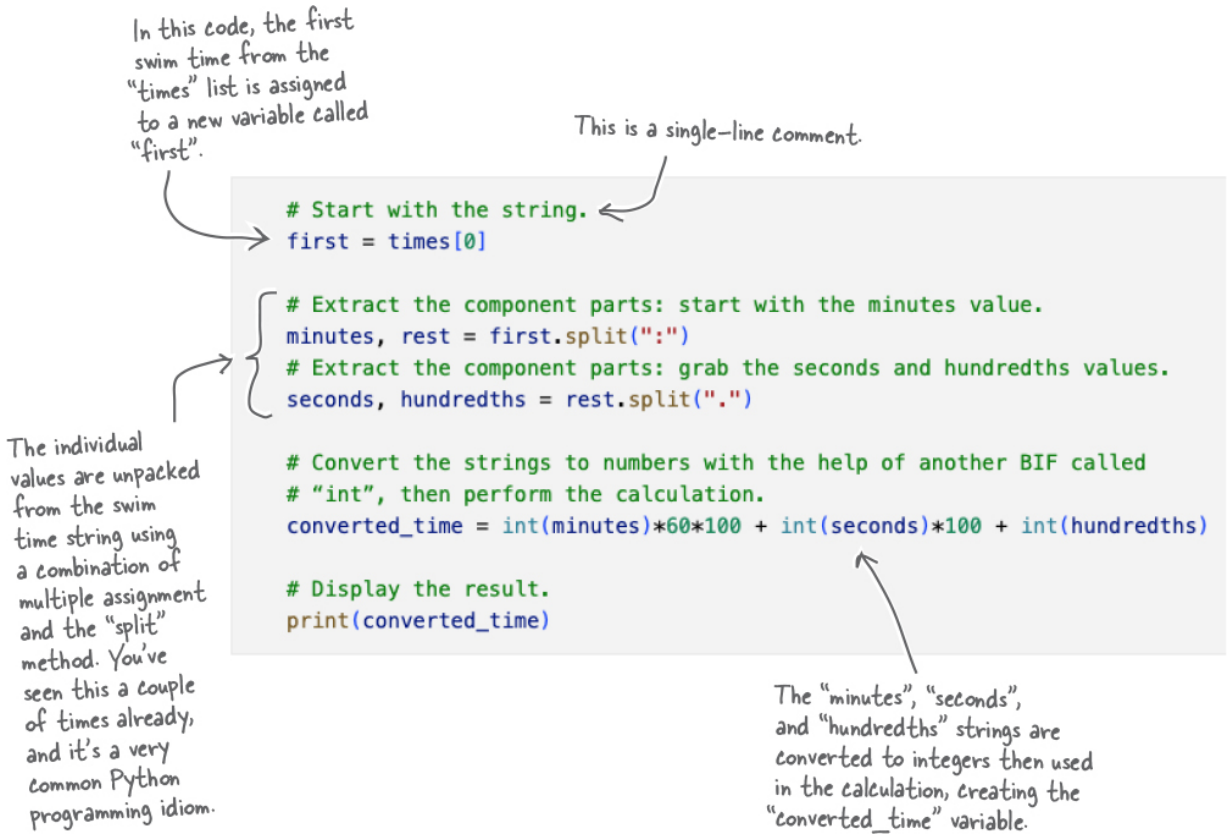
We went all out with that visual on the previous page, and converting the visual's steps to Python code is a near one-for-one match. The code shown below performs the calculation for the first swim time taken from your `times` list, with the code typed into a single cell in the `Average.ipynb` notebook.



To help keep you straight, we've added some comments to this code. When Python sees the `#` character, it *ignores* everything which follows the `#` until the end of the current line. (Note how VS Code helpfully displays the comments in **green**).

Type this code into your notebook, and don't feel guilty if you decide to exclude the comments (don't worry, we won't tell). We put them in

to match up the code with the conversion steps from the preview page's visual.



With this code typed into an empty cell in your notebook, press **Shift+Enter** to run it. The value `8795` appears on screen. Sweet.



If you can convert one swim time...

You can convert them all. And, there's no extra credit for guessing you need a loop here.

Like most programming languages, Python provides many ways to loop, with the **for** loop being a favorite. Let's look at a simple loop which takes each of the strings from the `times` list and displays them on screen.

We don't know why, but programmers love to use single-letter variable names as their loop variables, and Python programmers are no exception. Not wishing to rock the boat, we're using "t" as our loop variable here. Every time the loop runs, "t" is assigned a values from the "times" list. Think of "t" as containing the current swim time string.

"for" is a Python keyword.

```
for t in times:
    print(t)
```

Another important keyword is "in", which we'll have more to say about in a later chapter. Of course, you already know how important that colon is (being your new BFF and all).

And here they are. All of the swim times from the "times" list displayed on screen.

```
1:27.95
1:21.07
1:30.96
1:23.22
1:27.95
1:28.30
```



Cool, isn't it?

The **for** loop is smart enough to know all about the length of the list it is processing.

There's always a temptation to use the **len** BIF to work out how big your list is before it's looped over, but with **for** this is an unnecessary

step. The **for** loop starts with the first value in the list, takes each value in order, processes the value, then moves onto the next. When the list is exhausted, the **for** loop terminates.

This is the sort of magic we love.

THERE ARE NO DUMB QUESTIONS

Q:Didn't we see the "for" loop in Chapter 0: Why Python? Similar But Different?

A: Yes, but unlike in that chapter where you mucked about with playing cards to get a feel for how things worked, in this chapter you're doing real work in support of the system you're building for the Coach.

Q: I'm just wondering if there's anything in the PSL that might help with converting these time strings?

A: Good question, and the answer is: *maybe*. The PSL (the *Python Standard Library*) includes two modules which might help: `time` and `datetime`. To be honest, our eyes started to glaze over while scanning through the copious amount of documentation included with both modules (especially with `datetime`). Although there might be sokme functionality in either module which might help, we felt that the custom conversion required for the Coach's timing data is the right fit for now. We reserve the right, of course, to change our mind on this should our date manipulations become a thorny mess sometime down the road...

Q: I take it Python supports other operators over and above "`+`"?

A: It sure does. All the usuals are there: `+`, `-`, `*`, `/`, and so on. There are a couple extras which are Python-specific, for instance `//` and `:=`, and (should you need them) we'll be sure to tell you all you need to know.

EXERCISE



Now that you've seen the **for** loop in action, take a moment to experiment in your notebook to combine the *Ready Bake Code* from a few pages back with a **for** loop in order to convert all of the swim times to hundredths of seconds, displaying the swim times and their converted values on screen as you go. When you are done, write the code you used into the space below. Our code is coming up in two pages time.

_____

_____

_____

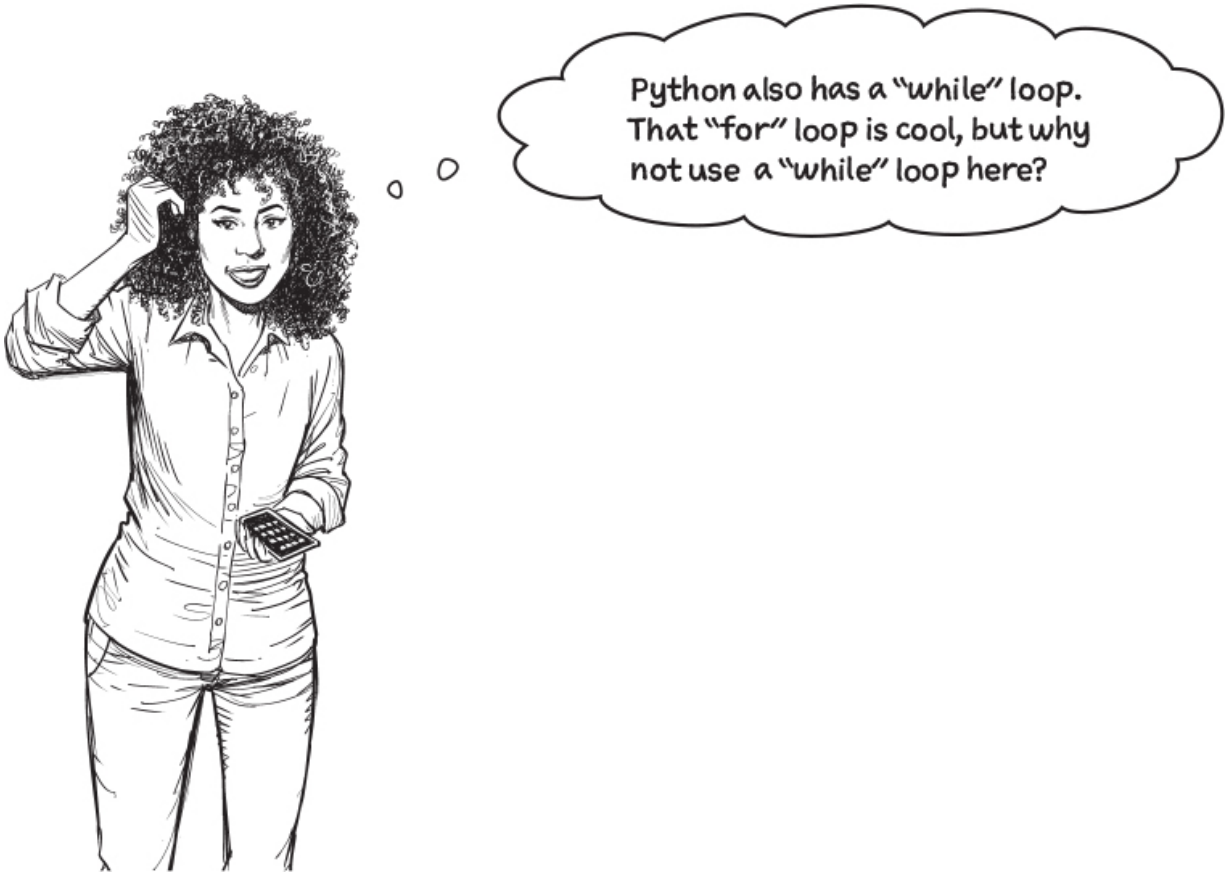
_____

_____

_____

_____

_____



Python does indeed support while.

But, the **while** loop in Python is used much less than an equivalent **for**.

Before getting to our solution code for the above exercise, let's take a moment to compare **for** loops against **while** loops.

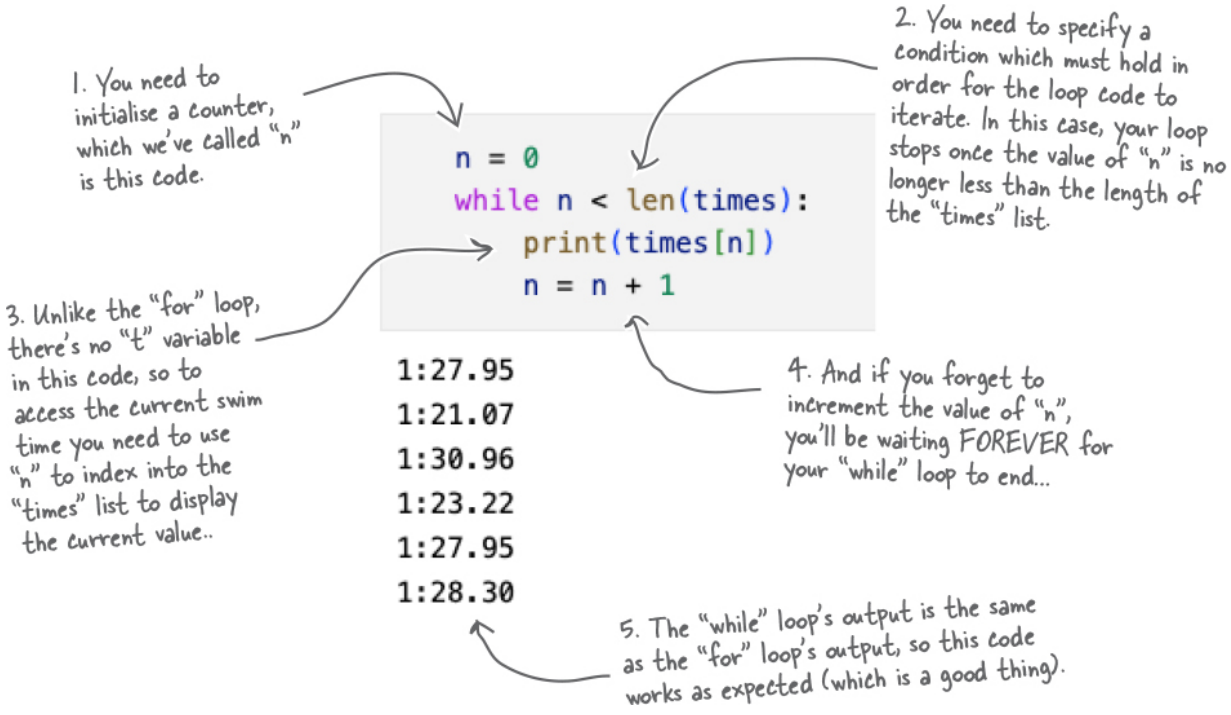
The gloves are off... for loops vs. while loops

Here's the **for** loop from earlier, together with its output:

```
for t in times:
    print(t)
```

```
1:27.95
1:21.07
1:30.96
1:23.22
1:27.95
1:28.30
```

And here's an equivalent **while** loop which does exactly the same thing:



Not only is the **while** loop's code twice the number of lines as the **for** loop, but look at all the extra stuff you have to concern yourself with! There's so many places where the **while** loop can go wrong, unlike the **for** loop. It's not that **while** loops shouldn't be used, just remember to reach for the **for** loop *first* in most cases.

EXERCISE SOLUTION



Now that you've seen the **for** loop in action, you were to take a moment to experiment in your notebook to combine the *Ready Bake Code* from a few pages back with a **for** loop in order to convert all of the swim times to hundredths of seconds, displaying the swim times and their converted values on screen. You were to write the code you used into the space below. Here's the code we came up with:

Loop over all the swim times. →

```
for t in times:
```

Extract the component parts. →

```
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
```

Display the calculated output. →

```
    print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

TEST DRIVE



Taking the *Exercise Solution* code for a spin produces the expected output:

```
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

```
1:27.95 -> 8795
1:21.07 -> 8107
1:30.96 -> 9096
1:23.22 -> 8322
1:27.95 -> 8795
1:28.30 -> 8830
```

Whoo hoo! Looks good.

There's two things of note here. First off, the "print" BIF takes any number of objects, separated by commas, to display on screen (three in this case). Secondly, our solution code dispenses with the "converted_time" variable from the Ready Bake Code as it's not needed here. All you're interested in is the result of the calculation which is performed first, then fed to the "print" BIF for display.

You're motoring now!

You are now past the mid-point of your sub-tasks for Task #2:

a. Read the lines from the file ✓

b. Ignore the second line ✓

c. Break the first line apart by "," to produce a list of times ✓

d. Take each of the times and convert them to a number from the ".hundredths" format ✓

e. Calculate the average time, then convert it back to the ".hundredths" format (for display purposes)

f. Display the variables from Task #1, then the list of times and the calculated average from Task #2

With the first part of sub-task (e), you have choices.



Either approach works.

However, if you think the converted times might be needed later, perhaps creating a new list of converted times is the way to go...

What do you think?



Let's keep a copy of the conversions

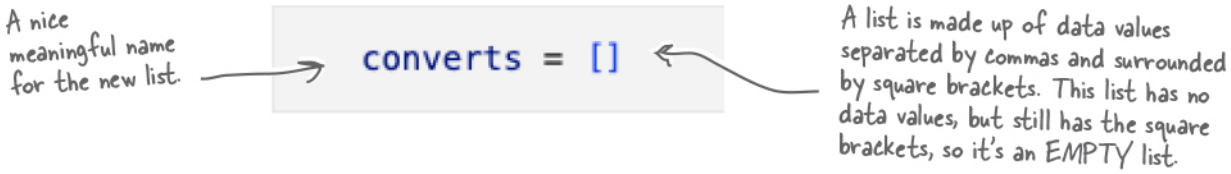
To be honest, *either* of the two approaches from the bottom of the last page would work for the first part of sub-task (e) of Task #2: *Calculate the average time*. However, we're guessing those converted values will be needed at least once more, so best if we put them in another list as we perform the conversions.

To do this, you need to learn a bit more about lists. Specifically, how to create a new, empty list, and how to incrementally add data values to your list as you iterate over the `times` list.

Creating a new, empty list

Step 1: think up a meaningful variable name for your list. Step 2: assign an empty list to your new variable name.

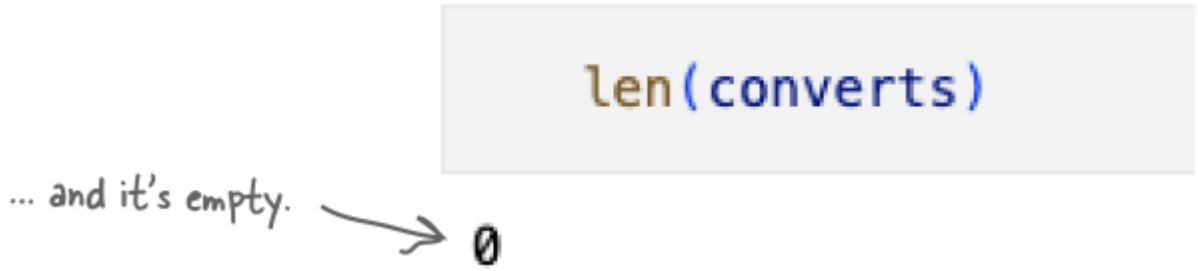
Let's call your new list `converts`. Here's how to perform Step 1 and 2 in a single line of code:



Recall that the `type` BIF is used to determine what *type* a variable refers to. A quick call to `type` confirms you're working with a list, and a call to the `len` BIF confirms your new list is *empty*:

```
type(converts)
```

It's a list... ⟶ `list`



Can you remember what you need to do to display your new list's built-in methods?

Displaying a list of your list's methods

It's combo mambo time!

As with any object in Python, the **print dir** combination lists the object's built-in attributes and methods. And as everything in Python is an object, lists are objects too!



Simply pass in your list's name.

```
print(dir(converts))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

As always, there's a big list shown here. And as before, and for now, ignore all those dunders.

The first non-dunder method name is **append**. You can likely guess what it does, but let's use the **help** BIF to confirm:

```
help(converts.append)
```

```
Help on built-in function append:

append(object, /) method of builtins.list instance
    Append object to the end of the list.
```

Oooooh. Interesting.

Ah ha!

That final line of output (*"Append object to the end of the list."*) is all you need to know, even though it's tempting to take some time to experiment with those other methods, some of which sound cool. But, let's not do that right now.

NOTE

Of course, if you feel the need to experiment with those other methods, don't let us stop you.

Let's stick to the task of building a new list of converted swim time values as you go.



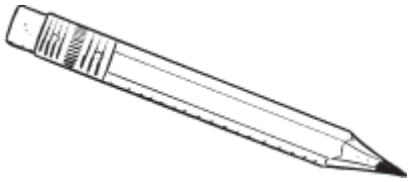
No, you do not need to worry.

In the previous chapter, we made a big deal about lists in Python being *like* arrays in other programming languages. This let us introduce the use of the square bracket notation with lists, which is a common technique when working with arrays *and* lists.

However, *unlike* with arrays, where you typically have to say how big your array is likely to get (e.g., 1000 slots) and what type of data it's going to contain (e.g., integers), there's no need to declare either of these with your Python lists.

Python lists are *dynamic*, which means they grow as needed (so there's no need to pre-declare the number of slots beforehand). And Python lists don't contain data values, they contain **object references**, so you can put any data of any type in a Python list. You can even mix'n'match types.

SHARPEN YOUR PENCIL



Grab your pencil, as you've work to do. Here's the most recent code which displays the swim time strings together with equivalent conversion to hundredths of seconds:

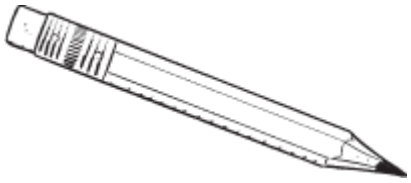
```
for t in times:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
    print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

Adjust the above code to do two things: (1) Create a new empty list called `converts` right before the loop starts, and (2) Replace the line that starts with a call to the **print** BIF with a line of code that adds the converted value onto the end of the `converts` list. Write your code in the space below (and, when you're ready, check your code against ours on the next page):



We're hoping, at this stage in this book, that you aren't scribbling down the first bit of code that pops into your head without first trying out said code in your notebook. The best way to get good at programming Python is to practice, and we think there's no better way to practice than to experiment with code in a Jupyter notebook. It is not cheating if you spend some time working out the code you need in VS Code before scribbling it into the space above. In fact, that's what we want you to do: experiment with your Python code within your Jupyter notebook.

SHARPEN YOUR PENCIL SOLUTION



You were to grab your pencil, as you'd work to do. You'd been shown the most recent code which displays the swim time strings together with equivalent conversion to hundredths of seconds:

```
for t in times:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
    print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

Your job was to adjust the above code to do two things: (1) Create a new empty list called converts right before the loop starts, and (2) Replace the line which starts with a call to the print BIF with a line of code which adds the converted value onto the end of the converts list.

Here's the code we came up with:

We created a new, empty list called "converts" by assigning an empty list to it. We do this outside the loop's code block as it only needs to happen once (obviously).

```
converts = []
for t in times:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
    converts.append(int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

As these three lines are indented under the "for" loop, they are part of the loop's code block, executing each time the loop runs.

Rather than printing the converted values, this code adds each converted time to the "converts" list (which dynamically grows as the loop runs).

TEST DRIVE



Let's take your latest code for a spin. Recall the previous version of your loop produced this output:

```
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

```
1:27.95 -> 8795
1:21.07 -> 8107
1:30.96 -> 9096
1:23.22 -> 8322
1:27.95 -> 8795
1:28.30 -> 8830
```

Your new loop code is similar, but does not produce any output. Instead, the `converts` list is populated with the conversion values. Below, the new loop code executes in a code cell (producing no output) then, in two subsequent code cells, the contents of the `times` list as well as the (new) `converts` list is shown:

```
converts = []
for t in times:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
    converts.append(int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

```
times
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

```
converts
```

```
[8795, 8107, 9096, 8322, 8795, 8830]
```

Here's the list of converted values, which are the hundredths of seconds equivalents of the swim time strings from the "times" list.

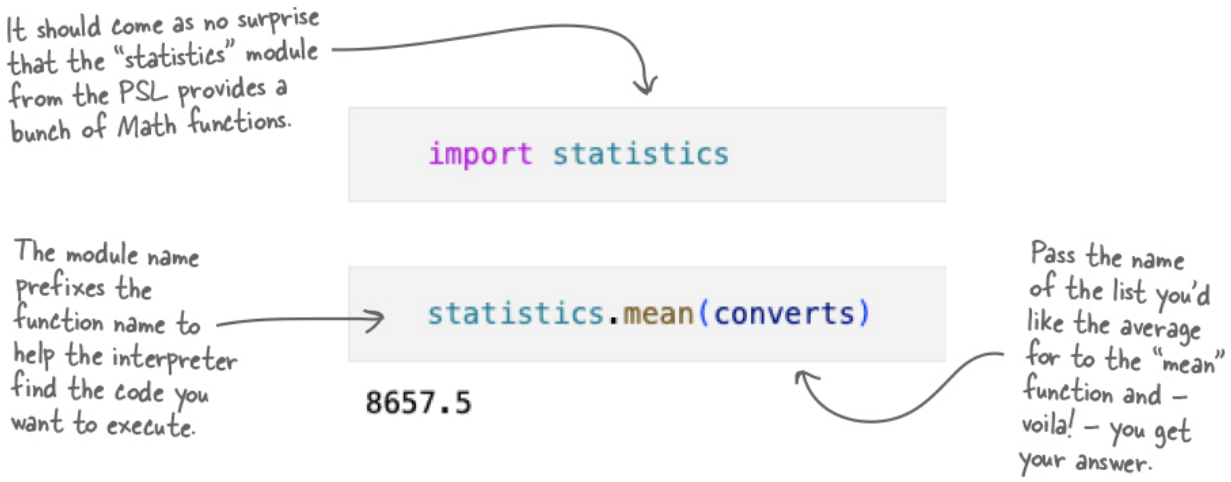
It's time to calculate the average

You don't need to be a programmer to know how to calculate an average when given a list of numbers. The code is not difficult, but this fact alone does not justify your decision to actually write it. When you happen upon a coding need which feels like someone else may have already coded it, ask yourself this question: *I wonder if there's anything in the Python Standard Library which might help?*

There is no shame in reusing existing code, even for something you consider *simple*. With that in mind, here's how to calculate the aver-

age from the `converts` list with some help from the PSL:

> **Hey, remember that handy PSL? No, not the delicious seasonal latte, the other PSL!**



Although calculating the average is easy, as shown above you haven't had to write a loop, maintain a count, keep a running total, nor perform the average calculation. All you do is pass the name of the list of numbers into the **mean** function which returns the arithmetic mean (i.e., the average) of your data. Cool. That'll do.



Yes, as mins:secs.hundredths.

In effect, you need to reverse the process from earlier which converted the original swim time string into it's numeric equivalent.

It can't be that hard, can it?

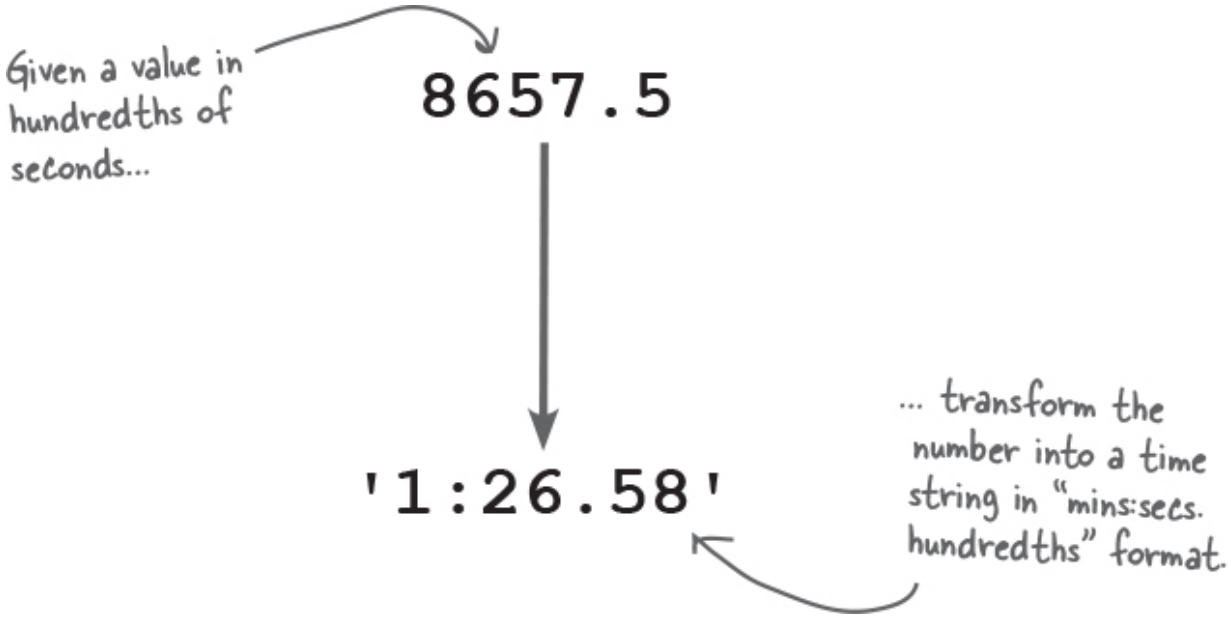
Convert the average to a swim time string

Experienced Python programmers knows enough to apply a few "tricks" to the problem of converting your hundredths of seconds back into the *mins:secs.hundredths* string format. You'll learn about these techniques later in this book, as showing them to you now would like-

ly double the size of this chapter. So, for now, let's (mostly) stick with the Python you already know to perform this task.



Follow along in your notebook while you're walked through the five steps to perform the conversion. Here's what you're trying to do:



1 Begin by converting the hundredths value to its seconds equivalent.

Let's create a new variable to store the average (and be sure to follow along).

```
average = statistics.mean(converts)
```

```
average / 100
```

```
86.575
```

Dividing by 100 gives you the seconds and hundredths of seconds values (either side of that period).

```
round(average / 100, 2)
```

Yet another BIF ("round"), rounds your division to two decimal points as opposed to three.

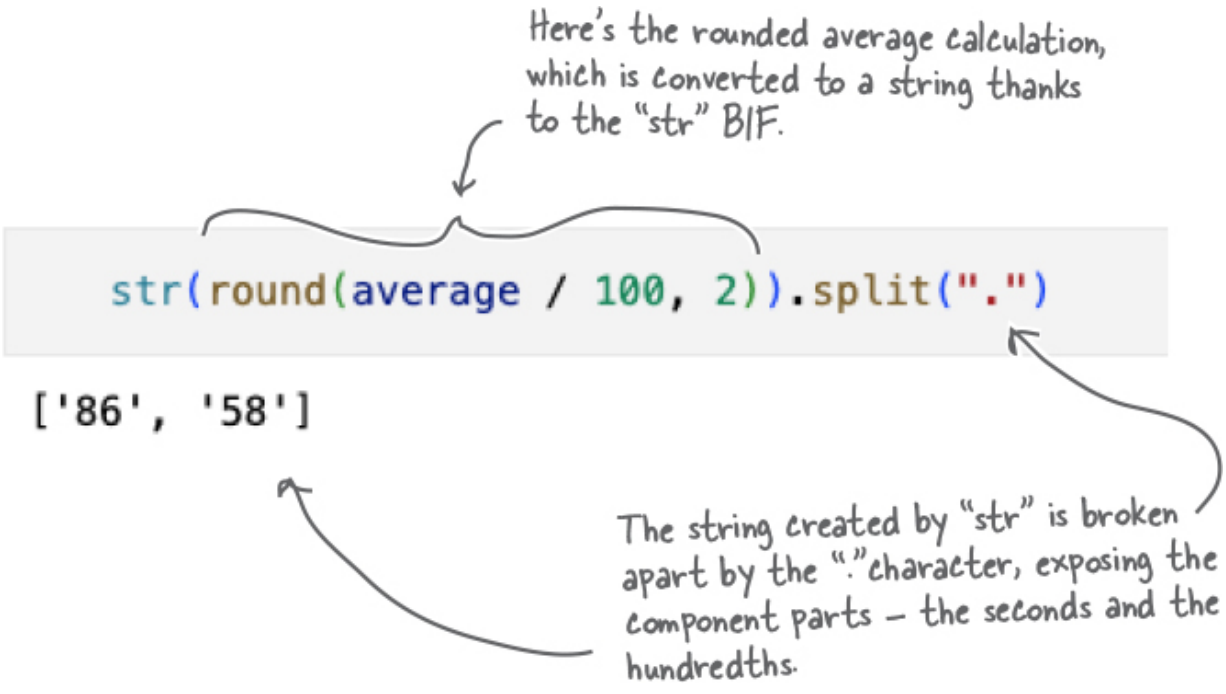
```
86.58
```

Seconds.

Hundredths.

2 Break the rounded average into it's component parts.





3 Calculate the number of minutes.

The results of the call to "split" are assigned to variables. Note: both variables are string objects.

```
mins_secs, hundredths  = str(round(average / 100, 2)).split(".")
```

```
mins_secs = int(mins_secs)
```

The assigned variables are strings, so you need to convert the "mins_secs" value to an integer as you're about to use it within a mathematical context.

```
minutes = mins_secs // 60
```

This // operator might look a little strange. It's Python's "floor division" operator, which rounds down to the nearest integer (unlike the standard / division operator which can return a decimal result).

```
minutes
```

```
1
```

And there's how many whole minutes there are in 86 seconds.

4 Calculate the number of seconds.



The number of seconds are what's left over after the number of minutes are subtracted from 86. The Maths is straightforward.

```
seconds = mins_secs - minutes*60
```

```
seconds
```

```
26
```

In step #3, you worked out there is one minute. When you subtract one minute's worth of seconds from 86 you left with 26.

5 With minutes, seconds, and hundredths now known, build the swim time string.

```
minutes, seconds, hundredths
```

```
(1, 26, '58')
```

The three calculated values...

```
str(minutes) + ":" + str(seconds) + "." + hundredths
```

```
'1:26.58'
```

... are used to build a swim time string in the required format. And, yes, the + operator works with strings as well as numbers, performing concatenation.

```
average = str(minutes) + ":" + str(seconds) + "." + hundredths
```

The formatted string is assigned to the "average" variable.

```
average
```

```
'1:26.58'
```

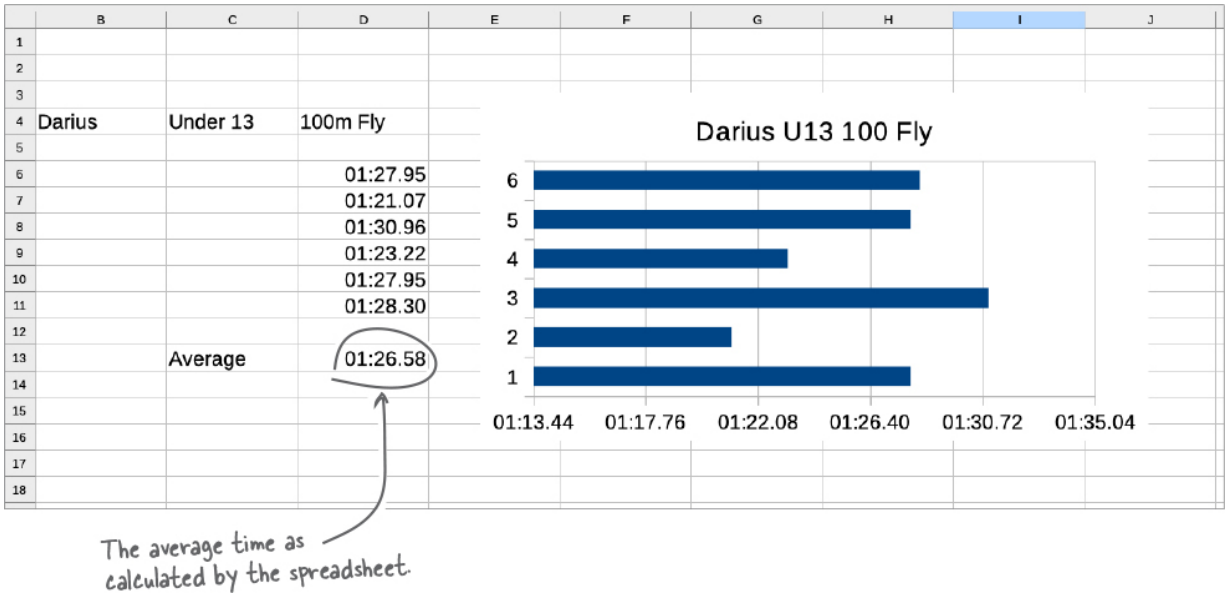


Yes, and it's easier than you think.

You could go off and learn how to write automated tests in Python, then code-up any number of tests to check your calculations...

Or you could simply take another look at the swim coach's spreadsheet to confirm your calculated swim time of '`1:26.58`' matches the average as calculated by the Coach's spreadsheet.

And it does, as shown below.



It's been a while since your last checkmark...

Congratulations! You are finally able to place a well-deserved tick against sub-task (e).

All that remains is to combine the code from the previous chapter with the code seen so far in this chapter. Once that's done, sub-task (f) will be done too:

a. Read the lines from the file ✓

b. Ignore the second line ✓

c. Break the first line apart by "," to produce a list of times ✓

d. Take each of the times and convert them to a number from the ".hundredths" format ✓

e. Calculate the average time, then convert it back to the ".hundredths" format (for display purposes) ✓

All that remains... → f. Display the variables from Task #1, then the list of times and the calculated average from Task #2

EXERCISE



At this stage, you should have a number of Jupyter notebooks in your `Learning` folder. To complete this exercise, you'll need to study the code in two of them: `Darius.ipynb` and `Average.ipynb`.

Create a new notebook, called `Times.ipynb`, which contains the Python code you need to execute to complete sub-task (f) above. All the code you need already exists, and all you're doing here is copying the relevant code from your two existing notebooks into your new one.

Be sure to execute all the code in your new notebook to confirm it executes as expected.

Take your time with this exercise then, when you're ready, flip the page to see our `Times.ipynb` in action.

EXERCISE SOLUTION



At this stage, you should have a number of Jupyter notebooks in your `Learning` folder. To complete this exercise, you had to study the code in two of them: `Darius.ipynb` and `Average.ipynb`.

You were to create a new notebook, called `Times.ipynb`, which contained the Python code you needed to execute to complete sub-task (f). All the code you need already existed.

You were to be sure to execute all the code in your new notebook to confirm it executes as expected.

Here's the code we copied into `Times.ipynb` and executed. How does the code you copied compare?

```
FN = "Darius-13-100m-Fly.txt"
FOLDER = "swimdata/"
```

We started with the code that defines the two constant values, identifying the filename to use as well as its location.

Next up was that powerful line of code from the end of the previous chapter which extracted the values which identifies the swimmer, their age, the distance classification and the stroke swam.

```
swimmer, age, distance, stroke = FN.removesuffix(".txt").split("-")
```

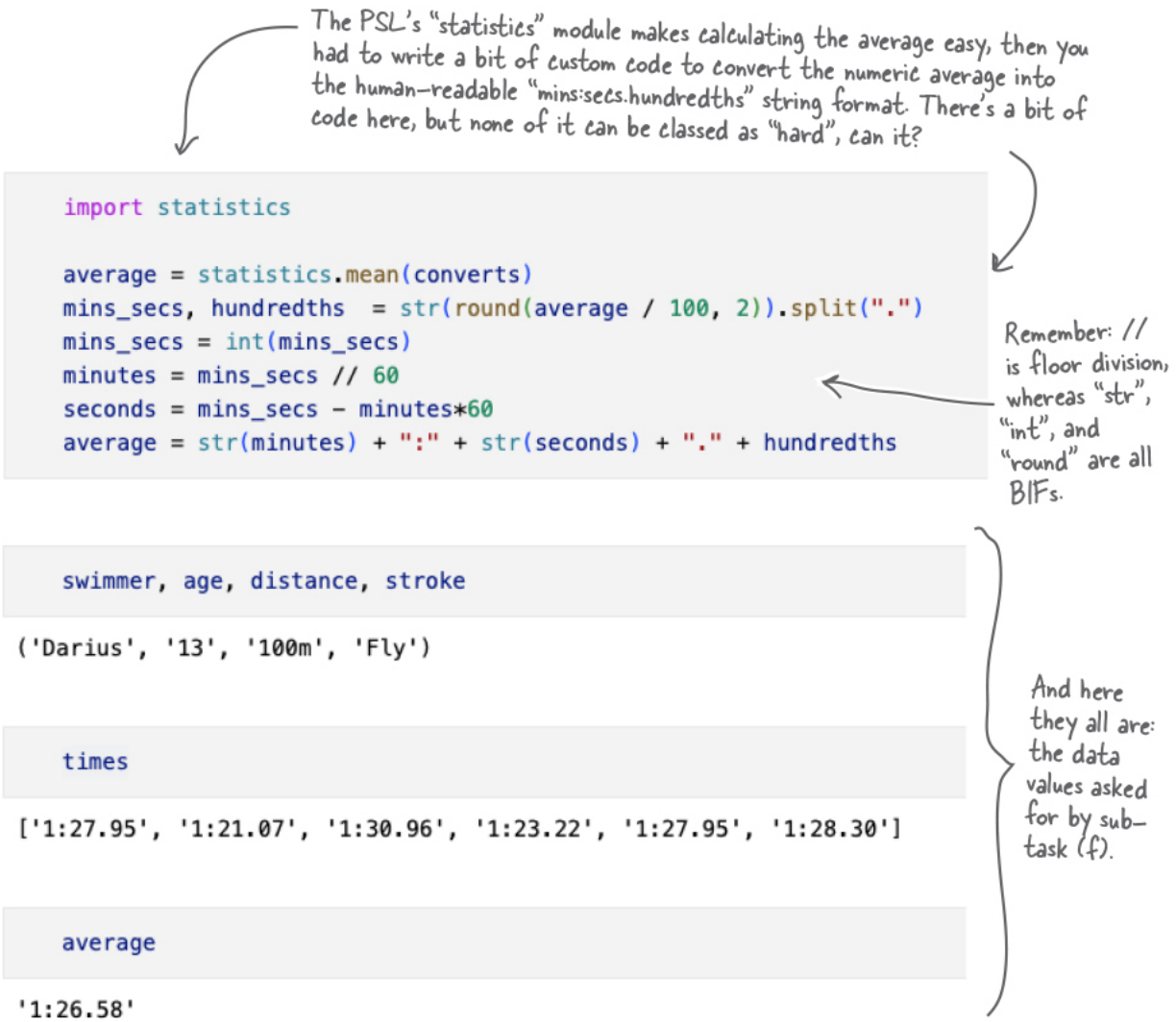
In the original code, this was called "fn" (i.e., written in lowercase). As it is meant to be used as a constant, we tweaked this code to show it in UPPERCASE.

```
with open(FOLDER+FN) as df:
    data = df.readlines()
times = data[0].strip().split(",")
```

The "with" statement opens the data file, reads the lines in the file into a list, then closes the file (automatically for you). A quick strip-split combo on the first line of data from the file creates another list, called "times", which contains this swimmer's swim time strings.

The strings in "times" are converted from the "mins:secs.hundredths" format into numeric hundredths of seconds, ending up in yet another list called "converts". The "for" loop, together with the "append" method, makes this as easy as it can be.

```
converts = []
for t in times:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
    converts.append(int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```



Task #2 (finally) gets over the line!

Well done! With the creation (and execution) of the `Times.ipynb` notebook, the two tasks identified at the start of the previous chapter are now complete. It's a case of checkmarks all around!

1. **Extract data from the file's name**

a. Read the filename ✓

b. Break the filename apart by the "–" character ✓

c. Put the swimmer's name, age group, distance , and stroke into variables (so they can be used later) ✓

2. **Process the data in the file**

a. Read the lines from the file ✓

b. Ignore the second line ✓

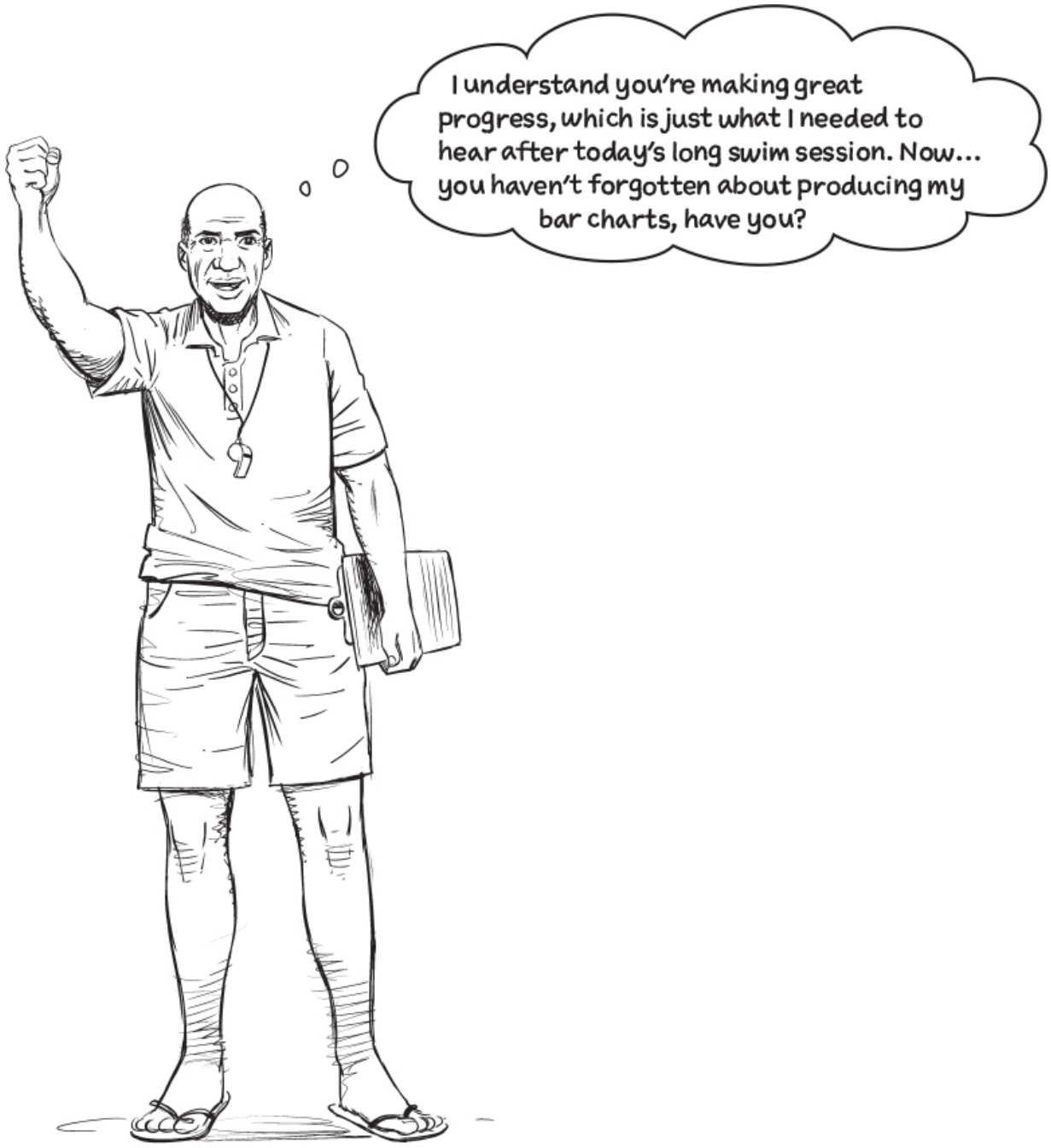
c. Break the first line apart by "," to produce a list of times ✓

d. Take each of the times and convert them to a number from the ".hundredths" format ✓

e. Calculate the average time, then convert it back to the "mins:secs.hundredths" format (for display purposes) ✓

f. Display the variables from Task #1, then the list of times and the calculated average from Task #2 ✓

Of course, getting to this point doesn't necessarily mean you're done...



No, we haven't forgotten.

The next chapter lays the groundwork for getting to the point where you can tackle the charting requirement, so we've ask the Coach to

bear with us.

At the moment, your code only works with the data for one specific data file. There are another 60+ files in the Coach's dataset. It would be nice if there was a way to use this code with *any* of them on demand, and as needed.

Doing so is something you can mull over on your way to the next chapter when we'll work through a solution to this problem *together*.

For now, let's conclude this chapter with another summary and a super-topical crossword puzzle. Enjoy!

CHAPTER REVIEW BULLET POINTS

- Python has no real notion of a **constant** value, so a programming *convention* has emerged whereby constants are assigned to UPPERCASE variable names.
- The **with** statement is used to manage the context within which its block of code runs.
- When used with the **open** BIF, the **with** statement *automatically* arranges to close any still opened file once the **with**'s block of code terminates.
- Additionally, the **with** statement assigns the opened file to a **file object** which is used as shorthand for the opened file.
- The file object has lots of methods built-in, including **readlines** which... emm, eh... reads lines. The lines are returned as a **list**.
- The rest of the file object methods can be displayed with the **print dir** combo mambo.
- By default, the **open** BIF opens a named file for reading. The file is assumed to contain **textual data**.
- The colon, in addition to being your **BFF**, indicates to Python that an **indented** block is about to start on the very next line of code.
- The **.strip().split()** chain is very popular. It used in lots of places, by lots of Python programmers.
- Python has all the usual **operators** (e,g., `+`, `-`, `*`, `/`), and then some (e.g., `//` and `:=`).

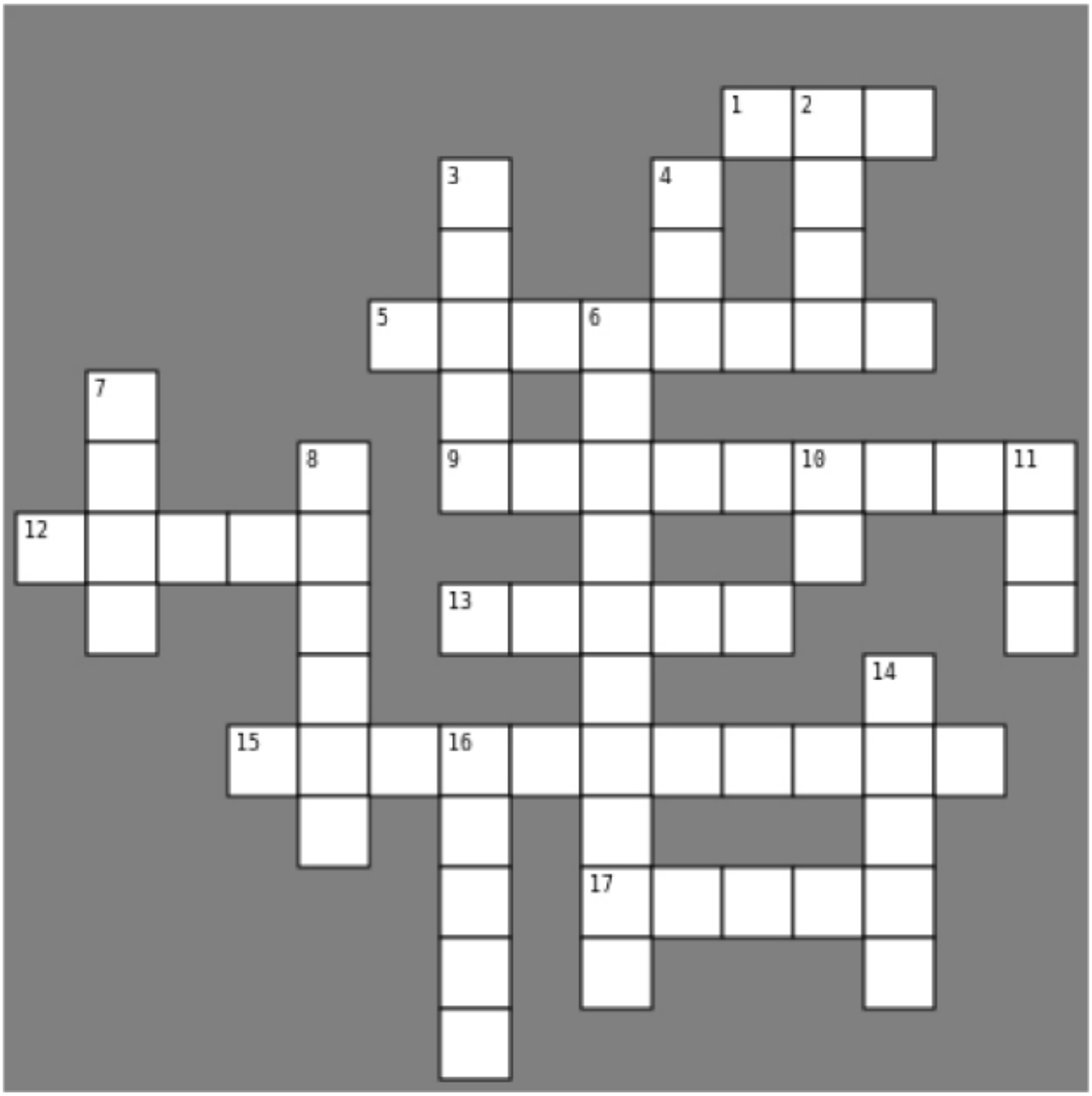
- The `//` operator performs **floor division** which ensures, among other things, that the result of the division is always a whole number.
- Although you are well within your rights to *love* the **for** and **while** loops equally, the **for** loop sees a lot more use in the community, and it's typically reached for *first* whenever a Python programmer needs to iterate.
- Python's **list** is a very popular built-in data structure, which understands the familiar square bracket notation.
- An empty list looks like this: `[]`.
- Items in a list are enclosed in **square brackets** and separated from each other by a **comma.** Knowing this, lists easy to spot.
- Lists come with lots of built-in functionality, but the *star of this show* (in that chapter) is the **append** method.
- The **int** BIF converts its single argument to an integer, if it can. The value `"42"` converts without issue, whereas `"forty-two"` sends **int** into meltdown.
- The **PSL** (not the coffee!!) provides the `statistics` module which, among other things, provides a handy **mean** function. The **mean** function has everything to do with calculating averages, as opposed to not being nice to be around...
- The **round** BIF gets rid of unwanted decimal places.
- The **str** BIF converts its argument to a string.

- The Coach is really **happy** with your progress to date, as should you be. There's a lot of **good work** in this chapter.

THE LISTS CROSSWORD



All of the answers to the clues are found in this chapter's pages, and the solution is on the next page. Got for it!



Across

1. Python programmer's favorite looping construct.

5. When a variable name is in UPPERCASE, it's meant to be treated as one of these.

9. This method creates a list from your file's data.

12. Part of a famous combo, when paired with **split**.

13. The less-used looping construct.

15. Another name for whitespace when used with code blocks.

17. Your new BFF.

Down

2. Another powerful combo when used with 7 down.

3. `//` performs _____ division.

4. A numeric conversion BIF.

6. A module loved by Maths-heads.

7. The recommended statement to use when opening files.

8. This method grows lists.

10. A small keyword which you'll learn more about in a later chapter.

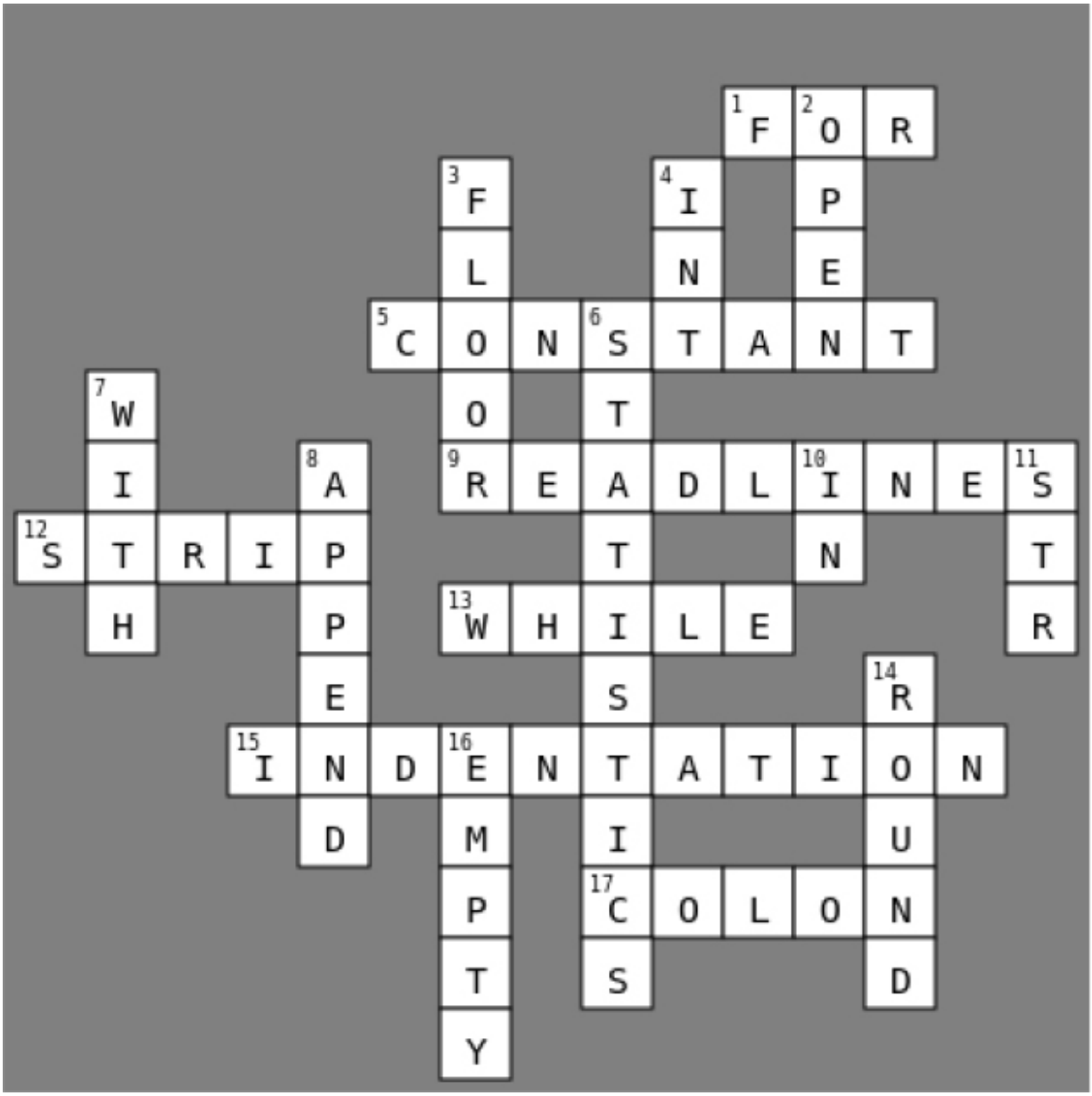
11. A string-creating BIF.

14. A BIF to control decimal places.

16. This `[ ]` signifies an _____ list.

THE LISTS CROSSWORD SOLUTION





Across

1. Python programmer's favorite looping construct.

5. When a variable name is in UPPERCASE, it's meant to be treated as one of these.

9. This method creates a list from your file's data.

12. Part of a famous combo, when paired with **split**.

13. The less-used looping construct.

15. Another name for whitespace when used with code blocks.

17. Your new BFF.

Down

2. Another powerful combo when used with 7 down.

3. `//` performs _____ division.

4. A numeric conversion BIF.

6. A module loved by Maths-heads.

7. The recommended statement to use when opening files.

8. This method grows lists.

10. A small keyword which you'll learn more about in a later chapter.

11. A string-creating BIF.

14. A BIF to control decimal places.

16. This `[ ]` signifies an _____ list.

Chapter 3. List of Files: *Functions, Modules & Files*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 3 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



Your code can't live in a notebook forever. It wants to be free. And when it comes to freeing your code and **sharing** it with others, a bespoke **function** is the first step, followed shortly thereafter by a **module,** which lets you organize and share your code. In this chapter, you'll create a function directly from the code you've written so

far, and in the process create a **shareable** module, too. You'll immediately put your module to work as you process the Coach's swim data with **for** loops, **if** statements, conditional tests, and the **PSL** (Python's *Standard Library*). You'll learn how to **comment** your functions, too (which is always a *good idea*). There's lots to be done, so let's get to it!



Cubicle Conversation

Sam: I've updated the Coach on the progress-to-date.

Alex: And is he happy?

Sam: To a point, yes. He's thrilled things have started. However, as you can imagine, he's only really interested in the final product, which for the Coach is the bar chart.

Alex: Which should be easy enough to do now that the most-recent notebook produces the data we need, right?

Mara: Well... sort of.

Alex: How come?

Mara: The current notebook, `Times.ipynb`, produces data for Darius swimming the 100m Fly in the Under 13 age group. But, there's a need to perform the conversions and the average calculation for *any* swimmer's file.

Alex: Sure that's easy: just replace the filename at the top of the notebook with another filename, then press the *Run All* button and – voila! – you've got your data.

Mara: And you think the Coach will be happy to do that?

Alex: Errr... I hadn't thought about how the Coach is going to run this stuff.

Sam: We are heading in the right direction, in that we do need a mechanism which works with any swimmer's filename. If that can be

produced, we can get on with then creating code for the bar chart.

Alex: So we have a ways to go yet...

Mara: Yes, but not far. As you already mentioned, all the code we need is in the `Times.ipynb` notebook...

Alex: ...which you don't want to give to the Coach...

Mara: ...well, not it it's current form.

Alex: Then how?

Sam: We need a way to package-up the code so it can be used with any filename and accessed outside of the notebook...

Alex: Ah, but of course: we need a function!

Sam: Which gets us part of the way.

Mara: If the function is put inside a Python module it can be shared in lots of places.

Alex: Sounds good to me. Where do we start?

Mara: Let's start by turning the existing notebook code into a function that we can call, then share.

How to create a function in Python

In addition to the actual code for the function, you need to worry about the function's *signature*. There are three things to keep in mind. You need to:

1 Think up a nice, meaningful name

The code in the `Times.ipynb` notebook first processes the file-name, then processes the file's contents to extract the data required by the Coach. So let's call this function `get_swim_data`. It's a nice name, it's a meaningful name... golly, it's nearly perfect!

2 Decide on the number and names of any parameters

Your new `get_swim_data` function takes a single parameter which identifies the filename to use. Let's call this parameter `fn`.

3 Indent your function's code under a def statement

The **def** keyword introduces the function, letting you specify the function's name and any parameters. Any code indented under the **def** keyword is the function's code block.

NOTE

It can be useful to think of "def" as shorthand for "define function".

ANATOMY OF A FUNCTION SIGNATURE



1 **Use a nice, meaningful name**

This names gives the user of your function a good idea of what it does.

2 **Name any parameters**

You only have a single parameter here.

```
def get_swim_data(fn):
```

3 **Note the use of def and your BFF (the colon)**

The use of `def` and the colon is your clue indented code is on it's way.

Be sure to add a comment to your function

Full disclosure: we told a little white lie... we've led you up the garden path to believe all you need to do is copy your code into your function, suitably indented. But, you should *also* add a comment to the start of your function.

You've already seen how a single `#` character switches on a single-line comment in your code. When you need to add a multi-line comment, you can enclose your comment in **triple-quotes** which, believe it or not, is *also* a string object.

NOTE

A string can span many lines when your surround it with triple-quotes.

When Python encounters a string object which isn't assigned to a variable name, it's simply ignored, which makes using a triple-quoted string *perfect* for multi-line comments. Here's an example:

The function's signature.

```
def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
```

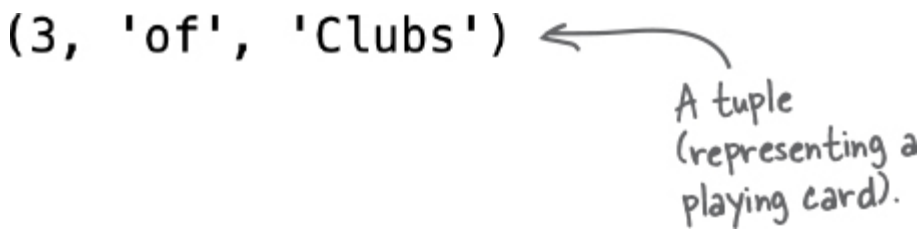
The comment (which future you will thank you for).

Note the triple-quotes surrounding this text.



Great question (and well spotted).

The **tuple** is one of Python's built-in data structures, which you briefly met in Chapter 0: Why Python? Similar But Different. Back then we over-generalized a little, suggesting tuples were quite like lists. We'll have more to say about tuples later in this book. For now, recall what one looks like:



Now, what's your second question?



That's another great observation.

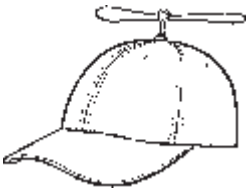
No, we haven't forgotten to declare the parameter's type. In fact, in Python you don't need to declare the types of your parameters, as every function parameter can be of *any* type.

Python is a *dynamically-typed* language. This means, among other things, that typing decisions are not made until run-time, which is very unlike those *stodgy* statically-typed languages where everything must be known up-front, at compile-time.

NOTE

Depending on your programming background, this might strike you as controversial. Our advise is not to let this worry you. This page's "Geek Note" discusses a Python 3 feature which can be thought of as meeting the statically-typed die-hards half-way.

GEEK NOTE



Recent versions of Python 3 have introduced the notion of **type hints**. This extra syntax adds annotations to your function's signature to *indicate* the types expected. As an example, here's how you'd add type hints to your `get_swim_data` function to indicate a string is expected as input, with a tuple returned:

```
def get_swin_data(fn: str) -> tuple:
```

These annotations are "hints" and are otherwise **ignored** by Python: there's nothing checking at run-time to ensure a string is actually sent as input to the function. This means the use of type hints is *optional*. Python is dynamically-typed, after all. Some programmers believe type hints are the best thing since sliced bread, whereas others are... less enthusiastic. We're on the fence and, consequently, use type hints in this book sparingly (if at all).

Create a file for your function's code

There's nothing stopping you defining any function within a Jupyter Notebook code cell. In fact, when you are trying to work out what code you need to create, experimenting in a notebook *first* is our recommended approach.



However, all the code you need to add to your `get_swim_data` function already exists in your `Times.ipynb` notebook, so there's no need for any further experimentation. Instead, let's use VS Code to create a new Python file called `swimclub.py`.

NOTE

In VS Code, select "File", then "New File", then select "Python File" as the type. When an empty editing window appears, save the file as "swimclub.py". (And be sure to save the file into your existing "Learning" folder).

Here's the code we copied into `swimclub.py`, which we've applied a few minor tweaks to (note the annotations):

"Python File" as the type. When an empty editing window appears, save the file as "swimclub.py". (And be sure to save the file into your existing "Learning" folder).

The code relies on this PSL module, so be sure to import it.

Define any required constant values (note: the "FN" constant is *not* needed here as the filename is passed as a parameter to the function).

```
import statistics

FOLDER = "swimdata/"

def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths
```

There are minor edits here. Rather than using the constant "FN", we're referring to the parameter from the function's signature.

All of this code is indented under that "def" statement.

Another small edit was to change this line's assigned-to variable name to "average_str". The "average" name was used earlier to store the numeric average, and we decided to remember both the number and the converted string in their own variables.



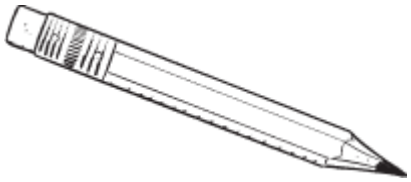
Yes, that's it exactly.

There's just a small addition to apply to the function's code to ensure it's really useful.

Functions can return results to their calling code

In addition to accepting data *on the way in*, function's can deliver data to the code which invoked them (aka data *on the way out*). Arguments sent into a function are assigned to the parameter names defined in the function's signature, whereas any results are sent back to the calling code with a **return** statement.

SHARPEN YOUR PENCIL

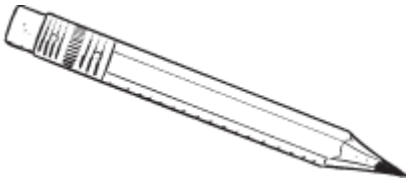


Take a moment to read the code on the previous page then, in the space below, write in the **return** statement you'd add to the end of your function in order to send values back to the function's caller.

You'll find our suggested **return** statement on the next page, but do have a go at creating this single line of code yourself before flipping the page:

_____

SHARPEN YOUR PENCIL SOLUTION



You were to take a moment to read the code from two pages back then, in the space below, write in the **return** statement you'd add to the end of your function in order to send values back to the function's caller.

Here's the **return** statement we'd use. How does yours compare?

```
return swimmer, age, distance, stroke, average, average_str, times
```

A collection of values are returned from the function to the calling code. Note the absence of parentheses around that list of variable names (Python doesn't require them).

Update and save your code before continuing...

Before moving onto the next page, be sure to add the following line of code as the last line in your `get_swim_data` function within your `swimclub.py` file. Be careful to match the indentation of this line of code with the indentation used for all the other code in your function:

Use VS Code to add this line of code to the end of your function, then save your file.

```
    return swimmer, age, distance, stroke, average, average_str, times
```

WATCH IT!



Remember to save your code often when using the VS Code editor

When you use a Jupyter Notebook, you execute your code immediately using **Shift+Enter** *without having to save anything beforehand.*

However, when you use an editor to create Python code (as you just did with VS Code and your `swimclub.py` *file), nothing run until you specifically tell Python to run your code. When you do, the* **latest saved version** *of your code is executed, so be sure to save early and often when working within an editor.*

Use modules to share code

If you look at the code in your `swimclub.py` file, it consists of a single **import** statement, a single constant definition, and a single function. Once you move code into its own file, it becomes is a Python *module*. Congratulations: you've created your first Python module, called swimclub.



That's pretty much it.

Things can get a lot more complex but, as you've just seen, creating a **shareable module** in Python is as simple as popping some code in a file and giving it a name. It's so easy, there ought to be a law against it. ..

EXERCISE



With the code in `swimclub.py` complete, you were to create another new notebook in your `Learning` folder, giving it the name `Files.ipynb`.

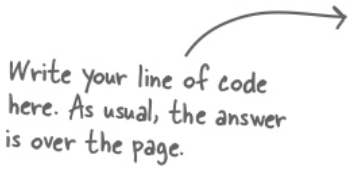
In your new notebook's first cell, you were to type this line of code, being sure to type **Shift+Enter** to execute it:

```
import swimclub
```

When importing any module, you never include the ".py" extension.

This imports any constants and functions defined in the `swimclub` module from your `swimclub.py` file. In the space below, you were

to provide the line of code which calls the `get_swim_data` function from this module, passing the name of Darius's file from the previous chapter:



EXERCISE SOLUTION



With the code in `swimclub.py` complete, you were to create another new notebook in your **Learning** folder, giving it the name **Files.ipynb**.

In your new notebook's first cell, you were to type this line of code, being sure to type **Shift+Enter** to execute it:

```
import swimclub
```

This imports any constants and functions defined in the `swimclub` module from your `swimclub.py` file. In the space below, you were to provide the line of code which calls the `get_swim_data` function from this module, passing the name of Darius's file from the previous chapter:

This is the line of code we used. We prefix the name of the function with the name of the module, then pass the name of the file as the argument value.

```
swimclub.get_swim_data("Darius-13-100m-Fly.txt")
```



This is a fully qualified name.

When you refer to your function with “module DOT function”, you are qualifying the name of your function with the name of the module which contains it. This is very common in practice, although there are other common importing techniques. You’ll see examples of these as you continue to work through this book.

TEST DRIVE



If you haven't done so already, press **Shift+Enter** on each of your notebook's code cells to execute them. The first cell produces no output as all that happens is the `swimclub` module is imported. The **import** statement runs the code in the identified module from top-to-bottom. The second cell run your functiona and produces output as shown below:

```
import swimclub
```

```
swimclub.get_swim_data("Darius-13-100m-Fly.txt")
```

```
('Darius',
 '13',
 '100m',
 'Fly',
 8657.5,
 '1:26.58',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'])
```

The data extracted from the file name.

The average (as a numeric and a time string).

The list of time strings.



Good eye. Well spotted, too.

This may not be the explanation you're expecting here, but those parentheses are meant to be there.

Let's dig into this a little so you can appreciate what's going on.

TODAY'S INTERVIEW



Chatting with Python's **function.**

Head First: Thank you for taking time out of your busy day to chat to us.

Function: No problem.

Head First: How come you're so busy?

Function: I'm used everywhere, all the time.

Head First: And you'll work with anything?

Function: If you're referring to the data I'll accept then, yes, I'll happily accept anything you give me.

Head First: Care to expand?

Function: Sure. You can send me any number of argument values which I'll happily match up with my parameters, and all you need to

do is ensure you send me the correct number. If I've got two parameters, be sure to send me two argument values.

Head First: What happens if I send, say, one argument or three instead?

Function: I get cranky.

Head First: I see. It's like that, is it?

Function: Yes. I'm pretty strict on that sort of thing. Unless, of course, one of your two example parameters happens to be declared as *optional*.

Head First: What happens then?

Function: Continuing on with my example of a two-parameter function... if, for instance, the second parameters is optional, I'll happily accept one or two argument values, no questions asked.

Head First: And if I call you with a single argument, what's assigned to the second parameter?

Function: Typically I'll assign a default value which has been decided upon by the programmer who created me.

Head First: That sounds kind of complex.

Function: It's not really and – to be honest – it's not something every function needs, but it's part of me if you need it. I'm pretty flexible.

Head First: And what about returning values? Is it the same deal? Can any number of values be returned?

Function: No.

Head First: Seriously? No? That's all you're going to say on this?

Function: Well... I thought is was kind of obvious. Think of mathematics where functions return a single result, not multiple results. It's the same with me. Any number of values in, but only ONE result back.

Head First: But... emm... err... look at the `get_swim_data` function call on the previous page. There are *seven* results returned.

Function: No there isn't, there's only ONE.

Head First: What the...

Function: If you look closely, you'll notice those parentheses surrounding the seven data values, right?

Head First: Yes, but...

Function: No “buts” about it. Those parentheses are a single tuple which happens to contain the seven individual data values. As I said, only ONE result is returned, either a single result on it’s own, or a single tuple result with multiple values contained therein.

Head First: But the code doesn’t convert the seven return values into a tuple.

Function: Yeah, the code didn’t, *but I did*. I do it automatically when I see a programmer attempting to return more than ONE result. You can thank me later.

Head First: No, I’ll thank you now. That’s important to know. Thanks for the chat!

Function: I’m always happy to clear things up. Bye!

Functions return a tuple when required

When you call a function which looks like it returns multiple results, it isn’t. Instead, you get back a single tuple containing a collection of results (regardless of how many there are).

This looks like seven objects are being returned from the function. But this is not allowed as functions can only every return one result. So Python bundles the returned objects into a single tuple.

```
return swimmer, age, distance, stroke, average, average_str, times
```

```
('Darius',
 '13',
 '100m',
 'Fly',
 8657.5,
 '1:26.58',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'])
```

Tuples surround their objects with parentheses (unlike lists, which use square brackets).



That's a great suggestion.

Not that we’re suggesting there’s a bit of mind reading going on here, but it is a little spooky we had the same idea...

BEHIND THE SCENES



Up close and personal with Python's tuple.

Python's docs state that a **tuple** is an *immutable sequence*.

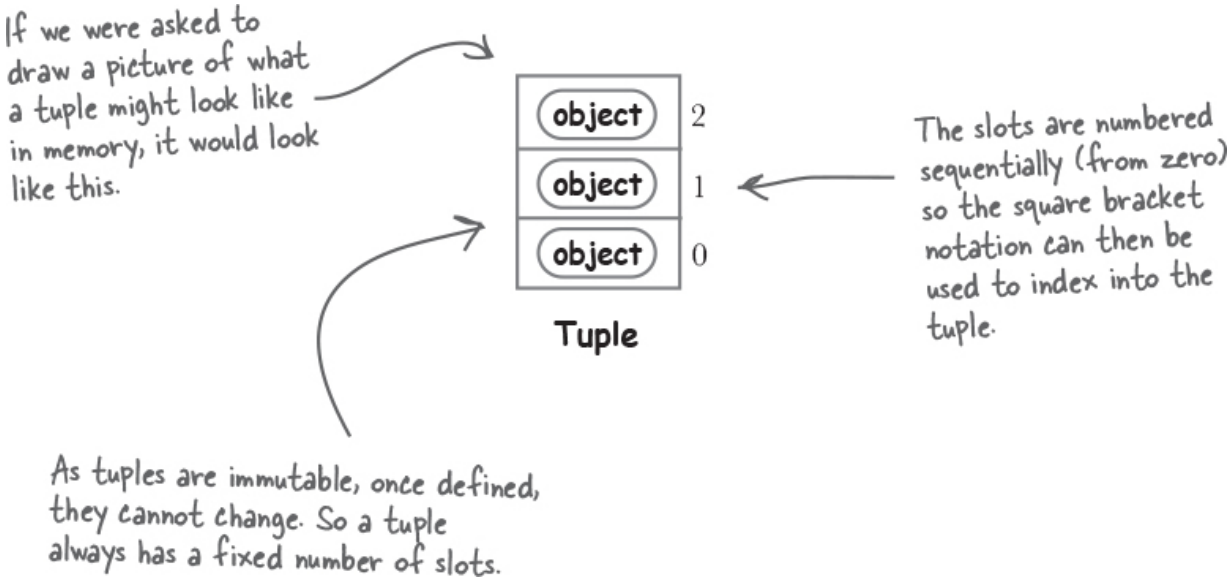
Immutable?

You've already met two *immutable* data types: numbers and strings. Both share the characteristic that once a value is created in code, the value **cannot** be changed. For instance, `42` is always `42`: it cannot change, it's *immutable*. The same thing goes for strings in Python: `"hello"` is always `"hello"`: once created, it cannot change, it's *immutable*.

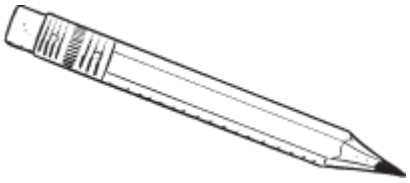
Python's **tuple** takes this idea of immutability and applies it to a collection of data values. It can sometimes help to think of a tuple as a *constant list*. Once values are assigned to a tuple, the tuple cannot change, it's *immutable*.

Sequence?

If you can use the square-bracket notation to access items (or slots) from a collection, then you're working with a *sequence*. Python's list is the most familiar sequence type, but others do exist, including strings and... **tuple**. In addition to supporting the use of square brackets, sequences also maintain the *order* of the items they contain.



SHARPEN YOUR PENCIL



Assume the following line of code has executed in a new, empty code cell in your notebook (hint: go ahead and execute this line of code in your notebook so you can experiment as needed):

```
data = swimclub.get_swim_data("Darius-13-100m-Fly.txt")
```

Using the square bracket notation, extract the list of swim times from the `data` tuple, assigning them to a list called `times`. Then display the contents of the `times` list. Write the code you used here:

_____

_____

_____

Using Python's unpacking technology (aka *multiple assignment*), extract the list of swim times from the `data` tuple a second time, as-

signing them to a list called `times2`. Display the contents of the `times2` list. Write the code you used here:

_____

_____

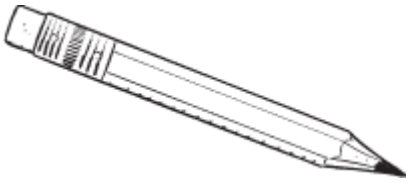
_____

THERE ARE NO DUMB QUESTIONS

Q: I just did a "print dir" on my `data` tuple and there's only two methods that aren't dunders. Is it the case that you can't do anything useful with tuples?

A: No, that's not the case. Now, that said, if you compare the output from the **print dir** combo mambo for a list to a tuple, it certainly appears that tuples are inferior to lists as they have next to no methods associated with them. But, recall: tuples are immutable, so you can never change a tuple once it's assigned data (this fact alone reduces the number of methods you need with tuples). As you work through this book you'll see examples of where it makes sense to use a tuple over a list, and vice versa. Bottom line: you need both.

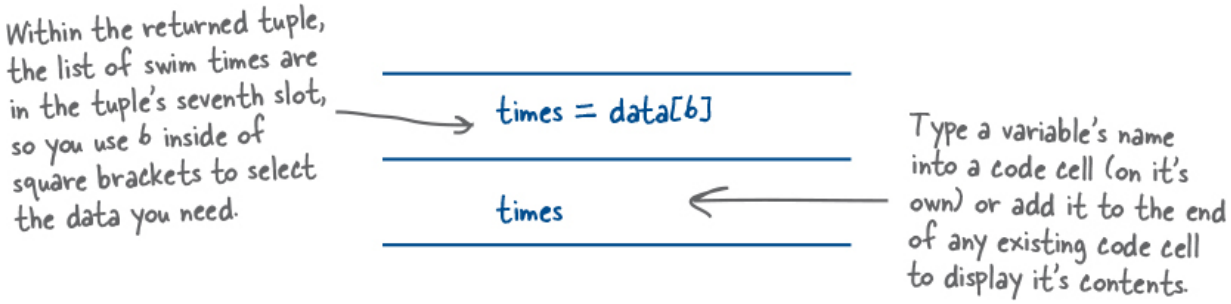
SHARPEN YOUR PENCIL SOLUTION



You were to assume the following line of code has executed in a new, empty code cell in your notebook:

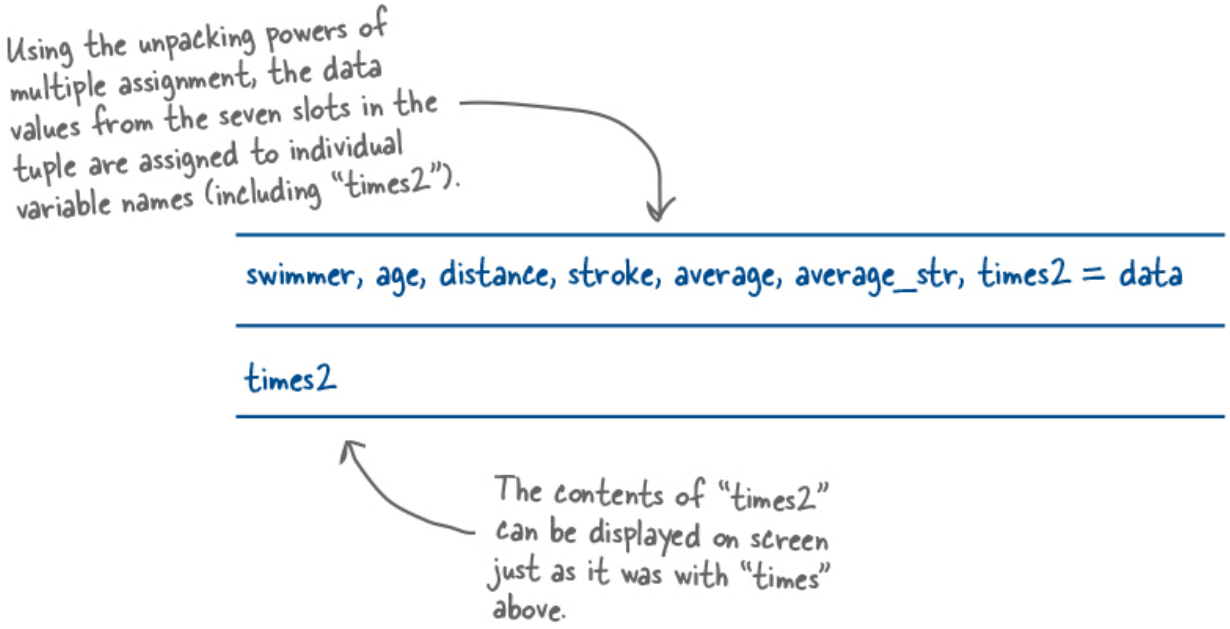
```
data = swimclub.get_swim_data("Darius-13-100m-Fly.txt")
```

Using the square bracket notation, you were to extract the list of swim times from the `data` tuple, assigning them to a list called `times`. You then were to display the contents of the `times` list on screen:



Using Python's unpacking technology (aka *multiple assignment*), you were to extract the list of swim times from the `data` tuple a second

time, assigning them to a list called `times2`. You then were to display the contents of the `times2` list on screen:





There's often a "better" way.

The code on the last page works and does what you expect it to. (You did try it out in your notebook, didn't you?!?). It is possible to make both examples more *Pythonic* with a couple of small changes, which you'll explore over the page.

NOTE

"Pythonic": when code is written to take advantage of what Python has to offer, as opposed to Python code which clearly looks like it been written to conform to some other programming language's idea of "the right way".

The hope is, once these changes are applied, your code won't feel quite so unwieldy.

BEHIND THE SCENES



Multiple assignment, default variables, and Pythonic code, 1 of 2.

With the "data" variable created (by calling your "get_swim_data" function), the good ol' square bracket notation can pick out the data your need.

```
times = data[6]
times
```

The item in the seventh slot of the "data" tuple.

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

The list of swim times is in the tuple's last slot, so it's more Pythonic (and convenient) to index from the other end using negative indexing with square brackets.

```
times = data[-1]
times
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

The item in the last slot of the "data" tuple (which is the same item in "data[6]").

If you want to avoid square brackets altogether, the use of multiple assignment assigns each of the tuple's items to their own variable name.

```
swimmer, age, distance, stroke, average, average_str, times2 = data
times2
```

"times2" is the same list of swim times.

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

If all you need is this list of swim times, it's more Pythonic to replace those unused individual variable names with the default variable

```
_, _, _, _, _, _, times2 = data
times2
```

"times2" is still the same list of swim times.

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

BEHIND THE SCENES



Multiple assignment, default variables, and Pythonic code, 2 of 2.

If you want to remember the rest of the data, but not as individual variables, it's more Pythonic to rewrite the previous code to pack the rest of the data into a different list (called "the_rest" here). Note the variable's name is prefixed with * which enables this behavior.

```
*the_rest, times3 = data
times3
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

Still the same list of swim times.

```
the_rest
```

```
['Darius', '13', '100m', 'Fly', 8657.5, '1:26.58']
```

The rest of the data values from "data" in a new list called "the_rest".

If you do not want to keep the rest of the data in it's own list variable, you can assign it (in one go) to the default variable. This is also a very Pythonic thing to do. Note that in this form of multiple assignment (as in the previous code, above) it doesn't matter how many other items are in the data list. The last item is always assigned to "times4", with whatever's left assigned to the variable.

```
*_, times4 = data
times4
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

The rest of "data" is in the default variable.

```
_
```

```
['Darius', '13', '100m', 'Fly', 8657.5, '1:26.58']
```

WATCH IT!



Be careful: don't rely on the default variable.

The use of the default variable is very Pythonic. However, care is needed. Even though you can assign values to the default variable (the single underscore), you can't rely on the value staying assigned to `_` *for any length of time. Python (and Jupyter) uses* `_` *behind the scenes in lots of places, so it's value can change... and you won't know. Yikes.*

If you need to remember a value, assign it to a variable with a proper variable name. Don't rely on the default variable.



A list of filenames would be nice.

Your `get_swim_data` function, part of the `swimclub` module, takes any swimmer's filename and returns a tuple of results to you.

What's needed need now is the full list of filenames which you should be able to get from your underlying operating system. As you can imagine, Python has you covered.

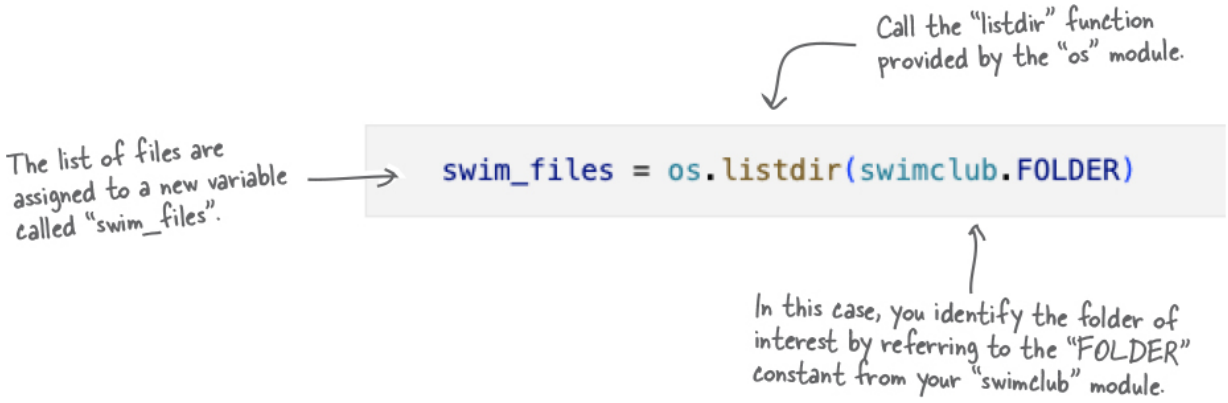
Let's determine the list of filenames

When it comes to working with your operating system (whether you're on *Windows*, *macOS*, or *Linux*), the PSL has you covered. The `os` module lets your Python code talk to your operating system in an platform-independent way, and you'll now use the `os` module to grab a list of the files in the `swimdata` folder.

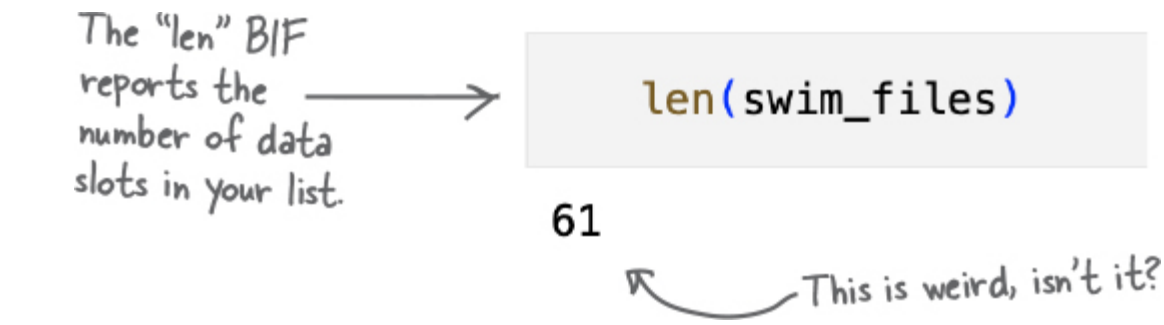
Be sure to follow along in your `Files.ipynb` notebook.



You want the names of the files in your `swimdata` folder, and the `os` module provides the handy-dandy `listdir` function to do just that. When you pass in the location of a folder, `listdir` returns a list of all the files it contains:



You'd be forgiven for expecting the `swim_files` list to contain 60 pieces of data. After all, there are 60 files in your folder. However, on our *Mac*, we were in for a shock when we double-checked how big `swim_files` is:



It's time for a bit of detective work...

You were expecting your list of files to have 60 filenames, but the **len** BIF is reporting 61 items in your `swim_files` variable.

In order to begin to try and work out what's happening here, let's first display the value of `swim_files` list on screen:



Type this into an empty code cell, then press "Shift+Enter".

```
print(swim_files)
```

```
['Hannah-13-100m-Free.txt', 'Darius-13-100m-Back.txt', 'Owen-15-100m-Free.txt', 'Mike-15-100m-Free.txt',
'Hannah-13-100m-Back.txt', 'Mike-15-100m-Back.txt', 'Mike-15-100m-Fly.txt', 'Abi-10-50m-Back.txt', 'Ruth-
13-200m-Free.txt', '.DS_Store', 'Tasmin-15-100m-Back.txt', 'Erika-15-100m-Free.txt', 'Ruth-13-200m-
Back.txt', 'Abi-10-50m-Free.txt', 'Maria-9-50m-Free.txt', 'Elba-14-100m-Free.txt', 'Tasmin-15-100m-
Free.txt', 'Abi-10-100m-Back.txt', 'Abi-10-50m-Breast.txt', 'Mike-15-200m-IM.txt', 'Ali-12-100m-Back.txt',
'Ruth-13-100m-Back.txt', 'Chris-17-100m-Back.txt', 'Ali-12-100m-Free.txt', 'Darius-13-100m-Breast.txt',
'Ruth-13-100m-Free.txt', 'Aurora-13-50m-Free.txt', 'Katie-9-100m-Breast.txt', 'Alison-14-100m-Breast.txt',
'Ruth-13-400m-Free.txt', 'Emma-13-100m-Free.txt', 'Calvin-9-50m-Fly.txt', 'Darius-13-100m-Fly.txt', 'Mike-
15-200m-Free.txt', 'Emma-13-100m-Breast.txt', 'Tasmin-15-100m-Breast.txt', 'Blake-15-100m-Free.txt', 'Abi-
10-100m-Breast.txt', 'Chris-17-100m-Breast.txt', 'Blake-15-100m-Back.txt', 'Katie-9-100m-IM.txt', 'Bill-18-
200m-Back.txt', 'Darius-13-200m-IM.txt', 'Dave-17-100m-Free.txt', 'Alison-14-100m-Free.txt', 'Lizzie-14-
100m-Free.txt', 'Katie-9-50m-Fly.txt', 'Katie-9-50m-Breast.txt', 'Katie-9-50m-Back.txt', 'Lizzie-14-100m-
Back.txt', 'Tasmin-15-200m-Breast.txt', 'Katie-9-50m-Free.txt', 'Dave-17-200m-Back.txt', 'Erika-15-200m-
Breast.txt', 'Calvin-9-50m-Back.txt', 'Calvin-9-50m-Free.txt', 'Carl-15-100m-Back.txt', 'Bill-18-100m-
Back.txt', 'Katie-9-100m-Free.txt', 'Blake-15-100m-Fly.txt', 'Erika-15-100m-Breast.txt', 'Katie-9-100m-
Back.txt']
```

The 61 data values currently in the "swim_files" list.



What a great idea.

Let's use the combo mambo to see what's built into lists.

What can you do to lists?

Here's the **print dir** combo mambo output for your `swim_files` list:

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```



WATCH IT!



Care is needed with the list built-ins (especially if they change your list).

If you're rubbing your hands in glee at the prospect of using the **sort** *method to reorder your list, take a step back. The* **sort** *method performs it's reordering "in-place", which means the new order* **overwrites** *(!!) what was previously in the list. The old list ordering is lost forever... and there's no undo.*

If you'd like to maintain the existing ordering in your list but still need to sort, there's a BIF which can help here. The **sorted** *BIF returns a ordered copy of your list's data, all the while leaving the existing list ordering intact. There's no undo here, either, as your original list is unchanged.*

What's this then?

```
print(sorted(swim_files))
```

```
['.DS_Store', 'Abi-10-100m-Back.txt', 'Abi-10-100m-Breast.txt', 'Abi-10-50m-Back.txt', 'Abi-10-50m-
Breast.txt', 'Abi-10-50m-Free.txt', 'Ali-12-100m-Back.txt', 'Ali-12-100m-Free.txt', 'Alison-14-100m-
Breast.txt', 'Alison-14-100m-Free.txt', 'Aurora-13-50m-Free.txt', 'Bill-18-100m-Back.txt', 'Bill-18-200m-
Back.txt', 'Blake-15-100m-Back.txt', 'Blake-15-100m-Fly.txt', 'Blake-15-100m-Free.txt', 'Calvin-9-50m-
Back.txt', 'Calvin-9-50m-Fly.txt', 'Calvin-9-50m-Free.txt', 'Carl-15-100m-Back.txt', 'Chris-17-100m-
Back.txt', 'Chris-17-100m-Breast.txt', 'Darius-13-100m-Back.txt', 'Darius-13-100m-Breast.txt', 'Darius-13-
100m-Fly.txt', 'Darius-13-200m-IM.txt', 'Dave-17-100m-Free.txt', 'Dave-17-200m-Back.txt', 'Elba-14-100m-
Free.txt', 'Emma-13-100m-Breast.txt', 'Emma-13-100m-Free.txt', 'Erika-15-100m-Breast.txt', 'Erika-15-100m-
Free.txt', 'Erika-15-200m-Breast.txt', 'Hannah-13-100m-Back.txt', 'Hannah-13-100m-Free.txt', 'Katie-9-100m-
Back.txt', 'Katie-9-100m-Breast.txt', 'Katie-9-100m-Free.txt', 'Katie-9-100m-IM.txt', 'Katie-9-50m-
Back.txt', 'Katie-9-50m-Breast.txt', 'Katie-9-50m-Fly.txt', 'Katie-9-50m-Free.txt', 'Lizzie-14-100m-
Back.txt', 'Lizzie-14-100m-Free.txt', 'Maria-9-50m-Free.txt', 'Mike-15-100m-Back.txt', 'Mike-15-100m-
Fly.txt', 'Mike-15-100m-Free.txt', 'Mike-15-200m-Free.txt', 'Mike-15-200m-IM.txt', 'Owen-15-100m-Free.txt',
'Ruth-13-100m-Back.txt', 'Ruth-13-100m-Free.txt', 'Ruth-13-200m-Back.txt', 'Ruth-13-200m-Free.txt', 'Ruth-
13-400m-Free.txt', 'Tasmin-15-100m-Back.txt', 'Tasmin-15-100m-Breast.txt', 'Tasmin-15-100m-Free.txt',
'Tasmin-15-200m-Breast.txt']
```



Yes, that's a potential issue.

As the `swimdata.zip` file was initially created on a Mac, the `.DS_Store` file was automatically added to the ZIP archive. This type of OS-specific issue is always a concern.

Before moving on, it's important to *remove* that unwanted filename from the `swim_files` list.

EXERCISE



Here's the list of built-in methods available when working with lists:

```
'append', 'clear', 'copy', 'count', 'extend', 'i
'insert', 'pop', 'remove', 'reverse', 'sort'
```

You already know what the **append** and **sort** methods do, but what of the others?

Using the **help** BIF, spend some time in your notebook to learn what some of these other methods do. Your goal is to identify the method to use to delete that unwanted `.DS_Store` filename from your list. When you think you've identified a method, write its name in the space below:

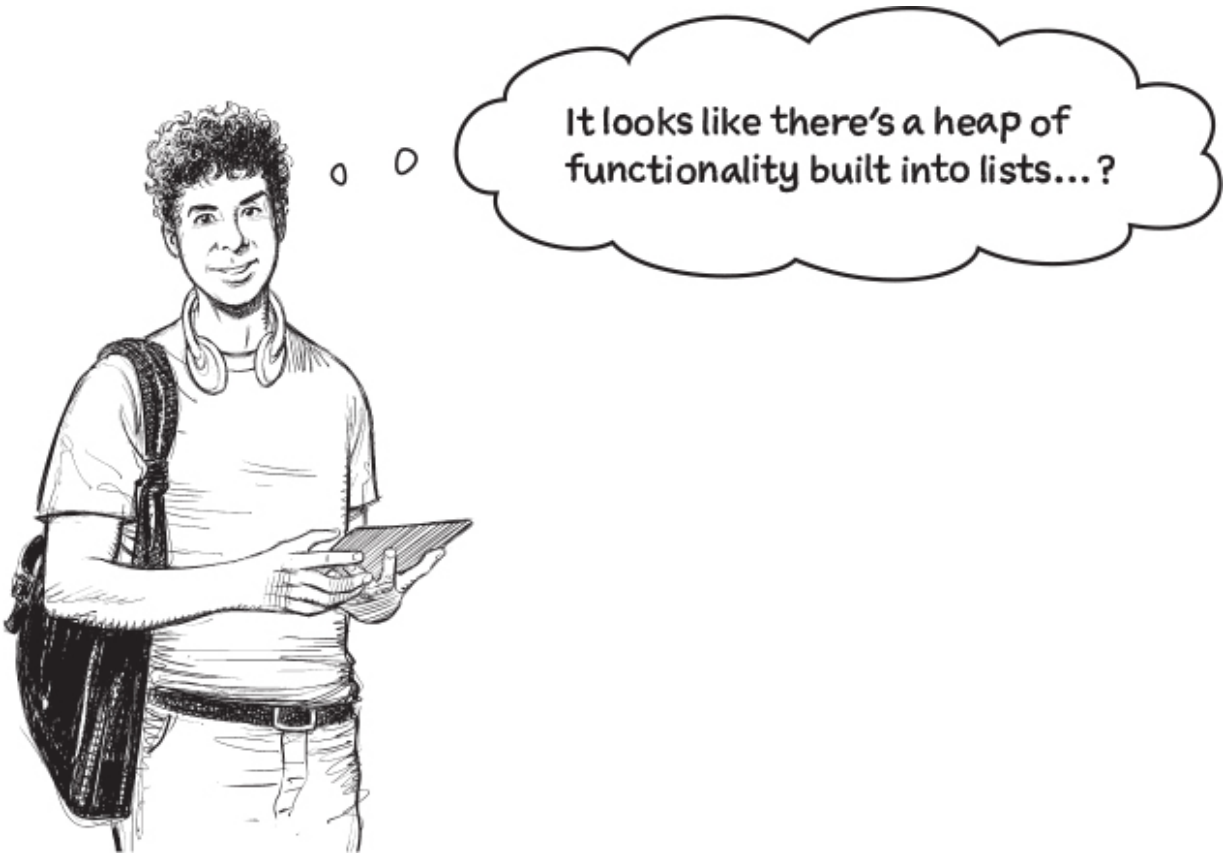
_____

_____

RELAX



Don’t stress yourself if, as a result of completing the above exercise, you identify more than one way to accomplish the assigned task. This is OK, as sometimes the same outcome can be accomplished in many different ways. There’s rarely an absolutely right way to do something. As with most things, you should concentrate on getting your code to do what you need it to do *first* before attempting any sort of optimization.



There sure is.

You'll get to see most of the list built-in methods in action as you progress though this book. However, for now, concentrate on just enough to complete the exercise.

EXERCISE SOLUTION



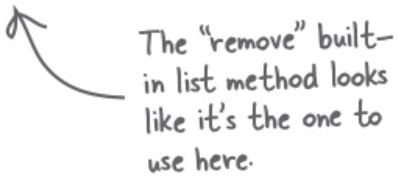
You were shown the list of built-in methods available when working with lists:

```
'append', 'clear', 'copy', 'count', 'extend', 'i
'insert', 'pop', 'remove', 'reverse', 'sort'
```

You already knew what the **append** and **sort** methods do, but what of the others?

Using the **help** BIF, you were to spend some time in your notebook to learn what some of these other methods do. Your goal was to identify the method to use to delete that unwanted `.DS_Store` filename from your list. You were to write the method's name in the space below:

remove


The "remove" built-in list method looks like it's the one to use here.

THERE ARE NO DUMB QUESTIONS

Q: If I unzip `swimdata.zip` on something other than a Mac, will I still see that Mac-specific `.DS_Store`?

A: Unfortunately, yes. The ZIP archive was created on an Apple device, so the `.DS_Store` file is always going to be there.

Q: Can't we avoid this issue by asking the Coach to use something other than a Mac for this?

A: We asked, and the Coach told us he'd rather eat glass...

Q: I can see the relationship between tuples and lists, in that they are somewhat similar. I guess a list is also a sequence, but it is immutable?

A: Lists are indeed a sequence, but – no – lists are not immutable. Rather, lists are *mutable*, in that they can dynamically change as your code runs (unlike tuples).

Q: I guess lists are way more useful than tuples, right?

A: It depends. Both lists and tuples have their uses. Overall, lists see more action than tuples, in that lists support many more use-cases. But it's not the case that lists are "better" or "more useful" than tuples: lists and tuples are designed for different purposes.

TEST DRIVE



Of the eleven methods built-into every Python list, the one which caught our eye was **remove**.

Here's what the **help** BIF reported about **remove**:

```
help(swim_files.remove)
```

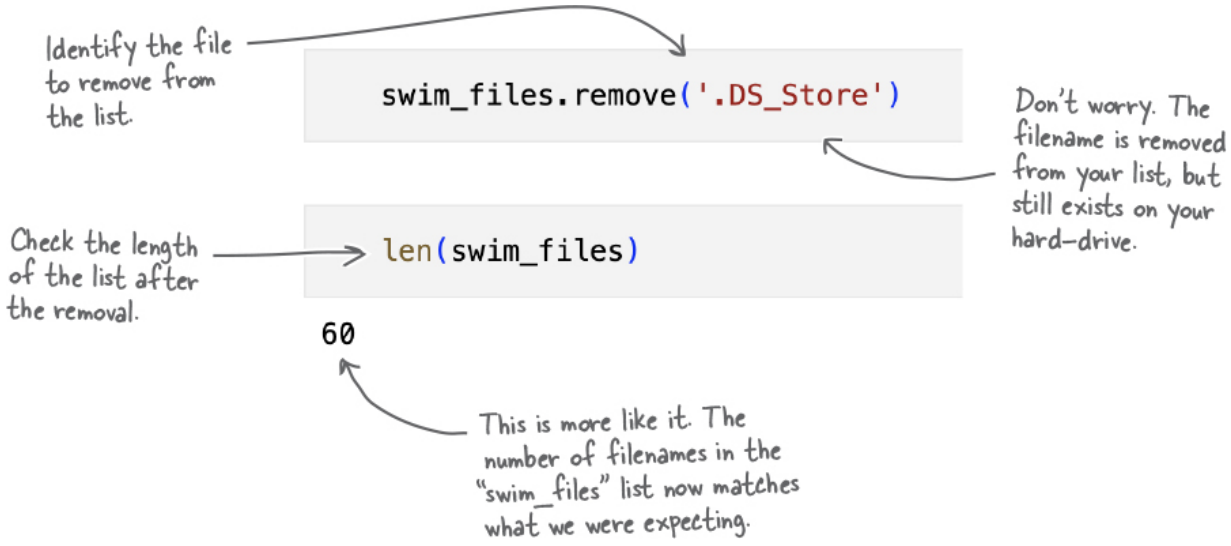
```
Help on built-in function remove:

remove(value, /) method of builtins.list instance
    Remove first occurrence of value.

    Raises ValueError if the value is not present.
```

This method looks like what we need.

Sure enough, when the **remove** method is invoked against the **swim_files** list (with `.DS_Store` given as the filename ear-marked for the chop) the list *shrinks* from 61 items to 60:


Identify the file to remove from the list.
swim_files.remove('.DS_Store')
Don't worry. The filename is removed from your list, but still exists on your hard-drive.
Check the length of the list after the removal.
len(swim_files)
60
This is more like it. The number of filenames in the "swim_files" list now matches what we were expecting.

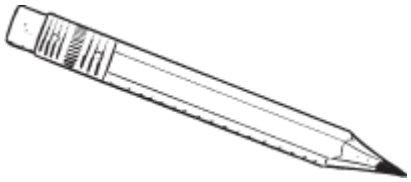


That would be nice, wouldn't it?

We could throw caution to the wind and dive into creating some bar charts, but it might be too soon for that.

Your `get_swim_data` function has worked so far, but can you be sure it'll work for *any* swimmer's file? Let's spend a moment ensuring our `get_swim_data` function works as expected no matter the data file it's presented with.

SHARPEN YOUR PENCIL



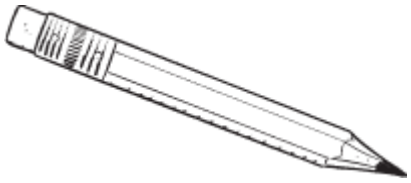
Let's see if you can confirm your `get_swim_data` function works with any swimmer's file. Write a **for** loop which processes the file-names in `swim_files` one at a time. On each loop iteration, display the name of the file currently being processed, then arrange to call your `get_swim_data` function with the current filename. You do not need to display the data returned from your function, but you do need to call it.

Write the code you used here and note down anything your learned from running your **for** loop in the space below:

Put your code here.

Put your notes here.

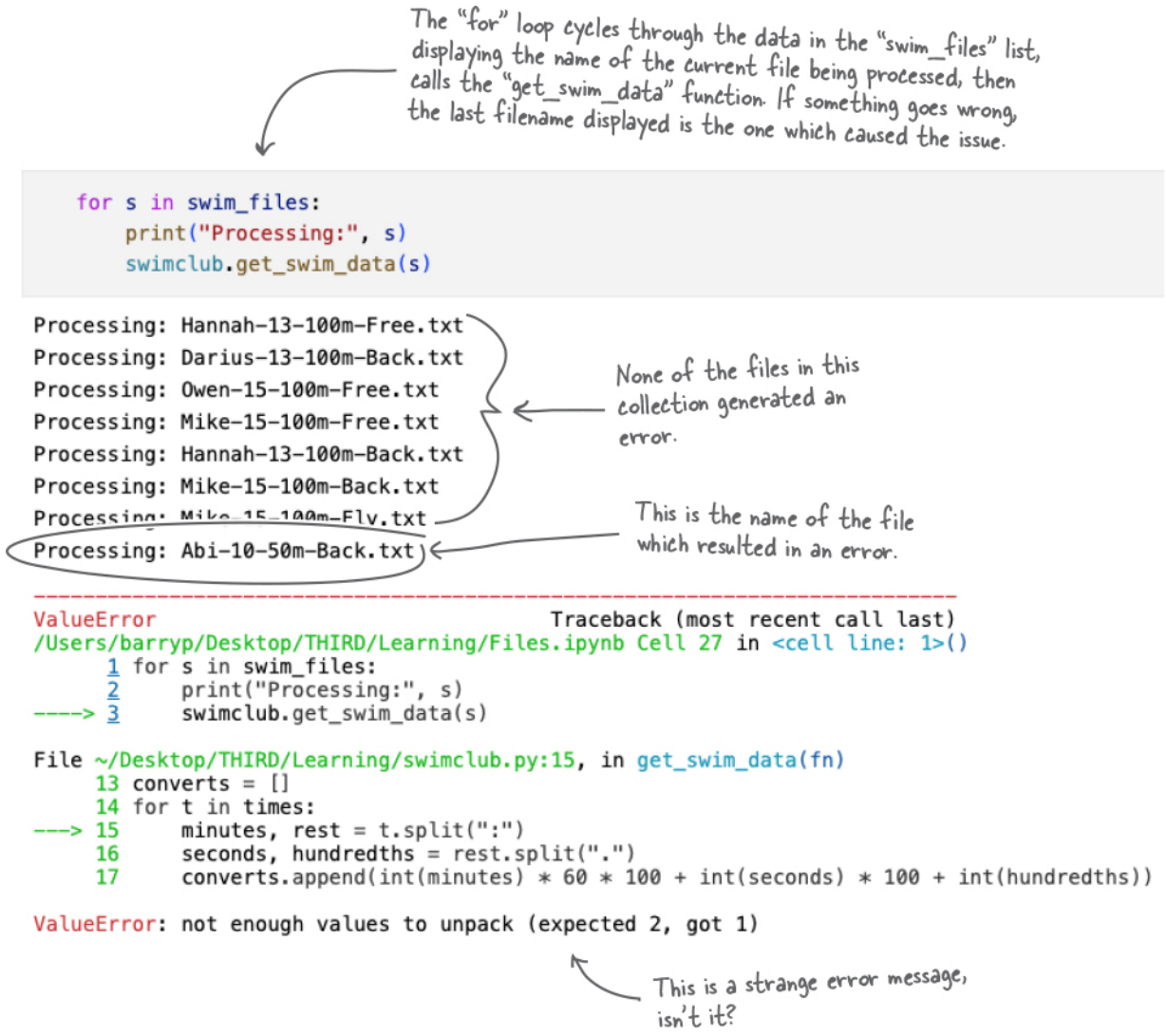
SHARPEN YOUR PENCIL SOLUTION



You were to try out your `get_swim_data` function on all the files in your `swim_files` list..

You were to write a **for** loop which processes the filenames in `swim_files` one at a time. On each iteration, you were to display the name of the file currently being processed, then arrange to call your `get_swim_data` function with the current filename. You didn't need to display the data returned from your function, but you do need to call it.

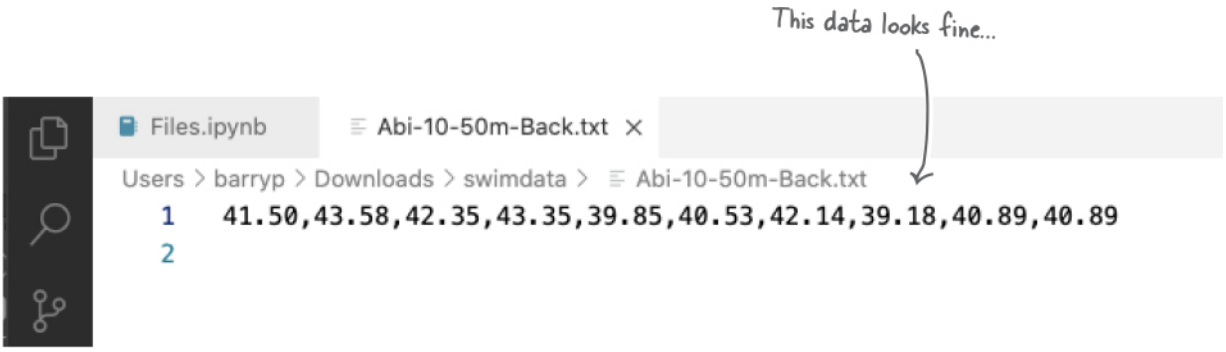
Here's the code which we came up with after experimenting in our `Files.ipynb` notebook, together with the output generated by the code:



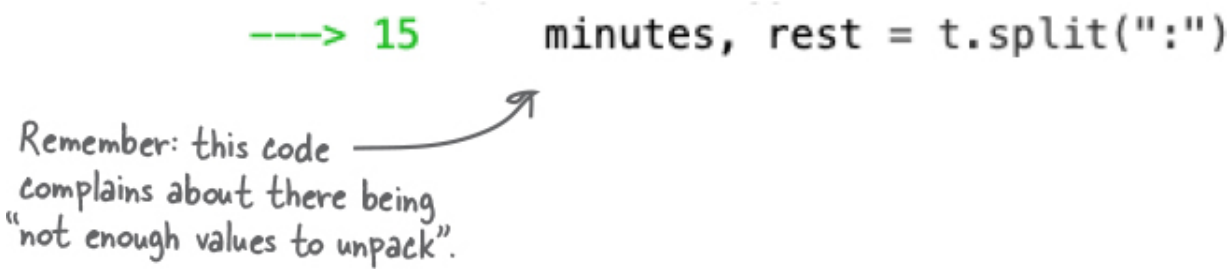
Is the issue with your data or your code?

Now that you've identified the offending file, let's take a look at its contents to see if you can get to the root of the problem. Here's the

`Abi-10-50m-Back.txt` file opened in VS Code:



Here's the line of code which is throwing the error. Can you see what the issue is?





An incorrect assumption is the problem.

Your code, as written, assumes every swim time conforms to the *mins:secs.hundredths* format, but this is clearly not the case with Abi's 50m swim times, and this is why you're getting that `ValueError`.

Now that you know what the problem is, what's the solution?

Cubicle Conversation

Sam: What are our options here?

Alex: We could fix the data, right?

Mara: How so?

Alex: We could pre-process each data file to make sure there's no missing minutes, perhaps by prefixing a zero and a colon when the minutes are missing? That way, we won't have to change any code.

Mara: That would work, but...

Sam: ...it would be messy. Also, I'm not too keen to preprocessing all the files, as the vast majority won't need to be changed, which feels like it might be wasteful.

Mara: And although, as a strategy, we wouldn't have to change any existing code, we would have to create the code to do the pre-processing, perhaps as a separate utility.

Sam: Recall, too, that the data is in a fixed format, and that it's generated by the Coach's smart stopwatch. We really shouldn't mess with the data, so let's leave it as is.

Alex: So, we're looking at changing our `get_swim_data` function, then?

Mara: Yes, I think that's a better strategy.

Sam: Me, too.

Alex: So, what do we need to do?

Mara: We need to identify where in our code the changes need to be made...

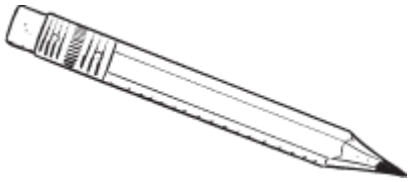
Sam: ...and what those code changes need to be.

Alex: OK, sounds good. So were's going to take a closer look at our `get_swim_data` function so we can decide what code needs to change?

Mara: Yes, then we can use an **if** statement to make a decision based on whether or not the swim time currently being processed has a minute value.



SHARPEN YOUR PENCIL



Here's the most recent version of the code from your `swimclub.py` module.

Grab your pencil and encircle the area of the code you think needs to incorporate an **if** statement:

```
import statistics

FOLDER = "swimdata/"

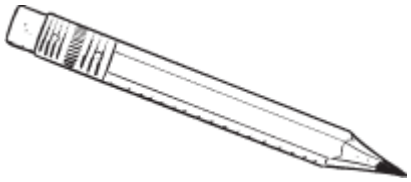

def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, average, average_str, times
```

NOTE

Flip the page to see our selection. 

SHARPEN YOUR PENCIL SOLUTION



You were shown the most recent version of the code from your `swimclub.py` module.

Grabbing your pencil, you were asked to encircle the area of the code you think needs to incorporate an **if** statement:

```
import statistics

FOLDER = "swimdata/"


def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, average, average_str, times
```

This is where we think you need to put the "if" statement, as you need to decide which of the two swim time formats you're currently processing.

Decisions, decisions, decisions

That's what **if** statements do, day-in and day-out: they make decisions.



Yes, that is what's needed here.

Let's take a closer look at the two possible swim time formats.

First up, here is one of the times recorded for Darius in his file:

```
'1:30.96'
```

And here's a time taken from Abi's data:

```
'43.35'
```

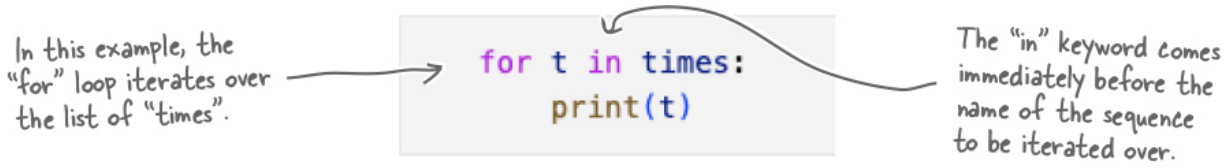
It's easy to spot the difference: *Abi's data doesn't show any minutes*. With this in mind, it's possible to come up with a condition to check when making a decision. Can you work out what it is? (Hint: consider your BFF, the colon).

Let's look for the colon "in" the string

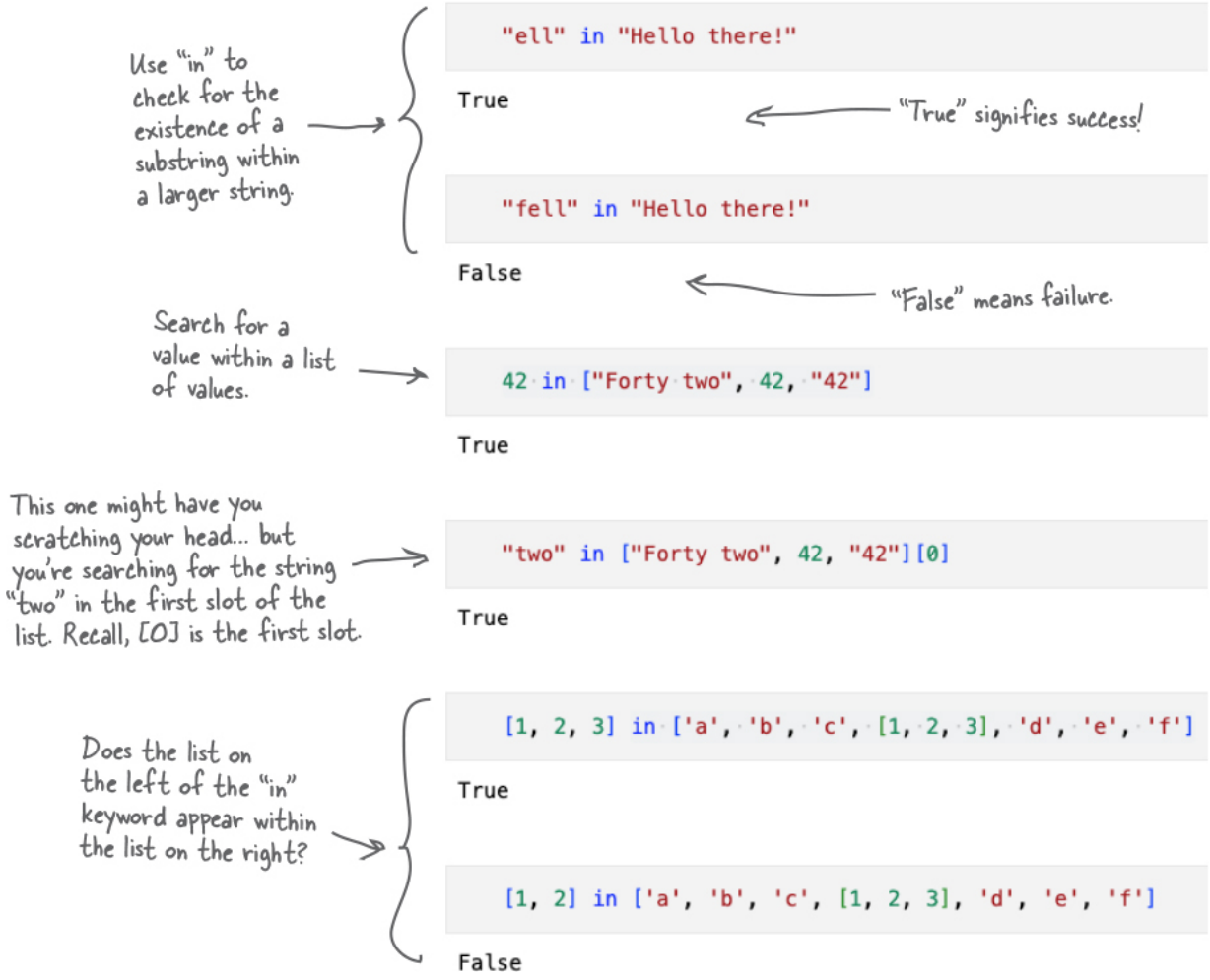
If the colon appears in any swim time string, then the time has a minute value. Although strings come with lots of built-in methods, including methods which can perform a search, let's not use any of these here. As searching is such a common requirement, Python provides the **in** operator. You've seen **in** before, with **for**:

NOTE

The "find" and "index" string methods both perform searching.



Using **in** with **for** identifies the sequence being iterated over. However, when **in** is used outside a loop it takes on *searching* powers. Consider these example uses of **in**:





We love the `in` keyword, too.

It's a Python superpower.

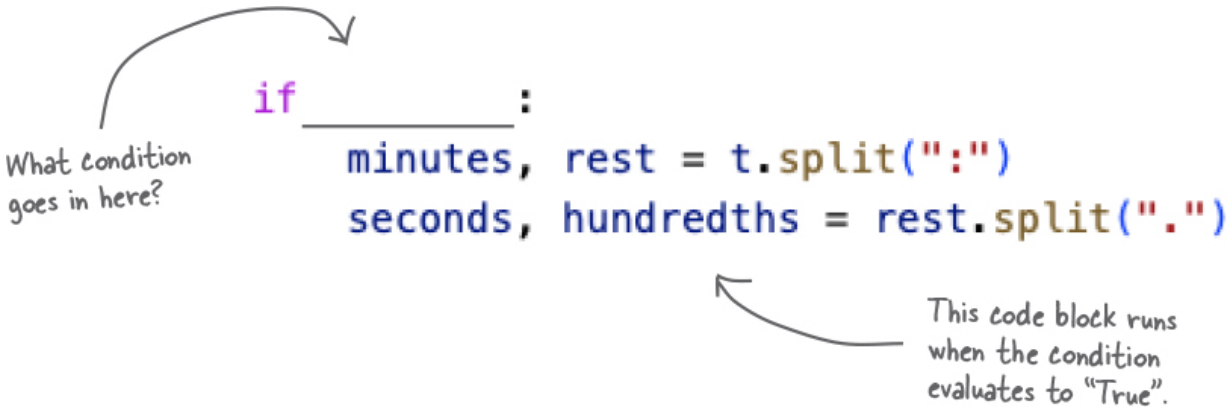
EXERCISE



As previewed in Chapter 0: Why Python? Similar But Different, here's the general structure of a simple **if** statement:

```
if <some condition>:
    Code to execute when <some condition> is True
else:
    Code to execute when <some condition> is Fals
```

Now that you know about the **in** keyword, can you come with a conditional statement which checks to see when the colon appears in the current swim time (which is stored in a variable named `t`)? Fill-in the blank space below:


What condition goes in here?
if _____:
minutes, rest = t.split(":")
seconds, hundredths = rest.split(".")
This code block runs when the condition evaluates to "True".

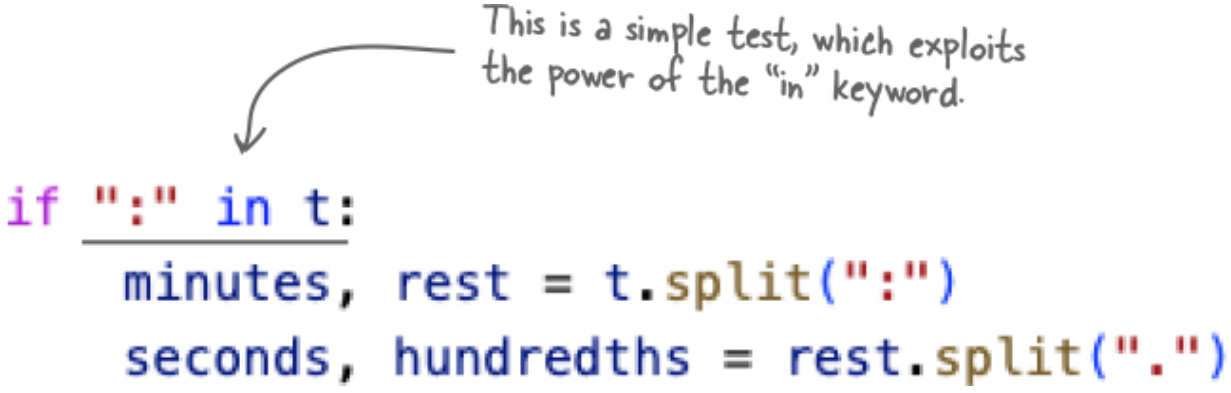
EXERCISE SOLUTION



Here's the general structure of a simple **if** statement:

```
if <some condition>:
   Code to execute when <some condition> is True
else:
   Code to execute when <some condition> is False
```

Now that you know about the **in** keyword, you were asked to you come with a conditional statement which checks to see when the colon appears in the current swim time (which is stored in a variable named `t`)? You were to fill-in the blank space below:



WATCH IT!



Be sure to write your if statements in a Pythonic way

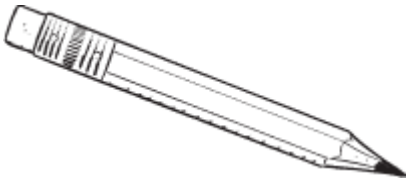
If you are coming to Python from one of the C-like languages, your fingers might insist on coding extra parentheses around the condition. Or perhaps you'll feel the need to explicitly check to see if the condition evaluates to **True** *(or* **False** *). Don't be tempted to do either. The Python interpreter won't stop you doing what's shown below, but you need to resist, resist, resist!*

There's only one word to describe both of these versions of your "if" statement: Yuk!

```
if (":" in t):
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")

if (":" in t) == True:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
```

SHARPEN YOUR PENCIL



There's one final task, and that's to work out the code which runs when the condition evaluates to **False**. Here's your **if** statement so far. Can you come up with the code which needs to be added to the **else** code block? Experiment in your notebook, then add the code below. As always, our solution is over the page.

```
if ":" in t:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
else:
```

Add code in here to execute when the "else" block runs.

THERE ARE NO DUMB QUESTIONS

Q: I'm guessing "True" and "False" are Python's boolean values? Can I use 1 and 0 in their place?

A: Sort of. Within a *boolean context*, `1` evaluates to **True** and `0` evaluates to **False**. That said, Python programmers rarely use `1` and `0` in this way. This has to do with the fact that any value in Python can be used within a boolean context.

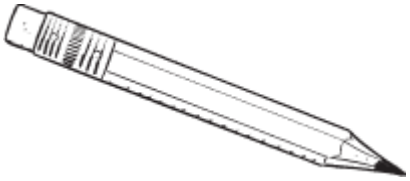
Q: Is there a way to check what a value evaluates to within a boolean context?

A: Yes, there's a BIF for that. It's called **bool**. You can use it interactively (or in your code) to check any value's boolean evaluation.

Q: What about using "true" and "false", i.e., all lowercase? How do those values evaluate?

A: Not the way you might expect. Using `true` and `false` is a bad idea, as case is significant. **True** is a boolean value, as is **False**, whereas `true` and `false` are not. They're valid variable names, *not* booleans. And as variables, they exist, which means they'll always evaluate to **True** (how can something that exists be anything other than **True**?). And, yes, we agree that something called `false` evaluating to **True** is a little on the weird side. Life lesson: don't ever use lowercase `true` nor `false` as variable names.

SHARPEN YOUR PENCIL SOLUTION



You had one final task, and that was to work out the code which runs when the condition evaluates to **False**. You were given your **if** statement so far, and were to come up with the code which needs to be added to the **else** code block. Here's what we used. Is your code similar, the same, or entirely different?

```
if ":" in t:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
else:
    minutes = 0
    seconds, hundredths = t.split(".")
```

When the colon is not found in the time string, the "minutes" value is set to zero.

When there are no minutes recorded, you need to break apart the time string (stored in the "t" variable) on the "." character. This gives you the "seconds" and "hundredths" values.



Absolutely, after a bit of housekeeping.

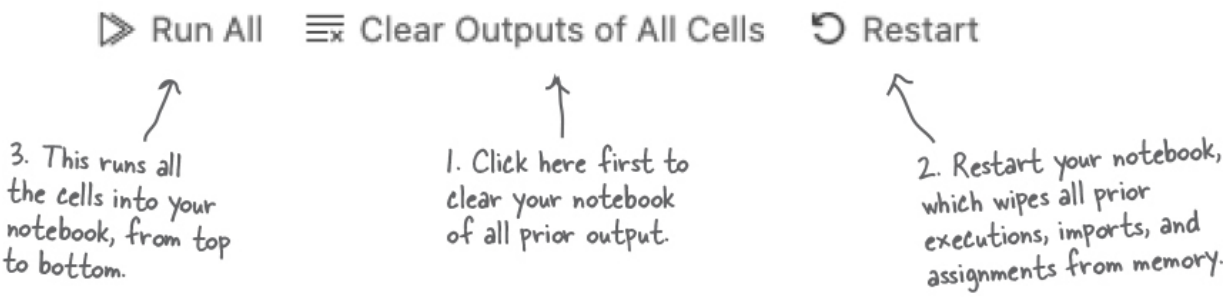
Be sure to add the above code to your `get_swim_data` function within your `swimclub.py` module, and don't forget to **save** your file.

With that done, return to your notebook and follow along with the *Test Drive* on the next page.

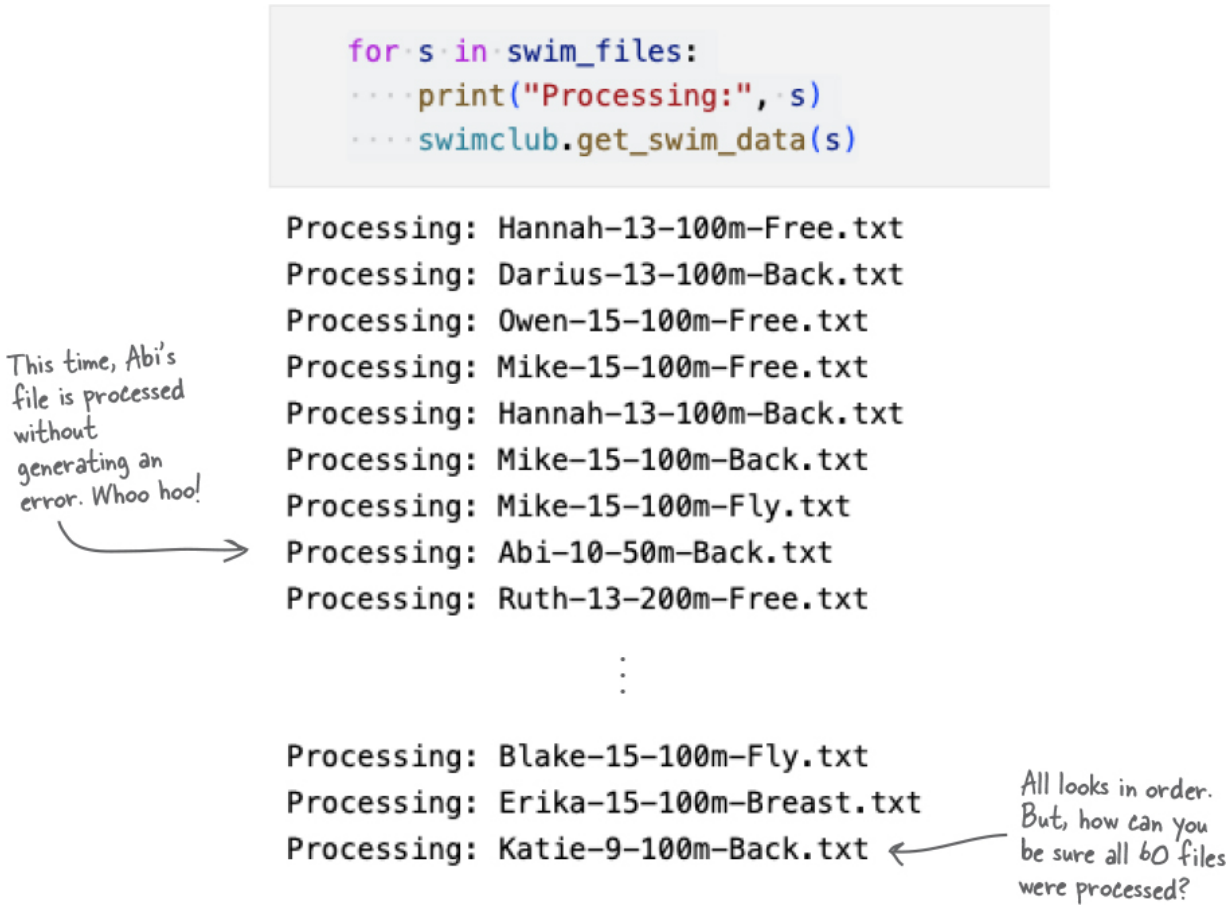
TEST DRIVE



In order to use the latest version of your `swimclub` module, you need to reload it. The easiest way to do this is to clear all of your notebook's cell, restart your notebook, then run every cell. At the top of VS Code, notice the following options, which you should click on in the order shown (1, then 2, and then 3):



Once you complete the three steps above, your **for** loop produces (hopefully) 60 lines of output. This time, hopefully, there is no runtime error:

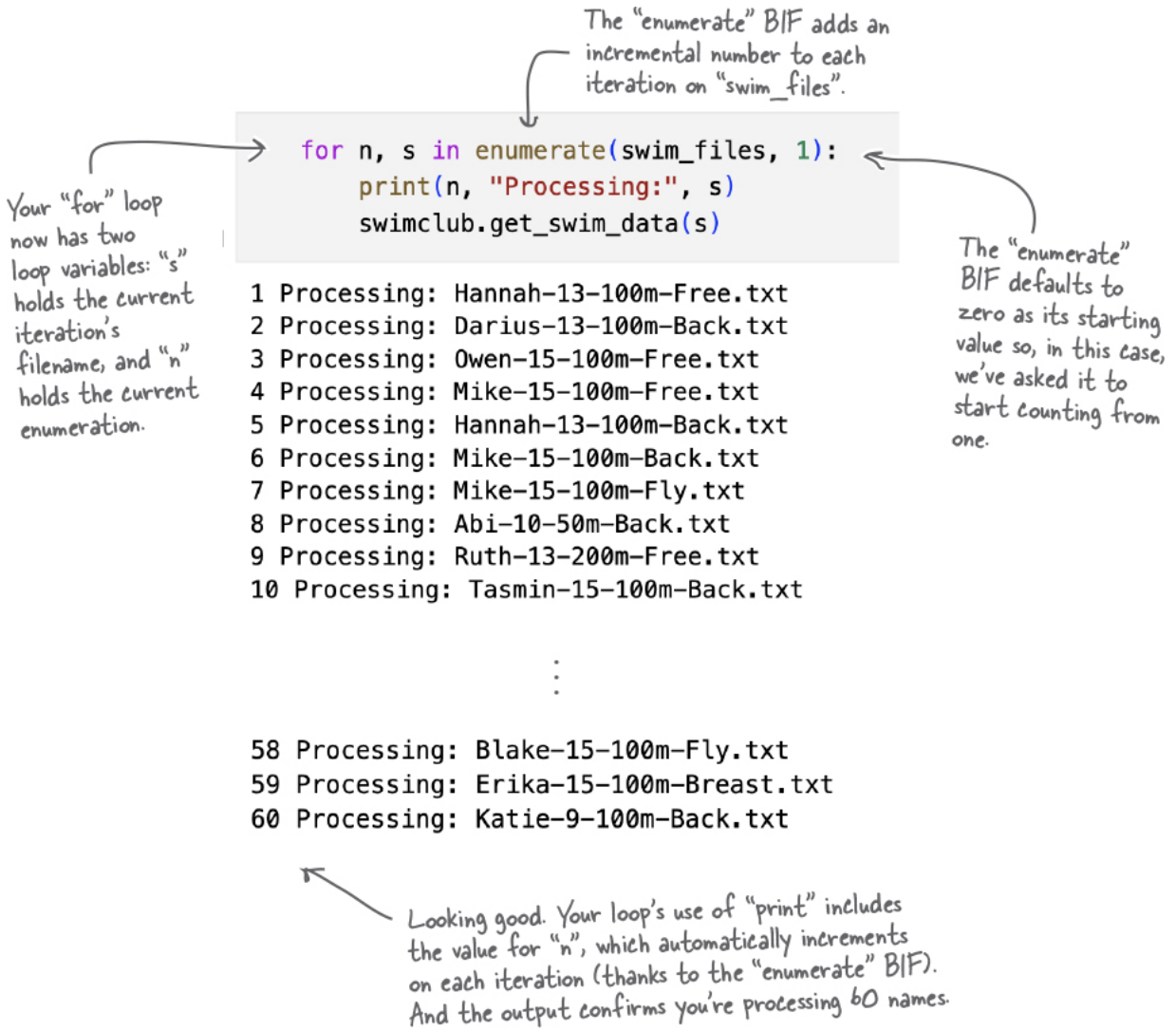


Did you end up with 60 processed files?

You are likely feeling confident that your most recent code is processing all the files in your `swimdata` folder. We are, too. However, it is often nice to double-check these things. As always, there's any number of ways to do this, but let's number the results from your **for** loop

by adding an enumeration, starting from 1, to each line of output displayed on screen.

To do so, let's use yet another BIF created for this very purpose called **enumerate**:



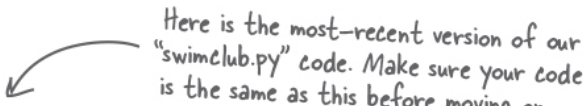
THERE ARE NO DUMB QUESTIONS

Q: This may be nitpicking, but why did we do a restart on our notebook? Surely typing `import swimclub` into an empty cell reloads the previously imported module's newest code?

A: (We hope you're sitting down...) No, it doesn't. The Python interpreter implements an aggressive caching scheme for imported modules which, for all intents and purposes, effectively forbids the re-importation of an already imported module even if the module's code has changed in the meantime. Although there are some techniques for overriding this default module caching behavior, in our experience, the safest thing to do is to *always* restart a notebook after a change to a module's code. This way you are guaranteed to be running the code you think you're running.

```
import statistics

FOLDER = "swimdata/"
```




```
def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    for t in times:
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, average, average_str, times
```

The Coach's code is taking shape...

Your `swimclub` module is now ready. Given the name of a file which contains a collection of swim time strings, your new module can produce usable data. The Coach is expecting to see some bar charts created from this data, so let's dive into implementing that functionality in the next chapter.

As always, you can move on after you've reviewed the chapter summary, then tried your hand at this chapter's crossword.



CHAPTER REVIEW BULLET POINTS

- The **def** keyword defines a new, bespoke function.
- **Triple-quoted** strings are often used to add a multi-line comments to code, and are often found at the top of a function's code block (right after the function signature).
- When you put code in it's own file (with a `.py` extension), you create a **module**.
- The **import** statements lets you reuse a module, e.g., `import swimclub`.
- Use a **fully qualified name** to invoke a function from a module, e.g., `swimclub.get_swim_data`.
- The **return** statement allows a bespoke function to return a result.
- If a function tries to return more than one result, the collection of returned values are **bundled together** as a single **tuple**. This is due to the fact that Python function's only ever return a single result.
- A tuple is an **immutable sequence** data structure. Once a tuple is assigned values, the tuple cannot change.
- Lists are like tuples, expect for the fact that lists are **mutable**.
- The `os` module (included as part of the PSL) lets your Python code talk to your underlying operating system.
- Although lists come with a handy **sort** method, be careful using it as the ordering is applied *in-place*. If you want to keep any list's

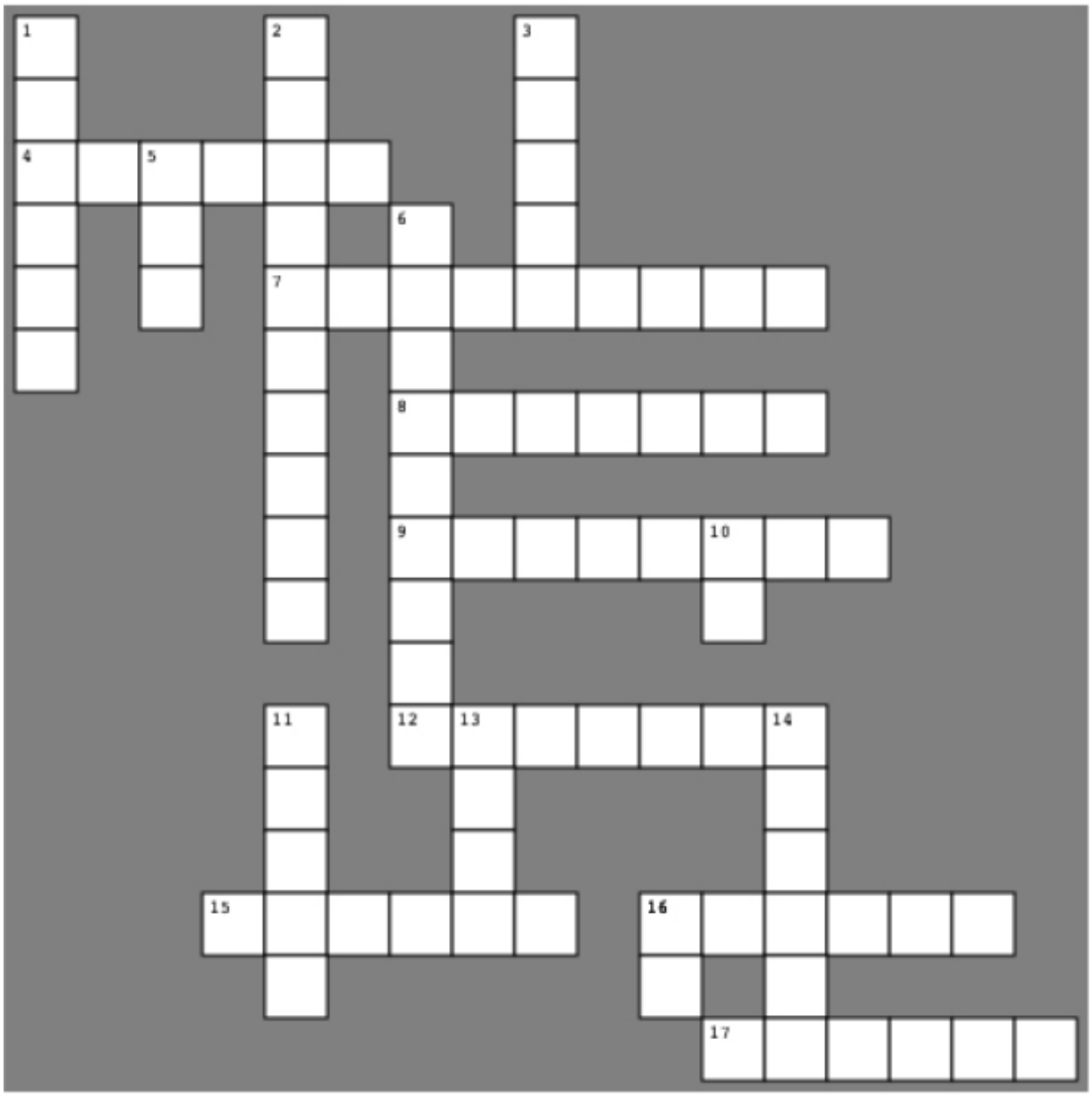
current order, use the **sorted** BIF instead.

- Lists come built-in with lots of methods (not just **sort**), including the useful **remove** method.
- The **in** operator is one of our favorites, and should be one of yours, too. It's great at searching (aka *checking for membership*).
- When you need to make a decision, nothing beats the **if else** combo.
- An often overlooked, but truly wonderful, BIF is **enumerate**. It can be used to number the iterations of any **for** loop.

THE MODULE CROSSWORD



The answers to the clues are found in this chapter's pages, and the solution is, as always, on the next page.



Across

4. A place to put 9 across.

7. A BIF which provides numbers on iteration.

8. From `os` : list a folder's content.

9. A named chunk of code.

12. The underscore is the _____ variable.

15. BIF for applying order to your data (while leaving the current ordering untouched).

16. Loads code from a file.

17. Can start the last line of 9 across.

Down

1. Finds and deletes a value from a list.

2. What you get when you've not enough values to unpack.

3. It's like an immutable list.

5. This keyword introduces 9 across.

6. Be explicit with a fully _____ name.

10. Used when making decisions.

11. Your new BFF.

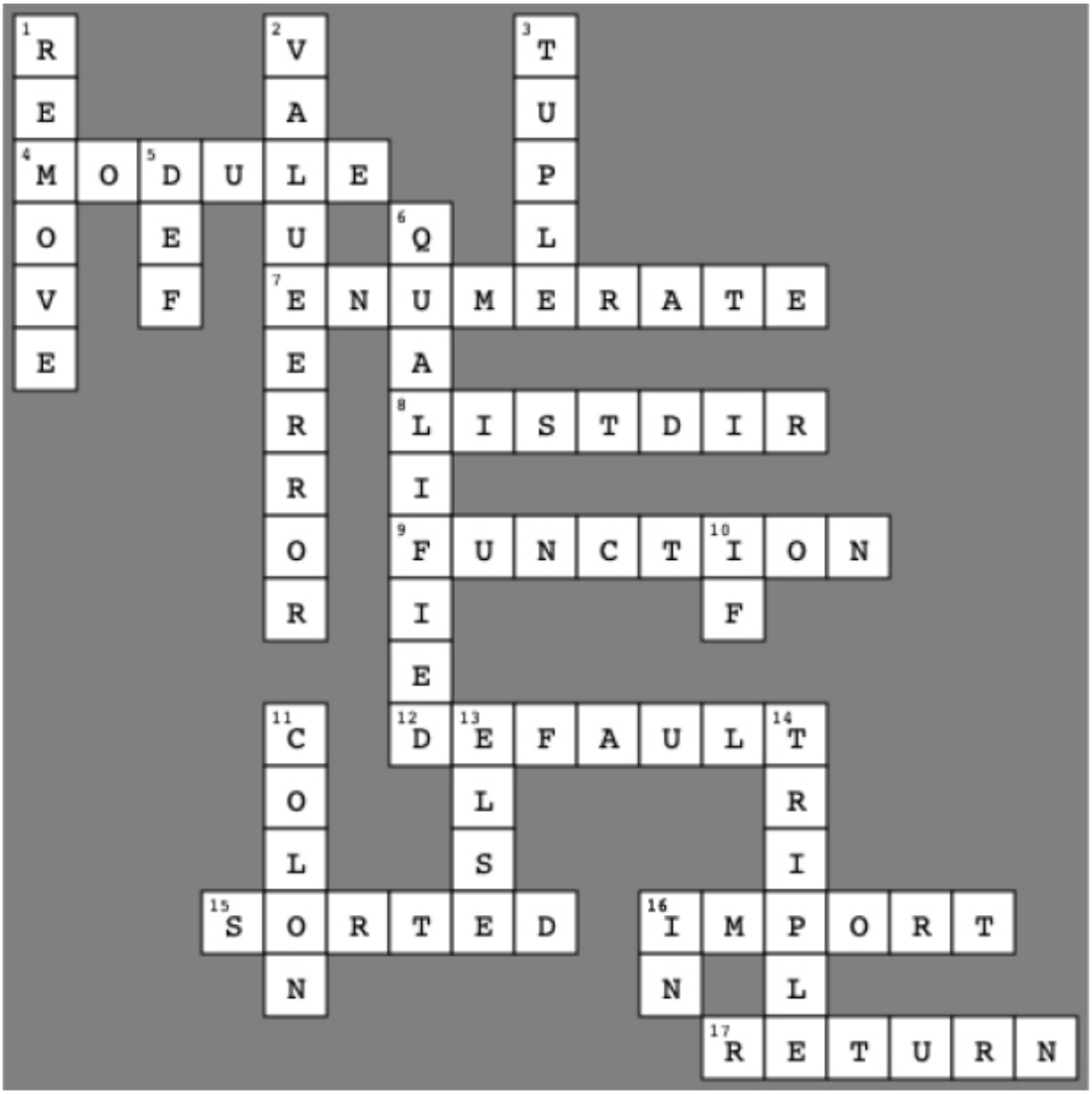
13. The false part of 10 down.

14. _____ quotes, looks like: `" " "` .

16. A powerful little operator.

THE MODULE CROSSWORD SOLUTION





Across

4. A place to put 9 across.

7. A BIF which provides numbers on iteration.

8. From `os` : list a folder's content.

9. A named chunk of code.

12. The underscore is the _____ variable.

15. BIF for applying order to your data (while leaving the current ordering untouched).

16. Loads code from a file.

17. Can start the last line of 9 across.

Down

1. Finds and deletes a value from a list.

2. What you get when you've not enough values to unpack.

3. It's like an immutable list.

5. This keyword introduces 9 across.

6. Be explicit with a fully _____ name.

10. Used when making decisions.

11. Your new BFF.

13. The false part of 10 down.

14. _____ quotes, looks like: `" " "` .

16. A powerful little operator.

Chapter 4. Formatted String Literals: *Make Charts From Data*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 4 of the final book. Please note that the GitHub repo will be made active later on.

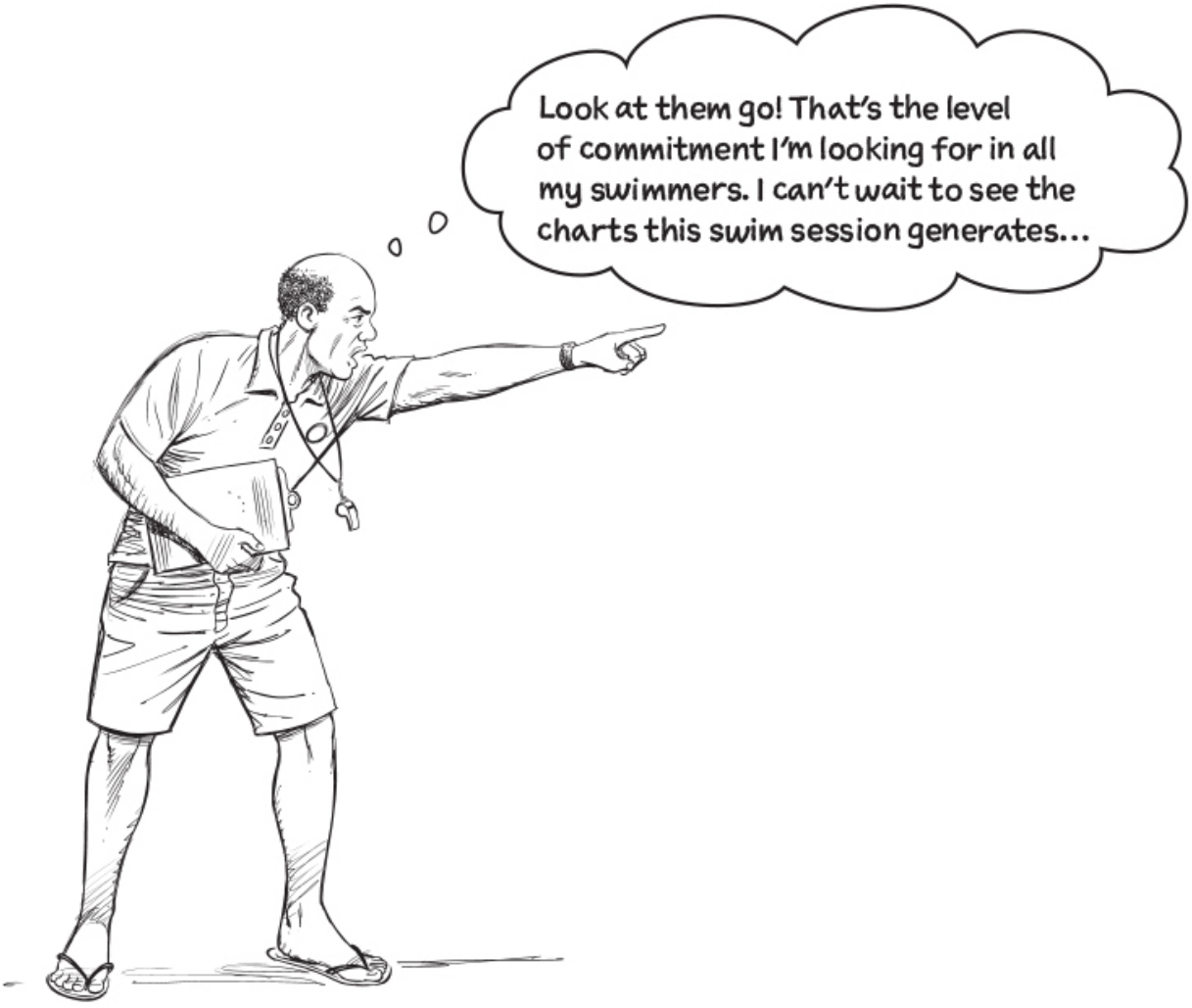
If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



Sometimes it's the simplest approaches that get the job done.

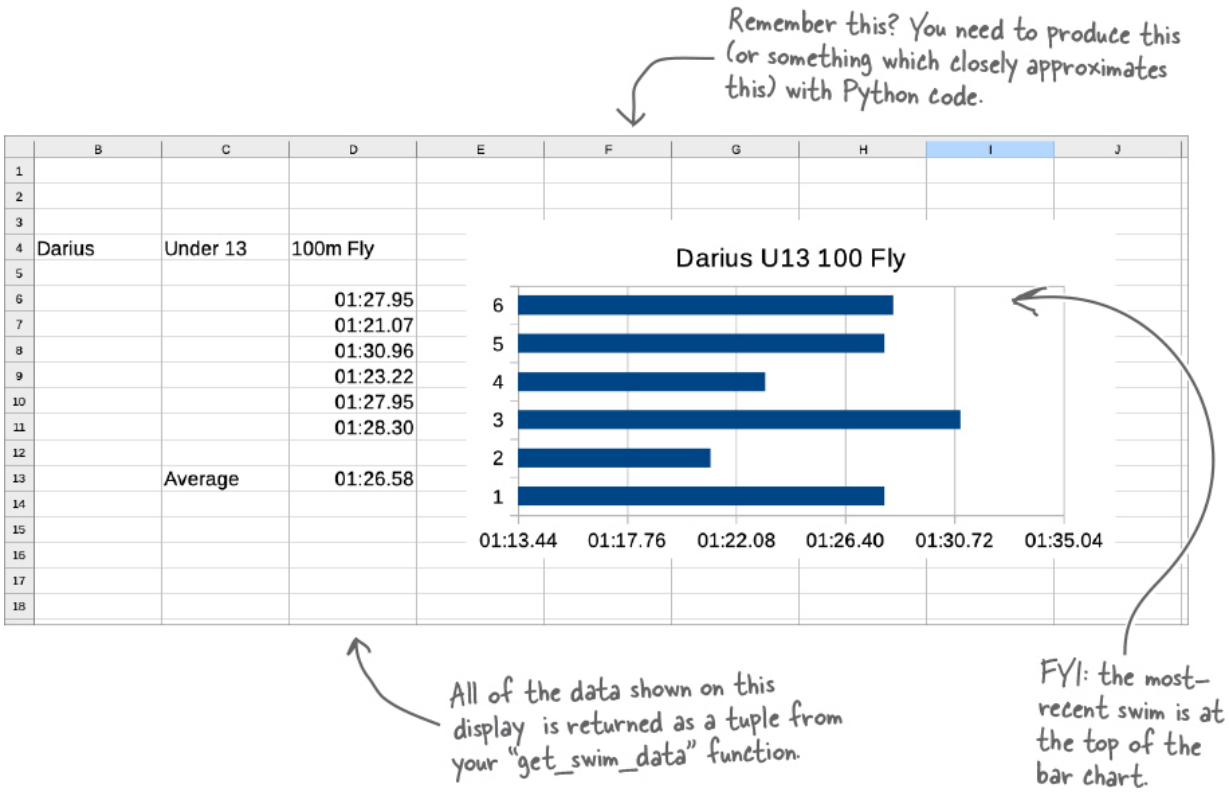
In this chapter you finally get around to producing bar charts for the Coach. You'll do this using nothing but **strings**. You already know that Python's strings come packed full of **built-in goodness**, and the (relatively recent) addition of **formatted string literals**, aka *f-strings*, enhances what's possible in some rather neat ways. It might feel weird that we're proposing creating bar charts with text but, as you're about

to discover, it's not as *absurd* as it sounds. Along the way you'll use Python to create **files** as well as launch a web browser with just a few lines of code. Finally, the Coach is gets his wish: the **automatic generation** of charts from his timing data. Let's get to it!



That's great to hear, as we have all the data we need.

Now all we have to do is work out *how* to turn the data into a bar chart.



Cubicle Conversation

Sam: I just checked on-line, and Python has a heap of visualization libraries on *PyPI*. There's Plotly, Altair, Seaborn, Bokeh, and Matplotlib, to name but a few.

Alex: Hmmm… but which do we choose?

Mara: I'd be a little worried about getting side-tracked. The goal is not to spend time working out which of the many choices are a best fit. The goal is to quickly produce the Coach's bar chart. We need to keep things as simple as possible.

Alex: I'm all for keeping it simple.

Sam: And I'm all for making it look good.

Mara: And using one of those libraries will likely help with both requirements once you select the "best" one, which might be a project in itself.

Sam: Seriously? How long can it possibly take?

Mara: Well… that's the problem. It's hard to know.

Alex: And we are under some pressure here.

Mara: Agreed. The Coach loves what we've done so far, but is impatient for bar charts!

Sam: All right. Let's decide not to use any of these libraries. Is there anything in the PSL?

Mara: Not really. The standard library doesn't have anything specific, but we might not even need to use a visualization library at all.

Alex: Ehhh… really? How come?

Mara: We could mock-up something quickly in HTML, then refine things later?

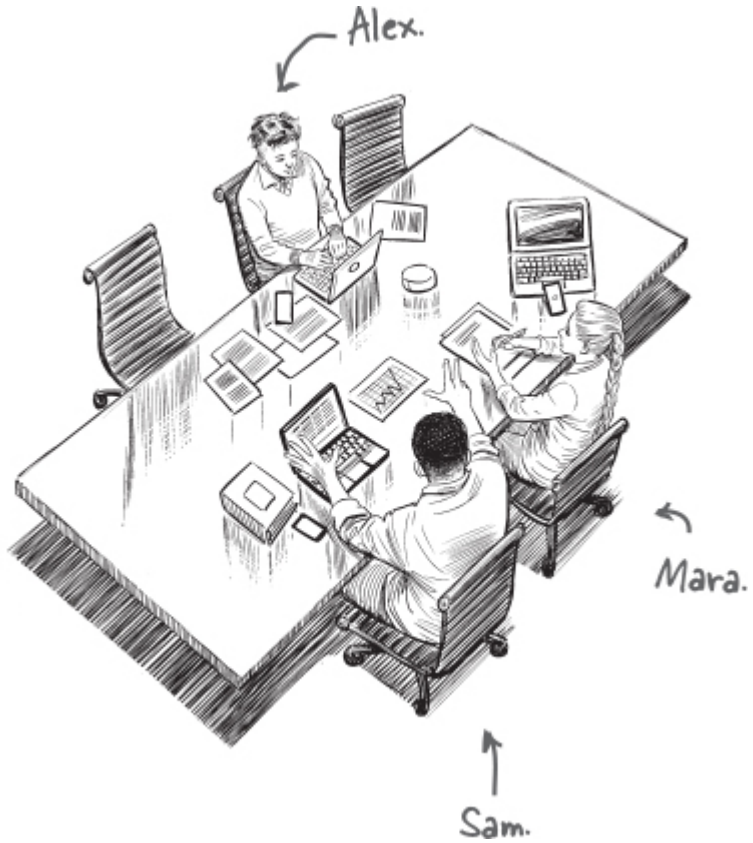
Sam: Get out of town! HTML? Pull the other one.

Mara: Well… a combination of HTML and SVG.

Alex: S…V…G… I just checking on-line… it stands for *"Scalable Vector Graphics"* which is *"an XML-based vector image format for defining two-dimensional images"*. Is this what we want?

Mara: Works for me.

Sam: I'm sorry, but I'll need to see what you mean before I agree to using HTML with SVG.

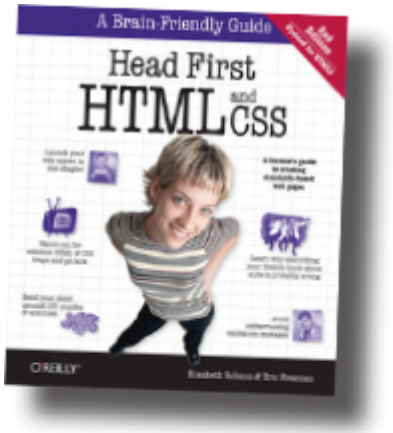


Create simple bar charts with HTML and SVG



Now, before anyone has a panic attack, note that it's not this book's intention to act as a primer for either HTML or SVG. The assumption is that you know enough HTML to be *dangerous*, in that you can create a basic webpage and know your `<head>` tags from your `<p>` tags. If this is not the case, pause this book now and bone-up on the basics of HTML before continuing (there are lots of resources available).

A further assumption is that you know next to *nothing* about SVG. The *Head First Coders* have provided a small example file which, they tell us, contains all you need to know about creating a simple bar chart with HTML and SVG. Take a look, below.





This single HTML file contains HTML markup *as well as* some SVG. Create and save this file as `bar.html` in our `Learning` folder (or you can download it from this book's support website to save you from having to type this markup into your editor).



Launch a web browser from with Python

You could manually launch your favorite web browser then load the `bar.html` file using your browser's **File**… **Open**… menu option, but where's the fun in that? Why not *automatically* launch your browser to view your HTML page directly from your Python code?

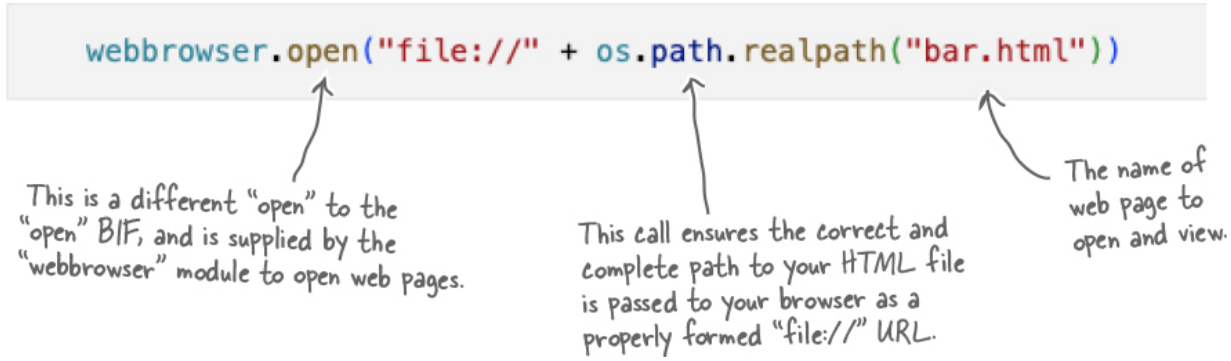
Begin by creating yet another new notebook in your `Learning` folder. Call this one `Charts.ipynb`. Next, import two libraries from the PSL: `os` (which you've already been exposed to and talks to your

underlying operating system) and `webbrowser` (which let's you control your favorite browser with Python code):



With the modules imported, let's use both to open the `bar.html` web page inside your favorite web browser.

If your browser is already open, `bar.html` appears in a new tab. If you browser is not currently opened, Python launches your browser then loads `bar.html`. A single line of Python code does all this heavy lifting:



And, as if by magic, you're webpage appears…

TEST DRIVE



When you run the code in your `Charts.ipynb` notebook, the cell which contains the call to `webbrowser.open` responds with a "helpful" **True**:

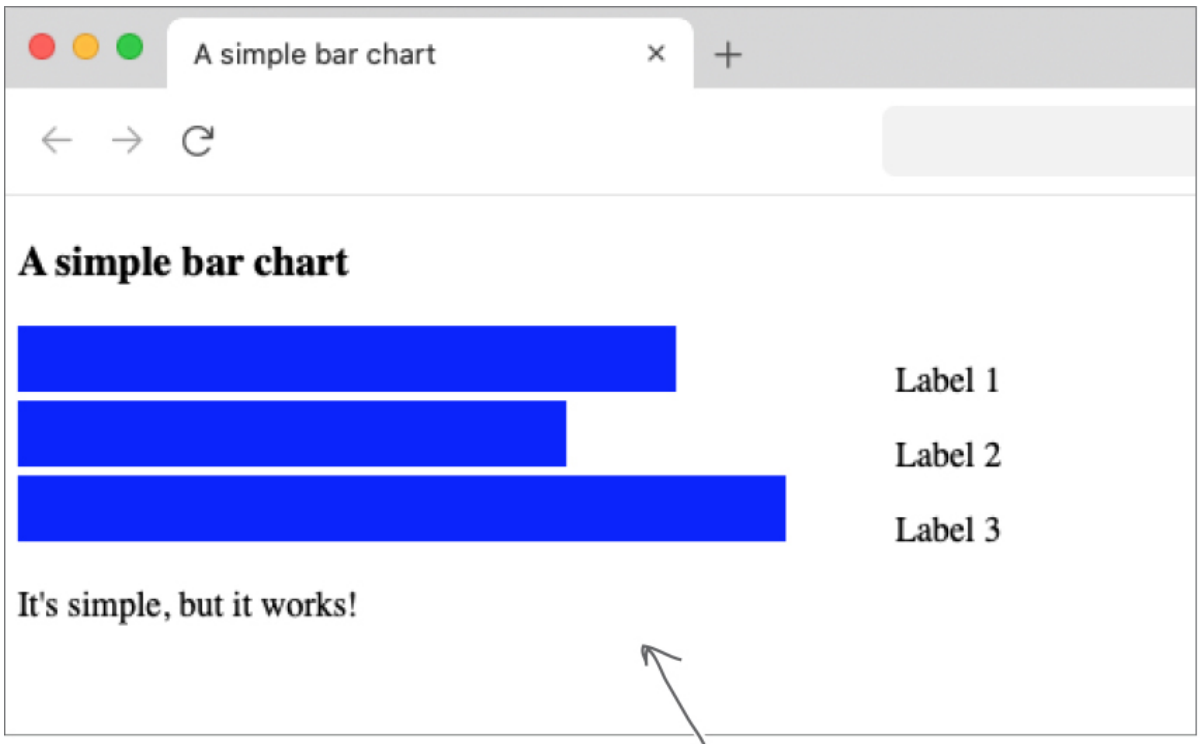
```
import os
import webbrowser
```

```
webbrowser.open("file://" + os.path.realpath("bar.html"))
```

```
True
```

This is confirmation the call to "open" succeeded.

Now, take a look at your web browser (which may be running in a background tab):



This page won't win any HTML design awards, but it is a reasonable approximation of what the Coach wants to see (albeit without any real data).

Let's match up your HTML and SVG to the output you see on screen:



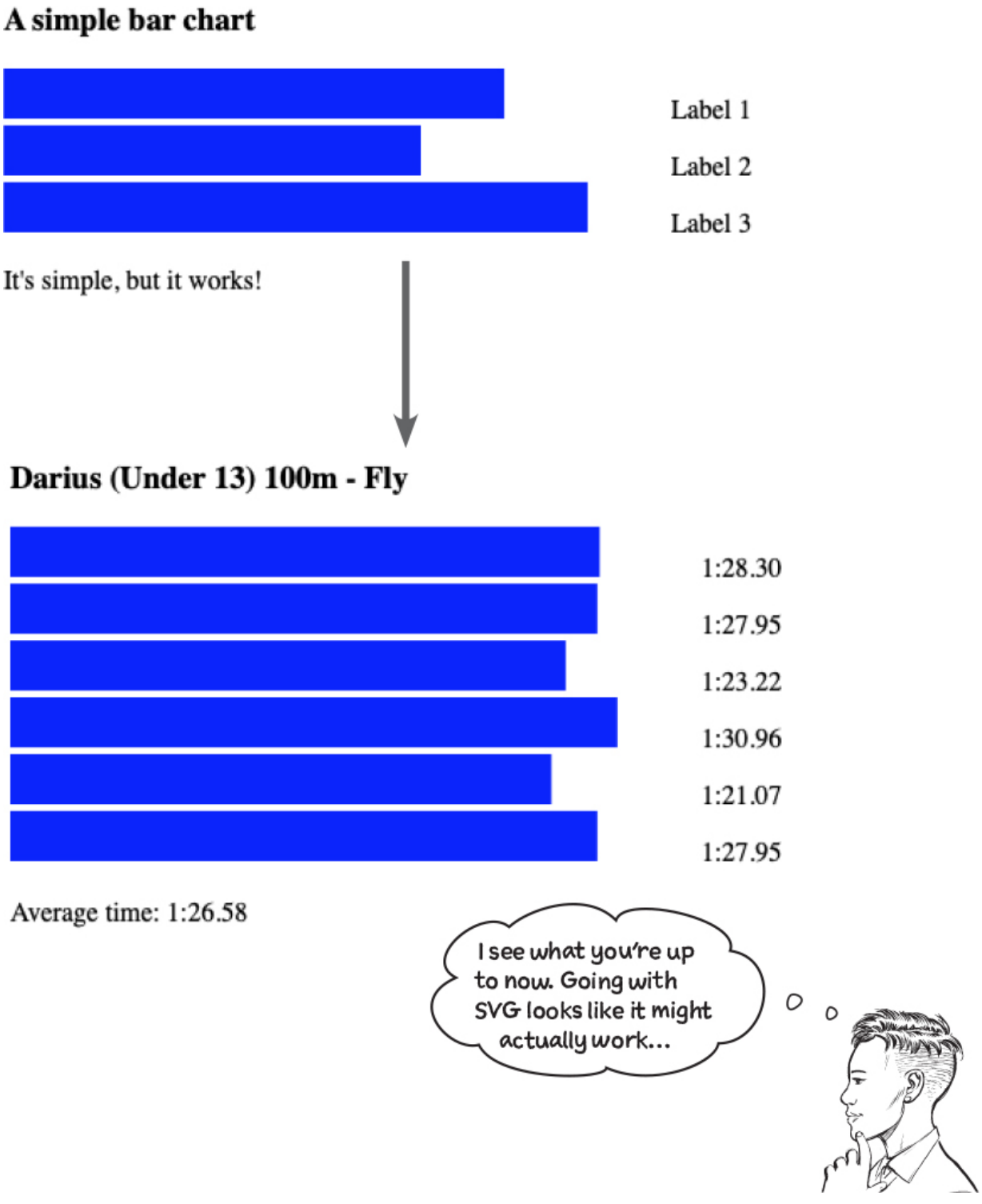
Behind the Scenes



Getting from a simple chart to a Coach chart

Hopefully you're beginning to see where we're going with this. The simple chart in `bar.html` is not a million miles away from a chart

you need to satisfy the Coach's requirement:



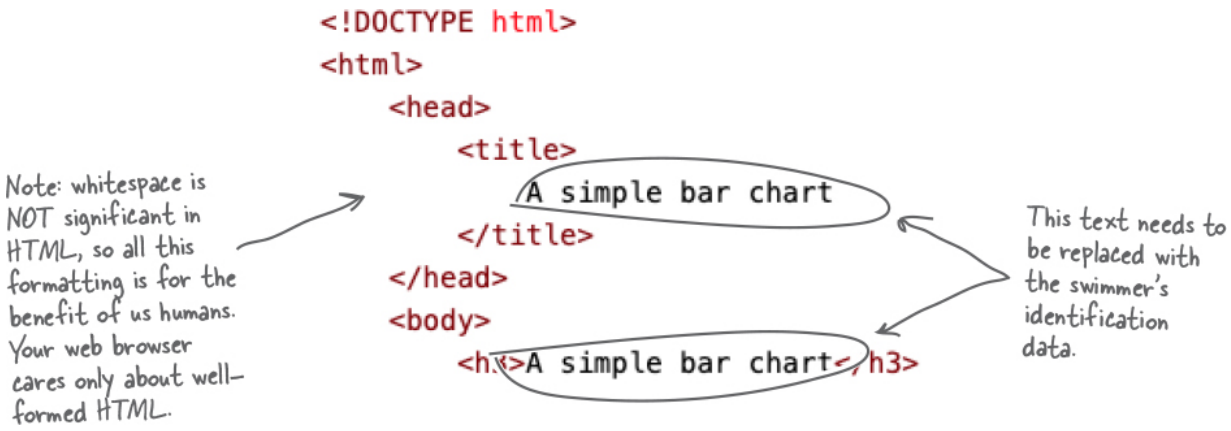


You're right!

We can write code to generate the HTML and SVG *on-the-fly*. And you'll use Python to help you do this as efficiently as possible.

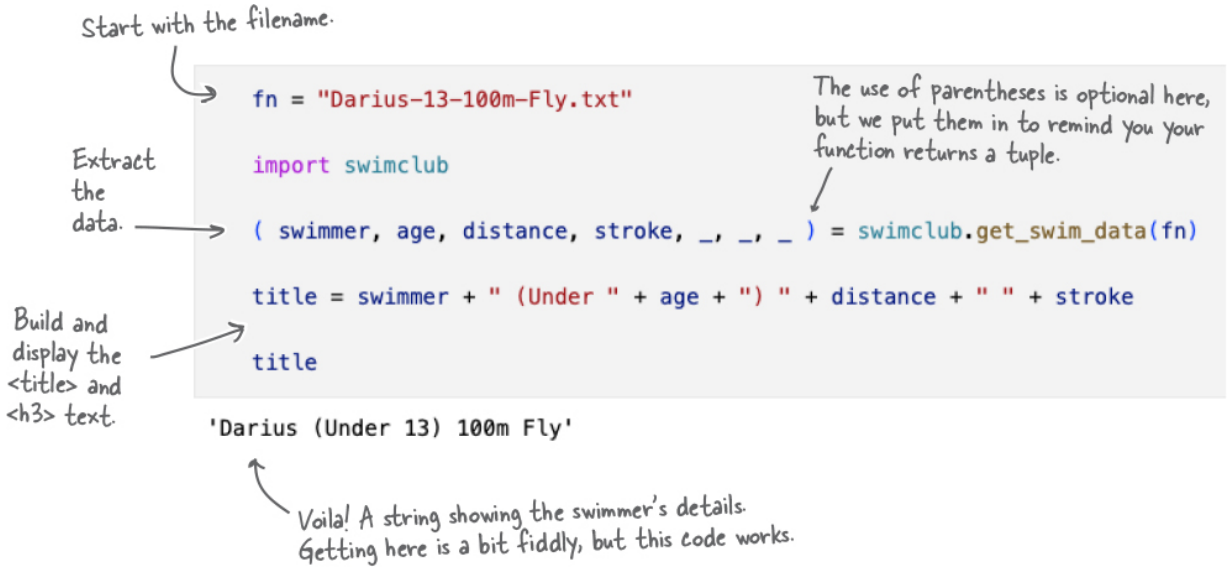
Build the strings your HTML needs in code

Here's the first chunk of HTML from `bar.html`, up to and including the `<h3>` tag. Most of this markup can be kept, although the `<title>` and `<h3>` text needs to change so that the page included the swimmer's details.



The swimmer's identification data is returned as part of the tuple from your `get_swim_data` function. Once you have that data, you can create HTML markup like that shown above with the text replaced as needed.

Building a string is possible, even though it can be a bit fiddly. Take a look:



String concatenation doesn't scale

Now that the `title` variable exists and has been assigned the correct text to use, it can be used to build a chunk of HTML:



At first glance, the code snippets on this and the previous page seems fine, and both snippets run (go ahead and try them in your notebook). But, look at all those concatenation operators (`+`) and all those double quotes! You really do need to pay attention to see what's happening here. This type of code makes most programmer's head's hurt – there's a lot to keep an eye on while working out what this code does, and this makes things extra tricky when it comes to making changes.



Yes, f-strings will help.

F-strings, short for *formatted string literals*, are a relatively new addition to Python 3, as of the 3.6 release. And they are waaaaaay cool.

Today' Interview: Getting to know f-string.



Head First: Let me begin by asking how it feels to be referred to as "waaaaaay cool"?

F-string: A little overwhelming, to be honest. I'm the latest in a long line of Python string-building mechanisms and, although what I do was always possible, I seem to be very popular.

Head First: I think it's more than that: these days it's like you're everyone's first choice when it comes to building a formatted string.

F-string: Why, thank you for saying that. I'm really just doing my job. Nothing more, nothing less.

Head First: Let's talk about your way of doing things as opposed to another. Let's compare you to using the `+` operator to build a formatted string with concatenation.

F-string: OK, sounds good. Imagine you have two variables defined, one called `first` and another called `second`. The `first` variable is assigned the string "Tim", whereas `second` is assigned the string "O'Reilly".

Head First: With those string values being chosen completely at random…?

F-string: But of course! Anyway, you have a need to build a string (aka *a formatted literal*) which evaluates to "Hi there, Tim O'Reilly!". Using string concatenation, this code would do the trick:

```
print("Hi there, " + first + " " + second + "!")
```

Head First: And there's nothing wrong with that code. When I run it in my Jupyter Notebook, it correctly displays the message.

F-string: It sure does. However, you do need to concentrate on all those `+` operators and those extra strings, such as `" "` and `"!"` in order to work out what's going on. But, I'm happy to concede that it does work, even though the code's a bit fiddly.

Head First: So, what would you do differently?

F-string: Take a look at my version of the same formatted string literal:

```
print(f"Hi there, {first} {second}!")
```

Head First: It doesn't look all that different… and I'm taking it that this also correctly displays the message?

F-string: Yep. Now, take a closer look. Did you spot the "`f`" character prefixing the string?

Head First: Ehhh… oh, yes, I see it now. Right at the beginning?

F-string: Yes, the "`f`" character tells Python that what follows is an *f-string*. And did you spot the use of the curly-braces *within* the string, `{` and `}`?

Head First: Those curly-braces surrounding the `first` and `second` variable names? Yes, I spotted those right away.

F-string: Cool. When I see a variable name surrounded by curly-braces, I go and get the value of the variable and *insert* it into the string in place of the curly-braces and the variable name. There's a name for this mechanism: it's called *variable interpolation*.

Head First: Which is just a fancy way of saying: *insert here*, right?

F-string: Yes. That's what *interpolation* means.

Head First: I've just run your line of code and am happy to report that it displays the exact same message as the string concatenation code. What I'm not getting it why your mechanism is better…

F-string: Well, once you understand what the curly-braces do, there's nothing else to concentrate on. With the concatenation code, you have a lot more to worry about, and as the format of your string gets more complex, it gets harder to understand ,whereas my mechanism, frankly, doesn't. It's that simple.

Head First: Which I guess says it all. Thanks for the explanation, f-string.

F-string: No problem. I enjoyed our chat.

EXERCISE



Although you've only just met f-strings, let's put them to use straight-away. Here's the code that uses concatenation to build the title of

your bar chart. Your first task is to rewrite this code to use an f-string instead.

```
title = swimmer + " (Under " + age + ") " + distance + " " + stroke
```

Grab your pencil, then write-in the f-string you'd use here.

Your second task is to rewrite the HTML-generating code from earlier to use a f-string. Here is the code from earlier:

```
html = "<!DOCTYPE html><html><head><title>" + title
html = html + "</title></head><body><h3>" + title + "</h3>"
```

We're leaving more space here for this f-string, not because we think it's more complex, but to give you enough room to write the f-string using a multi-line triple-quoted string (which can also be prefixed with an "f", transforming the triple-quoted string into an f-string, too).

EXERCISE SOLUTION



Although you'd only just met f-strings, we wanted you to put them to use straightaway. This code uses concatenation to build the start of your bar chart HTML page. Your first task was to rewrite this code to use an f-string instead.

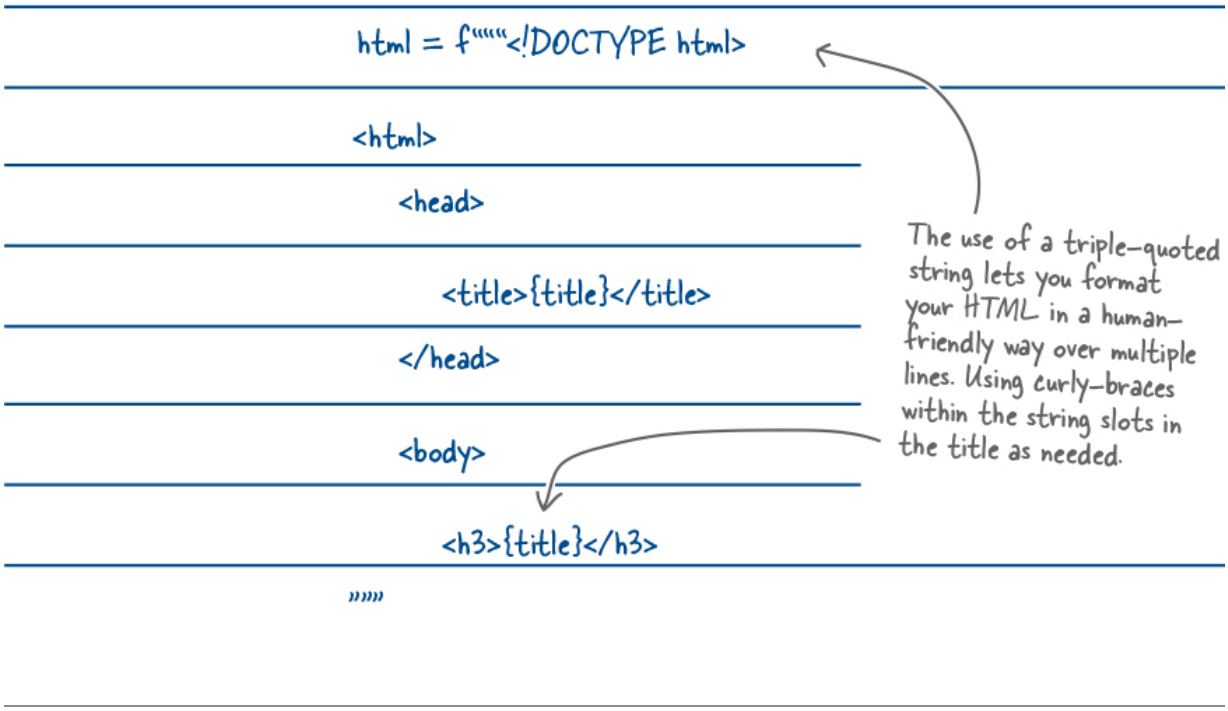
```
title = swimmer + " (Under " + age + ") " + distance + " " + stroke
```

A single-set of double-quotes, together with curly-braces around your variable names, does the trick here.

```
title = f"{swimmer} (Under {age}) {distance} {stroke}"
```

Your second task was to rewrite the HTML-generating code from earlier to use a f-string. Here is the code from earlier:

```
html = "<!DOCTYPE html><html><head><title>" + title
html = html + "</title></head><body><h3>" + title + "</h3>"
```



TEST DRIVE



Let's run your two f-strings in your notebook to see them in action.

Up first, the f-string which generates the value for `title`:

```
title = f"{swimmer} (Under {age}) {distance} {stroke}"

title
```

```
'Darius (Under 13) 100m Fly'
```

This looks good.

And, secondly, the HTML-generating f-string:

```
html = f"""<!DOCTYPE html>
<html>
    <head>
        <title>{title}</title>
    </head>
    <body>
        <h3>{title}</h3>
"""

html
```

Admittedly, this looks a little strange...

```
'<!DOCTYPE html>\n<html>\n    <head>\n        <title>Darius (Under 13) 100m
Fly</title>\n    </head>\n    <body>\n        <h3>Darius (Under 13) 100m Fly</h3>\n'
```

The output produced by displaying your triple-quoted string looks a little weird until, that is, you display it with the **print** BIF:

```
print(html)
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>Darius (Under 13) 100m Fly</title>
    </head>
    <body>
        <h3>Darius (Under 13) 100m Fly</h3>
```



THERE ARE NO DUMB QUESTIONS

Q: Do f-strings support format specifiers so I can zero-fill, format floating-point numbers, left-and right align, and so on?

A: Yes, but let's not worry about all of that until it's needed. (If you can't wait until then, start your exploration by reading the appropriate section of the Python documentation: https://docs.python.org/3/reference/lexical_analysis.html#f-strings).

F-strings have proven to be hugely popular

It did not take long for f-strings to catch on in the Python community. Once you get how they work, you start to see all manner of places where you can use them. A case in point is near the bottom of your `swimclub.py` code, where you have a line of code which looks like this:

```
    average_str = str(minutes) + ":" + str(secon
```

Refactoring the above line of code to use an f-string results in this:

```
    average_str = f"{minutes}:{seconds}.{hundred
```

which is not only shorter, but more clearly represents the swim time format. Note that there's no requirement to perform that explicit type conversion using that call to the **str** BIF. To be honest, we'd be hard-pressed to make a case for using a line of code that doesn't employ an f-string. The *Head First Coders* are *huge* fans.

TEST DRIVE



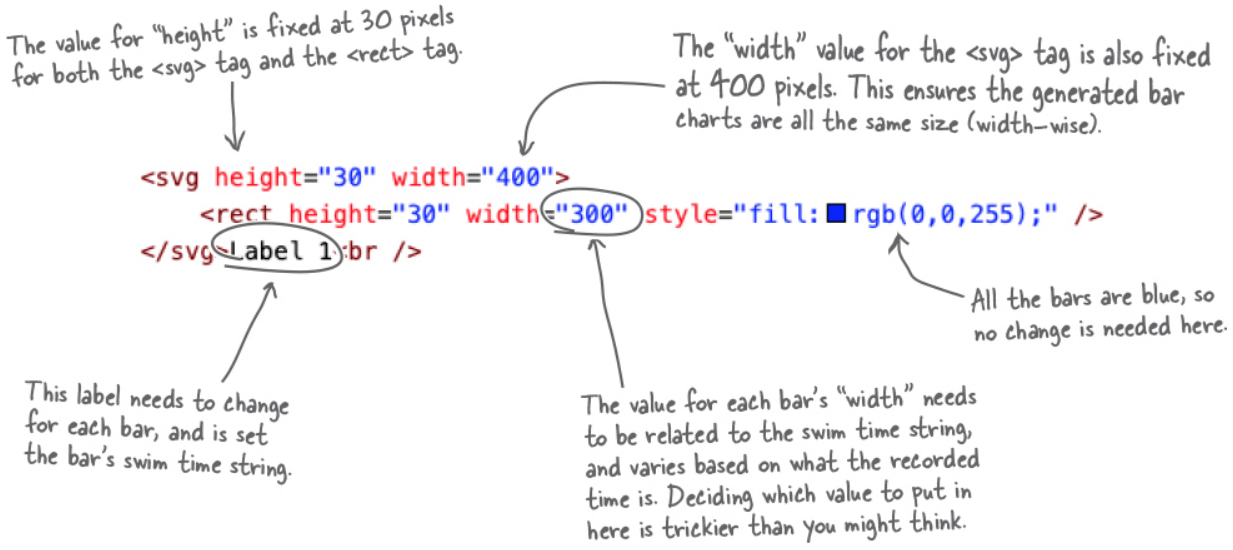
Before moving on, open up your `swimclub.py` file in VS Code, then adjust the bottom of your `get_swim_data` function to use an f-string when creating the `average_str` variable. Once the change is applied, be sure to **save** `swimclub.py`.

```
mins_secs, hundredths = f"{(average / 100):.2f}".split(".")
mins_secs = int(mins_secs)
minutes = mins_secs // 60
seconds = mins_secs - minutes * 60
average_str = f"{minutes}:{seconds}.{hundredths}"
return swimmer, age, distance, stroke, average, average_str, times
```

Your first f-string!

With f-strings in your toolset, SVG is easy!

Here's the HTML snippet from `bar.html` which creates a labelled blue SVG bar:



In order to use Python to programmatically generate the Coach's bar charts, you'll need to create as many `<svg>` tags as there are times in your swimmer's data. You've likely decided that using a **for** loop on the `times` data returned from your `get_swim_data` function is what's needed here, with each iteration of your **for** loop using a f-string to generate a custom `<svg>` tag as needed. And you'd be right to adopt this as a strategy.

BRAIN POWER



Take a moment to import the most-recent version of your `swimclub` module into `Charts.ipynb`, then invoke the `get_swim_data` function on the `fn` variable, which returns Darius's data. (Recall that you may need to restart your notebook to ensure the most-recently edited module is imported). Take some time to study the data returned from your function.

Noting that the `width` attribute above is to be set to a value drawn from this data, can you spot any issues with the data returned?

The data is all there, or is it?

The data returned from your function looks fine. But does this data alone support the creation of the `<svg>` tags you need to produce?

```
import swimclub

swimclub.get_swim_data(fn)
```

```
('Darius',
 '13',
 '100m',
 'Fly',
 8657.5,
 '1:26.58',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'])
```

Most of this data are strings, except for the large hundredths of a seconds number: 8657.5.

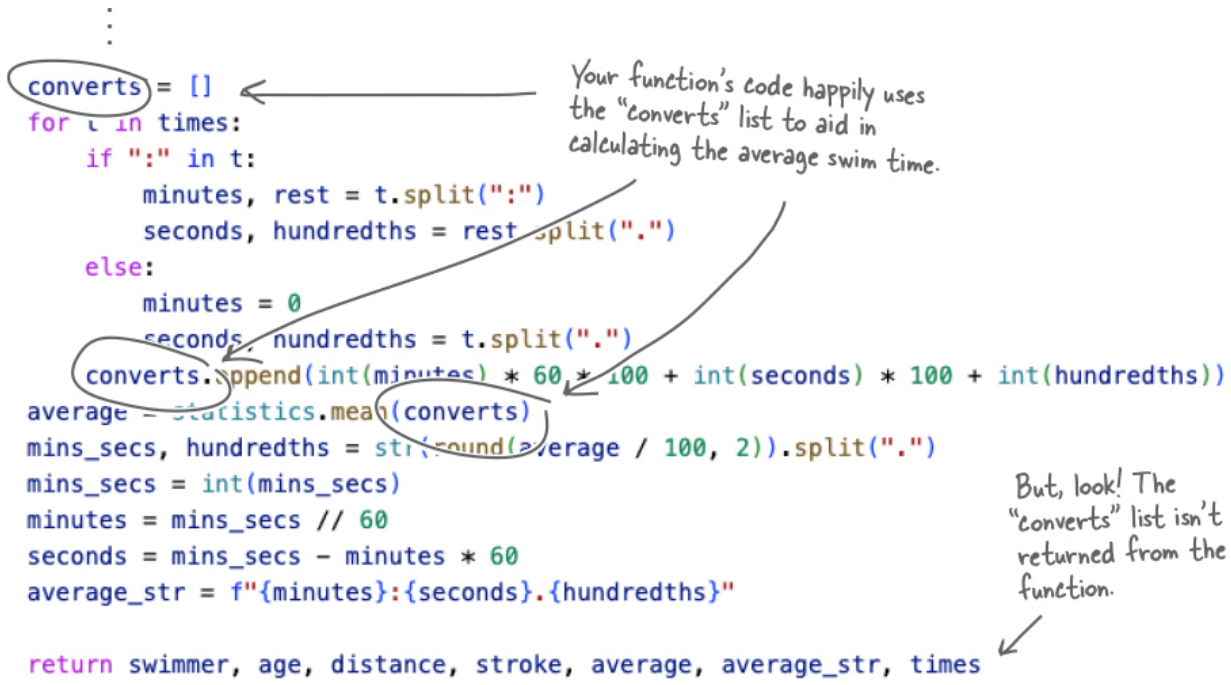


Yes, well spotted (on both issues).

Some data is missing here (but there's an easy fix for that one). And, yes, the numeric values are very large (much bigger than 400), which might make them hard to work with.

Make sure you return all the data you need

Take a look at your `get_swim_data` function, and notice how the `converts` list is populated with the numeric equivalents of the string-based swim times, before the average calculation is performed. The average value is then returned to the caller along with all the other values. Critically, the `converts` list is *not* returned to the caller, mainly because it didn't need to be… *until now, that is.*



It's not hard to fix this issue. All it takes is a small change to your function's `return` statement so that it looks like this:

```
return swimmer, age, distance, stroke, average, average_str, times, converts
```

An easy edit.

Go ahead and make this edit to your module's code, being sure to save your work. Then, *restart* your notebook once more so any imports use your updated module's code.

If you re-run your notebook's code from the beginning using repeated presses of **Shift+Enter**, you won't get very far, as the `Charts.ipynb` notebook complains of "too many values to un-

pack" the first time you call your updated `get_swim_data` function. **Fix this issue before moving on**.

NOTE

We aren't going to tell you what to do here. Mean old us, eh? If you're stuck, flip back to the middle of the previous chapter for coverage of these types of ValueErrors.

You have numbers now, but are they usable?

With the change made to your `swimclub.py` module, and your notebook restarted, your latest function *also* returns the numeric equivalents of the swim times strings:

```
import swimclub

swimclub.get_swim_data(fn)
```

```
('Darius',
 '13',
 '100m',
 'Fly',
 8657.5,
 '1:26.58',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 [8795, 8107, 9096, 8322, 8795, 8830])
```

A list of numbers representing the converted time-strings.

Although the returned data looks fine, the numbers are a little on the large side, and this is the second issue. Specifically, the SVG snippet from earlier has a *maximum* width value of `400`:

The width of the enclosing <svg> tag needs to be greater than the enclosed <rect> tag's width value.

```
<svg height="30" width="400">
    <rect height="30" width="300" style="fill: rgb(0,0,255);" />
</svg>Label 1<br />
```

Obviously, none of those six numbers greater than 8000 are less than 400. And if you were to use those numbers as is, your SVG rectangles would likely scroll off the right side of your screen by quite a margin!

What's needed here is some way to *scale* the larger numbers down to a value which fits within the 0 to 400 range, which sounds like it might need some fancy Math. Yikes!

As luck would have it, the *Head First Coders* just sent a small module (called `hfpy_utils.py`) via email, together with a small notebook showing you how to convert any number to it's equivalent within some other range (called `HowTo.ipynb)`. To use the module, copy it into your `Learning` folder, then cast your eye over the code in the `HowTo.ipynb` notebook, which is annotated on the next page.

NOTE

You'll find the module and notebook included with this book's download material.

Scaling numeric values so they fit

You are more than welcome to take a look at the code in the `hfpy_utils.py` module, but it's not necessary for you to understand how this code works. You do, however, need to understand what the `convert2range` function does for you. Here's an annotated copy of the `HowTo.ipynb` notebook:

There are only two code cells in the notebook.

```
import swimclub

fn = "Darius-13-100m-Fly.txt"

*rest, converts = swimclub.get_swim_data(fn)

converts
```

```
[8795, 8107, 9096, 8322, 8795, 8830]
```

The first cell populates the "converts" list.

```
import hfpy_utils

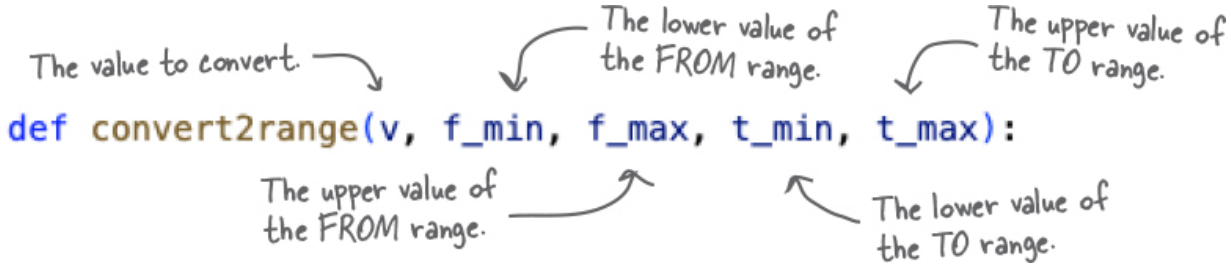
for n in converts:
    print(n, '->', hfpy_utils.convert2range(n, 0, max(converts), 0, 400))
```

When given a list of values, the "max" BIF returns the largest.

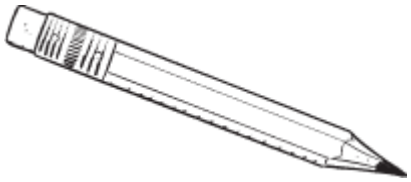
```
8795 -> 386.76
8107 -> 356.51
9096 -> 400.0
8322 -> 365.96
8795 -> 386.76
8830 -> 388.3
```

This loop calls the "convert2range" function to scale each of the numbers in the "converts" list to their equivalent scaled value in the 0 to 400 range.

Looking at the `convert2range` function's signature should help with your understanding of how it's invoked:



SHARPEN YOUR PENCIL



Now that you've seen f-strings in action, as well as studied the code in the *Head First Coders'* notebook, let's take some time to create the code required to generate the SVG tags for each of the current swimmer's swim times. We'll go through this step-by-step, with you thinking about what's required, experimenting with code in your notebook, then (grabbing your pencil and) writing down the code you need in the spaces below.

Start by writing a line of code which creates two variables from the swimmer's data by calling the `get_swim_data` function and creating two lists, one called `times` and another `converts`:

_____

_____

Create another variable called `from_max`, assigning the maximum value from the `converts` list to it:

_____

_____

Import the `hfpy_utils.py` module, then create a new variable called `svgs`, initializing it to an empty string:

_____

_____

_____

_____

Assume a value exists called `n` which contains a slot value that indexes into your `converts` list. Provide a line of code that creates a variable called `bar_width` and sets it to a value of `n` which scales from `0-from_max` to `0-400`:

_____

_____

NOTE

Hint: The Head First Coders did something very similar to this in their notebook, so all you need to do is adapt their code to this specific requirement.

Write the first line of a `for` loop which iterates over an *enumerated* `times` list, creating two new loop variables called `n` and `t`:

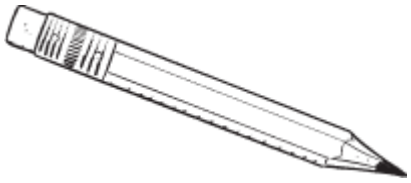
Here is the snippet of SVG taken from the `bar.html` file:

```
<svg height="30" width="400">
    <rect height="30" width="300" style="fill:rgb(0,0,255);" />
</svg>Label 1<br />
```

Create a triple-quoted multi-line f-string that uses the `bar_width` and `t` variables from the last page to build a SVG tag for the current swim time:

With your f-string ready, show the code that appends it to the end of the `svgs` variable created on the last page:

SHARPEN YOUR PENCIL SOLUTION



Now that you've seen f-strings in action, as well as studied the code in the *Head First Coders'* notebook, you were to take some time to create the code required to generate the SVG tags for each of the current swimmer's swim times. Proceeding step-by-step, you were to think about what's required, experiment with code in your notebook, then (grab your pencil and) write down the code you need in the spaces below.

You were to start by writing a line of code which creates two variables from the swimmer's data by calling the `get_swim_data` function and creating two lists, one called `times` and another `converts`:

```
*_, times, converts = swimclub.get_swim_data(fn)
```

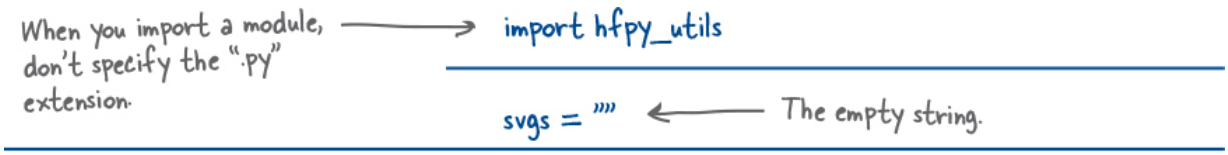
Grab the last two values from the returned tuple.

You were then to create another variable called `from_max`, assigning the maximum value from the `converts` list to it:

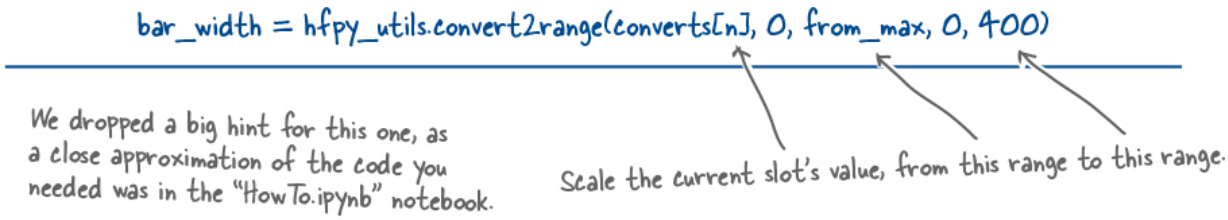
```
from_max = max(converts)
```

Determine the max value found in the list.

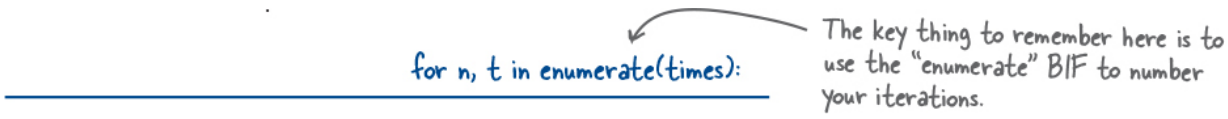
Next up was to import `hfpy_utils.py`, then create a new variable called `svgs`, initializing it to an empty string:



Assuming a value exists called `n` that contains a slot value which indexes into your `converts` list, you were to provide a line of code that creates a variable called `bar_width` and sets it to a value of `n` which scales from `0-from_max` to `0-400`:



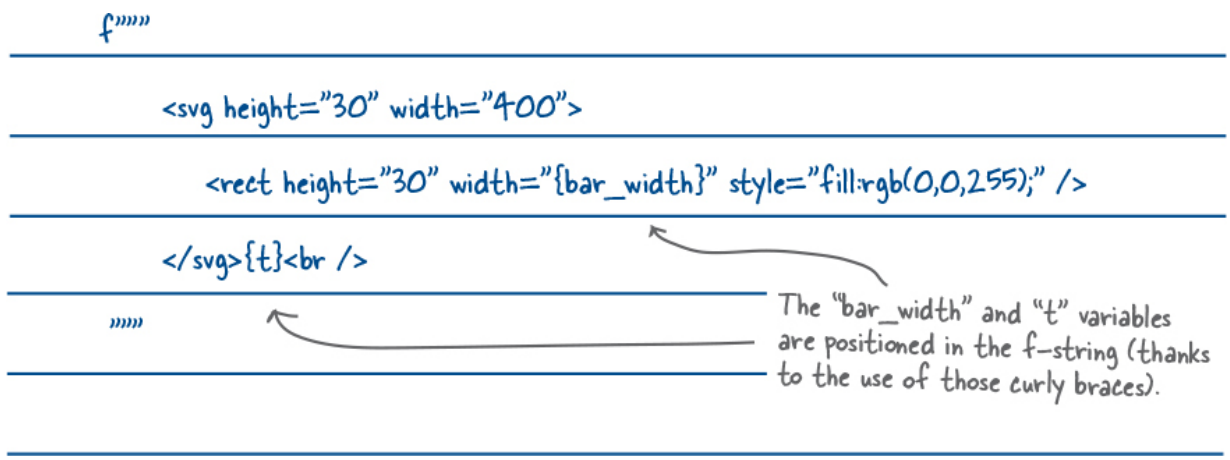
Then you were to write the first line of a `for` loop which iterates over an *enumerated* `times` list, creating two new loop variables called `n` and `t`:



Here is the snippet of SVG taken from the `bar.html` file:

```
<svg height="30" width="400">
    <rect height="30" width="300" style="fill: rgb(0,0,255);" />
</svg>Label 1<br />
```

You were asked to create a triple-quoted multi-line f-string that uses the `bar_width` and `t` variables from the last page to build a SVG tag for the current swim time:



Then to conclude, and with your f-string ready, you were to show the code that appends it to the end of the `svgs` variable created on the last page:

```
svgs = svgs + f"""
                <svg height="30" width="400">
                  <rect height="30" width="{bar_width}" style="fill:rgb(0,0,255);" />
                </svg>{t}<br />
              """
```

Concatenate the f-string with the "svgs" string's previous value. If you do this within a loop, the "svgs" string grows.

With all this code written, let's put it all together in a Test Drive and see what happens...

TEST DRIVE



With just a few minor tweaks, the code you created during your just-completed *Sharpen* can be combined into a single code cell in your notebook, like so:

```
import swimclub
import hfpy_utils

swimmer, age, distance, stroke, _, average_str, times, converts = swimclub.get_swim_data(fn)
from_max = max(converts)
svgs = ""
for n, t in enumerate(times):
    bar_width = hfpy_utils.convert2range(converts[n], 0, from_max, 0, 400)
    svgs = svgs + f"""
                    <svg height="30" width="400">
                        <rect height="30" width="{bar_width}" style="fill:rgb(0,0,255);" />
                    </svg>{t}<br />
                """
```

The minor tweaks involve importing the "swimclub" module, as well as grabbing most of the data returned as a tuple from the "get_swim_data" function.

Pressing **Shift+Enter** on this cell runs your code, but displays nothing. In the next cell, use the **print** BIF to take a look at your handiwork:

```
print(svgs)
```

There is an individual <svg> tag for each of this swimmer's recorded times.

```
<svg height="30" width="400">
    <rect height="30" width="386.76" style="fill:rgb(0,0,255);" />
</svg>1:27.95<br />

<svg height="30" width="400">
    <rect height="30" width="356.51" style="fill:rgb(0,0,255);" />
</svg>1:21.07<br />

<svg height="30" width="400">
    <rect height="30" width="400.0" style="fill:rgb(0,0,255);" />
</svg>1:30.96<br />

<svg height="30" width="400">
    <rect height="30" width="365.96" style="fill:rgb(0,0,255);" />
</svg>1:23.22<br />

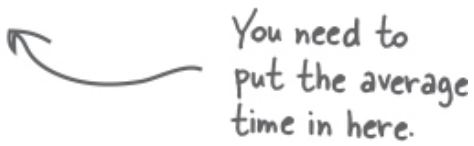
<svg height="30" width="400">
    <rect height="30" width="386.76" style="fill:rgb(0,0,255);" />
</svg>1:27.95<br />

<svg height="30" width="400">
    <rect height="30" width="388.3" style="fill:rgb(0,0,255);" />
</svg>1:28.30<br />
```

All that's left is the end of your webpage

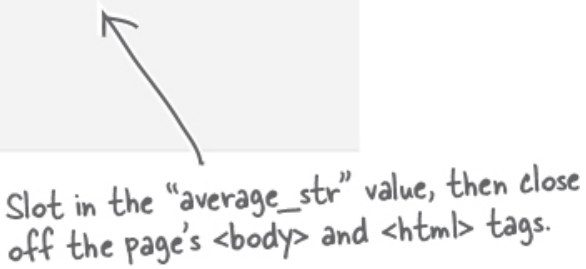
You are likely anxious to see how your generated HTML and SVG markup looks in your browser, but you're not done yet. Recall that the end of the `bar.html` file has this markup:

```
        <p>It's simple, but it works!</p>
    </body>
</html>
```



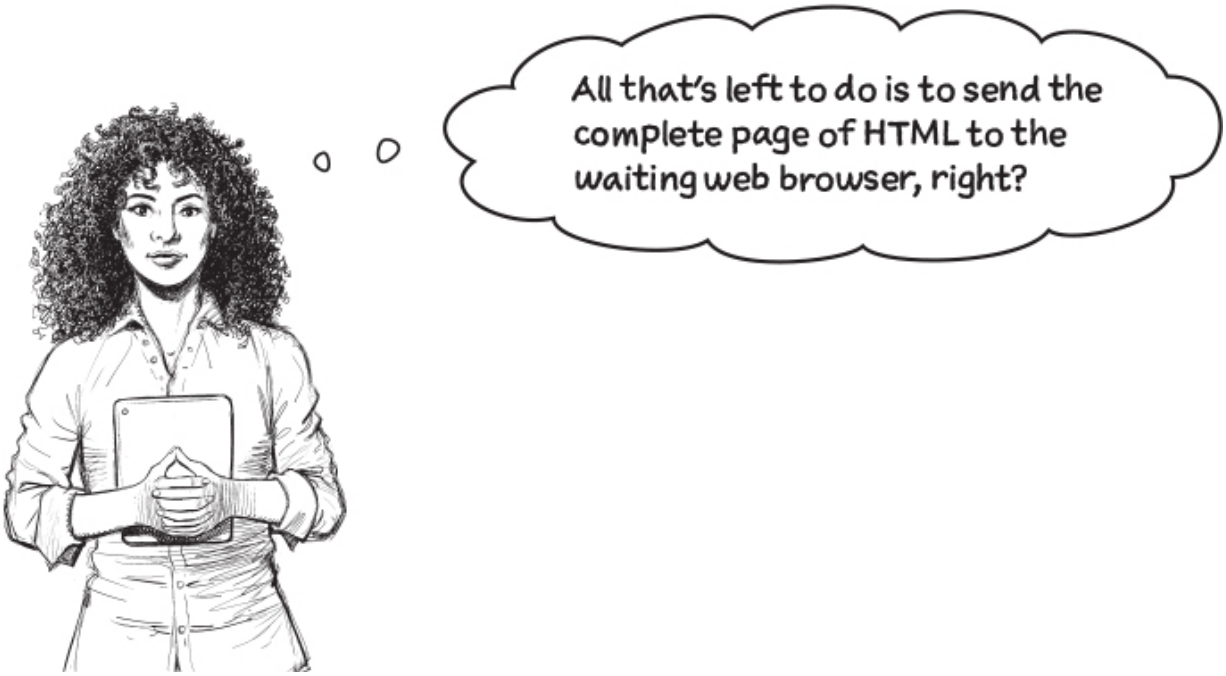
As you're well on your way to becoming an f-string ninja, we're just going to give you the code for this bit:

```
    footer = f"""
            <p>Average time: {average_str}</p>
        </body>
    </html>
    """
```



Creating the complete page of HTML markup is not difficult:

```
page = html + svgs + footer
```

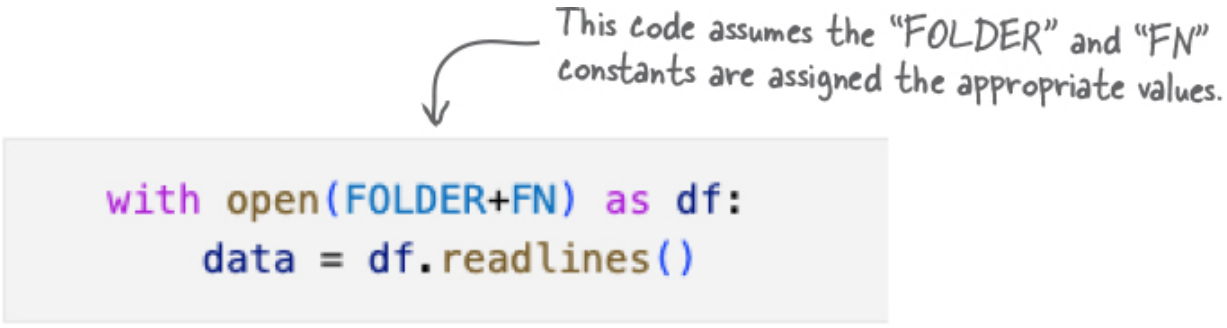


Yes. But let's save the HTML to a file first.

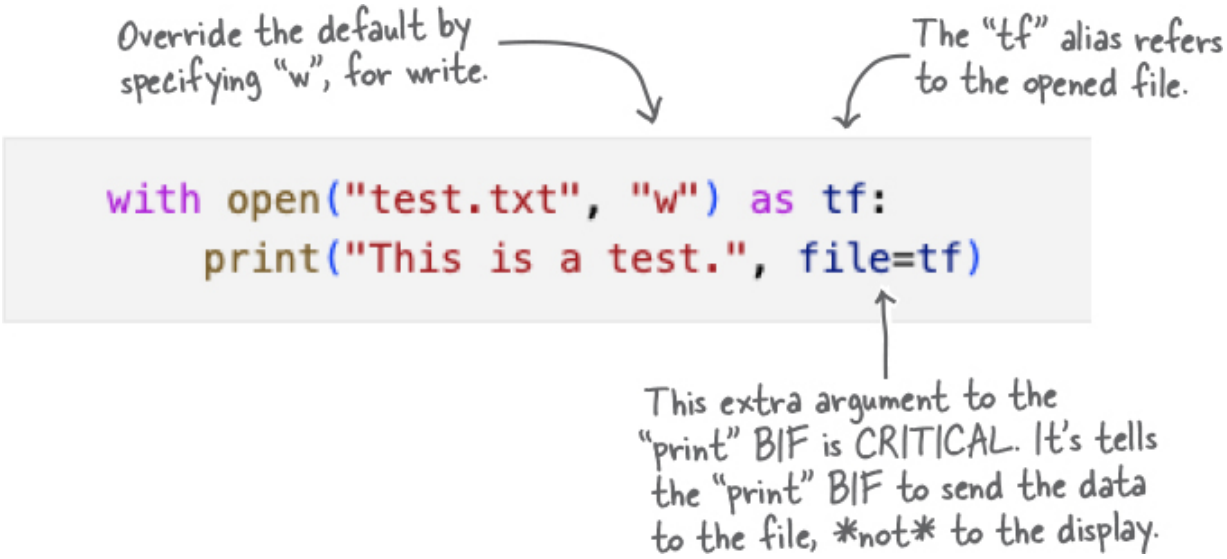
The PSL's `webbrowser` module, which you used to display `bar.html` at the start of this chapter, expects a valid URL to open. It's a pity, but you can't send the module a big string of HTML to display. The HTML either has to be available on the web or saved to a file.

Writing to files, like reading, is painless

Recall this code which you used to read swim times from a file into a new list called



The **open** BIF, by default, opens a file for *reading*, but it's also possible to open a file in *write mode* by providing a second argument to **open**:



If run this code in a new code cell, it creates `test.txt` if it doesn't exist, then stores the (too exciting) message in the file, before closing

the file automatically (thanks to your old friend, the `with` statement).

If `test.txt` already exists, this code overwrites (!!!) the old contents of the file with the message. If you want to preserve the existing contents of the file, *append mode* can do that for you, assuming you replace the `"w"` argument to the **open** BIF with `"a"`.

Knowing all of this, let's use code similar to what's shown above to save your generated web page to a file.

EXERCISE

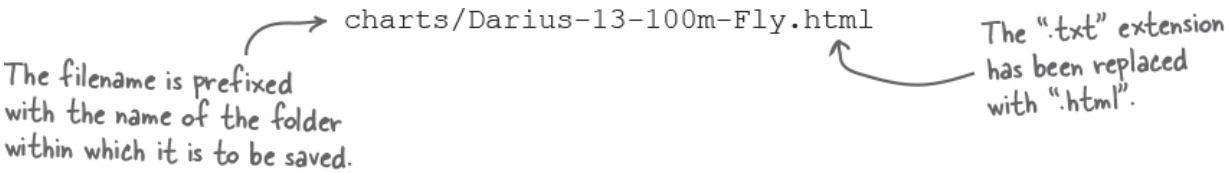


Before starting this exercise, create a sub-folder in your `Learning` folder and call it `charts`.

Rather than use a generic name for your web page's file, let's generate one from the name already stored in the `fn` variable. Specifically, let's craft an f-string which converts a filename that conforms to this pattern:

```
Darius-13-100m-Fly.txt
```

into one that conforms to this pattern:



When you've worked out the f-string that achieves this (bearing in mind that you can put Python code between the curly braces in an f-

string, not just variable names), grab your pencil and write-in the f-string you'd use in the space below. Assign the f-string to a variable called `save_to`:

_____

_____

Provide the code you'd use to save your HTML (currently in the `page` variable) to the file whose name is in the `save_to` variable:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

EXERCISE SOLUTION

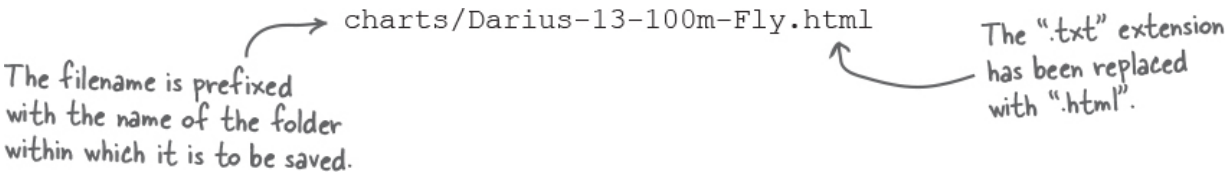


Before starting this exercise, you needed to create a sub-folder in your `Learning` folder and call it `charts`.

Rather than use a generic name for your web page's file, you were to generate one from the name already stored in the `fn` variable. Specifically, you were to craft an f-string which converts a filename that conforms to this pattern:

```
Darius-13-100m-Fly.txt
```

into one that conforms to this pattern:



When you've worked out the f-string that achieves this (bearing in mind that you can put Python code between the curly braces in an f-string, not just variable names), you were asked to grab your pencil

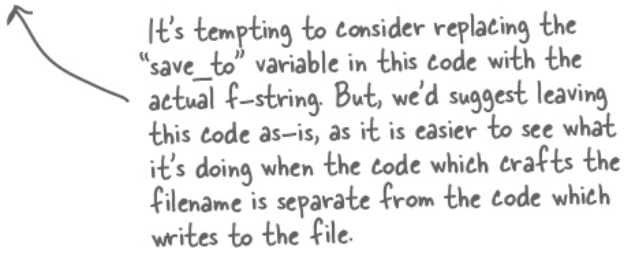
and write-in the f-string you'd use in the space below, assigning the f-string to a variable called `save_to`:

```
save_to = f"charts/{fn.removesuffix('.txt')}.html"
```

To conclude, you were to provide the code you'd use to save your HTML (currently in the `page` variable) to the file whose name is in the `save_to` variable:

```
with open(save_to, "w") as sf

print(page, file=sf)
```



It's time to display your handi work

With your HTML and SVG saved to a file in the `charts` folder, asking the PSL's `webbrowser` module to display the chart is straightforward:

Define the file location to use.

```
save_to = f"charts/{fn.removesuffix('.txt')}.html"

with open(save_to, "w") as sf:
    print(page, file=sf)
```

Save your webpage to the file location.

Import the libraries you need.

```
import os
import webbrowser
```

Ask your web browser to display your chart.

```
webbrowser.open("file://" + os.path.realpath(save_to))
```

```
True
```

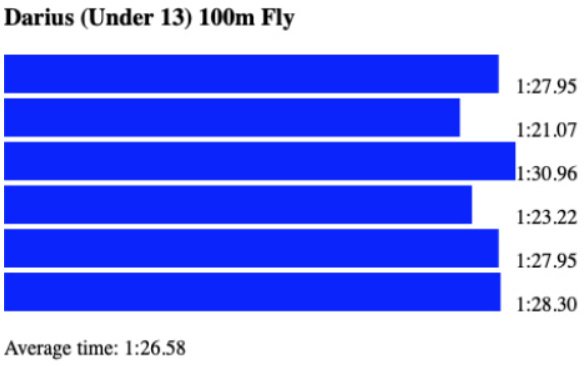
And here's how your chart looks in a browser.

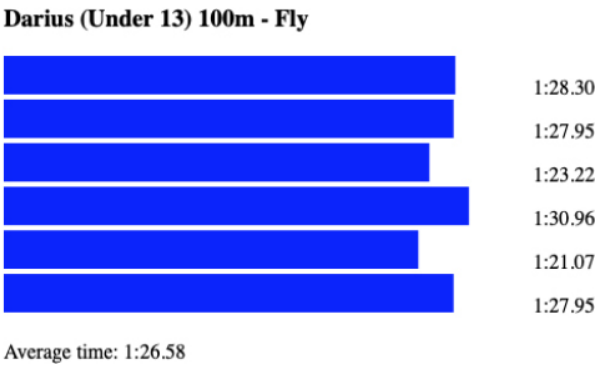


This looks great! This bar chart is generated from your Python code and the Coach's data. Cool.

You've almost produced the correct chart…

Take a look at the two charts shown below. Your code's chart is on the left, while the "target" chart from earlier in this chapter is on the right:







Cubicle Conversation

Alex: Both of these are very similar…

Mara: But there are subtle differences.

Sam: For starters, the chart on the left displays its bars in the wrong order.

Mara: Yeah, the Coach was pretty clear when he told us he wanted to see the most-recent swim time at the top of the chart.

Alex: Surely that's an easy fix?

Sam: I think so, yes. Lists come with a built-in **reverse** method which I think we can exploit here.

Mara: It's just a matter of calling **reverse** in the right place so that the SVG-generating code processes the data in the correct order.

Alex: What else is wrong? Nothing else jumps out at me.

Sam: Mara did say the difference was subtle. Notice how the bar chart on the right has extra whitespace after each of the bars, before the times are shown.

Mara: That's not a big issue, but it certainly means the chart on the right is easier on the eye. The chart on the left appears cluttered to me.

Alex: Ah, yes, I see that now. It that an easy fix, too?

Mara: I hope so.

Sam: Remember that the *width* of each bar is determined by calling the `hfpy_utils.convert2range` function. I think we'll be able to fiddle with those calls to sort out this issue without too much trouble.

Alex: Cool. It's two quick-fixes, then. That shouldn't take too long…

TEST DRIVE



Adjust your code cell as shown below, restart your notebook, then re-run all your code again to see the differences these small edits make:

Reversing both lists before running your SVG-generating "for" loop fixes the ordering issue.

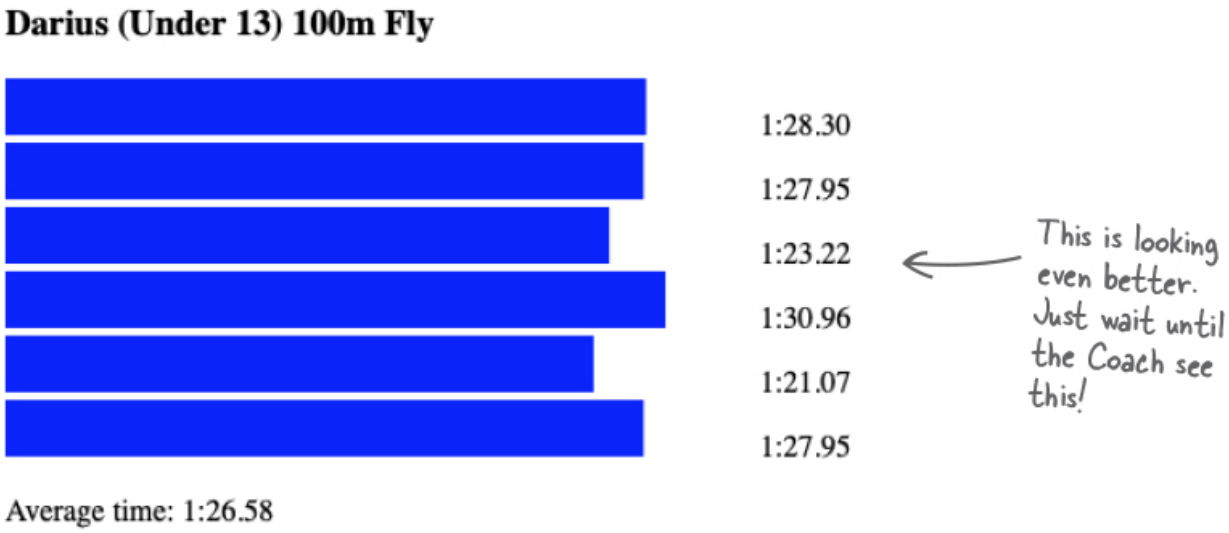
```
import swimclub
import hfpy_utils


swimmer, age, distance, stroke, _, average_str, times, converts = swimclub.get_swim_data(fn)
from_max = max(converts)
svgs = ""
times.reverse()
converts.reverse()
for n, t in enumerate(times):
    bar_width = hfpy_utils.convert2range(converts[n], 0, from_max, 0, 350)
    svgs = svgs + f"""
                    <svg height="30" width="400">
                        <rect height="30" width="{bar_width}" style="fill:rgb(0,0,255);" />
                    </svg>{t}<br />
                """
```

Changing the maximum value for the "to range" from 400 to 350 should give you the whitespace you need.

Sure enough, your latest version of the bar chart looks a lot better than before. The ordering is fixed, and everything looks a bit neater

now that there's some additional spacing:



It's time for another custom function

The code in your `Charts.ipynb` notebook produces a bar chart which is a nice approximation of what the Coach needs, as well as closely matching what the Coach created with the spreadsheet. The bar chart is not an *exact* match, and this has to do with the fact that your chart's x-axis starts at zero, whereas the spreadsheet's x-axis starts at 1:13.44, so the scale is a little off. Not to worry, what you've produced here is more than good enough.

All that's missing is the ability to call your code for *any* filename, not just the one which contains the swim times for Darius. Creating an-

other function should do the trick. Recall the function-creating three-step strategy from the last chapter:

1 Think up a nice, meaningful name

Let's call your function `produce_bar_chart` which, even if we do say so ourselves, is as fine a function name as you're likely to find.

2 Decide on the number and names of any parameters

Like `get_swim_data`, your new function takes a single parameter which identifies the filename to generate a bar chart for.

3 Indent your function's code under a def statement

def introduces the function, specifying the function's name and any parameters. As always, code indented under the **def** keyword is the function's code block.



Agreed. That is a good idea.

After all, the code you're creating is all part of the system you're building for the coach, so let's keep all the code you're planning to use in one module.

Take a look (on the next page) at the code we added to our `swimclub.py` file. You should do this, too, while paying attention to the *minor tweaks* we made to our final version.

Let’s add another function to your module

Add the next two lines to the *top* of your `swimclub.py` file, then add the tweaked function as shown:

Make sure any required module is imported before your function uses it.

```
import hfpy_utils

CHARTS = "charts/"
```

Define a constant which identifies where the bar charts are to be stored.

```
    ⋮
```

A useful comment is added as a triple-quoted string.

```
def produce_bar_chart(fn):
    """Given the name of a swimmer's file, produce a HTML/SVG-based bar chart, saving
    it to the CHARTS folder. Return the path to the bar chart file."""
    swimmer, age, distance, stroke, _, average_str, times, converts = get_swim_data(fn)
    from_max = max(converts)
    times.reverse()
    converts.reverse()
    title = f"{swimmer} (Under {age}) {distance} {stroke}"
    header = f"""<!DOCTYPE html>
                 <html>
                     <head>
                         <title>{title}</title>
                     </head>
                     <body>
                         <h3>{title}</h3>"""
    body = ""
    for n, t in enumerate(times):
        bar_width = hfpy_utils.convert2range(converts[n], 0, from_max, 0, 350)
        body = body + f"""
                         <svg height="30" width="400">
                             <rect height="30" width="{bar_width}" style="fill:rgb(0,0,255);" />
                         </svg>{t}<br />"""
    footer = f"""
                         <p>Average time: {average_str}</p>
                     </body>
                 </html>"""
    page = header + body + footer
    save_to = f"{CHARTS}{fn.removesuffix('.txt')}.html"
    with open(save_to, "w") as sf:
        print(page, file=sf)

    return save_to
```

As this function is now part of the "swimclub" module, you do not need to fully-qualify this call.

The original code referred to this f-string as "html". We think "header" is a better name, so we changed it.

We also changed the "svgs" name to "body".

This line now reads better. The HTML page is made up from a header, a body, and a footer.

The function returns the location/name of the saved file.

TEST DRIVE



Make sure your `swimclub` module is updated with the tweaked code from the last page.

Return to your `Charts.ipynb` notebook, clicking on VS Code's **Restart** button to clear the interpreter of the remnants of any previously executed code cells. Working at the bottom of your notebook, add and run the code as shown below. Make sure the results you see here are replicated on your screen.

```
import os
import swimclub
import webbrowser
```

Begin by importing the modules your code relies on.

Then run "produce_bar_chart" to create the HTML page as well as confirm the location of the created file.

```
swimclub.produce_bar_chart("Darius-13-100m-Fly.txt")
```

```
'charts/Darius-13-100m-Fly.html'
```

Assign the location to a variable called "chart".

```
chart = swimclub.produce_bar_chart("Darius-13-100m-Fly.txt")

webbrowser.open("file://" + os.path.realpath(chart))
```

```
True
```

Display the bar chart in your default web browser.



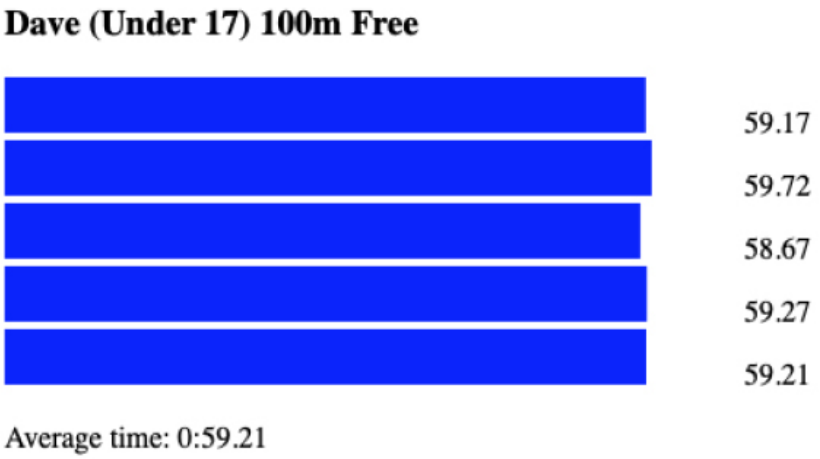
Looking good!

```
chart = swimclub.produce_bar_chart("Dave-17-100m-Free.txt")

webbrowser.open("file://" + os.path.realpath(chart))
```

```
True
```

Rinse and repeat for Dave's 100m Free style data.

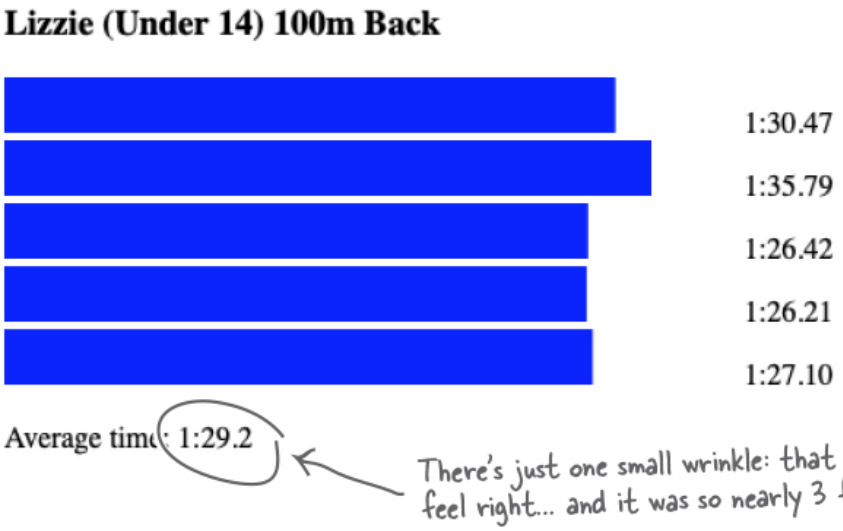



```
chart = swimclub.produce_bar_chart("Lizzie-14-100m-Back.txt")

webbrowser.open("file://" + os.path.realpath(chart))
```

```
True
```

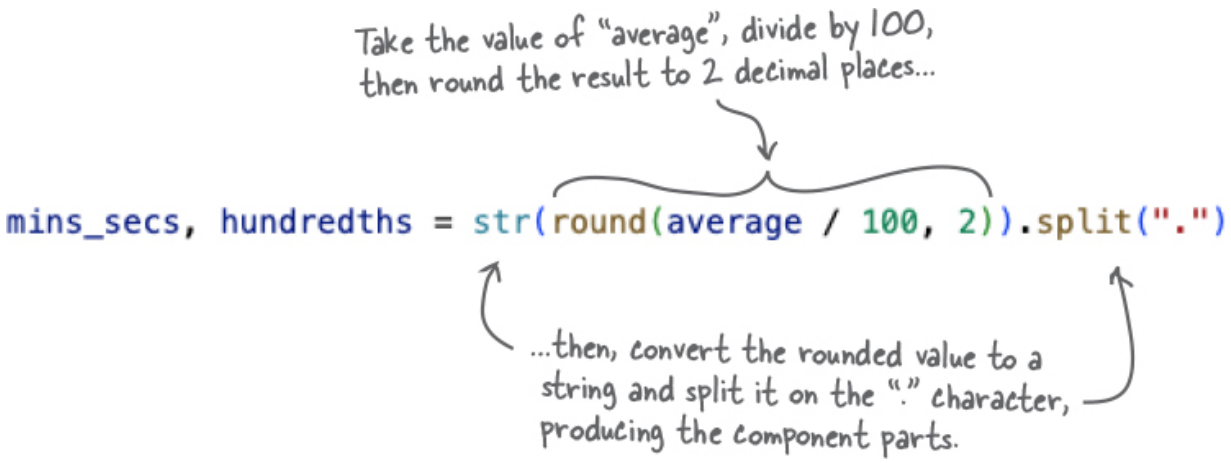
One more time for luck: Lizzie's 100m Back stroke data.



There's just one small wrinkle: that 2 doesn't feel right... and it was so nearly 3 for 3! 😉

What's with that hundredths value?

Your code has produced a value of " `2` " for the hundredths value for Lizzie. The problem is it's not clear if this value should be " `20` " or " `02` ". Here's the line of code from your `swimclub.py` module which produces this value:

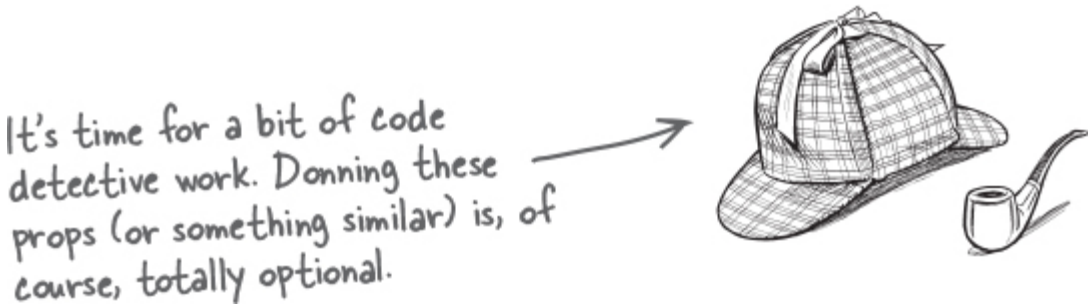




It's a bit of a mystery, isn't it?

That code does indeed look fine, but there's no escaping the fact it doesn't quite work for Lizzie's data. Bummer.

What to do?



The case of the missing digit (which may not be missing at all)

Your code, as written, appears to work fine for *some* of the Coach's data but not *all* of the Coach's data. Why is Lizzie's data giving an average time which doesn't conform to the *mins:seconds:hundredths* format, where the value of hundredths is a two-digit number? How did Lizzie end up with a value of 2 instead?



It's a mystery…

Initial analysis has determined this is the offending line of code:

```
    mins_secs, hundredths = str(round(average /
```

But, how can this be? This code works for Darius and Dave. Is there something wrong with Lizzie? Is there something wrong with Lizzie's data? Is there something wrong with – *heavens forbid!* – your code?

There's nothing else for it… professional help is required in this case. Professional *code detective* help. There's only one person for the job: Holmes, *Head First Holmes*.

Sweeping into the room, *Holmes* flashes a knowing grin, takes one look at your code and declares: *"Ah ha! I see the problem! The digit you're after isn't missing at all, it's being ignored."*

Aghast that *Holmes* has worked so quickly, you splutter: *"How can this be?!?!?"*. Pausing to take what feels like a rather unhealthily long puff on that trademark pipe, *Head First Holmes* replies:

"Why it's elementary. Let me show you."

The solution to the case of the missing digit

Grabbing hold of your keyboard, *Head First Holmes* works at a pace that defies the speed mere mortals can code at (that's you and me). *Holmes* be-gins by grabbing a copy of Lizzie's data using the `get_swim_data` function:

```
swimclub.get_swim_data("Lizzie-14-100m-Back.txt")
```

```
('Lizzie',
 '14',
 '100m',
 'Back',
 8919.8,
 '1:29.20',
 ['1:27.10', '1:26.21', '1:26.42', '1:35.79', '1:30.47'],
 [8710, 8621, 8642, 9579, 9047])
```



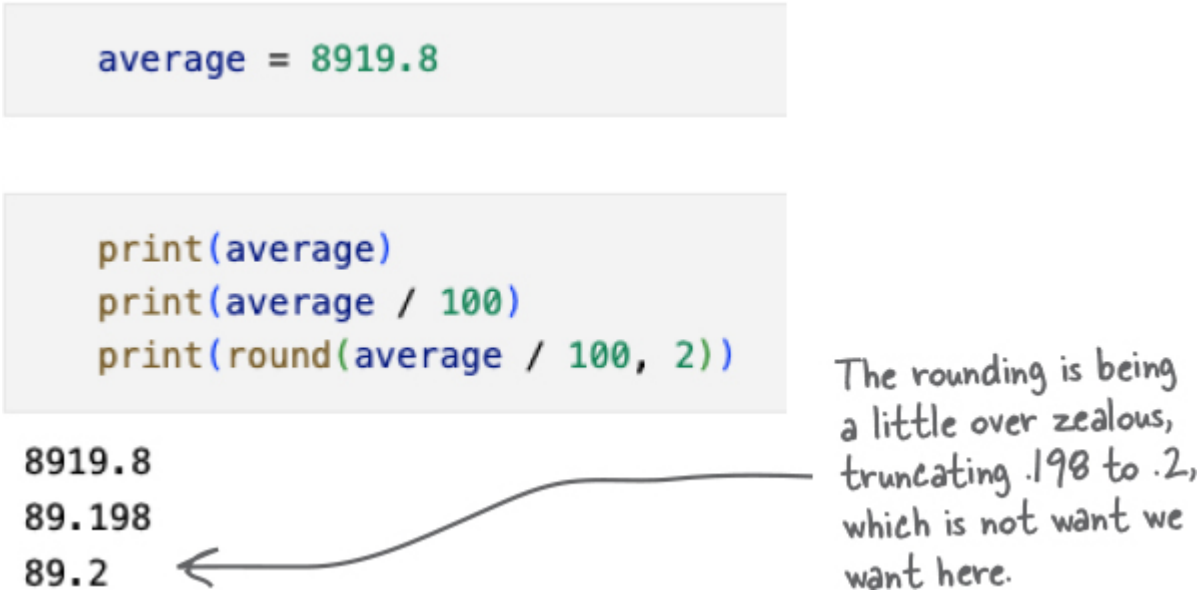
Coughing out a hearty *"Ah ha!"*, Holmes demonstrates what can happen when this offending line of code executes:

NOTE

Holmes really should lay off that pipe!

```
    mins_secs, hundredths = str(round(average /
```

Holmes assigns the calculated average value to a variable then types three **print** statements into a code cell, which he then executes with quick **Shift+Enter**:



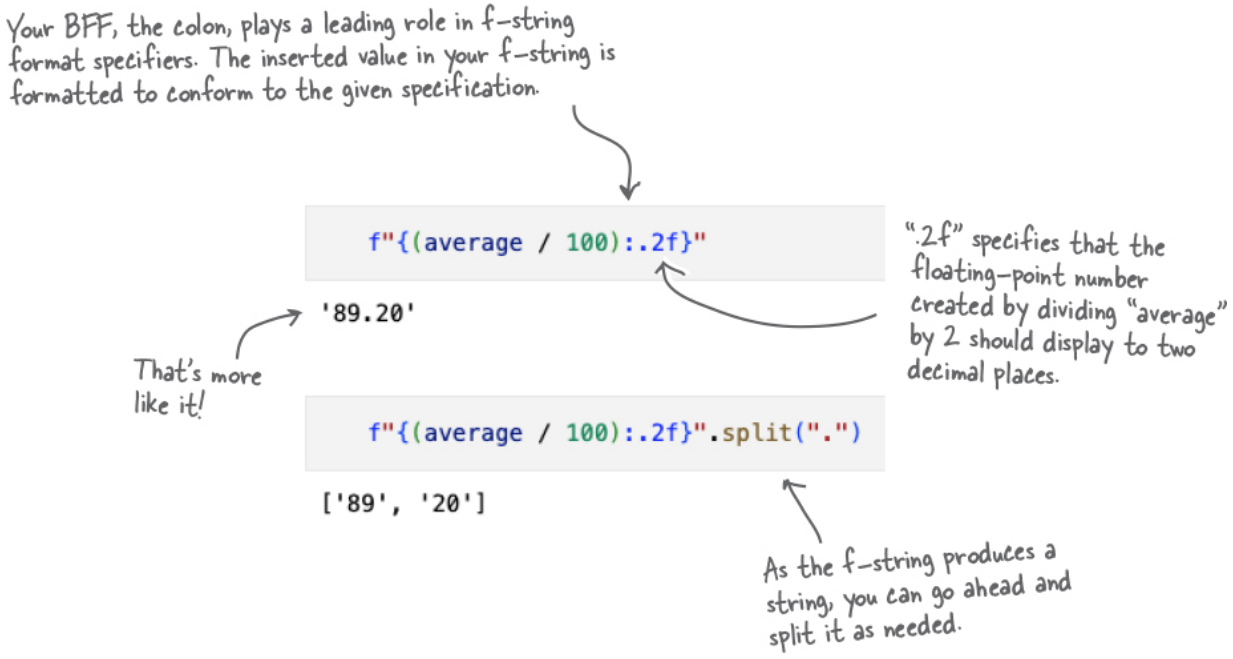
Pointing at the screen, *Holmes* declares: *"You should've used an f-string."*

Rounding is not what you want (in this case)

When it comes to performing rounding with the **round** BIF, the emphasis is on correctly *performing* the calculation, not correctly *format-*

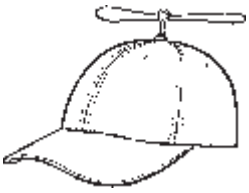
ting the result for display. It's as if rounding is too blunt an operation when it comes to using **round**'s result directly for display purposes.

As well as making the *production* of strings easy, Python's f-string technology provides sophisticated **string formatting specifiers**. There are a lot of these specifiers, and we're not going to get into describing what they all do here. Instead, let's look at the example *Holmes* told us to use in place of the offending line of code. Here it is:



Assigning the split f-string to `mins_secs` and `hundredths` (as you did with the offending line of code) gives you a drop-in replacement which fixes the display wrinkle and solves the coding mystery. Clever old *Holmes*, eh?

GEEK NOTE



If you want to know a bit more about f-string format specifiers, as suggested earlier, start with the examples included in the Python docs: https://docs.python.org/3/reference/lexical_analysis.html#f-strings.

TEST DRIVE

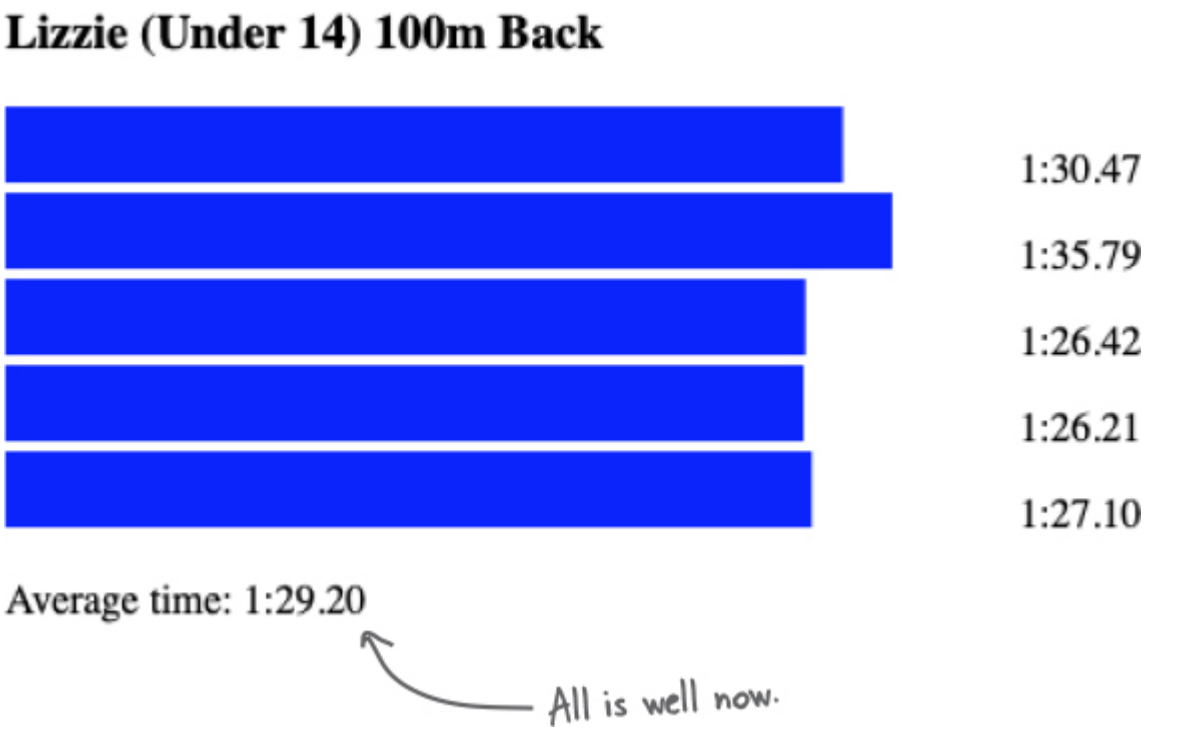


Adjust the `get_swim_data` function in your `swimclub.py` module to replace the line of code which used the **round** BIF with the f-string code from the previous page. Here's the snippet of code you need to change:

```
                ⋮
average = statistics.mean(converts)
mins_secs, hundredths = f"{(average / 100):.2f}".split(".")
mins_secs = int(mins_secs)
                ⋮
```

This is the replacement line of code

With the `swimclub.py` module saved, restart your `Charts.ipynb` notebook, then re-run the code cells which produce Lizzie's bar chart. When you do, you'll see the same chart as last time, but the annoying wrinkle is fixed and the average time is display correctly:

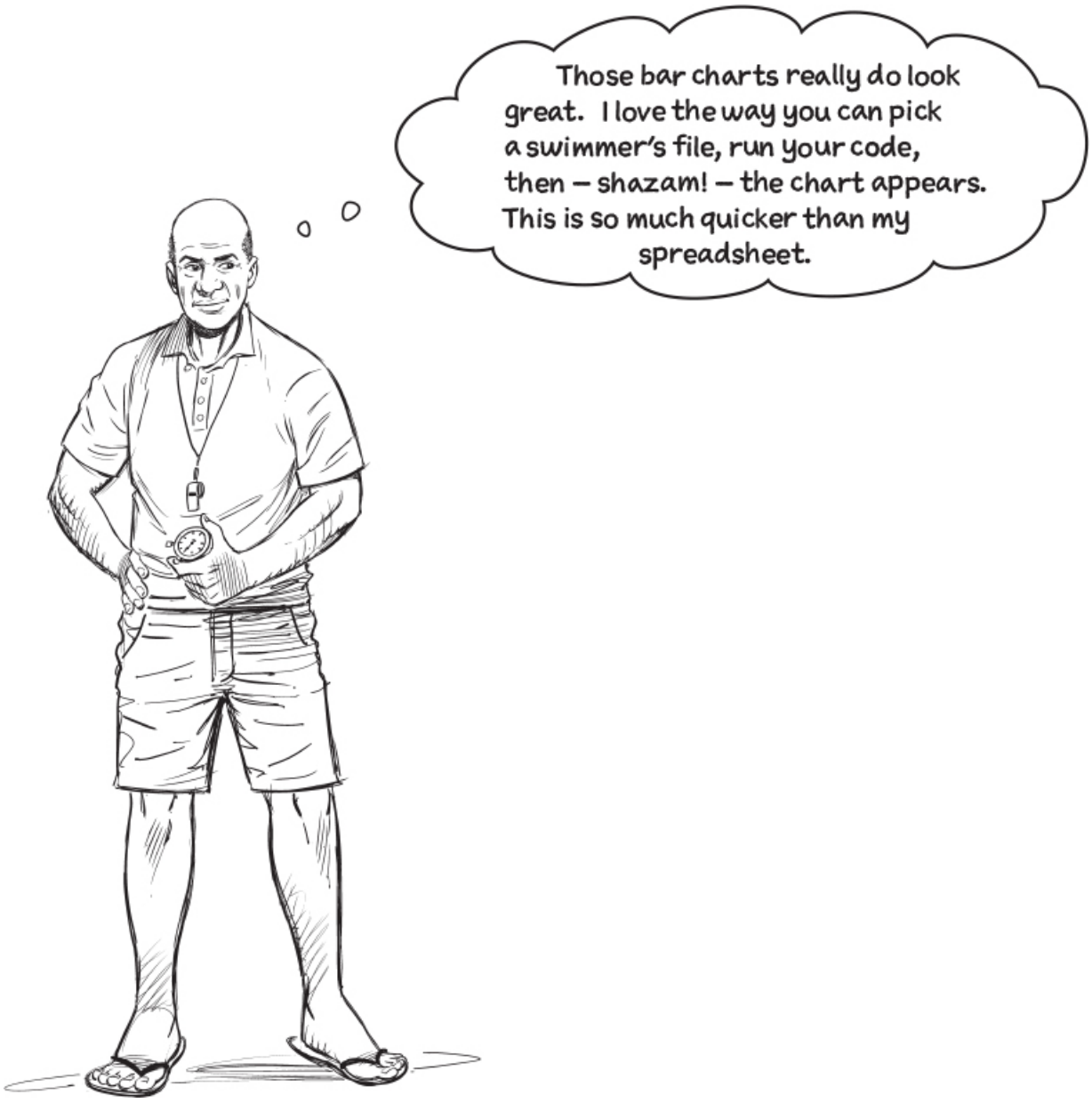


Things are progressing well…

With this latest fix applied, you now have a mechanism which, given any swimmer's filename, can produce an associated bar chart. This is a major step towards providing the Coach with a system which replaces his trusty clipboard and spreadsheet program.

Just how you'll provide this functionality to the Coach is still an open question. But, let's not sweat those details right now. Bask in the glory of your beautiful HTML bar charts. We'll work out how to deliver your functionality to the Coach in a little while.

Until then, relax, take a break, review this chapter's summary, and don't forget to have a go at this chapter's crossword.



CHAPTER REVIEW BULLET POINTS

- Although this book isn't designed to teach you **HTML**, as a modern Python programmer, you'll find it hard not to come across processing HTML at some point.
- In addition to generating a complete HTML page, this chapter also utilizes **SVG** markup to draw the bars on the Coach's charts.
- The PSL comes with the **webbrowser** module which, as the name suggests, lets Python interact with your favorite web browser. The **webbrowser** module can read HTML from a file, as well as from a web address.
- When it comes to building **formatted strings**, Python gives you choices. In addition to building strings with the venerable concatenation operator (`+`), the cool kids (as well as us old fogies) increasingly reach for **f-strings** for this type of thing.
- f-strings make it easy to include the value of a variable in a formatted string thanks to **interpolation**. Think "insert into" and you'll be fine. Curly braces are used to indicate where in the formatted string the value of the variable should go. Remember: in Python, curly braces typically surrounds data, *not* code.
- The **hfpy_utils** module provide a conversion function which can convert a value from one range into a value in another range. Called (in a feat of unknowable imagination) `convert2range`, this function was used to ensure the width of the bars on the Coach's chart didn't end up being thousands of pixels wide.

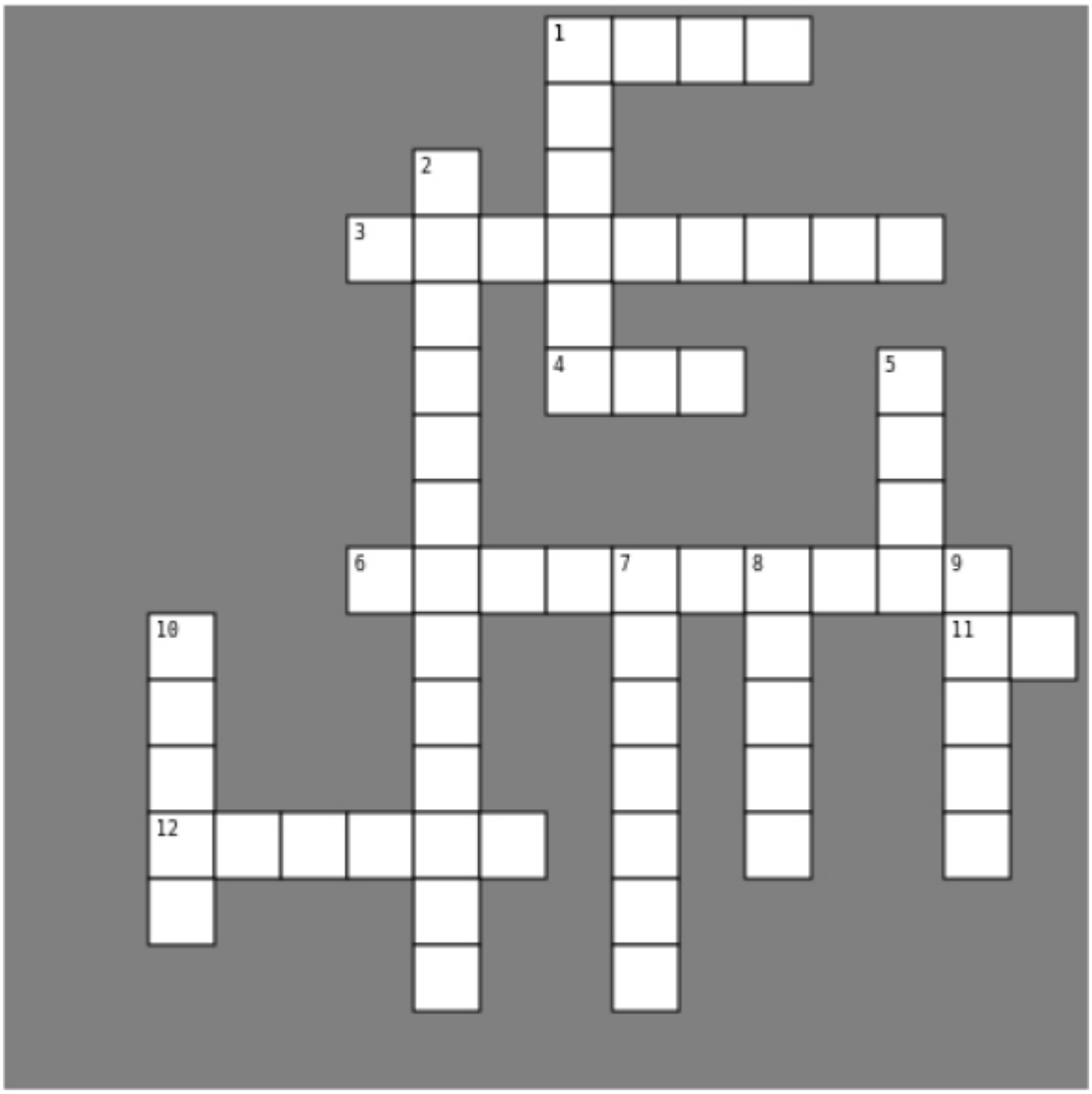
- If you think we're really smart, what which creating the `convert2range` function, think again. Take a look at the `htpy_utils.py` code (part of this book's download) and note the comment which tells you who the really smart person is.
- In addition to opening files for reading, the **open** BIF can also write to files, which is very useful when your code needs to create a new file. The "`w`" mode tells **open** to create/overwrite the named file.
- The **reverse** method (which comes built into every Python list) swaps the order of a named list *in-place*, and saved us from the embarrassment of showing the Coach a swimmer's chart with an incorrect bar order.
- To be clear, an "incorrect bar order" has nothing to do with ordering three beers instead of two.
- Once everything was working to specification, you created the `produce_bar_chart` function, then added it to your **swimclub** module, so that your new functionality can be more easily shared.
- Of course, having done that, you spotted a small bug (isn't that always the way?), which resulted in your hundredths of seconds value displaying incorrectly. The ever versatile f-string came to your rescue here once you'd realized that the **round** BIF was doing its best to ruin your day.

- The f-string mechanism's **format specifier** saved you from the wrath of **round**, just in time for you to show off your bar chart generating code to the Coach, who was thrilled with your progress. Phew!

The f-string Crossword



The answers to the clues are found in this chapter's pages, and the solution is on the next page.



Across

1. The markup language of the web.

3. What the “f ” in f-string stands for.

4. Short for scalable vector graphics.

6. The name of a PSL module which can automatically open a webpage.

11. The PSL module which talks to your operating system.

12. This swimmer's bar chart exhibited a little wrinkle which your use of an f-string fixed.

Down

1. The last name of the world's most famous code detective.

2. As a technique, this does not scale at all well when building strings.

5. The name of the parameter to the **print** BIF which saves data to disk.

7. This list method swaps the order.

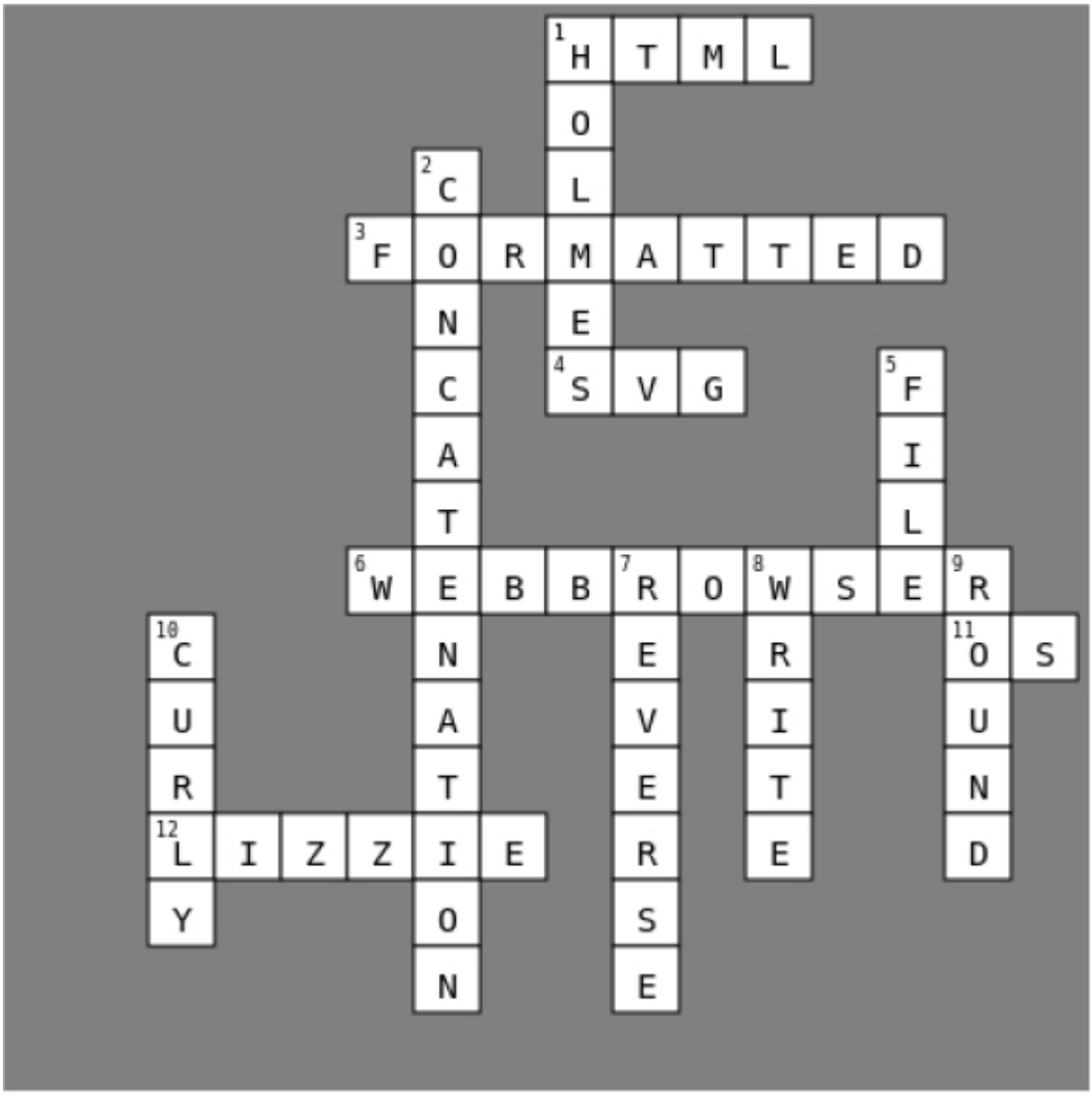
8. When used with the **open** BIF, the `"w"` value enables _____ mode

9. The use of this BIF resulted in the creation of an offending line of code.

10. With f-strings, _____ braces surround embedded variable names.

The f-string Crossword Solution





Across

1. The markup language of the web.

3. What the “f ” in f-string stands for.

4. Short for scalable vector graphics.

6. The name of a PSL module which can automatically open a webpage.

11. The PSL module which talks to your operating system.

12. This swimmer’s bar chart exhibited a little wrinkle which your use of an f-string fixed.

Down

1. The last name of the world’s most famous code detective.

2. As a technique, this does not scale at all well when building strings.

5. The name of the parameter to the **print** BIF which saves data to disk.

7. This list method swaps the order.

8. When used with the **open** BIF, the `"w"` value enables _____ mode

9. The use of this BIF resulted in the creation of an offending line of code.

10. With f-strings, _____ braces surround embedded variable names.

Chapter 5. Getting Organized: *Data structure choices*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 5 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



Your code needs to put its in-memory data somewhere…

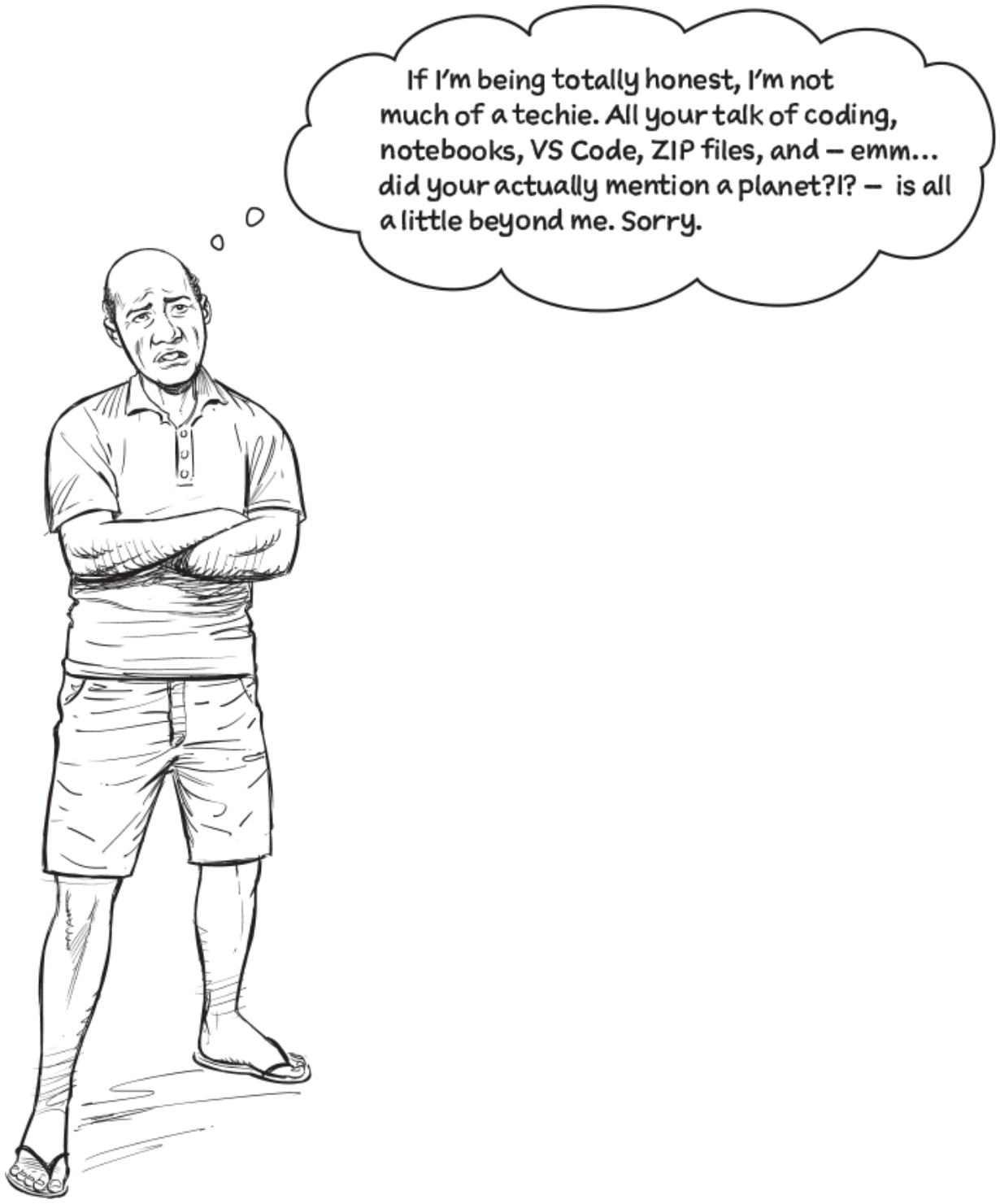
And when it comes to arranging data **in memory**, your choice of which data structure to use can be critical, and is often the difference between a messy solution that *works* and an **elegant** solution that

works well. In this chapter, you'll learn about another of Python's built-in data structures, the **dictionary**, which is often combined with the ubiquitous list to create **complex** data structures. The Coach needs an easy way select any swimmer's data, and when you put the Coach's data in a dictionary, lookups are a breeze!

Your code's beginning to take shape, but…

Over the last four chapters, you've learned just enough Python to be able to create code which, when given a swimmer's filename, reads the file's data, processes it, then turns it into a HTML-viewable bar chart. This is great work.

But (isn't there *always* a but?) running *your* code within *your* notebook on *your* computer is great for you, but *not* for the Coach. It's still an open question as you how you're going to deliver the developed functionality to the Coach…



Not to worry, Coach, we hear you.

Leave the technical bits'n'pieces to us.

We've made a mental note to ensure the delivered solution is as easy to use as possible for the Coach.

Cubicle Conversation

Sam: OK, folks, what's the best way to give this developed functionality to the Coach?

Alex: Maybe we should install everything on the Coach's laptop?

Mara: I like that idea, as all the tools we're using a free. What's not to like?

Sam: Hang on, let's take a step back.

Mara: Why?

Sam: On second thoughts, we might be jumping the gun a little, as our code only works for one swimmer's file, not any one of them.

Alex: Not to mention *all* of them.

Mara: I vote to get this in front of the Coach as soon as possible, with no more delays.

Sam: I agree with that sentiment, but we have to do this right, so that things are easy for the Coach.

Alex: We could write a **for** loop which quickly creates all 60 bar charts and pops each into an individual tab…

Mara: I don't think anyone will thank you for that, as that many browser tabs is going to be unmanageable.

Sam: Think about what the Coach needs to do…

Alex: Produce bar charts quickly, surely?

Sam: Yes, eventually. But where does the Coach start?

Alex: Emmmm…

Mara: By picking a swimmer to work with, right?

Sam: Yes. At the moment, the code we have works for a single swim file for Darius, and no one else. The Coach needs to start by selecting a swimmer to work with.

Alex: Of course! And once we know the swimmer's name, we can grab their file, then our code runs as-is to draw the graph.

Mara: And – fingers crossed – we'll have a happy Coach.

Sam: That's the plan!



The Head First Coders: Alex, Mara, and Sam.



Yes, that would be a good start.

If you can display a list of the swimmers names to the Coach so one can be selected, you'd be off and running.

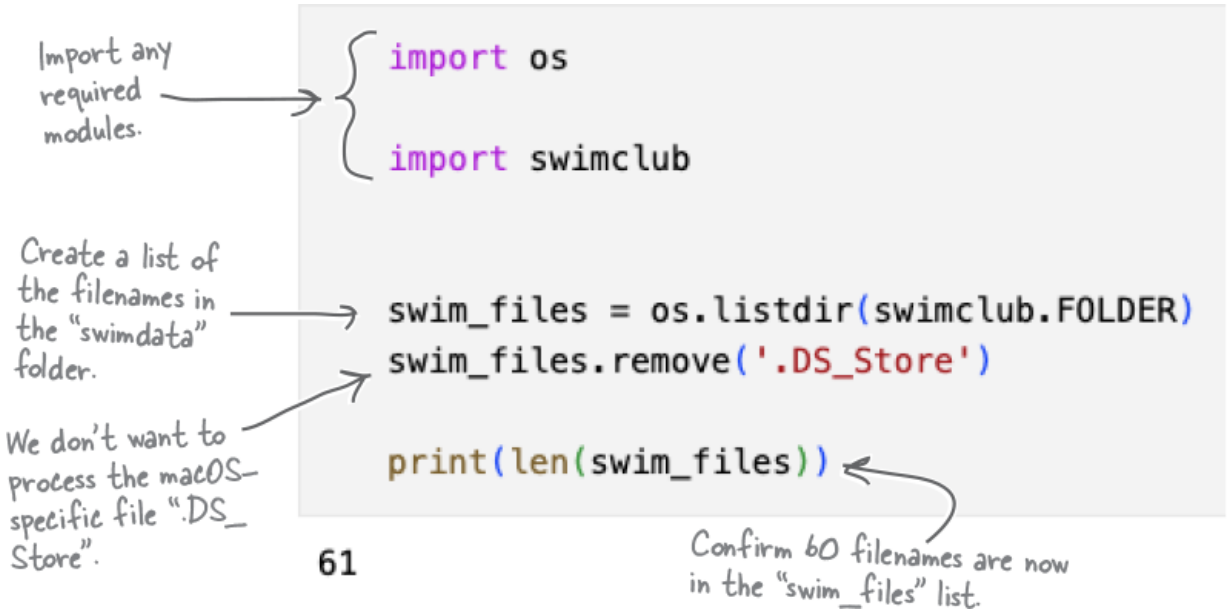
Get to know the data you'll be working with

If you take a quick look at all the files in your `swimdata` folder, you'll likely notice that every swimmer (regardless of age-category) has a *unique* name. That is, there's only one Darius, one Ruth, one Ali, and so on. This makes what you're about to do easier than it would be if you had, for instance, a Dave swimming in the under 17 age category and *another* Dave swimming in under 12. But, let's not worry about those type of complications right now.

As the Head First Coders are fond of saying: *Work with the data you have.*

Let's extract a list of swimmers names

Recall this code from an earlier chapter which, given the name of a folder, returns a list of files contained therein:



Let's process all the files in your `swim_files` list to extract the unique list of swimmers. This feels like a job for the **for** loop.

EXERCISE

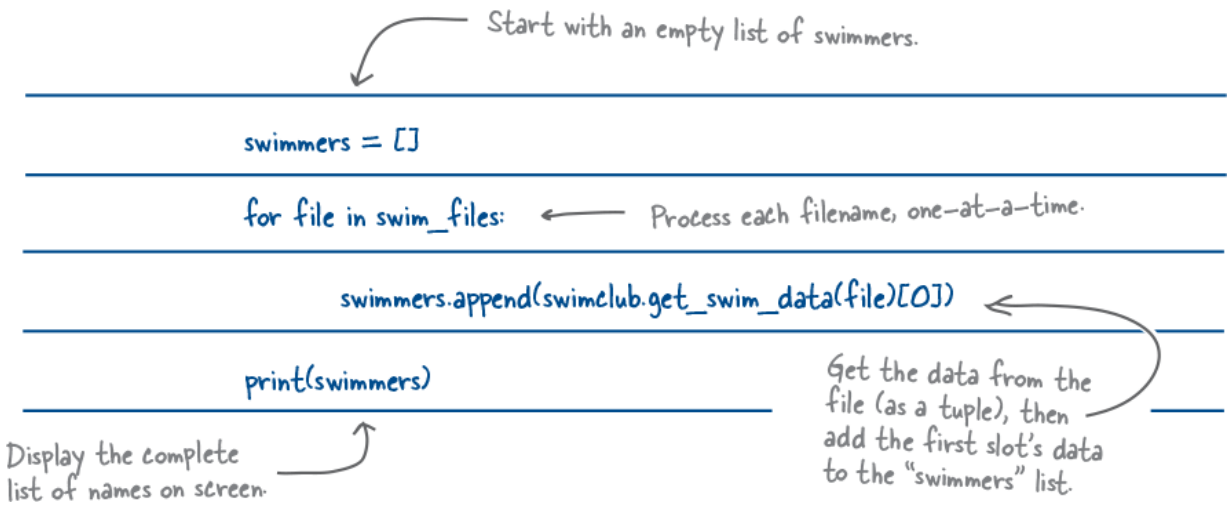


Create a new notebook called `OrganizingData.ipynb`, and add the code above into the first code cell, being sure to press **Shift+Enter** to execute those five lines of code. In the next cell, create a new, empty list called `swimmers`, then construct a **for** loop which processes all the files in the `swim_files` list. On each iteration, add the swimmer's name to the `swimmers` list. Once done, display the list of names from the `swimmers` list on screen. Write the code you came up with in the space below:

EXERCISE SOLUTION



You were asked to create a new notebook called `OrganizingData.ipynb`, then add the provided code into the first code cell, being sure to press **Shift+Enter** to execute those five lines of code. In the next cell, your were further asked to create a new, empty list called `swimmers`, before constructing a **for** loop which processes all the files in the `swim_files` list. On each iteration, you needed to add the swimmer's name to the `swimmers` list. To conclude, you needed to display the list of names from the `swimmers` list on screen. Here's our code. How does your code measure up?



TEST DRIVE



Here the results we ended up with on our screen when we ran our code:

```
swimmers = []
for file in swim_files:
    swimmers.append(swimclub.get_swim_data(file)[0])
print(swimmers)
```

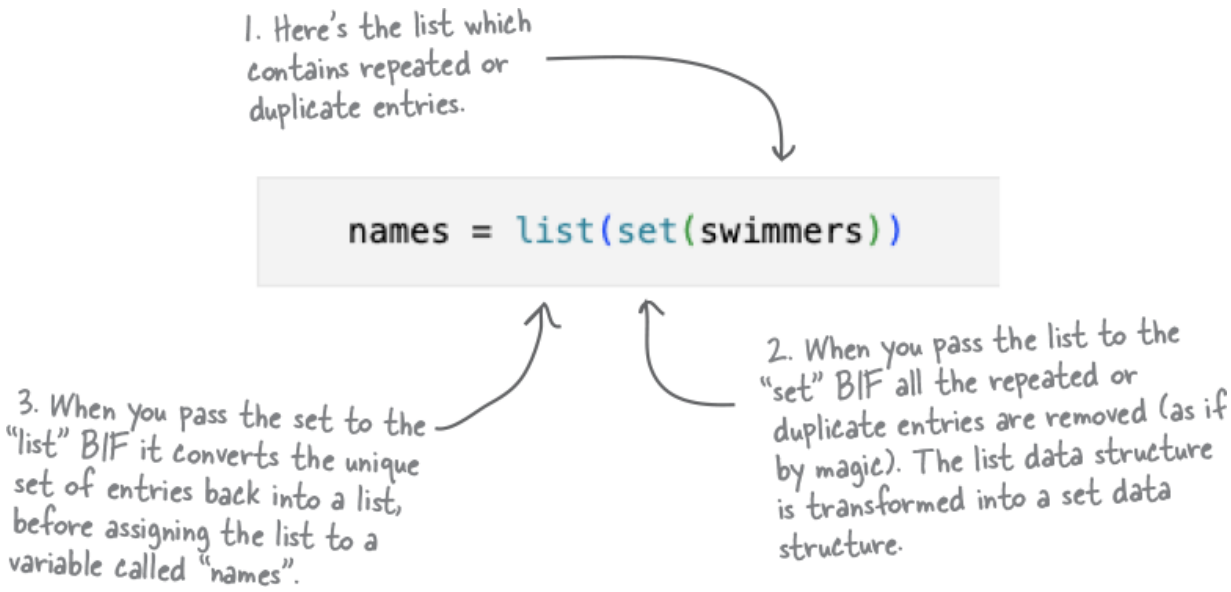
The "get_swim_data" function returns a tuple whose first slot contains the swimmer's name. That's what [0] accesses.

```
['Hannah', 'Darius', 'Owen', 'Mike', 'Hannah', 'Mike', 'Mike', 'Abi', 'Ruth', 'Tasmin', 'Erika', 'Ruth',
'Abi', 'Maria', 'Elba', 'Tasmin', 'Abi', 'Abi', 'Mike', 'Ali', 'Ruth', 'Chris', 'Ali', 'Darius', 'Ruth',
'Aurora', 'Katie', 'Alison', 'Ruth', 'Emma', 'Calvin', 'Darius', 'Mike', 'Emma', 'Tasmin', 'Blake', 'Abi',
'Chris', 'Blake', 'Katie', 'Bill', 'Darius', 'Dave', 'Alison', 'Lizzie', 'Katie', 'Katie', 'Katie',
'Lizzie', 'Tasmin', 'Katie', 'Dave', 'Erika', 'Calvin', 'Calvin', 'Carl', 'Bill', 'Katie', 'Blake', 'Erika',
'Katie']
```

This is a list of swimmer names. But, it is *all* 60 of them, with many of them repeated. You need to somehow remove all those repeated and duplicate names!

The list-set-list duplicate removing trick

When faced with a situation where you have a list of data which contain many repeated or duplicate entries, a useful *trick* is to quickly convert the list to a **set** which, rather conveniently, removes all duplicate or repeated entries. Once the set has done it's thing, you can convert the set back into a list. It's as straightforward as it sounds:



TEST DRIVE



Go ahead and type the above line of code into your next code cell, then execute it. Then use the **print** BIF to display the `names` list on screen:

```
names = list(set(swimmers))
```

← The list–set–list trick.

```
print(names)
```

```
['Carl', 'Elba', 'Calvin', 'Erika', 'Owen', 'Ali', 'Abi', 'Aurora', 'Mike', 'Darius',
'Maria', 'Dave', 'Hannah', 'Ruth', 'Chris', 'Emma', 'Bill', 'Alison', 'Katie',
'Tasmin', 'Blake', 'Lizzie']
```

The list of swimmer's names with any duplicates or repeated entries removed.

```
len(names)
```

```
22
```

A quick check with the Coach confirms there are 22 swimmers in the club.


Surely that trick is inefficient? Would it not be better to build that list of unique names in the "for" loop on-the-fly? Or... do you have something against "if" statements?!?

Those are valid observations, for sure.

If you'd known you were going to have to concern yourself with repeated entries *ahead of time*, you may have written your **for** loop like this:

```
swimmers = []
for file in swim_files:
    name = swimclub.get_swim_data(file)[0]
    if name not in  swimmers:
        swimmers.append(name)
print(swimmers)
```

This loop produces the same 22 element list in `swimmers` as the list-set-list trick put into `names` (albeit in a different order – it is the *same* list of swimmer names). But, you didn't know you needed to worry about those duplicates until *after* you'd executed your original **for** loop… so the *list-set-list trick* was a quick'n'dirty fix. What's that they say about hindsight?

The Coach now has a list of names

Whether you use the *list-set-list trick* or amend your original **for** loop to build the unique list as you go, you end up with the 22 names the Coach needs.

To confirm that both these techniques produce the same results, you can use the **sorted** BIF to display your lists as an ordered copy, like so:

Using the "print" BIF displays the data horizontally across the screen.

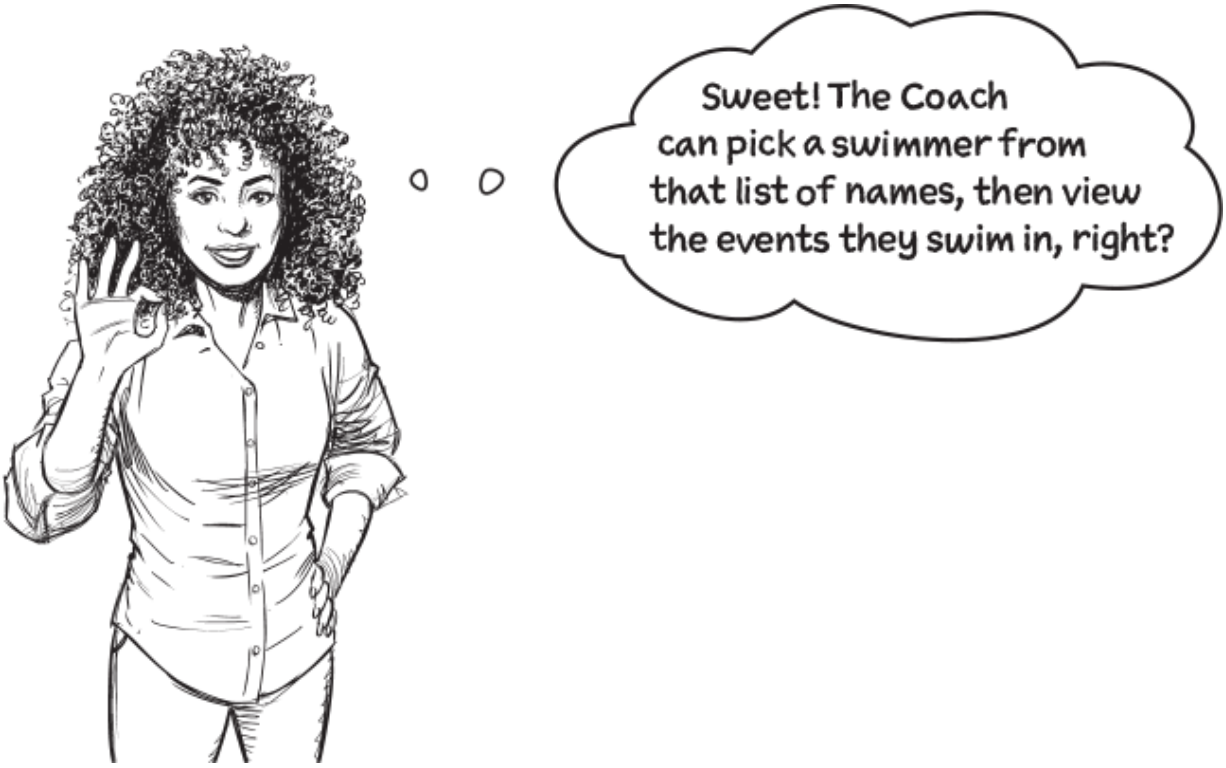
```
    print(sorted(names))
```

```
['Abi', 'Ali', 'Alison', 'Aurora', 'Bill', 'Blake', 'Calvin', 'Carl', 'Chris',
'Darius', 'Dave', 'Elba', 'Emma', 'Erika', 'Hannah', 'Katie', 'Lizzie', 'Maria',
'Mike', 'Owen', 'Ruth', 'Tasmin']
```

```
    print(sorted(swimmers))
```

```
['Abi', 'Ali', 'Alison', 'Aurora', 'Bill', 'Blake', 'Calvin', 'Carl', 'Chris',
'Darius', 'Dave', 'Elba', 'Emma', 'Erika', 'Hannah', 'Katie', 'Lizzie', 'Maria',
'Mike', 'Owen', 'Ruth', 'Tasmin']
```

Note: the "sorted" BIF produces an ordered *copy* of your list data. The original ordering in your lists remains unchanged.



Yes, getting the list is the first step.

Once a swimmer is identified, we can further remember the name of their data files, then use the data in those file to draw the correct number of graphs. The problem is: *How do we do this if all we know – at the moment – is the swimmer's name?*

A small change makes a "big" difference

Here's the original code which displays the swimmer's names extracted from each of the files in the `swimdata` folder:

Remember: You're only interested in the first item of data returned from the function, i.e., the swimmer's name, which is added to the "swimmers" list on each iteration.

```
swimmers = []
for file in swim_files:
    swimmers.append(swimclub.get_swim_data(file)[0])
print(swimmers)
```

Recall that the above code displays 60 names on screen, and you can successfully deploy the *list-set-list trick* to remove any unwanted repeat data. Adjusting the code ever so slightly allows you to remember not only the swimmer's name, but also the names of their associated filename, like so:

The swimmer's associated filename.

The swimmer's name.

```
swimmers = []
for file in swim_files:
    swimclub.get_swim_data(file)[0]
    swimmers.append( (swimclub.get_swim_data(file)[0], file) )
```

Did you spot the parentheses around the appended data? When you use parens in this way, you create a tuple.



Yes, but there's a bit of a catch.

Each tuple contains the swimmer's name *as well as* their associated filename, and each filename is unique. Regrettably, the *list-set-list trick* no longer removes any repeated swimmer names from the `swimmers` list: there are no duplicated entries in `swimmers` any more. That's the catch. You'll now need to write *another* **for** loop to process the data in `swimmers` to determine the unique list of names.

TEST DRIVE

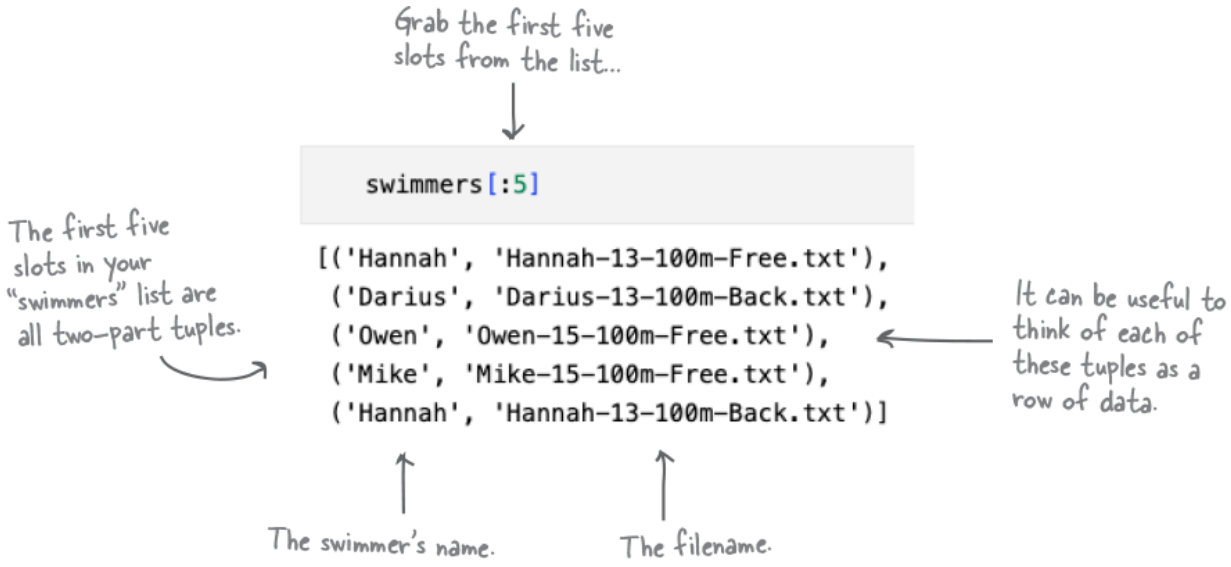


The assumption is you've added (and executed) the code from the second code cell shown on the previous page to your most-recent notebook. The **len** BIF confirms the `swimmers` list-of-tuples has 60 slots:

```
len(swim_files)
```

```
60
```

An extension to Python's square bracket notation lets you take a look at (in this example) the first five slots in your newly created `swimmers` list:



Displaying the first five tuples/rows in the list is enough to demonstrate the problem: Hannah appears twice above due to the fact that Hannah's filenames differ. Of course, there's only one Hannah in the club, and to get back to the point where you have a list of unique names to display to the Coach, you have to process all the rows in the `swimmers` list to remove the repeated and duplicate entries. This works, but feels somewhat wasteful:

```
names = []
for row in swimmers:
    if row[0] not in names:
        names.append(row[0])
print(sorted(names))
```

Start with an empty "names" list, then process all the rows in the "swimmers" list-of-tuples to build a unique list of names. Note the use of square brackets to select the first object in each tuple. It works, but looks a little brittle.

```
['Abi', 'Ali', 'Alison', 'Aurora', 'Bill', 'Blake', 'Calvin', 'Carl', 'Chris',
'Darius', 'Dave', 'Elba', 'Emma', 'Erika', 'Hannah', 'Katie', 'Lizzie', 'Maria',
'Mike', 'Owen', 'Ruth', 'Tasmin']
```

The unique list of names (again).


Hang on a second. I get what you're doing here, but isn't there an awful lot of repeated iteration going on? It's like loop after loop after loop...

That's a good point. There is a lot of looping going on.

As you're using a list here (or, more correctly, a list of two-part tuples), you often need to rely on the **for** loop to scan the *entire* list when trying to find the data you want. This isn't the case when you know where in the list (or tuple) a specific piece of data is.

For instance, in the most-recent *Test Drive*, you used `row[0]` to grab the swimmer's name because you know the name is the first item of each tuple row. But, if, for example. you need to extract all of Dave's data from your `swimmers` list-of-tuples, you have to use a loop as you have no clue where in the list-of-tuples Dave's data is without first searching for it. Code like this not only finds Dave's tuple rows, but also reports the relevant slot numbers from the `swimmers` list-of-tuples:

```
for n, event in enumerate(swimmers):
    if event[0] == "Dave":
        print(n, event)
```

```
42 ('Dave', 'Dave-17-100m-Free.txt')
51 ('Dave', 'Dave-17-200m-Back.txt')
```

And, of course, you're using *another* loop here, which means your iterating through the entire list-of-tuples *again*.

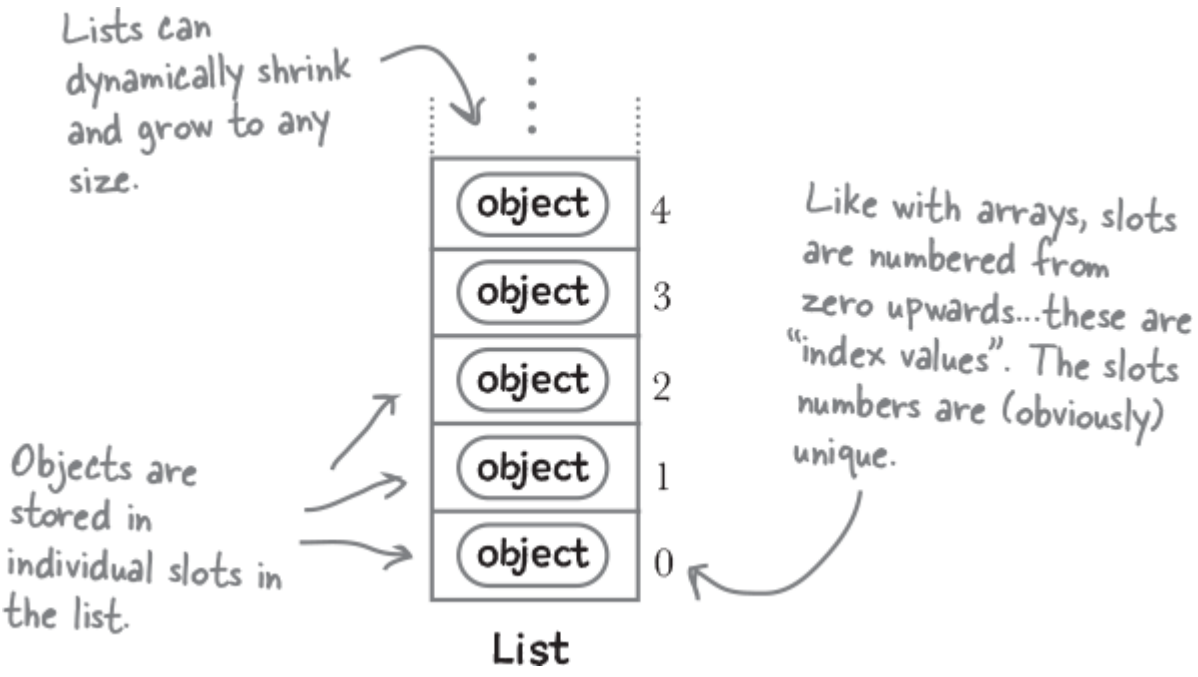


Perform super fast lookups with dictionaries

In addition to lists, tuples, and sets, Python provides a fourth built-in data structure which goes by the name **dictionary**. Depending on the programming language you've previous experience with, you may know Python's dictionary by another name, such as *associative array*, *map*, *hash*, or *symbol table*. Regardless of the name used, this data structure provides optimized lookup functionality against a multi-row table of data.

To help you understand what a dictionary is, let's compare it to a list.

Conceptually, here's what a list looks like:



Here's a… em, eh… *list* of things to know about lists

1 Lists are like arrays on steroids

What differentiates a list from an array (in Python, anyway) is their ability to shrink and grow to any size dynamically. Python handles all the messy memory management details for you.

2 Lists are a sequenced collection of Python objects

You can put anything in a list slot – *anything*. As everything in Python is an object, any object can be stored in a list, with each slot num-

bered with an index value which starts from zero.

3 Lists are a perfect fit when the order of your data is important to you

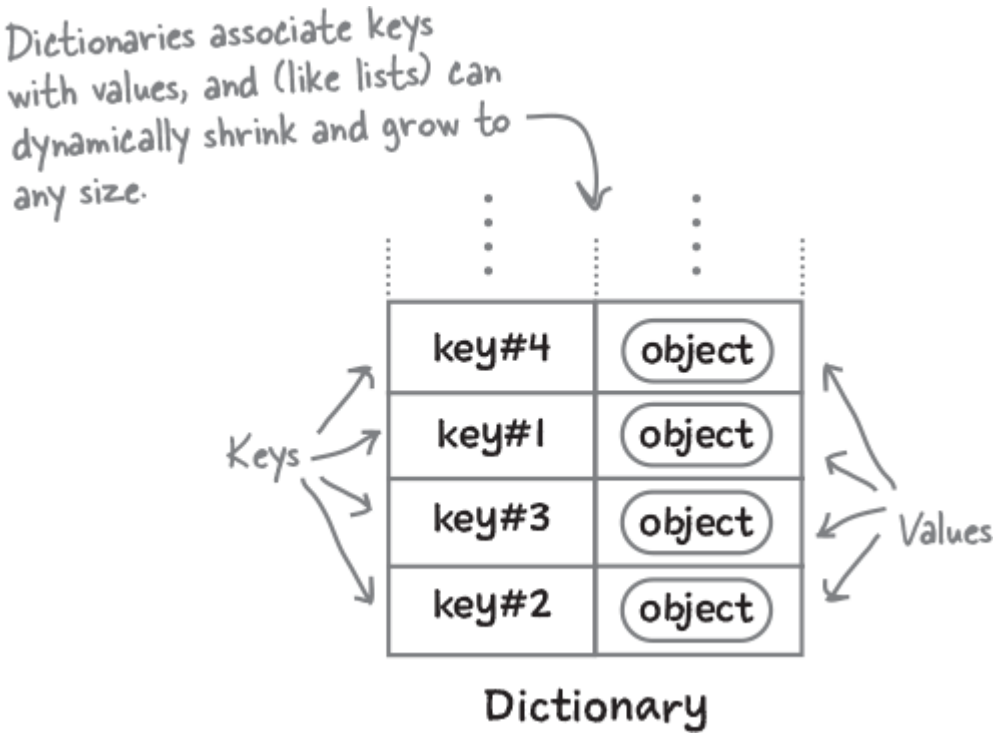
If you need to keep your data in a specific order, there is no better data structure to use than a list (even though you could also use a tuple but, remember: tuples are immutable).

4 List can be processed in many different ways

You can work with the data in any list using the square bracket notation, the list methods, Python's BIFs, and loops (with Python's **for** loop a near perfect fit with lists).

Dictionaries are key/value lookup stores

Compare the list diagram from the last page with a similar conceptual representation of Python's dictionary:



Here's a bunch of things to know about dictionaries:

1 Dictionaries are like a two-columned list

But unlike lists, which use a numeric index to identify the slots which contain objects, dictionaries use keys to identify values, with the values being *any* Python object.

2 A dictionary can contains any number of rows of data

Each key/value pair represents a row of data: a **key** is associated with a **value**. Just like indices in lists are unique, dictionary keys need to be unique for the lookup to work.

3 Dictionaries provide speedy lookup functionality

When provided with a key, your dictionary looks up and returns the associated value. Thanks to the underlying functionality used, these lookups do *not* search using iteration.

4 Dictionaries can be processed in many different ways

You can work with the data in any dictionary using a dictionary-specific version of the square bracket notation, the dictionary methods, Python's BIFs, and loops.

Fireside Chats



Tonight's talk: **LIST chats with DICTIONARY on the topic of selecting the best data structure…**

LIST:	**DICTIONARY:**
Well, look who has decided to make an appearance in (cough, cough) the *sixth* chapter?	
	Hello, List, how are you?
How am I? *How am I?* For the four chapters before this one, I've been working, I tell you. Working *flat out* to keep the Coach's data in order.	
	Which is what you're really good at: keeping order.
Don't you patronize me, you lookup... upstart!	
	Now, there's no need to resort to name-calling. I'm simply stating that when the order of data is important, there's no better data structure than you, List.

No better…? NO BETTER?!? Oh, yes… I'm great when it comes to ordering data. Em, eh… thanks Dictionary. And I can do lookups, too, if you're happy with a little bit of sequential searching. I'm the ultimate all-rounder, IMHO.

Well, let's agree to disagree on that one, as there's no better built-in lookup data structure than me.

As I said: *upstart*. Humph. [Turns to look away from Dictionary].

Ah, now, come on, you know it's true. A lookup is always going to beat a sequential search, hands-down. But a lookup table is probably the worst choice when it comes to remembering the order in which data was created. In that

	case, we all defer to you, the mighty list.
[Turns back around]. I like the sound of that: The Mighty List™. I think I'll put that on a T-Shirt.	
	[Winking] Just don't tell Tuple and Set…
[Laughing] Don't get me started on those two…	
	OK, just best to keep in mind that we all have our uses, and we all have things we do waaaaay better than anything else.
Do you think I should got with a V or a crew neck on the T-Shirt?	
	Are you even listening to me?!?
Listening to you? I'm The Mighty List™… everyone	

needs to listen to me!

Oh, my, I knew this was a bad idea. List never listens. Now… where did I put Tuple's number?

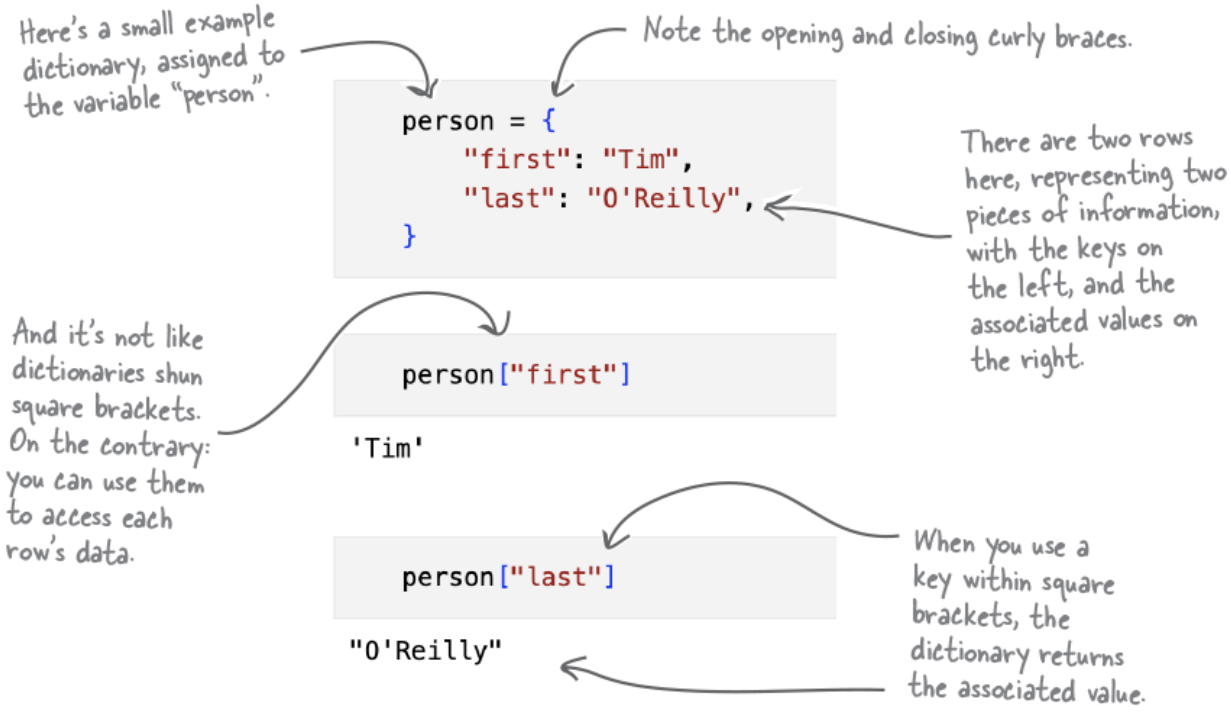


Yes. They go together.

Of course, it's not that you haven't seen curly braces used with data before. Back in Chapter 0, you used curly braces to surround your `deck` of playing cards, creating a set in the process.

Unlike with sets, which surround a list of unique values with curly braces, Python's dictionary surrounds *rows of data* with curly braces. As introduced earlier, each row has two parts, a **key**-part and a **value**-part, and each part is separated from the other by your BFF: *the colon*.

Let's take a look at a quick example.



ANATOMY OF BUILDING A DICTIONARY



1 Dictionaries surround data with curly braces

In Python, curly braces surround data, *not* code. Here's how to create an empty dictionary.

```
names = {}
```

2 There are no rows here

This dictionary is empty of any rows of data, so there are no key/value pairs in the `names` dictionary right now.

3 Use square brackets to add a row of data

With dictionaries, the key goes between the square brackets, which is then assigned an *associated* value, which adds a row of data to the dictionary.

```
names["Dave"] = "Dave-17-100m-Free.txt"
```

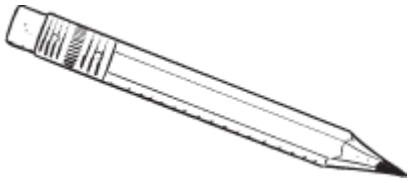
4 The key

5 The value

6 The dictionary has dynamically grown by one row

And that row's value can be accessed using `names["Dave"]`. Cool, eh?

SHARPEN YOUR PENCIL



Let's take a moment to apply what you already know about dictionaries to the problem at hand, namely using a dictionary to store the unique names of all the Coach's swimmers. Follow along in your notebook to determine the four lines of code needed, writing in each line of code into the spaces below.

Begin by creating a new, empty dictionary called `swimmers`:

1

Working within the same code cell in your notebook, provide the first line of a **for** loop which iterates over all the files in the `swim_files` list:

2

Within the block of the **for** loop, provide the line of code which calls the `get_swim_data` function, remembering the swimmer's name in a variable called `name`:

3

With the `name` variable assigned the current swimmer's name, use `name` as a new dictionary key and assign an empty list as the dictionary value:

4

THERE ARE NO DUMB QUESTIONS

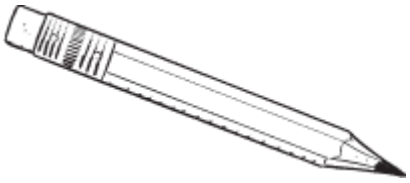
Q: Can dictionaries store anything as their value-part?

A: Dictionary values can be any Python object. Just like with lists, the value stored in a dictionary row can be *any* Python object. This means as well as storing the likes of strings and numbers as your dictionary values, it also possible to store lists, tuples, set, and/or dictionaries. This opens up all types of in-memory data management arrangements.

Q: Can dictionaries have any amount of keys?

A: Yes, but there is a caveat: each key needs to be a unique, so you can't have duplicate keys. The reason for this has to do with the main dictionary use-case, which is as a lookup table. If you think about things for any length of time, you'll quickly realize that allowing duplicate key values can only lead to a whole heap of trouble…

SHARPEN YOUR PENCIL SOLUTION



You were to take a moment to apply what you already know about dictionaries to the problem at hand, namely using a dictionary to store the unique names of all the Coach's swimmers. Following along in your notebook, you were asked to determine the four lines of code needed, writing in each line of code into the spaces below, as we have done.

You were to begin by defining a new, empty dictionary called `swimmers`:

1 `swimmers = {}` ← The empty dictionary is assigned to the "swimmers" variable name.

Working within the same code cell, we asked you to provide the first line of a **for** loop which iterates over all the files in the `swim_files` list:

2 `for file in swim_files:` ← No change here, as this line of code has been used many times before.

Within the block of the **for** loop, you were to provide the line of code which calls the `get_swim_data` function, remembering the swimmer's name in a variable called `name`:

```
:
name, *_ = swimclub.get_swim_data(file)
```

3

The "*_" gobbles up the rest of the data, while the current swimmer's name is assigned to the "name" variable.

And, finally, with the `name` variable assigned the current swimmer's name, you were to use `name` as a new dictionary key and assign an empty list as the dictionary value:

```
swimmers[name] = []
```

4

The value of "name" is used as the row's key, while the value-part of the row is set to an empty list.



No, as dictionary keys must be unique.

This means that every time there's a duplicate name seen, that last line of code assigns an empty list to the already existing row. Duplicate dictionary keys are simply not possible.

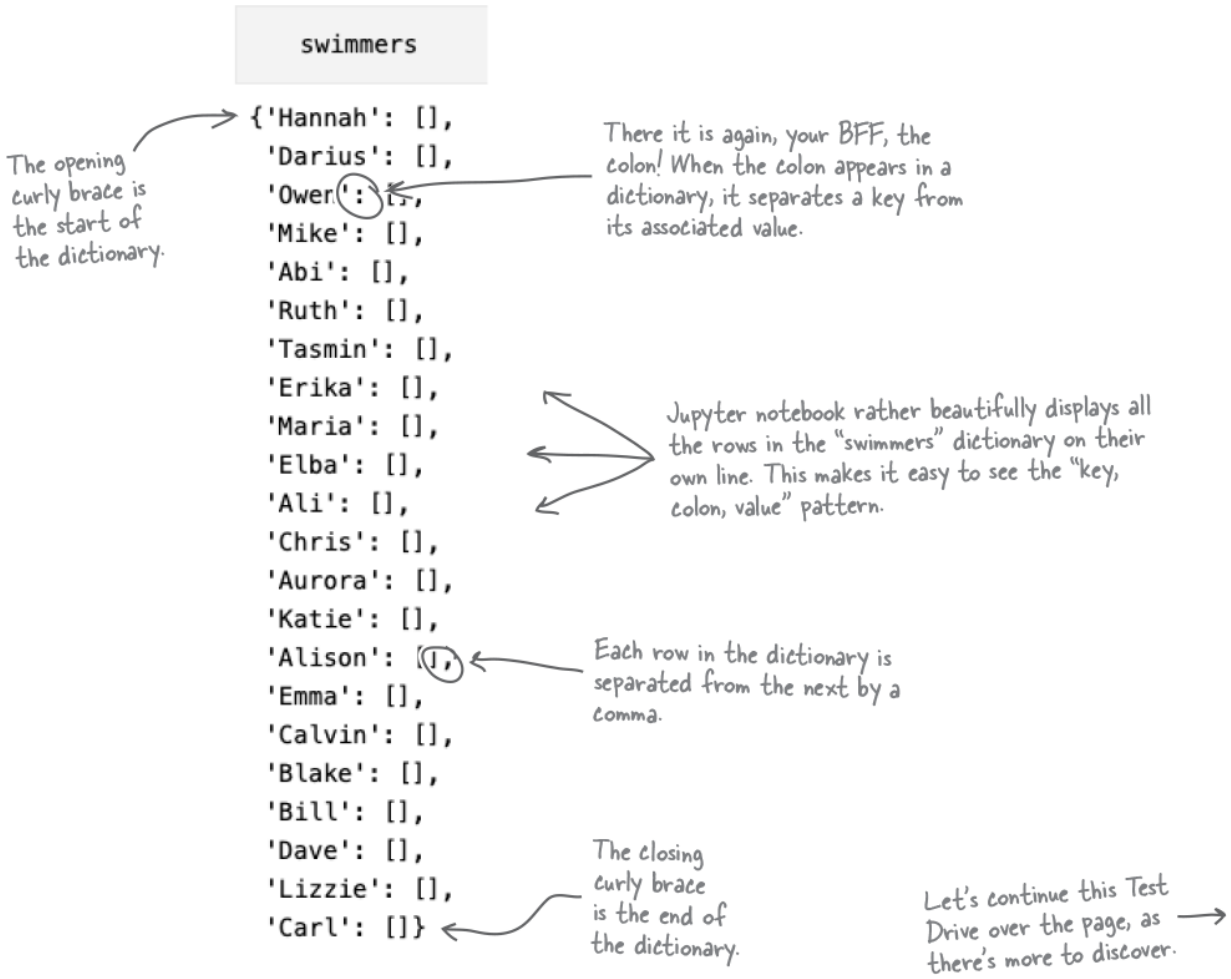
TEST DRIVE



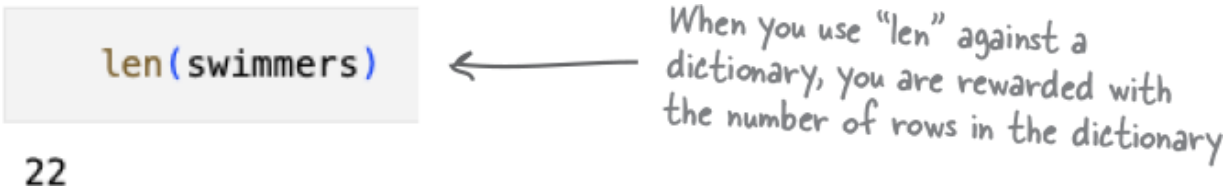
Let's take your latest code cell for a spin. Your code should match these four lines:

```
swimmers = {}
for file in swim_files:
    name, *_ = swimclub.get_swim_data(file)
    swimmers [name] = []
```

Go ahead and press **Shift+Enter** to run the above loop. In your next cell, type "swimmers", then press **Shift+Enter** again:



The BIFs work with dictionaries, too. Here's what the **len** BIF reports the dictionary's size as:



Take a look at what is displayed when you use the **sorted** BIF with your dictionary:

```
print(sorted(swimmers))
```

```
['Abi', 'Ali', 'Alison', 'Aurora', 'Bill', 'Blake', 'Calvin', 'Carl', 'Chris',
'Darius', 'Dave', 'Elba', 'Emma', 'Erika', 'Hannah', 'Katie', 'Lizzie', 'Maria',
'Mike', 'Owen', 'Ruth', 'Tasmin']
```

Unless told otherwise, the "sorted" BIF returns an ordered list of the dictionary's keys.

You can access any row's value-part using a key within square brackets:

```
swimmers["Dave"]
```

The value associated with the "Dave" key is currently an empty list.

```
[]
```

And of course (your old friend) the **print dir** combo mambo reports the list of methods included with every dictionary:

```
print(dir(swimmers))
```

```
['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__ior__',
'__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__ror__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

Continuing to ignore the dunders, you're left with eleven methods which manipulate your dictionary's rows. More on these, later in this book, as and when you need them.

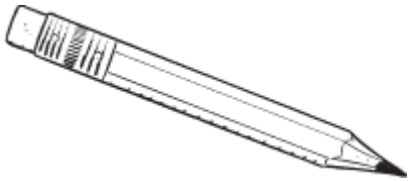


Yes, indeed we will.

It's almost as if you read our mind. Freaky.

The choice of an empty list as the value-part of the `swimmers` dictionary was no accident, as using a list opens up all sorts of possibilities. Specifically, the list can grow with the names of the data files associated with each swimmer. All that's needed to enable this is a change to your **for** loop code.

SHARPEN YOUR PENCIL

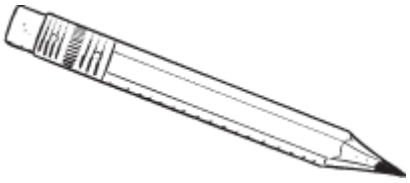


Here's the most-recent version of your **for** loop. Replace the last line with code that grows the associated list with the name of the swimmer's data files.

```
swimmers = {}
for file in swim_files:
    name, *_ = swimclub.get_swim_data(file)
    swimmers[name] = []
```

Write your replacement code here:

SHARPEN YOUR PENCIL SOLUTION



You were asked to replace the last line of your **for** loop with code that grows the list with the associated name of the swimmer's data files:

```
swimmers = {}
for file in  swim_files:
    name, *_ = swimclub.get_swim_data(file)
    swimmers [name] = []
```

You were to write your replacement code here. Did you end up with the same code as us?

```
if name not in swimmers:
    swimmers[name] = []
    swimmers[name].append(file)
```

An empty list is only ever assigned to a key when the key is not already in the "swimmers" dictionary.

Grow the list associated with the identified key by appending the name of the current data file.

TEST DRIVE



Go ahead and run the latest version of your **for** loop:

```
swimmers = {}
for file in swim_files:
    name, *_ = swimclub.get_swim_data(file)
    if name not in swimmers:
        swimmers[name] = []
    swimmers[name].append(file)
```

Then check to see the list of filenames associated with, for instance, Dave:

```
swimmers["Dave"]
```

```
['Dave-17-100m-Free.txt', 'Dave-17-200m-Back.txt']
```

Neat!



Think about what's going on here.

Here's the list-of-tuples looping code from earlier, which searches the *entire* list looking for Dave's files:

```
    for n, event inenumerate(swimmers):
        if event [0] == "Dave":
```

```
            print(n, event)

42 ('Dave', 'Dave-17-100m-Free.txt')
51 ('Dave', 'Dave-17-200m-Back.txt')
```

And here's the dictionary-of-lists code from the last page which gives you the same results using a lookup on the `swimmers` dictionary:

```
    swimmers["Dave"]

[' Dave-17-100m-Free.txt', 'Dave-17-200m-Back.txt
```

And here's why using a dictionary is better: that lookup is *lightning fast* no matter how big your dictionary gets, which is not something you can *ever* say about the sequential search of your list.

Dictionaries are optimized for speedy lookup

With the `swimmers` dictionary-of-lists populated with data, determining which data files belong to which swimmer is almost *too easy*:

```
    swimmers["Darius"]
```

```
['Darius-13-100m-Back.txt',
 'Darius-13-100m-Breast.txt',
 'Darius-13-100m-Fly.txt',
 'Darius-13-200m-IM.txt']
```

```
swimmers["Emma"]
```

```
['Emma-13-100m-Free.txt', 'Emma-13-100m-Breast.tx
```

```
swimmers["Calvin"]
```

```
['Calvin-9-50m-Fly.txt', 'Calvin-9-50m-Back.txt',
```

```
swimmers["Katie"]
```

```
['Katie-9-100m-Breast.txt',
 'Katie-9-100m-IM.txt',
 'Katie-9-50m-Fly.txt',
 'Katie-9-50m-Breast.txt',
 'Katie-9-50m-Back.txt',
 'Katie-9-50m-Free.txt',
 'Katie-9-100m-Free.txt',
 'Katie-9-100in-Back.txt']
```

There's no point in adding annotations to the above code cells, as it very clear what's happening. Identifying the swimmer by name within

the square brackets selects the associated list of filenames. There's nothing to it. Dictionary lookups are *super fast*.

Display the entire dictionary

When it comes to viewing the entire `swimmers` dictionary, typing "swimmers" into a code cell and pressing **Shift+Enter** displays the entire dictionary-of-lists on screen which we've *squashed* to make it fit on this page (sorry):

```
swimmers
```

```
{'Hannah': ['Hannah-13-100m-Free.txt', 'Hannah-13-100m-Back.txt'],
 'Darius': ['Darius-13-100m-Back.txt',
  'Darius-13-100m-Breast.txt',
  'Darius-13-100m-Fly.txt',
  'Darius-13-200m-IM.txt'],
 'Owen': ['Owen-15-100m-Free.txt'],
 'Mike': ['Mike-15-100m-Free.txt',
  'Mike-15-100m-Back.txt',
  'Mike-15-100m-Fly.txt',
  'Mike-15-200m-IM.txt',
  'Mike-15-200m-Free.txt'],
 'Abi': ['Abi-10-50m-Back.txt',
  'Abi-10-50m-Free.txt',
  'Abi-10-100m-Back.txt',
  'Abi-10-50m-Breast.txt',
  'Abi-10-100m-Breast.txt'],
 'Ruth': ['Ruth-13-200m-Free.txt',
  'Ruth-13-200m-Back.txt',
  'Ruth-13-100m-Back.txt',
  'Ruth-13-100m-Free.txt',
  'Ruth-13-400m-Free.txt'],
 'Tasmin': ['Tasmin-15-100m-Back.txt',
  'Tasmin-15-100m-Free.txt',
  'Tasmin-15-100m-Breast.txt',
  'Tasmin-15-200m-Breast.txt'],
 'Erika': ['Erika-15-100m-Free.txt',
  'Erika-15-200m-Breast.txt',
  'Erika-15-100m-Breast.txt'],
 'Maria': ['Maria-9-50m-Free.txt'],
 'Elba': ['Elba-14-100m-Free.txt'],
 'Ali': ['Ali-12-100m-Back.txt', 'Ali-12-100m-Free.txt'],
 'Chris': ['Chris-17-100m-Back.txt', 'Chris-17-100m-Breast.txt'],
 'Aurora': ['Aurora-13-50m-Free.txt'],
 'Katie': ['Katie-9-100m-Breast.txt',
  'Katie-9-100m-IM.txt',
  'Katie-9-50m-Fly.txt',
  'Katie-9-50m-Breast.txt',
  'Katie-9-50m-Back.txt',
  'Katie-9-50m-Free.txt',
  'Katie-9-100m-Free.txt',
  'Katie-9-100m-Back.txt'],
 'Alison': ['Alison-14-100m-Breast.txt', 'Alison-14-100m-Free.txt'],
 'Emma': ['Emma-13-100m-Free.txt', 'Emma-13-100m-Breast.txt'],
 'Calvin': ['Calvin-9-50m-Fly.txt',
  'Calvin-9-50m-Back.txt',
  'Calvin-9-50m-Free.txt'],
 'Blake': ['Blake-15-100m-Free.txt',
  'Blake-15-100m-Back.txt',
  'Blake-15-100m-Fly.txt'],
 'Bill': ['Bill-18-200m-Back.txt', 'Bill-18-100m-Back.txt'],
 'Dave': ['Dave-17-100m-Free.txt', 'Dave-17-200m-Back.txt'],
 'Lizzie': ['Lizzie-14-100m-Free.txt', 'Lizzie-14-100m-Back.txt'],
 'Carl': ['Carl-15-100m-Back.txt']}
```

The entire dictionary-of-lists contained within "swimmers". Everything is here, but it's not exactly easy on the eye, is it?

Flip the page to learn about a better way to view "swimmers".

The pprint module prett y-prints your data

```
import pprint

pprint.pprint(swimmers)
```

```
{'Abi': ['Abi-10-50m-Back.txt',
         'Abi-10-50m-Free.txt',
         'Abi-10-100m-Back.txt',
         'Abi-10-50m-Breast.txt',
         'Abi-10-100m-Breast.txt'],
 'Ali': ['Ali-12-100m-Back.txt', 'Ali-12-100m-Free.txt'],
 'Alison': ['Alison-14-100m-Breast.txt', 'Alison-14-100m-Free.txt'],
 'Aurora': ['Aurora-13-50m-Free.txt'],
 'Bill': ['Bill-18-200m-Back.txt', 'Bill-18-100m-Back.txt'],
 'Blake': ['Blake-15-100m-Free.txt',
           'Blake-15-100m-Back.txt',
           'Blake-15-100m-Fly.txt'],
 'Calvin': ['Calvin-9-50m-Fly.txt',
            'Calvin-9-50m-Back.txt',
            'Calvin-9-50m-Free.txt'],
 'Carl': ['Carl-15-100m-Back.txt'],
 'Chris': ['Chris-17-100m-Back.txt', 'Chris-17-100m-Breast.txt'],
 'Darius': ['Darius-13-100m-Back.txt',
            'Darius-13-100m-Breast.txt',
            'Darius-13-100m-Fly.txt',
            'Darius-13-200m-IM.txt'],
 'Dave': ['Dave-17-100m-Free.txt', 'Dave-17-200m-Back.txt'],
 'Elba': ['Elba-14-100m-Free.txt'],
 'Emma': ['Emma-13-100m-Free.txt', 'Emma-13-100m-Breast.txt'],
 'Erika': ['Erika-15-100m-Free.txt',
           'Erika-15-200m-Breast.txt',
           'Erika-15-100m-Breast.txt'],
 'Hannah': ['Hannah-13-100m-Free.txt', 'Hannah-13-100m-Back.txt'],
 'Katie': ['Katie-9-100m-Breast.txt',
           'Katie-9-100m-IM.txt',
           'Katie-9-50m-Fly.txt',
           'Katie-9-50m-Breast.txt',
           'Katie-9-50m-Back.txt',
           'Katie-9-50m-Free.txt',
           'Katie-9-100m-Free.txt',
           'Katie-9-100m-Back.txt'],
 'Lizzie': ['Lizzie-14-100m-Free.txt', 'Lizzie-14-100m-Back.txt'],
 'Maria': ['Maria-9-50m-Free.txt'],
 'Mike': ['Mike-15-100m-Free.txt',
          'Mike-15-100m-Back.txt',
          'Mike-15-100m-Fly.txt',
          'Mike-15-200m-IM.txt',
          'Mike-15-200m-Free.txt'],
 'Owen': ['Owen-15-100m-Free.txt'],
 'Ruth': ['Ruth-13-200m-Free.txt',
          'Ruth-13-200m-Back.txt',
          'Ruth-13-100m-Back.txt',
          'Ruth-13-100m-Free.txt',
          'Ruth-13-400m-Free.txt'],
 'Tasmin': ['Tasmin-15-100m-Back.txt',
            'Tasmin-15-100m-Free.txt',
            'Tasmin-15-100m-Breast.txt',
            'Tasmin-15-200m-Breast.txt']}
```

Import the "pprint" module from the PSL then use the "pprint.pprint" function to pretty-print your dictionary.

Not only is this output easier on the eye, but the "pprint" function also sorts the output by key – how cool is that?

Your dictionary-of-lists is easily processed

Now that the `swimmers` dictionary contains an individual list of file-names for every swimmer in the club, you're only a short **for** loop away from displaying any individual swimmer's set of bar charts. Here's how to do it for Calvin:

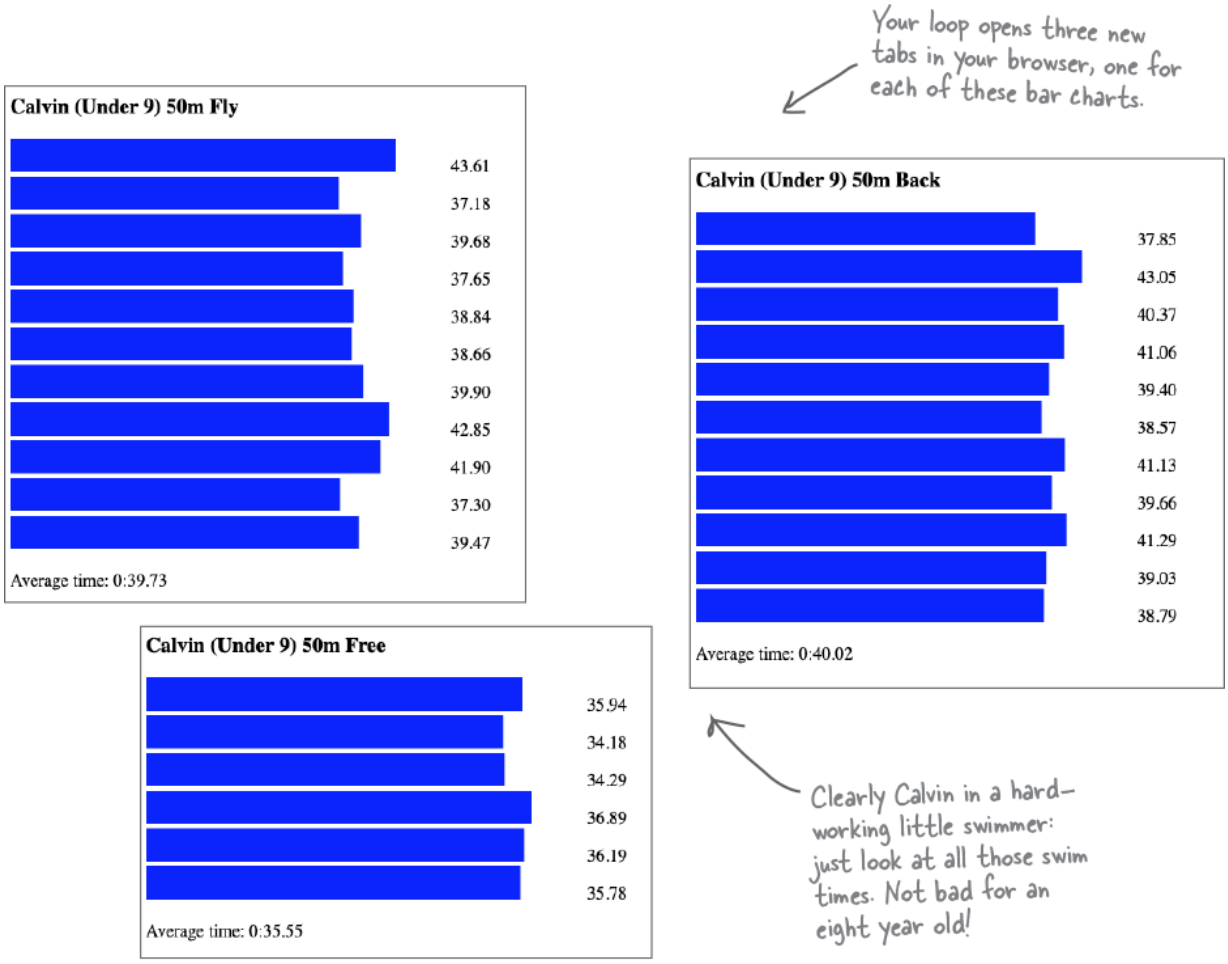
```
import os
import webbrowser

import swimclub


for fname in swimmers["Calvin"]:
    webbrowser.open("file://" + os.path.realpath(swimclub.produce_bar_chart(fname)))
```

There's nothing new here, as you've seen all this code (or something very close to it) before.

Calvin recorded times in three events during the most-recent swim sessions:



This feels like a major milestone

At this point, you've developed code which can process all of the data files in the `swimdata` folder, extract a list of swimmer names then, assuming a name is selected for the list, display all of the bar charts for the chosen swimmer. And your code does this in the blink-of-an-eye. This is big improvement over the Coach having to hand-enter all his swimmer's data into his spreadsheet program.

But, does this mean you're done?



Cubicle Conversation

Mara: This is great work, folks, but what happens now?

Sam: My vote is to get this into the Coach's hands as quickly as possible. Let's install everything the Coach needs onto his computer and let him have at it.

Alex: We could do that, but I'm not so sure we want to suggest the Coach run this code within a notebook. Remember: computers aren't really his thing, and having him mess about with our notebook's code

inside VS Code (or even a web browser) might head south very quickly.

Mara: That's a good point, but there's something to be said about getting this functionality in front of the Coach ASAP.

Sam: And think of the time the Coach will save?

Alex: Yes, but at what cost?

Sam: What do you mean… "at what cost"?

Alex: Well, if something goes wrong, such as the Coach deleting a code cell or changing a line of code, the "system" will break, resulting in the Coach losing confidence in us and our work. It might be hard to regain the Coach's trust after a simple mishap like that.

Sam: I guess you're right. The tools we use as developer's aren't really supposed to be "user-facing". What's needed here is something that let's the Coach do what he needs to do while keeping him away from our code.

Mara: I agree. Let's avoid creating extra work for ourselves if we can.

Alex: What about if we build the Coach webapp?

Mara & Sam: Of course! What a great idea.



Yes, that's the plan!

One of Python's strengths is its ability to build great webapps. And the good news is that doing so rarely takes "forever".

In fact, the work completed to date can be reused pretty much as-is, and whatever can't be used as-is can be easily adapted.

Here's a promise: we'll build a webapp for the Coach in a single chapter, which is starting in a few pages time. Before getting to that, though, there's the usual chapter summary to cast your eye over, then the obligatory crossword.

See you in the next chapter when you'll use Python to build a webapp!

CHAPTER REVIEW BULLET POINTS

- Python's list is great when you need to maintain the **order** of your data. However, things become problematic when you use a list for lookup, especially when your list grows. Can you spell s-e-q-u-e-n-t-i-a-l?
- Although we all love the **for** loop, using it with a list to provide lookup functionality is (more times than now) a *bad* idea.
- Even the use of the **in** operator (which we love nearly as much as we love **for**) needs to be considered carefully, as **in** also performs searches sequentially with a loop.
- Of course, we not saying lists aren't useful. Oh, but golly, they are. And in this chapter you used a list that *contained* tuples as well as *combined* a call to list with a call to set in a chain. Not that we're suggesting the set was "in chains", it's just that you used a function chain to call set on a list, then you called list again on the set. We called it the **list-set-list trick**.
- When it comes to performing lookups, there can be only one data structure of choice, and that's Python's **dictionary**.
- List sets, dictionaries surround data with curly braces. Unlike sets (which contain a collection of unque values), the data in a dictionary is arranged as a collection of **rows**, with a **key**-part and a **value**-part separated from each other by your BFF, the colon.

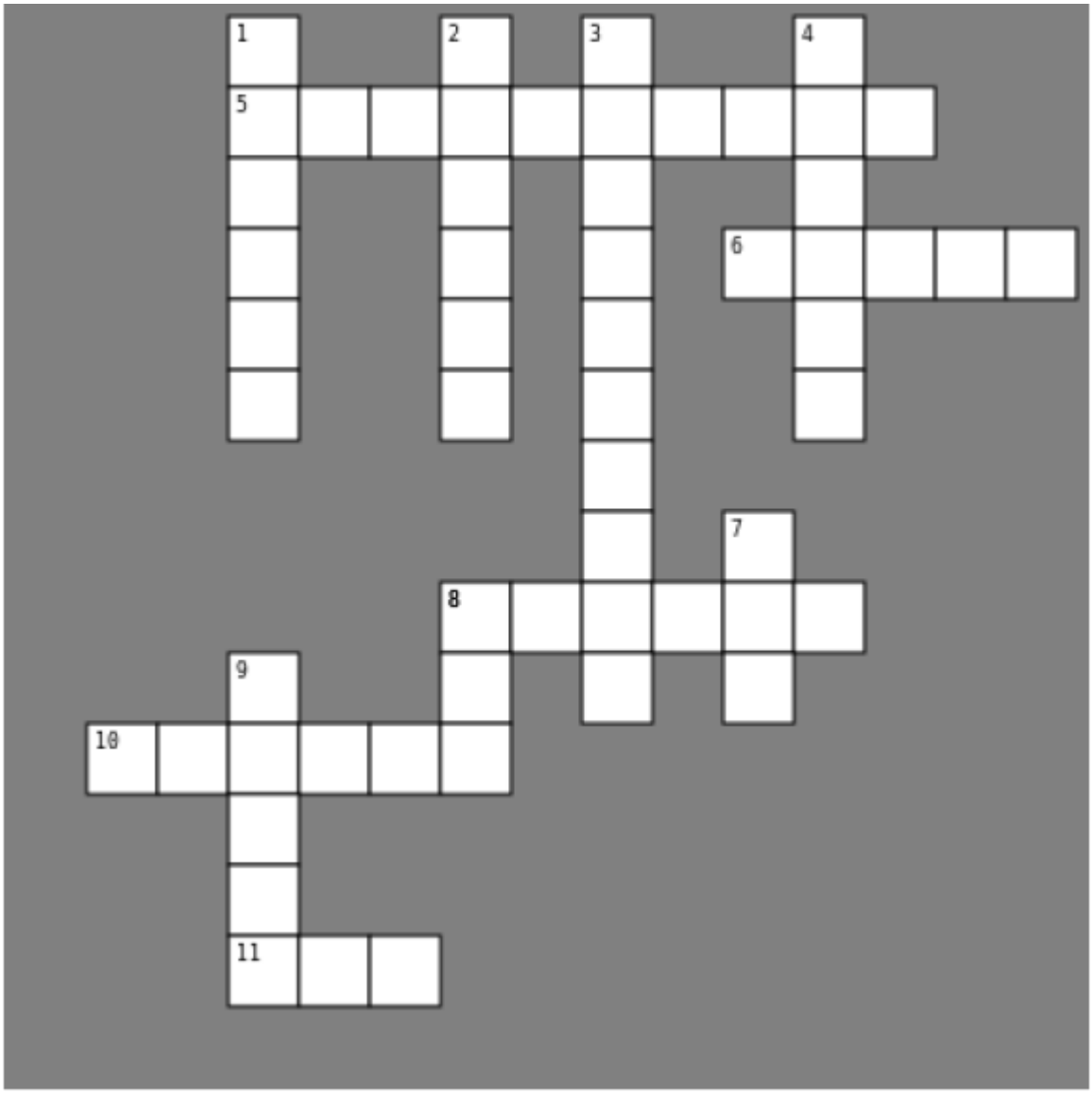
- One caveat with dictionaries is to remember that the keys must be **unique**: you cannot have a key appear more than once in your dictionary.
- Don't think for a second that lists and dictionaries don't get on. Far from it: they are *best-buddies*, and their **combined strengths** work well together.
- Square brackets also work with dictionaries. They can be used to **retrieve** a value given a key (with the key placed between the square brackets), and square brackets are also used to **assign** values to keys, creating new rows in the dictionary when the key doesn't already exist, replacing the assigned value otherwise.
- We may not have called this out specifically in this chapter (which was very bold of us), but dictionaries are another example of a **mutable** data structure, in that they can shrink and grow at run-time.
- As a dictionary value-part can be **any** Python object, it's possible to build **complex** data structures as required. A dictionary value-part can be a list or another dictionary. In fact, it can be anything you want. Remember: values can be any Python object.
- When a dictionary gets large and/or complex, Python can struggle to display the dictionary's data in a easy-to-read way. That's why the PSL provides the **pprint** module which prettily prints Python data structures.

- The dictionary is a foundational technlogy within the Python world, and you've only scratched the surface in this chapter. It's possible to do lots of things with dictionaries, as you'll see in upcoming chapters.

The Dictionary Crossword



As always, all the answers to the clues are found in this chapter (and the solution is over the page).



Across

5.The right hand side of this chapter's data structure _____ with the names on the left hand side of this chapter's data structure.

6. The name given to the right hand column of 3 down'.

8. A BIF which returns an ordered copy of whatever its given.

10. The PSL's pretty printer.

11. This chapter's data structure can be thought of as a collection of these (singular).

Down

1. He's a great little swimmer, despite being only eight years old.

2. A type of table which enables speedy retrieval of data.

3. Known as an associative array in some other languages.

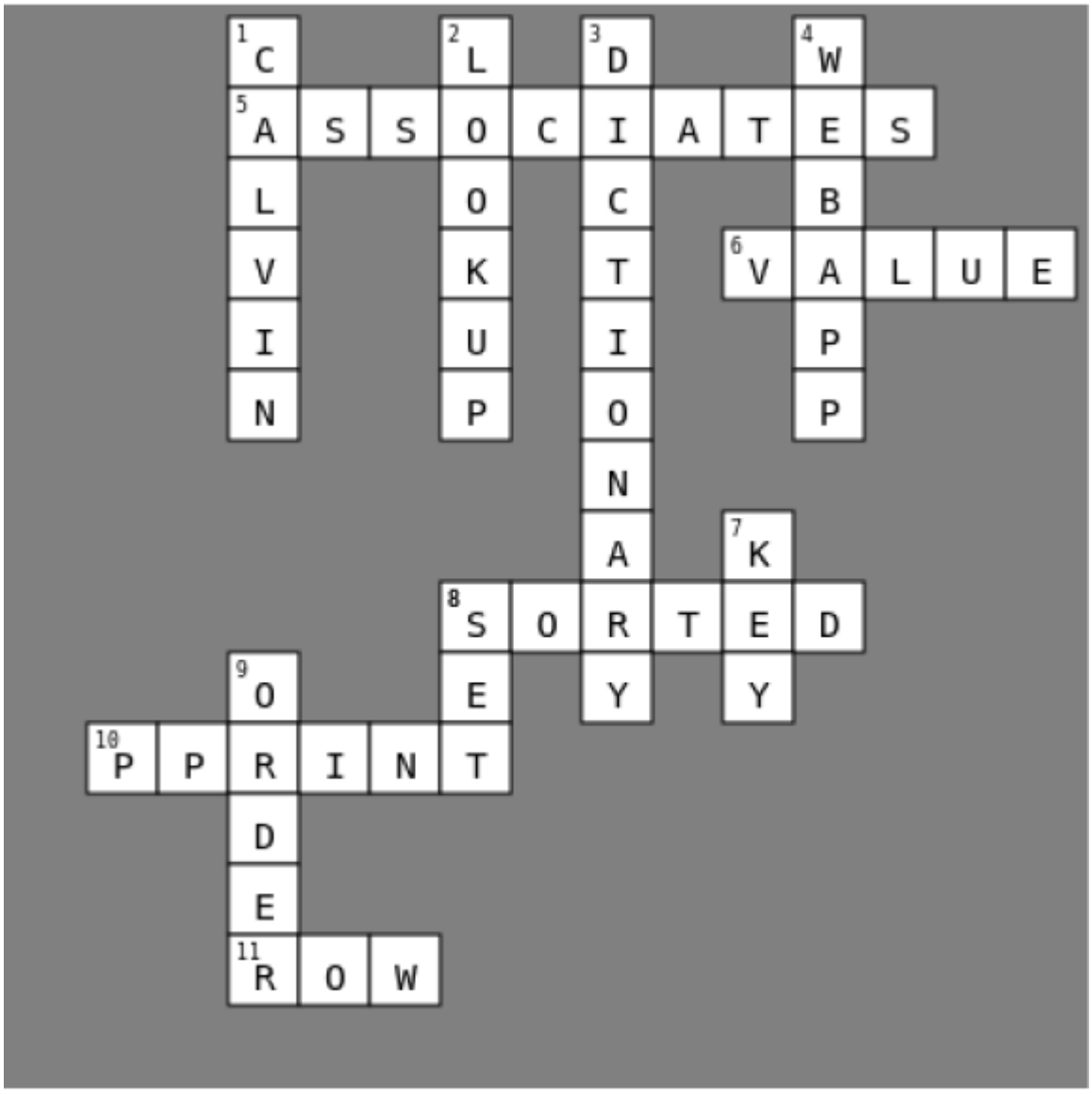
4. You'll build one of these in the next chapter.

7. The name given to the left hand column of 3 down.

8. This built-in data structure makes removing repeated data a doddle.

9. Lists are great at maintaining _____.

The Dictionary Crossword Solution


1 C
2 L
3 D
4 W
5 A S S O C I A T E S
L O C B
V K T 6 V A L U E
I U I P
N P O P
N
A 7 K
8 S O R T E D
9 O E Y Y
10 P P R I N T
D
E
11 R O W

Across

5.The right hand side of this chapter's data structure _____ with the ames on the left hand side of this chapter's data structure.

6. The name given to the right hand column of 3 down.

8. A BIF which returns an ordered copy of whatever its given.

10. The PSL's pretty printer.

11. This chapter's data structure can be thought of as a collection of these (singular).

Down

1. He's a great little swimmer, despite being only eight years old.

2. A type of table which enables speedy retrieval of data.

3. Known as an associative array in some other languages.

4. You'll build one of these in the next chapter.

7. The name given to the left hand column of 3 down.

8. This built-in data structure makes removing repeated data a doddle.

9. Lists are great at maintaining _____.



We're interrupting this book with an Important Ser vice Announcement. Please stand by…

Your brain may well be looking at the size of this chapter – the shortest so far – and thinking: *"That didn't take long. Maybe dictionaries aren't as important as all that other stuff "*.

If this is the case, banish all such thoughts from your mind!

Of all the programming technologies built into Python, the humble dictionary is by far the most important as it is used *EVERYWHERE* in Python. A good understanding of what is (and isn't) possible with dictionaries lets you exploit this important language feature, turning you from a good Python programmer into a *great* Python programmer.

You've only scratched the surface of what's possible with dictionaries in this chapter, and there's lots more to come, so stay tuned.

Bottom line: Lists are great, Tuples are useful, and Sets are cool, but Dictionaries are **outstanding**.

Normal programming resumes in just a moment…

Chapter 6. Building a Webapp: *Web Development*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 6 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

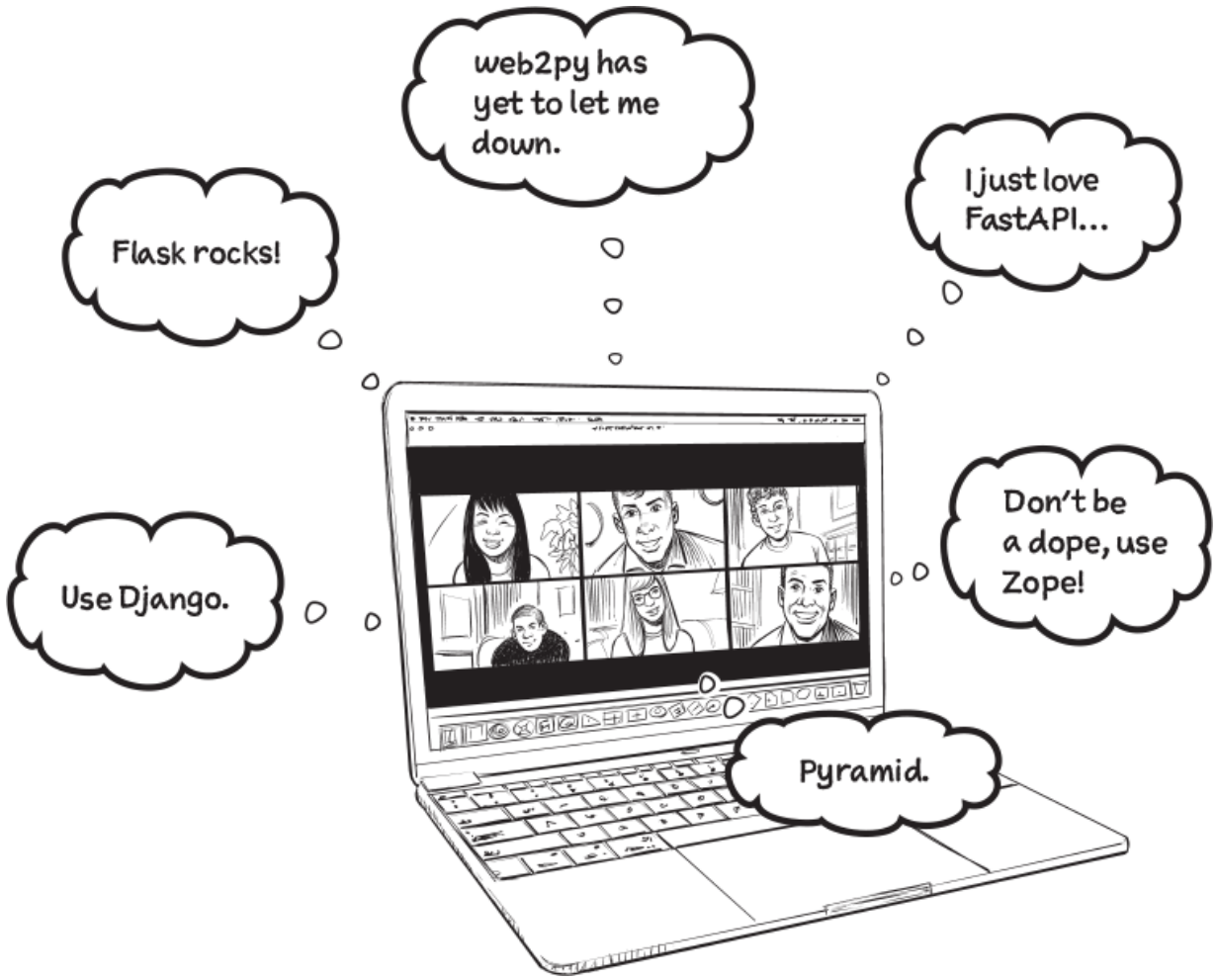


Figure 6-1.

Ask a bunch of programmers which web framework to use…

…and you'll likely get many conflicting answers. When it comes to web development, Python is not short of technology **choices**, each with a loyal and supportive developer community. In this chapter, you'll dive into **web development**, quickly building a webapp to view any swimmer's bar chart data. Along the way, you'll learn how to use HTML **templates**, function **decorators**, **get** and **set** HTTP methods,

and more. There's no time to waste, as the Coach is keen to show-off his new system. Let's get to it!

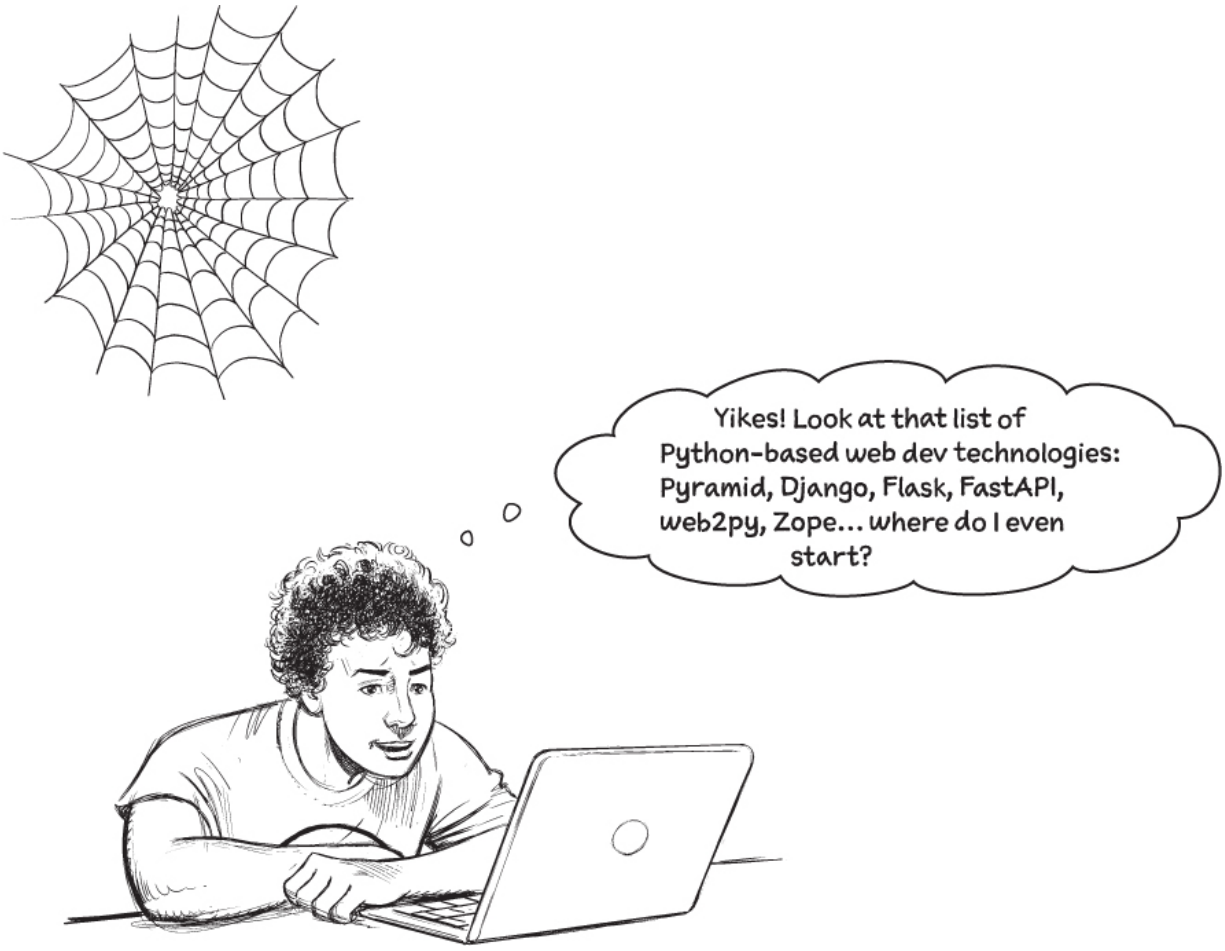


Figure 6-2.

There are lots of choices.

And, of course, depending on what you're planning do, some of these technologies are "better" than some of the others.

Let’s build the Coach’s webapp with Flask

NOTE

We made an Executive Decision. 😉

Flask has a well-deserved reputation for getting developers from zero to working webapp in no time at all, so it seems like the best option for us. The Coach is not going to wait for much longer, so you need to show results, fast!

To begin, let’s take a moment to install Flask.

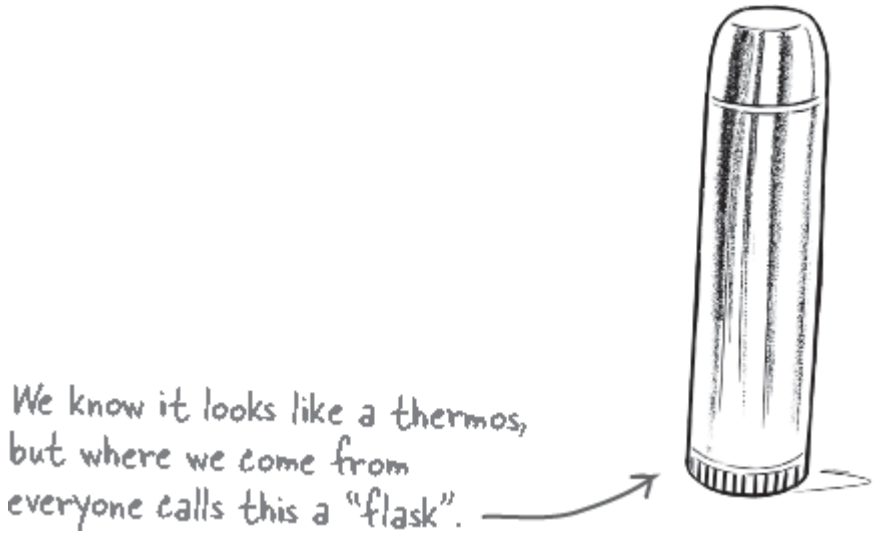


Figure 6-3.

Install Flask from PyPI

Flask is technically a *micro-framework*, providing just enough to get your webapp offthe-ground with the minimal amount of fuss. As an optional extra to Python, Flask is installable from **PyPI**, the Python Package Index. Performing the installation is integrated into Jupyter Notebook.

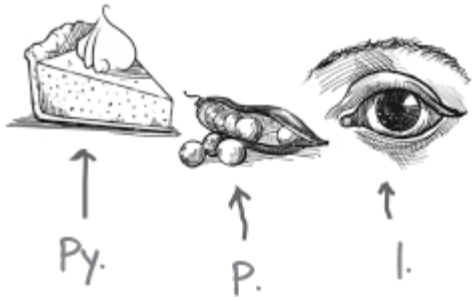


Figure 6-4.

As always, be sure to follow along as you work through this chapter.

Begin by creating a new sub-folder in your `Learning` folder called `webapp`.

Within VS Code, use **File**, then **Open Folder…** to select the just-created `webapp` folder as your current working directory. Then, create a *new* Jupyter Notebook called `WebappSupport.ipynb`.

This latest notebook will act as a place for you to experiment with bit-s'n'pieces of code as you build-out your first Flask webapp. For now, let's use this notebook to instruct Python to install Flask into your Python interpreter, using **pip**:

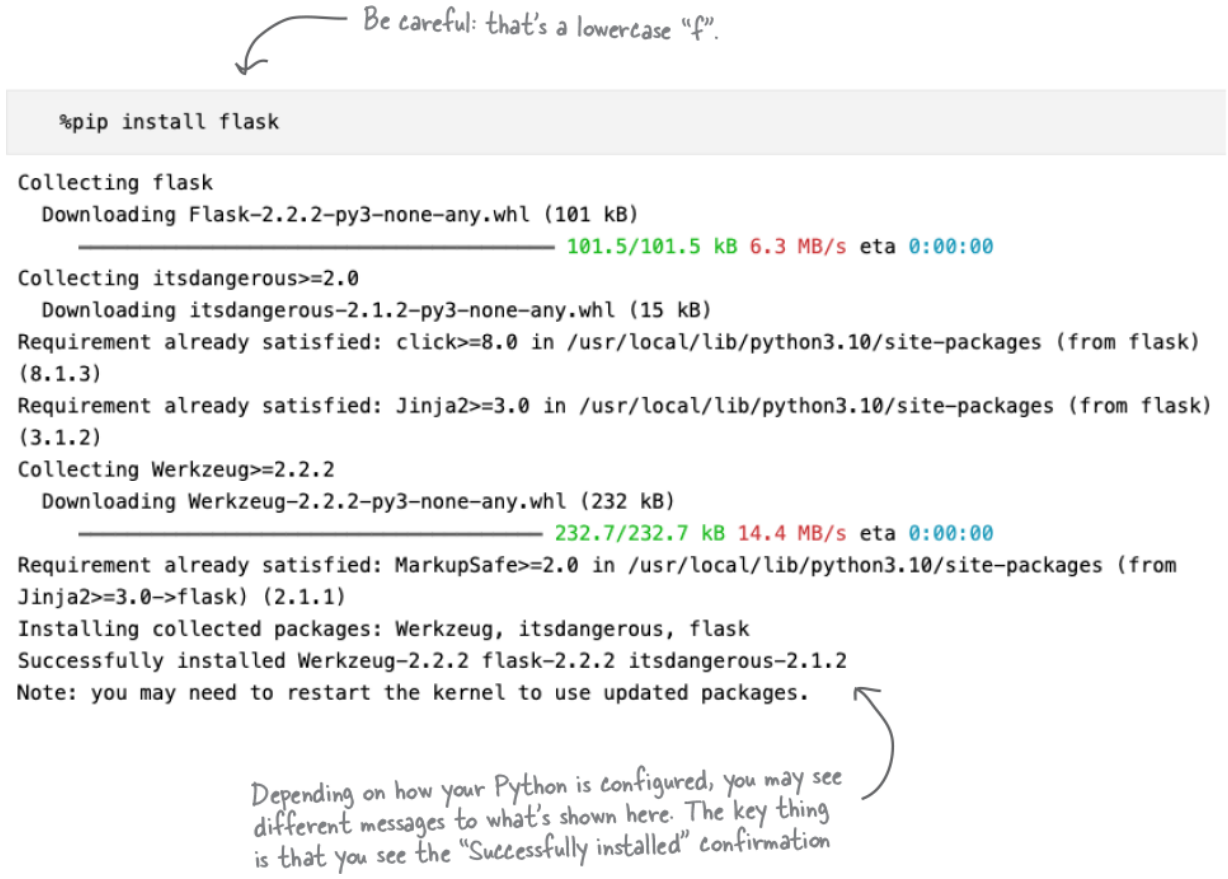


Figure 6-5.

Prepare your folder to host your webapp

Within the `webapp` folder, create two further sub-folders, one called `static`, and another called `templates`. Flask expects these folders to exist for most all webapps.

Next, *copy* the `swimdata` folder from `Learning`, together with the `hfpy_utils.py` and `swimclub.py` modules, into your `webapp` folder, too.

With that done, the contents of your `webapp` folder should mimic what's shown here:

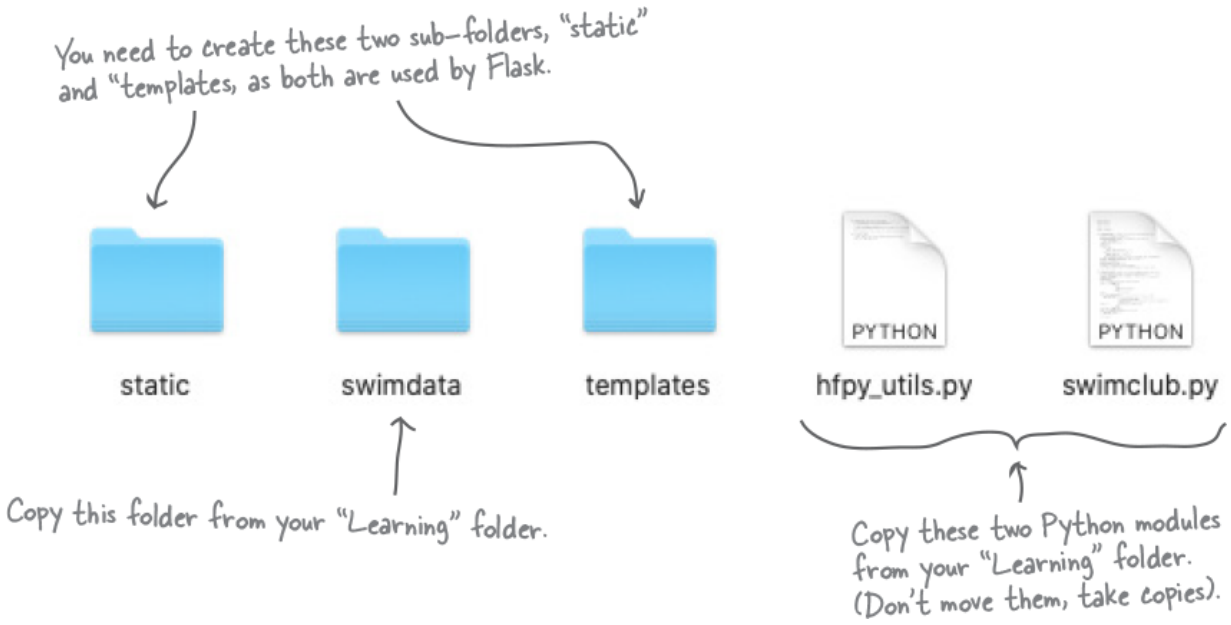


Figure 6-6.

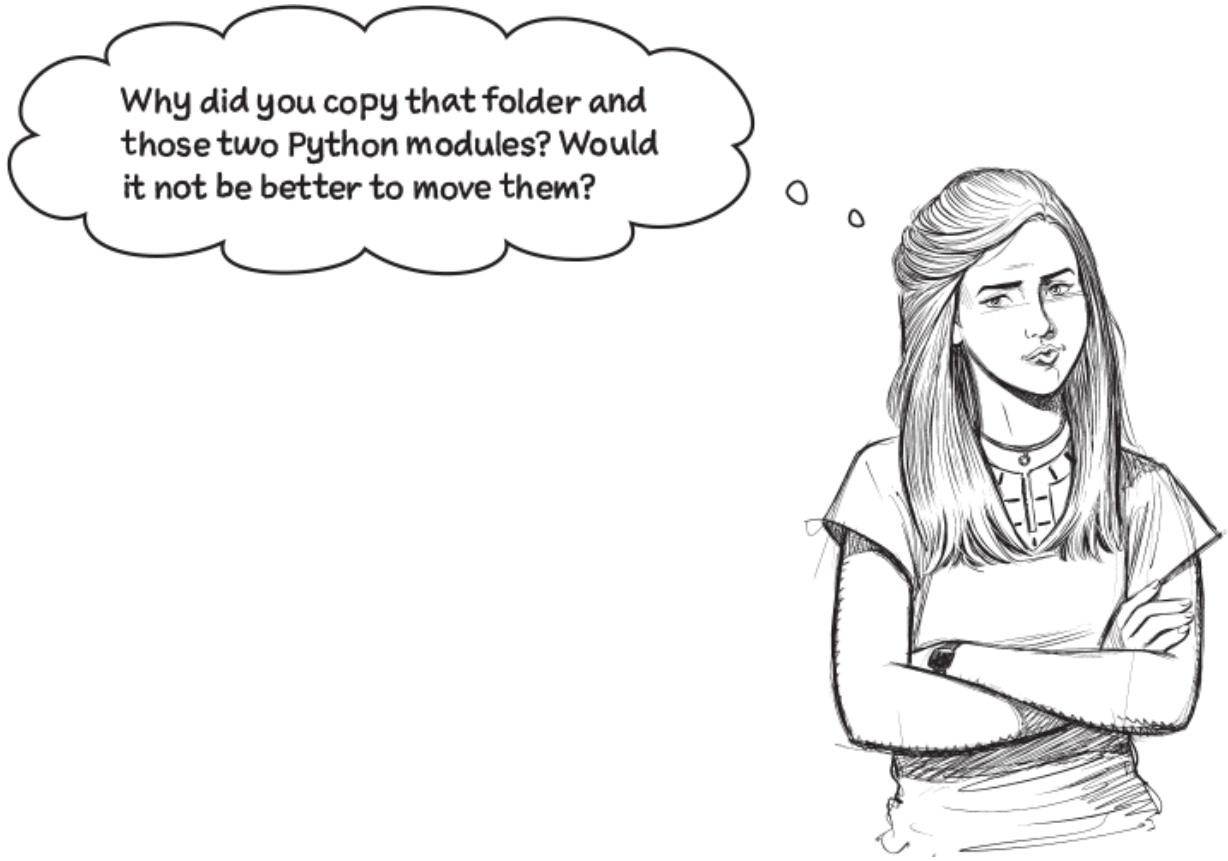


Figure 6-7.

No, copying is the best choice here.

The notebooks in your `Learning` folder expect the Coach's data to reside in `swimdata`, and your notebook's also expect those two modules to exist in `Learning`, too. If you moved those three items, your notebooks would break the next time you run them (with a bunch of file-not-found errors).

Having said that, you might well be getting an icky feeling about copying, as you likely only want one canonical copy of each code file, right? But this is not something to worry about *yet*, as you're still build-

ing the first deliverable for the Coach, and you've yet to decide on what code's needed (and what isn't).

The Flask MVP

When it comes to getting something up-and-running with Flask, many coders are often surprised by how little code is typically needed.

With that in mind, let's look at the code for the Flask *minimum viable product* or, as many like to refer to this example, Flask's *Hello World*:

```
from flask import Flask

app = Flask(__name__)


@app.get("/")
def index():
    return "This is a placeholder for your webapp


if __name__ == "_main_":
    app.run()
```

You'll learn what each of these lines of code do in a little bit. For now, use VS Code to create a new file (*not* a new notebook) called `app.py`, then type the seven lines of code shown above into the VS Code edit window.



Figure 6-8.

That's a good question.

When it comes to test-running your webapp code during development, you could certainly *try* to run a Flask-based webapp within Jupyter. However – if we're totally being honest – doing so makes little sense. You'll have more success test-running your webapps within VS Code, and that's what we'll do here (starting on the next page).

TEST DRIVE



Figure 6-9.

With the code from the last page typed into VS Code's edit window (and saved as `app.py` within your `webapp` folder), let's take your very first Flask webapp for a quick spin.

From the VS Code menu, select **Run** then **Run Without Debugging** to ask VS Code to execute the code currently residing in the edit window. A new terminal opens at the bottom of VS Code, and the following messages appear:

```
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use
production deployment. Use a production WSGI serv
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

These messages are appearing as a result of Flask executing your code.

The most important message (for now) is on the second-last line of the above output, which informs you your webapp is running on port `5000` on your computer with an IP address of `127.0.0.1` (also referred to as `localhost`).

Go ahead and type `http://127.0.0.1:5000` into your browser's location bar. The placeholder message appears on screen, like so:

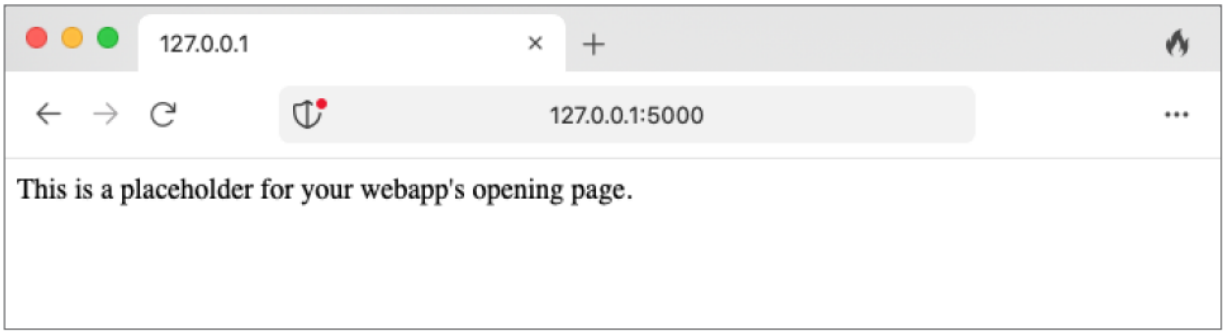


Figure 6-10.

Flask comes with an integrated web server which lets you run and test your webapp as you build it. Although the Flask web server is not designed for "production use", it's more than good enough during development.

Right after you use your browser to view the placeholder message, the terminal area in VS Code updates with a message confirming the

successful processing of a HTTP `GET` request:

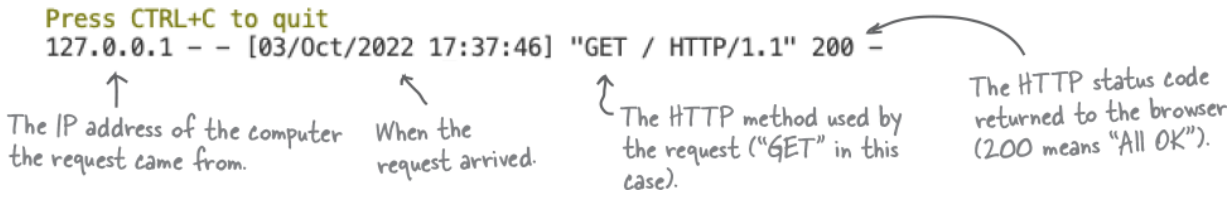


Figure 6-11.



Figure 6-12.

Here, in detail, is what your webapp's code is doing…

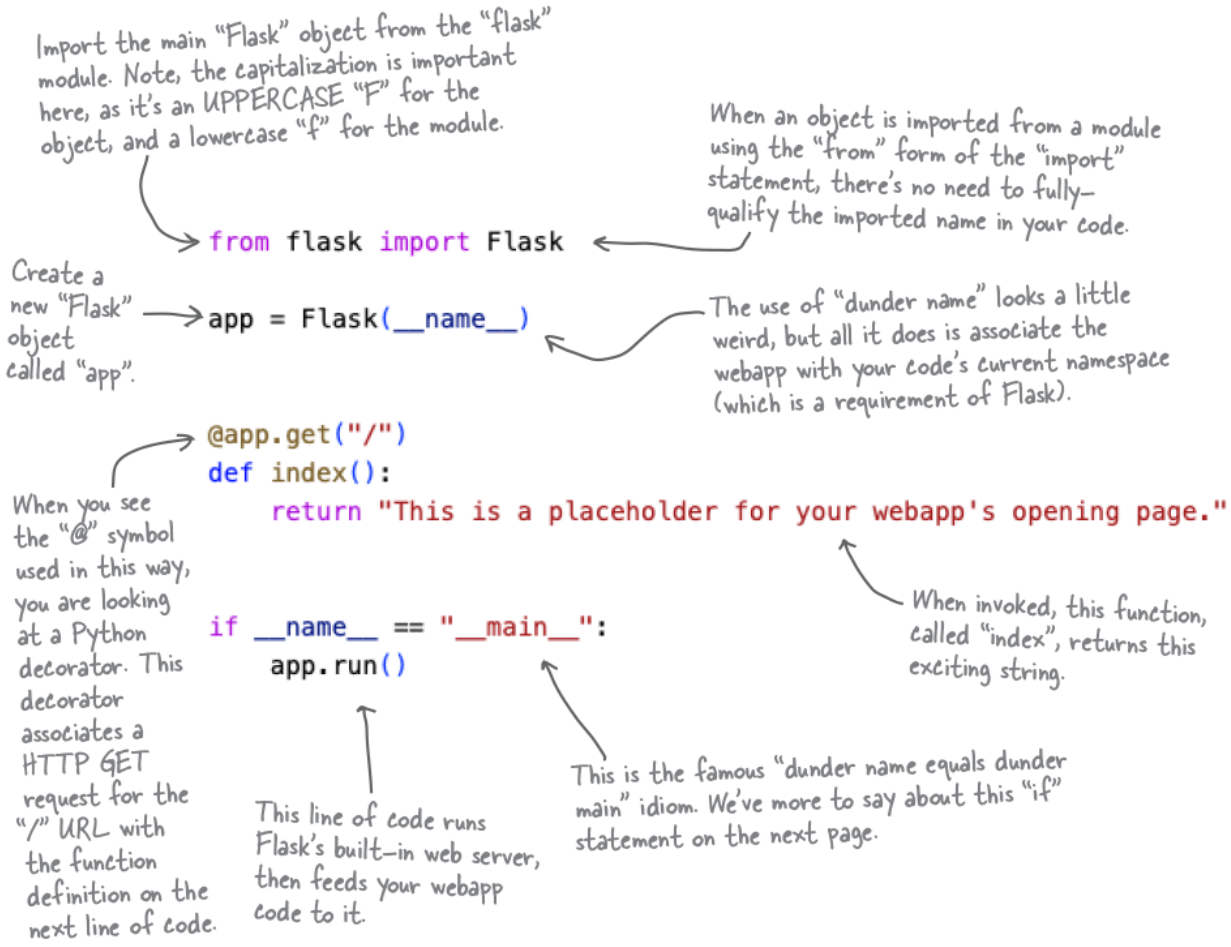

Import the main "Flask" object from the "flask" module. Note, the capitalization is important here, as it's an UPPERCASE "F" for the object, and a lowercase "f" for the module.
When an object is imported from a module using the "from" form of the "import" statement, there's no need to fully-qualify the imported name in your code.
from flask import Flask
Create a new "Flask" object called "app".
app = Flask(__name__)
The use of "dunder name" looks a little weird, but all it does is associate the webapp with your code's current namespace (which is a requirement of Flask).
@app.get("/")
def index():
return "This is a placeholder for your webapp's opening page."
When you see the "@" symbol used in this way, you are looking at a Python decorator. This decorator associates a HTTP GET request for the "/" URL with the function definition on the next line of code.
When invoked, this function, called "index", returns this exciting string.
if __name__ == "__main__":
app.run()
This line of code runs Flask's built-in web server, then feeds your webapp code to it.
This is the famous "dunder name equals dunder main" idiom. We've more to say about this "if" statement on the next page.

Figure 6-13.

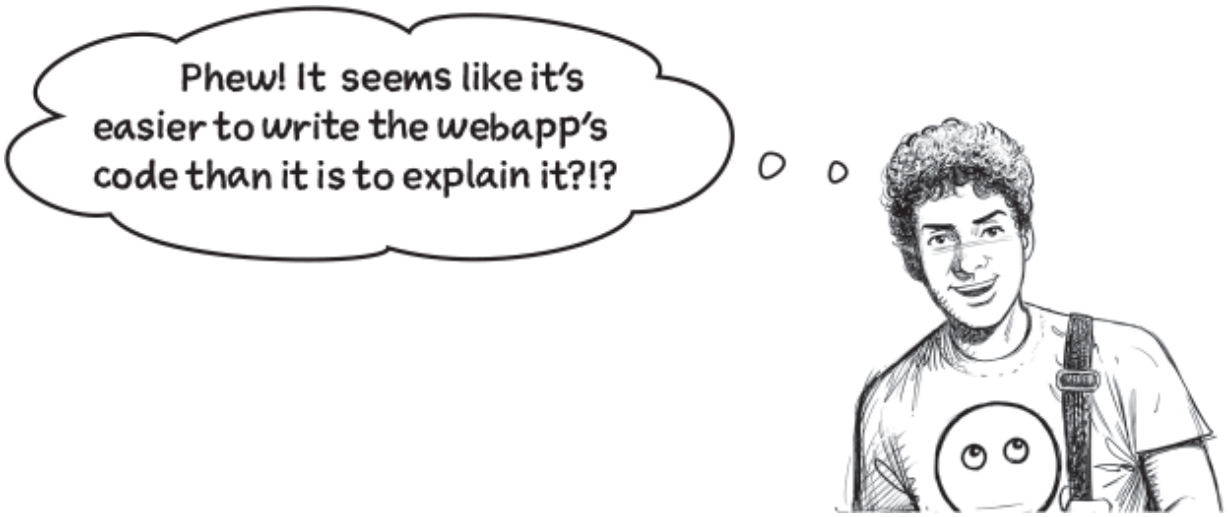


Figure 6-14.

There's certainly a lot less typing!

Those seven lines do pack a punch. The first two lines and the last two lines are common to all Flask webapps, and the bit in between specifies the webapp's distinct behavior.

DUNDER NAME & DUNDER MAIN UP CLOSE



Figure 6-15.

The **if** statement near the bottom of your webapp's code is a bit of a celebrity, as it appears near the bottom of more Python programs than most any other statement. It is an important Python programming idiom, as it lets you control what happens when your code is either (1) *imported* or (2) *executed directly* by the Python interpreter. Let's explain what happens with the help of the world's smallest Python module.

Continuing to work within your `webapp` folder, create a file called `whoami.py` and add the following line of code to it:

```
print(f"Hello, I'm {__name__}.")
```

There's nothing earth-shattering here. When this line runs, the current value of "dunder name" appears as part of this module's friendly message.

Figure 6-16.

Now, choose **Run** then **Run Without Debugging** from the VS Code menu. After a moment, the following message appears in the

Terminal:

```
Hello, I'm __main__.
```

Now, with the code in `whoami.py` saved, return to your `WebappSupport.ipynb` notebook and, in the next empty code cell, type the following (then press **Shift+Enter**):

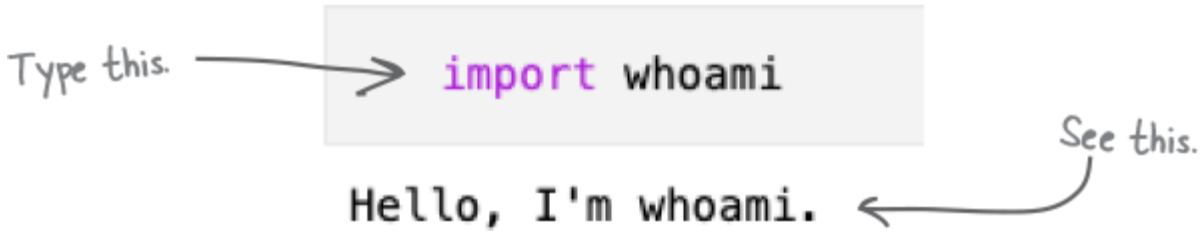


Figure 6-17.

The exact same code executed, but a *different* message is displayed.

When the code is *executed directly* by the Python interpreter, `__name__` is set to the value `__main__` by default (as shown in the first displayed message, above).

However, the second time the code in `whoami.py` is executed, it is *imported* by the Python interpreter, *not* directly executed. When this happens, `__name__` is *always* set to the the name of the module being imported, which is "whoami" in this second displayed message.

Remember: the code in all modules is executed top-to-bottom when it is imported.

Knowing all this, it's now clear what these two lines of code are doing at the bottom of your webapp code:

```
if __name__ == "__main__":
    app.run()
```

When your webapp is *executed directly* by Python, it fires-up Flask's built-in web server and runs your webapp code. However, if your webapp is *imported* as a module, the `app.run()` line does *not* execute. (Just when and why you'd want to import your webapp but not run it will become clear in the next chapter).

SHARPEN YOUR PENCIL

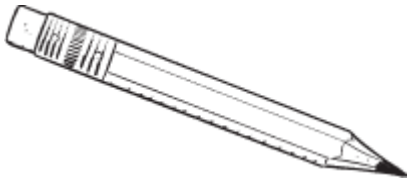


Figure 6-18.

To extend your existing webapp to do more, write more code. Specifically, you need to create additional functions that do what you need your webapp to do.

Let's extend your webapp with another function called `display_swimmers` which, when invoked, returns a sorted list of the unique swimmer names (extracted from the filenames in your `swimdata` folder).

There's a few things to consider. First up, you'll need to add two import statements to the top of `app.py` to ensure the `os` and `swimclub` modules are available to your code. Write in the two import statements you'd use in this space:

Next, every function you add to your webapp needs an `@app` decorator if you want to make it executable via an associated URL. Write

in the decorator line you'd use to associate the `/swimmers` URL with a HTTP GET request:

Now all you need is a function to return the ordered list of swimmer names. We've already decided to call this function `display_swimmers`, so all that's needed now is the function's code. Write the code you'd use for this function here:

Hint: when working on the web, the dominant data format is text. For example, all HTML content is textual in nature. As such, web

browsers typically assume everything sent to them from a web server arrives as text. This can sometimes trip you up if you don't remember to convert your data to it's textual equivalent when delivering it to your web server. To do this in Python, convert your webapp responses with the **str** BIF as needed.

SHARPEN YOUR PENCIL SOLUTION

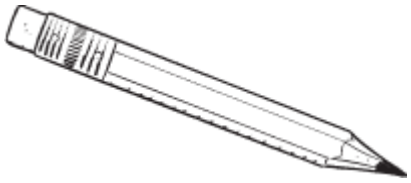


Figure 6-19.

To extend your webapp to do more, you write more code. Specifically, you need to create additional functions that do what you need your webapp to do. You were to extend your webapp with another function called `display_swimmers` which, when invoked, returns a sorted list of the unique swimmer names (extracted from the filenames in your `swimdata` folder).

There's a few things to consider. First up, you needed to add two import statements to the top of `app.py` to ensure the `os` and `swimclub` modules are available to your code. You were to write in the two import statements you used here:



Figure 6-20.

Every function you add to your webapp needs an `@app` decorator if you want to make it executable via an associated URL. You were to write in the decorator line you'd use to associate the `/swimmers` URL with a HTTP GET request:



Figure 6-21.

Now all you need is a function to return the ordered list of swimmer names. With the decision to call this function `display_swimmers` made, all that's needed now is the function's code. You were to write the code you'd use for this function here:

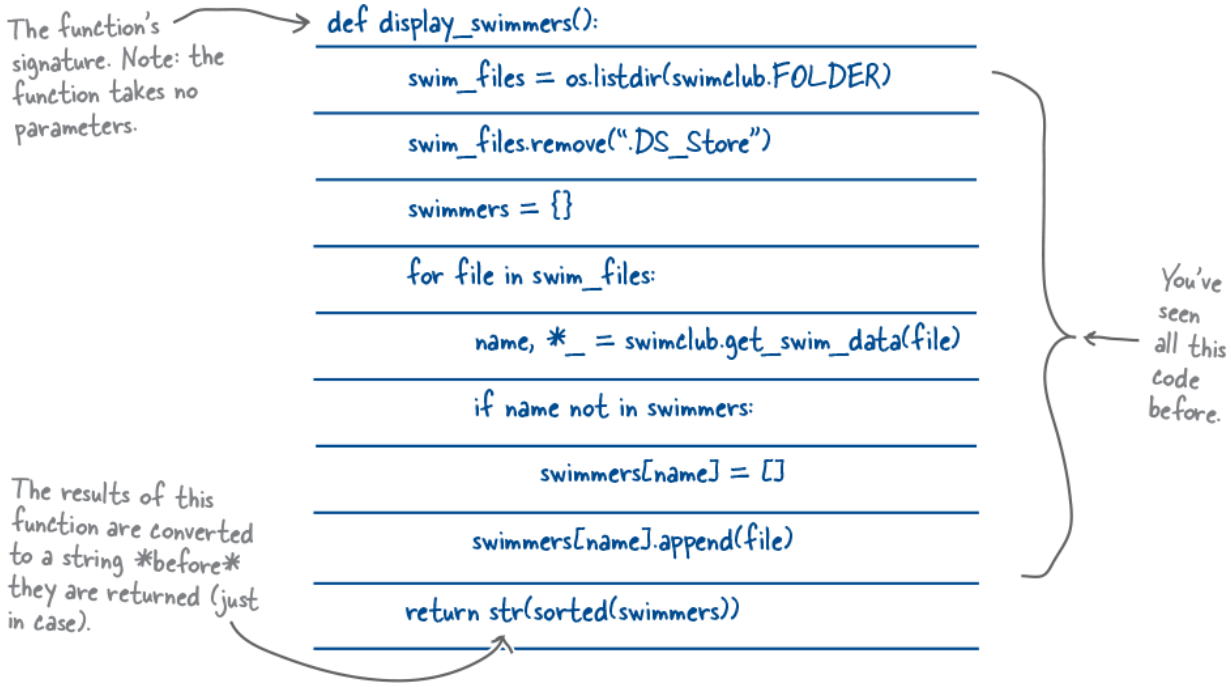

The function's signature. Note: the function takes no parameters.
def display_swimmers():
swim_files = os.listdir(swimclub.FOLDER)
swim_files.remove(".DS_Store")
swimmers = {}
for file in swim_files:
name, *_ = swimclub.get_swim_data(file)
if name not in swimmers:
swimmers[name] = []
swimmers[name].append(file)
return str(sorted(swimmers))
You've seen all this code before.
The results of this function are converted to a string *before* they are returned (just in case).

Figure 6-22.

TEST DRIVE



Figure 6-23.

Go ahead and add your new function to your `app.py` file (above the dunder name **if** statement). Your new code should match what's shown here:

```
@app. get ("/ swimmers")
def display_swirimers():
    swim_files = os.listdir(swimclub.FOLDER)
    swim_files.remove(".DS_Store")
    swimmers ={}
    for file in swim_files:
        name, *_ = swimclub.get_swim_data(file)
        if name not in swimmers:
            swimmers[name] = []
        swimmers[name].append(file)
    return str(sorted (swimmers))
```

Don't type `http://127.0.0.1:5000/swimmers` into your browser's location bar just yet. Instead, within the VS Code Terminal,

press **Ctrl+C** to terminate the previous running instance of your webapp, as that instance is running the older version of your `app.py` code.

Once the termination is confirmed in the Terminal, choose **Run** then **Run Without Debugging** from the VS Code menu to start the *latest* version of your webapp, then visit the swimmers URL. If all is well, you should see the stringed version of your ordered list of swimmer names appear within your browser window, like so:

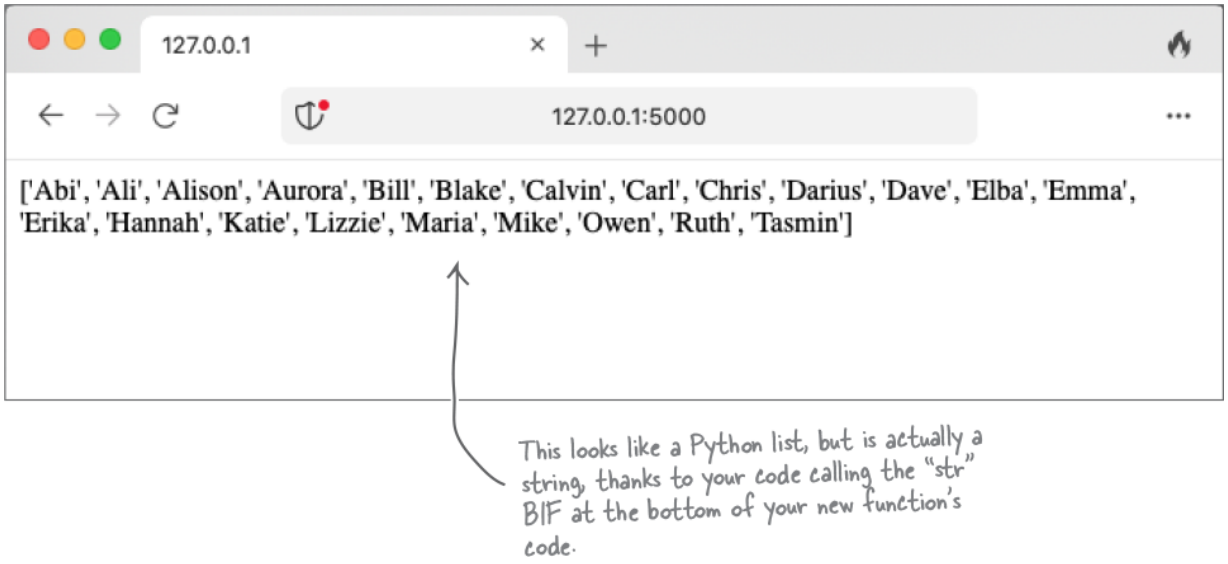


Figure 6-24.

Building your webapp, bit by bit…

Granted, being able to display an ordered list of unique swimmer names in a web browser is of little or no use to the Coach right now. But, the goal here is to build up to a functioning webapp which does do what the Coach needs, so we're deliberately taking our time. You're taking baby-steps, with each modification to your `app.py` code getting you closer to your end goal, which is a functioning we-bapp for the Coach.

With that said, note that it's also possible to parameterize URLs with Flask, and you can use this Flask feature to help you display the list of a selected swimmer's files. Take a look at this small function which allows you to provide an individual swimmer's name when entering the URL into your browser's location bar:

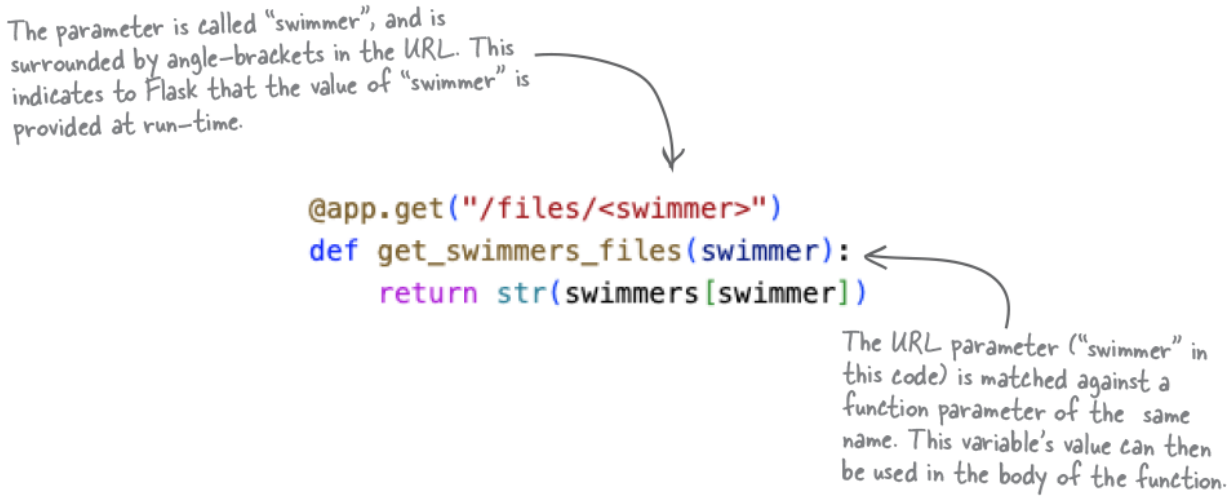


Figure 6-25.

If your webapp is running on port `5000` of your computer, then `http://127.0.0.1:5000` would display your webapp's default

content. However, once the three lines of code shown above are added to your webapp, you can use this URL to request (for example) Ruth's data files:

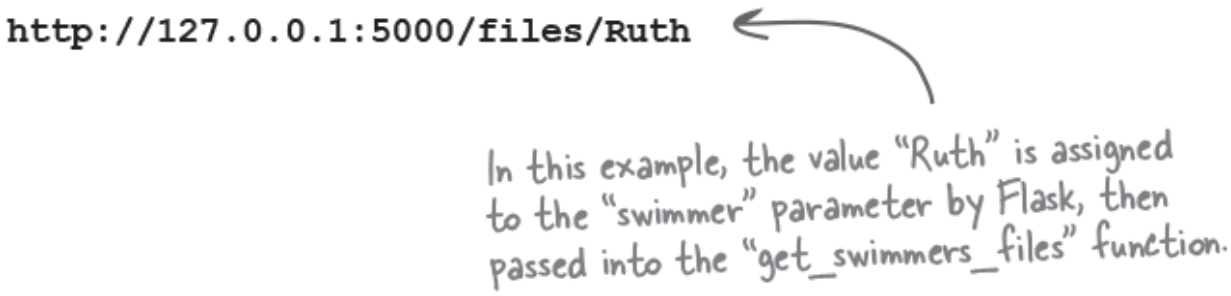


Figure 6-26.

Spoiler Alert!

Sadly, when you add the above code to `app.py`, restart your webapp, then try to use that URL to list Ruth's files, things don't go quite to plan…

TEST DRIVE



Figure 6-27.

If you haven't done so already, add the three lines of code from the last page to your `app.py` (remember to save!), press **Ctrl+C** in the Terminal to stop your previous webapp from executing, the select **Run** then **Run Without Debugging** from the VS Code menu.

Your latest webapp runs…

Pop on over to your web browser and enter `http://127.0.0.1:5000/files/Ruth` into the location bar. Prepare yourself to be disappointed:

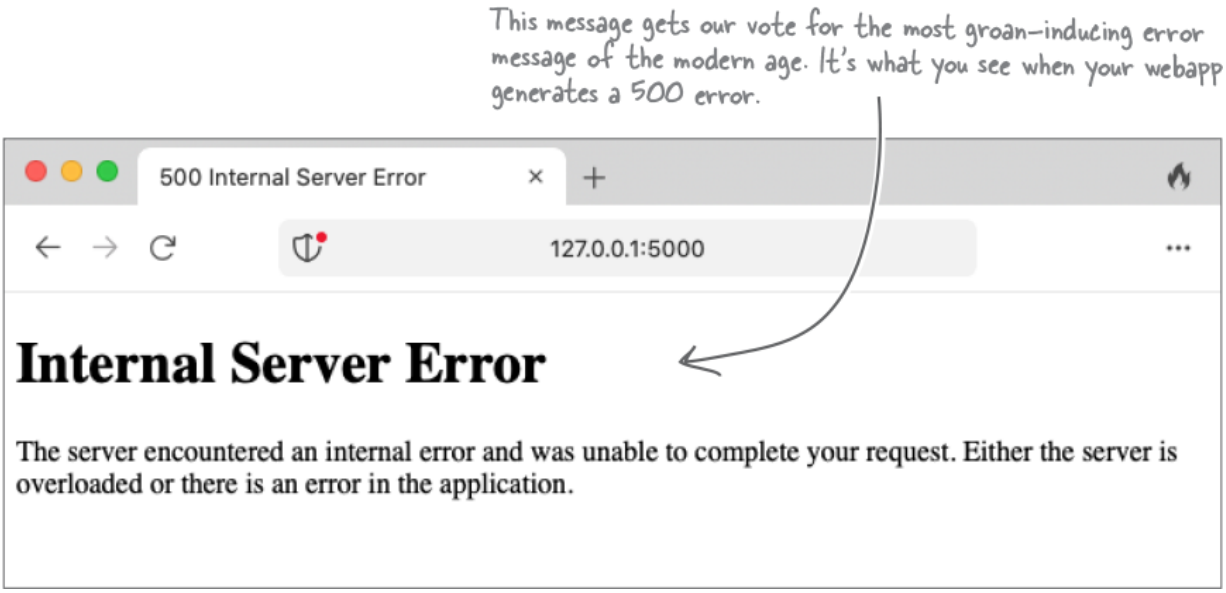


Figure 6-28.

As error messages go, this **Internal Server Error** doesn't really tell you much about the *specifics* of what went wrong. If you take a quick look at VS Code's Terminal, you'll see a plethora of error messages on screen, which offers a bit more information:

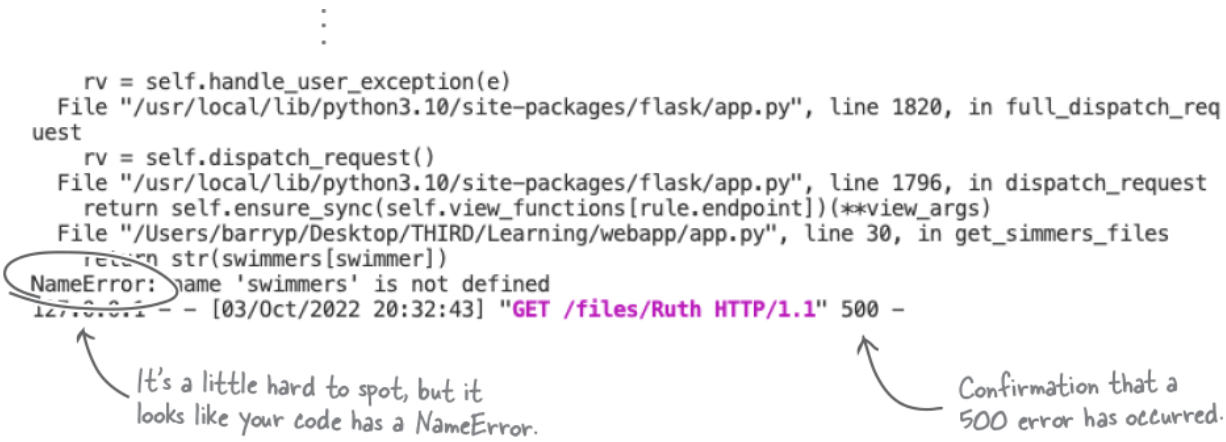


Figure 6-29.



Figure 6-30.

A little extra help is always good.

The bare-bones 500 error message doesn't really tell you much about what specifically went wrong, and having to look at the error messages littering VS Code's Terminal is a little clunky.

Help is at hand. If you adjust the last line of your `app.py` code to look like this:

```
app.run(debug=True)
```

you enable Flask's *debugging mode* which, as you're about to see, makes a big difference.

TEST DRIVE



Figure 6-31.

Press **Ctrl+C** in your Terminal *yet again* to stop your webapp from running. Change the last line in `app.py` to include the `debug=True` parameter to the `app.run()` call, then start your webapp *once more* by selecting **Run**, then **Run Without Debugging** from the VS Code menu.

A few things happen when your webapp runs this time. The first is that Flask's debugging mode is activated, resulting in messages appearing in the Terminal confirming this state of affairs:

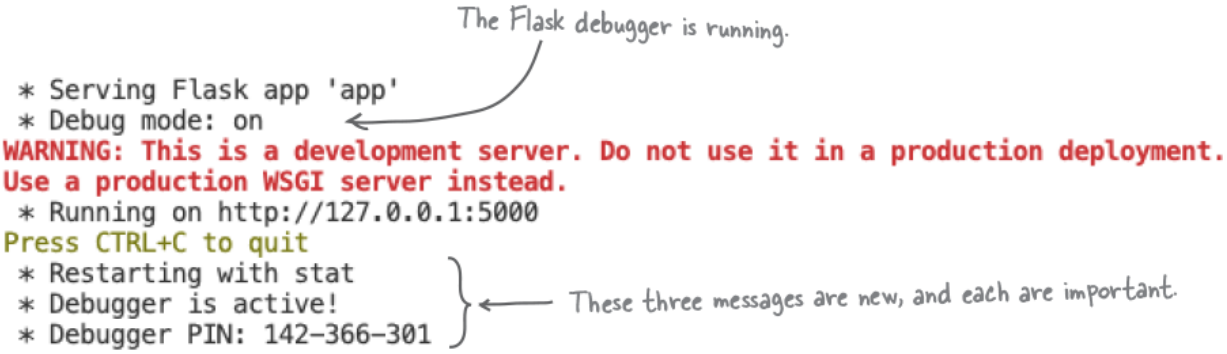


Figure 6-32.

Restarting with stat

The “stat” messages enables a development mode where Flask watches your webapp’s code for any saved edits. When it spots a change has occurred, Flask’s run-time *automatically restarts* your webapp for you. This means you no longer have to manually stop, edit, save, then restart your webapp after each code change: Flask’s debugging mode does all this for you. There’s just one caveat: if your edit introduces a syntax error, Flask will terminate your webapp. To continue, you’ll have to fix the error, save, then restart your webapp manually.

Debugger is active!

The “active” message confirms the Flask debugger is running (but you already knew this from the “on” message on the second line).

Debugger PIN: 142-366-301

The “PIN” message displays a code (142-366-301 in the above example) which can be used to interactively debug your webapp’s code directly within your web browser. It’s hard to describe what this does without showing you an example, so we’re going to defer doing so until you actually need the feature. When you do, you be somewhat amazed by what it lets you to do.

Note that you’ve yet to fix the issue with the current version of your webapp. The *Internal Server Error* has not gone away. However, let’s

see what happens when you pop back to your web browser and re-load the last request…

Rather than the plain, groan-inducing *Internal Server Error* message reappearing, when debug mode is enabled, Flask goes out of its way to be helpful, rendering a developer-friendly error message within your browser, like so:

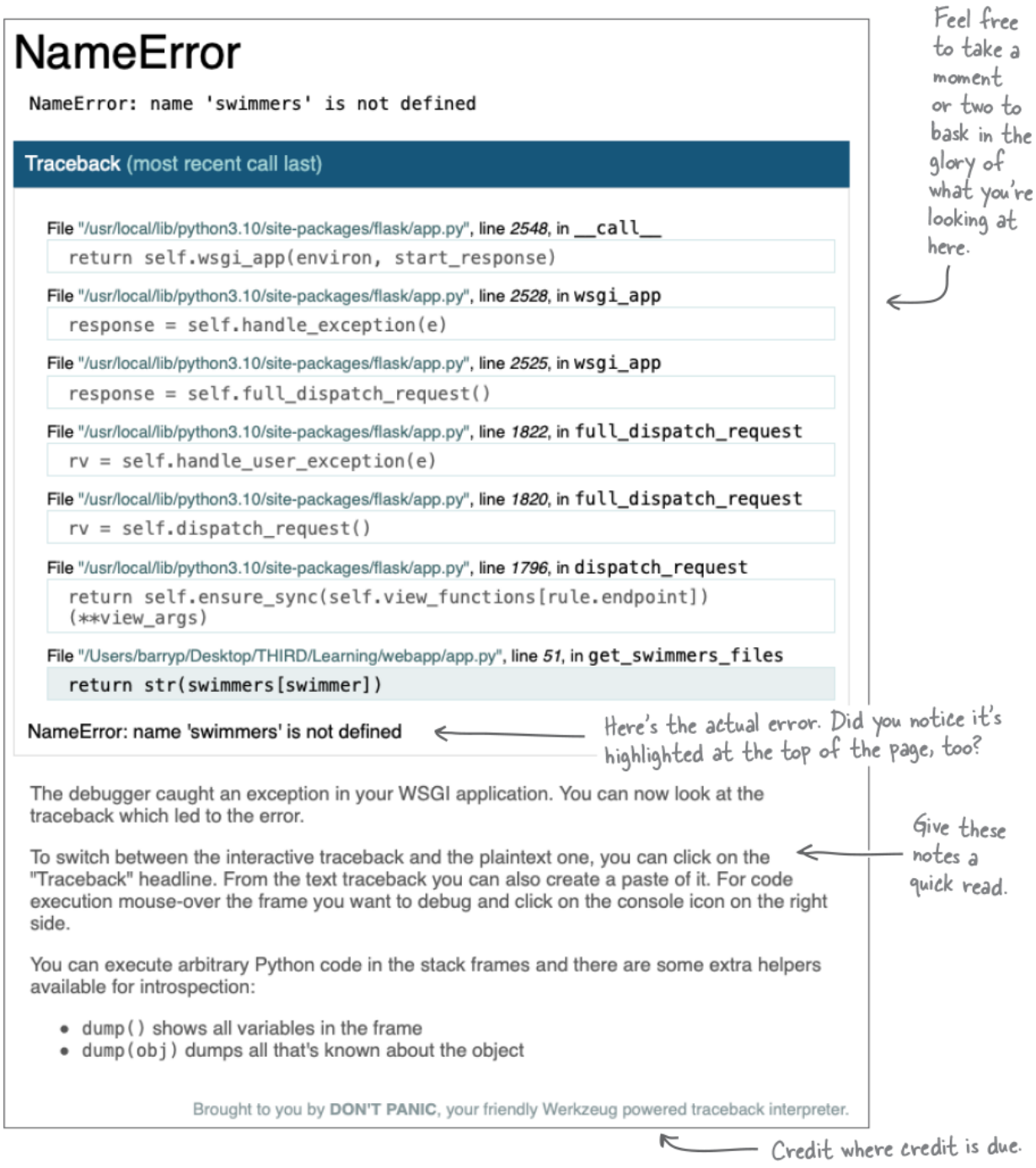


Figure 6-33.

What's the deal with that NameError?

Let's take a look at the code in `app.py`, paying particular attention to the line of code which debug mode tells you is causing the error:



Figure 6-34.

Why do you think the use of `swimmers` in the `get_swimmers_files` function is raising an error?



Figure 6-35.

Cubicle Conversation

Mara: I think we have a visibility issue here.

Sam: I agree. Python's scoping rules are hurting us.

Alex: How so?

Mara: Think about the visibility of the `swimmers` variable. Do you see how it's defined in the `display_swimmers` function?

Alex: Sure…

Sam: That means it's *local* to `display_swimmers`.

Mara: And not visible anywhere else in your code, and certainly not *globally* visually.

Alex: So why not make it a global variable so all of the webapp's code can see it?

Mara: If only that worked all the time…

Alex: Eh?

Sam: That's a subtle one. Using a global variable in a webapp works when the webapp serves a single user, for example: when being tested by *you* on *your* computer. But, pop the same webapp on the cloud and it can lead to *carnage*…

Mara: Ha! Carnage, that's funny. Sam's maybe being a little *OTT* there, but – yes – global variables in webapps are best avoided.

Alex: I'm none the wiser as to why, folks.

Sam: The problem has to do with the fact that your webapp has *one* instance of your code running in the cloud. If you share data in a global variable in that code, everyone interacting with your webapp sees the *same* data. This is OK is no one is changing the value, but think what happens to the `swimmer` variable.

Mara: It's value is *changed* each time a user of the webapp *selects* a swimmer to work with.

Alex: Which might *overwrite* the value *another* user just selected?

Mara: Precisely!

Sam: As I said, *carnage*…

Alex: OK, now I get it now. I guess we need is a safe way to share the value of variables within our webapp code?

Sam: Yes, and I think Flask can help us here…

Flask includes built-in session support

If you've done any previous web development, you'll likely know that using a global variable to share data on the web server is fraught with danger, especially when multiple users preform a web request which updates the global data. It can sometimes work fine, but – honestly – you need a mechanism here that works *all the time*.

As you can imagine, you're not the first programmer to bump up against this need, and the good folks who built Flask provide what's

known as *session support*, which you can easily enable for your webapp. Using a technology on the web server which piggybacks on top of the web browser-based cookie technology, you can safely share data within your webapp by putting the shared data into a *dictionary* called `session`.

NOTE

We did mention at the end of the last chapter that dictionaries are *everywhere* in Python!

To enable sessions, you need to do two things: (1) import the `session` object from Flask, then (2) set a *secret key*. This results in two edits to your `app.py` code:

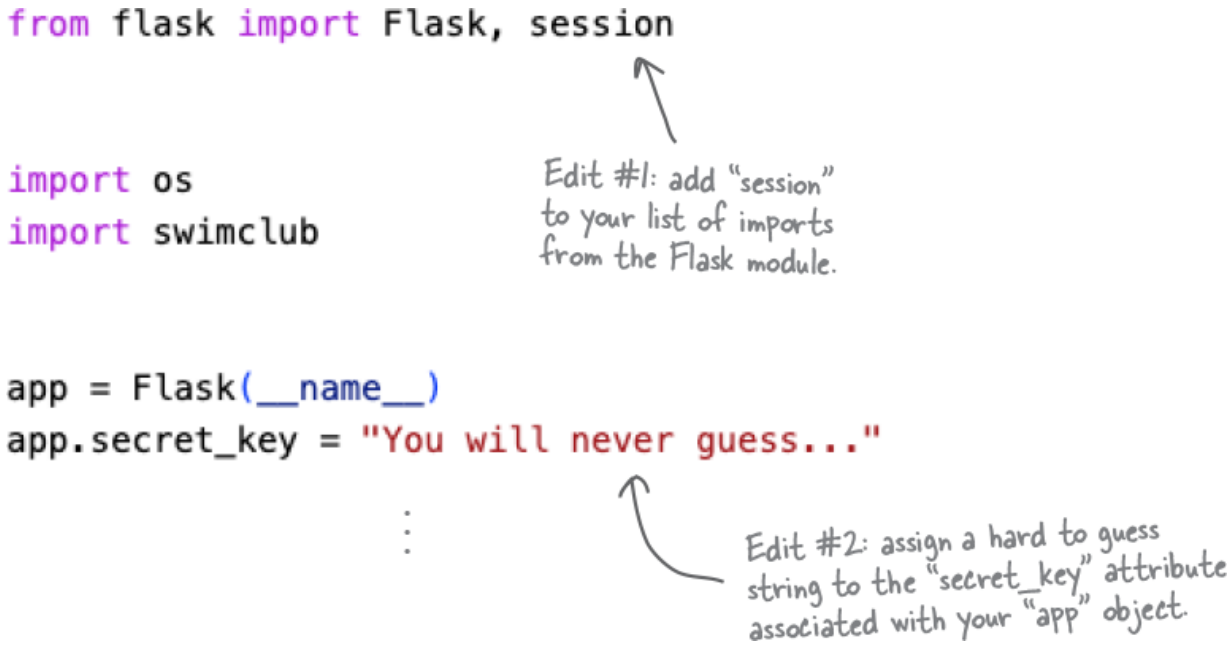


Figure 6-36.

The secret key used in the above code should be changed to something much harder to guess. This string is then used by Flask to encrypt any cookies sent to your web browser to ensure nothing can be messed with on your client.

The only time you'll want to use a secret key as shown above may be when you want to annoy your teenage kids. If they've been bold, change the WiFi password in your house to "You will never guess". Then, when the little darlings ask you what the new password is, tell them. Guaranteed, they'll storm off in a huff, before they eventually (some hours later) realize you've been answering their question all along.

Your secret key needs to be hard to guess.

Not that we know *anyone* who has actually done this to their kids, of course…

Flask's session technology is a dictionary

Here's a really *quick-fix* to the current problem (how to safely share a variable's value between functions), which like a lot of quick-fixes, works… but, depending on how it's used, might break.

The `session` technology included with Flask can be thought of as a *server-side dictionary*. Anything stored in the `session` dictionary can be shared among all functions in your webapp.

Knowing this (and here's the quick-fix), replace all the occurrences of the `swimmers` variable with `session["swimmers"]` in your `app.py` code. The code snippet below indicates where the six edits occur:



Figure 6-37.

Go ahead and make those quick-fix edits to your `app.py` code, remembering to save your code to ensure Flask automatically reloads your webapp. When you're ready, follow along with the *Test Drive* on the next page.

TEST DRIVE



Figure 6-38.

With the latest version of your webapp code running, visit `http://127.0.0.1:5000/swimmers` to confirm that the previous functionality still works:

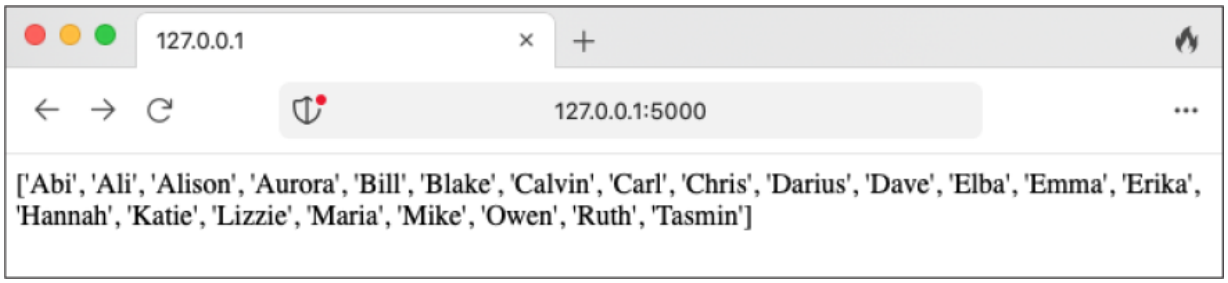


Figure 6-39.

Now, visit `http://127.0.0.1/files/Calvin` to view the filenames associated with Calvin. Recall that a request like this reported an error previously. This time, however, it works!

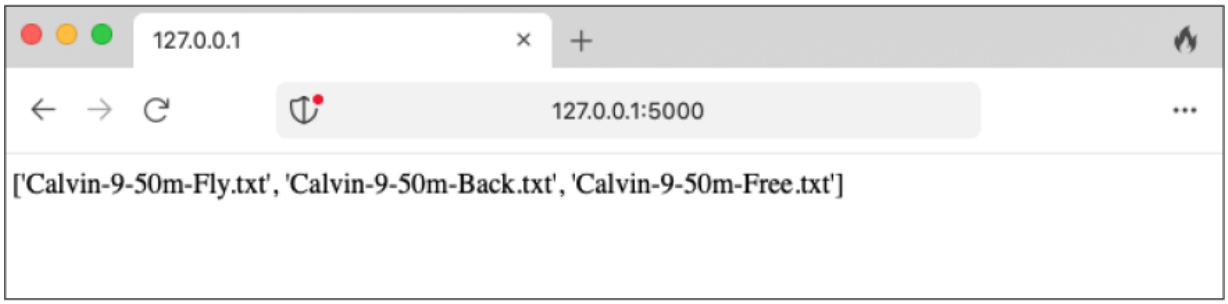


Figure 6-40.

That all seems in order, as both URLs and their associated functions are working without errors.

Do the following: quit your web browser, then stop (with **Ctrl+C**) your webapp within VS Code, then quit VS Code. Pause for a moment… stand up, stretch, and look out the nearest window (but only for a moment). Return to your computer and re-open VS Code, then restart your webapp by choosing **Run** then **Run Without Debugging** from the VS Code menu. Start your web browser and enter `http://127.0.0.1/files/Calvin` into the location bar, then press the **Enter** key. Disaster strikes!

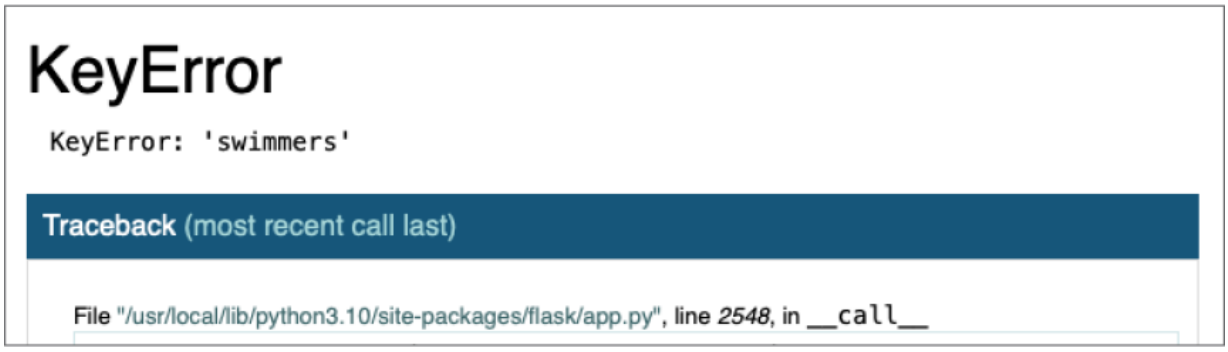

KeyError
KeyError: 'swimmers'
Traceback (most recent call last)
File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 2548, in __call__

Figure 6-41.



Figure 6-42.

The execution order is the issue.

The code is fine, however, the ordering of execution is making a difference here. Namely, if the `get_swimmers_files` function is executed *before* the `display_swimmers` function, the `session` isn't populated with the `swimmers` key. Whoops!

Fixing your quick-fix

Flask's `session` technology lets you safely use your shared data in more than one scope. At the moment, only one of your functions populates `session`, which is *good*. What's *bad* is that there's a sequence of events which result in the `session` variable containing nothing which leads to *carnage* (to borrow a term used earlier). A fix to your quick fix can ensure `session` is populated with the correct data *at all times*, so that when you need to access the shared data, *it's actually there*.

Let's extract the `session` populating code from the `display_swimmers` function so it can be called from any other place. Additionally, let's ensure for any one execution of your webapp, that `session` is only ever populated once. As doing so is a straightforward activity, we've gone ahead and made these changes for you. You'll find the updated code on the next page.

Adjusting your code with the "better fix"

Here's the updates we made to the `display_swimmers` and `get_swimmers_files` functions. Additionally, the code to a new function called `populate_data` is shown:

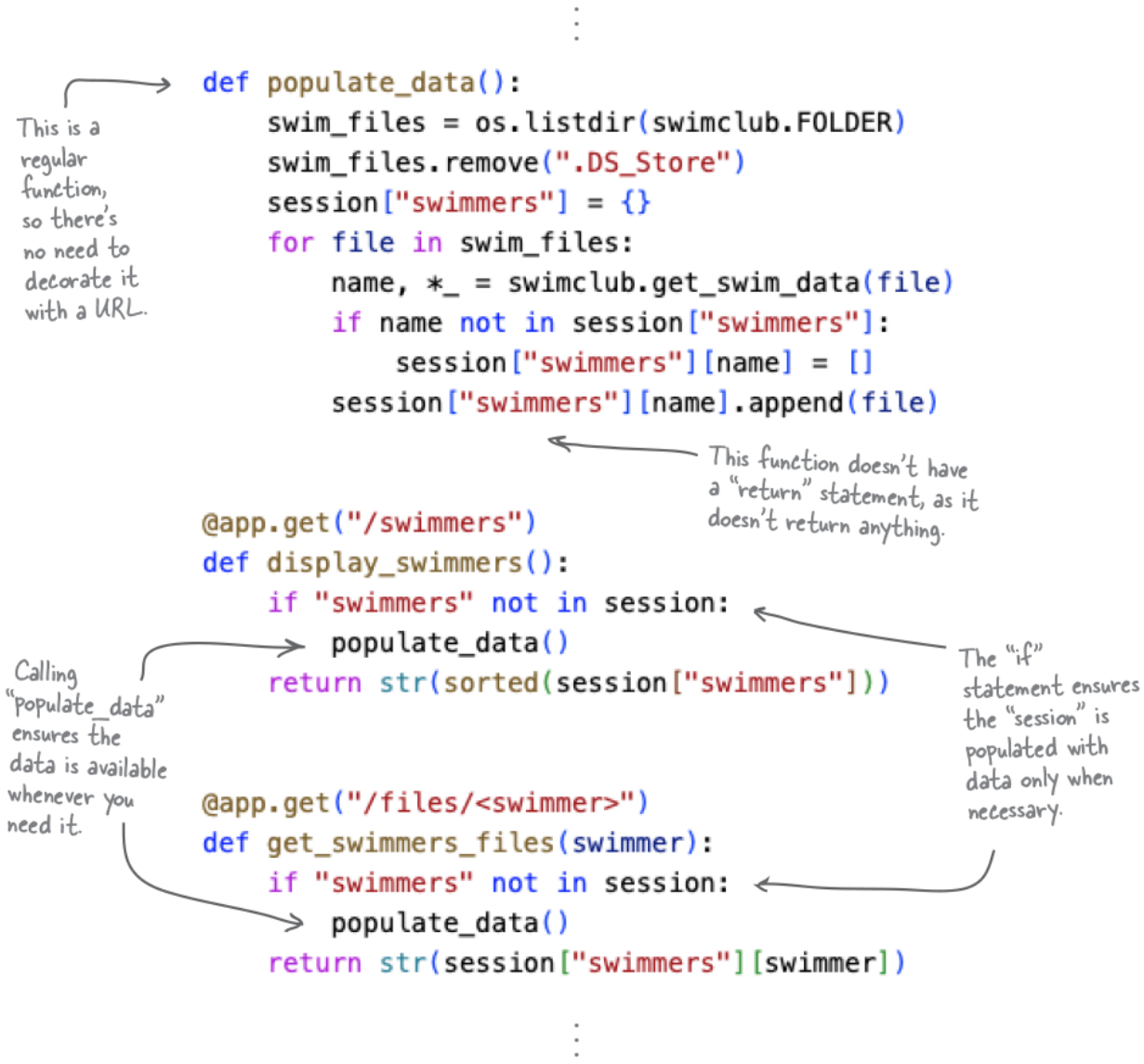


Figure 6-43.

Be sure to make the above changes to your code before flipping over to the next page.



Figure 6-44.

It's time for some more HTML.

Every webapp is built with HTML pages, and what you'll want to do here is dynamically create two drop-down lists using HTML's

`<select>` tag.

The first drop-down list will alllow the Coach to select a swimmer's name, then the second drop-down list will allow the Coach to select an event from the list of events the selected swimmer swam. Once the event is selected, the system can draw the associated bar chart.

Although this sounds like a bit of work, Flask together with one of its built-in technologies – **Jinja**2 – makes this very straightforward.

WATCH IT!



Figure 6-45.

Some prior knowledge is being taken for granted here.

As this book's goal is to introduce you to the wonders of Python, we assume you are comfortable with reading and writing HTML. We also assume you have a basic understanding of the difference between HTTP's GET and POST methods. If our assumptions are incorrect, you may wish to source some introductory material on HTML and/or HTTP before continuing (although we'll do our best to describe what we're up to as we proceed).

Use render_template to display web pages

Flask comes with a function called `render_template` which, in it most basic form, provides a mechanism to display a file of HTML in your web browser as served up from your webapp. The only requirement is that your actual HTML files are stored in your webapp's `templates` folder (which you created at the start of this chapter).

Here's the code from your webapp which provides a default entry-point (or page):

```
@app.get("/")
def index():
    return "This is a placeholder for your webapp's opening page."
```

Assuming the existence of a HTML file called `templates/index.html`, you can change the `return` statement to use Flask's `render_template` function, like so:

```
@app.get("/")
def index():
    return render_template("index.html")
```

There's no need to include "templates/" here as Flask will look in the "templates" folder by default.

Figure 6-46.

For this updated code to work you'll need to add `render_template` to the list of imports for Flask (on the first line of `app.py`) *and* you'll need to create your own `index.html` page (and save it in your webapp's `templates` folder).

NOTE

Be sure to do this.

TEST DRIVE



Figure 6-47.

Go ahead and apply the edits discussed on the last page to your `app.py` code, then create a new file in your `templates` folder called `index.html`, with the following HTML as the file's content:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to the Swinclub system</title>
    </head>

    <body>
        <h2>Welcome to the Swimclub system</h2>
        <p>
            This is the Coach's swimclub system.
            Begin by selecting a <a href="/swimmers">swimmer</a> to work with.
        </p>
    </body>
</html>
```

Save this HTML into your "index.html" file inside your "templates" folder.

Figure 6-48.

With your `app.py` code changed (as discussed on the last page), reload your webapp's default URL in your web browser. Your new HTML-rendered output appears:

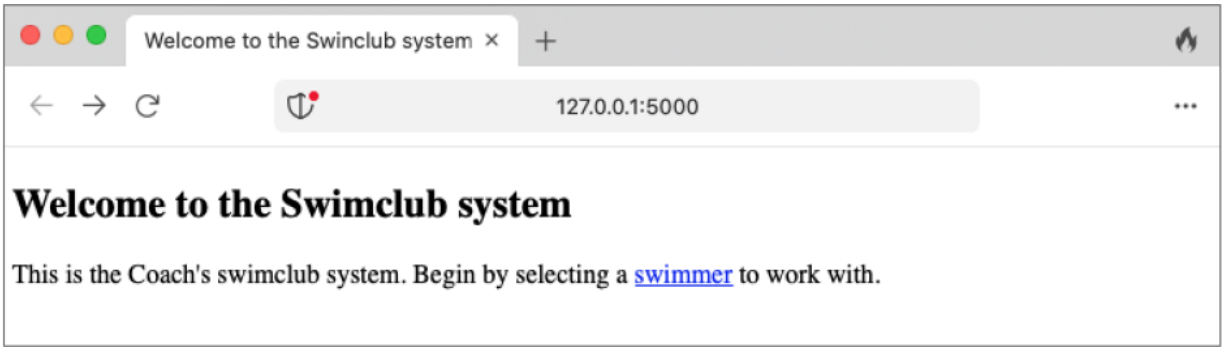


Figure 6-49.

That blue link is just *begging* to be clicked. When you do, the list of the Coach's swimmers appears:

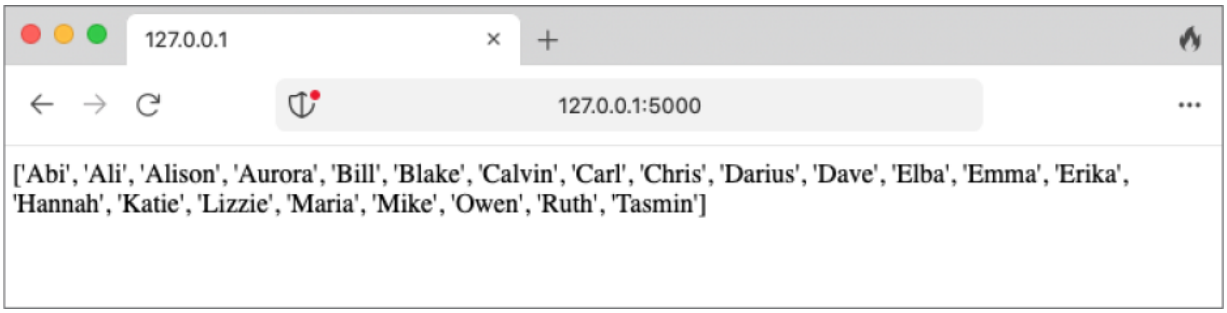


Figure 6-50.



Figure 6-51.

We agree, it does!

The goal here is not to produce a system that's stunningly beautiful. What we want is a system that works. So, yes, it's basic. And, yes, it looks that something from the 1990s.

And that's OK.

That list of swimmers needs to be a drop-down list

Clicking on your “swimmer” link on the home page sends the `/swimmers` URL to your waiting Flask web server, which runs the `display_swimmers` function resulting in your browser displaying the “raw” list of swimmer names.



Figure 6-52.

Rather than seeing that raw data, what you really want here is a drop-down list of swimmer names to appear, rendered on your web browser's page, using the HTML `<select>` tag.

There are a number of ways you could do this, but you definitely don't want to have to write a hard-coded HTML page with all the required

data added by you by hand (just think of the wear'n'tear on your poor fingers).

Instead, it would be nice if you could apply some of your programming chops to the problem, only writing the absolute minimum amount of HTML as needed, while generating the rest with a small amount of Python code…

If only such an arrangement were possible?



Figure 6-53.

That's certainly worth considering.

Using f-strings with your SVG code was a great fit, but you'll probably want something that operates at a higher level when generating a collection of HTML pages.

Flask's bundled *templating engine*, Jinja2, is packed full of HTML-generating goodies.

Jinja2 processes templates which are a mixture of HTML and Jinja2's markup extensions, which are combined to produce dynamically-generated web pages. Jinja2's files are (typically) standard HTML files with some Jinja2 markup added.

And – yes – you guessed correctly: Jinja2 using the word "templates" explains why Flask expects all your HTML files to reside in your `templates` folder, because that's where Jinja2 expects them to reside, too.

THERE ARE NO DUMB QUESTIONS

Q: Just to be clear, all of the HTML files used by my Flask webapp needs to be in the `templates` folder?

A: Yes, and you need to be careful with the spelling used, in "templates" has to be in *all lowercase*. Some operating system play fast-and-loose with case and, under certain circumstances, you can use "Templates" or "TEMPLATES", and your OS won't mind. But Flask and Jinja2 will, so be sure to create the `templates` folder in all lowercase (all you'll be fine).

Building Jinja2 templates saves you time

Let's transform your existing `index.html` into a new template called `base.html`.

The `base.html` templage contains the HTML you'll want to share with all your subsequent web pages, in addition to Jinja2 markup that indicates content which is to be added to the template at some later time.

Take a look at the transformation:

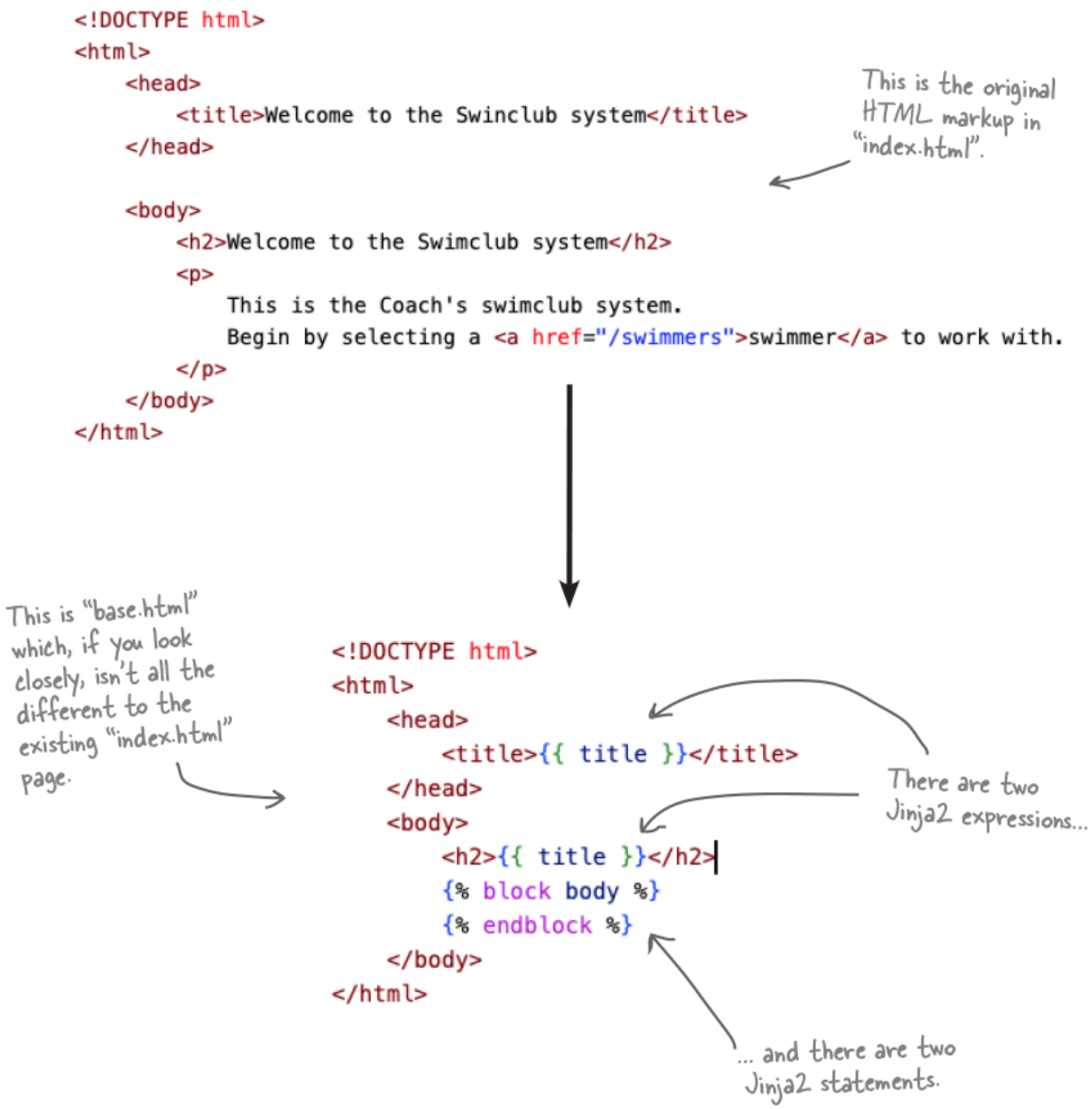


Figure 6-54.

Let's get to know a bit about Jinja2's markup extensions to HTML

BEHIND THE SCENES



Figure 6-55.

Like most mature template technologies, there's a lot included with Jinja2. That said, it's possible to be productive when all you have is a working ability in a small amount of Jinja2.

Here's the first thing you need to know: Jinja2 allows you to apply the notion of inheritance (from object oriented programming) to your HTML pages. You create a "base" template containing all the HTML markup that you want to share (typically called `base.html`), then inherit from this base template when creating new HTML pages. There's a Jinja2 *statement* which lets you do this, and here's what it looks like:

The {% and %} delimiters surround Jinja2 statements.

```
{% extends "base.html" %}
```

When Jinja2 see this markup, it inherits all the HTML from the "base.html" file.

Figure 6-56.

As well as the `extends` statement, Jinja2 also supports (among others), `if`, `for`, and `block` statements. You'll see some of these used when you start to build your webapp's templates.

In addition to Jinja2 *statements*, there's also Jinja2 *expressions*, which are delimited within a template by *double* curly braces, like so:

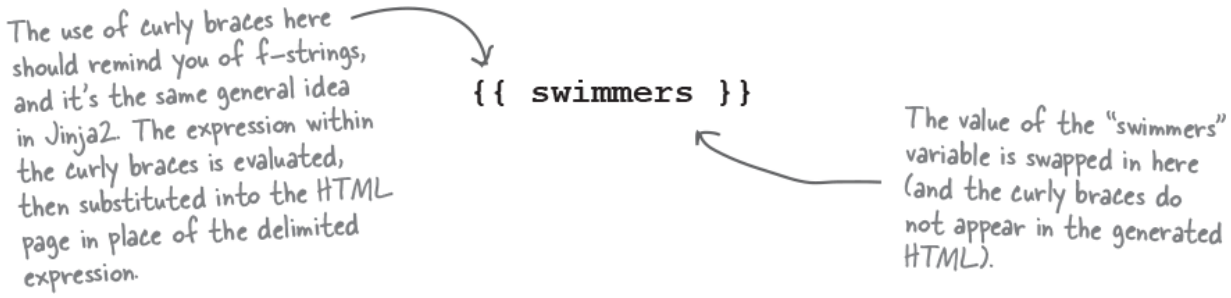


Figure 6-57.

Most any valid Python expression can appear between the double curly braces and, again, the expression is evaluated *before* the substitution occurs. For example:

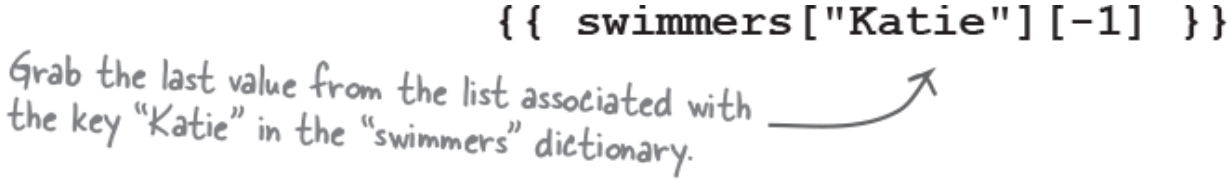


Figure 6-58.

Extend base.html to create more pages

Now that `base.html` exists, you can edit `index.html` to inherit from the base template, then provide some of the "missing" content:

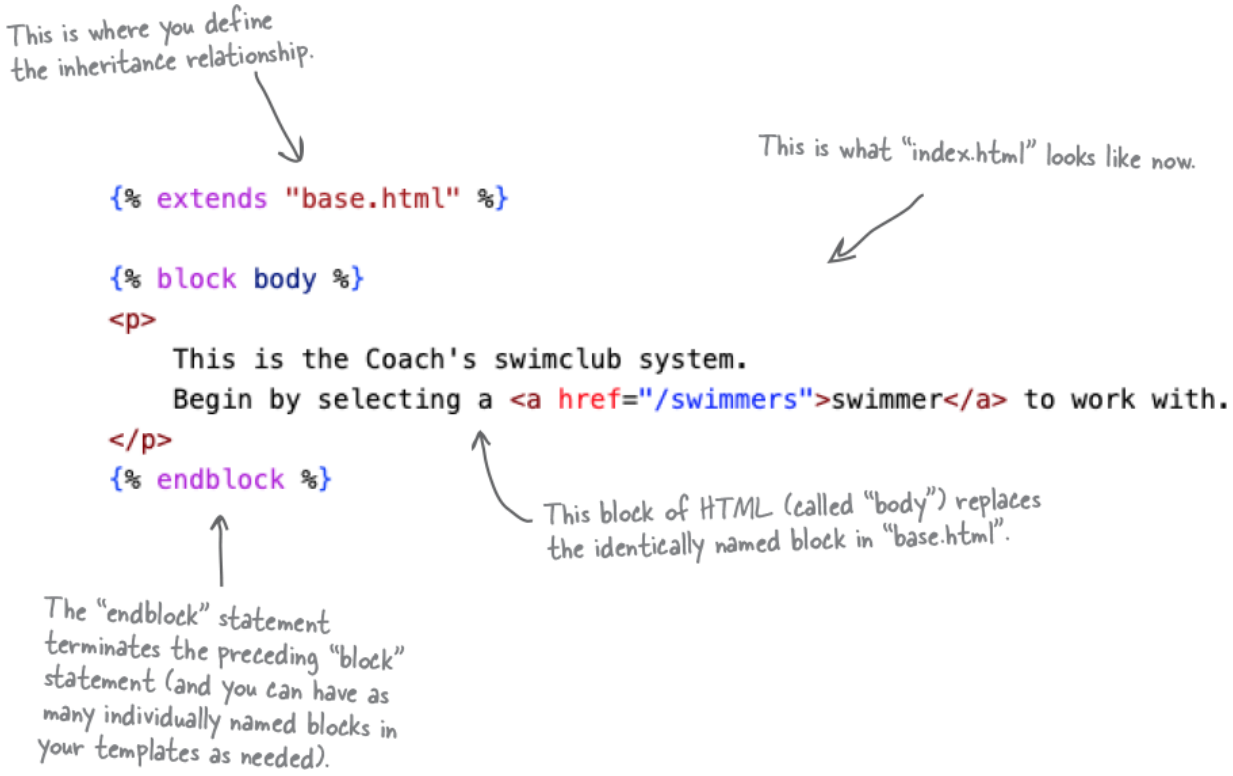


Figure 6-59.



Figure 6-60.

The expression values come from your code.

In this example, the value to be used by the `title` expression is passed in from your Python code when you call Flask's `render_template` function.

Let's see how to do this overleaf.

EXERCISE



Figure 6-61.

Here's what we want you to do.

If you haven't already created `base.html` and edited `index.html`, please jump back and create/edit those HTML files *now*. Be sure to save both `base.html` and `index.html` into your `templates` folder.

With the templates ready (and in place), edit the `index` function in your `app.py` code to look like this, then save your code:



Figure 6-62.

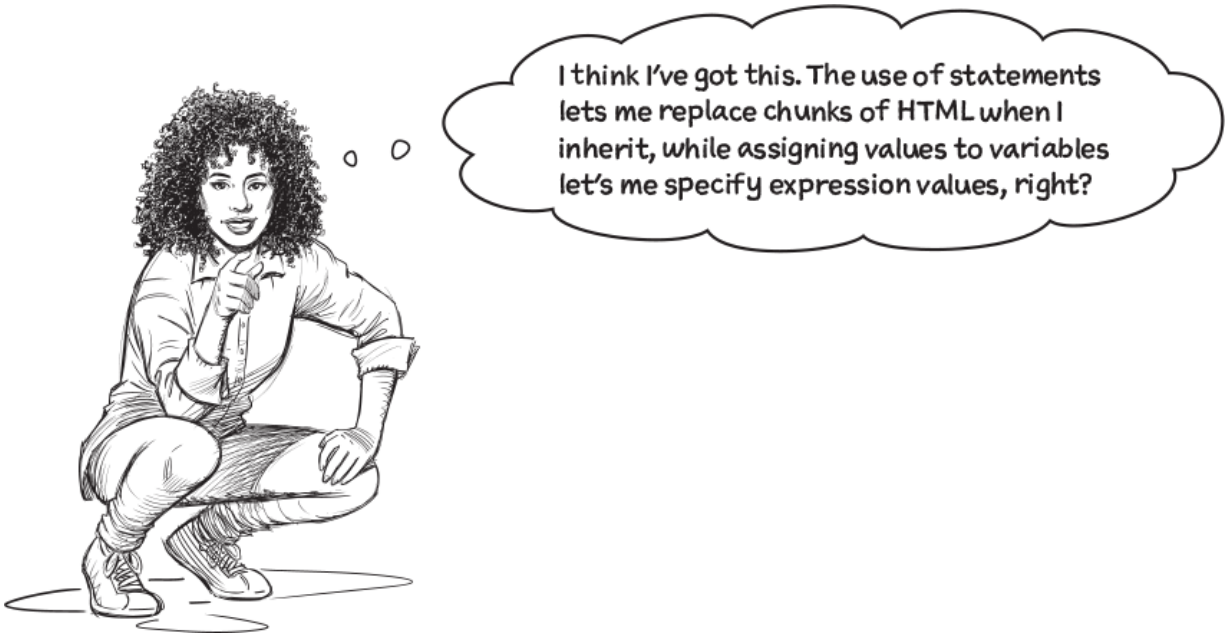


Figure 6-63.

Yes, that's it exactly.

Let’s see all of this in action…

TEST DRIVE



Figure 6-64.

With your templates ready, your `app.py` code edited (and saved), you're ready to take your Jinja2-enabled webapp for a spin. Double-check that Flask has successfully restarted your webapp (check in VS Code's Terminal), then reload your default page in your web browser:

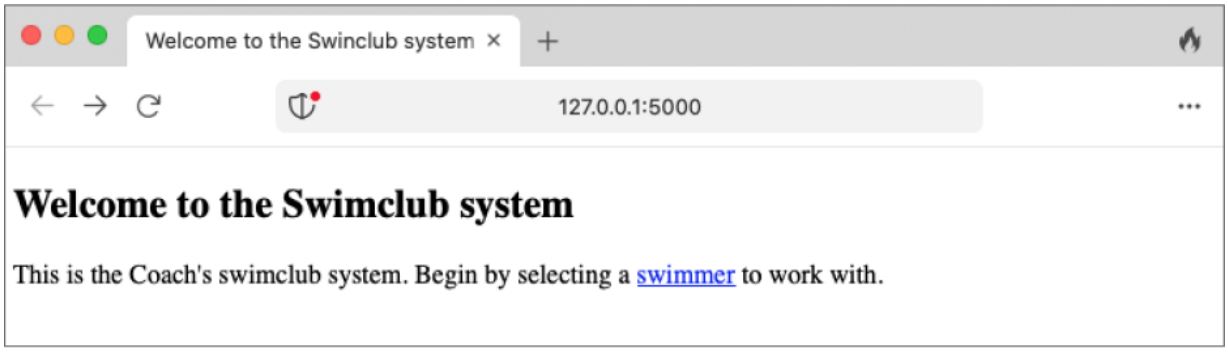


Figure 6-65.

Clicking on the blue link does exactly what you expect it to: the list of the Coach's swimmers appears on screen:

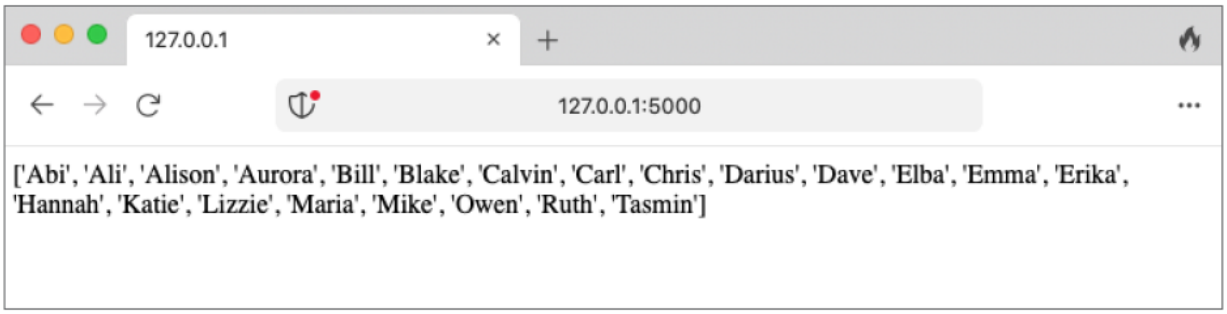


Figure 6-66.



Figure 6-67.

And so it should be!

You should not be expecting to see anything new here. However, your webapp is now using Jinja2 templates which makes adding new pages to your webapp a breeze.

To illustrate this new web dev *superpower*, let's look at Jinja2 template code which generates a `<select>` drop-down list.

Dynamically creating a drop-down list

Here's a Jinja2 template (called `select.html`) which dynamically renders a HTML `<select>` tag of any size when provided with a list of data values (in a variable called – in a fit of overpowering imagination – `data`):

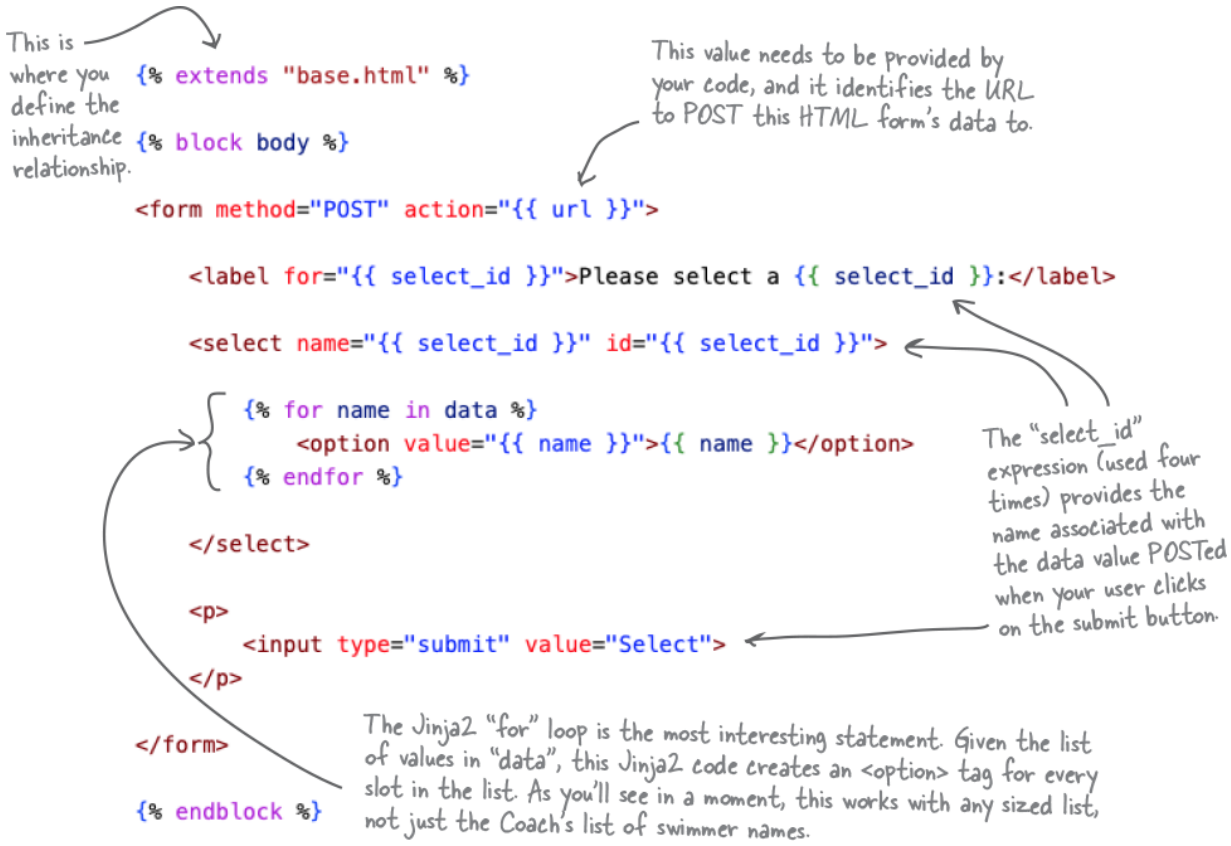


Figure 6-68.



Figure 6-69.

Fear not, Coach, we haven't forgotten why we're doing this.

This template does looks complex, but when you remove the Jinja2 markup, it's just HTML. The double curly braces substitute the value of variables. The `for` loop builds as many `<option>` tags as needed, dynamically building the `<select>` options for the Coach to choose from.

SHARPEN YOUR PENCIL

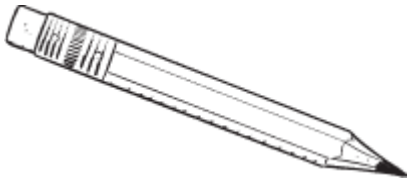


Figure 6-70.

Here is the current version of your `display_swimmers` function from your `app.py` code:

```
@aapp.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return str(sorted[session("swimmers")])
```

It's possible to adjust this code to replace the `return` statement with one which exploits the `select.html` template from the last page. If you haven't done so already, create `select.html` in your `templates` folder before continuing.

What we'd like you to do is provide the complete call you'd make to the `render_template` function to generate a new page which displays the drop-down list of swimmer names. You can assume the URL in the HTML `<form>` posts to `/showfiles`. The name of the

`<select>` tag should be set to “swimmer”. Fill in the missing code below. We’ve got you started:

```
@app.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return render_template(
        "select.html",

        _____
        _____
        _____
        _____
        _____
        _____
    )
```

The template(s) are expecting four values. Can you name them and provide their values?

Figure 6-71.

SHARPEN YOUR PENCIL SOLUTION

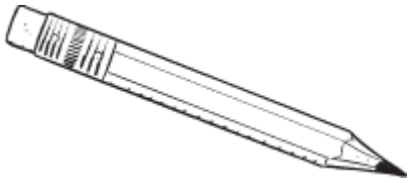


Figure 6-72.

Here is the current version of your `display_swimmers` function from your `app.py` code:

```
@aapp.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return str(sorted(session["swimmers"]))
```

It's possible to adjust this code to replace the `return` statement with one which exploits the `select.html` template from the last page. You were advised to create `select.html` in your `templates` folder before continuing.

What we wanted you to do was provide the complete call you'd make to the `render_template` function in order to generate a new page which displays the drop-down list of swimmer names. You were to assume that the URL to use is called `/showfiles`. The name of the

`<select>` tag was to be set to “swimmer”. You were to fill in the missing code. We’d started things for you:

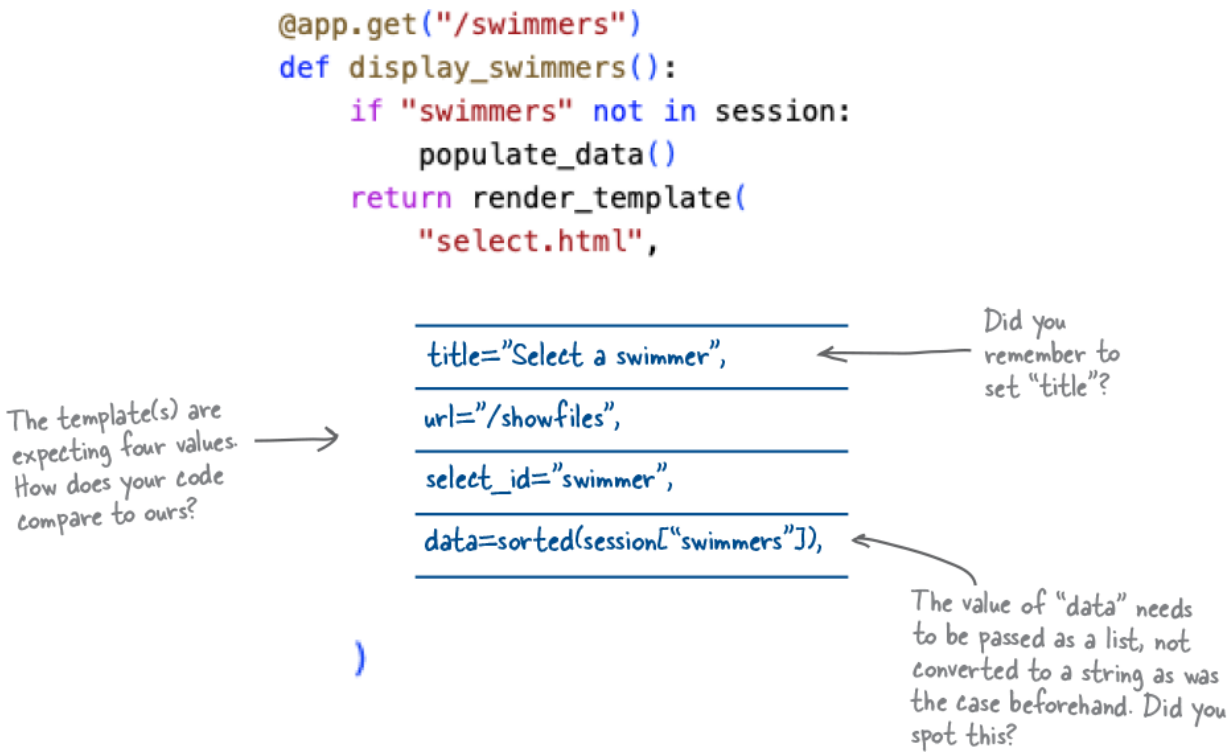


Figure 6-73.

TEST DRIVE



Figure 6-74.

Your code in `app.py` has been updated, saved, and your webapp has restarted. You now have three files in your `templates` folder: `base.html`, `index.html`, and `select.html`. Reload your webapp's home page:

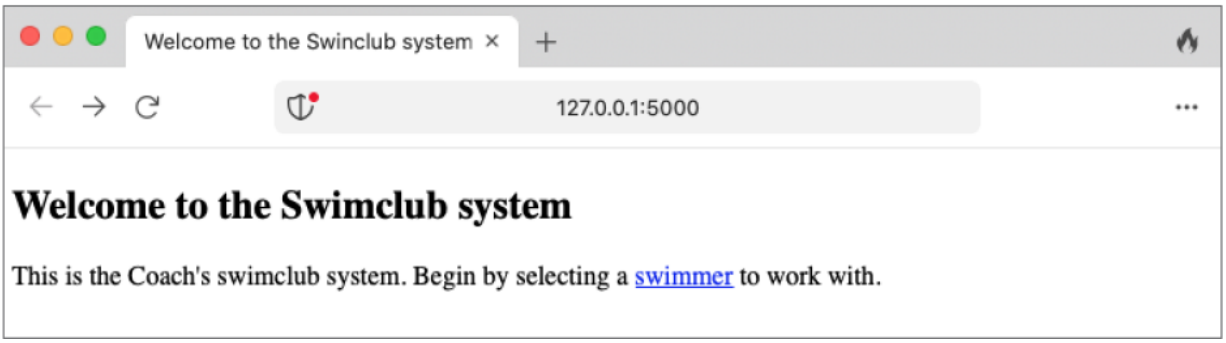


Figure 6-75.

There's no change there. But, click on that blue link and you are presented with a page inviting you to select a swimmer from a drop-down list thanks to your Jinja2-generated `<select>` tag:

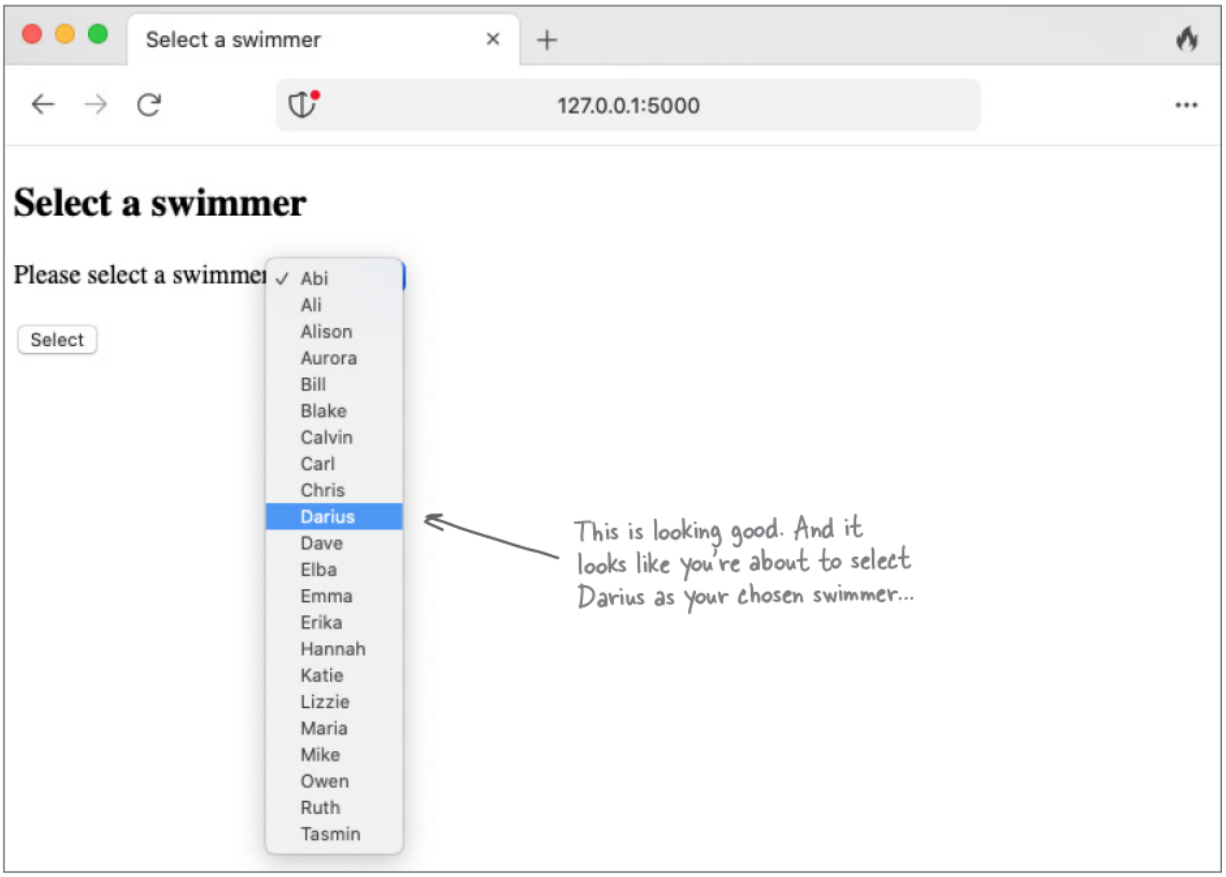


Figure 6-76.

Selecting a swimmer

The instant you select Darius from the drop-down list, then click on the **Select** button, you're greeted with the second most groan-inducing error message on the web, the dreaded `404` *Not Found* error:

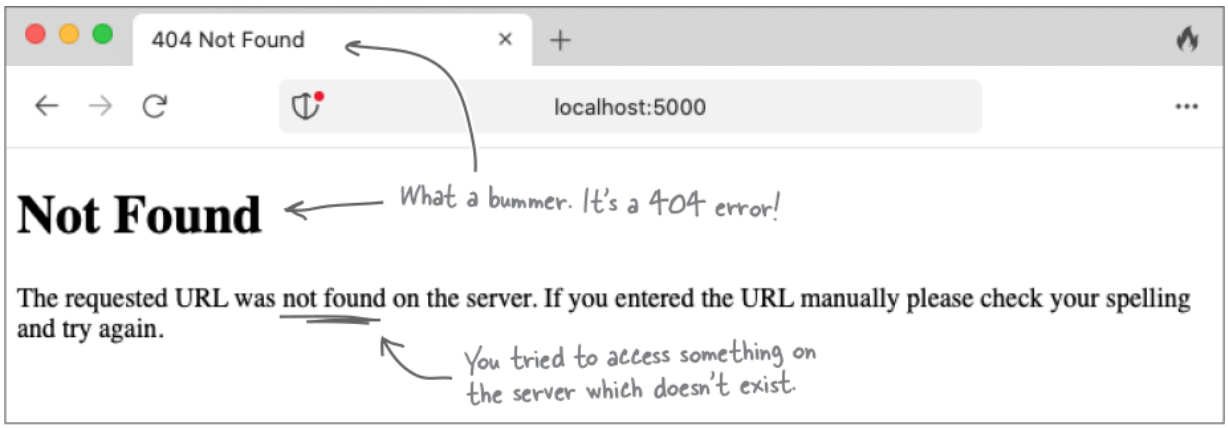


Figure 6-77.

This is exactly what we expected to happen, as the `/showfiles` URL does not currently exist within your webapp's code. You do have a `/files/<swimmer>` URL in you `app.py` code, but that URL responds to a HTTP `GET` request, whereas the HTML `<form>` in the `select.html` template uses `POST` to send data to your Flask web server.

Here's the current version of your `get_swimmers_files` code from `app.py`, which can be used to identify where code changes are needed:

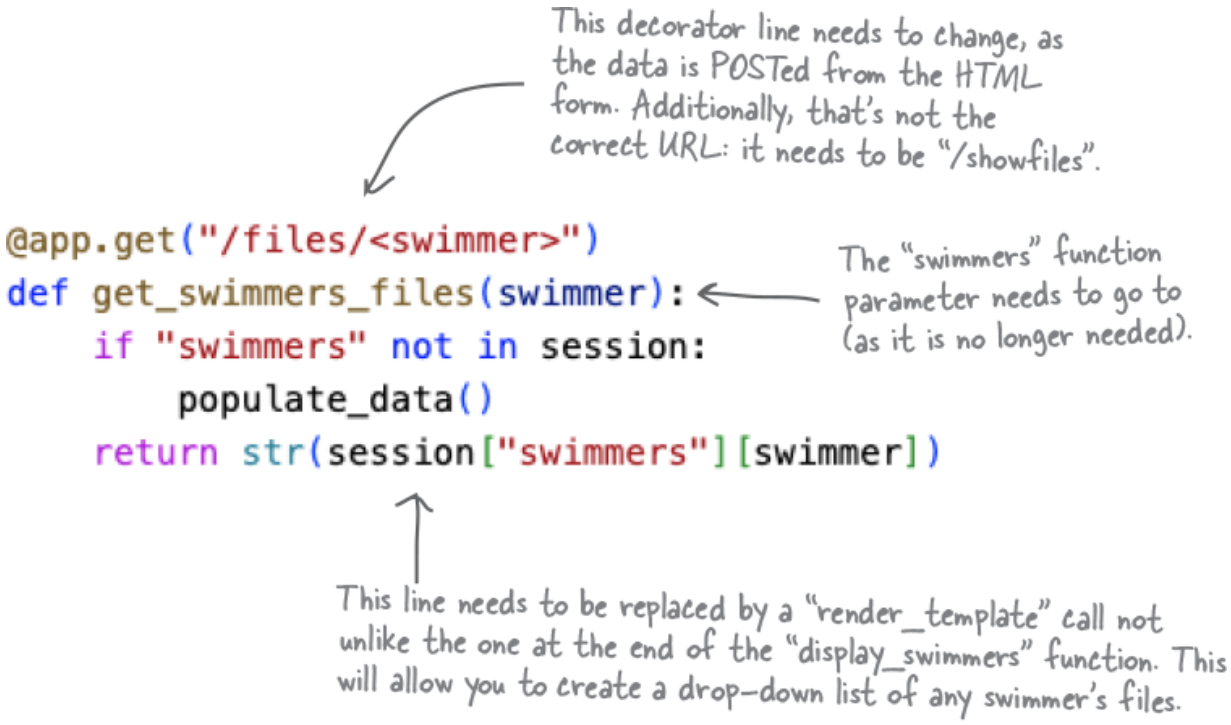


Figure 6-78.

You need to somehow process the form's data

The other issue you have is that the value for the `swimmer` variable as passed into the to `get_swimmers_files` function is provided by the `/files/<swimmer>` URL. This will no longer work as you're about to change the URL to `/showfiles`, with the data being POSTed to your webapp from your HTML form.

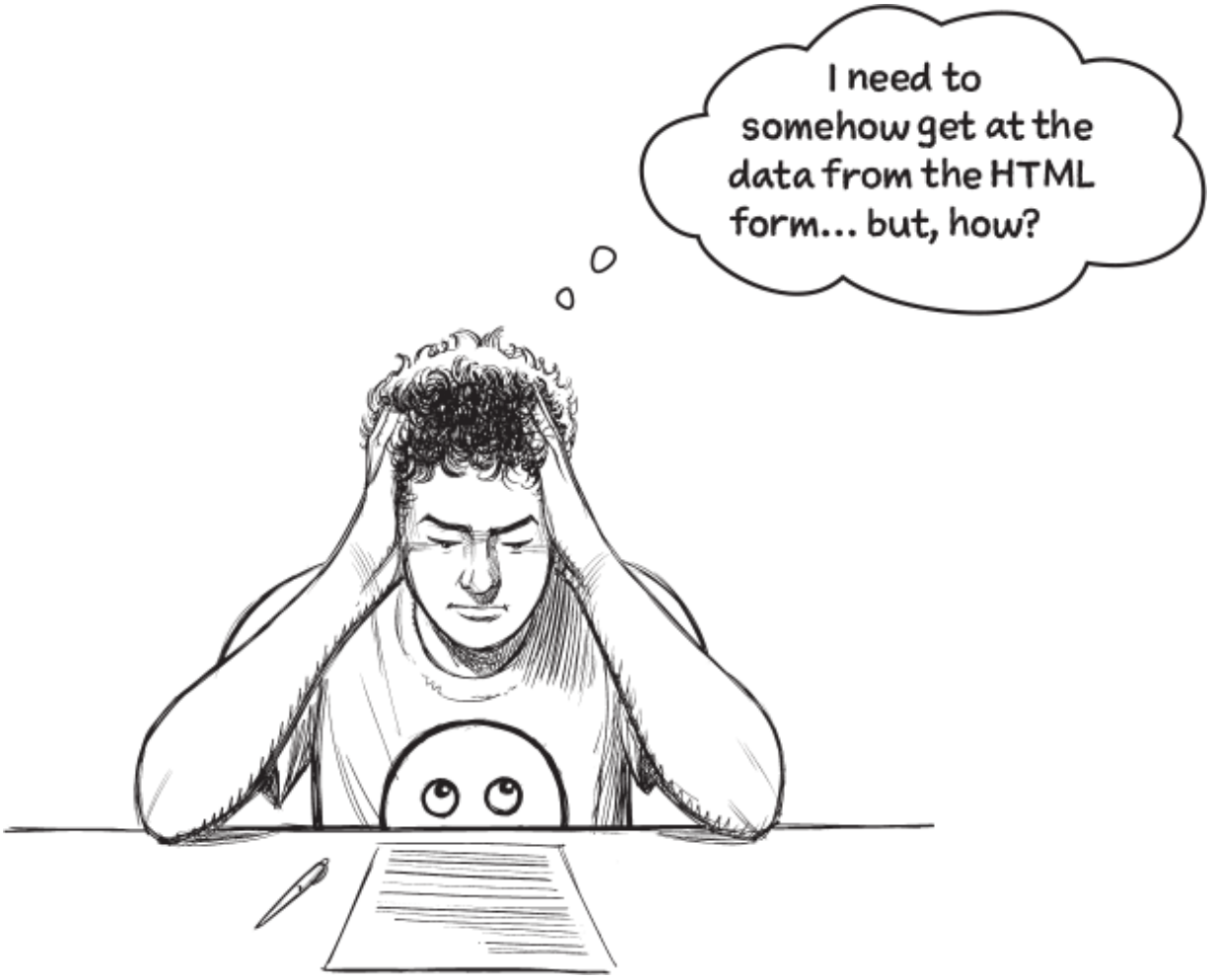


Figure 6-79.

Flask makes grabbing you form's data super easy…

…Thanks to the `request` object.

Every time your web browser communicates with any web server, it sends extra data to the web server behind the scenes. Depending on the browser you're running, this data might include the name of the browser, its underlying HTML engine, its version, your name, what

you had for breakfast, and so on. (OK, maybe not the breakfast bit, but you get the idea).

Additionally, any data sent via `POST` is also included, and Flask bundles up all this extra data into an object called `request`. All you need to do is add `request` to your list of Flask imports at the top of `app.py` and you're good to go.

NOTE

Do this now.

Your form's data is available as a dictionary

When the `request` object receives any data from your form, it adds it as a dictionary, using the `select_id` value as the key ("swimmer" in this example) and the selected swimmer's name as the associated value ("Darius" in this example).

In your webapp code, you can retrieve the name "Darius" (assuming "Darius" was selected from the drop-down list) from the HTML form using this line of code:

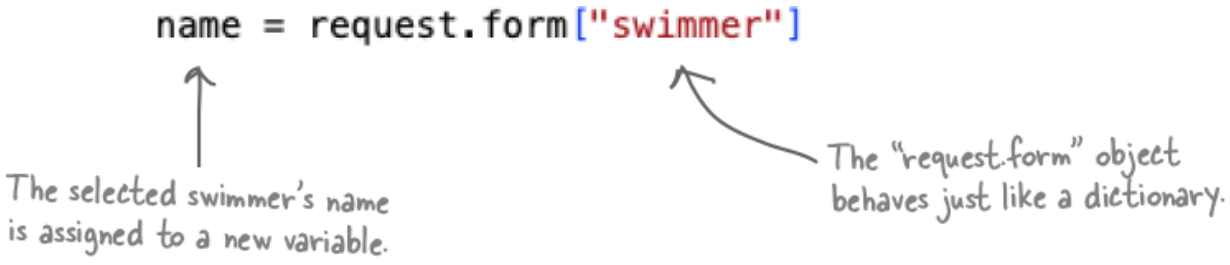


Figure 6-80.

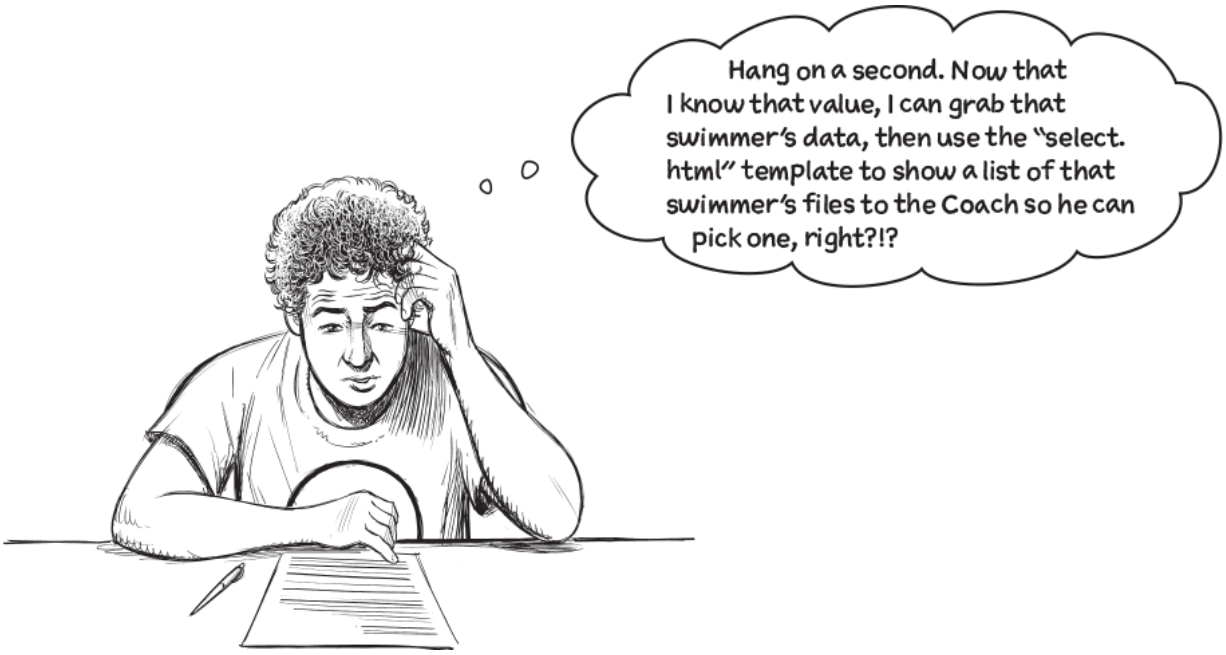


Figure 6-81.

Yes! Precisely!

You can call the `select.html` template with different values (in this case, the list of files associated with the chosen swimmer) and display another drop-down list for the Coach to pick from.

Flask Code Magnets



Figure 6-82.

What with all the progress you've been making, the *Head First Coders* got a little excited and created another function called `display_swimmer_files`. Decorated with `@app.post` and associated with the `/showfiles` URL, the function grabs the `request` object to access the swimmer's name, then uses the `select.html` template to display a drop-down list of filenames. The coders then set the `select_id` value to `file`, with `/showbarchart` as the next URL

In celebration, the code was turned into fridge magnets (of all things), affixed to the refridgerator, then the party started. In the resulting melee, someone bumped the fridge, spilling most of the magnets on the floor. Your job is to reassemble them in the correct order.

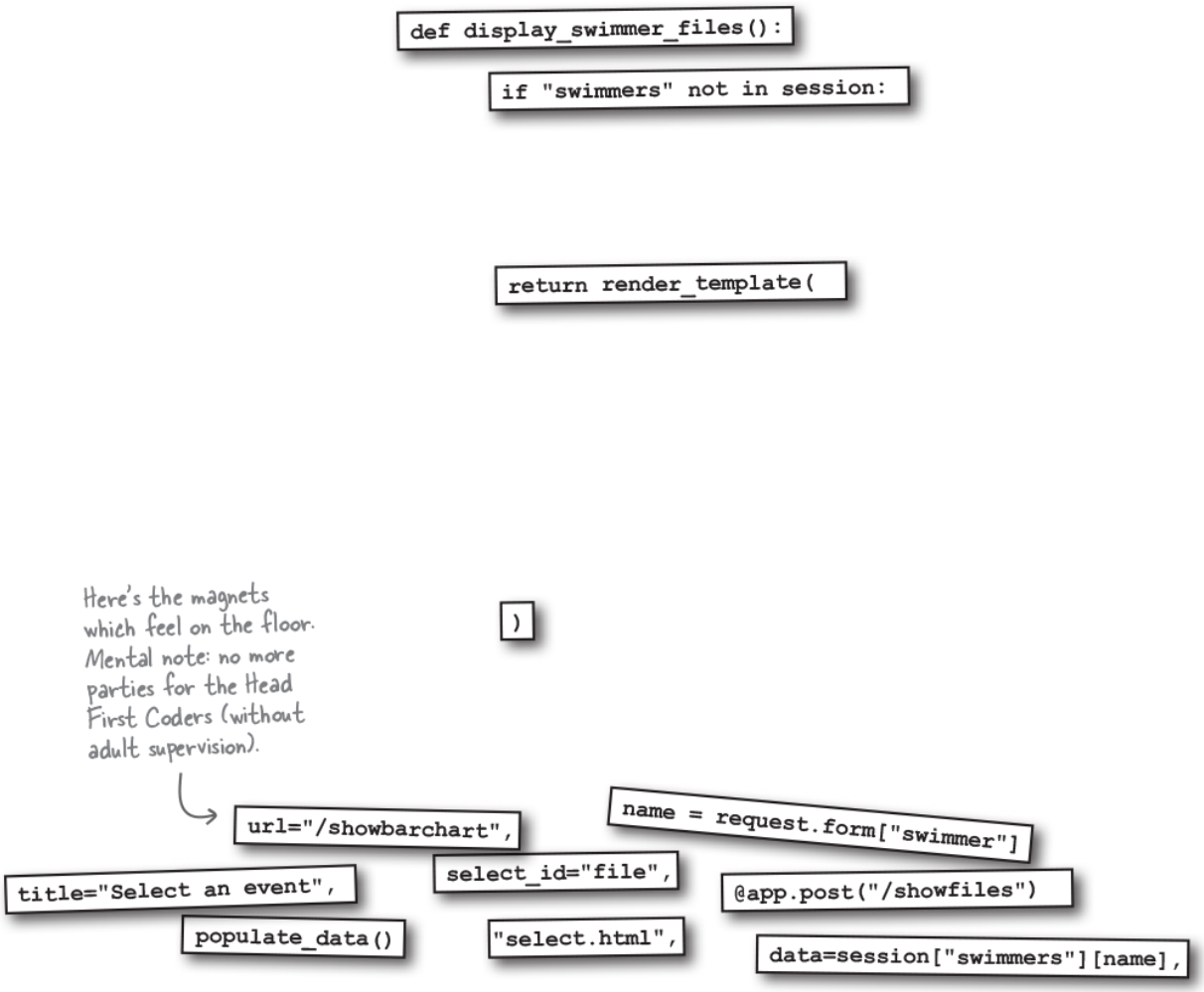


Figure 6-83.

Flask Code Magnet Solution



Figure 6-84.

What with all the progress you've been making, the *Head First Coders* got a little excited and created another function called `display_swimmer_files`. Decorated with `@app.post` and associated with the `/showfiles` URL, the function grabs the `request` object to access the swimmer's name, then uses the `select.html` template to display a drop-down list of filenames. The coders then set the `select_id` value to `file`, with `/showbarchart` as the next URL

In celebration, the code was turned into fridge magnets (of all things), affixed to the refridgerator, then the party started. In the resulting melee, someone bumped the fridge, spilling most of the magnets on the floor. Your job was to reassemble them in the correct order. How did you get on?

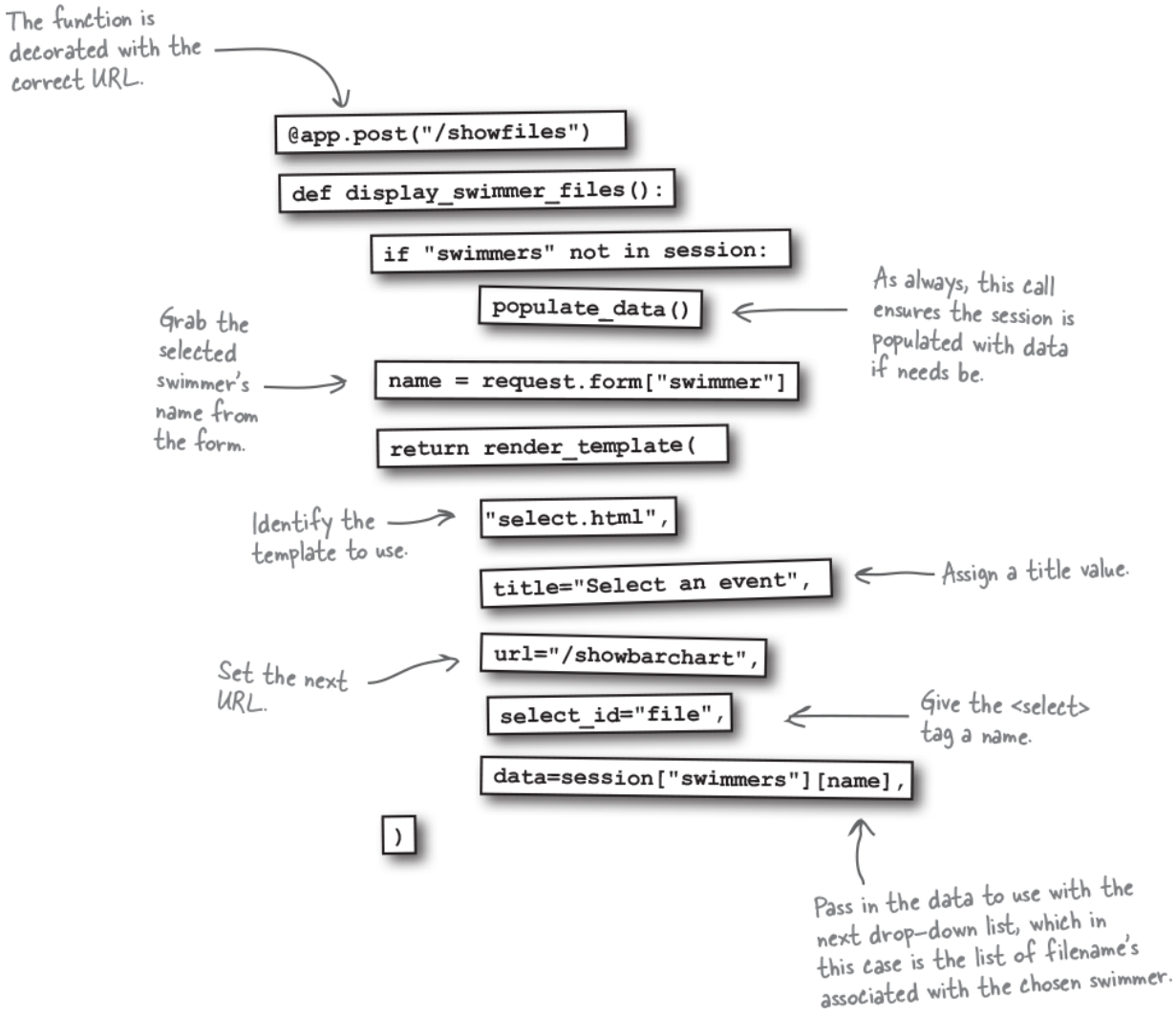

The function is decorated with the correct URL.
@app.post("/showfiles")
def display_swimmer_files():
if "swimmers" not in session:
populate_data()
As always, this call ensures the session is populated with data if needs be.
Grab the selected swimmer's name from the form.
name = request.form["swimmer"]
return render_template(
Identify the template to use.
"select.html",
title="Select an event",
Assign a title value.
Set the next URL.
url="/showbarchart",
select_id="file",
Give the <select> tag a name.
data=session["swimmers"][name],
)
Pass in the data to use with the next drop-down list, which in this case is the list of filename's associated with the chosen swimmer.

Figure 6-85.



Figure 6-86.

Selecting Darius from the first drop-down menu displays the list of events Darius swam in, shown in a new drop-down:

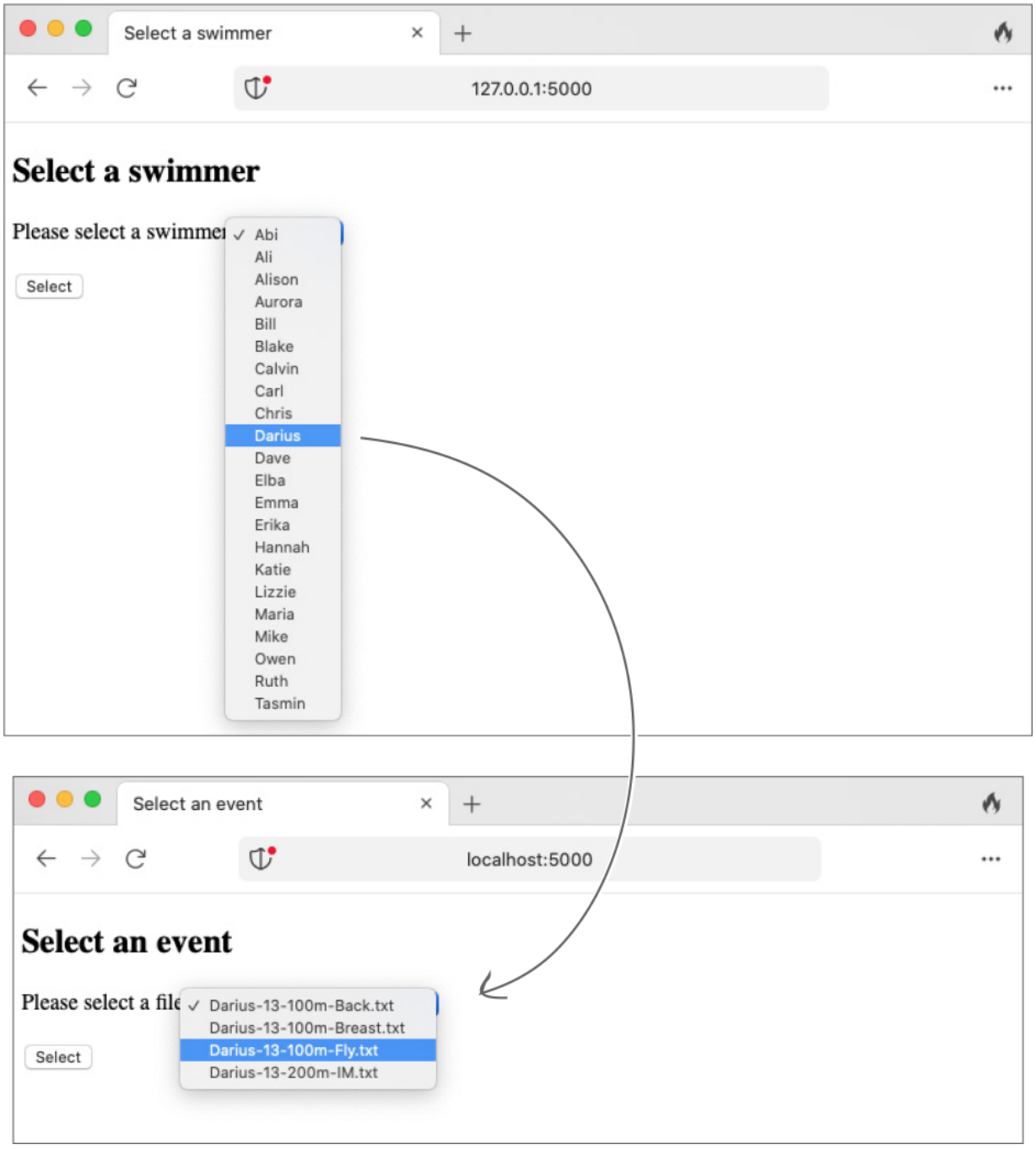

Select a swimmer
127.0.0.1:5000
Select a swimmer
Please select a swimmer
Select
Abi
Ali
Alison
Aurora
Bill
Blake
Calvin
Carl
Chris
Darius
Dave
Elba
Emma
Erika
Hannah
Katie
Lizzie
Maria
Mike
Owen
Ruth
Tasmin
Select an event
localhost:5000
Select an event
Please select a file
Select
Darius-13-100m-Back.txt
Darius-13-100m-Breast.txt
Darius-13-100m-Fly.txt
Darius-13-200m-IM.txt

Figure 6-87.

You’re inching closer to a working system

Selecting any one of the files for Darius results in another 404, and taking a quick look at your most recent function reveals why this is the case:

```
@app.post("/showfiles")
def display_swimmer_files():
    if "swimmers" not in session:
        populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"][name],
    )
```

You've yet to associate any code with this URL, so it's a case of yet another 404.

Figure 6-88.



Figure 6-89.

That sounds like a good idea.

There's just one small issue. As it's written, the `produce_bar_chart` function in `swimclub.py` expects to create its HTML files in a folder called `charts`. Flask and Jinja2 expect their HTML files to reside in the `templates` folder. We have a bit of a clash…

Functions support default parameter values

The `produce_bar_chart` function included in the `swimclub` module hard-codes the save location for any SVG-based bar charts it creates. This location is defined in the constant `CHARTS` near the top of the module's code:

NOTE

Go ahead and open the module in VS Code. Remember, you copied it into your webapp folder at the start of this chapter.

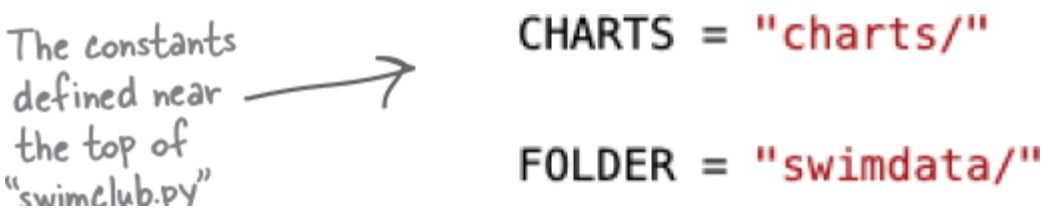


Figure 6-90.

If you jump to near the bottom of the module's code, there's a line of code which uses the `CHARTS` constant, and it looks like this:



Figure 6-91.

It's tempting to change the value for `CHARTS` to `templates/` to force the code to save to your `templates` folder: *don't do this*. Such a change would *break* any existing code which other users of your module have written, and which expects the output to be saved in the `charts` folder.

NOTE

This includes the code you've already written in earlier chapters. Yes, you depend on your own module.

Clearly you aren't going to throw your hands in the air and give up. So, *what to do*?

Python's functions support **default parameters**, which allows you can provide a function parameter with a *default* value. The default is used only if no other argument value is provided.

Let's enable this feature in `swimclub.py`. Near the bottom of your module's file, change the `save_to` line of code to look like this:

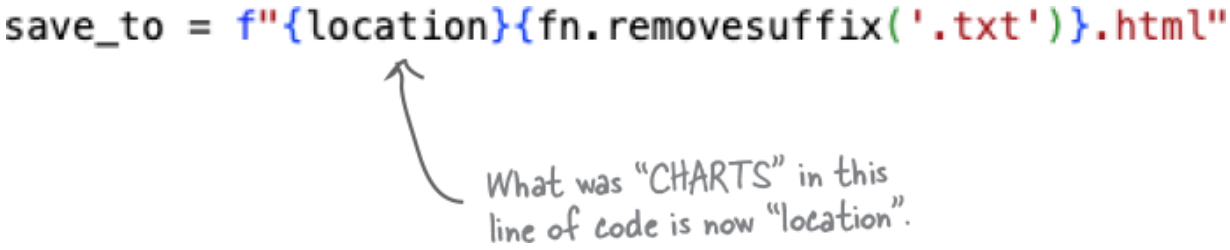


Figure 6-92.

Let's now, over the page, make the value of `location` a **default parameter** to the `produce_bar_chart` function within `swimclub.py`.

Default parameter values are optional

Jump to the top of your `produce_bar_chart` function and change the **def** line:

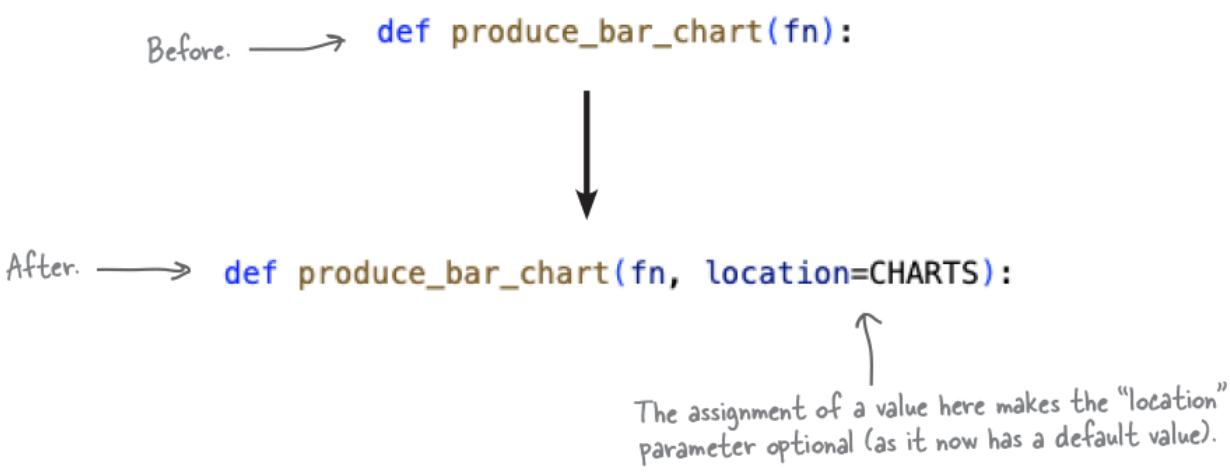


Figure 6-93.

Assigning a *default* makes that parameter *optional*.

In the “After” line, above, `location` is assigned the value of `CHARTS` if no alternate value is provided when the function is called. This small, but *crucial*, change makes the creation of your webapp’s last required function straightforward (as the bar chart creation is handed off to the `swimclub` module).

Here are the five lines of code which we used to create the `show_bar_chart` function in `app.py`, which enables your we-bapp to draw charts given a chosen swimmer’s filename. Take a look:

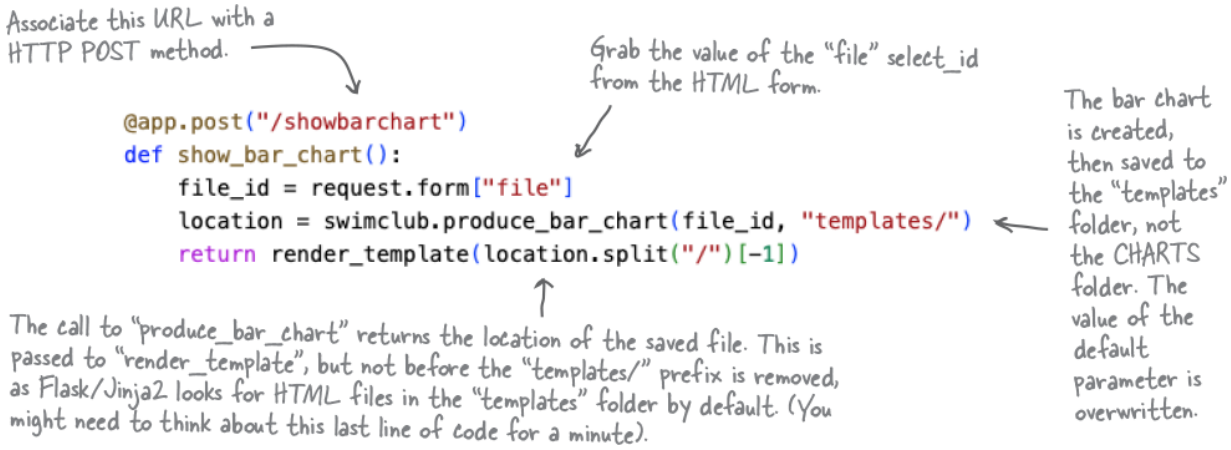


Figure 6-94.

The final version of your code, 1 of 2

Before running your latest code, compare our `app.py` code with yours, and make suer you’ve implemented as the changes we have:

```
from flask import Flask, session, render_template
```

```
from flask import Flask, session, render_template


import os
import swimclub


app = Flask(__name__)
app.secret_key = "You will never guess..."


@app.get ("/")
def index():
    return render_template(
        "index.html",
        title="Welcome to the Swimclub system",
    )


def populate_data():
    swim_files = os.listdir(swimclub.FOLDER)
    swim_files.remove(".DS_Store")
    session["swimmers"] = {}
    for file in swim_files:
        name, *_ = swimclub.get_swim_data(file)
        if name not in session["swimmers"]:
            session["swimmers"][name] = []
        session["swimmers"] [name].append(file)
```

The final version of your code, 2 of 2

```
@app.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]),
    )
```

Make sure your code matches ours.

```
@app.post("/showfiles")
def display_swimmer_files():
    if "swimmers" not in session:
        populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"][name],
    )
```

It's likely a very good idea to add a comment to the top of each of these functions. We didn't do this in order to save a bit of space on these pages (and to keep your focus on understanding the code). But adding comments is a good idea.

```
@app.post("/showbarchart")
def show_bar_chart():
    file_id = request.form["file"]
    location = swimclub.produce_bar_chart(file_id, "templates/")
    return render_template(location.split("/")[-1])


if __name__ == "__main__":
    app.run(debug=True)
```

Figure 6-95.

TEST DRIVE



Figure 6-96.

With the latest version of your `app.py` code running inside VS Code, use the system to view any bar chart for any of the Coach's swimmers. Works well, doesn't it? Even though it still looks like something from the 1990s…

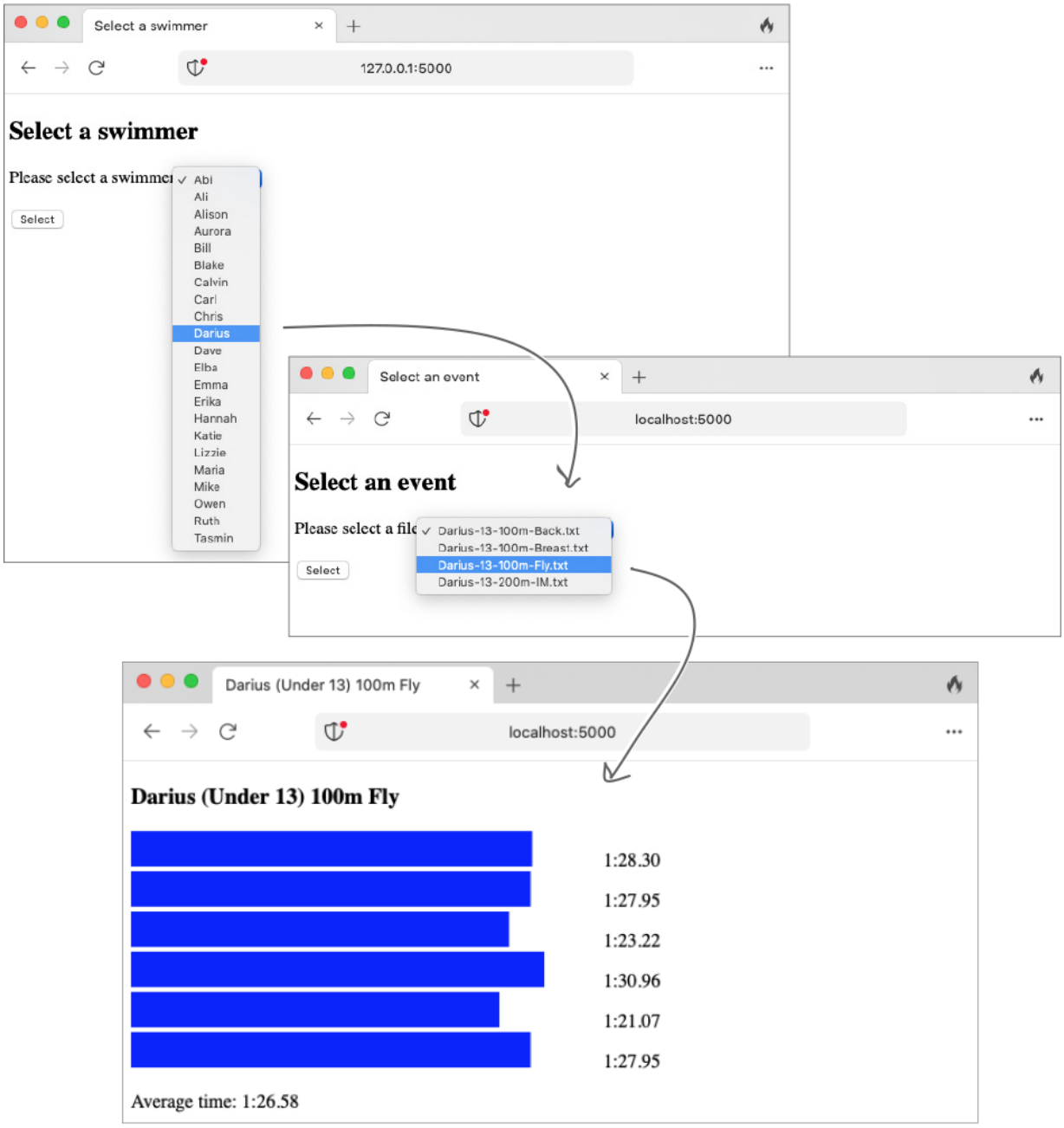


Figure 6-97.

It's a thing of beauty, isn' it?

We promised a single chapter would take the code you've developed so far in this book and turn it into a working webapp, and that's what you've gone a done in the last fifty pages.

The Coach can now select *any* swimmer from his list of swimmers, then select *any* swim event from those shown, before viewing the bar chart. Although the resulting webapp could likely do with some "CSS-love", what's here is truely useful. And I know we're biased, but we think it's *beautiful*, too.

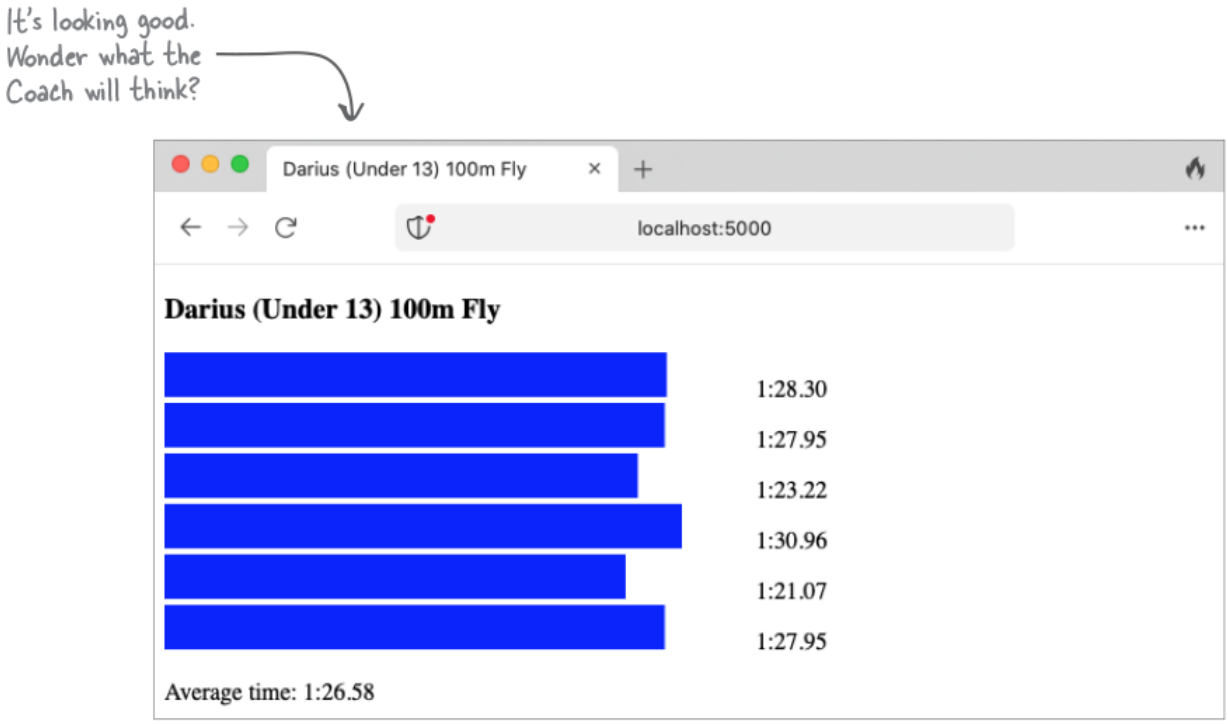


Figure 6-98.

THERE ARE NO DUMB QUESTIONS

Q: When I add code to a function associated with a Flask @app decorator, am I restricted as to what code I can use?

A: No. You can put *any* Python code in there, and the code can do anything. The only restriction is that the function must conclude with a `return` statement (as your webserver gets very confused otherwise).

Q: Can I have any number of default parameters when I create a new function?

A: Yes, sort of. Once you assign a value to a parameter (making it optional), Python assumes all subsequent parameters are optional too. So, you also have to provide default values for those parameters. Bottom line: the ordering of your function's parameters makes a difference.

The Coach's system is ready for prime time

You finally have something to demonstrate to the Coach. When the Coach sees what you've developed, he's thrilled. He can't stop high-

fiving everyone he meets. You're also pretty sure you spot a little tear in the corner of his eye…

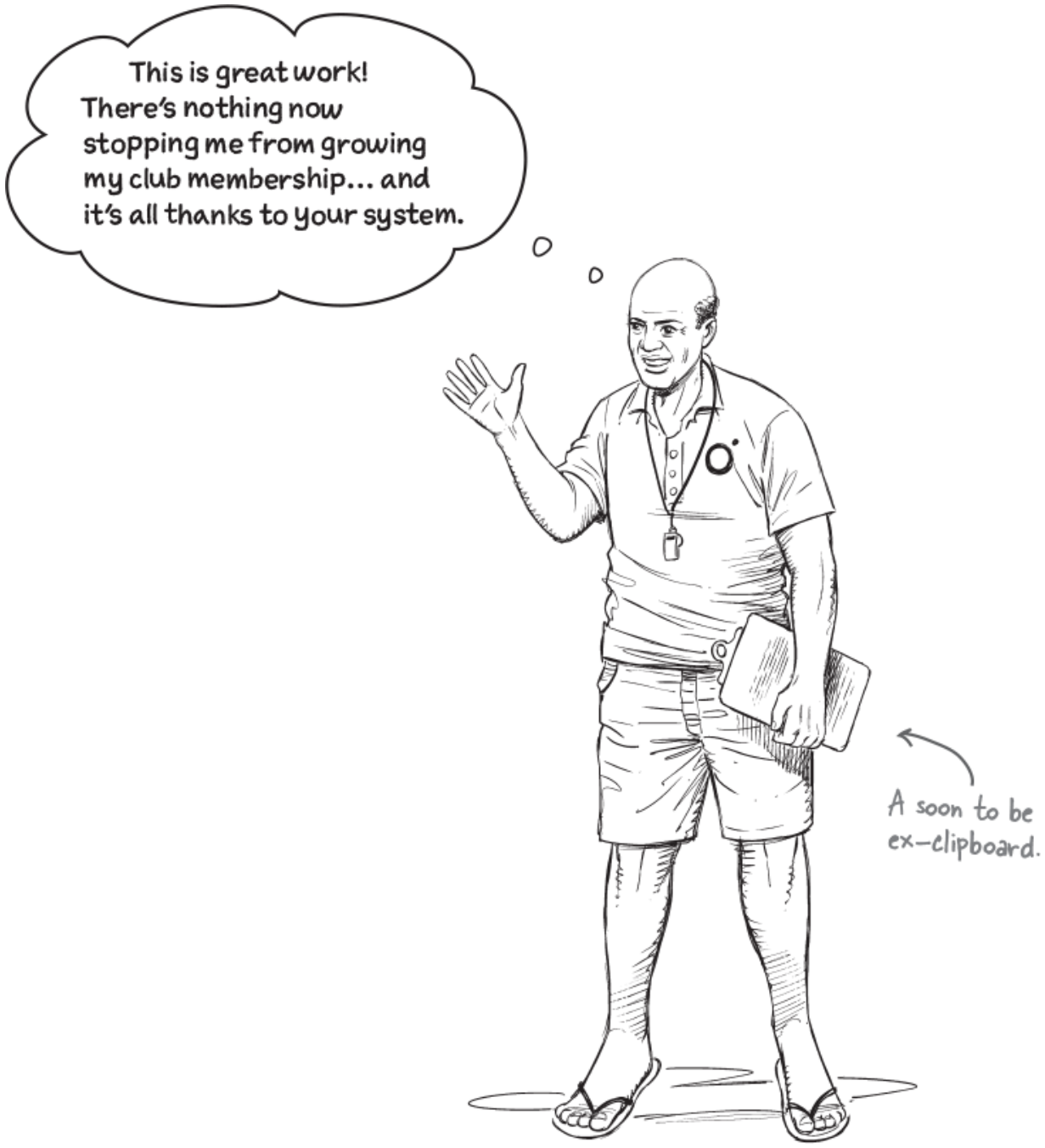


Figure 6-99.

It’s time to sit back, relax, grab a cup of your favorite brew, review the chapter summary, then (as always) have a go at this chapter’s cross-word puzzle.

See you in the next chapter when, not only will you improve the look of your webapp, you’ll deploy it to the cloud, too.

CHAPTER REVIEW BULLET POINTS

- When it comes to building a webapp with Python, there are lots of choices. We like **Flask** as it lets use get going quickly.
- Flask take advantage of Python's **decorator** feature, which uses the `@` symbol to mark a function for special treatment. In Flask, the `@app` decorator lets you associate an incoming URL with any Python function.
- Out of the box, the `@app` decorator supports the *GET* and *POST* **methods** of HTTP.
- The **minimal** Flask webapp is only seven lines of code, with four of them common to most Flask webapps no matter their purpose.
- Flask utilizes the "dunder name equals dunder main" **trick** to let you control which code runs depending on how your module is imported.
- Care is needed when creating webapp code which needs to **share data** among functions. Global variables should be avoided (unless you enjoy *carnage*). Use Flask's `session` technology instead.
- And don't forget to provide an ever-so-secret "you will never guess" **secret key** to prime Flask's session and cookie technology.
- Flask's **debugging mode** is a life-saver, so be sure to enable it with `@app.run(debug=True)`.

- Flask comes with the **Jinja2** template library built-in. Although primarily associated with build HTML pages, Jinja2 can be used for any template use-case.
- Flask's `render_template` does all the heavy lifting when hooking into Jinja2's template mechanism.
- Among other things, Jinja2 can include **expressions** within its templates, as well as **statements**. Expressions are surrounded by double curly-braces, `{{` and `}}`, whereas statements are surrounded by `{%` and `%}`.
- Examples of Jinja2 statements seen in this chapter include `block`, `endblock`, `extends`, `for`, and `endfor`. There are others, but these five are more than enough to be getting on with.
- The `extends` statement is our favorite, as it lets us create HTML pages which **inherit** from a common master web page, typically called `base.html`. In this chapter, `select.html` inherits from `base.html`, then's used in more than one place. Cool.
- Even cooler is how easy Flask makes it to access data **submitted** from a HTML form. Just read the data value you want from Flask's `request.form` dictionary. As always, square brackets are your friend here.
- Python's functions support the notion of **optional, default variables**. When a function's parameter is assigned a value, the val-

ue becomes its default which is used whenever the function is called without an argument value for that parameter.

The Web Dev Crossword



Figure 6-100.

As always, all the answers to the clues are found in this chapter and the solutions are coming right up..

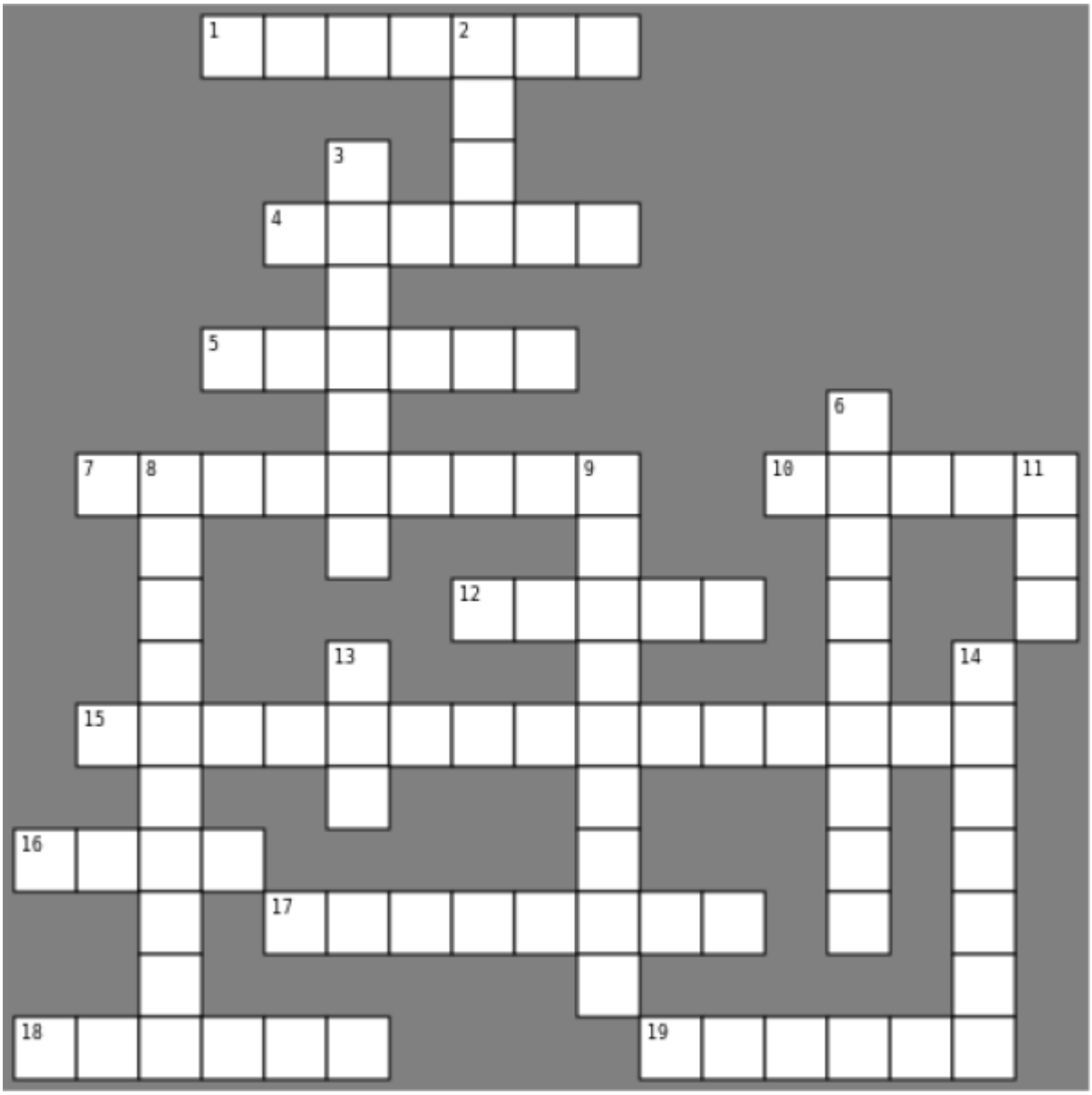


Figure 6-101.

Across

1. The inheritance keyword.

4. The drop-down menu HTML tag.

5. A required folder (empty right now).

7. Your HTML files go in here.

10. Set this to `True` in `app.run()`

12. This chapter's web framework.

15. A handy function when it comes to displaying web pages.

16. This HTTP method sends form data.

17. Parameters with assigned values are this (see 3 down).

18. The template engine used by 12 across.

19. The HTML tag for a drop-down line item (see 4 across).

Down

2. To control imports, check this dunder against dunder main.

3. Parameters with assigned values are *also* this (see 17 across).

6. The `@` symbol introduces this. What is it called in Python?

8. The `{{ }}` markup surrounds an _____.

9. The {% %} markup surrounds a _____.

11. This HTTP method (typically) makes requests to web servers.

13. It's a secret which *you will never guess*.

14. Allows for the safe sharing of data.

The Web Dev Crossword Solution



Figure 6-102.

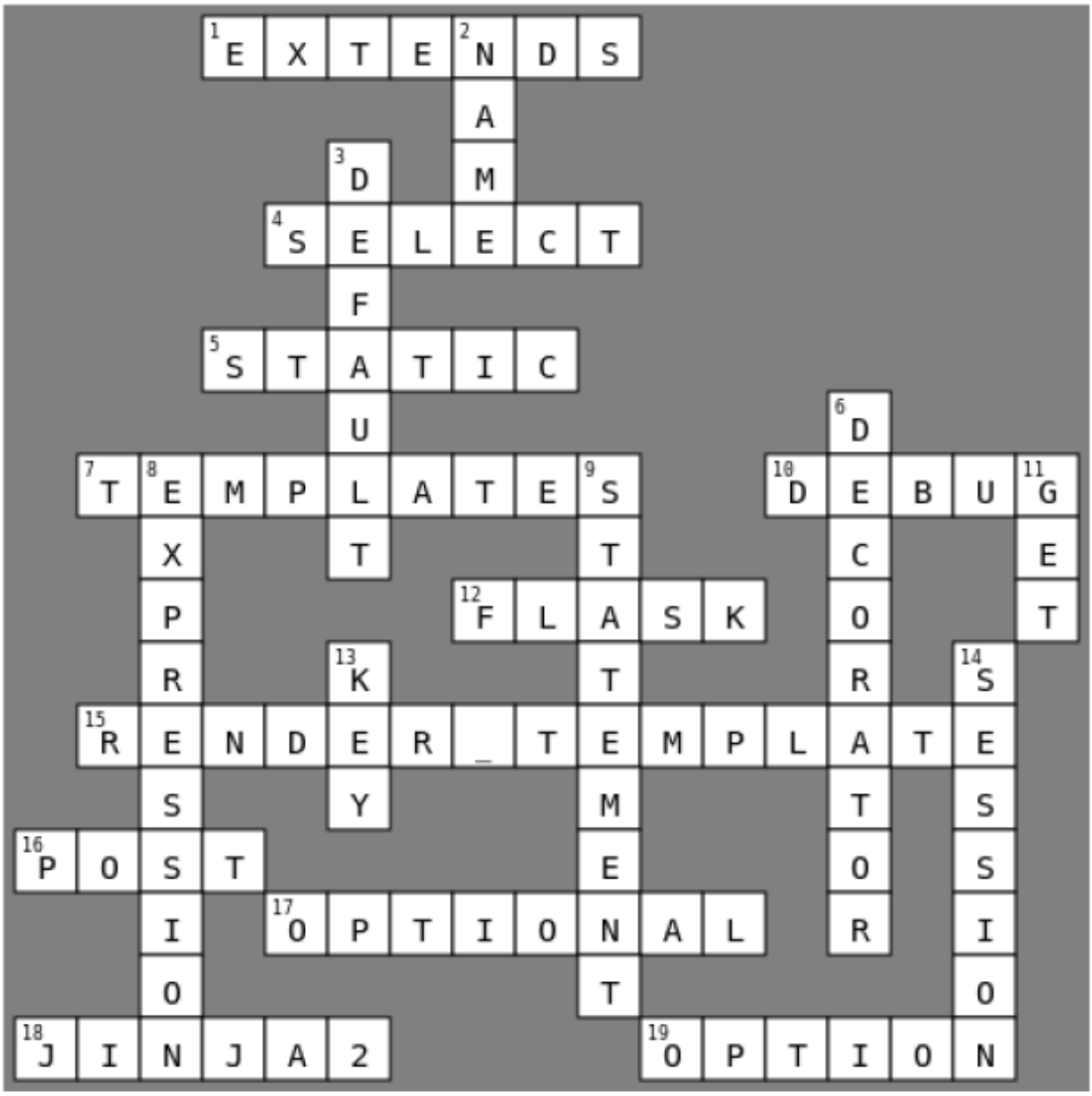


Figure 6-103.

Across

1. The inheritance keyword.

4. The drop-down menu HTML tag.

5. A required folder (empty right now).

7. Your HTML files go in here.

10. Set this to `True` in `app.run()`

12. This chapter's web framework.

15. A handy function when it comes to displaying web pages.

16. This HTTP method sends form data.

17. Parameters with assigned values are this (see 3 down).

18. The template engine used by 12 across.

19. The HTML tag for a drop-down line item (see 4 across).

Down

2. To control imports, check this dunder against dunder main.

3. Parameters with assigned values are *also* this (see 17 across).

6. The `@` symbol introduces this. What is it called in Python?

8. The `{{ }}` markup surrounds an _____.

9. The {% %} markup surrounds a _____.

11. This HTTP method (typically) makes requests to web servers.

13. It's a secret which *you will never guess*.

14. Allows for the safe sharing of data.

Chapter 7. Deployment: *Run your code anywhere*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 7 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



Getting your code to run on your computer is one thing…

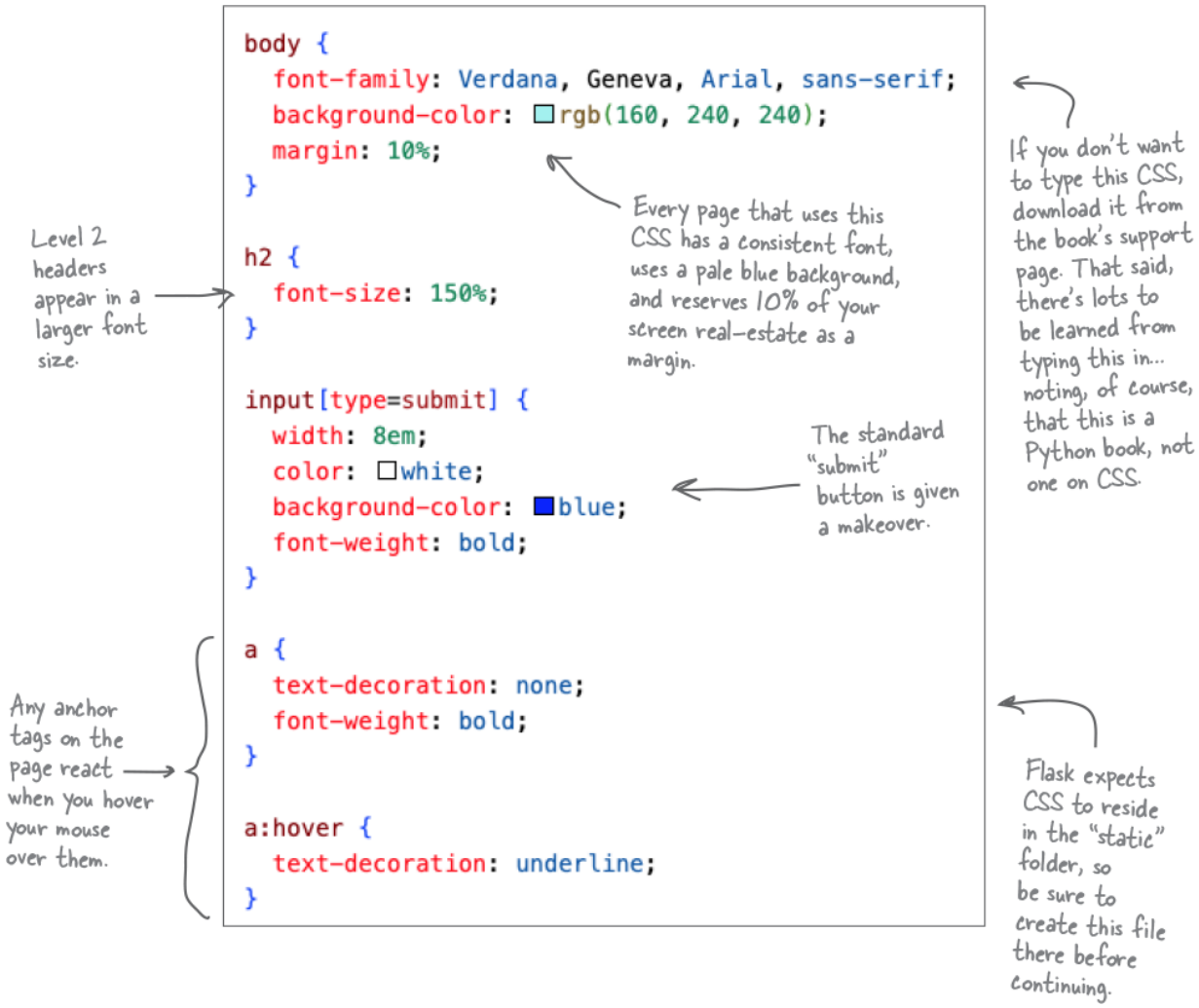
What you really want is to **deploy** your code so it's easy for your users to run it, too. And if you can do that *without* too much fuss, so much the better. In this chapter, you finish off the Coach's webapp by adding a bit of **style**, before **deploying** the Coach's webapp to the **cloud**. But, don't go thinking this is something that dials up the complexity to 11 – far from it. A previous edition of this book boasted that

deployment took “about 10 minutes”, and it’s still a quick job now… although in this chapter you’ll deploy your webapp in 10 **steps**. The cloud is waiting to host the Coach’s webapp. Let’s deploy!.

There’s nothing quite like a bit of style…

With the last chapter completed, you now have something the Coach can actually use. Before making the system available to the Coach, let’s adjust your webapp *ever so slightly* to improve its look’n’feel.

At the moment – and as pointed out earlier – it’s all looking very 1990s. Adding a stylesheet can improve things without too much work. Here’s the contents of a small CSS file called `webapp.css` which can be added to your webapp (you’ll see how in a moment):



Before adding this stylesheet to your webapp, let's take the time to make two small cosmetic changes to `app.py` and `index.html`, then see what they do.

TEST DRIVE



These are minor edits but, now that you've had time to run your web-app a few times, they tidy up the UI. The first change is to the `app.py` code (near the top), with a small edit applied to the value for the `title` parameter:

```
@app.get("/")
def index():
    return render_template(
        "index.html",
        title="Welcome to Swimclub",
    )
```

Go ahead and shorten this string.

The `index.html` template also gets a small edit, reducing the amount of text within the `<p>` tag:

```
{% extends "base.html" %}

{% block body %}
<p>
    Begin by selecting a <a href="/swimmers">swimmer</a> to work with.
</p>
{% endblock %}
```

Remove the message that refers to the Coach (as it isn't needed).

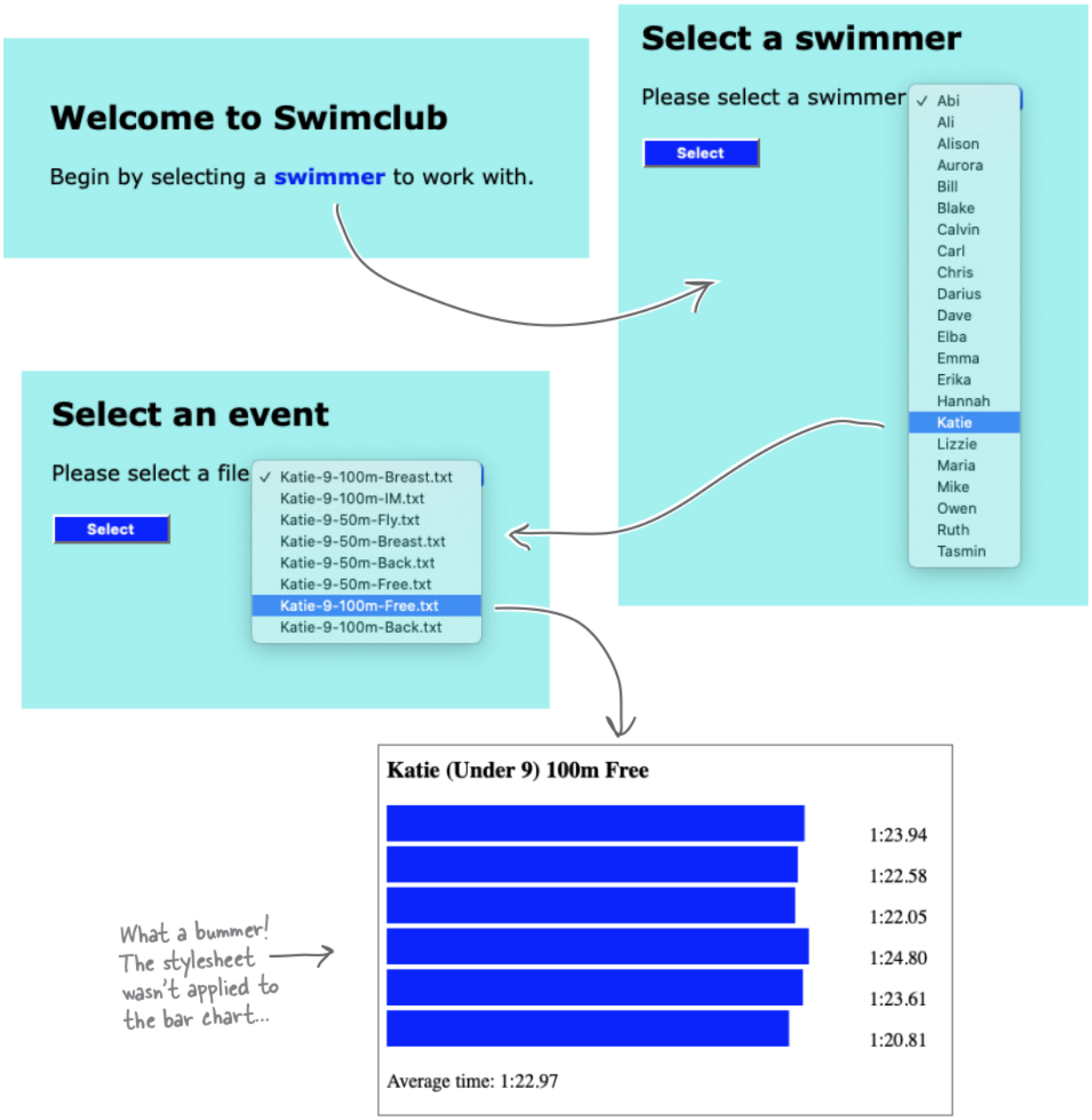
With the two cosmetic changes applied, it's now time to edit your `base.html` template so it refers to your new stylesheet. If you haven't created (or downloaded) `webapp.css` yet, do so now, then copy your CSS file into your webapp's `static` folder. When that's done, add a `<link>` tag to `base.html` so that it looks like this:

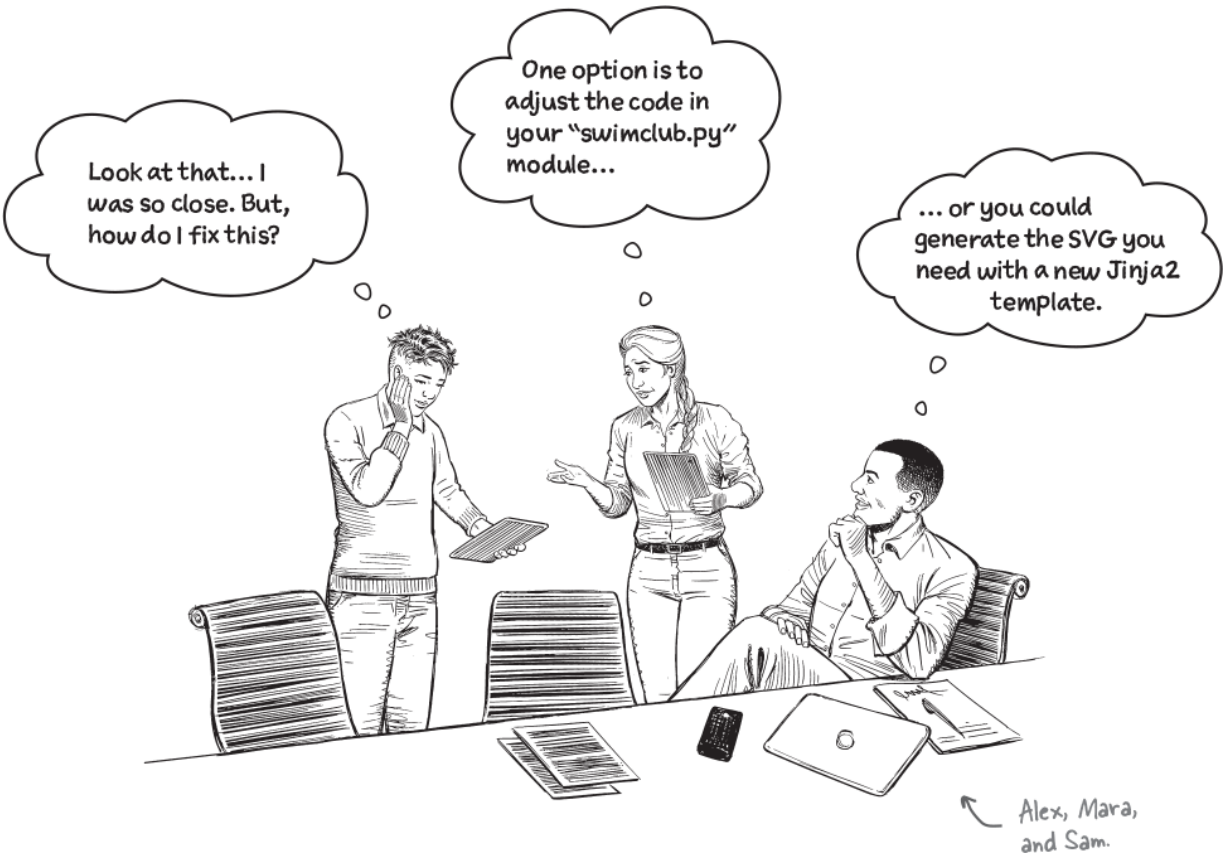
```
<!DOCTYPE html>
<html>
    <head>
        <title>{{ title }}</title>
        <link rel="stylesheet" href="/static/webapp.css">
    </head>
    <body>
        <h2>{{ title }}</h2>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

Add this <link> tag to enable your stylesheet, and don't forget to *save everything* once you've made these edits.

This Test Drive concludes on the next page. →

Select **Run** then **Run Without Debugging** to fire up your webapp (assuming it's not already running), then switch to your web browser to see what a difference these edits make, starting from your we-bapp's home page:


Welcome to Swimclub
Begin by selecting a swimmer to work with.
Select a swimmer
Please select a swimmer
Select
Abi
Ali
Alison
Aurora
Bill
Blake
Calvin
Carl
Chris
Darius
Dave
Elba
Emma
Erika
Hannah
Katie
Lizzie
Maria
Mike
Owen
Ruth
Tasmin
Select an event
Please select a file
Select
Katie-9-100m-Breast.txt
Katie-9-100m-IM.txt
Katie-9-50m-Fly.txt
Katie-9-50m-Breast.txt
Katie-9-50m-Back.txt
Katie-9-50m-Free.txt
Katie-9-100m-Free.txt
Katie-9-100m-Back.txt
Katie (Under 9) 100m Free
1:23.94
1:22.58
1:22.05
1:24.80
1:23.61
1:20.81
Average time: 1:22.97
What a bummer! The stylesheet wasn't applied to the bar chart...



There's always more than one way to do something

Each of these suggested solutions have their merits.

Adjusting the code in your `swimclub.py` module falls into the "quick-fix" category but – in this case and in our mind, anyway – doesn't appear to have any nasty side-effects waiting to bite you. Not only is the edit quick, it's short too, as it's achieved by adding a single line of HTML.

However, there's something to be said for abstracting away the dynamic creation of all of your webapp's HTML pages in a template. What's nice about this solution is that your `app.py` code isn't *littered* with chunks of HTML, unlike the code in `swimclub.py` (which is). Your webapp's logic is in your `app.py` code, while your webapp's UI and HTML are in your Jinja2 templates. This separation is attractive and appeals to your *Inner Software Engineer*, even though there's a bit more work than a single line of HTML required to realize this solution.

That said, your *Inner Coder* likes the idea of fixing this issue with a single, one line edit.

TEST DRIVE



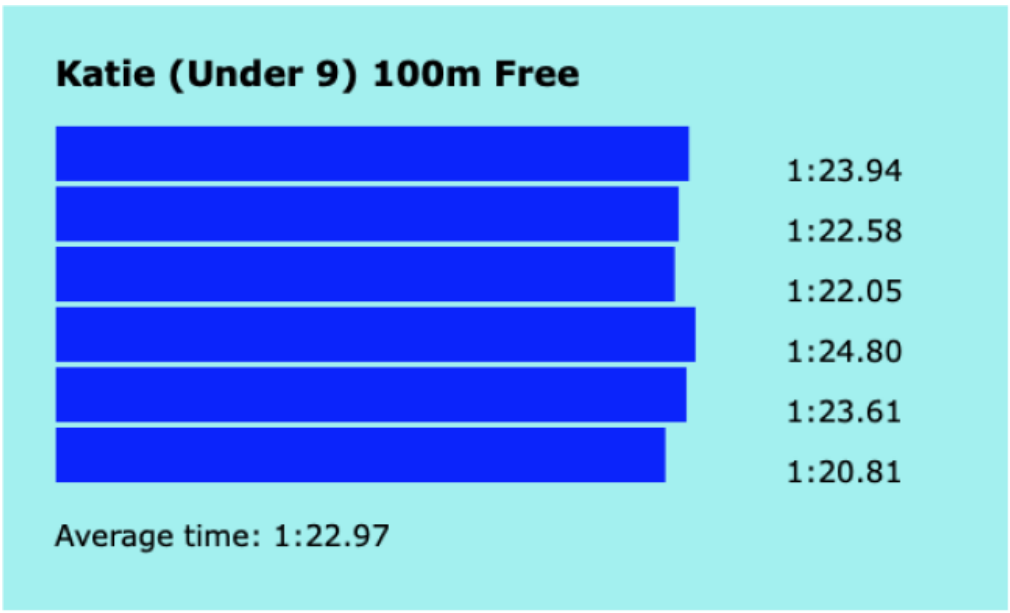
Let's listen to our *Inner Coder* (this time) and edit your `swimclub.py` code to adjust the HTML page generated for each created bar chart. Add your one-line CSS `<link>` tag to the `header` f-string in the `produce_bar_chart` function, as shown here:

```
header = f"""<!DOCTYPE html>
                <html>
                    <head>
                        <title>{title}</title>
                        <link rel="stylesheet" href="/static/webapp.css">
                    </head>
                    <body>
                        <h3>{title}</h3>"""
```

As promised, it's a one line edit.

As always, be sure to save all your code before continuing.

With the one-line edit applied and saveds, you select Katie's 100m Free file from the drop-down menu, and your webapp's stylesheet is applied. Things aren't looking too shabby, are they?

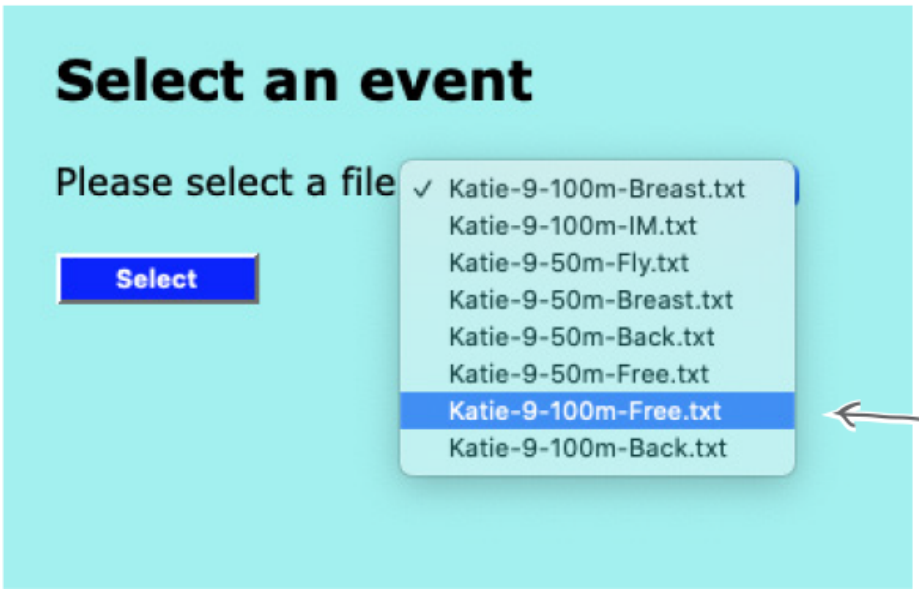


Looking, stylish, eh?

There's still something that doesn't feel right

Although your *Inner Coder* won the day when it came to adding that single line of HTML markup to `swimclub.py`, there's still something that doesn't feel quite right, and your *Inner Software Engineer* won't be quiet about it.

Take a look at the drop-down menu generated after your selected a swimmer's name from the list:



Although you now have a working webapp, this drop-down list is bugging you a little..



We agree, it doesn't look right.

Let’s adjust things to improve the look’n’feel of that drop-down menu so that it’s easier for the Coach to work with.

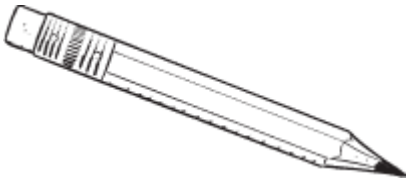


Maybe… but, let’s see.

Let’s work out what’s needed to improve the look’n’feel of the second drop-down list (the swimmer files). Then, if that works, we can worry

about what the change does to the first drop-down list (the swimmer names).

SHARPEN YOUR PENCIL



When you find yourself having to create some code, whether a single line or many pages worth, one of the best places for your experimentation is within a Jupyter Notebook. So, within VS Code, return to your `WebappSupport.ipynb` notebook, and type the following line into your next empty code cell:

```
name = "Katie-9-100m-Free.txt"
```

In the spaces below, write in the line of code that you think does what's being asked. To get things going, provide the one-liner which removes "`.txt`" from the end of the `name` variable:

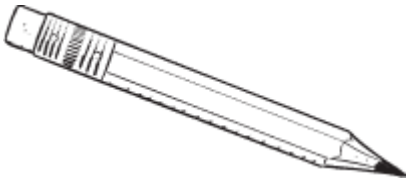
Extend your one-liner to break apart what's left over using the "-" character as a delimiter:

If you're following along in your notebook while you experiment with your one-liners, you might well believe the last line of code was a case of "close, but no cigar". It's almost there, but not quite.

Before you try to guess what to try next, take a moment to read the built-in documentation for the **split** method (hint: use the **help** BIF), paying particular attention to any optional/default parameters. Based on what you learn, extend your most-recent line of code to control how **split** breaks apart your string so the splitting occurs *twice*:

Using the square bracket notation, extend your one-liner once more to select the last item from the list returned by **split**:

SHARPEN YOUR PENCIL SOLUTION



When you find yourself having to create some code, whether a single line or many pages worth, one of the best places for your experimentation is within a Jupyter Notebook. So, within VS Code, you were asked to return to your `WebappSupport.ipynb` notebook, then type the following line into your next empty code cell:

```
name = "Katie-9-100m-Free.txt"
```

In the spaces below, you were to write in the line of code which you think does what's being asked. To get things going, you were to provide the one-liner which removes " `.txt` " from the end of the `name` variable:

```
name.removesuffix(".txt")
```

There's nothing new here, as you've seen "removesuffix" before.

You were then to extend your one-liner to break apart what's left over using the "-" character as a delimiter:

```
name.removesuffix(".txt").split("_")
```

Chain another method call to break apart the string using the "_" character.

As you're following along in your notebook while experimenting with your one-liners, you might well have believed the last line of code was a case of "close, but no cigar". It was almost there, but not quite.

Before you try to guess what to try next, you were to take a moment to read the built-in documentation for the **split** method, paying particular attention to any optional/default parameters. Based on what you learned, you were to extend your most-recent line of code to control how **split** breaks apart your string so the splitting occurs *twice*:

```
name.removesuffix(".txt").split("_", 2)
```

An optional parameter to "split" controls how many times the split occurs.

Using the square bracket notation, you were to extend your one-liner once more to select the last item from the list returned by **split**:

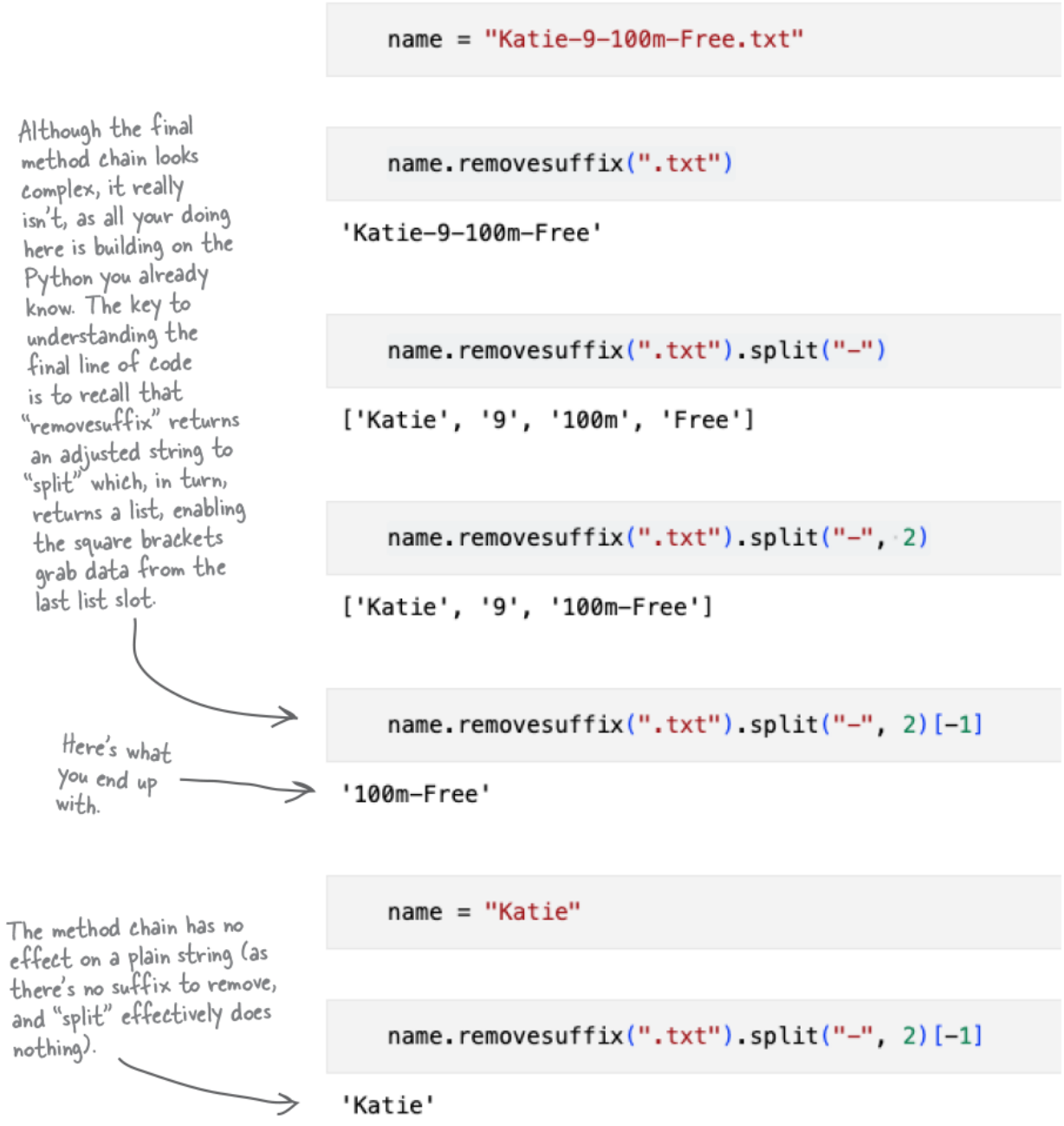
```
name.removesuffix(".txt").split("_", 2)[-1]
```

As a list is returned by "split", you can easily grab the last item from the list using a negative index value.

TEST DRIVE



After completing the last *Sharpen*, here's the code we ended up with in our notebook. Note how, in the last two code cells, we decided to see what happens if the method chain is applied to a plain string:



Jinja2 executes code bet ween {{ and }}

Now that you have your one-liner, you can add it into your `select.html` template.

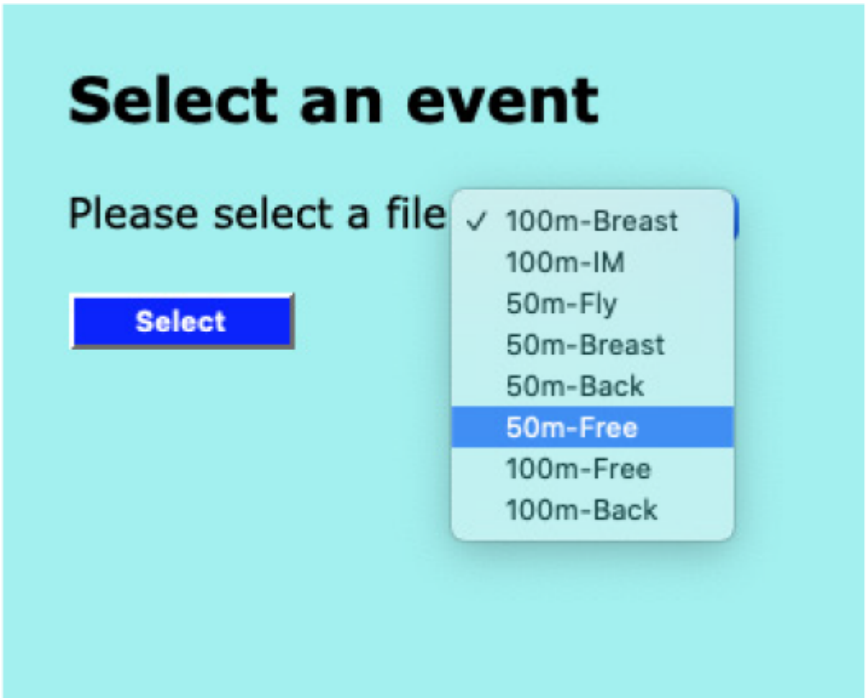
Go ahead and adjust your `<option>` markup to look like this:

```
{% for name in data %}
    <option value="{{ name }}">{{ name.removesuffix(".txt").split("-", 2)[-1] }}</option>
{% endfor %}
```

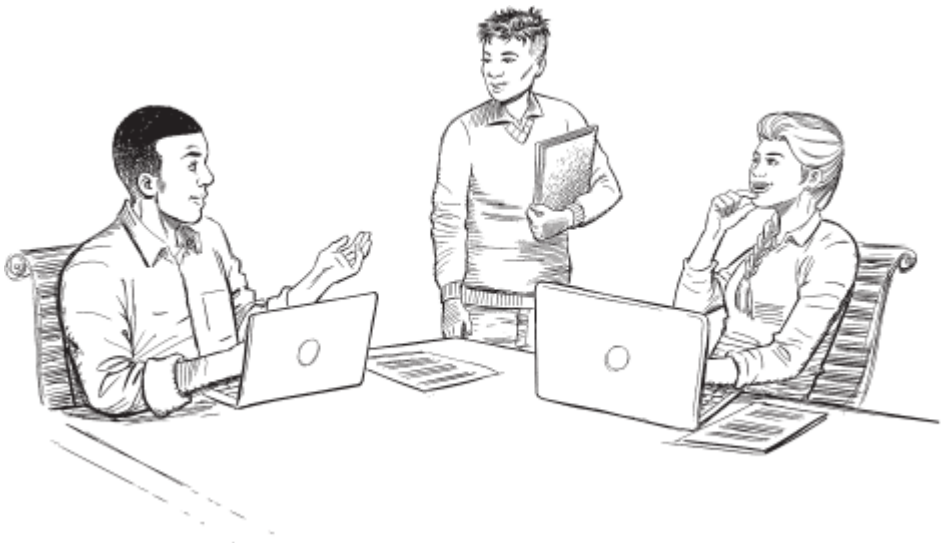
Although this is likely to be on the more complex side of things, this does show that *any* Python expression (no matter how complex) can appear between those double curly braces. (Gotta love Jinja2, eh?).

With your edit applied (and everything saved), returning to your running webapp now produces a drop-down list for the swimmers events which is a little nicer on the eye:

This is looking good!



With this last change applied, you're done. The Coach's webapp is working (and looking lovely) on your computer. Now all you need to do is deliver it to the Coach.



Cubicle Conversation

Mara: Let's make this webapp available to the Coach. Any suggestions as to how? I'm all ears…

Sam: Even though we've built a webapp, we can't expect the Coach to be in a position to self-host, can we?

Mara: Not really. He's a swim coach, not an IT expert.

Alex: So… we should look at something like a managed hosting service?

Sam: Yes. We just have to pick one.

Alex: Everyone seems to use *Amazon Web Services* these days…

Sam: …or *Microsoft Azure* …or *Google Cloud* …or *Heroku from Salesforce* …or *IBM SmartCloud*. The list goes on and on.

Mara: And it's like there's a new one popping up every week!

Alex: My heads spinning, folks.

Sam: Let's consider what the Coach *needs*.

Mara: Obviously, support for Python is a must.

Sam: And whatever we use needs to be inexpensive. Free would be even better.

Alex: I'm guessing the Coach doesn't really care what we use just so long as we give him access to his functionality?

Mara: Yes. That's a good point, which means we're free to pick whichever hosting service we want given the Coach's requirements.

Sam: In that case, I'd opt for *PythonAnywhere*.

Alex: Python*what*? I've never heard of that…

Mara: Of course! I always forget about that one. That's Anaconda's hosting service, specifically built for Python webapp developers looking to host their Python projects online. It might well be a perfect fit.

Sam: Agreed. There's no better starter service when deploying a Python webapp, IMHO.

Alex: Is *PythonAnywhere* easy to setup and use?

Mara: Let's find out, shall we?

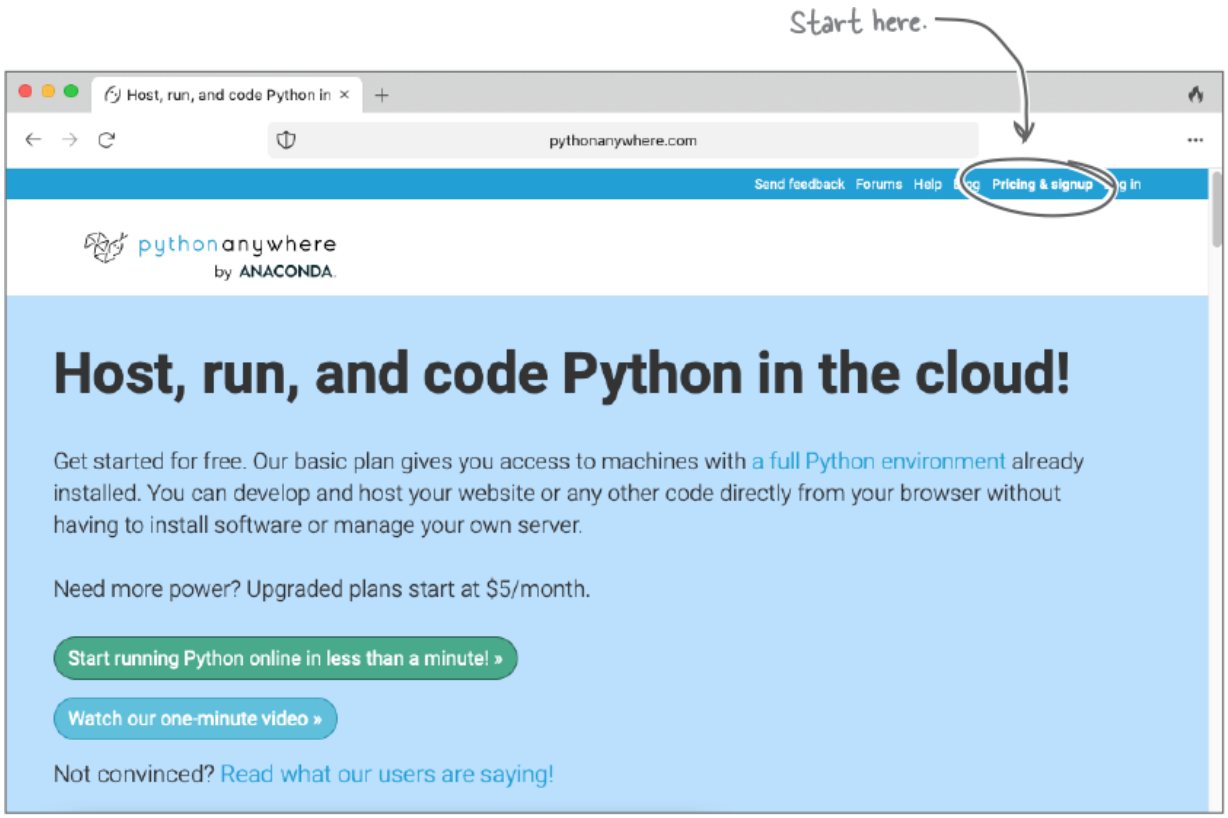
The ten steps to cloud deployment

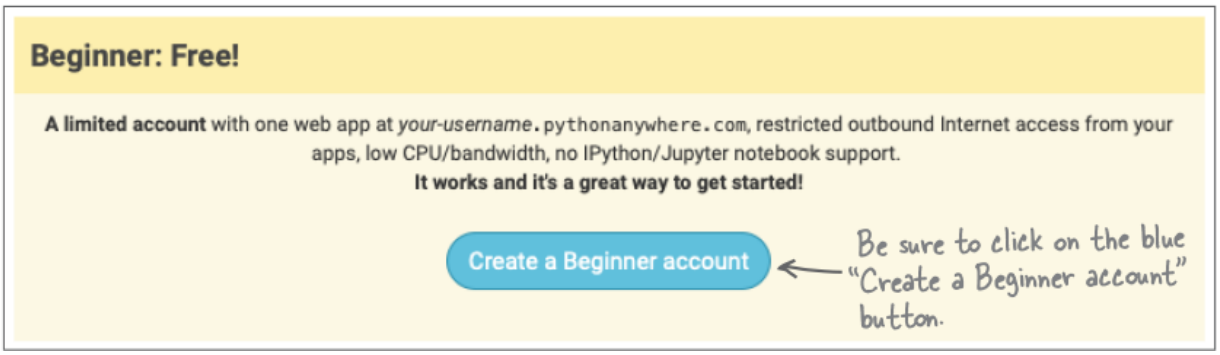
With a decision made to run on *PythonAnywhere*, let's work through the process of deploying to this Python-focused cloud platform. There are ten steps in total.

Be sure to follow along on as you learn about the process.

1 Create a Beginner Account on PythonAnywhere.

Surf on over to https://pythonanywhere.com to begin the process of creating a new beginner account.





A beginner account is all you need

The PythonAnywhere *Beginner* account does have *some* usage restrictions. However, for what you need with the Coach's webapp, you

won’t be bothered by them any time soon.

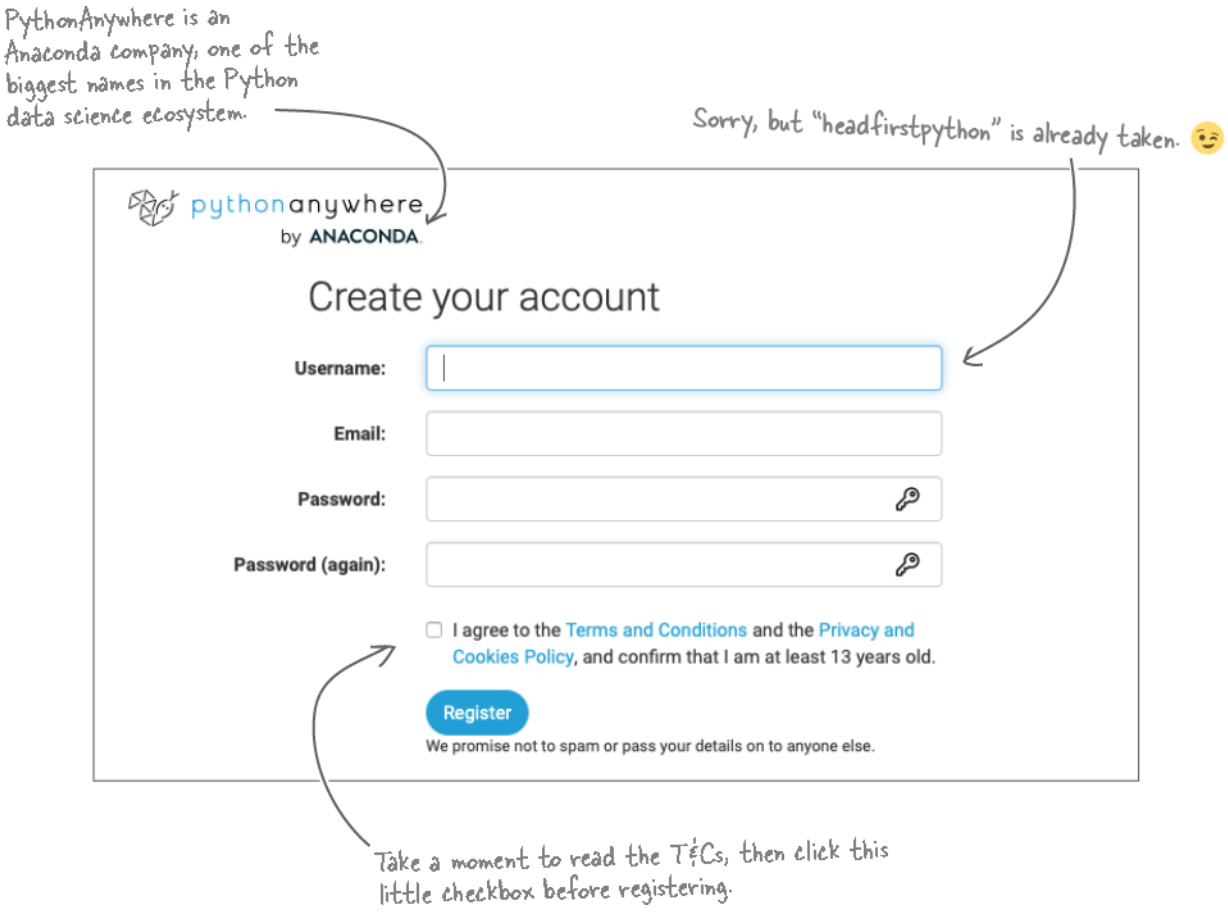
NOTE

Even if you do end up hitting the beginner limits, paid plans start from as little as $5 per month.

---

2 Provide all the usual details.

Go ahead and register your details with PythonAnywhere using the provided sign-up form.



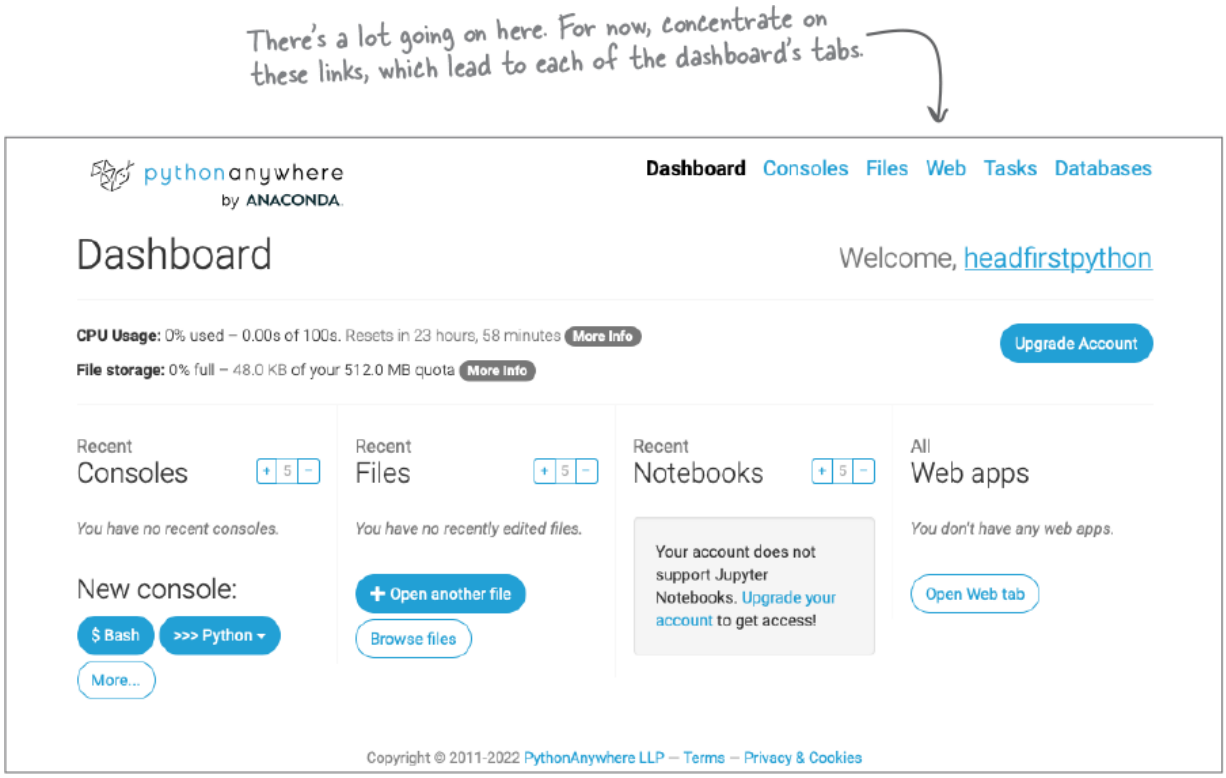
You might think you'll have to wait to confirm your email address before continuing, right? Well, the PythonAnywhere folks work hard at *not* getting in your way…

There's nothing stopping you from starting…

Although PythonAnywhere just sent an email to confirm your email address, they are trusting sorts and are more than happy to let you

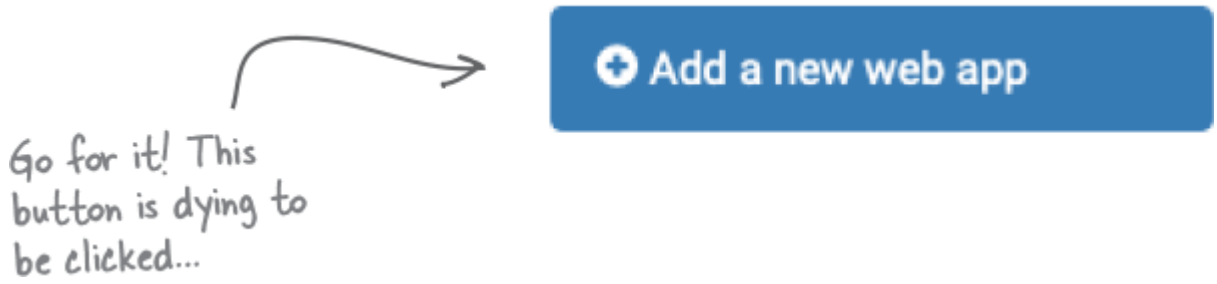
proceed (with the strict understanding that you'll confirm your email soon).

PythonAnywhere's dashboard appears.



3 Click on the Web tab, then create a new webapp.

Beginner accounts on PythonAnywhere can have one webapp active at any one time. You have yet to create any, so let's fix that.

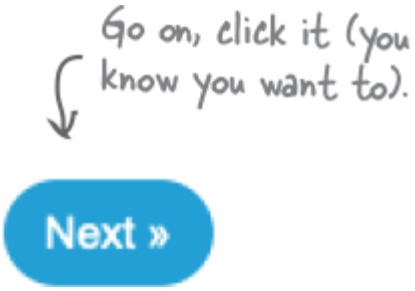


When in doubt, stick with the defaults

Upon clicking that big blue button, you'll see a message informing you your account can't employ a custom domain name (without upgrading). That's no biggie. By default, your webapp runs on a domain name created by prefixing your newly-registered PythonAnywhere username with the `.pythonanywhere.com` postfix. This combination (for us) results in the following web address:

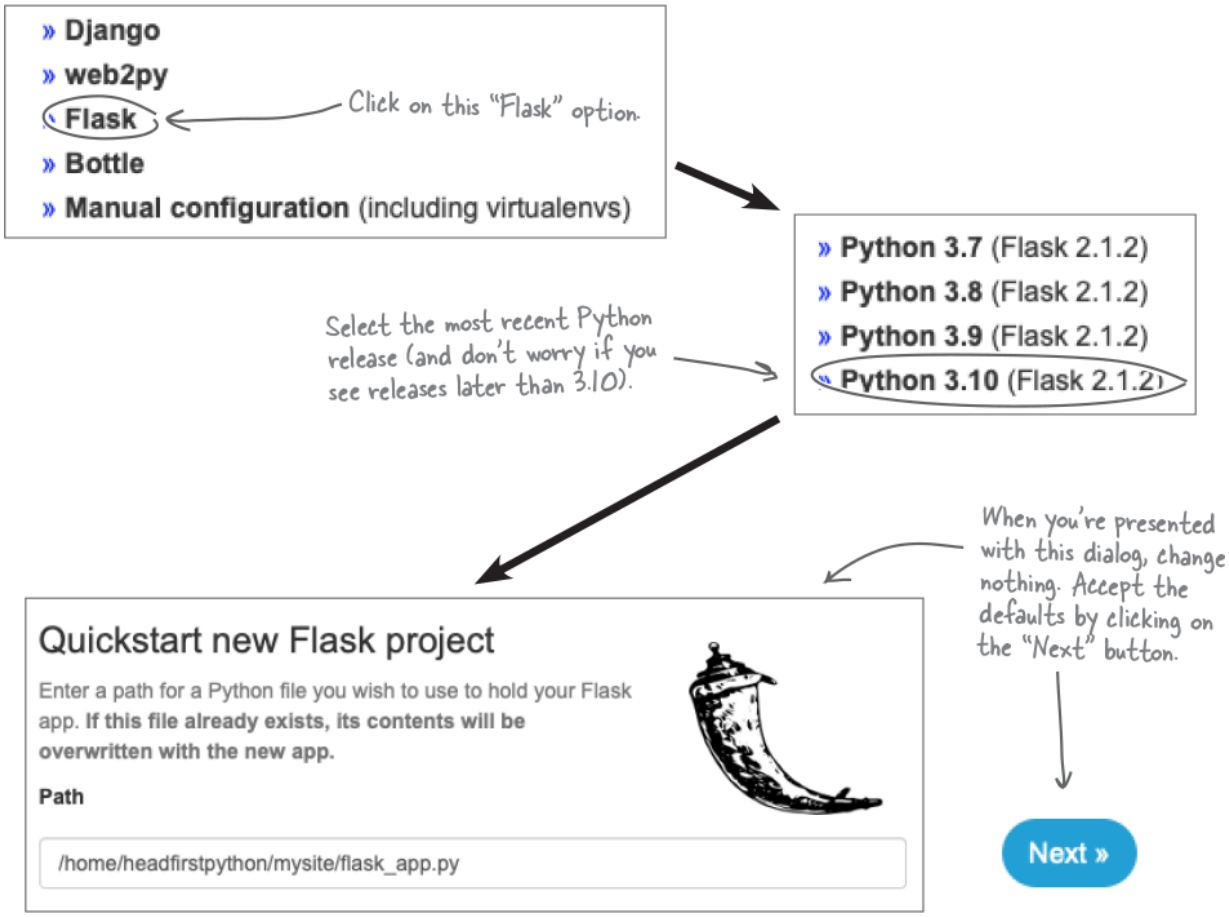
```
headfirstpython.pythonanywhere.com
```

Go ahead and click on the **Next** button.



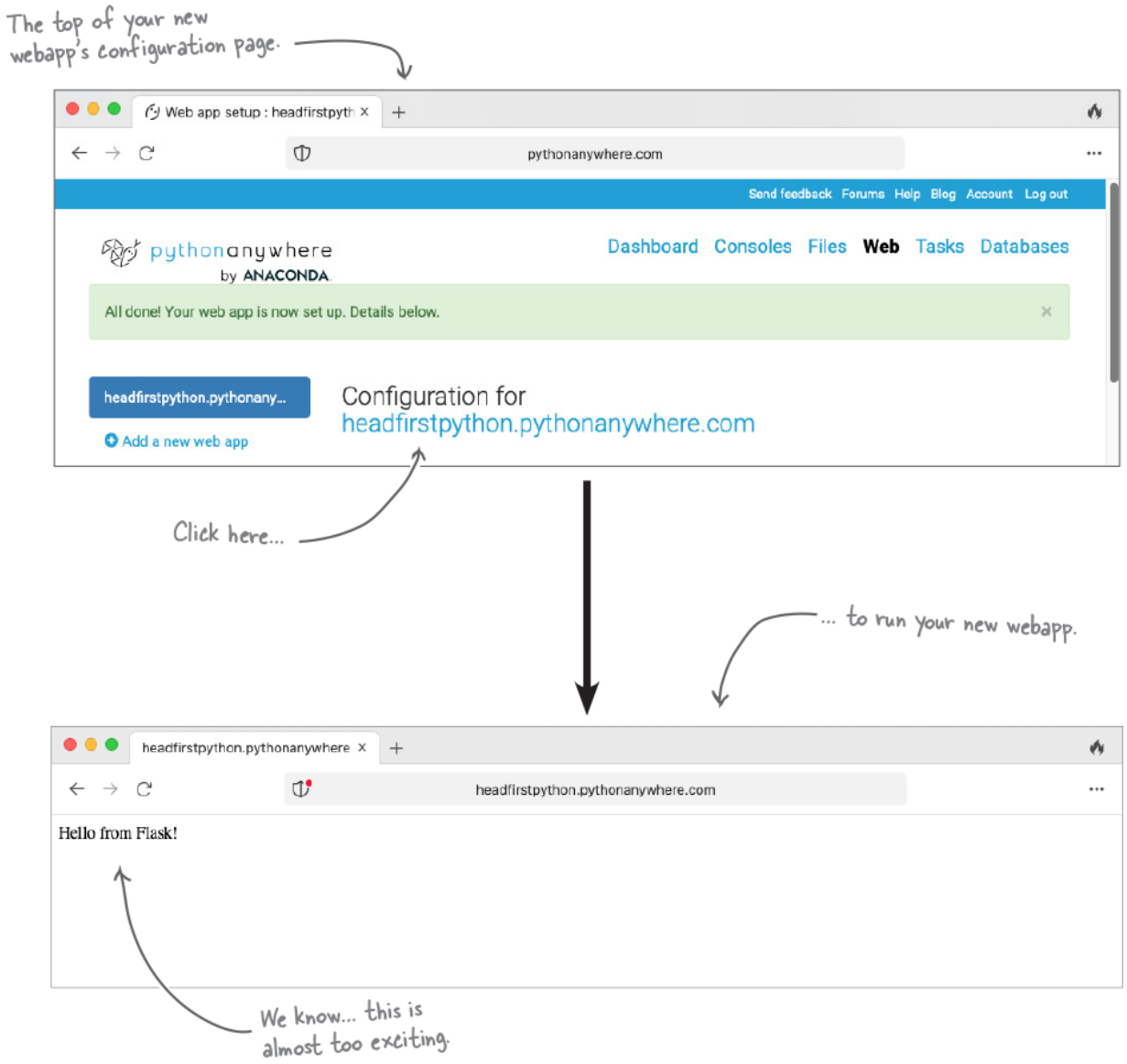
4 Select “Flask” to create a default, placeholder webapp.

A list of choices of web framework appears. Click on the “Flask” option, then select the most recent Python release from the list that follows.



The placeholder webapp doesn’t do much

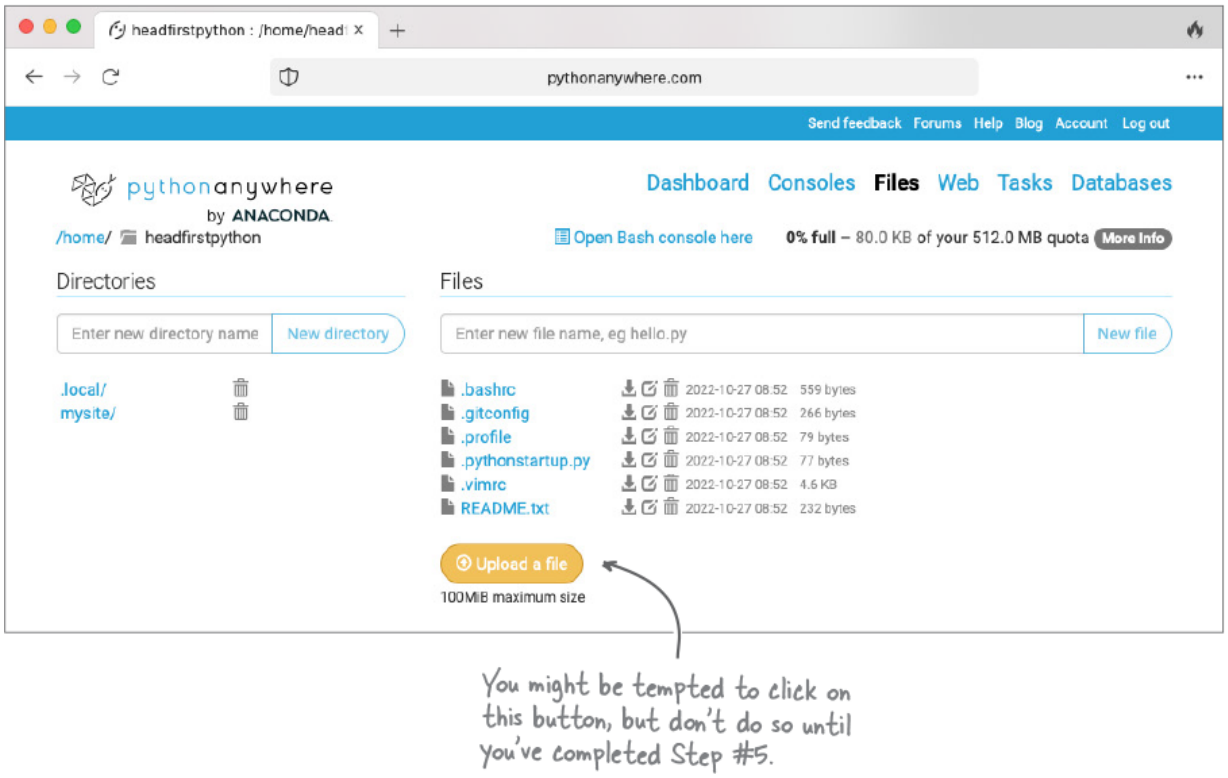
You're presented with the web set-up confirmation page for your new webapp, which provides a bunch of configuration options. For now, let's not adjust any of these. Instead, click on the blue-highlighted URL for your new webapp to see it in action:



When you're done basking in the glory of the default Flask webapp, press your browser's **Back** button to return to the configuration page.

Deploying your code to PythonAnywhere

Having pressed your browser's **Back** button, you are returned to the PythonAnywhere **Web** tab. Move to the **Files** tab by clicking on "Files". Your browser should display a page not unlike this:



5 Prepare the Coach's webapp code for deployment.

You have a copy of the Coach's webapp (code, templates, CSS, et al) in your `webapp` folder on your computer. Use your operating system's compression technology to ZIP the entire contents of your `webapp` folder into a file called `webapp.zip`.

6 **Use the "Upload a file" button to copy `webapp.zip` to the cloud.**

WATCH IT!

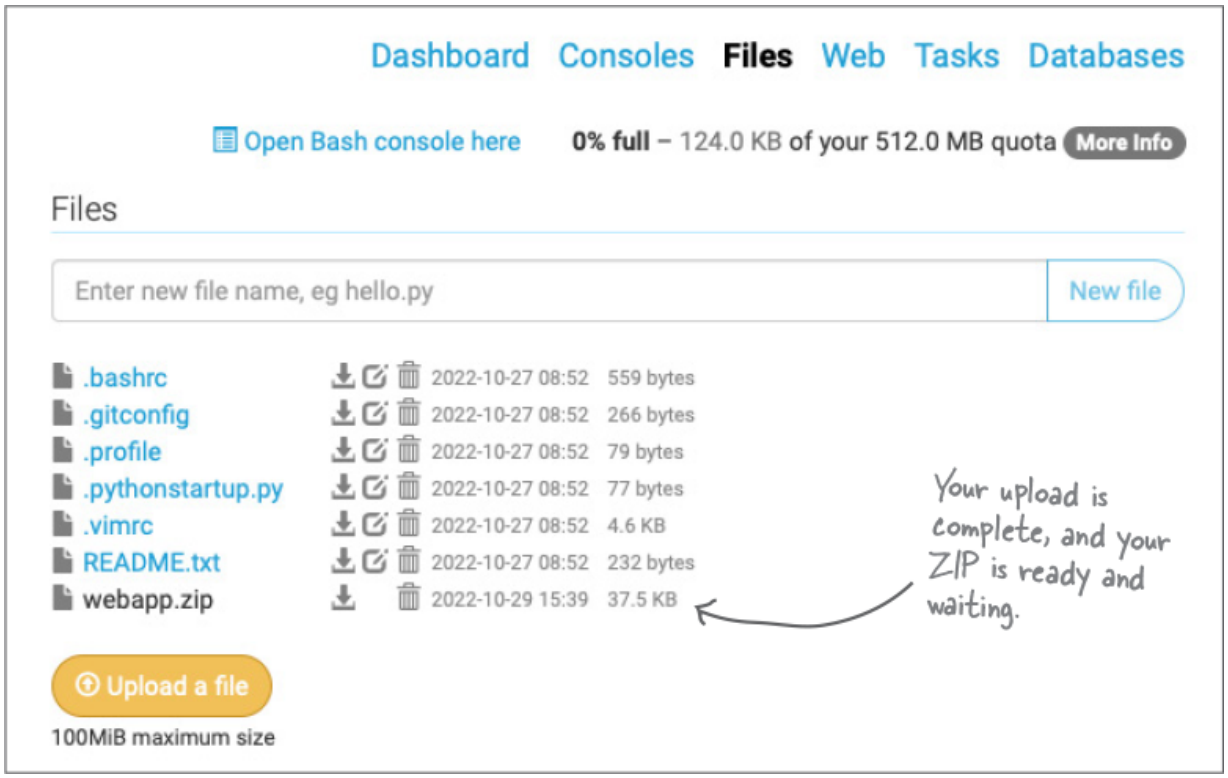


Care is needed when first uploading your code to PythonAnywhere.

Don't be tempted to upload your webapp's files individually, which is possible, but prone to error (because sure-as-shootin', you'll forget something). ZIP is your friend.

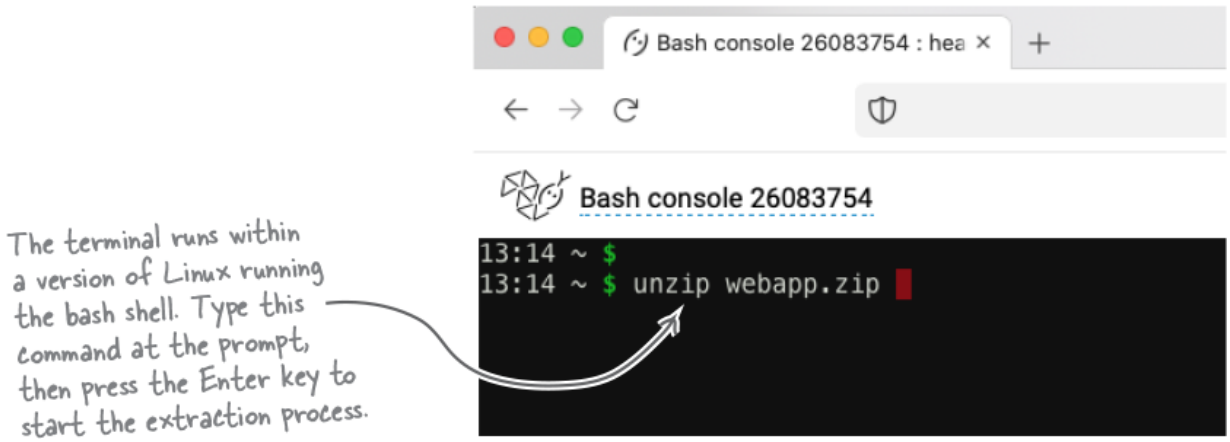
Extract your code in the console

With your code uploaded, you can open a terminal on the PythonAnywhere cloud to *unzip* `webapp.zip`:



7 Unzip your just-uploaded ZIP file.

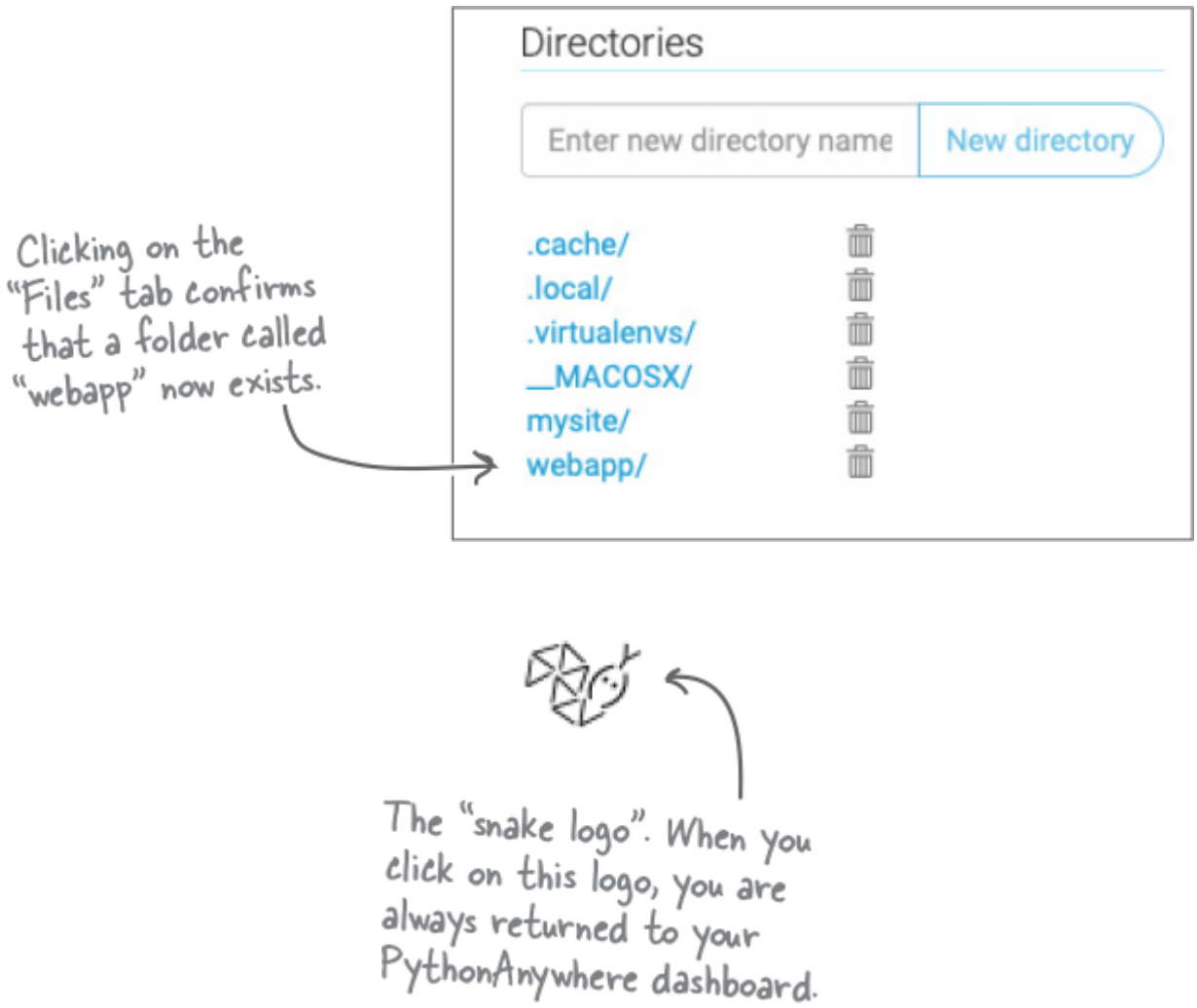
Do you see that “Open Bash console here” link underneath the word “Dashboard”? Go ahead and click on that link to open a browser-based terminal running on the PythonAnywhere cloud.



Configure the Web tab to point to your code

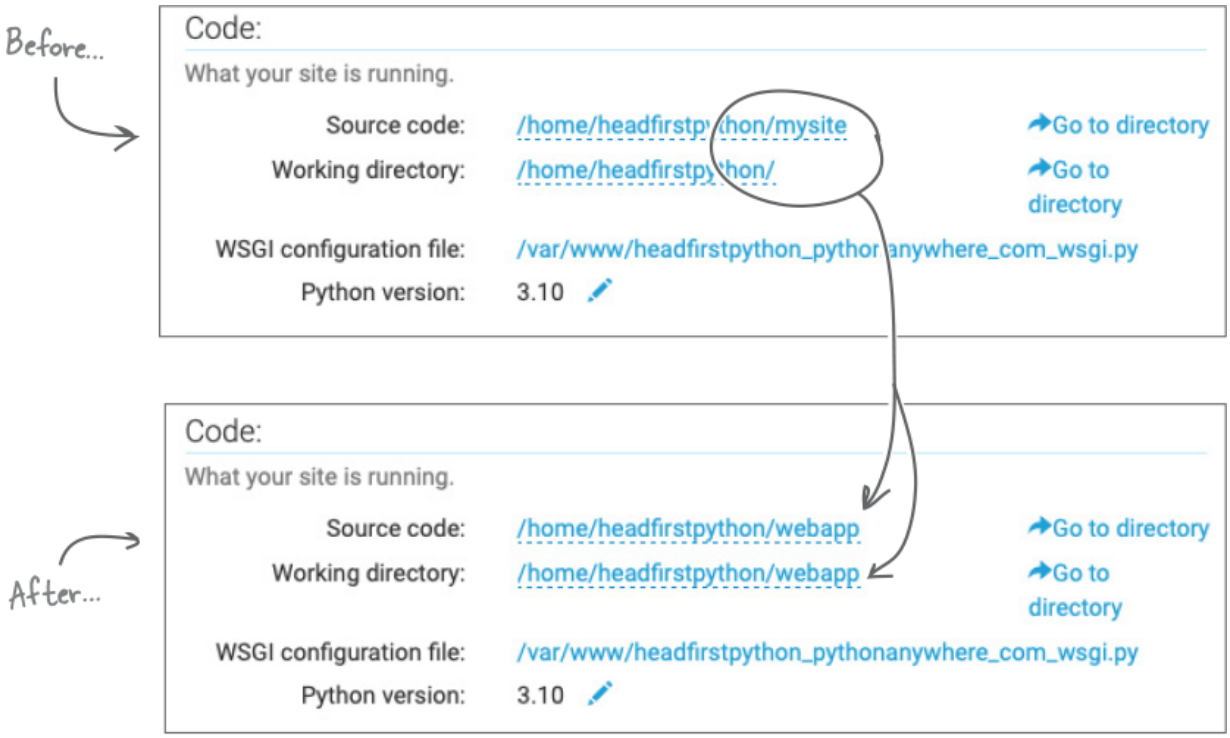
The `unzip` command (at the bottom of the last page) unpacks your code and creates a folder called `webapp` on your PythonAnywhere top-level folder.

Once the command completes, type `exit` at the console prompt to terminate the terminal, then click on the PythonAnywhere “snake logo” to return to your dashboard.



8 Configure your webapp's folders.

Click on your dashboard's **Web** tab, then scroll down until you see the **Code** settings. Click on the "Source code" and "Working directory" entries to adjust both to refer to your `webapp` folder.



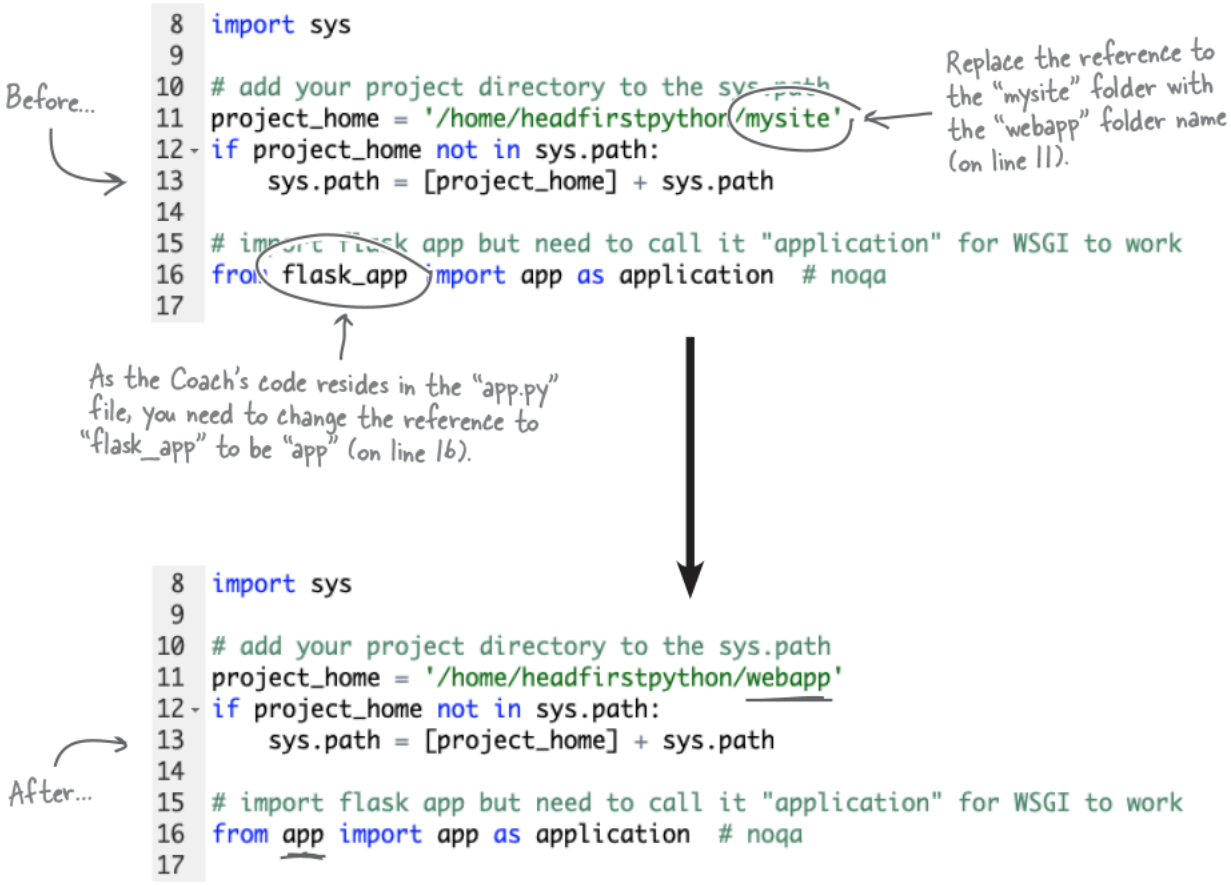
Edit your webapp's WSGI file

The third link in the **Code** section of your webapp's set-up tab refers to your webapp's WSGI configuration. WSGI is short for *Web Server Gateway Interface*.

On PythonAnywhere, the WSGI configuration file tells the server where to find your code as well as how to start it. The default webapp puts its code in the `mysite` folder and uses the `flask_app.py` file as it's starting point, and these settings are refered to in the WGSI file. Both need to change in order to run the Coach's webapp.

9 Edit the WSGI file to refer to the Coach's webapp code.

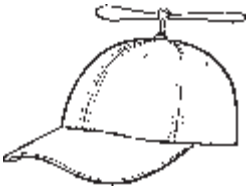
Click on WSGI link to load the configuration code into PythonAnywhere's programmer's editor. Apply the two changes as shown below.



With these two small edits applied, click on that big green **Save** button, then click on the snake logo to return to your dashboard, then return to the **Web** tab once more.



GEEK NOTE



The term “WSGI” is pronounced “whiskey” by most Pythonistas as opposed to spelling out the four individual letters. Nobody says “W…S…G…I”. The whiskey pronunciation can lead to some confusion, especially when newbies attend Python conferences or meetups for the first time. There’s many a poor soul that’s been left wondering if they’re in the right place…. as everyone in the *Web Track* seems to be talking about hard liquor. 😉

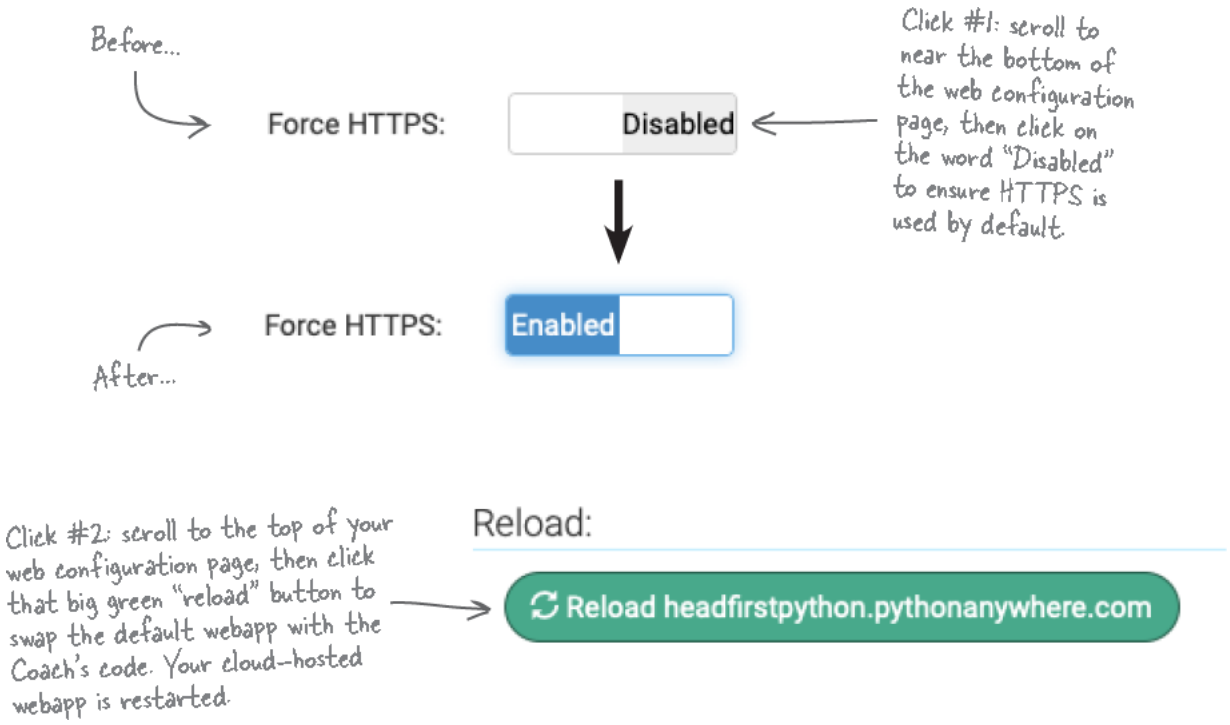
Just t wo more clicks and you’re all set…

You’re almost there. You’ve created a PythonAnywhere account, created a default Flask webapp, uploaded the Coach’s code, and recon-

figured your hosted webapp's configuration. Two quick clicks stand between you and your first test-run of your PythonAnywhere-hosted webapp.

10 Switch on HTTPS, then reload your running webapp.

The good folk at PythonAnywhere provide HTTPS support at no extra cost, so be sure to switch it on before reloading your webapp code.

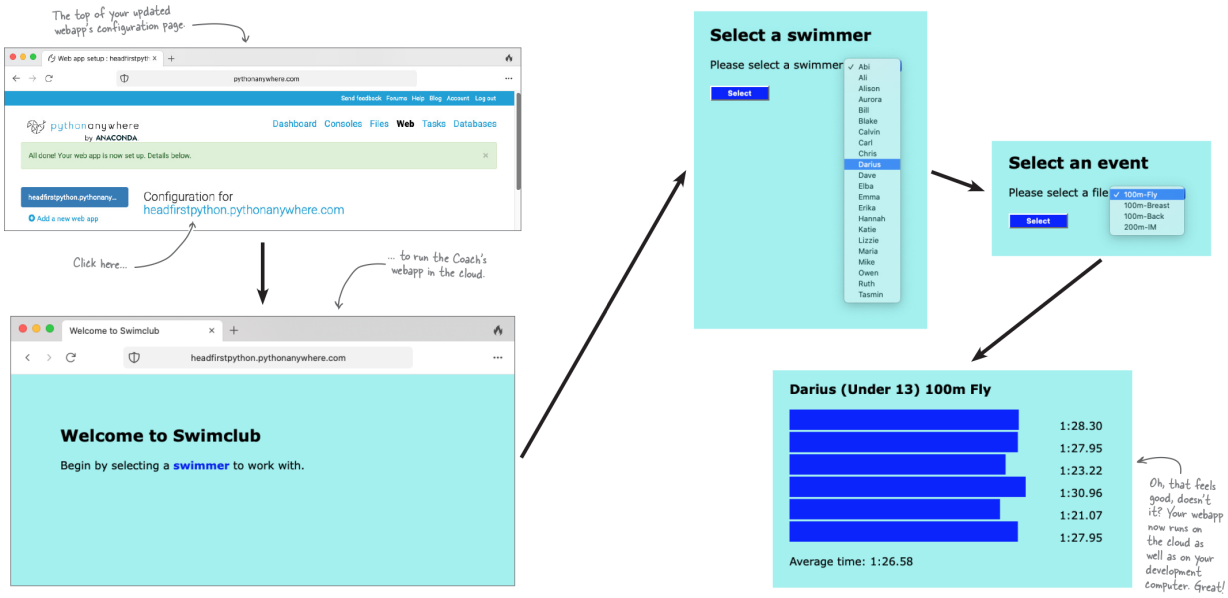


That's the 10 steps done, which depending on how quickly you read/type, may have only consumed 10 minutes. All that remains is to take your cloud-hosted version of the Coach's webapp for a spin. *Drum roll, please*… can you feel the *excitement*?

TEST DRIVE



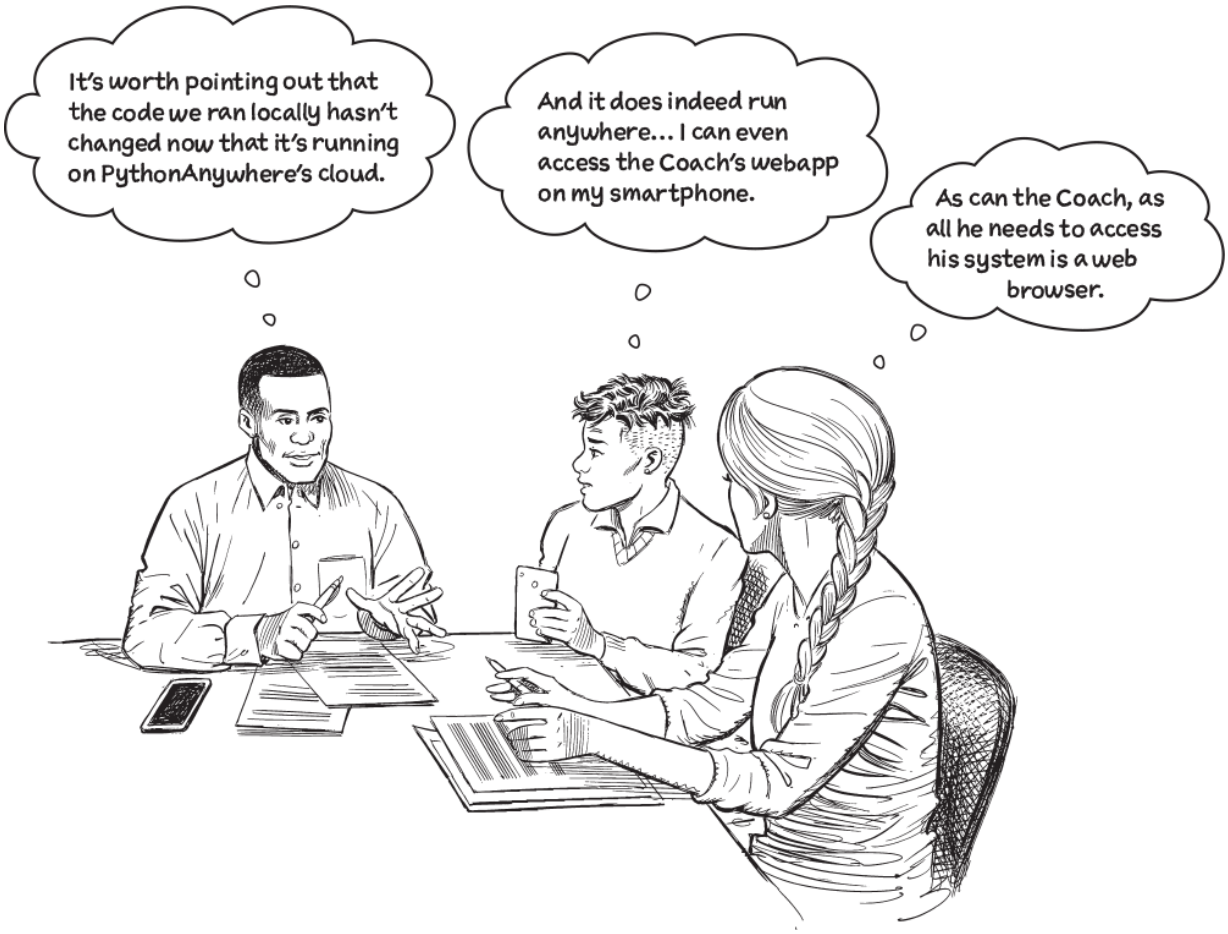
It's time to return to the top of your webapp's configuration page and click on the blue link:



Go ahead and confirm that your cloud-hosted webapp is running *exactly* as it did on your local computer:

Your cloud-hosted webapp is ready!

Before moving on, take a moment to return to PythonAnywhere, then logout from the cloud service. Your webapp continues to run, and if you provide the Coach with the web address of your running service he can access your webapp from *anywhere*.



THERE ARE NO DUMB QUESTIONS

Q: That PythonAnywhere service is cool. But, how do I know it won't disappear from the cloud at any moment?

A: To be honest, you can't ever be sure of that. However, PythonAnywhere is an Anaconda company, who are themselves a big player in the Python data science ecosystem. This means the fact that Anaconda owns PythonAnywhere is noteworthy. On top of this, PythonAnywhere is deployed on top of Amazon's AWS technology, which is (probably) the largest cloud-hosting platform in the world. Even if PythonAnywhere was to disappear (for whatever reason), recall that the Coach's webapp is a Python Flask webapp that wasn't changed in anyway to make it run on PythonAnywhere. This means you're free to take your code and deploy it elsewhere as required.

Q: What will happen if PythonAnywhere decide to drop their free-tier beginner accounts? Is the Coach on the hook to pay to keep his webapp running?

A: To their credit, PythonAnywhere have offered a free-tier since 2011, and they constantly and publicly state that free beginner accounts are here to stay. They deserve credit for this.

Q: When creating a new webapp on PythonAnywhere does it matter which version of Python I run?

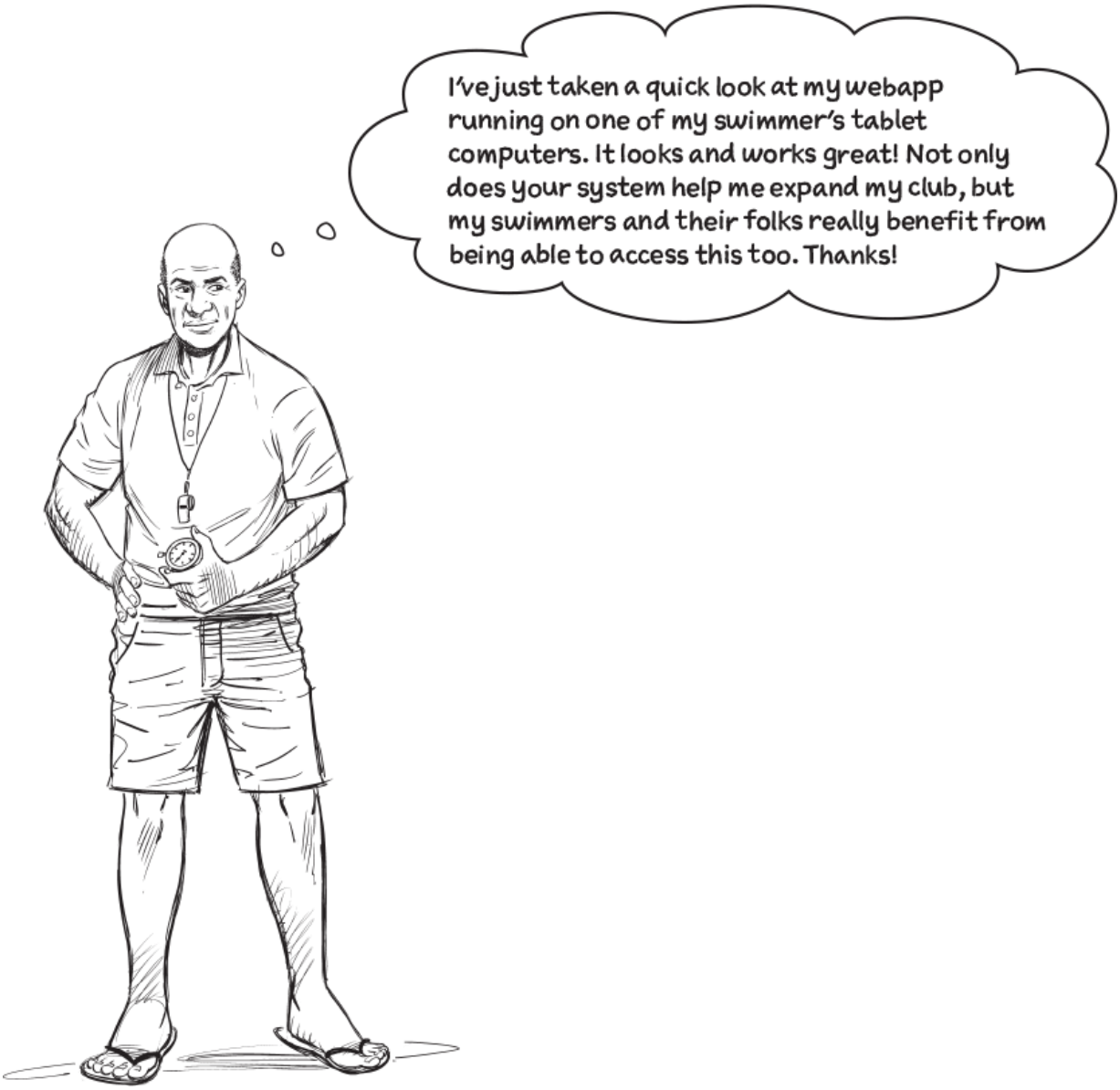
A: Rarely. The code used with the Coach's webapp happily runs on Python 3.6 and above (due to it's use of f-stings). We always opt for the most-recent Python on PythonAnywhere when asked to choose. However, if you are using a language feature that requires a specific release of Python (such as the `match` statement added in Python 3.10), then more care is needed when chooing which Python to use. (As an aside: it is possible to adjust which version of Python your webapp runs on. There's a drop-down list in the Code section of the Web tab).

Q: What am I not getting when I use a Beginner account on PythonAnywhere?

A: There are some restrictions but, in our mind, none of these are show-stoppers. Check out the comparison table at the bottom of this page: https://www.pythonanywhere.com/pricing/. Consider upgrading to a paid account if there's a feature you need which isn't provided for free.

CHAPTER REVIEW BULLET POINTS

- Add a **stylesheet** to your webapp to add a splash of color. And be sure to put your stylesheet in the `static` folder.
- Remember: any **changes** made to your `base` HTML template are **applied** to **all** of the templates which **inherit** from `base`.
- If your code generates HTML without reference to the `base` template, any CSS applied in the `base` (or anything else, for that matter) is **not** shared.
- It's possible to put any arbitrary Python code between the `{{` and `}}` tags within a Jinja template. Just be careful not to use too **complex** an expression.
- As in previous chapters, the use of the `removesuffix` and `split` string methods lets you **chain** methods together to get lots of work done with a small amount of code.
- Deploying your webapp to the **cloud** is possible with many different online services, but our favorite remains **PythonAnywhere**.
- Your locally running Flask code can (more times than not) be **deployed** on PythonAnywhere with next to no code changes.
- WSGI stands for *Web Server Gateway Interface*, and is a standard **specification** for embedding Python code within a web browser. Flask ships with a small WSGI-compliant test web server which exists to let you **test** your webapp locally on your own computer. PythonAnywhere supports WSGI, too, as do all the other big players in the web-hosting ecosystem.



That's a great endorsement.

A happy Coach is a happy developer.

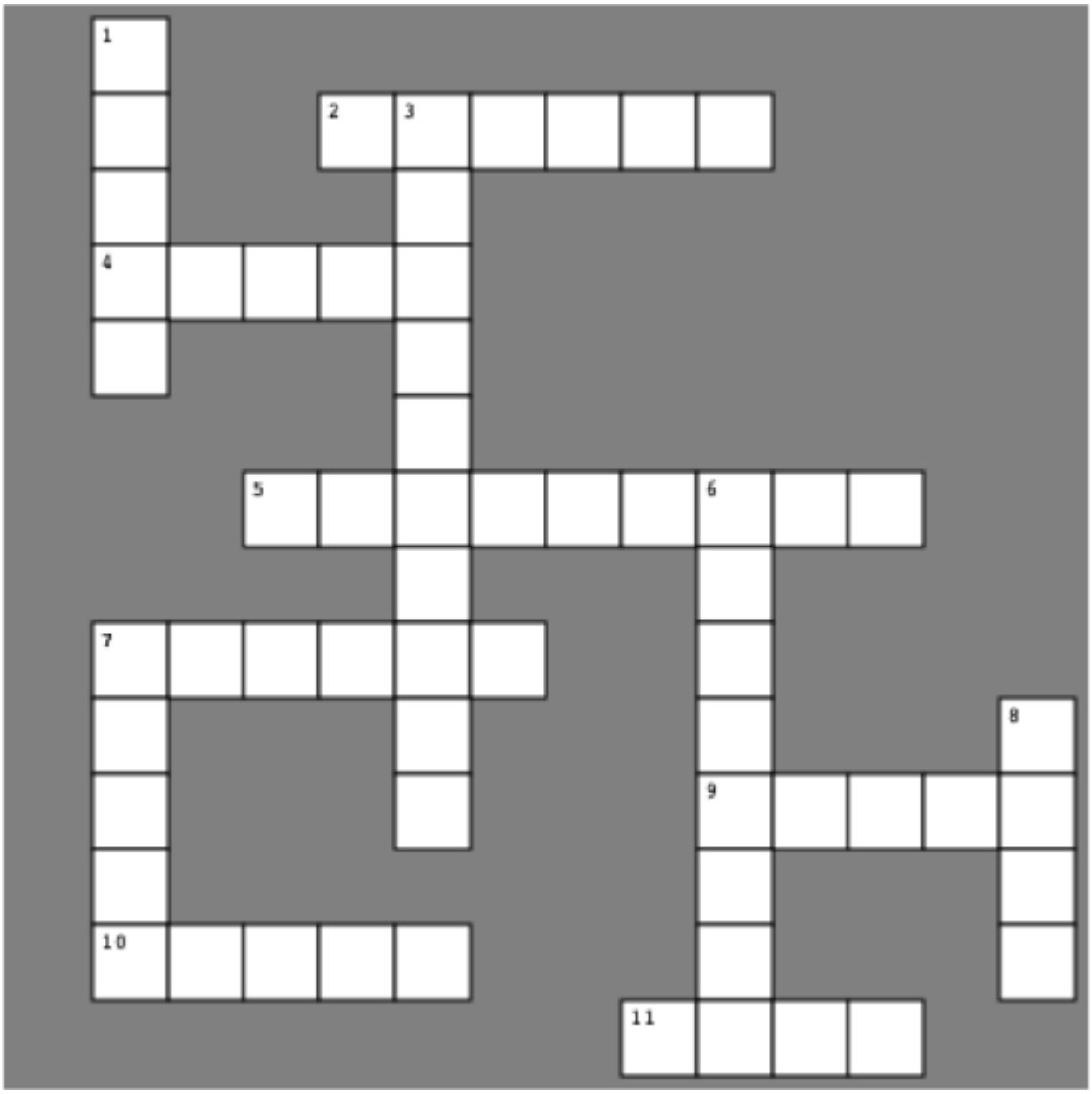
You've successfully worked your way through over half of this book's chapters and, in the process, delivered a working webapp to a very appreciative Coach (not to mention learning a chunk of Python on the way). This is great going. Feel free to pat yourself on the back.

With a cloud-deployed version of your webapp running, it's time for a well-earned break. That's after you blast your way through this chapter's crossword, of course. 😉

The Deployment Crossword



As always, all the answers to the clues are found in this chapter and the solutions are on the next page.



Across

2. The name of the folder in the cloud that contains your code.

4. Use this when it's time to decompress.

5. PythonAnywhere's control center.

7. The folder for web resources that don't often change.

9. Enabled for that little extra bit of web security.

10. The middle word of CSS.

11. The button that everyone just loves to click.

Down

1. It's the web, it's also fluffy... or maybe it's cumulonimbus?

3. In Jinja templates, the Python code between `{{` and `}}` is referred to by this name.

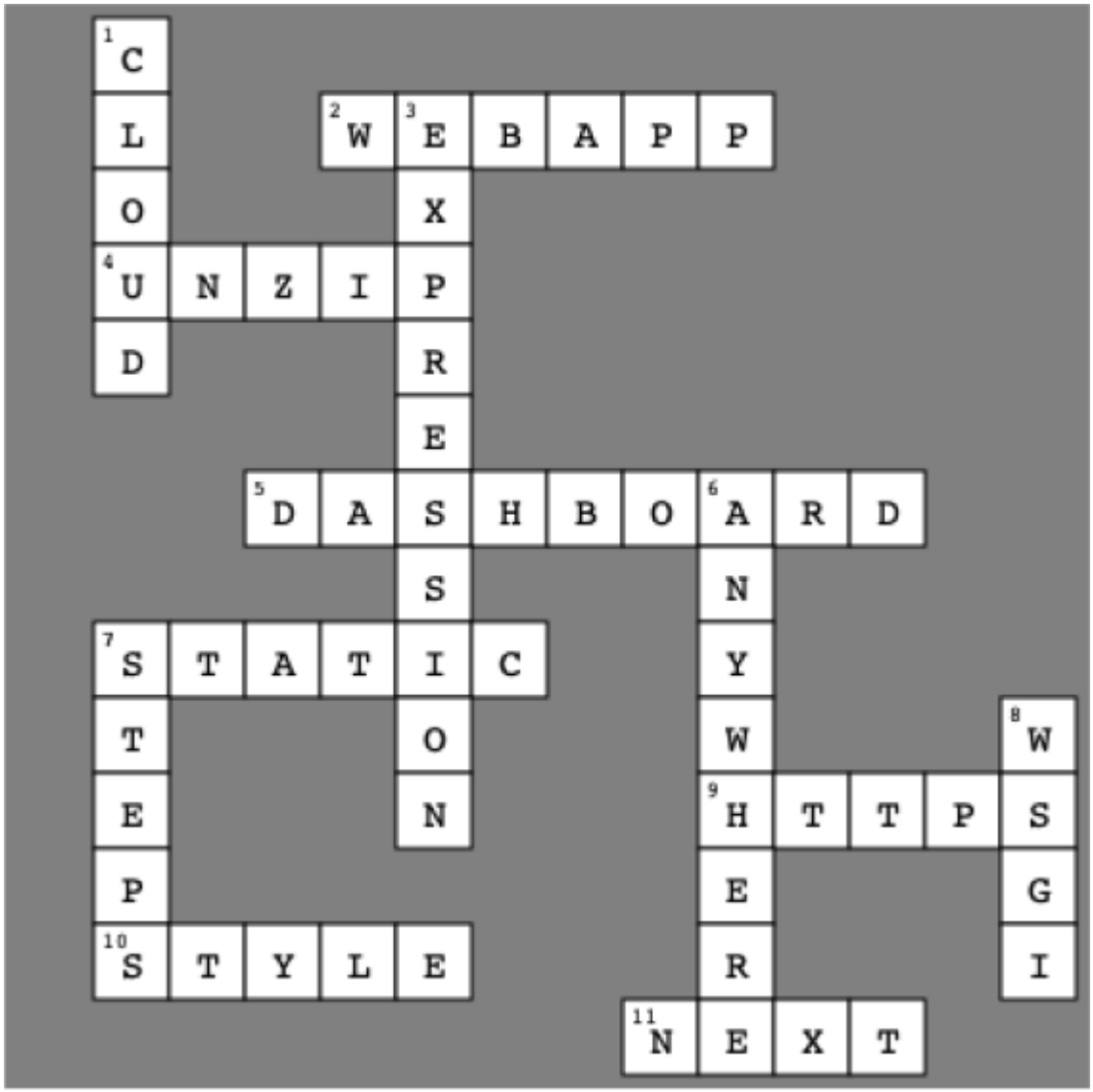
6. Python is everywhere and _____.

7. You learned about ten of these.

8. *Not* a form of hard liquor.

The Deployment Crossword Solution





Across

2. The name of the folder in the cloud that contains your code.

4. Use this when it’s time to decompress.

5. PythonAnywhere’s control center.

7. The folder for web resources that don't often change.

9. Enabled for that little extra bit of web security.

10. The middle word of CSS.

11. The button that everyone just loves to click.

Down

1. It's the web, it's also fluffy... or maybe it's cumulonimbus?

3. In Jinja templates, the Python code between `{{` and `}}` is referred to by this name.

6. Python is everywhere and _____.

7. You learned about ten of these.

8. *Not* a form of hard liquor.

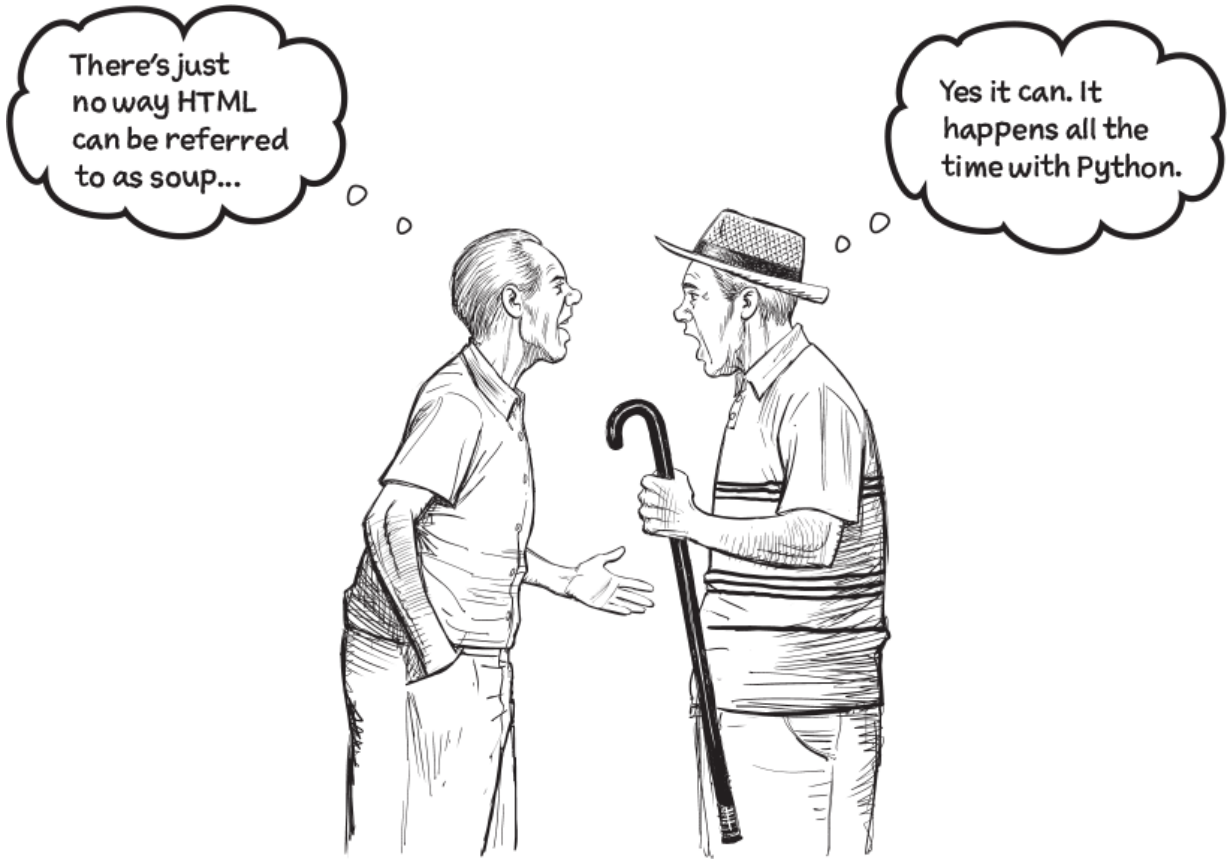
Chapter 8. Working with HTML: *Web Scraping*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 8 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



In a perfect world, it would be easy to get your hands on all the data you need.

Alas, this is rarely true. Case in point: data which is published on the web. Data embedded in HTML is designed to be **rendered** by web browsers and **read** by humans. But what if you need to **process** HTML-embedded data with code? Are you out of luck? Well, as luck would have it, Python is somewhat of a star when it comes to **scraping** data from web pages, and in this chapter you'll learn how to do just that. You'll also learn how to **parse** those scraped HTML pages to

extract usable data. Along the way, you'll meet **slices** and **soup**. But, don't worry, this is still *Head First Python*, not *Head First Cooking*…

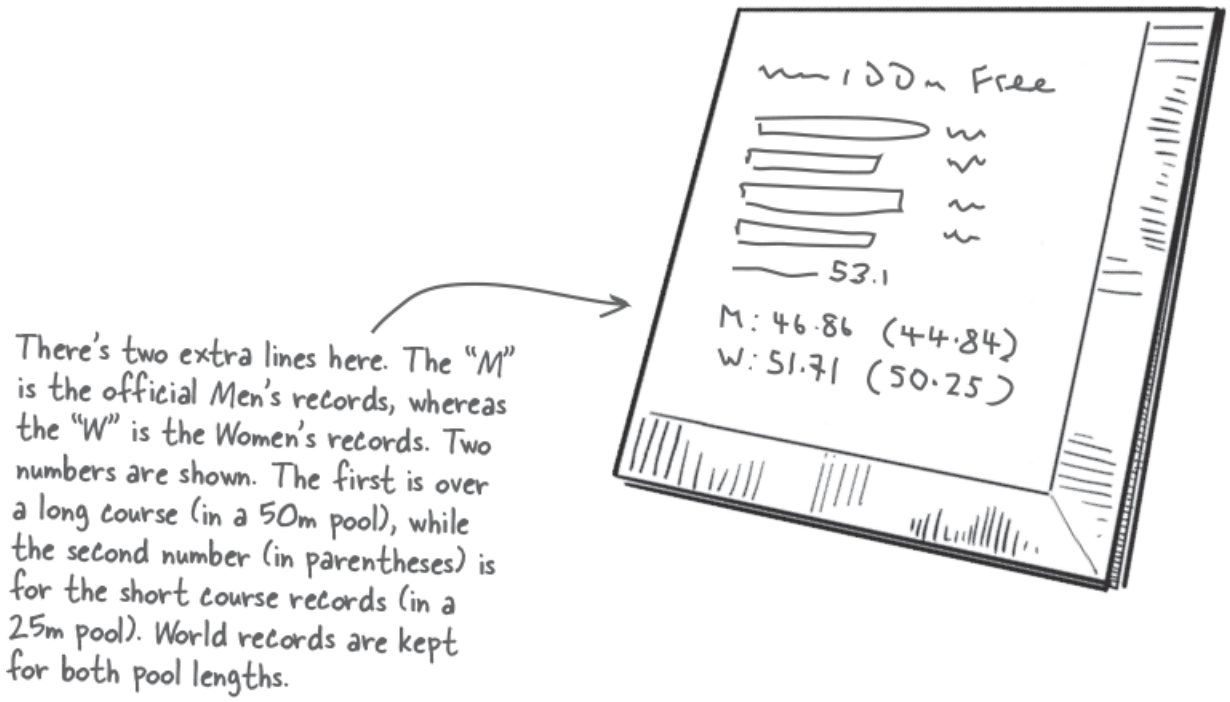
The Coach has been in touch…



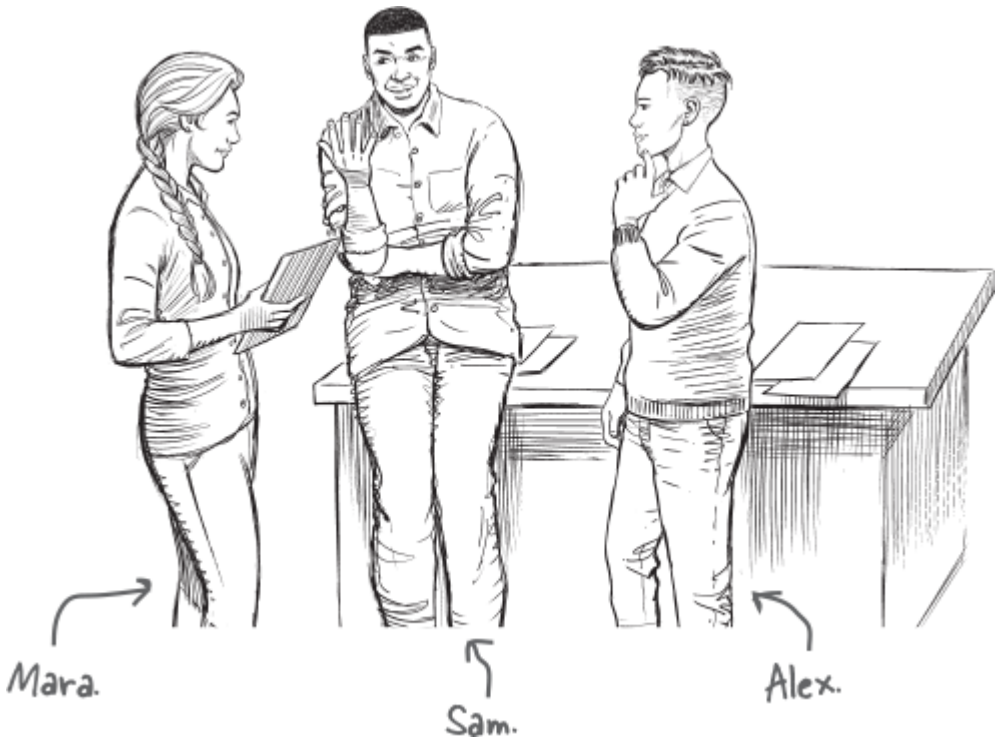
There's no harm in asking.

You've sat down with the Coach over coffee and he's explained what he wants. In addition to the current bar chart, the Coach wants to see the current world records for both men and women, for both course lengths, for any selected distance and stroke. The Coach is convinced that sharing the world record times with his swimmers gives them "something to aim for".

The Coach even sketched out his idea on the back of a paper napkin.



Cubicle Conversation



Alex: Can't we just show one number at the bottom instead of four?

Mara: To do so we'd need to know the gender of the swimmer as well as the length of the pool they achieved their time in…

Sam: And we don't know either of these data.

Alex: So… we need to show all four. OK. I get that. But, why does the Coach need this information?

Mara: It's to do with swimmer motivation. When the Coach shows the bar chart to each swimmer, he likes to refer to the current world record times to give the swimmer an idea of how that are progress-ing, as well as highlighting what's possible.

Sam: The Coach is nothing if not ambitious for his swimmers.

Alex: Wow. The Coach has big plans, eh?

Mara and **Sam** (together): All coaches have big plans!!

Sam: So… where can we get our hands on this world records data?

Mara: Although the Coach tells me the FINA website is the ultimate word on this, a copy of the data is also on Wikipedia.

Alex: FINA?

Sam: It's the *Fédération Internationale de Natation*, which translates in English to the *International Swimming Federation*. Aren't you glad you asked?

Alex: And why not get the data from them?

Mara: The data is available on FINA's website, but Wikipedia does a better job of providing it in tabular form.

Alex: Which is important because…

Sam: Because it's easier to scrap.

Mara: It sounds like you have a plan, Sam?

Sam: Yes. Let's scrape some Wikipedia pages.

Get to know your data before scraping

Like with lots of things, *Wikipedia* has most of the data you need, and its's waiting patiently for you to get down'n'dirty with it using web scraping:

https://en.wikipedia.org/wiki/List_of_world_records_in_swimming

As you peruse the entire *Wikipedia* page in your web browser, here's a portion of the page that's of particular interest to us:

This is a chunk of the 50m pool (long course) records for Women, as of late 2022.

Women [edit]

Event	Time		Name	Nationality	Date	Meet	Location	Ref
50m freestyle	23.67	sf	Sarah Sjöström	Sweden	29 July 2017	World Championships	Budapest, Hungary	[56][57]
100m freestyle	51.71	r	Sarah Sjöström	Sweden	23 July 2017	World Championships	Budapest, Hungary	[58][59]
200m freestyle	1:52.98	ss	Federica Pellegrini	Italy	29 July 2009	World Championships	Rome, Italy	[60][61][62]
400m freestyle	3:56.40		Ariarne Titmus	Australia	22 May 2022	Australian Championships	Adelaide, Australia	[63]
800m freestyle	8:04.79		Katie Ledecky	United States	12 August 2016	Olympic Games	Rio de Janeiro, Brazil	[64][65]
1500m freestyle	15:20.48		Katie Ledecky	United States	16 May 2018	TYR Pro Swim Series	Indianapolis, United States	[66][67]
50m backstroke	26.98		Liu Xiang	China	21 August 2018	Asian Games	Jakarta, Indonesia	[68][69]
100m backstroke	57.45		Kaylee McKeown	Australia	13 June 2021	Australian Olympic Trials	Adelaide, Australia	[70]
200m backstroke	2:03.35	sf	Regan Smith	United States	26 July 2019	World Championships	Gwangju, South Korea	[71][72]
50m breaststroke	29.30	sf	Benedetta Pilato	Italy	22 May 2021	European Championships	Budapest, Hungary	[73][74]

These two columns of data are of most use to the Coach. The other columns are interesting, but the Coach made no mention of them on his napkin, so let's ignore them for now.

BRAIN POWER



Can you think of a strategy you might employ in an attempt to identify which of the tables you need to scrape?

Before moving on, take as much time as you need to review the *Swimming World Records* web page on *Wikipedia*. Specifically, note how many **tables** of data appear on the page.

There are the four for the gender-specific world records: the long course tables for men and women, and the short course tables for men and women.

There are another two tables for the long and short course mixed relays, then three more “rankings” tables near the bottom of the page. There’s a lot of data, that’s for sure.

We need a plan of action…



No manual transcription is needed here.

Instead, you're going to write Python code which grabs the entire page from *Wikipedia*, then processes the HTML to identify the tables

which contain the data you need. You'll then process the values in each of those first two columns to build a dictionary of events and their associated world record times.

With that dictionary built, you can then use it as a *lookup table* to add the world record times to any produced bar chart.

NOTE

A lookup table. Is there a more perfect use-case for Python's dictionary?

A step-by-step guide to web scraping

The *Head First Coders* sent over a plan of attack for getting this latest chunk of work done. Here's our list of tasks:

1. **Grab the raw HTML page from Wikipedia.**

2. **Identify the tables that contain the data you need.**

3. **Process the first two columns of each table to extract the record data.**

4. **Convert the record data into a dictionary of swimming world records.**

5 **Use the dictionary to add world records to the bottom of each bar chart.**



There is a bit of work to be done.

Each task likely has sub-tasks, which might take a while. But, we're up for the challenge if you are?

In fact, we are going to split these five tasks over this and the next chapter. But, don't worry, we are taking our time to ensure we give each task the time it needs, as – like with all coding projects – the devil is in the details.



Let's take the Coach's advice and go with a three/two split

NOTE

That's not a swimming metaphor, is it?!?

The recommendation is to split the five tasks so that the first unit of work concentrates on processing the HTML page to extract useful data. The second unit of work then takes this data and transforms it into a format that can then be integrated into the Coach's webapp.

With that in mind, concentrate on these three tasks for this chapter:

1. **Grab the raw HTML page from Wikipedia.**
2. **Identify the tables that contain the data you need.**
3. **Process the first two columns of each table to extract the record data.**

The next chapter, called *Data Manipulation*, continues by working through the remaining two tasks which tackle the integration:

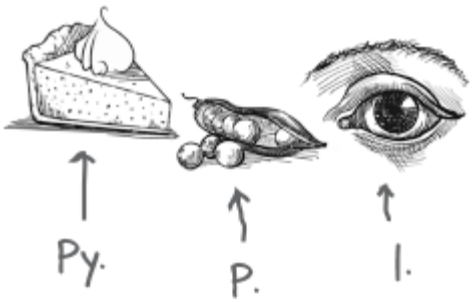
4. **Convert the record data into a dictionary of swimming world records.**
5. **Use the dictionary to add world records to the bottom of each bar chart.**

With your plan of attack finalized, let's get going.

It's time for some HTML-parsing technology

As with most other technologies in the Python space, there is more than one way to grab data off the Internet.

The classic technique involves using the most-downloaded third-party Python package from PyPI, called `requests`, which is often twinned with another classic HTML-parsing technology, called *Beautiful Soup* (and known the world-over as `bs4`). To parse HTML effectively, `bs4` needs to employ the services of a third-party parsing engine, with `lxml` a popular choice (although there are others).



If you've been paying attention (and counting) we're up to three potential package installs from PyPI (`requests`, `bs4`, and `lxml`). Although these three PyPI packages have excellent (and well-de-

served) reputations, it would be nicer if a single package could do enough to handle all of our needs here.

It's time for some… em… eh… cold soup!

The `gazpacho` package (also on PyPI) is designed to make web scraping as simple as possible. It may not be as powerful as the `requests` / `bs4` / `lxml` combo, but that's OK, as `gazpacho` does more than enough for what's needed here. Let's start by installing `gazpacho`, then use it to grab the raw HTML page from *Wikipedia*.

NOTE

It sounds like it shouldn't be allowed, but "gazpacho" is the name given to a tasty Spanish soup, which is served *cold* !!!



That's a tasty question…

Using the word "soup" to refer to HTML-parsing technologies may strike you as *odd*.

This is `gazpacho`'s way of paying homage to the grandparent of all Python HTML-processing libraries: *Beautiful Soup*. When used with `requests`, *Beautiful Soup* is a potent tool, but its use here is – in our view – overkill. As you can tell from the name, *Beautiful Soup* es-

tablished the practice of referring to parsed HTML as “soup”, and it’s a convention that’s stuck.

EXERCISE



Let’s get things going by installing `gazpacho` into your Jupyter Notebook set-up within VS Code.

Begin by creating a new notebook called `WorldRecords.ipynb`. As this notebook is in support of the Coach’s webapp, lets save it into the `webapp` folder within your `Learning` folder.

Near the start of Chapter 6, *Building a Webapp*, you used `pip` to install Flask. Use `pip` again to install the `gazpacho` package from within the first cell of your latest notebook. Write in the command you used here:

THERE ARE NO DUMB QUESTIONS

Q: Should we not stick with the tried and tested `requests` + `bs4` + `lxml` combination? After all, and based on some quick *DuckDuckGo* searches, it looks like they are a popular choice, so why are we taking a different route?

A: That's a valid question. Ask most anybody to recommend a HTML parsing library and `bs4` comes up a lot. So does *Scrapy* which is an open source framework built to create "web spiders" capable of advanced web scraping. Both `bs4` and *Scrapy* work well, but we like `gazpacho` because it's so easy to get started with and is easy to use. We also like the fact that `gazpacho` has no dependencies, which means it can be added to your Python projects without you having to worry about extra dependent modules being added. When you decide to use `bs4`, you get `requests` and `lxml` too, whether or not you actually wanted them.

Q: It there not an API that can give us the swimming world records data as needed?

A: Not that we could find, although FINA's site does provide a way to download spreadsheets for various world records. However, the UI is designed for human point'n'click interactions, not computer generated API calls.

Q: What about using something like *Selenium* for this type of work?

A: *Selenium* is a great tool, but it's designed to help with the automated testing of web sites/applications. You could probably make *Selenium* perform web scraping, but it might end up being more trouble than it's worth, as that's not *Selenuim's* primary use case. It's an thought-provoking question, though…

Q: And what about `wget` and `curl` on the command-line? Why not run those then parse their output?

A: That might work too, but – as you're about to see – `gazpacho` is purpose-built to perform web scraping via HTML parsing with the minimal amount of fuss.

EXERCISE SOLUTION



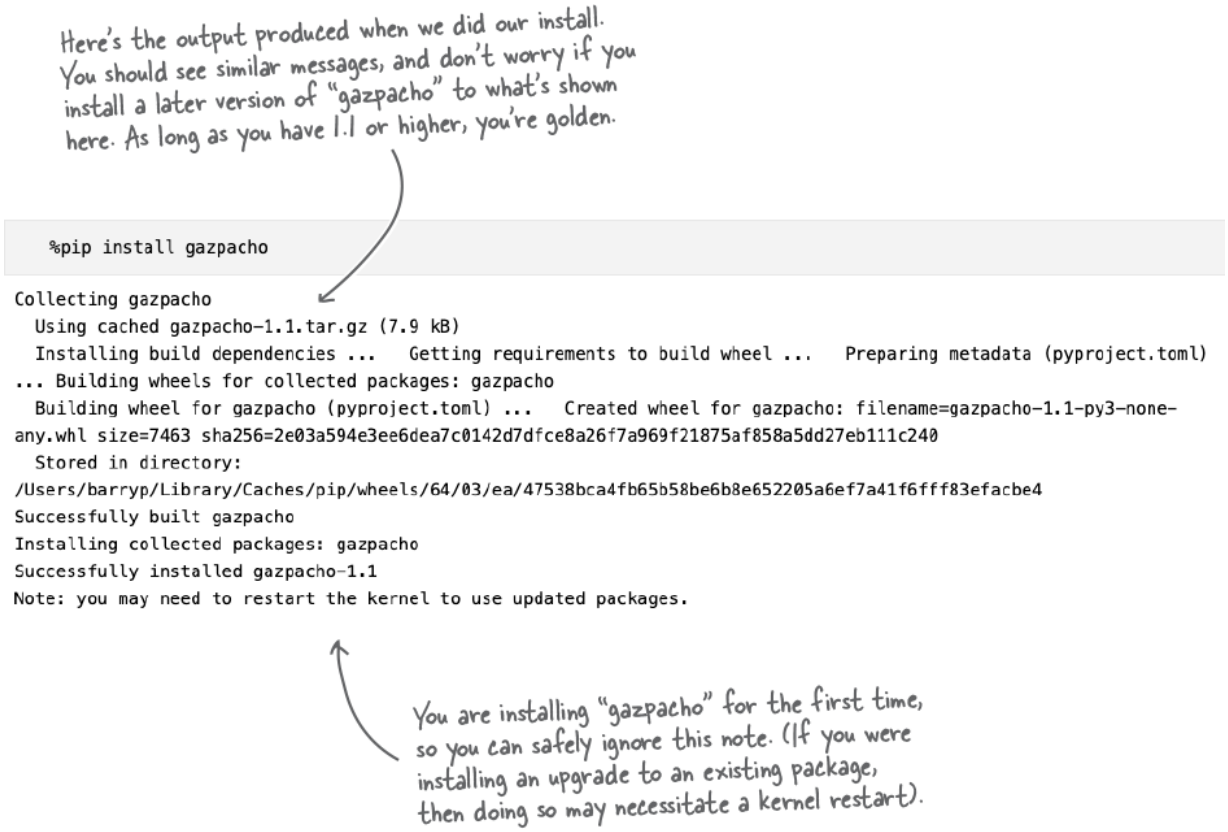
You were to get things going by installing `gazpacho` into your Jupyter Notebook set-up within VS Code.

You were asked to create a new notebook called `WorldRecords.ipynb` saving it into the `webapp` folder within your `Learning` folder.

You were then asked to use `pip` to install the `gazpacho` package from within the first cell of your latest notebook. You were to write in the command you used here:

```
%pip install gazpacho
```

Did you remember to prefix the "pip" command with the % symbol? This is what enables your notebook to run "pip" in this instance.



Grab the raw HTML page from Wikipedia

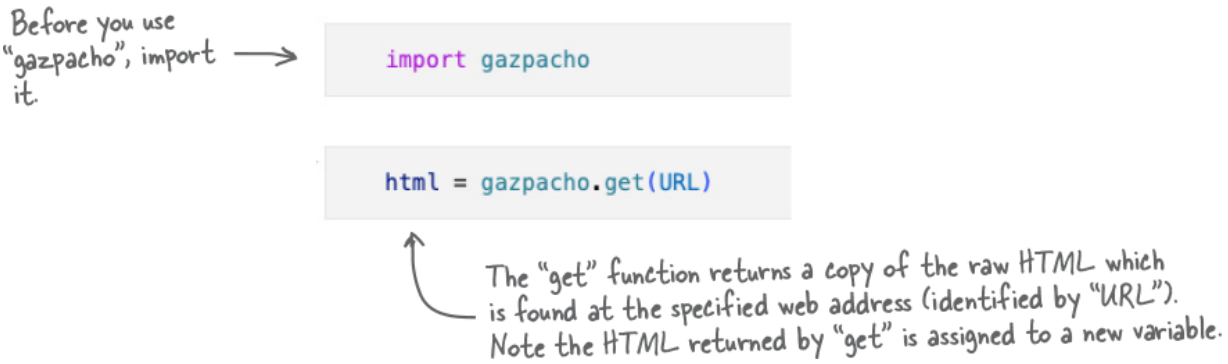
With `gazpacho` installed, let's put it immediately to work.

Begin by assigning the web address of the *Wikipedia* page (you want to scrape from) to a constant variable called URL:

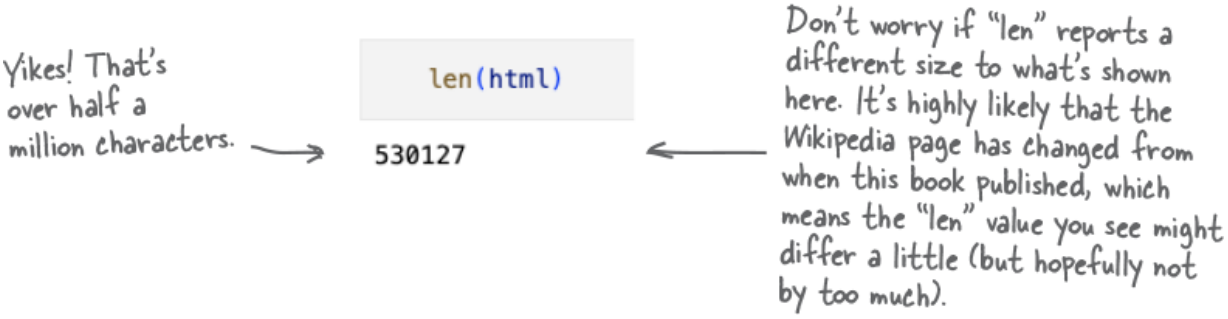
```
URL =  "https://en.wikiped.org/wiki/List_of_worl
```

The `URL` value identifies the web page containing the swimming world records.

Let's use `gazpacho` to retrieve the raw HTML from the `URL` web address, creating a new variable within which to store the results, which we've called (in a moment of madness) `html`:



As the data returned from the `get` function is a string, let's get an idea of what we're working with by asking the **len** BIF to report how big `html` is:



That's Task #1 of your plan done and dusted, and you haven't even broken a sweat.

Get to know your scraped data

With the first task of this chapter's unit of work complete, let's apply a check-mark beside that task:

1. **Grab the raw HTML page from Wikipedia.** ✔

Before moving on, let's take a look at the text which Task #1 has popped into the `html` variable. Thanks to your use of the **len** BIF, you know that there are over half-a-million characters in `html`. That's a *lot* of text (aka a *really big* string).

You could type the name `html` into an empty notebook cell then press **Shift+Enter** to view the entire string, but *let's not do that now* as the output quickly scrolls off your screen and takes an age to display…

Instead, let's look at a *portion* of the `html` string, perhaps the first few hundred characters, or the last few hundred characters, or maybe a few hundred characters taken from the middle?

The start of the "html" string.

```
'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n<head>\n<meta
charset="UTF-8"/>\n<title>List of world records in swimming -
Wikipedia</title>\n<script>document.documentElement.className="client-js";RLCONF=
{"wgBreakFrames":false,"wgSeparatorTransformTable":["",""],"wgDigitTransformTable":
["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":
```

The end of the "html" string.

```
02T16:51:14Z","image":"https:\\/\\/upload.wikimedia.org\\/wikipedia\\/commons\\/a\\/ae
\\/Caeleb_Dressel_before_winning_100_fly_%2842769914221%29.jpg","headline":"Wikimedia
list article"}</script>\n<script>(RLQ=window.RLQ||[]).push(function()
{mw.config.set({"wgBackendResponseTime":101,"wgHostname":"mw1431"});});
</script>\n</body>\n</html>'
```

Somewhere in the middle of the "html" string.

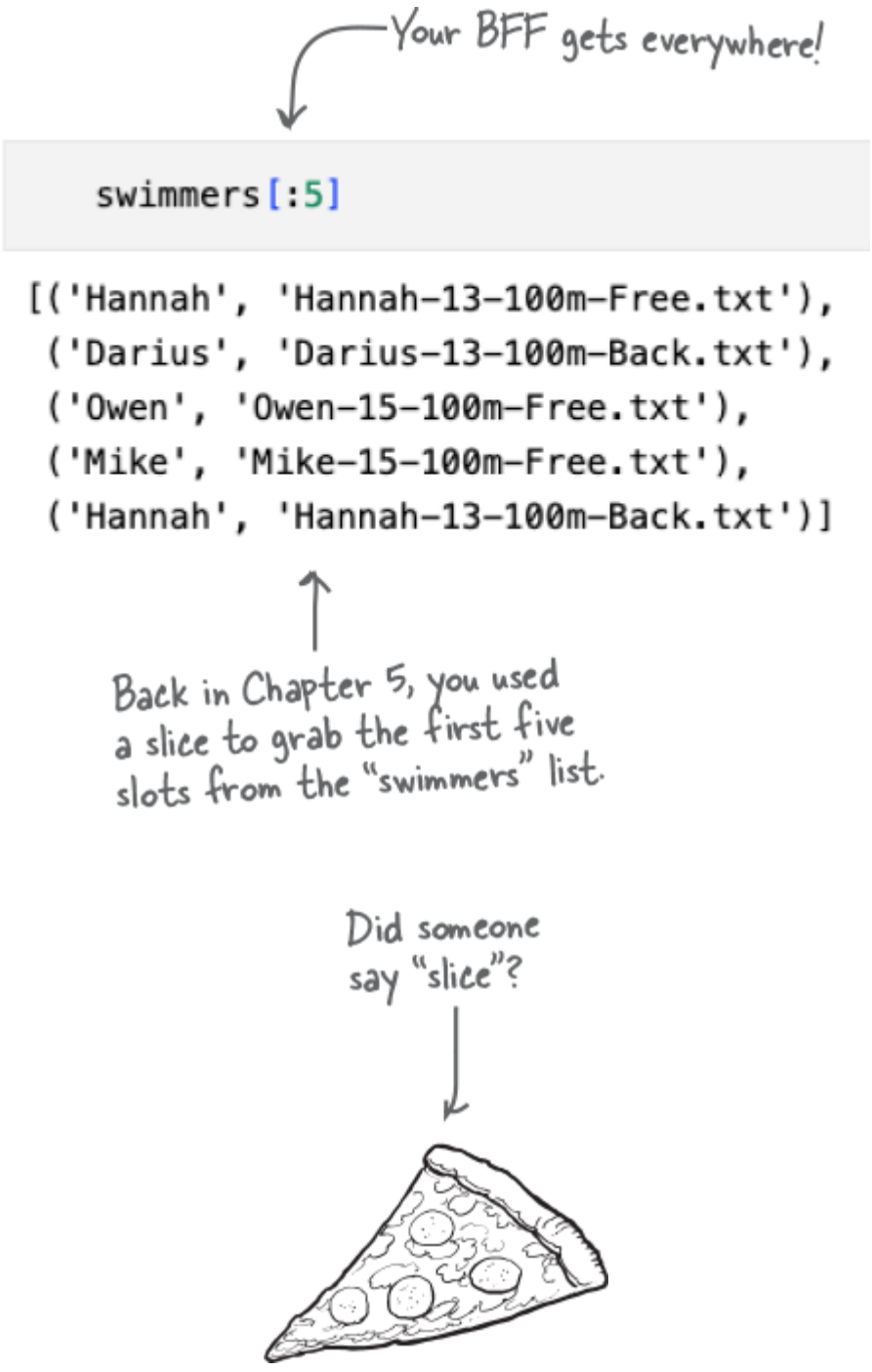
```
<tr>\n<th>Event\n</th>\n<th style="width:4em" class="unsortable">Time\n</th>\n<th
class="unsortable">\n</th>\n<th>Name</th>\n<th>Nationality</th>\n<th>Date</th>\n<th>Me
et</th>\n<th>Location\n</th>\n<th style="width:2em" class="unsortable">Ref\n</th>
</tr>\n\n<tr>\n<td><span data-sort-value="01  !"> <a
href="/wiki/World_record_progression_50_metres_freestyle" title="World record
```

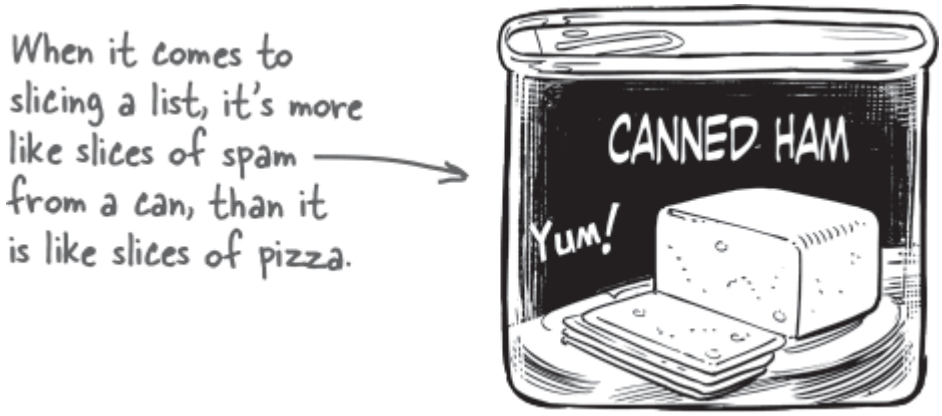


Yes. And you've already seen a slice in action.

Albeit in reference to lists, not strings.

You may recall from Chapter 5, *Getting Organized*, that you viewed the first five entries in the `swimmers` list using an extension to the square bracket notation. Let's take another look:

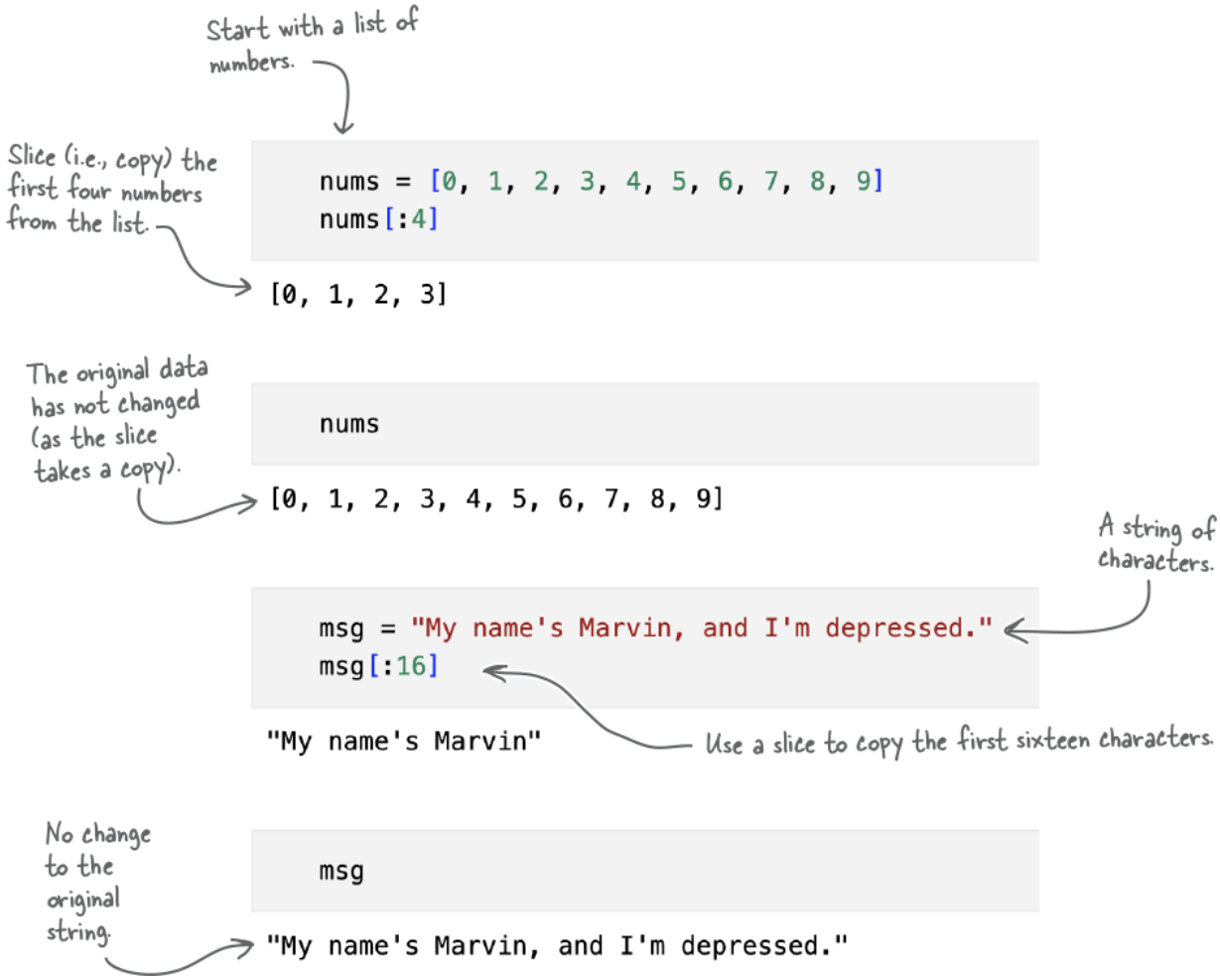




You can copy a slice from any sequence

If truth be told, we did sneak that notational nugget (shown on the previous page) into Chapter 5 without getting into what's happening behind the scenes, which was very *un-Head First* of us. It's now time to come clean, right that wrong, and take a closer look at Python's *slice notation*.

You can ask Python to *copy* a slice from any sequence. As lists and strings are *both* sequences, they support the **slice notation**. When you slice a sequence, you get back a *copy* of the sliced data. The original data remains unchanged.



Let's create another notebook, which we'll call `StartStopStep.ipynb`, to learn more about slicing. Go ahead and use VS Code to create your new notebook.

ANATOMY OF SLICES, 1 OF 3



1 Square brackets let you pick out individual values

In Python, when you use square brackets on any sequence you can pick out individual values, with the first slot numbered *zero*.

```
fav = "Life, the Universe and Everything."
```

```
fav[0], fav[1]
```

```
('L', 'i')
```

```
fav[-1], fav[-2]
```

```
('.', 'g')
```

2 Python lets you use of negative index values, too

Negative index values start counting "from the other end" of the sequence, with the last slot numbered *minus-one*.

3 Another notation extension defines a slice

A *slice* lets you extract a *sequential portion* of any sequence. You specify where the portion *starts* and *stops* within the square brackets.

```
fav[0:4]
```

```
'Life'
```

4 Start at the first number, stop before the second

The slice notation `0:4` translates as *"start at slot 0, then stop at slot 4, but don't include the data from slot 4"*. This is weird at first, but you'll get used to it.

ANATOMY OF SLICES, 2 OF 3



To save you from flipping back, here's the original string assignment.

```
fav = "Life, the Universe and Everything."
```

5 Your start and stop values can specify any slice

Here's a little trick: subtract the first *positive* number from the second *positive* number to work out how big the slice should be.

```
fav[10:18]
```

```
'Universe'
```

```
fav[23:33]
```

```
'Everything'
```

```
fav[23:-1]
```

```
'Everything'
```

6 Negative values can be used with slices, too

The slice *starts* from the first number, then *stops* (but does *not* include) the second. In this example, the slice *starts* at slot 23, then *stops* at the last slot (which is excluded from the slice).

7 The start and stop values have defaults

If you exclude the *start* value, it is assumed to be zero (the start of the sequence). Excluding the *stop* value extends the slice to the end of the current sequence. Here the *start* value defaults to zero.

```
fav[:4]
```

```
'Life'
```



```
fav = "Life, the Universe and Everything."
```

This is your favorite string, isn't it?

8 You can mix'n'match defaults with negative index values

Sometimes you don't want "`Everything.`", but you *do* want "`Everything`". You can experiment with the defaults in conjunction with negative indices until you get what you want.

```
fav[23:]
```

```
'Everything.'
```

```
fav[-11:]
```

```
'Everything.'
```

```
fav[-11:-1]
```

```
'Everything'
```

Spot the dot? You can't, 'cause it's missing.

9 Be careful you don't take things too far...

Relying on the default values for *start* and *stop* doesn't always makes sense. A case in point: although both of these final examples result in the same output, we know which of the two usages we prefer.

```
fav[:]
```

```
'Life, the Universe and Everything.'
```

Your fingers will thank you to use this one.

```
fav
```

```
'Life, the Universe and Everything.'
```



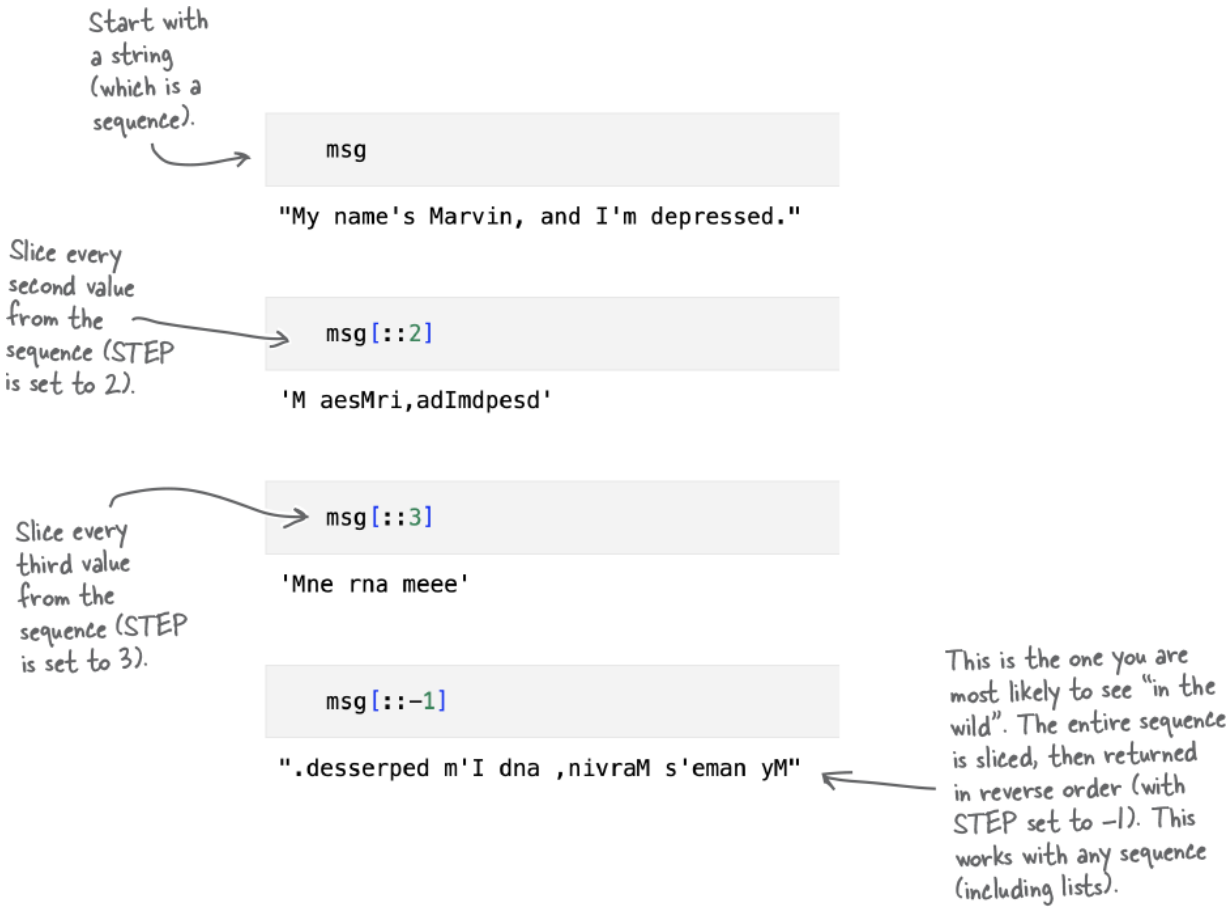
No, we didn't forget…

Although it isn't used that often, an *optional* third number can be added to your slice specification. Called *step*, it indicates the *frequency* of the slice extraction from any sequence.

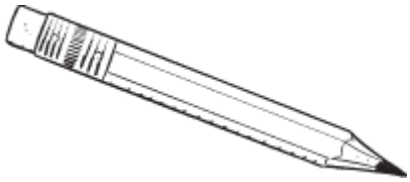
Here's the general slice form:

`[` *start* `:` *stop* `:` *step* `]`

The best way to describe *step* is with a few examples:



SHARPEN YOUR PENCIL



Now that you know a bit about slices, let's put them to use.

Start by specifying a slice which returns the first 500 characters from your `html` string.

Experiment in your `WorldRecords.ipynb` notebook, then write the slice you used into the space below:

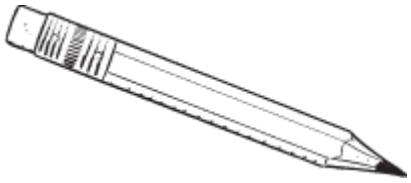
How about a slice which extracts the last 500 characters from `html`? Write in the slice you used here:

This next one might require a little bit of extra thought (and an extra line of code). Can you extract the first 500 characters of the first HTML table in the `html` string? Write in your two lines of code here:



If this one has you scratching your head, consider for a moment that "html" is a string. You can do lots of things to strings in Python. Recall, too, that the "print dir combo mambo" lists a big bunch of string methods, one of which might help. Hint: The one you're looking for should be easy to *find*.

SHARPEN YOUR PENCIL SOLUTION



You were to put your new-found knowledge about slices to good use by specifying a slice which returns the first 500 characters from your `html` string. Here's what we came up with in our `WorldRecords.ipynb` notebook:

```
html[:500]   <-- The value for "start" defaults to zero, the start of the string.
```

```
'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n<head>\n<meta
charset="UTF-8"/>\n<title>List of world records in swimming -
Wikipedia</title>\n<script>document.documentElement.className="client-js";RLCONF=
{"wgBreakFrames":false,"wgSeparatorTransformTable":["",""],"wgDigitTransformTable":
["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":
["","January","February","March","April","May","June","July","August","September","Oct
ober","November","December"],"wgRequestId":"81d5ca1b-444d-42d5-'
```

Next up was a slice which extracts the last 500 characters from `html`:

```
html[-500:]
```

The value for "stop" defaults to the end of the string.

```
'.","logo":
{"@type":"ImageObject","url":"https:\\/\\/www.wikimedia.org\\/static\\/images\\/wmf-
hor-googpub.png"}},"datePublished":"2007-03-15T21:20:10Z","dateModified":"2022-11-
02T16:51:14Z","image":"https:\\/\\/upload.wikimedia.org\\/wikipedia\\/commons\\/a\\/ae
\\/Caeleb_Dressel_before_winning_100_fly_%2842769914221%29.jpg","headline":"Wikimedia
list article"}</script>\n<script>(RLQ=window.RLQ||[]).push(function()
{mw.config.set({"wgBackendResponseTime":101,"wgHostname":"mw1431"});});
</script>\n</body>\n</html>'
```

And then, finally, the tricky one: extract the first 500 characters of the first HTML table in the `html` string:

Use the "find" string method to calculate the value to use for "start".

```
from_where = html.find("<table")
html[from_where:from_where+500]
```

Add 500 to the "start" value to specify the "stop" value.

```
'<table class="wikitable sortable" style="font-size: 95%;">\n<tbody>
<tr>\n<th>Event\n</th>\n<th style="width:4em" class="unsortable">Time\n</th>\n<th
class="unsortable">\n</th>\n<th>Name</th>\n<th>Nationality</th>\n<th>Date</th>\n<th>Me
et</th>\n<th>Location\n</th>\n<th style="width:2em" class="unsortable">Ref\n</th>
</tr>\n\n<tr>\n<td><span data-sort-value="01  !"> <a
href="/wiki/World_record_progression_50_metres_freestyle" title="World record
progression 50 metres freestyle">50m freestyle</a> </span>\n</td>\n<'
```



Yes, gazpacho can help you here.

Writing the code to “manually” parse those half-million characters is no picnic, and is likely to lead to much gnashing of teeth and maybe a tear or two. Thankfully, that was never the plan.

The plan is to apply the power of `gazpacho` to this problem.

It's time for some HTML parsing power

Let's remind ourselves of the next task in this chapter's three-point plan:



2 **Identify the tables that contain the data you need.**

Before getting to the point of identifying the tables which contain the swimming world records of interest to the Coach, let's employ `gazpacho` to extract all the HTML tables from the `html` string, then take things from there.

Before doing anything else, you need to turn the raw HTML in the `html` string into *soup*:

Passing the raw HTML string into the "Soup" constructor creates a parsed representation of the HTML, which we've assigned to a new variable called – what else? – "soup".

```
soup = gazpacho.Soup(html)
```

```
type(soup)
```

Confirmation you're now dealing with a "Soup" object.

```
gazpacho.soup.Soup
```

Try reading this line without smiling.

It's combo mambo time. Pass the `soup` object to **print dir** to get an idea of what the `gazpacho` Soup object can do:



```
print(dir(soup))
```

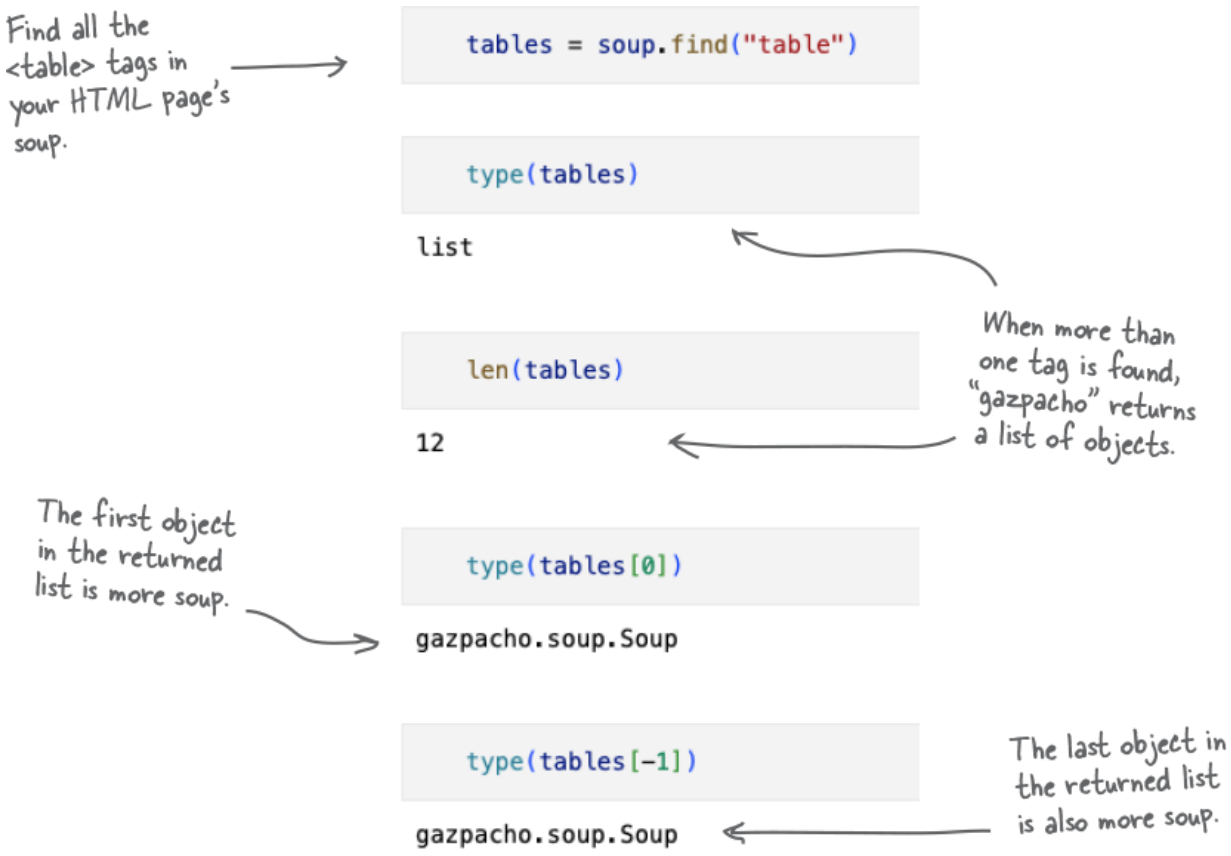
```
['attrs', 'find', 'get', 'html', 'strip', 'tag', 'text']
```

As method lists go, this might look a little skimpy, but don't be fooled. Parsing power awaits in this list of methods.

Searching your soup for tags of interest

Although the **print dir** combo mambo has revealed seven characteristics of the `gazpacho` Soup object, the **find** method is of most interest right now. The **find** method lets you locate and extract HTML elements from your soup.

Let's see what the **find** method can do with your soup…



The gazpacho defaults can sometimes trip you up.

When you use the **find** method, you need to be careful, especially if any subsequent code assumes a list is always returned.

By default, **find** actually returns one of three possible objects.

If the searched-for HTML tag is not found, `None` is returned. If a single tag is found, a *single* Soup object is returned. If more than one tag is found, a *list* of Soup objects is returned.

You can override this default behavior to ensure a list is *always* returned, that is: an empty list for `None`, a one-slot list for one Soup object, and a list of Soup objects for more than one. To do this, pass the `mode="all"` argument to **find**.

The returned soup is also searchable

Let's take a look at the first table found in the HTML page, using the **find** method once more to extract a list of the table's rows. Displaying the contents of the first row of data from the first table in the page is easily matched against the rendered output.

Grab all the rows from the first table (in slot #0).

```
rows = tables[0].find("tr")
```

```
type(rows)
```

```
list
```

Another list is returned.

```
len(rows)
```

```
21
```

Display what's in the first row.

```
rows[0]
```

```
<tr>
  <th>Event
</th>
  <th style="width:4em" class="unsortable">Time
</th>
  <th class="unsortable">
</th>
  <th>Name</th>
  <th>Nationality</th>
  <th>Date</th>
  <th>Meet</th>
  <th>Location
</th>
  <th style="width:2em" class="unsortable">Ref
</th>
</tr>
```

This looks like HTML text, but it isn't. It's "gazpacho" soup.

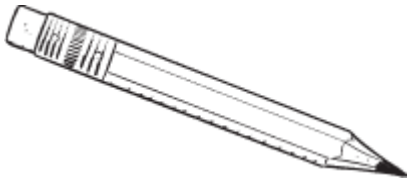
The page's first table is the data for the Men's long course records.

Long course (50m pool) [edit]

Men [edit]

	Event	Time		Name	Nationality	Date	Meet	Location	Ref
rows[0]									
rows[1]	50m freestyle	20.91	ss	César Cielo	Brazil	18 December 2009	Brazilian Championships	São Paulo, Brazil	[9][10][11][12]
rows[2]	100m freestyle	46.86		David Popovici	Romania	13 August 2022	European Championships	Rome, Italy	[13] [14]

SHARPEN YOUR PENCIL



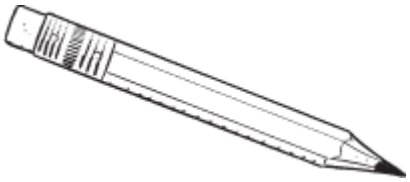
Now that you know a bit about the **find** method and how to use it, we'd like you to write a **for** loop which processes each of the tables in your HTML soup. On each enumerated iteration, calculate how many rows of data the current table has, then display the table number and the number of rows on screen (using an f-string, if possible). Work out the code you need in you `WorldRecords.ipynb` notebook, then write in the code you used here:



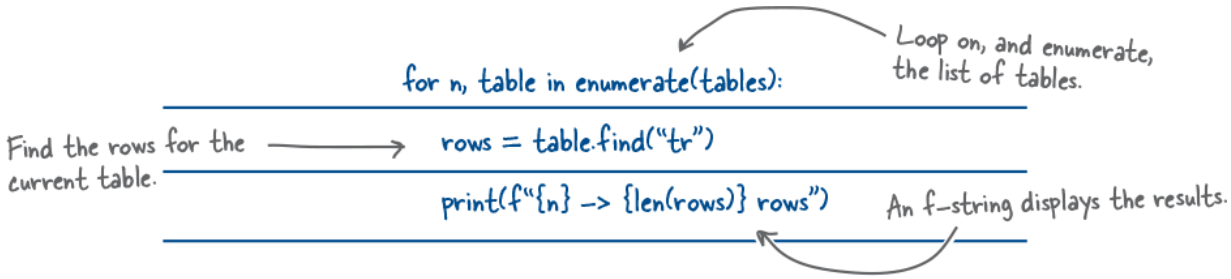
You'll find it in two places.

For starters, it's still in your `html` variable in an *unsoupified* form. Additionally, `soup.html` also has a copy of the original HTML markup (should you ever need it).

SHARPEN YOUR PENCIL SOLUTION



Now that you know a bit about the **find** method and how to use it, we asked you to write a **for** loop which processes each of the tables in your HTML soup. On each enumerated iteration, you were to calculate how many rows of data the current table has, then display the table number and the number of rows on screen (using an f-string, if possible). Here's the code we came up with. How does yours compare?



TEST DRIVE



Here's what the above code produced in our notebook:

```
for n, table in enumerate(tables):
    rows = table.find("tr")
    print(f"{n} -> {len(rows)} rows")
```

```
0 -> 21 rows
1 -> 21 rows
2 -> 3 rows
3 -> 27 rows
4 -> 30 rows
5 -> 3 rows
6 -> 26 rows
7 -> 7 rows
8 -> 7 rows
9 -> 7 rows
10 -> 5 rows
11 -> 8 rows
```

There are twelve tables on the page...

...with each table having a varying number of rows. But which of these tables contain the swimming world records data?

Which table contains the data you need?

Based on a cursory review of the output from the bottom of the previous page, it's more than likely the data you need is in the tables numbered 0, 1, 3, 4, and/or 6, as they have a large number of rows. To determine which tables to concentrate on, let's also display information on the number of columns each table has.

NOTE

Remember: the world record tables have many, many rows of data.

EXERCISE



Here is the code from the previous page:

```
for n, table in enumerate(tables):
    rows = table.find("tr")
    print(f"{n} -> {len(rows)} rows")
```

To include information on the number of rows each table contains, use a similar technique to counting the `<tr>` tags in each `<table>` tag but, instead, count the number of `<td>` tags in the *last* row of each table. Adjust the code above to display the row and column count information, then write in the updated loop code here:

BRAIN POWER



Can you think of a reason why this page's exercise suggests you count the number of `<td>` tags in the *last* row of each table as opposed to any other row, specifically, the first row?

EXERCISE SOLUTION



You were asked to include information on the number of rows each table contains, using a similar technique to counting the `<tr>` tags in each `<table>` tag but, instead, you were told to count the number of `<td>` tags in the *last* row of each table. Here's the code we *initially* came up with which (as you are about to see) causes an error:

```
for n, table in enumerate(tables):
    rows = table.find("tr")
    cols = rows[-1].find("td")  <-- Grab the <td> tags from the table's last row.
    print(f"{n} -> {len(rows)} rows, {len(cols)} columns")
```

TEST DRIVE



Our updated loop gives an error when it is executed which, if truth be told, is a total bummer:

```
for n, table in enumerate(tables):
    rows = table.find("tr")
    cols = rows[-1].find("td")
    print(f"{n} -> {len(rows)} rows, {len(cols)} columns")
```

```
0 -> 21 rows, 9 columns
1 -> 21 rows, 9 columns
2 -> 3 rows, 9 columns
3 -> 27 rows, 9 columns
4 -> 30 rows, 9 columns
5 -> 3 rows, 9 columns
6 -> 26 rows, 11 columns
7 -> 7 rows, 3 columns
8 -> 7 rows, 4 columns
```

The updated loop worked with the first nine tables, but something went wrong while processing the tenth table in slot #9.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/barryp/Desktop/THIRD/Learning/webapp/WorldRecords.ipynb Cell 22 in <cell line: 1>()
      2 rows = table.find("tr")
      3 cols = rows[-1].find("td")
----> 4 print(f"{n} -> {len(rows)} rows, {len(cols)} columns")

TypeError: object of type 'Soup' has no len()
```

The current Soup object has no length... which seems decidedly weird.

How can a Soup object have no length?

Especially if the Soup object so clearly exists? Look at what appears on screen when you ask your notebook to display the `<td>` tags contained within the *last* row of data in the tenth HTML table:



You can use your notebook to confirm you have a Soup object, as well as show that your Soup object doesn't understand how to work

with the **len** BIF:

```
type(tables[9].find("tr")[-1].find("td"))
```

```
gazpacho.soup.Soup
```

```
len(tables[9].find("tr")[-1].find("td"))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/barryp/Desktop/THIRD/Learning/webapp/WorldRecords.ipynb Cell 24 in <cell line: 1>()
----> 1 len(tables[9].find("tr")[-1].find("td"))

TypeError: object of type 'Soup' has no len()
```





Yes, sequences and collections, too.

To be pedantic about it, the **len** BIF assumes it's working with a sequence (for example, a list) or a collection (for example, a dictionary). For everything else, **len** raises a `TypeError`.

As Soup objects are *neither* sequences *nor* collections, **len** throws up its hands in despair and complains. The problem with our `<td>` processing code is that everyone's forgotten that the `gazpacho` defaults can *sometimes* trip you up.

WATCH IT!



Care is needed: gazpacho's defaults can trip you up.

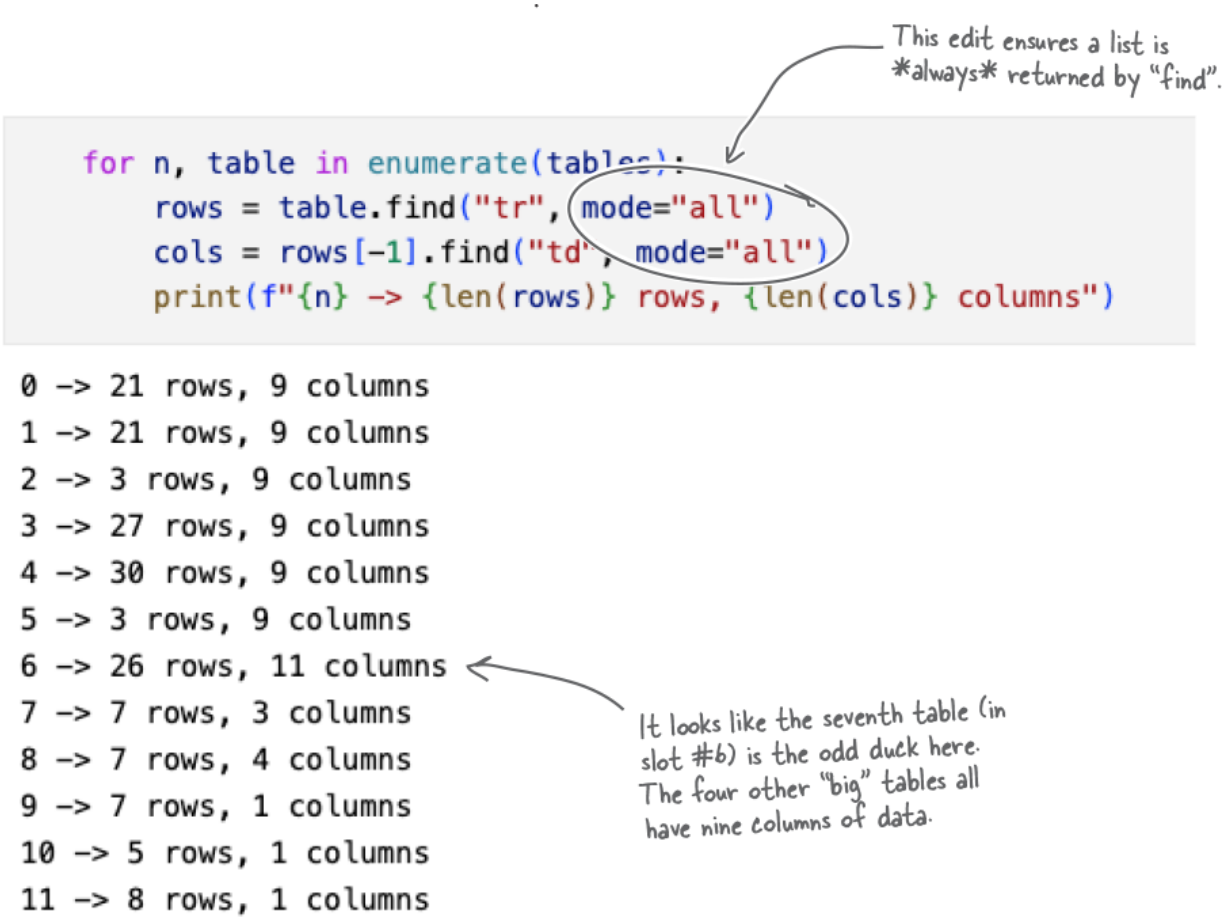
Recall that you may need to override **find***'s default behavior to ensure a list is always returned, that is: an empty list for* `None`*, a one-slot list for one Soup object, and a list of Soup objects for more than one.*

Remember how you do this: pass the `mode="all"` *argument to* **find***.*

Ask the find method to always return a list

It's not a big change to your loop code, but by asking the `gazpacho` **find** method to *always* return a list, you can guarantee your calls to the **len** BIF return zero, one, or some other positive number.

Key point: you can wave goodbye to that `TypeError`.



Take a moment to review the *Wikipedia* HTML page in your browser.

Sure enough, the swimming world record tables all have *nine* columns. The *Record holder's ranking (by nation)* table has *eleven* columns – it's the odd duck among the bigger tables on this web page. The tables in slots 0, 1, 3, and 4 are the ones you need.

2 **Identify the tables that contain the data you need.**



It's time to extract the actual data

Having identified the world records tables (in slots 0,1, 3, and 4 from the `tables` list), let's recall the remaining task for this chapter's unit of work:

3 **Process the first two columns of each table to extract the record data.**

Now that you know how to find HTML tags in your page, let's build on this technique to extract the data from the first two columns of each of the four world records tables.

Recall the “skimpy” list reported by **print dir** when applied to a Soup object:

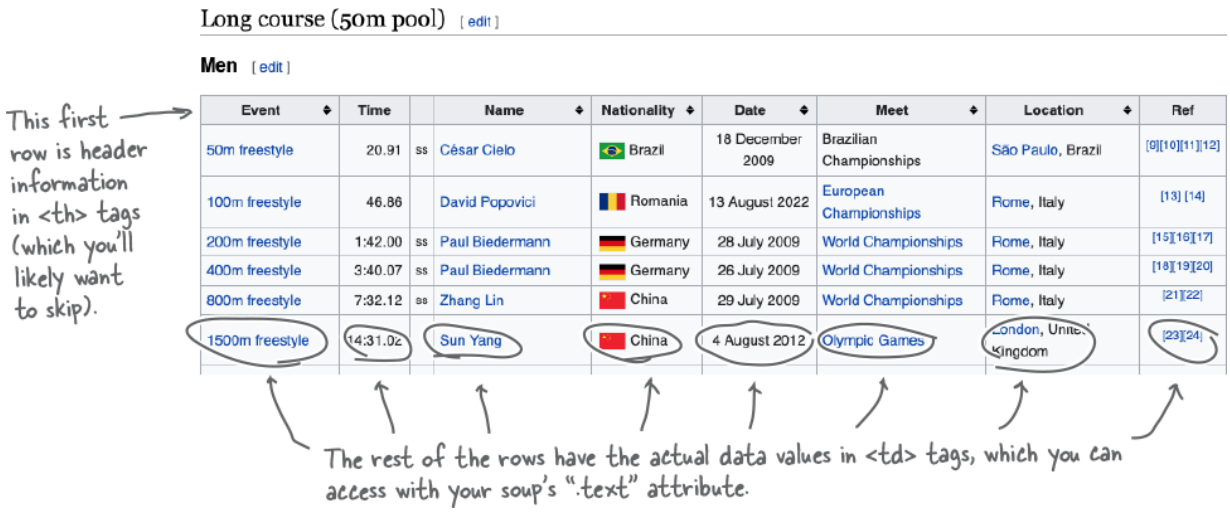
```
print(dir(soup))
```

```
['attrs', 'find', 'get', 'html', 'strip', 'tag', 'text']
```

The “text” attribute returns any textual data associated with an identified HTML tag.

Now take another look at the first handful of rows from the first of your tables: the world records for Men swimming in a 50m pool.

Pay particular attention to the rows:



EXERCISE



If you can extract the data you need from the first table, you can then extract the rest of the data you need from the remaining tables. So, let's start with the first table (in slot #0).

You are going to create a **for** loop which processes the first two columns of data from the first table. For now, your code simply displays on screen the data values it finds. We'll walk you through each line of code you are required to create, and we'd suggest you experiment as needed in your notebook. Write in the lines of code you create in the spaces below. There are six lines of code in total.

Begin by creating a new variable called `table` which is assigned the first slot from your `tables` list:

Your **for** loop iterates over each of the rows (the `<tr>` tags) in `table`, but skips the first row as it contains a row of header informa-

tion. Use the name `row` for your loop variable:

Your **for** loop's block of indented code now starts. Begin this block of code by finding all the `<td>` tags in the current `row`, assigning the soup to a new variable called `columns`:

Create a new variable called `event` which takes as its value the textual data associated with the first column:

Create another new variable called `time` which takes as its value the textual data associated with the second column:

And, finally, in the last line of code in your **for** loop's block, use an f-string together with the **print** BIF to display the `event` and the `time` values on screen:

EXERCISE SOLUTION



If you can extract the data you need from the first table, you can then extract the rest of the data you need from the remaining tables. So, the plan was to start with the first table (in slot #0).

You were asked to create a **for** loop which processes the first two columns of data from the first table. For now, your code needed to display on screen the data values it finds. We walked you through each line of code you were required to create, and we suggested you experimented as needed in your notebook. You were to write in the lines of code you created in the spaces below. There are six lines of code in total.

You were to begin by creating a new variable called `table` which is assigned the first slot from your `tables` list:

```
table = tables[0]
```

Your **for** loop iterates over each of the rows (the `<tr>` tags) in `table`, but skips the first row as it contains a row of header informa-

tion. Use the name `row` for your loop variable:

```
for row in table.find("tr")[1:]:
```

This is a clever use of a slice to skip the first row.

Your **for** loop's block of indented code now starts. Begin this block of code by finding all the `<td>` tags in the current `row`, assigning the soup to a new variable called `columns`:

```
columns = row.find("td")
```

Find all the <td> tags for the current row.

Create a new variable called `event` which takes as its value the textual data associated with the first column:

```
event = columns[0].text
```

The first column in slot #0.

Create another new variable called `time` which takes as its value the textual data associated with the second column:

```
time = columns[1].text
```

The second column in slot #1.

And, finally, in the last line of code in your **for** loop's block, use an f-string together with the **print** BIF to display the `event` and the `time` values on screen:

```
print(f"{event} -> {time}")
```

TEST DRIVE



With your latest **for** loop typed into a single code cell in your `WorldRecords.ipynb` notebook, let's give it a whirl to see the output produced:

Here's the code from the last exercise typed into a single cell.

```
table = tables[0]
for row in table.find("tr")[1:]:
    columns = row.find("td")
    event = columns[0].text
    time = columns[1].text
    print(f"{event} -> {time}")
```

Start with the second row in slot #1 (skipping the header information in slot zero).

```
50m freestyle -> 20.91
100m freestyle -> 46.86
200m freestyle -> 1:42.00
400m freestyle -> 3:40.07
800m freestyle -> 7:32.12
1500m freestyle -> 14:31.02
50m backstroke -> 23.71
100m backstroke -> 51.60
200m backstroke -> 1:51.92
50m breaststroke -> 25.95
100m breaststroke -> 56.88
200m breaststroke -> 2:05.95
50m butterfly -> 22.27
100m butterfly -> 49.45
200m butterfly -> 1:50.34
200m individual medley -> 1:54.00
400m individual medley -> 4:03.84
4×100m freestyle relay -> 3:08.24
4×200m freestyle relay -> 6:58.55
4×100m medley relay -> 3:26.78
```

Here's the list of events extracted from the table.

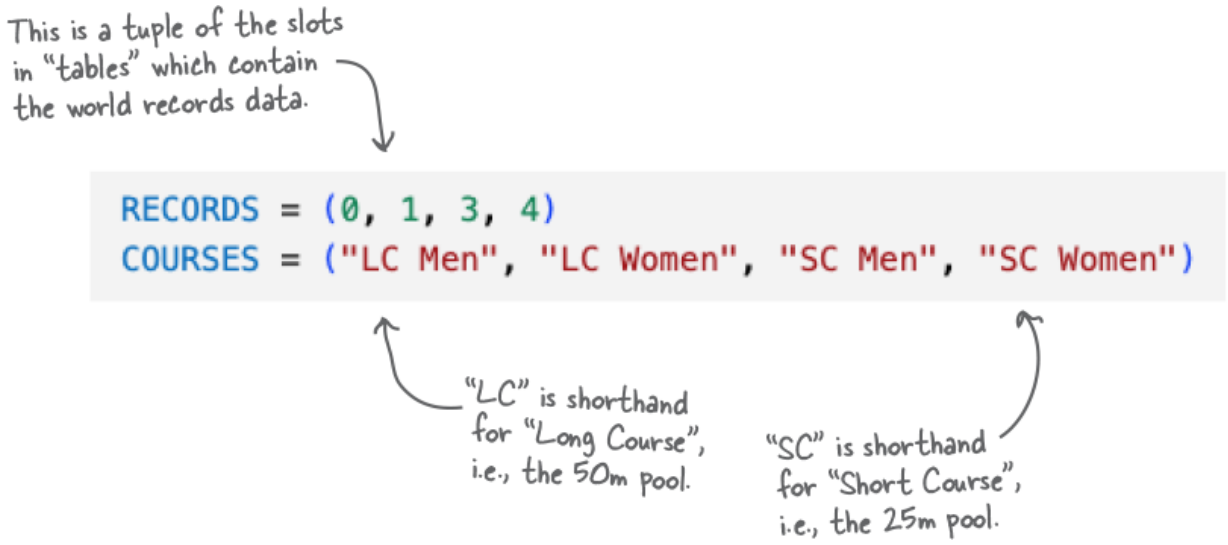
And here's the associated times for each of the identified events.

We don't need all of these associations. But, no worries for now. We'll adjust for this in the next chapter.

Extract data from all the tables, 1 of 2

Having processed one table of world records, it should be easy to apply what your **for** loop does to the rest of them.

To help keep things organized, let's begin by defining two constant tuples to store the slot numbers of the four tables as well as shorthand names of the four courses:



With these tuples defined, the **zip** BIF can be used to associate the slot numbers with course names. You can see how this happens by sending the output from **zip** to the **list** BIF:

```
list(zip(RECORDS, COURSES))
```

```
[(0, 'LC Men'), (1, 'LC Women'), (3, 'SC Men'), (4, 'SC Women')]
```

The two-element tuples are zipped together, then turned into a list. You end up with a list of two-element tuples.

A **for** loop can iterate over your list of two-element tuples, with each slot number and course name being assigned to *two* individual loop variables on each iteration, using a line of code like this:

```
for table, course in zip(RECORDS, COURSES):
```

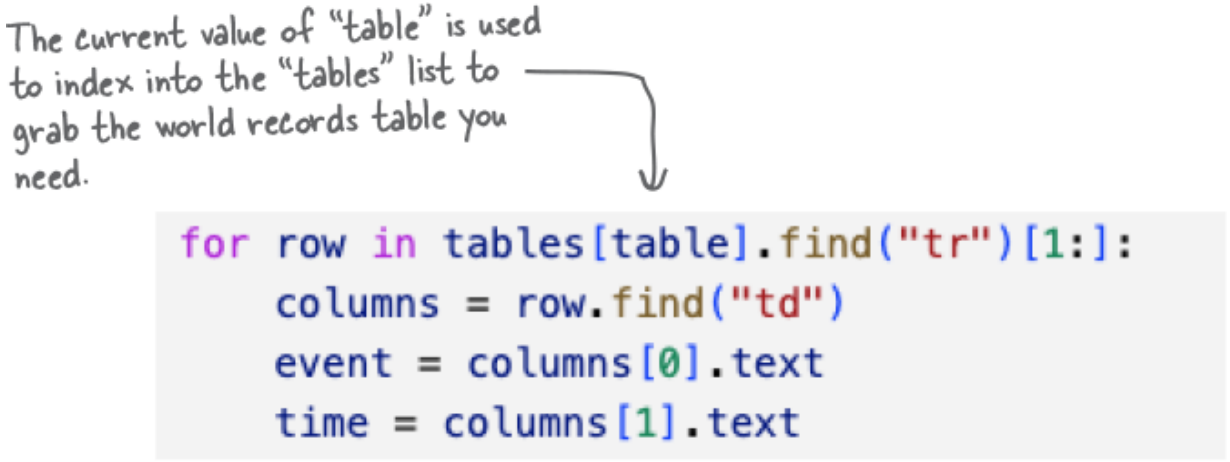
There's no need to explicitly convert the output from "zip" to a list, as the "for" loop does so for you based on context.

Extract data from all the tables, 2 of 2

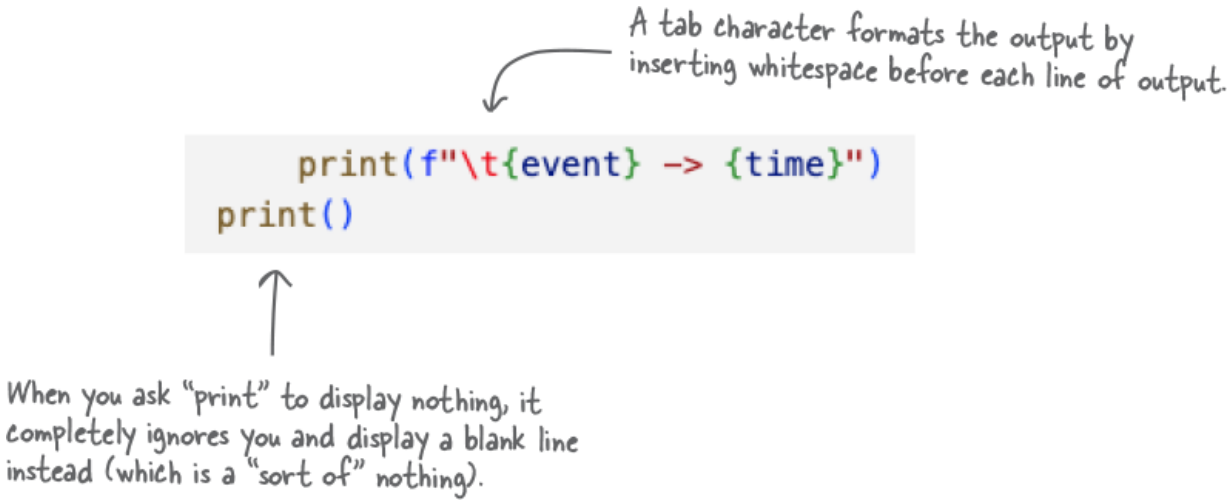
Within your **for** loop's code block, the value of the `table` and `course` loop variables can be used as needed. With that in mind, and to keep the output produced manageable, let's display the current `course` value on screen:

```
print(f"{course}:")
```

Having displayed the course name, another (nested) **for** loop can process the current table's rows of data, using code very similar to what you wrote for the most-recent *Exercise*:



With the `event` and `time` values determined, two calls to print format the output appropriately:



Let’s throw this latest chunk of code into a code cell, run it, then see what happens…

TEST DRIVE



Go ahead and type the code from the last two pages into a single cell in your notebook, then press **Shift+Enter**:

```python
RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

for table, course in zip(RECORDS, COURSES):
    print(f"{course}:")
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()
```




```
LC Men:
        50m freestyle -> 20.91
        100m freestyle -> 46.86
        200m freestyle -> 1:42.00
        400m freestyle -> 3:40.07
        800m freestyle -> 7:32.12
        1500m freestyle -> 14:31.02
        50m backstroke -> 23.71
        100m backstroke -> 51.60
        200m backstroke -> 1:51.92
        50m breaststroke -> 25.95
        100m breaststroke -> 56.88
        200m breaststroke -> 2:05.95
        50m butterfly -> 22.27
        100m butterfly -> 49.45
        200m butterfly -> 1:50.34
        200m individual medley -> 1:54.00
        400m individual medley -> 4:03.84
        4×100m freestyle relay -> 3:08.24
        4×200m freestyle relay -> 6:58.55
        4×100m medley relay -> 3:26.78

LC Women:
        50m freestyle -> 23.67
        100m freestyle -> 51.71
        200m freestyle -> 1:52.98
        400m freestyle -> 3:56.40
        800m freestyle -> 8:04.79
        1500m freestyle -> 15:20.48
        50m backstroke -> 26.98
        100m backstroke -> 57.45
        200m backstroke -> 2:03.35
        50m breaststroke -> 29.30
        100m breaststroke -> 1:04.13
        200m breaststroke -> 2:18.95
        50m butterfly -> 24.43
        100m butterfly -> 55.48
        200m butterfly -> 2:01.81
        200m individual medley -> 2:06.12
        400m individual medley -> 4:26.36
        4×100m freestyle relay -> 3:29.69
        4×200m freestyle relay -> 7:39.29
        4×100m medley relay -> 3:50.40

SC Men:
        50m freestyle -> 20.16
        100m freestyle -> 44.84
        200m freestyle -> 1:39.37
        400m freestyle -> 3:32.25
        800m freestyle -> 7:23.42
        1500m freestyle -> 14:06.88
        50m backstroke -> 22.22
        100m backstroke -> 48.33
        200m backstroke -> 1:45.63
        50m breaststroke -> 24.95
        100m breaststroke -> 55.28
        200m breaststroke -> 2:00.16
        50m butterfly -> 21.75
        50m butterfly -> 21.75
        100m butterfly -> 47.78
```

```
        200m butterfly -> 1:48.24
        200m butterfly -> 1:46.85
        100m individual medley -> 49.28
        200m individual medley -> 1:49.63
        400m individual medley -> 3:54.81
        4×50m freestyle relay -> 1:21.80
        4×50m freestyle relay -> 1:20.77
        4×100m freestyle relay -> 3:03.03
        4×200m freestyle relay -> 6:46.81
        4×50m medley relay -> 1:30.14
        4×100m medley relay -> 3:19.16

SC Women:
        50m freestyle -> 22.93
        100m freestyle -> 50.25
        200m freestyle -> 1:50.31
        400m freestyle -> 3:53.92
        400m freestyle -> 3:51.30
        800m freestyle -> 7:59.34
        800m freestyle -> 7:57.42
        1500m freestyle -> 15:18.01
        1500m freestyle -> 15:08.24
        50m backstroke -> 25.27
        100m backstroke -> 54.89
        200m backstroke -> 1:58.94
        50m breaststroke -> 28.56
        100m breaststroke -> 1:02.36
        100m breaststroke -> 1:02.36
        100m breaststroke -> 1:02.36
        200m breaststroke -> 2:14.57
        50m butterfly -> 24.38
        100m butterfly -> 54.59
        200m butterfly -> 1:59.61
        100m individual medley -> 56.51
        200m individual medley -> 2:01.86
        400m individual medley -> 4:18.94
        4×50m freestyle relay -> 1:32.50
        4×100m freestyle relay -> 3:26.53
        4×200m freestyle relay -> 7:32.85
        4×50m medley relay -> 1:42.38
        4×50m medley relay -> 1:42.38
        4×100m medley relay -> 3:44.52
```

The data.

Remember: "LC" is shorthand for "Long Course", whereas "SC" is shorthand for "Short Course". Be advised that you may see different numbers than what's shown here as some swimming world records may have fallen since this book published.

That nested loop did the trick!

With your most recent chunk of code, you've completed the third task. It looks like the Coach's suggestion of breaking up the tasks is working! Look at how much you accomplished:

1. **Grab the raw HTML page from Wikipedia.** ✔

2. **Identify the tables that contain the data you need.** ✔

3. **Process the first two columns of each table to extract the record data.** ✔

You now have access to the two columns of data from each of the four tables from the *Wikipedia* HTML page. You're done with HTML parsing (for now). All that's left to do is to manipulate these two columns of data to enable their integration into the Coach's webapp. That's coming in the next chapter.

As for the rest of this chapter, there's a quick review, followed by the obligatory end-of-chapter crossword. Enjoy!

CHAPTER REVIEW BULLET POINTS

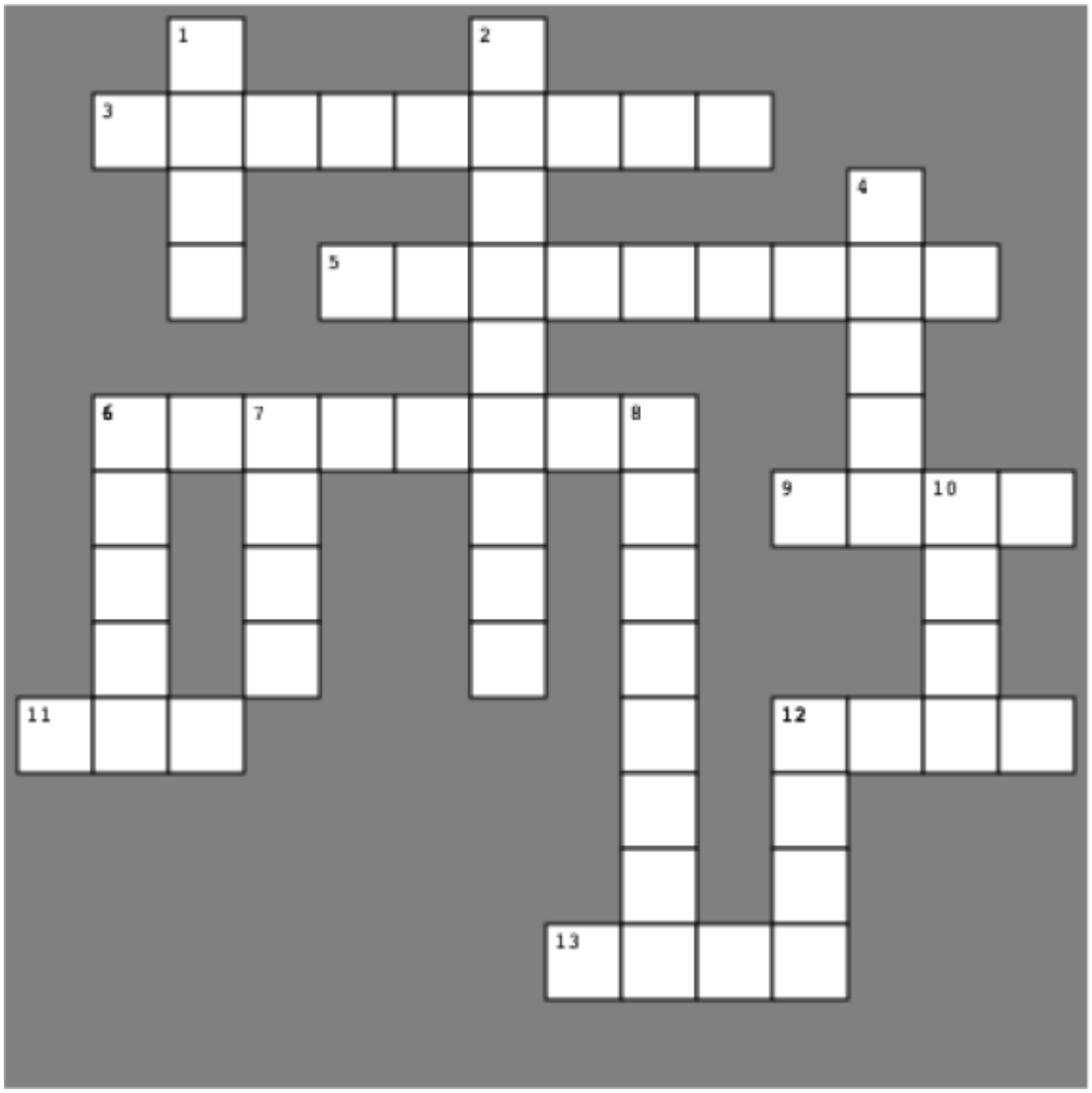
- When it comes to data on the web, HTML is king.
- HTML is a text-based format which can be processed easily with code assuming the use of a HTML-parsing technology.
- Python has many HTML-parsing libraries available on PyPI and, in this chapter, you learned about `gazpacho`.
- Gazpacho is a cold soup, popular in Spain and Portugal. `gazpacho` is a Python library which parses HTML, which it refers to as "soup".
- `gazpacho` comes with a powerful built-in method called **find** which can search all manner of parsed HTML quickly and efficiently.
- `gazpacho` can be configured to always **return parsed data as a list**, which means all the usual Python looping machinery can be employed to grab the data you're interested in.
- When processing HTML tables with `gazpacho`, the **find** method can return all the rows, as well as all the data items within each row. Think `<tr>` and `<td>` HTML tags.
- An extension to Python's square bracket notation lets you specify **slices** which can extract portions of any sequence.
- Slices are specified with the `start : stop : step` values (which you'll find used in many places within the Python world). Each of these three values employ sensible defaults which provide for compact, and powerful, slice specifications.

- One *gotcha* with specifying `stop` is that the returned slice includes the data **up-to-but-not-including** the `stop` slot. This behavior trips everyone up, so don't worry if you get this wrong (as it's an easy fix).

The HTML Parsing Crossword



As always, all the answers to the clues are found in this chapter (with solutions on the next page).



Across

3. It's a large, globally distributed source of knowledge, data, and facts.

5. A useful BIF which adds numbers to your **for** loops.

6. The term used when extracting tags from a web page.

9. Web pages are mainly this.

11. Use this method to grab a copy of a web page from a web site.

12. When specifying 6 down, this lets you select every second, third, forth (and so on) slot.

13. It can be hot, cold, or 2 down.

Down

1. This method searches a web page for specified tags.

2. Attractive (another word for).

4. When specifying 6 down, this lets you indicate where to begin.

6. It might have to do with toast, but doesn't.

7. Tables are made from many of these.

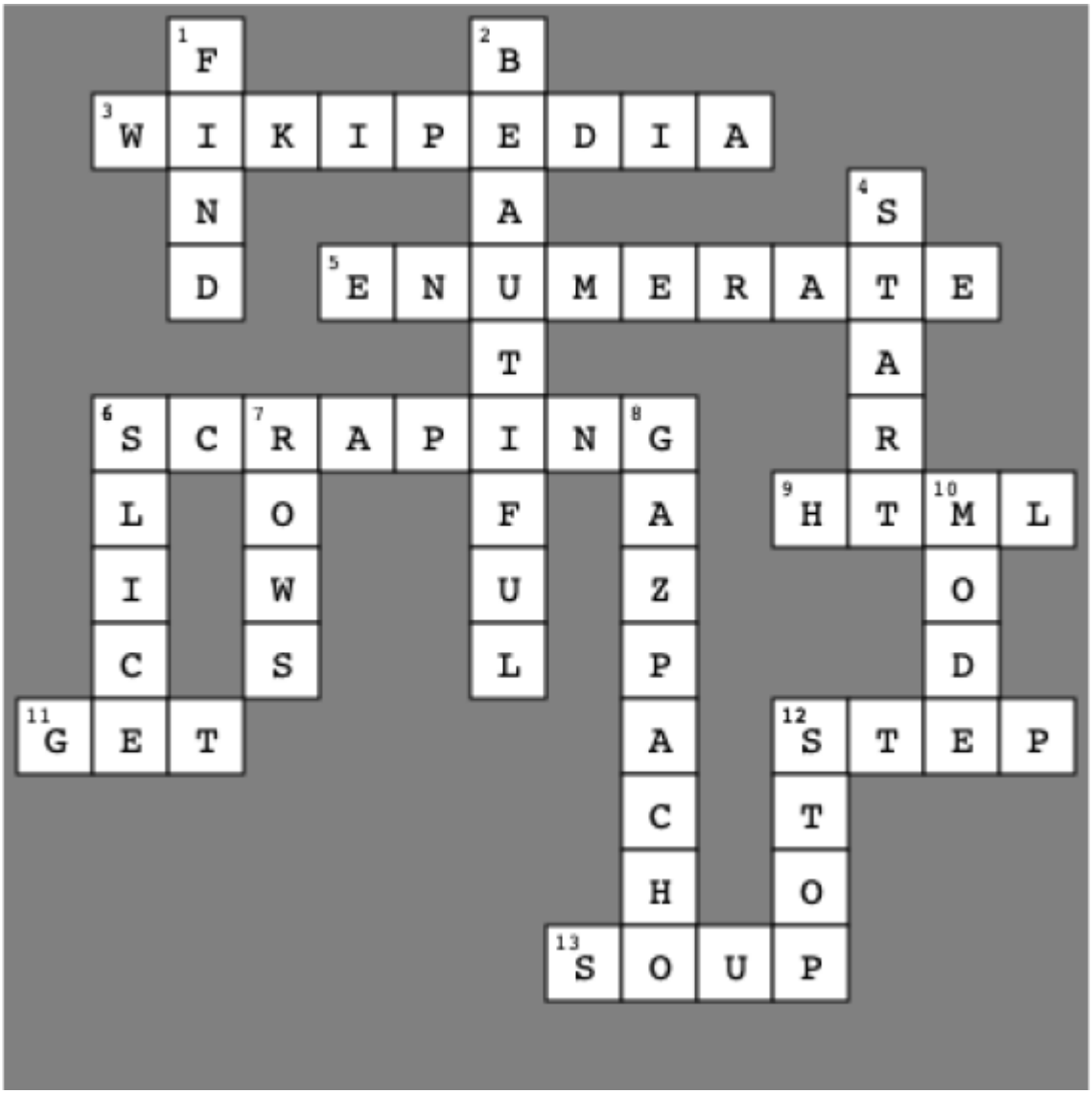
8. A type of 13 across, but also the name of a web parsing library.

10. Use this parameter to ensure a list is always returned

12. When specifying 6 down, this is the upto-but-do-not-include a value.

The HTML Parsing Crossword Solution





Across

3. It's a large, globally distributed source of knowledge, data, and facts.

5. A useful BIF which adds numbers to your **for** loops.

6. The term used when extracting tags from a web page.

9. Web pages are mainly this.

11. Use this method to grab a copy of a web page from a web site.

12. When specifying 6 down, this lets you select every second, third, forth (and so on) slot.

13. It can be hot, cold, or 2 down.

Down

1. This method searches a web page for specified tags.

2. Attractive (another word for).

4. When specifying 6 down, this lets you indicate where to begin.

6. It might have to do with toast, but doesn't.

7. Tables are made from many of these.

8. A type of 13 across, but also the name of a web parsing library.

10. Use this parameter to ensure a list is always returned

12. When specifying 6 down, this is the upto-but-do-not-include a value.

Chapter 9. Working with Data: *Data Manipulation*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 9 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



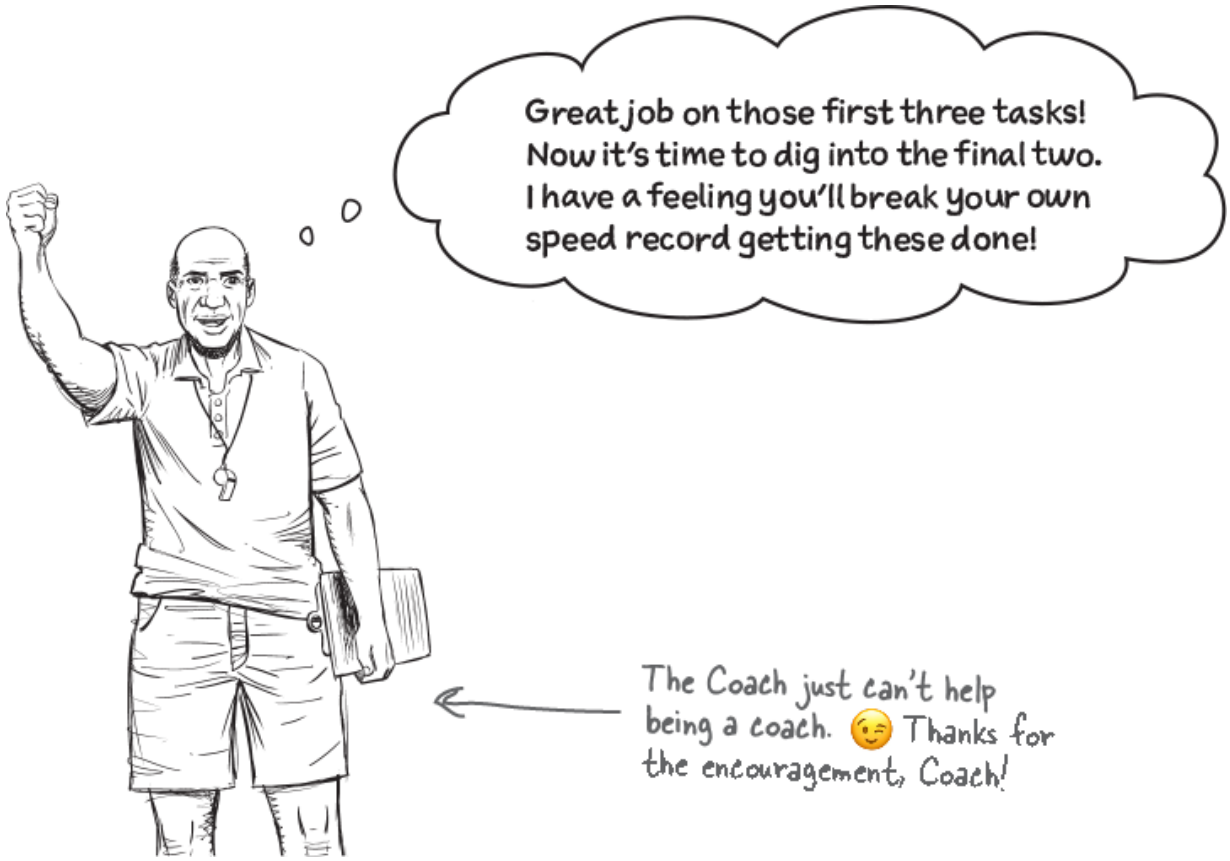
Sometimes your data isn't arranged in the way it needs to be.

Perhaps you have a *list of lists*, but you really need a *dictionary of dictionaries*. Or perhaps you need to relate a value in one data structure to a value in another, but the values don't quite match. Urrgh, that's so frustrating. Not to worry: the power of Python is here to help. In this chapter, you'll use Python to **wrangle** your data to change the scraped data from the end of the previous chapter into something truly *useful*. You'll see how to **exploit** Python's dictionary as a **lookup** table. There's conversions, integrations, updates, deployments, and more in this chapter. And, at the end of it all, the Coach's napkin-spec becomes *reality*. Can't wait? Neither can we… let's get to it!

Bending your data to your will…

You've now written the code which completes three of the five data manipulation tasks:

1. **Grab the raw HTML page from Wikipedia.** ✔
2. **Identify the tables that contain the data you need.** ✔
3. **Process the first two columns of each table to extract the record data.** ✔



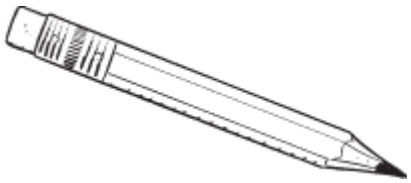
The remaining work involves two additional tasks:

4 **Convert the record data into a dictionary of swimming world records.**

5 **Use the dictionary to add world records to the bottom of each bar chart.**

Let's keep the momentum going by quickly tackling Task #4.

SHARPEN YOUR PENCIL



Assuming the existence of the `tables` list, this code processes the four world record tables and displays the extracted events and times on screen:

```
RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

for table, course in zip(RECORDS, COURSES):
    print(f"{course}:")
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()
```

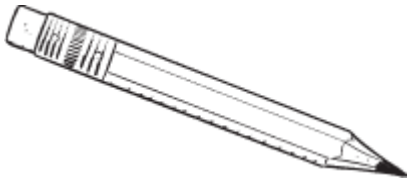
A nested "for" loop does the trick here (even though, for now, all the data appears on screen).

While continuing to work in your `WorldRecords.ipynb` notebook, your job now is to convert that code to instead populate a *dictionary of dictionaries*. The *outer* dictionary has four keys: the four values from the `COURSES` tuple. Associated with each of these keys is another *inner* nested dictionary which associates each of the events with their world record times.

Experiment within your notebook to adjust the loop code to build the required dictionary of dictionaries. When you are done, write-in the

code you created in the space below.

SHARPEN YOUR PENCIL SOLUTION



Your most-recent loop code processes the four world record tables and displays the extracted events and times on screen. Your job was to convert that code to instead populate a *dictionary of dictionaries*. The *outer* dictionary is to have four keys: the four values from the `COURSES` tuple. Associated with each of these keys is another *inner* nested dictionary which associates each of the events with their world record times.

Here's your code from earlier, as well as the code which we came up with. We're highlighting the changes that we made. How do our changes compare to yours?

The three print statements are *not* needed anymore.

```
for table, course in zip(RECORDS, COURSES):
    print(f"{course}:")
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()
```

We gave our "outer" dictionary the name "records", then initially set it to an empty dictionary.

Each course name is a key in the "records" dictionary, which (initially) has a new empty dictionary associated with it.

```
records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

The dictionary associated with each course name grows by a row, which associates the "event" with its "time". This line of code populates the dictionary of dictionaries with data.

TEST DRIVE



If you haven't done so already, go ahead a type your dictionary of dictionaries creating code into an empty code cell, then press **Shift+Enter** to populate `records`:

```
records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

This code isn't all that different to the code which used the "print" BIF to display the records data on screen. All that's different here is where the data ends up. This code saves a copy of the data two the "records" dictionary.

You can ask the `records` dictionary to fess-up it's list of keys:

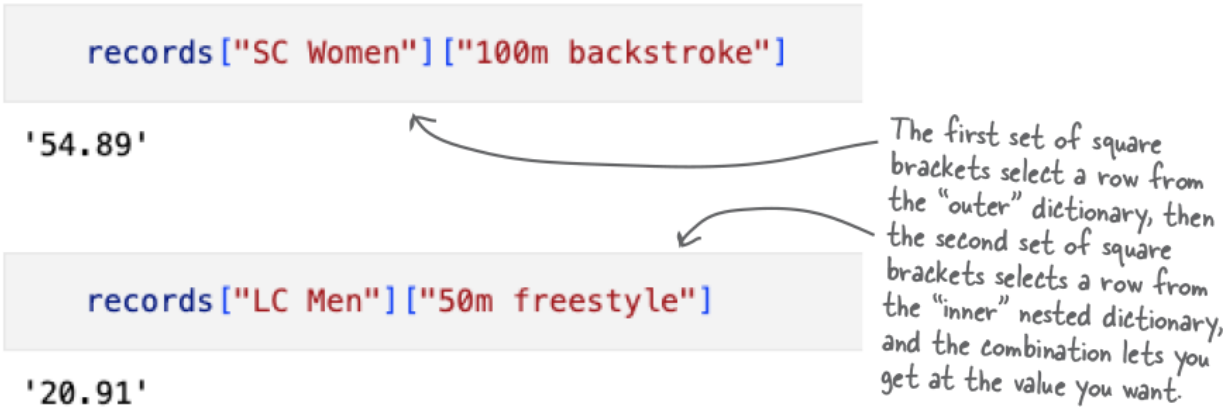
```
records.keys()
```

```
dict_keys(['LC Men', 'LC Women', 'SC Men', 'SC Women'])
```

Remember: a "dict_keys" object behaves like a list.

And then you can pick out specific records of interest by providing the course name followed by the event descriptor, which spits out the

world record time:



You now have the data you need…

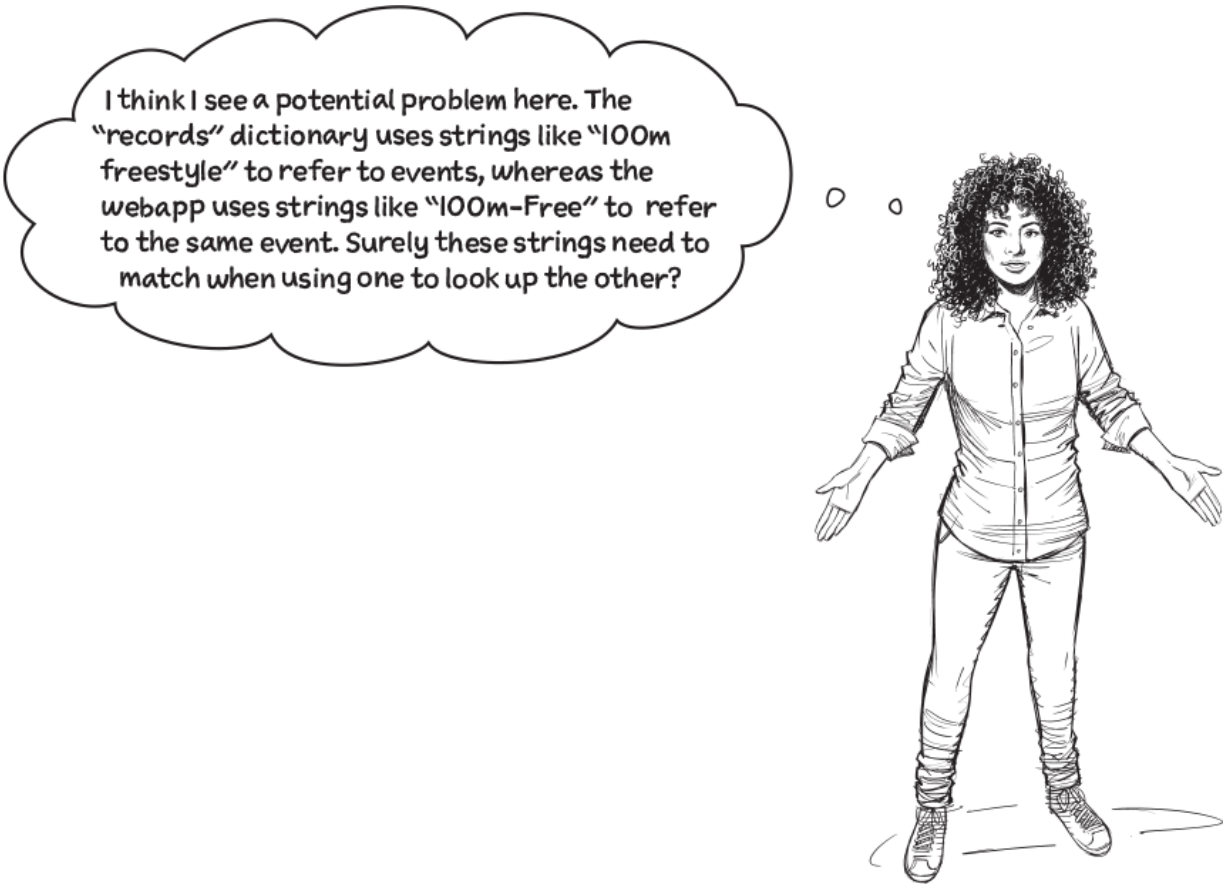
With the `records` dictionary of dictionaries created, Task #4 is done!

4 Convert the record data into a dictionary of swimming world records. ✔

5 Use the dictionary to add world records to the bottom of each bar chart.



However, a subtle problem might well be lurking…



Yes, we see the issue, too.

Once the Coach has selected an event from the webapp's second drop-down list, how do we match the chosen value against the keys in the `records` dictionary's *inner* dictionaries?

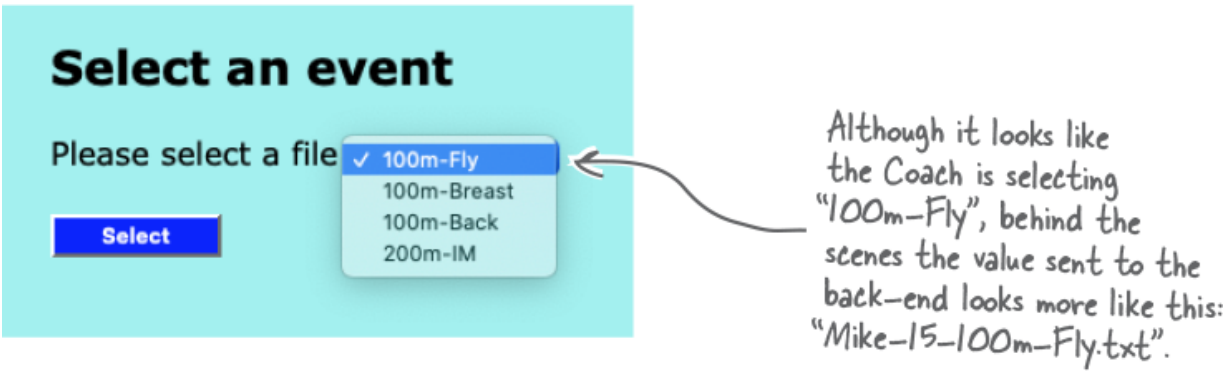
Let's consider this problem in detail.

How can the webapp's selected data values be related to the data in the records dictionary?

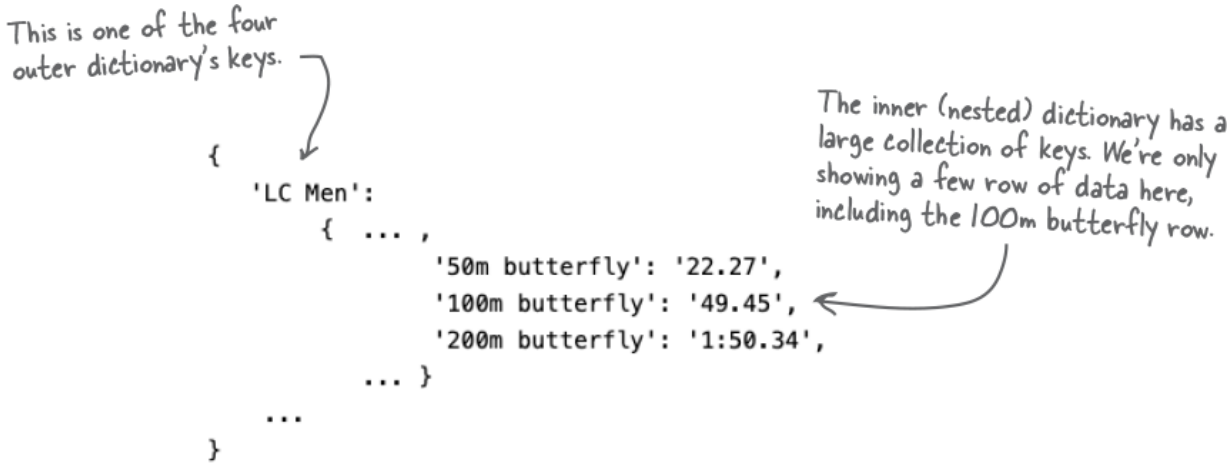


Behind the Scenes

1. **The webapp receives a value that looks like this: `Mike-15-100m-Fly.txt`. This value is then used to create, then display, the swimmer's bar chart.**



2 **The `records` dictionary has four keys (the four course names) with each key associated with an inner, nested dictionary. The keys of the inner dictionary are the event names, which take this form: `100m butterfly`.**

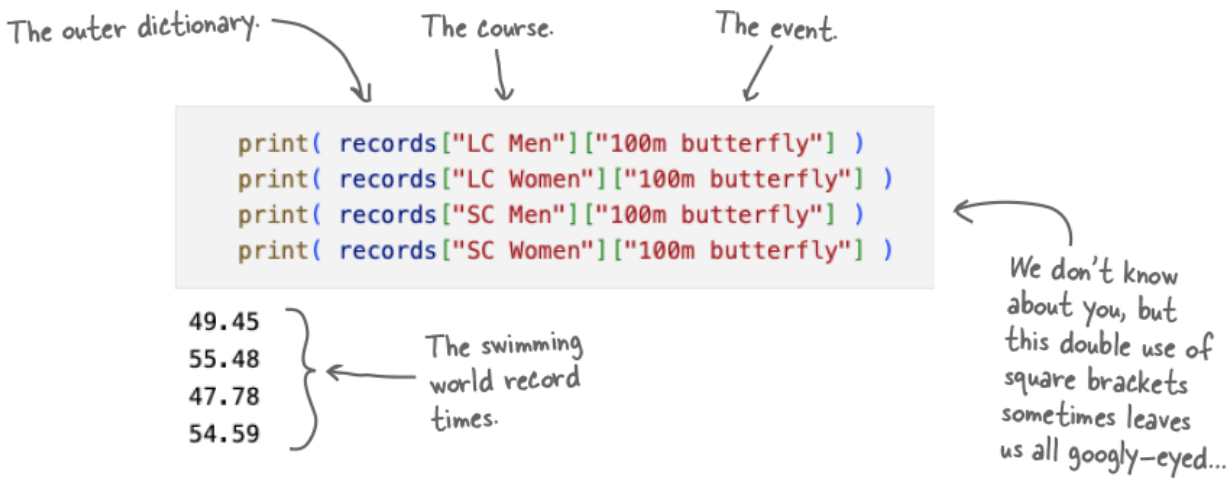


3 **If your can convert `Mike-15-100m-Fly.txt` into `100m butterfly` then you can use the latter value to extract the four 100m butterfly world records from the `records` dictionary.**

Apply what you already know…

The required conversion can be accomplished with a handful of lines of code.

Before we begin, here's how you'd "manually" retrieve the swimming world records for the 100m butterfly for each of the course types:



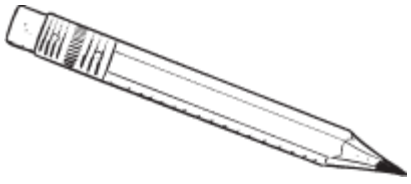
Using a dictionary is a great choice when performing conversions. In an effort to confirm just how imaginative we can be when it comes to naming variables (and when *pushed*), here's a dictionary called `conversions` which *associates* short stroke names with their longer equivalents:



The conversion from the webapp-supplied filename to the equivalent inner (nested) dictionary key uses techniques you've already exploited a number of times in this book:



SHARPEN YOUR PENCIL



The code on the last page has furnished you with a value for `lookup`.

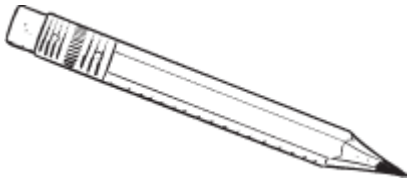
You task is to take the `lookup` variable and use it in a **for** loop to iterate through each of the course names in the `records` dictionary, then display the four swimming world records associated with the value of `lookup`.

The output displayed should match the output from the “manual” code at the top of the last page. Take all the time you need to experiment then, when it’s working, write the code you used into the space below:

FYI: Here's what the "100m butterfly" rows of data look like on the Wikipedia page.

100m butterfly	49.45		Caeleb Dressel	United States	31 July 2021	Olympic Games	Tokyo, Japan	[41]
100m butterfly	55.48		Sarah Sjöström	Sweden	7 August 2016	Olympic Games	Rio de Janeiro, Brazil	[80][81]
100m butterfly	47.78		Caeleb Dressel	United States	21 November 2020	International Swimming League	Budapest, Hungary	[115][116]
100m butterfly	54.59		Kelsi Dahlia	United States	3 December 2021	International Swimming League	Eindhoven, Netherlands	[160] [161]

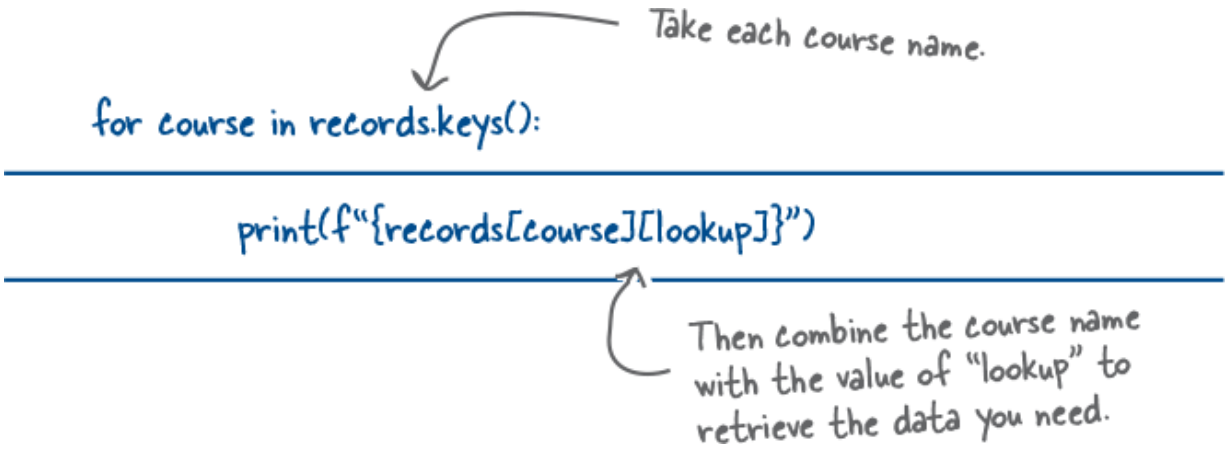
SHARPEN YOUR PENCIL SOLUTION



You were furnished you with a value for `lookup`.

You task was to take the `lookup` variable and use it in a **for** loop to iterate through each of the course names in the `records` dictionary, then display the four swimming world records associated with the value of `lookup`.

The output displayed had to match the output from the "manual" code from earlier. You were to take all the time you needed to experiment then, when write the code you used into the space below. Here's the code we came up with:



TEST DRIVE



Running your latest loop displays the same output as the “manual” code from earlier, confirming that – given a value for `lookup` – you can retrieve the four swimming world record values to include with the swimmer’s bar chart:

```
for course in records.keys():
    print(f"{records[course][lookup]}")
```

```
49.45
55.48
47.78
54.59
```

These swimming world record times match those produced by the “manual” code from earlier.

The “googly-eyed” use of double square brackets is still here, but it doesn’t look quite so bad in this code, does it?

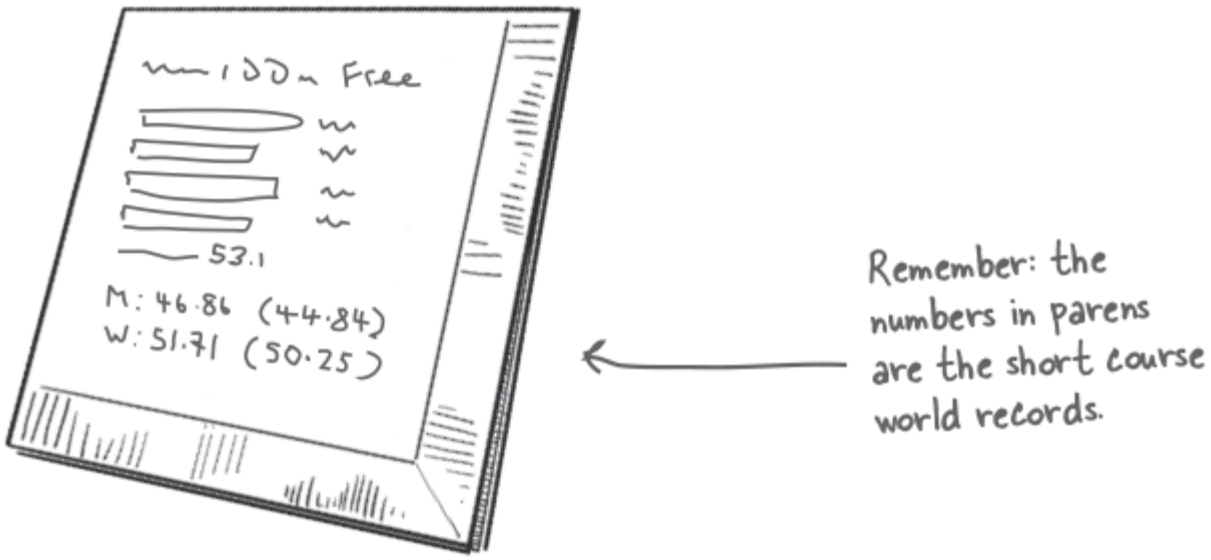
You’re now ready to update your bar charts

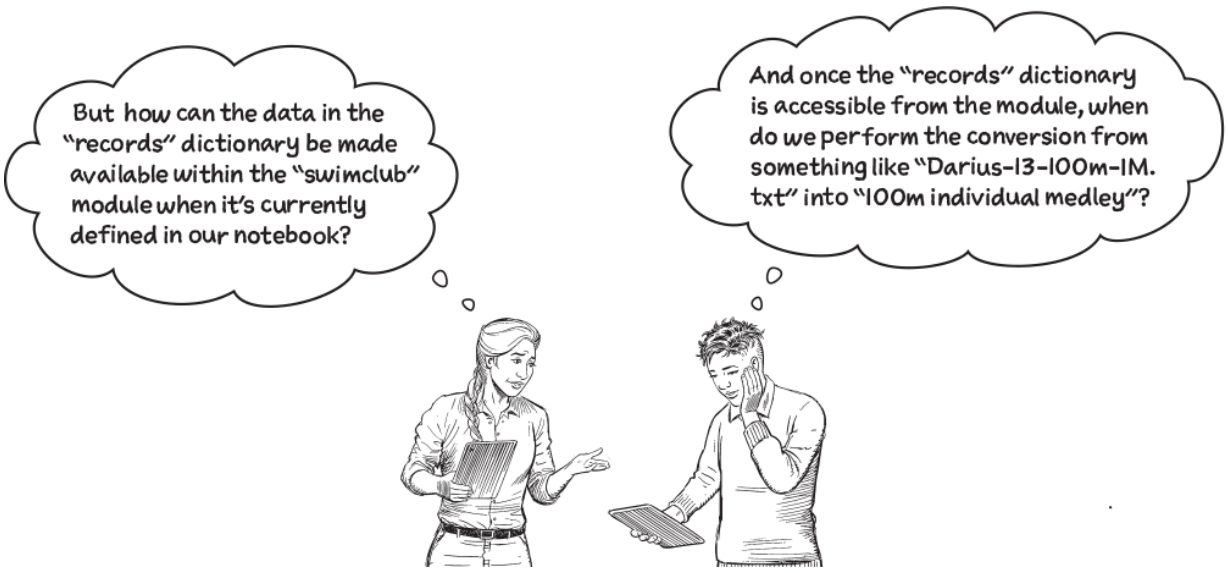
You're inching towards being able to contact the Coach to confirm you can display the world record times at the bottom of each swimmer's bar chart. You have the data you need in the `records` dictionary of dictionaries, and you can retrieve the four world record times by indexing into the `records` dictionary using two keys: the name of the course and the name of the event.

All that's left to do is Task #5:

5 Use the dictionary to add world records to the bottom of each bar chart.

Your bar chart producing code is in the `swimclub.py` module, so that's likely a good place to add in the code that retrieves the four world record times in order to include them at the bottom of the bar chart, per the Coach's napkin-spec:





Those are good questions.

And both are clearly something to think about… and *answer*.

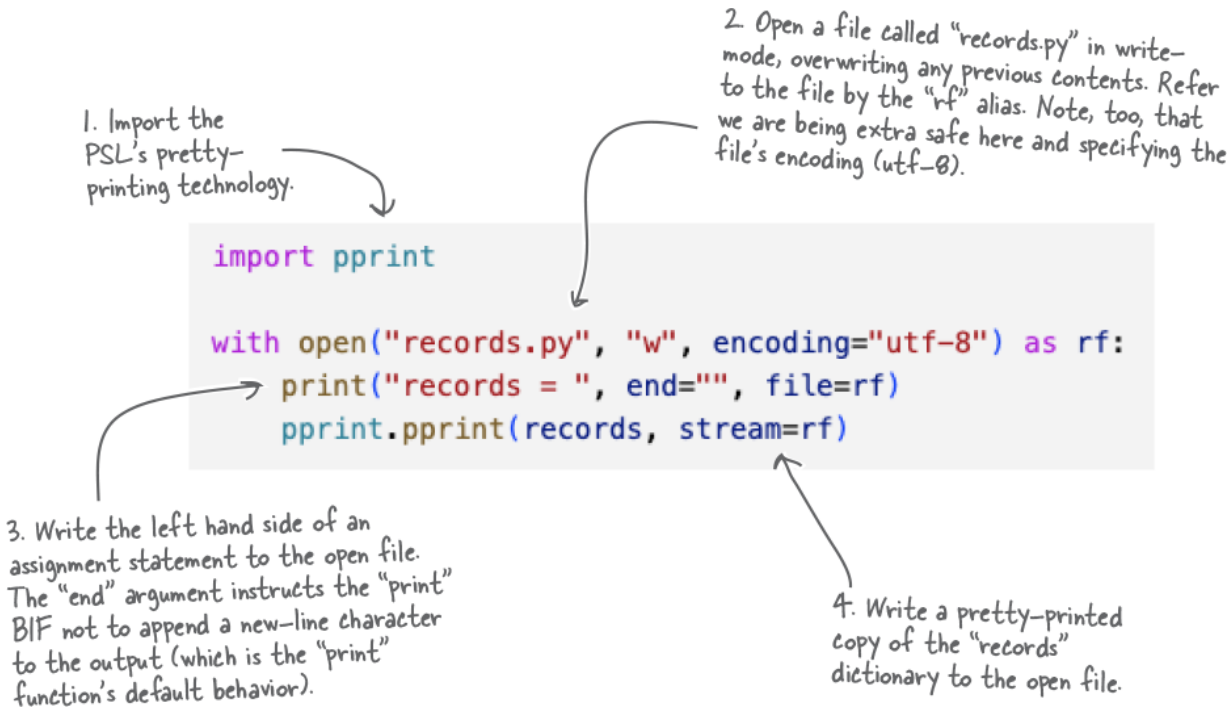
Convert the dictionary into a module

A quick'n'dirty solution to sharing the data in the `records` dictionary takes advantage of the fact that when a module is imported into the Python interpreter, the code in the imported file is *executed* top-to-bottom.

With this in mind, you could simply dump a copy of the `records` dictionary into a file, then import that file as needed. In fact, let's dump a pretty-printed copy of `records` to a file so that the dictionary is a

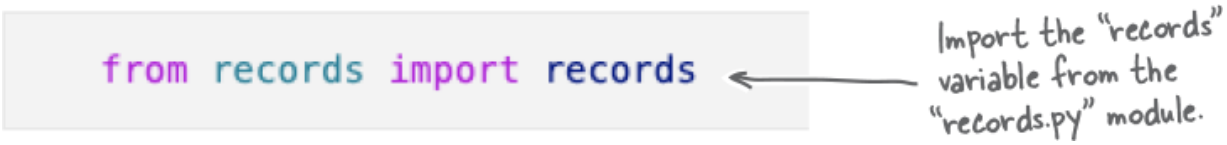
little easier on the eye (when, for instance, viewed with a text editor by a human).

The `pprint` module from the PSL (first seen in Chapter 5) pretty-prints data structures on demand. When combined with the **open** BIF, only four lines of code are needed to "save" the `records` dictionary of dictionaries into its own module:



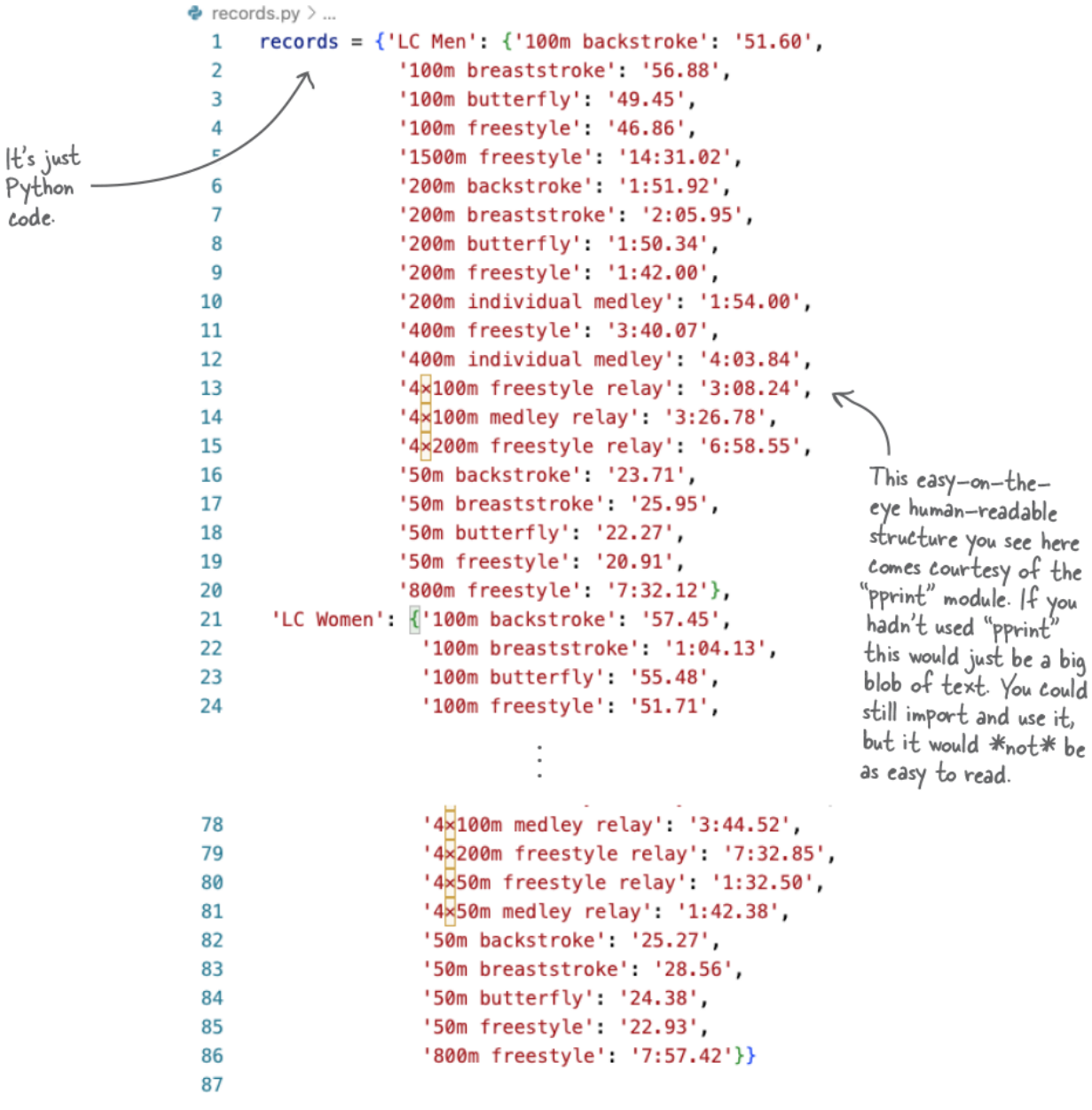
As with all **with** statements, as soon as its code block *ends*, the Python interpreter closes the open file (saving you the trouble).

As soon as the above code runs, any other Python code can import the `records` dictionary of dictionaries with this line of code:



The records module is just Python code

If you open `records.py` in VS Code you can confirm the code from the previous page has created an importable Python module:


records.py > ...
records = {'LC Men': {'100m backstroke': '51.60',
'100m breaststroke': '56.88',
'100m butterfly': '49.45',
'100m freestyle': '46.86',
'1500m freestyle': '14:31.02',
'200m backstroke': '1:51.92',
'200m breaststroke': '2:05.95',
'200m butterfly': '1:50.34',
'200m freestyle': '1:42.00',
'200m individual medley': '1:54.00',
'400m freestyle': '3:40.07',
'400m individual medley': '4:03.84',
'4×100m freestyle relay': '3:08.24',
'4×100m medley relay': '3:26.78',
'4×200m freestyle relay': '6:58.55',
'50m backstroke': '23.71',
'50m breaststroke': '25.95',
'50m butterfly': '22.27',
'50m freestyle': '20.91',
'800m freestyle': '7:32.12'},
'LC Women': {'100m backstroke': '57.45',
'100m breaststroke': '1:04.13',
'100m butterfly': '55.48',
'100m freestyle': '51.71',
'4×100m medley relay': '3:44.52',
'4×200m freestyle relay': '7:32.85',
'4×50m freestyle relay': '1:32.50',
'4×50m medley relay': '1:42.38',
'50m backstroke': '25.27',
'50m breaststroke': '28.56',
'50m butterfly': '24.38',
'50m freestyle': '22.93',
'800m freestyle': '7:57.42'}}
It's just Python code.
This easy-on-the-eye human-readable structure you see here comes courtesy of the "pprint" module. If you hadn't used "pprint" this would just be a big blob of text. You could still import and use it, but it would *not* be as easy to read.



It depends on your background.

If you are coming to Python from a programming language which requires you to *compile*, *link*, then *run* your code, what's just happened might seem like sacrilege. After all, having code that writes code at run-time, then proceeds to execute it, must surely be *against* the laws of the programming universe, right?

Granted, with those "traditional" types of programming languages, what's just happened isn't even possible, but that's not reason enough to believe generating, then running, code in this way should be outlawed. When presented with arguments against writing code which writes code (then executes it), most Pythonista's shrug, then ask: *"Why not take advantage of this programming ability?"*

THERE ARE NO DUMB QUESTIONS

Q: What's a "Pythonista"?

A: It's the name the Python programming community gives to someone who uses Python to get their job done (as well as someone who *loves* Python as a programming language). Congratulations, at this point in this book, you can call yourself a *Pythonista*.

Q: Is the "pprint" module of any practical use within a running program or is it primarily an interactive nicety?

A: Here's what the Python docs has to say: *The pprint module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter*. This means that not only does **pprint**'s output look nice, it's readable by Python, too, which can often be useful.

EXERCISE



With the `records.py` module now in existence, let's turn our attention to filenames which look like:

```
Darius-13-100m-IM.txt
```

so that they can be converted into a lookup key which looks like:

```
100m individual medley
```

We'd like you to create a function that takes a single argument value, the filename of any swimmer's data (for example, `Darius-13-100m-IM.txt`). Upon receipt of this value, your function should convert the filename into the correct lookup key, which the function then returns. All of the code you need to write this new function already exists within your `WorldRecords.ipynb` notebook. All you have to do is gather together the code to create your new function.

Call your new function `event_lookup`, and write it in such a way that it can be added to the `swimclub.py` module. That is, your

function should not rely on any of your notebook's pre-existing variable definitions. Write the code your come up with in the space below:

EXERCISE SOLUTION



With the `records.py` module now in existence, you were to turn your attention to filenames which look like:

```
Darius-13-100m-IM.txt
```

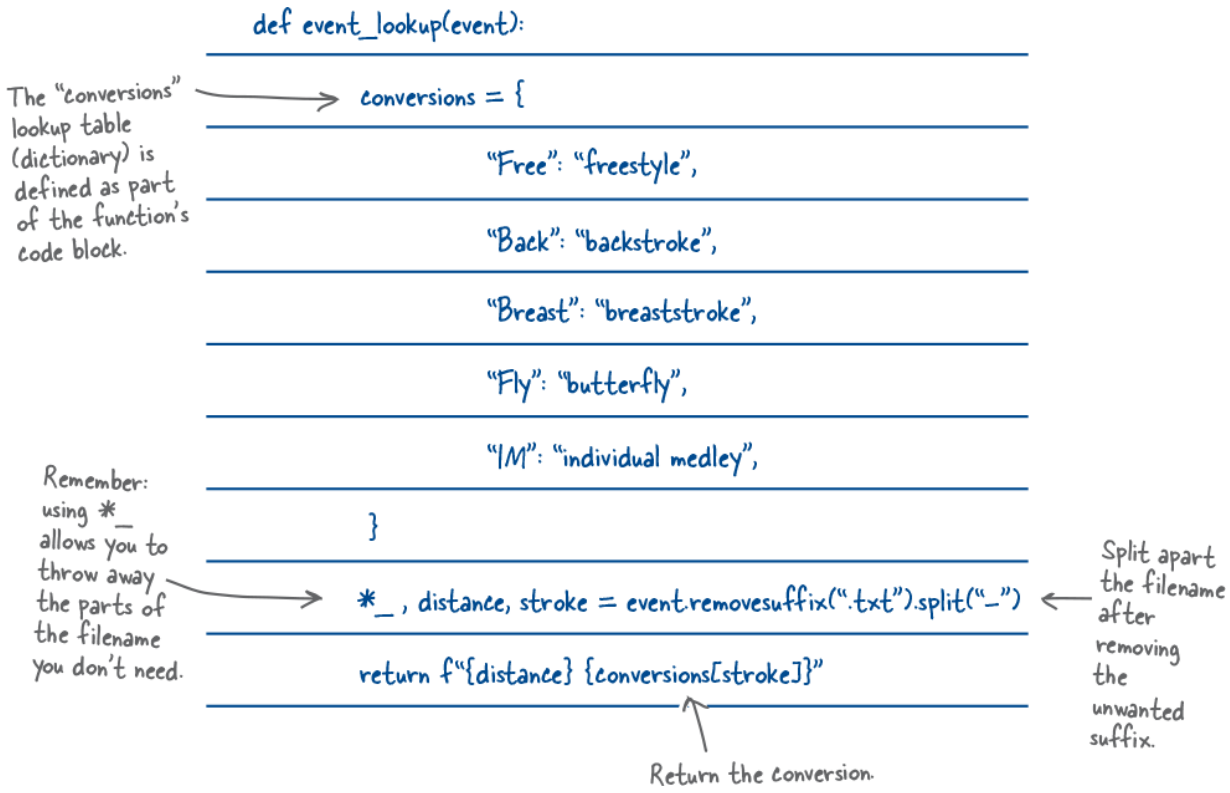
so that they can be converted into a lookup key which looks like:

```
100m individual medley
```

We asked you to create a function that takes a single argument value, the filename of any swimmer's data (for example, `Darius-13-100m-IM.txt`). Upon receipt of this value, your function was to convert the filename into the correct lookup key, which the function then returns. All of the code you need to write this new function already existed within your `WorldRecords.ipynb` notebook. All you had to do is gather together the code to create your new function.

You were to call your new function `event_lookup`, and write it in such a way that it can be added to the `swimclub.py` module. That

is, your function should not rely on any of your notebook's pre-existing variable definitions. Here's the code we created:



TEST DRIVE



Now that you have the `event_lookup` function, you can create a new notebook (we called ours `TestStandalone.ipynb`) to see if your latest code can execute correctly independently of the `WorldRecords.ipynb` notebook. Here's the code we added to our latest notebook:

Define the function.

```
def event_lookup(event):

    conversions = {
        "Free": "freestyle",
        "Back": "backstroke",
        "Breast": "breaststroke",
        "Fly": "butterfly",
        "IM": "individual medley",
    }

    *_, distance, stroke = event.removesuffix(".txt").split("-")

    return f"{distance} {conversions[stroke]}"
```

Test the new function against some filenames.

```
print(event_lookup("Darius-13-100m-Back.txt"))
print(event_lookup("Darius-13-100m-Breast.txt"))
print(event_lookup("Darius-13-100m-Fly.txt"))
print(event_lookup("Darius-13-100m-IM.txt"))
```

```
100m backstroke
100m breaststroke
100m butterfly
100m individual medley
```

Import the "records" dictionary of dictionaries.

```
from records import records
```

Given a course name, convert the filename, then use the converted value to lookup a specific swimming world record.

```
records["LC Men"][event_lookup("Darius-13-100m-Fly.txt")]
```

```
'49.45'
```

And there it is: the current men's long course 100m world record time for the Fly. Your code works!

Getting to the webapp integration

You now have chunks of code which you can start to integrate with the Coach's webapp. Specifically, you're going to add code to the `swimclub.py` module, as that's where your bar chart producing code resides.



Yes, that would be a good start.

Your `records` dictionary of dictionaries needs to be accessible from within the `swimclub` module's code, so importation is a must.

The `event_lookup` function, the code to which resides in your `WorldRecords.ipynb` notebook, can be copied verbatim into the

`swimclub.py` module.

Let's perform both of these edits now.

Leave the `WorldRecords.ipynb` notebook opened in VS Code, then click on the *Explorer* icon to see the list of files in your `webapp` folder. Find and double-click on the `swimclub.py` filename to load your module into a VS Code edit window.



All that's needed: an edit and a copy'n'paste…

You need to import the `records` dictionary of dictionaries from the `records.py` module into your `swimclub.py` module's code. The **from** variant of the **import** statement does the trick:

```
import statistics

import hfpy_utils
from records import records

CHARTS = "charts/"
```

You can place your import statement anywhere in your file (just so long as it appears *before* you use the "records" dictionary). We decided to add it below the list of "import" statements at the top of our module's file.

When it comes to positioning the `event_lookup` function within your module, as with the above **import** statement, you can position the function most *anywhere* in the file. We decided to add the `event_lookup` into the `produce_bar_chart` function, *nesting* it within the function. This has the effect of limiting `event_lookup`'s visibility to being only callable from `produce_bar_chart`:

Nesting a function within a function is OK with Python.

```
def produce_bar_chart(fn, location=CHARTS):
    """Given the name of a swimmer's file, produce a HTML/SVG-based bar chart, saving
    it to the "location" folder. Return the path to the bar chart file."""

    def event_lookup(event):
        """Given an event descriptor (the name of a swimmer's file), convert
        the descriptor into a lookup key which can be used with the "records"
        dictonary."""
        conversions = {
            "Free": "freestyle",
            "Back": "backstroke",
            "Breast": "breaststroke",
            "Fly": "butterfly",
            "IM": "individual medley",
        }

        *_, distance, stroke = event.removesuffix(".txt").split("-")

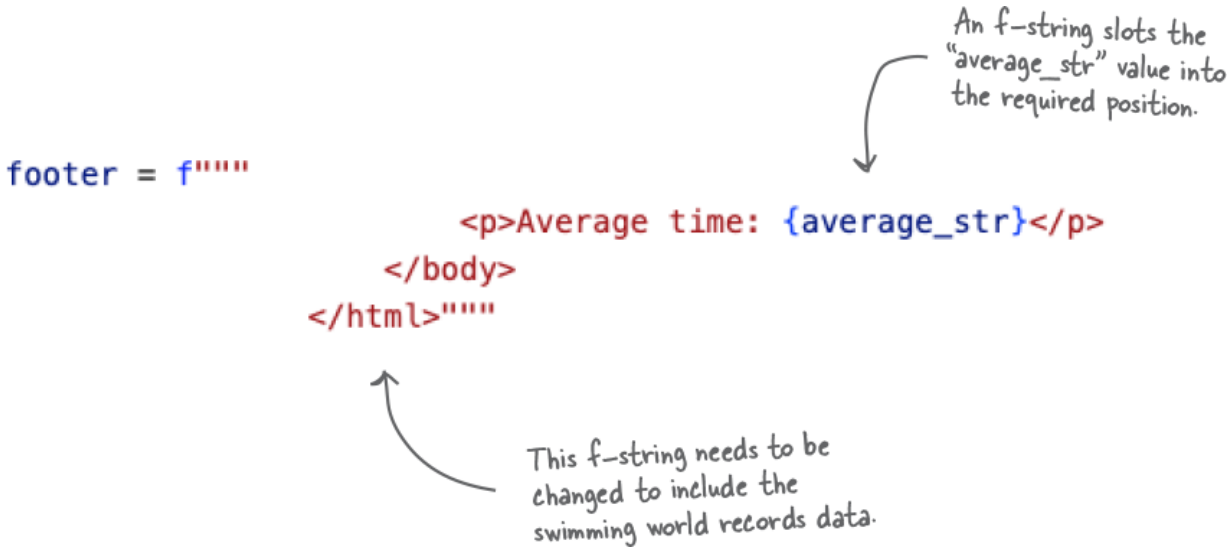
        return f"{distance} {conversions[stroke]}"

    swimmer, age, distance, stroke, _, average_str, times, converts = get_swim_data(fn)
    from_max = max(converts)
    times.reverse()
```

The "event_lookup" code is added to, and indented within, the "produce_bar_chart" function. Note, too, the addition of a triple-quoted function comment which we've added to the "event_lookup" function (something which is missing from the function's code within our notebook).

Adding the world records to your bar chart

If you skip to near the bottom of the `produce_bar_chart` function, you'll find the following code which adds the HTML footer to the end of your generated bar chart:



Near the bottom of your `WorldRecords.ipynb` notebook you'll find this code which, given a swimmer's filename, displays the four world record times:



An adaptation of the above **for** loop can be used to generate another chunk of HTML which contains the data the Coach wants you to add to each bar chart.

EXERCISE



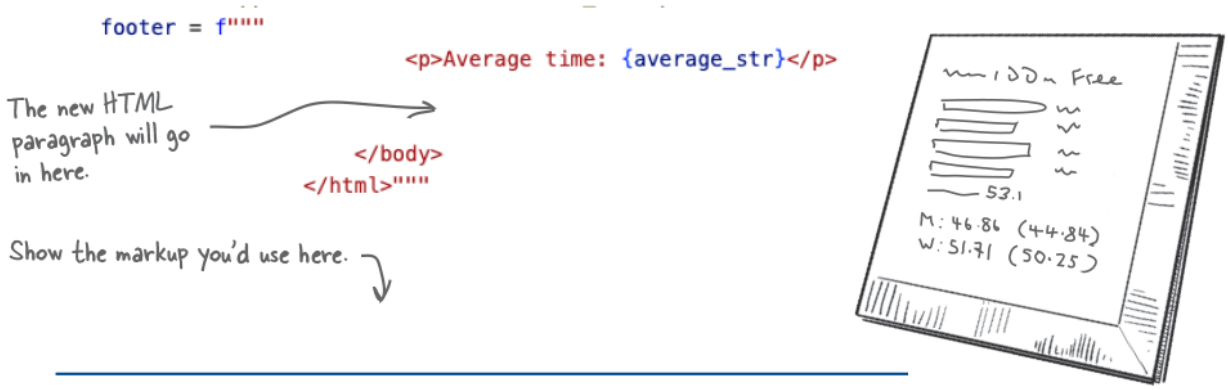
The code shown below extracts each of the world record times for the *100m butterfly*. Let's adjust this code to accept any swimmer's file-name as stored in the `fn` variable (available to you from within the `produce_bar_chart` function). Also, rather than printing the world records on screen, adjust this loop code to store the four times in a new list variable called `times`.

```
COURSES = ("LC Men", "LC Women", "SC Men", "SC W

for course in COURSES:
    print(records[course] [event_lookup("Darius-
```

Write in the code you came up with in this space:

With the `times` list now available to you, add additional HTML markup to the `footer` f-string. In the space below, add a second HTML paragraph which embeds the four world record times from the `times` list. Be sure to match up the order of the values against those from the Coach's napkin spec.



EXERCISE SOLUTION



The code shown below extracts each of the world record times for the *100m butterfly*. Your first task in this *Exercise* was to adjust this code to accept any swimmer's filename as stored in the `fn` variable (available to you from within the `produce_bar_chart` function). Also, rather than printing the world records on screen, you were to adjust this loop code to store the four times in a new list variable called `times`.

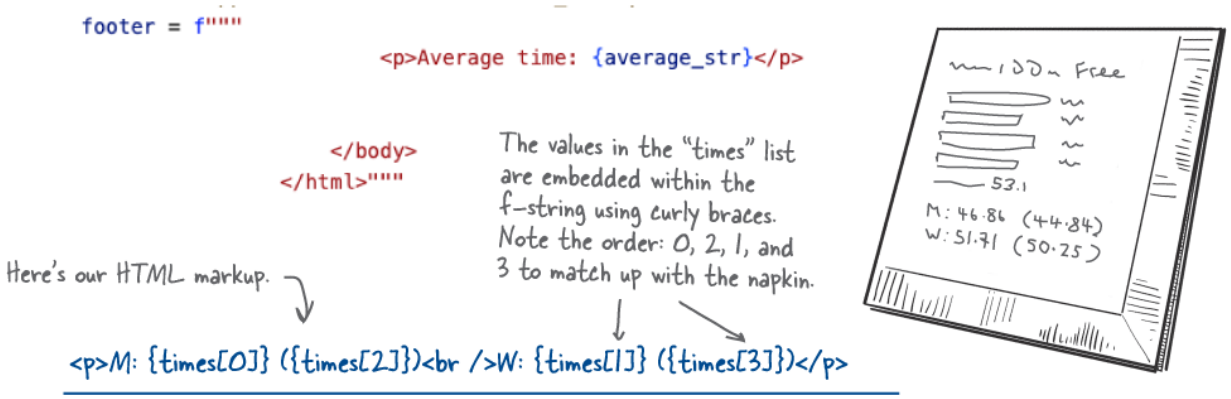
```
COURSES = ("LC Men", "LC Women", "SC Men", "SC W

for course in COURSES:
    print(records[course] [event_lookup("Darius-
```

Here's the code we came up with:



With the `times` list now available to you, you were then to add additional HTML markup to the `footer` f-string. You were to add a second HTML paragraph which embeds the four world record values from the `times` list, being sure to match up the order of the values against those from the Coach's napkin spec.



TEST DRIVE



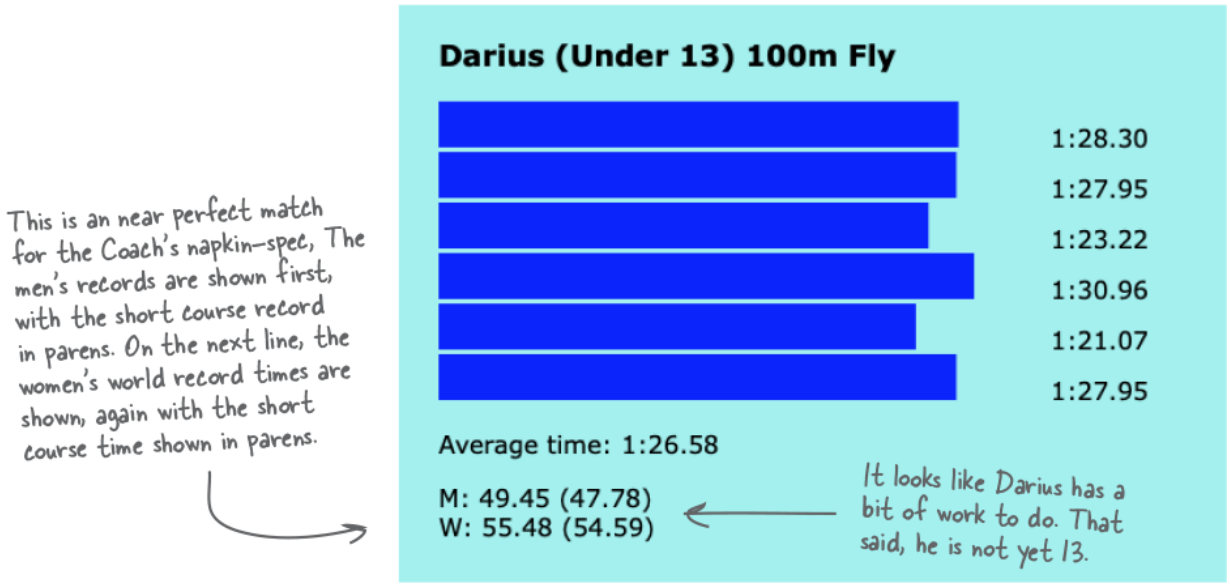
Here's the code snippet near the bottom of the `produce_bar_chart` function (in `swimclub.py`) with the changes from your most-recent *Exercise* applied:

```
                                                    .
                                                    .
                                                    .
COURSES = ("LC Men", "LC Women","SC Men", "SC Wor
times = []
for course in COURSES:
    times.append{records[course][event_lookup[fn]
footer = f"""
                        <p>Average time: {average_
                        <p>M: {times[0]} ({times[2
                    </body>
                </html>"""
                                                    .
                                                    .
                                                    .
```

To test this new functionality, open your webapp's `app.py` file in VS Code, then select **Run** then **Run Without Debugging** from the menu. Your webapp should start in VS Code's terminal window.

Go ahead and surf to `http://127.0.0.1:5000` using your favorite web browser, then click on the **swimmer** link. Select `Darius` from the drop-down menu, then select `100-Fly` from the next drop-down menu.

Assuming all is OK with your latest code, the stars align, birds start to sing, the sun shines, and all is well with the world. And the current *100m butterfly* world records appear at the bottom of your generated bar chart:



Is your latest version of the webapp ready?

With the requirement identified on the Coach's napkin-spec built, the updated webapp feels ready for prime time. You can confidently confirm you've completed the second unit of work by adding that final check-mark:



4 **Convert the record data into a dictionary of swimming world records.** ✔

5 **Use the dictionary to add world records to the bottom of each bar chart.** ✔

But… are you really done?

Your updated webapp is running on your local computer. If you wanted to, you could deploy this latest version of your code - the updated `swimclub.py` module as well as your new `records.py` module - to *PythonAnywhere*.

But, let's not do that just yet, even though (as you can imagine) the Coach is looking forward to getting his hands on your new functionality.



It's not that the code doesn't work.

As has been demonstrated, the latest version of your webapp is working, and there's a very high likelihood that your latest code will run *unchanged* on PythonAnywhere. But, it's not the code which is causing the hesitation here: *it's the data.*

Specifically, what happens if the source data *changes*? What happens when a swimmer breaks a world record and the *Wikipedia* page changes? What then?

Let's give this some thought…

Cubicle Conversation



Alex: When the data changes, all we have to do is re-run our notebook to regenerate `records.py`, right?

Mara: Yes, but how will we know when to do this?

Alex: The Coach will tell us, surely…

Sam: Yes, maybe, but he'll likely not be happy that his world record data is out-of-date.

Mara: Let's not upset the Coach if we can avoid it. I'd rather the Coach didn't notice when his data goes stale.

Alex: Can't we take turns running the notebook once a day?

Sam: Sure… just so long as *you* do it every weekend, and during the holidays. Thanks for volunteering.

Alex: …ehhhhhh…

Mara: [laughs] Whatever we do, it has to be an automated process.

Alex: We could regenerate `records.py` every time the webapp is visited, that way we're guaranteed not to miss anything.

Sam: The might be wasteful of resources, especially if the Coach decides to share his webapp with all his swimmers. We'd potentially be

regenerating multiple times per day which, to me anyway, seems excessive.

Mara: OK. We're agreed the `records.py` module has to be regenerated. Questions remain as to *how* we do this and *when*.

Sam: We already have the regenerating code in the `WorldRecords.ipynb` notebook, so it's really a matter of being able to run that code independently of *Jupyter Notebook*, right?

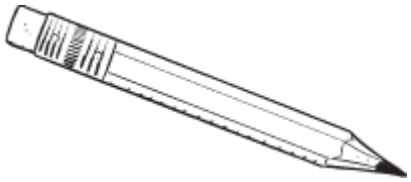
Mara: Yes.

Alex: Is that hard to do?

Mara: No. Let's turn the regenerating code into a standalone Python program, a *utility*, which we can execute at any time, then we can worry about executing it to some sort of schedule.

Sam: Sound good to me…

SHARPEN YOUR PENCIL



Here's all the code from `WorldRecords.ipynb` which you created in order to get to the point where you created the `records.py` module. All of the code your wrote is included here, including all of your experimentations while you tried to work out the code you actually needed. Grab your pencil and draw a line through the code which you do **not** intend to use with your new utility.

Note: the ↵ symbol indicates the line of code was too wide to fit in the available space, but appears as a single line of code in your notebook.

```
%pip install gazpacho

URL = "https://en.wikipedia.org/wiki/List_of_
      world_records_in_swimming"                ↵

import gazpacho

html = gazpacho.get(URL)
```

```
len(html)

html[:500]

html[-500:]

from_where = html.find("<table")
html[from_where : from_where + 500]

soup = gazpacho.Soup(html)

type(soup)

print(dir(soup))

tables = soup.find("table")

type(tables)

len(tables)

type(tables[0])

type(tables[-1])

rows = tables[0].find("tr")

type(rows)
```

```
len(rows)

rows[0]

for n, table in enumerate(tables):
    rows = table.find("tr")
    print(f"{n} -> {len(rows)} rows")

for n, table in enumerate(tables):
    rows = table.find("tr")
    cols = rows[-1].find("td")
    print(f"{n} -> {len(rows)} rows, {len(cols)}
       columns")

tables[9].find("tr")[-1].find("td")

type(tables[9].find("tr")[-1].find("td"))

len(tables[9].find("tr")[-1].find("td"))

for n, table in enumerate(tables):
    rows = table.find("tr", mode="all")
    cols = rows[-1].find("td", mode="all")
    print(f"{n} -> {len(rows)} rows, {len(cols)}
        columns")

table = tables[0]
```

```
for row in table.find("tr")[1:]:
    columns = row.find("td")
    event = columns[0].text
    time = columns[1].text
    print(f"{event} -> {time}")

RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC
        Women")

list(zip(RECORDS, COURSES))

RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC
        Women")

for table, course in zip(RECORDS, COURSES):
    print(f"{course}:")
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()

records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
```

```
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time

records.keys()

records["SC Women"]["100m backstroke"]

records["LC Men"]["50m freestyle"]

print(records["LC Men"]["100m butterfly"])
print(records["LC Women"]["100m butterfly"])
print(records["SC Men"]["100m butterfly"])
print(records["SC Women"]["100m butterfly"])

conversions = {
    "Free": "freestyle",
    "Back": "backstroke",
    "Breast": "breaststroke",
    "Fly": "butterfly",
    "IM": "individual medley",
}

event = "Mike-15-100m-Fly.txt"
*_, distance, stroke = event.removesuffix(".
        txt").split("-")
```

```
lookup = f"{distance} {conversions[stroke]}"
lookup

for course in records.keys():
    print(f"{records[course][lookup]}")

import pprint

with open("records.py", "w", encoding=
                "utf-8") as rf:
     print("records = ", end="", file=rf)
     pprint.pprint(records, stream=rf)

def event_lookup(event):

    conversions = {
        "Free": "freestyle",
        "Back": "backstroke",
        "Breast": "breaststroke",
        "Fly": "butterfly",
        "IM": "individual medley",
    }

    *_, distance, stroke = event.removesuffix(".
        txt").split("-")

    return f"{distance} {conversions[stroke]}"
```

```
print(event_lookup("Darius-13-100m-Back.txt"))
print(event_lookup("Darius-13-100m-Breast.txt"))
print(event_lookup("Darius-13-100m-Fly.txt"))
print(event_lookup("Darius-13-100m-IM.txt"))

from records import records

records["LC Men"][event_lookup("Darius-13-100m-
        Fly.txt")]

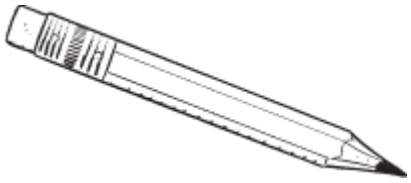
COURSES = ("LC Men", "LC Women", "SC Men", "SC
        Women")

for course in COURSES:
    print(records[course][event_lookup("Darius-
        13-100m-Fly.txt")])

times = []
for course in COURSES:
    times.append(records[course][event_
        lookup("Darius-13-100m-Fly.txt")])
times


f"<p>M: {times[0]} ({times[2]})<br />W:
        {times[1]} ({times[3]})</p>"
```

SHARPEN YOUR PENCIL SOLUTION



Given all the code from `WorldRecords.ipynb`, you were asked to grab your pencil and draw a line through the code which you did **not** intend to use with your new utility.

Note: the ↵ symbol indicates the line of code was too wide to fit in the available space, but appeared as a single line of code in your notebook.

```
%pip install gazpacho

URL = "https://en.wikipedia.org/wiki/List_of_
       world_records_in_swimming"

import gazpacho

html = gazpacho.get(URL)

len(html)

html[:500]

html[-500:]

from_where = html.find("<table")
html[from_where : from_where + 500]

soup = gazpacho.Soup(html)

type(soup)

print(dir(soup))

tables = soup.find("table")

type(tables)

len(tables)

type(tables[0])

type(tables[-1])

rows = tables[0].find("tr")

type(rows)

len(rows)

rows[0]

for n, table in enumerate(tables):
    rows = table.find("tr")
    print(f"{n} -> {len(rows)} rows")
```

Grab the raw HTML from the web page.

Convert the HTML to soup.

Create a list of <table> tags.

```
for n, table in enumerate(tables):
    rows = table.find("tr")
    cols = rows[-1].find("td")
    print(f"{n} -> {len(rows)} rows, {len(cols)}
        columns")

tables[9].find("tr")[-1].find("td")

type(tables[9].find("tr")[-1].find("td"))

len(tables[9].find("tr")[-1].find("td"))

for n, table in enumerate(tables):
    rows = table.find("tr", mode="all")
    cols = rows[-1].find("td", mode="all")
    print(f"{n} -> {len(rows)} rows, {len(cols)}
        columns")

table = tables[0]
for row in table.find("tr")[1:]:
    columns = row.find("td")
    event = columns[0].text
    time = columns[1].text
    print(f"{event} => {time}")

RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC
        Women")

list(zip(RECORDS, COURSES))

RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC
        Women")

for table, course in zip(RECORDS, COURSES):
    print(f"{course}:")
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()
```

Define two constant tuples.

Build the "records" dictionary of dictionaries from the data in each of the four identified swimming world record tables.

```
records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

```
records.keys()

records["SC Women"]["100m backstroke"]

records["LC Men"]["50m freestyle"]

print(records["LC Men"]["100m butterfly"])
print(records["LC Women"]["100m butterfly"])
print(records["SC Men"]["100m butterfly"])
print(records["SC Women"]["100m butterfly"])

conversions = {
    "Free": "freestyle",
    "Back": "backstroke",
    "Breast": "breaststroke",
    "Fly": "butterfly",
    "IM": "individual medley",
}

event = "Mike 15 100m Fly.txt"
*_, distance, stroke = event.removesuffix(".
        txt").split("-")
lookup = f"{distance} {conversions[stroke]}"
lookup

for course in records.keys():
    print(f"{records[course][lookup]}")
```

```
import pprint

with open("records.py", "w", encoding=
                "utf-8") as rf:
    print("records = ", end="", file=rf)
    pprint.pprint(records, stream=rf)
```

Save the dictionary of dictionaries to the "records.py" module.

The majority of your notebook's code isn't needed, as it was created as a result of experimentation (or was used elsewhere).

```
def event_lookup(event):

    conversions = {
        "Free": "freestyle",
        "Back": "backstroke",
        "Breast": "breaststroke",
        "Fly": "butterfly",
        "IM": "individual medley",
    }

    *_, distance, stroke = event.removesuffix(".
        txt").split("-")

    return f"{distance} {conversions[stroke]}"

print(event_lookup("Darius-13-100m-Back.txt"))
print(event_lookup("Darius-13-100m-Breast.txt"))
print(event_lookup("Darius-13-100m-Fly.txt"))
print(event_lookup("Darius-13-100m-IM.txt"))

from records import records

records["LC Men"][event_lookup("Darius-13-100m-
        Fly.txt")]

COURSES = ("LC Men", "LC Women", "SC Men", "SC
        Women")

for course in COURSES:
    print(records[course][event_lookup("Darius-
        13-100m-Fly.txt")])

times = []
for course in COURSES:
    times.append(records[course][event_
        lookup("Darius-13-100m-Fly.txt")])
times

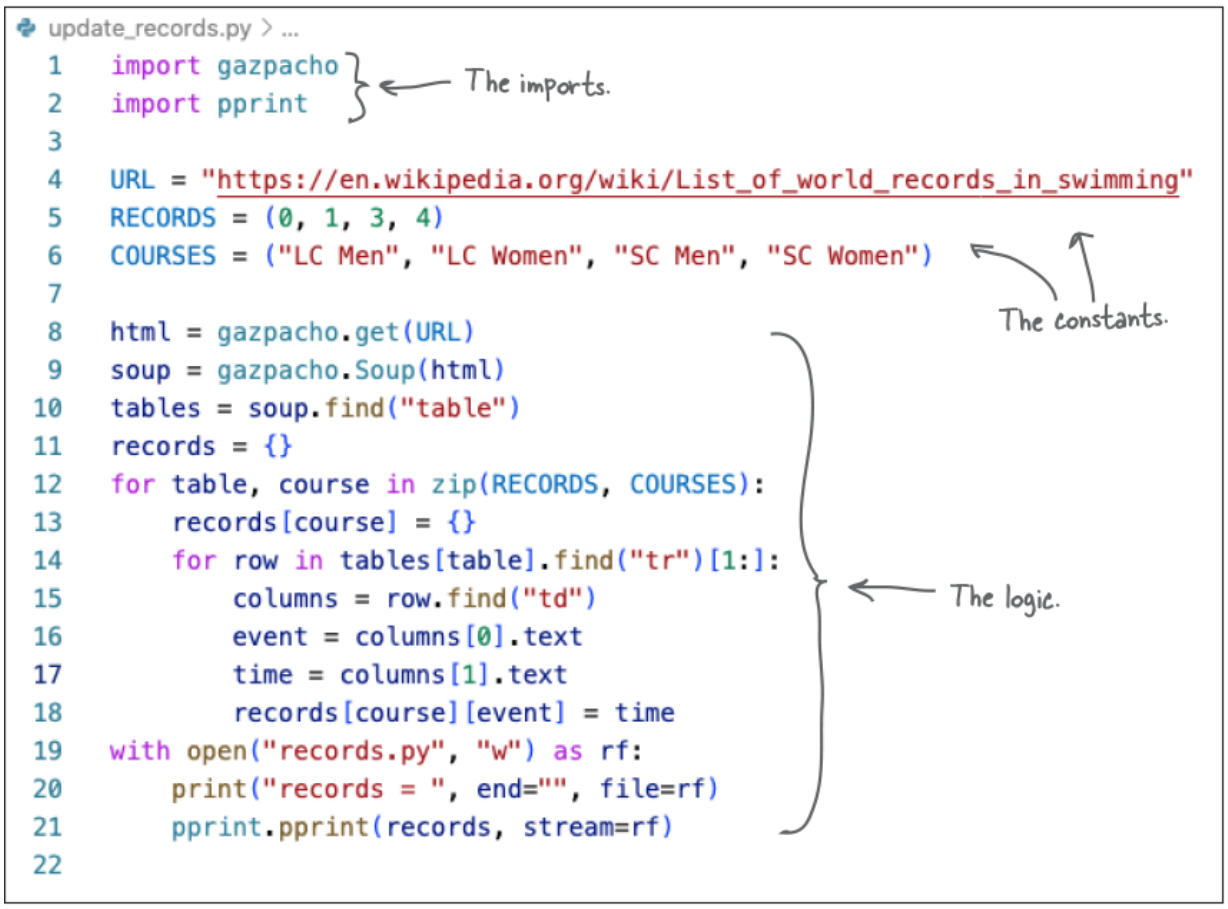

f"<p>M: {times[0]} ({times[2]})<br />W:
        {times[1]} ({times[3]})</p>"
```

Most of the code in this column is already integrated into the "swimclub.py" module.

TEST DRIVE



Create a new text file in VS Code (within your `webapp` folder), then rename the file to `update_records.py`. Copy the code from the last *Exercise* which performs the update into your new file. We did this, then rearranged things (slightly) resulting in code which looks like this:



You can, of course, execute this code directly from within VS Code by selecting **Run** then **Run Without Debugging** from the menu system. If you do this now, and a terminal window opens at the bottom of your screen, then the above code runs. Although there is no visible output, the code has connected to *Wikipedia*, downloaded the correct HTML page, used `gazpacho` to parse and extract the `<table>` tags, which are then converted into the `records` dictionary of dictionaries. The final three lines of code save a pretty-printed copy of `records` to the `records.py` module.

No matter which way you look at it, this not bad going for less than 20 lines of code.



Yes, PythonAnywhere can do this!

Recall that the Coach's webapp is deployed on *PythonAnywhere*, not on your development machine. You'll want to run your `update_records.py` utility on the *PythonAnywhere* cloud to ensure the displayed swimming world records are not stale. Perhaps once per day?

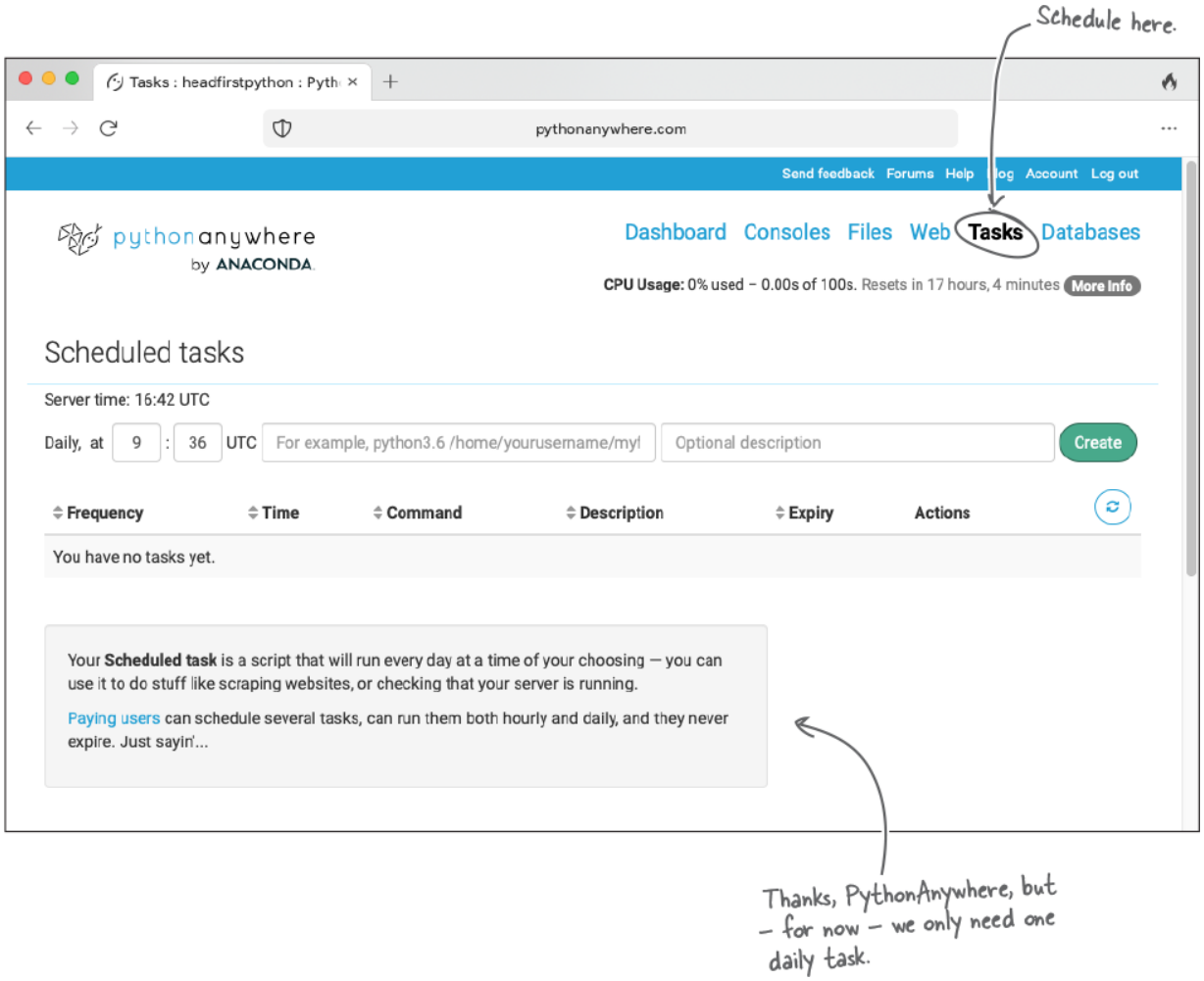
Question is… how?

PythonAnywhere has you covered…

Log into your *PythonAnywhere* account, then click the **Tasks** tab.



The displayed form lets you configure a single daily scheduled task (which is all you get with your *Beginners* account):



Don't fill-in this form just yet, as you've yet to update the currently deployed webapp with your latest webapp code.

Let's do that first, then you can return to this form and schedule your daily update.

You need to upload your utility code, too

If you upload your `update_records.py` module *as is*, a subtle *gotcha* awaits you.

As written, your utility code creates the `records.py` module in the folder the utility is executed from. This is no biggie when running the utility on *your* computer, as you're in control and can ensure the utility is always executed from the correct folder. However, this is not the case on *PythonAnywhere*, were you have to be a little more precise.

Specifically, you need to adjust your code to reference the *exact* folder location to use. Doing so isn't hard, and is only two small edits to your `update_records.py` code:



These two edits adjust `update_records.py` to run on *PythonAnywhere*, which is great.

Just be aware that your utility won't run on your computer as the folder referenced by `WHERE` likely doesn't exist.

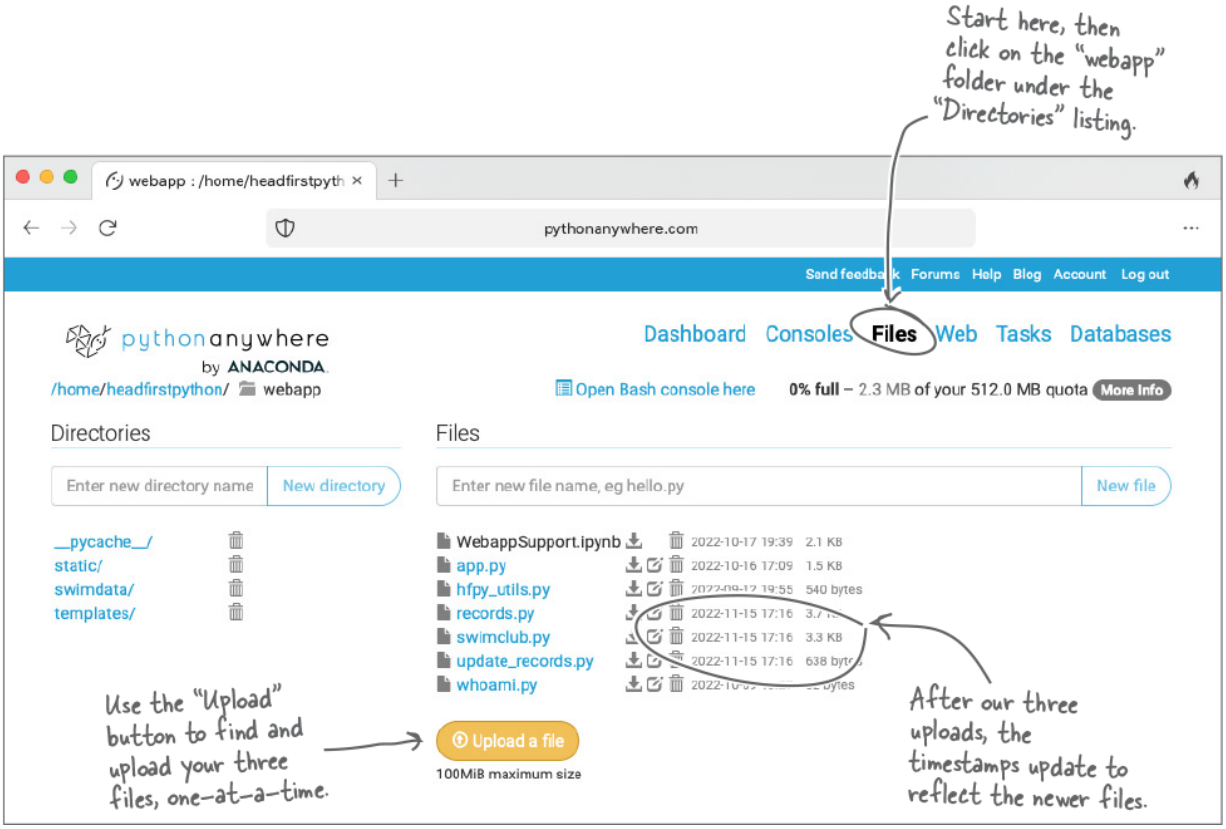
Deploy your latest webapp to PythonAnywhere

There is a temptation to *zip* your entire `webapp` folder for uploading and deploying in one go. If you do this, **all** the files in your `webapp` folder are copied to *PythonAnywhere*, including all of your notebooks.



If your fine with this, feel free to repeat the required steps from Chapter 7, *Deployment*, as needed.

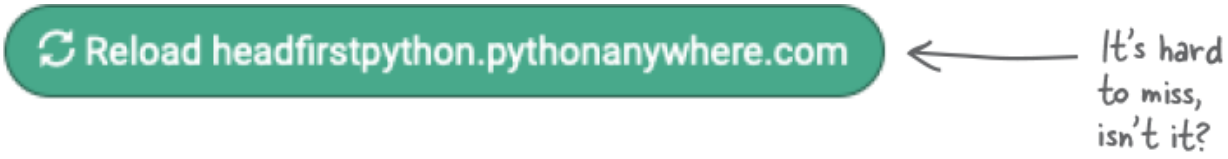
If you consider what you've done in this and the previous chapter, the only file which has *changed* in your webapp is `swimclub.py`. Additionally, two *new* files are now in existence: `records.py` and `update_records.py`. If you aren't into the zip-upload-unzip triple from Chapter 7, let's use the PythonAnywhere **Files** tab to upload these three files individually:



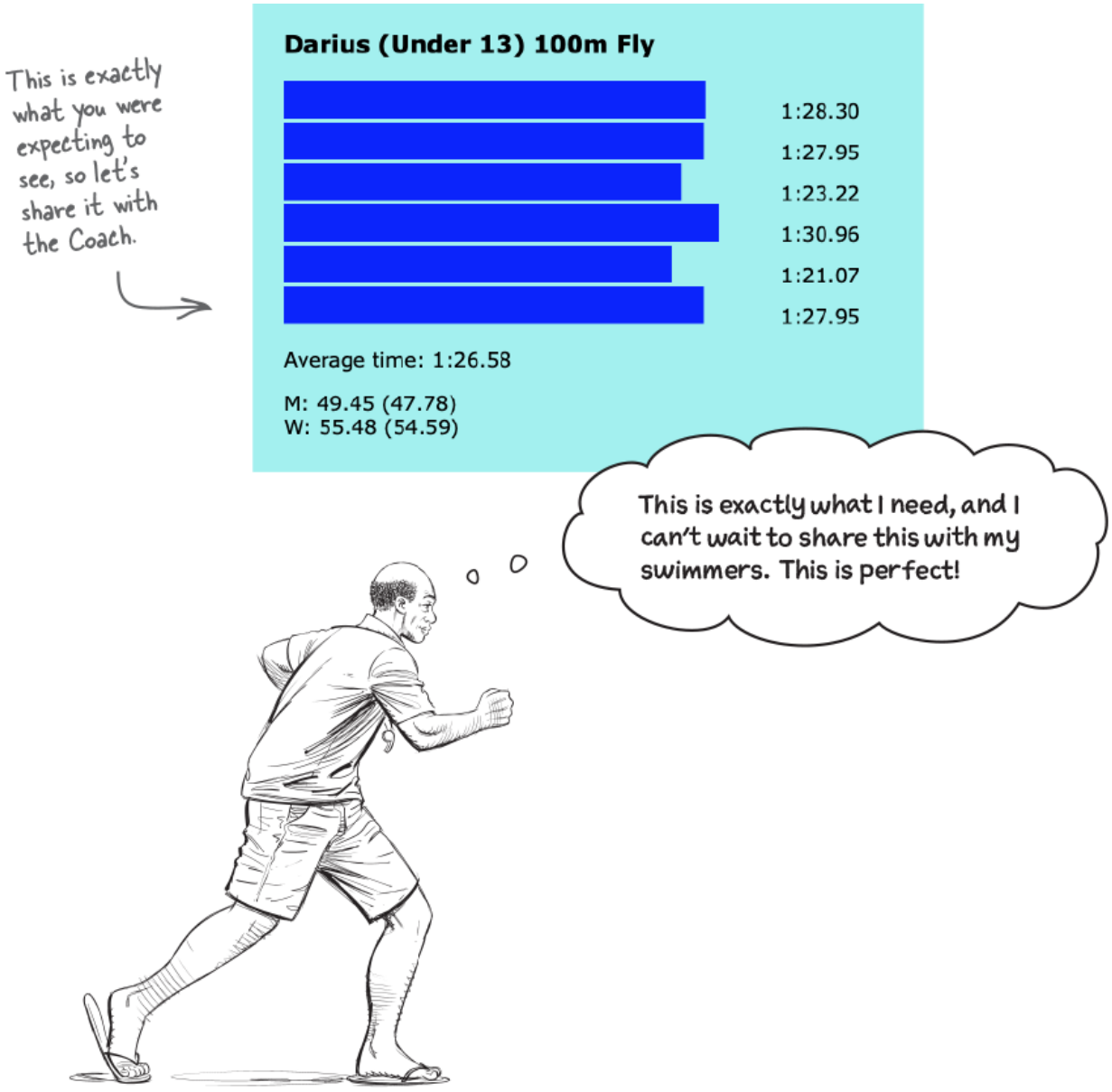
It's time for a click on the big green button!

Tell PythonAnywhere to run your latest code

Access the **Web** tab from the *PythonAnywhere* dashboard, then click on the big green button to deploy your updated webapp:



Clicking on the blue link on your *PythonAnywhere* Web tab brings up the latest version of your webapp in your browser. As expected it works on *PythonAnywhere* exactly as on your development computer:



That's great to hear.

While the Coach runs off to spread the news about your updated web-app (and while the pressure is *off*), let's stay logged into *PythonAnywhere* and ensure the `update_records.py` utility is good to go.

Test your utilities before cloud deployment

The good folk at *PythonAnywhere* work hard to ensure the Python environment they make available to you is as complete as possible. Here the word “complete” translates to providing recent releases of Python (including the PSL) as well as a large collection of popular third-party modules from *PyPI*.



But not *every* module is included. Look what happens when you run your `update_records.py` utility at a *PythonAnywhere* console. To do this: return to your dashboard, then click on the **Consoles** tab, then click on `Bash` to launch a terminal:



Although it's not possible to install `gazpacho` into *PythonAnywhere*'s global environment, you can install it *just for you* by appending `--user` to **pip**:

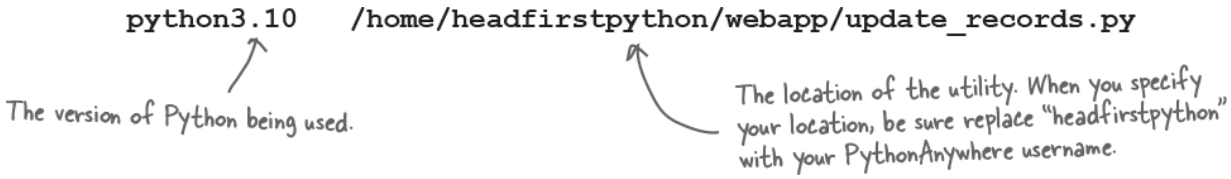


Let's run your task daily at 1:00am

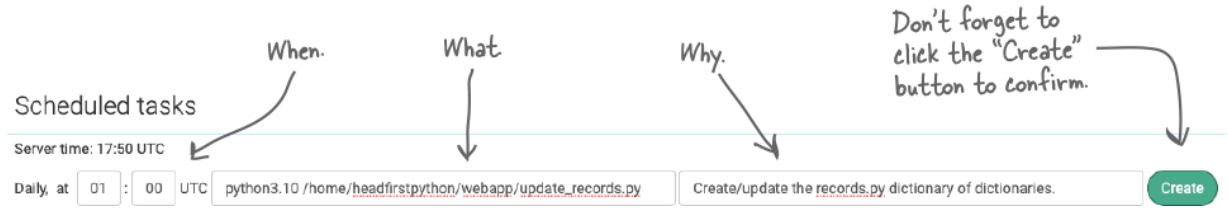
Now that `update_records.py` runs without error, you can return to the *PythonAnywhere* **Tasks** tabs to schedule it to run daily.



You specify when you want your task to run by specifying the *hour* and *minute* values. You also specify the command to execute, specifying the version of Python to use as well as the complete path to your utility code. We used this command-line:



If you'd like, provide an optional description for the task, before clicking on **Create**:



Your task is created, running daily for approximately four weeks (when it then expires). Make a note to return to the **Tasks** tab before

your task expires so you can click on the **Extend expiry** button to keep your Task active for a little while longer.

As you've probably guessed by now, upgrading to a paid *PythonAnywhere* account lets you specify more than one scheduled task which never expires. But, for now, this arrangement works well enough for your webapp:

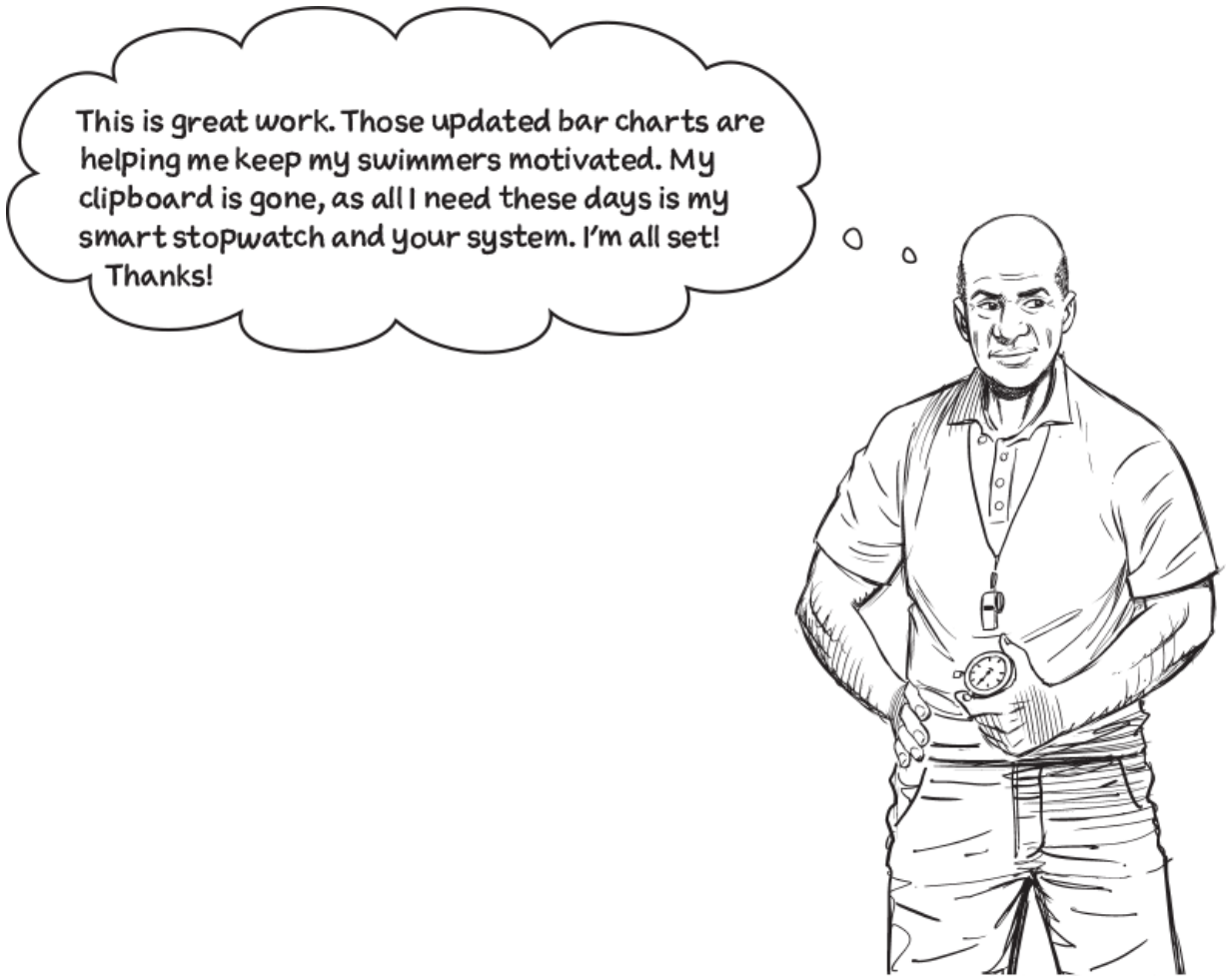


With your task scheduled, you can logout of *PythonAnywhere*. The latest release of your webapp is deployed, and your utility is scheduled to run on a daily basis.

Be sure to check back tomorrow to confirm the `records.py` module refreshed.

NOTE

All you need to do is check the timestamp on the "Files" tab.



There's nothing quite like a happy user.

Your Python chops are expanding, and you're now leveraging what you know to do good work.

Even though this is Chapter 9, recall that we started counting from zero, so you've now ten chapters under your belt. When you're ready, dive into the next chapter after checking out the chatper review before giving the latest crossword the once over.

See you in the next chapter!

CHAPTER REVIEW BULLET POINTS

- All of Python's four built-in data structures – **list**, **dictionary**, **tuple**, and **set** – are great on their own, but they really shine when used together or **nested** within themselves. In this chapter a classic arrangement, the **dictionary of dictionaries**, demonstrated how to build-out and use a hierarchical data structure.
- Another classic use-case was demonstrated when a dictionary provided a small **lookup table**, making conversions from one value to another a snap.
- The **open** BIF, used in earlier chapters to read data from files, became a data structure writing machine thanks to it's **write mode** ("w"). You used this BIF when you wrote code which wrote code, creating a module as you did so.
- With your new module created, you used the **from** version of the **import** statement to use the data defined therein.
- You added a **nested function** (called `event_lookup`) to your existing `produce_bar_chart` function, which restricted the **scope** of the nested function to within the code of the enclosing function.
- They may not have had a starring role in this chapter, but **f-strings**, when the did appear, stole the show.
- You learned how to create a small **utility** program which you then executed outside of VS Code and its Jupyter Notebook en-

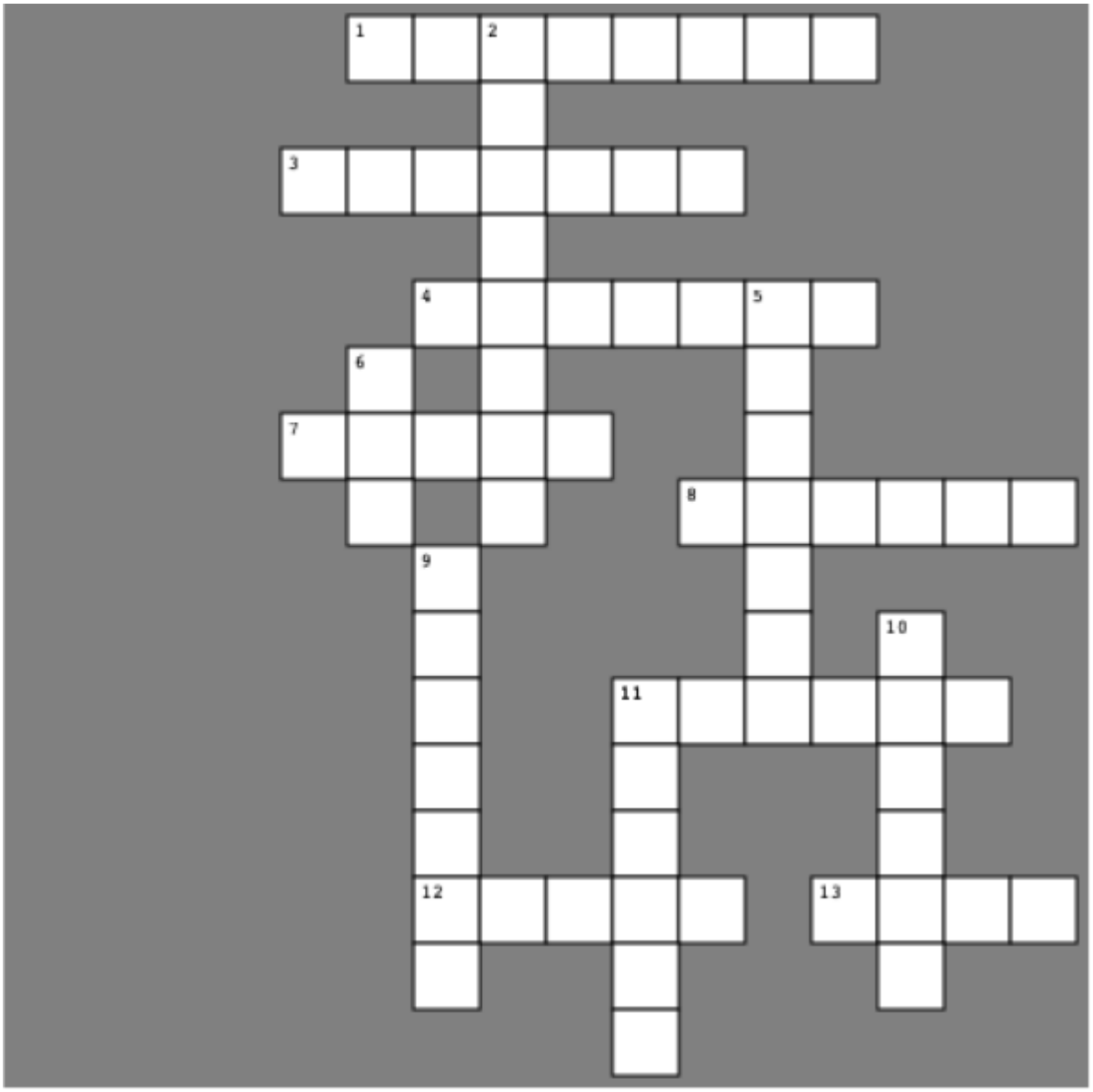
vironment. You even arranged for your utility to execute on a daily basis on *PythonAnywhere*.

- You've made a mental note to log into *PythonAnywhere* tomorrow to confirm your utility is working as expected.

The World Records Crossword



You'll find all the answers to the clues in this chapter (with solutions waiting on the next page).



Across

1. Set this argument to `utf-8` when creating a file (just to be safe).

3. Rhymes with strangle, but applies to data.

4. Transform from one value to another.

7. Among other things, this dashboard tab displays a list of folders.

8. You can look down or _____.

11. A function defined inside another.

12. This dashboard tab can schedule a daily execution.

13. What this chapter's all about.

Down

2. This dashboard tab gives you access to the Bash shell.

5. The name of this chapter's dictionary of dictionaries.

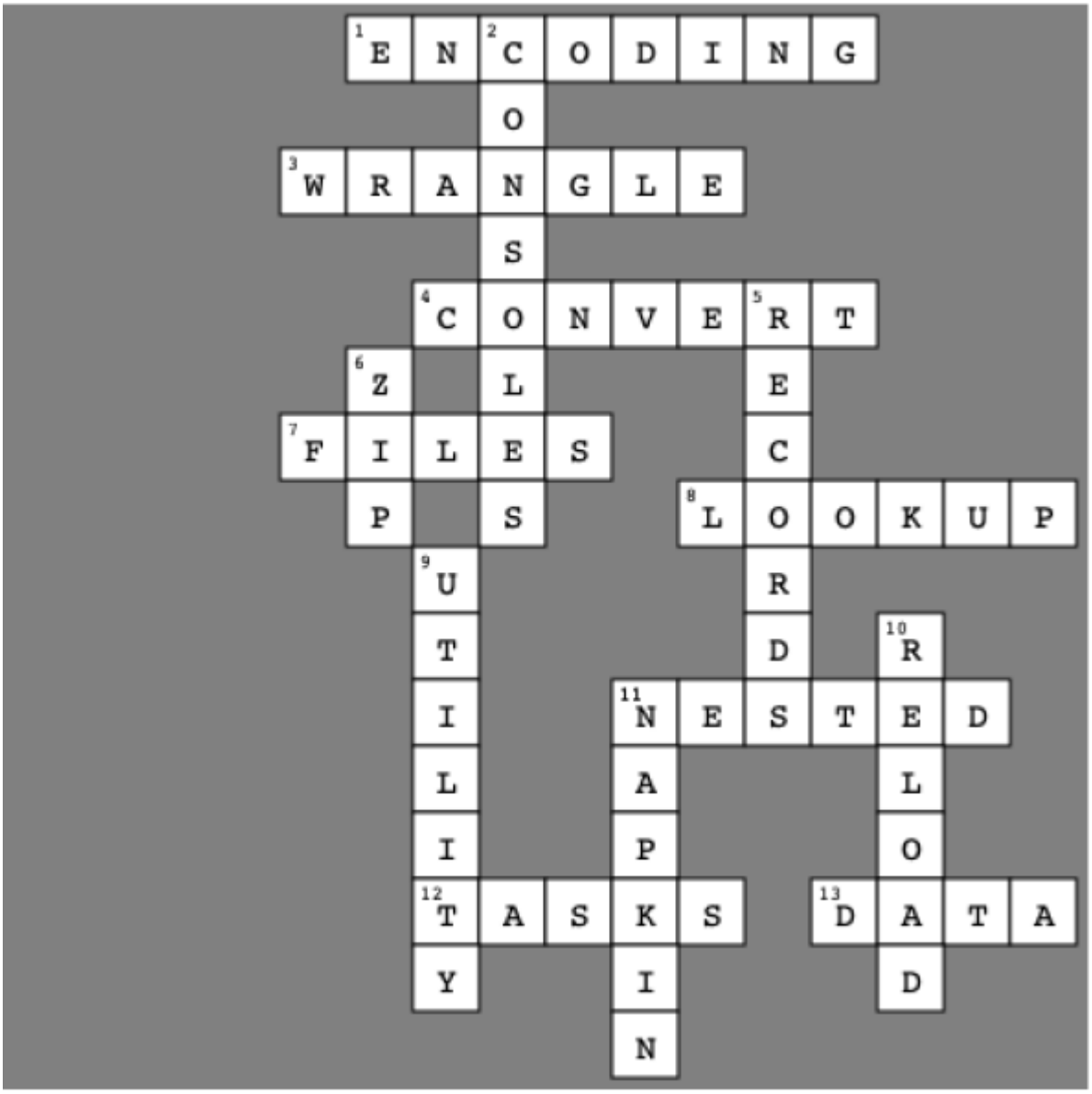
6. This BIF dovetails two lists.

9. The `update_records.py` program is one of these.

10. This big green button is hard to miss.

11. Useful when wiping your mouth and for stretching application requirements.

The World Records Crossword Solution


1 E N 2 C O D I N G
O
3 W R A N G L E
S
4 C O N V E 5 R T
6 Z L E
7 F I L E S C
P S 8 L O O K U P
9 U R
T D 10 R
I 11 N E S T E D
L A L
I P O
12 T A S K S 13 D A T A
Y I D
N

Across

1. Set this argument to `utf-8` when creating a file (just to be safe).

3. Rhymes with strangle, but applies to data.

4. Transform from one value to another.

7. Among other things, this dashboard tab displays a list of folders.

8. You can look down or _____.

11. A function defined inside another.

12. This dashboard tab can schedule a daily execution.

13. What this chapter's all about.

Down

2. This dashboard tab gives you access to the Bash shell.

5. The name of this chapter's dictionary of dictionaries.

6. This BIF dovetails two lists.

9. The `update_records.py` program is one of these.

10. This big green button is hard to miss.

11. Useful when wiping your mouth and for stretching application requirements.

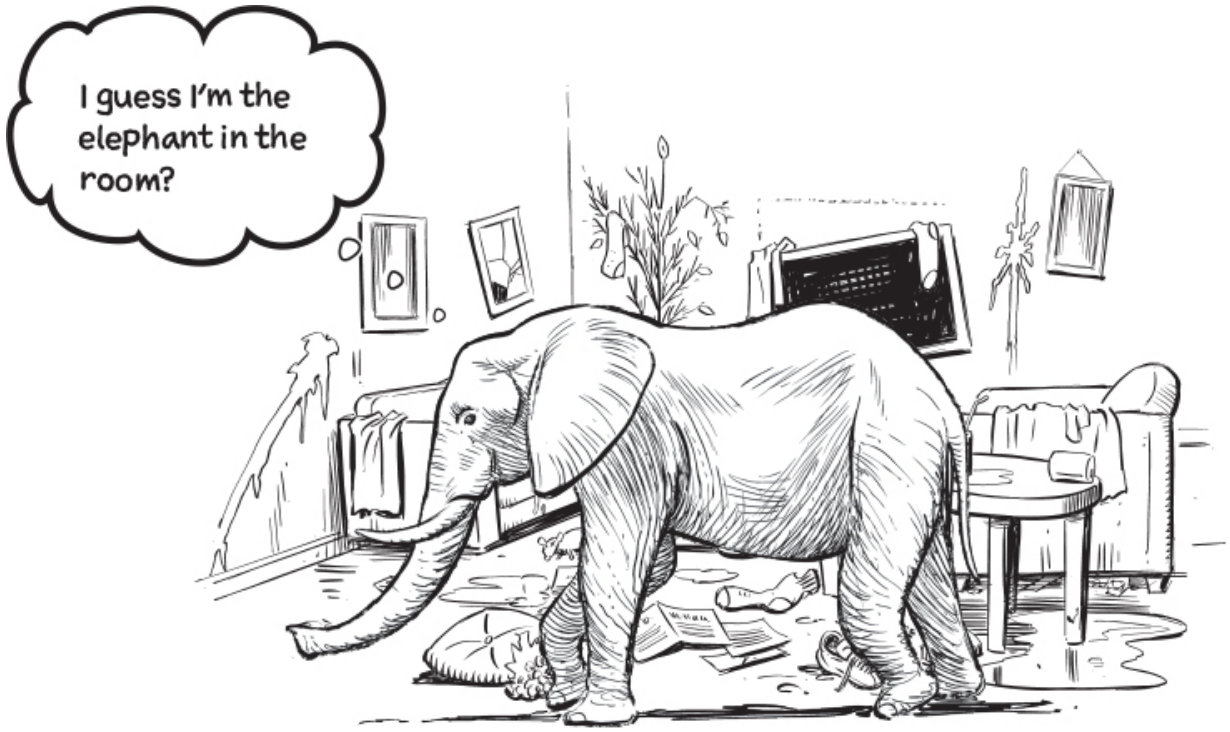
Chapter 10. 9½ Working with ~~elephants~~ Dataframes: *Tabular data*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 9½ of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



Sometimes it's as if all the world's data wants to be in a table.

Tabular data is *everywhere*. The swimming world records from the previous chapter is **tabular** data. If you're old enough to remember phone books, the data is tabular. Bank statements, invoices, spreadsheets: you guessed it, all tabular data. In this *short* chapter, you'll learn a bit about one of the most popular tabular data analysis libraries in Python: **pandas**. You'll only skim the surface of what pandas can do, but you'll learn enough to be able to exploit the most-used pandas data structure, the **dataframe**, when you're next faced with processing a chunk of tabular data.

The elephant in the room… or is it a panda?

The *Head First Coders* sent over their top code reviewer who took a quick look at the code you created to scrape the swimming world records from *Wikipedia*. They only had one thing to say…





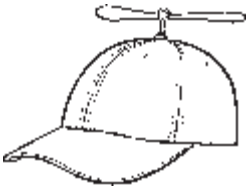
It's not that our code doesn't work. It does.

And the Coach is more than happy, and rightly so.

When we point this out, Sanjita continues to shake her head. When we further point out that we've never used `pandas`, Sanjita's frown turns into a smile, and she says: *"Let me show you."*

Sanjita announces she'll drive, grabs the keyboard, then suggests we pay close attention. Jeez…

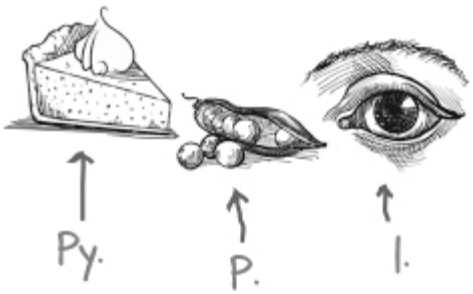
GEEK NOTE



Hang around enough Data Scientists and you'll likely hear the word "pandas" *at least once*. It's the name given to the premier third-party data manipulation library within the Python ecosystem, and is hugely popular. It is primarily known as a data science library, but can be used by anyone with a data manipulation need. What's important to know is that the main pandas data structure goes by the name "dataframe", and is tabular in nature (in that it is made up of rows and columns). Think "spreadsheet" when you hear "dataframe" and you won't go too far wrong.

A dictionary of dictionaries with pandas?

Sanjita boldly claims she can produce the same data in our `records` dictionary of dictionaries in a lot fewer lines of code. We are *aghast*, as we're quite proud of the twenty lines of code in our `update_records.py` utility. But, we're also keen to see what transpires here…



0 Create a new notebook, then install pandas.

Sanjita creates a new notebook in VS Code, saves it as `ScrapeWithPandas.ipynb`, then uses **pip** to install `pandas`.

Sanjita has control of our keyboard, but not yours. So, be sure to follow along.

```
%pip install pandas lxml
```

In addition to installing "pandas", Sanjita includes the "lxml" parser (which helps "pandas" to process HTML).



OK, we'll give you that one.

The above **pip** command install `pandas` (and its dependencies) into your currently installed Python.

You're now ready for *actual* `pandas` code.

Don't use "pandas", use "pd" instead

Although there's a temptation to simply import the `pandas` library and get going, there's a convention in the ecosystem to import `pandas` under an aliased name of "`pd`". This is a typing convenience, as the two-letter "pd" is a lot less typing than the six-letter "pandas". It might not seem like a big difference, but adds up over time.

1 Import pandas under the alias "pd", then define the URL.

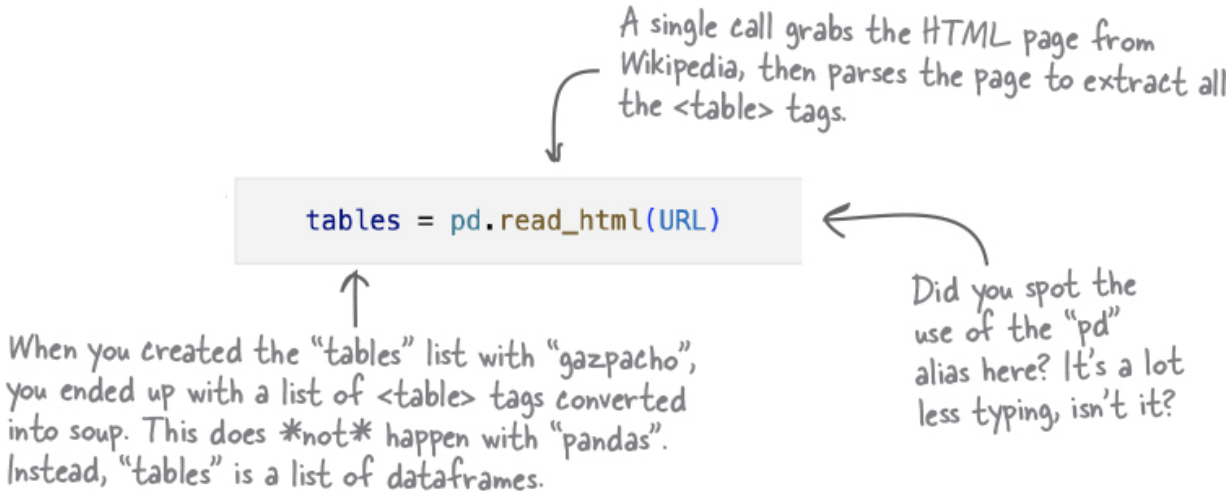
Declaring that all Python code which uses the `pandas` library start with this line, Sanjita imports the `pandas` library, then defines the `URL` constant.



2 Grab, then parse, the URL's HTML to extract all the included tables.

Recall that getting to this point with `gazpacho` involved a call to `get`, then a conversion to *soup*, then you performed a search with

the `find` method. The `pandas` library compacts these three steps into one.



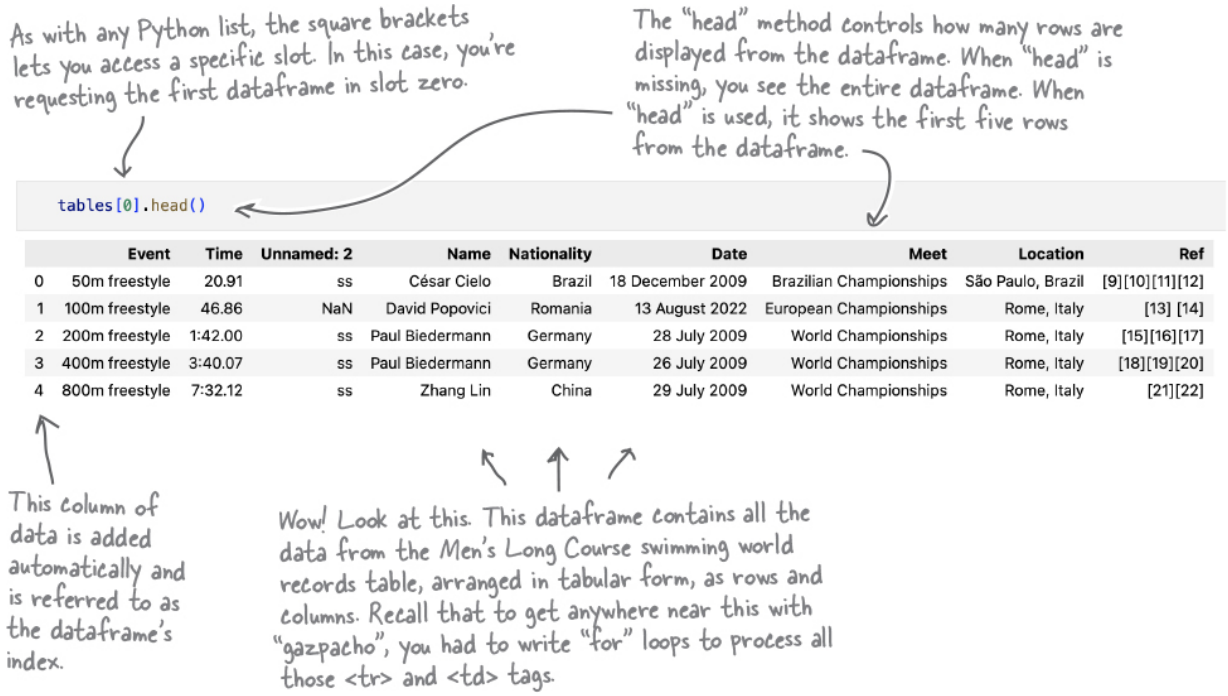
A list of pandas dataframes

The `read_html` function automatically finds, parses, and extracts the `<table>` tags from the identified web page, turning each table into a **dataframe**. When you hear the word "dataframe", think "rows and columns" or "tabular spreadsheet". To better understand what a dataframe is, let's take a look at one.

3 Select a table from the list using the square bracket notation.

For example, if you were to type `tables[0]` into a code cell (then press **Shift+Enter**), you'd see all the rows of data from the first table on the page displayed in tabular form, as rows and columns. If there

are a lot of rows, the data can quickly scroll off your screen, so let's tell `pandas` to display only the first five rows, using the `head` method.



	Event	Time	Unnamed: 2	Name	Nationality	Date	Meet	Location	Ref
0	50m freestyle	20.91	ss	César Cielo	Brazil	18 December 2009	Brazilian Championships	São Paulo, Brazil	[9][10][11][12]
1	100m freestyle	46.86	NaN	David Popovici	Romania	13 August 2022	European Championships	Rome, Italy	[13] [14]
2	200m freestyle	1:42.00	ss	Paul Biedermann	Germany	28 July 2009	World Championships	Rome, Italy	[15][16][17]
3	400m freestyle	3:40.07	ss	Paul Biedermann	Germany	26 July 2009	World Championships	Rome, Italy	[18][19][20]
4	800m freestyle	7:32.12	ss	Zhang Lin	China	29 July 2009	World Championships	Rome, Italy	[21][22]

You've only just started with `pandas`, but this is already impressive. Take a moment to consider all the code you *didn't* need to hand-craft to get to this point.

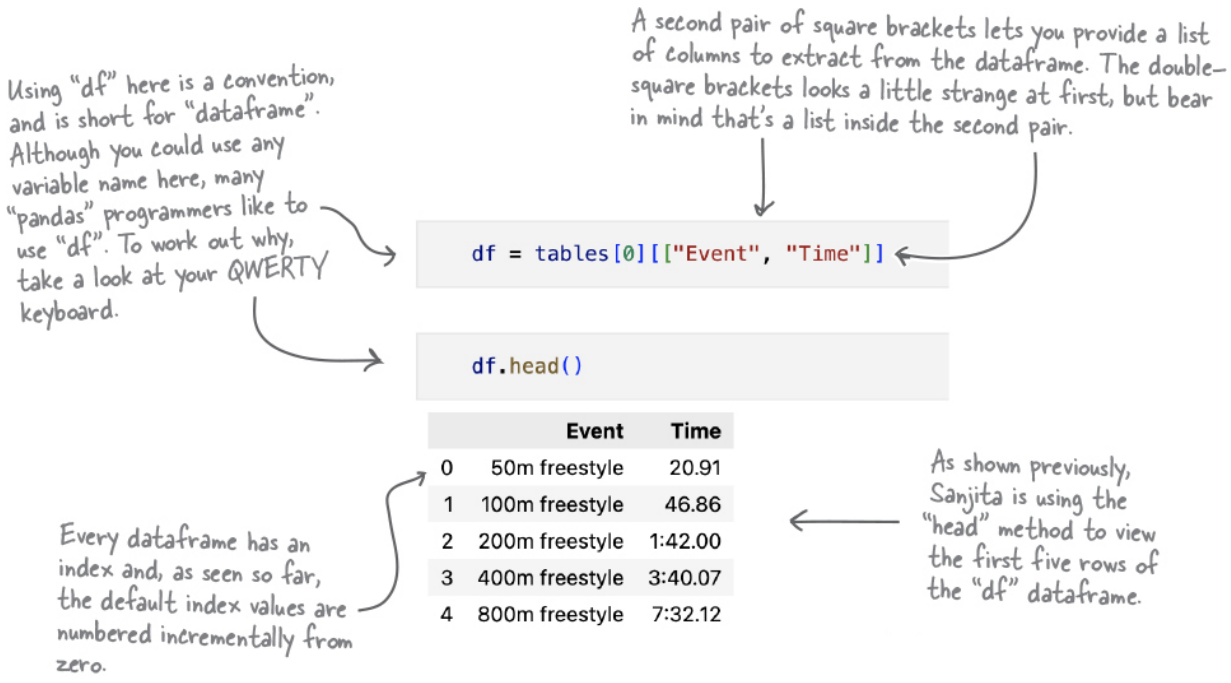
Selecting columns from a dataframe

The dataframes in your `tables` list contain – by default – all the rows and all the columns from each of the processed HTML `<table>` tags. Using yet another extension to the square bracket

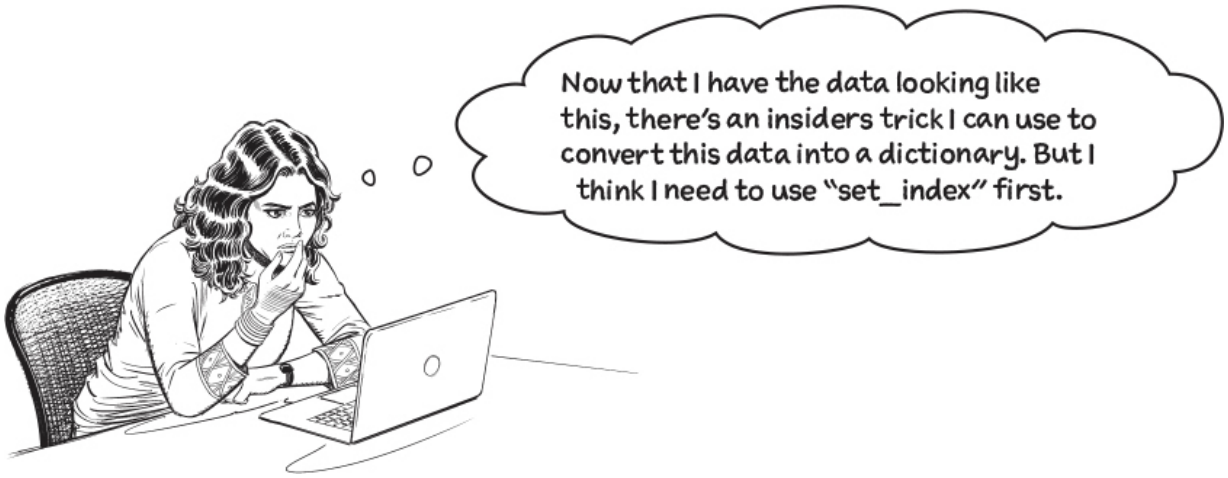
notation, you can specify columns to extract from a `pandas` dataframe.

4 Concentrate on the columns that provide the data you need.

Although having access to *all* the rows and columns in the dataframe is often what you need, for our purposes, only the `Event` and `Time` columns are important. Sanjita selects those columns from the dataframe, assigning them to a variable called `df`.



	Event	Time
0	50m freestyle	20.91
1	100m freestyle	46.86
2	200m freestyle	1:42.00
3	400m freestyle	3:40.07
4	800m freestyle	7:32.12



Sanjita is correct.

There's a `pandas` function called `to_dict` which converts any dataframe into a Python dictionary. However, the default conversions are often not what you want. Let's take a look.

Dataframe to dictionary, attempt #1

Calling the `to_dict` function on a dataframe produces a dictionary, but the default conversion may not be what you need:

```
df.to_dict()
```

```
{'Event': {0: '50m freestyle',
  1: '100m freestyle',
  2: '200m freestyle',
  3: '400m freestyle',
  4: '800m freestyle',
  5: '1500m freestyle',
  6: '50m backstroke',
  7: '100m backstroke',
  8: '200m backstroke',
  9: '50m breaststroke',
  10: '100m breaststroke',
  11: '200m breaststroke',
  12: '50m butterfly',
  13: '100m butterfly',
  14: '200m butterfly',
  15: '200m individual medley',
  16: '400m individual medley',
  17: '4×100m freestyle relay',
  18: '4×200m freestyle relay',
  19: '4×100m medley relay'},
 'Time': {0: '20.91',
  1: '46.86',
  2: '1:42.00',
  3: '3:40.07',
  4: '7:32.12',
  5: '14:31.02',
  6: '23.71',
  7: '51.60',
  8: '1:51.92',
  9: '25.95',
  10: '56.88',
  11: '2:05.95',
  12: '22.27',
  13: '49.45',
  14: '1:50.34',
  15: '1:54.00',
  16: '4:03.84',
  17: '3:08.24',
  18: '6:58.55',
  19: '3:26.78'}}
```



In this default case, and with this data, the "to_dict" function has created a dictionary of dictionaries which, despite including all the data, isn't what you were hoping for. What you actually want is a dictionary which associates the event names with the world record times. Let's have another go.

Dataframe to dictionary, attempt #2

You can adjust the behavior of the `to_dict` function by passing parameter values. On a quick read of the `pandas` docs, the `records` parameter looks interesting:

```
df.to_dict("records")
```

```
[{'Event': '50m freestyle', 'Time': '20.91'},
 {'Event': '100m freestyle', 'Time': '46.86'},
 {'Event': '200m freestyle', 'Time': '1:42.00'},
 {'Event': '400m freestyle', 'Time': '3:40.07'},
 {'Event': '800m freestyle', 'Time': '7:32.12'},
 {'Event': '1500m freestyle', 'Time': '14:31.02'},
 {'Event': '50m backstroke', 'Time': '23.71'},
 {'Event': '100m backstroke', 'Time': '51.60'},
 {'Event': '200m backstroke', 'Time': '1:51.92'},
 {'Event': '50m breaststroke', 'Time': '25.95'},
 {'Event': '100m breaststroke', 'Time': '56.88'},
 {'Event': '200m breaststroke', 'Time': '2:05.95'},
 {'Event': '50m butterfly', 'Time': '22.27'},
 {'Event': '100m butterfly', 'Time': '49.45'},
 {'Event': '200m butterfly', 'Time': '1:50.34'},
 {'Event': '200m individual medley', 'Time': '1:54.00'},
 {'Event': '400m individual medley', 'Time': '4:03.84'},
 {'Event': '4×100m freestyle relay', 'Time': '3:08.24'},
 {'Event': '4×200m freestyle relay', 'Time': '6:58.55'},
 {'Event': '4×100m medley relay', 'Time': '3:26.78'}]
```



This is a case of "close, but no cigar". This is closer to what you want, but not quite.



Enough said.

The `set_index` function let's you *reshape* an existing dataframe to use an identified column as its index (as opposed to the default incrementing numeric added to the start of each row).

Dataframe to dictionary, attempt #3

Sanjita did mention a few pages back that the `set_index` function would help. Clearly, as shown on the last two pages, the `to_dict` method is powerful. But we need to pay closer attention to what Sanjita is telling us and maintain our focus.

NOTE

As there's lots included with "pandas", it can be easy to get distracted. We'll try to keep our eye on the prize here.

---

5 Deploy the insider's trick to reshape your dataframe.

Being a bit of a `pandas` expert is helping Sanjita here, as they already know what using `set_index` does for them.

Sanjita is showing off a little bit here, but note how "set_index" is used to adjust the shape of the "df" dataframe, which is then assigned to "df", overwriting what was there before.

```
df = df.set_index("Event")
df.head()
```

Event	Time
50m freestyle	20.91
100m freestyle	46.86
200m freestyle	1:42.00
400m freestyle	3:40.07
800m freestyle	7:32.12

When the reshaped dataframe is converted to a dictionary this time, it turns into a dictionary of a dictionary... which looks very familiar, doesn't it? The outer dictionary has a single key ("Time") which is associated with a inner, nested dictionary which contains the event–time associations as a collection of key/value rows.

```
df.to_dict()
```

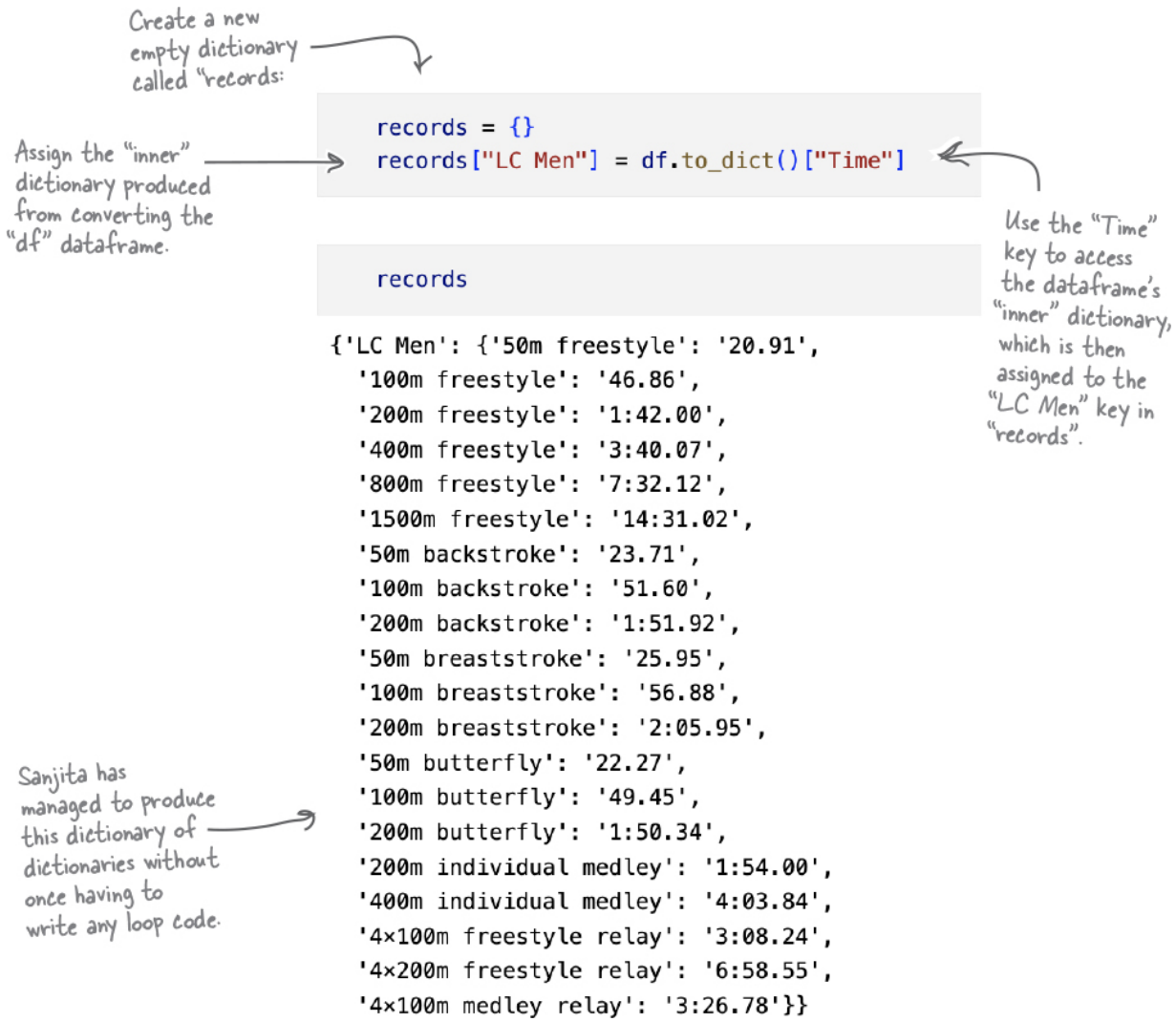
```
{'Time': {'50m freestyle': '20.91',
  '100m freestyle': '46.86',
  '200m freestyle': '1:42.00',
  '400m freestyle': '3:40.07',
  '800m freestyle': '7:32.12',
  '1500m freestyle': '14:31.02',
  '50m backstroke': '23.71',
  '100m backstroke': '51.60',
  '200m backstroke': '1:51.92',
  '50m breaststroke': '25.95',
  '100m breaststroke': '56.88',
  '200m breaststroke': '2:05.95',
  '50m butterfly': '22.27',
  '100m butterfly': '49.45',
  '200m butterfly': '1:50.34',
  '200m individual medley': '1:54.00',
  '400m individual medley': '4:03.84',
  '4×100m freestyle relay': '3:08.24',
  '4×200m freestyle relay': '6:58.55',
  '4×100m medley relay': '3:26.78'}}
```

It's another dictionary of dictionaries

Sanjita takes the dictionary from the previous page and starts to build the `records` dictionary of dictionaries.

6 Build out the records dictionary of dictionaries, one course at a time.

Sanjita can't help but smile as she shows us the code which starts to build the `records` dictionary of dictionaries.



EXERCISE



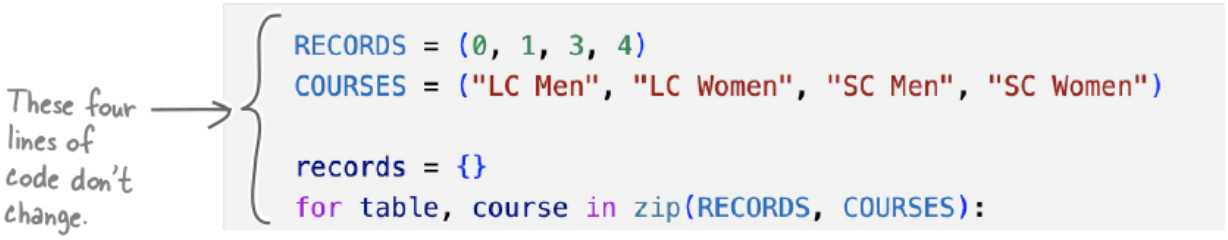
Here's the code from the previous chapter which, when executed against the data in the `tables` list created by `gazpacho`, turns the scraped data into the `records` dictionary of dictionaries:

```
RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC W

records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr") [1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

Your job is to convert the six lines of code from the outer **for** loop to process the `tables` list created by `pandas`, creating the

`records` dictionary of dictionaries from `pandas` dataframes instead of from the `gazpacho` soup:



Write your code in this space.

EXERCISE SOLUTION



Here's the code from the previous chapter which, when executed against the data in the `tables` list created by `gazpacho`, turns the scraped data into the `records` dictionary of dictionaries:

```
RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC W

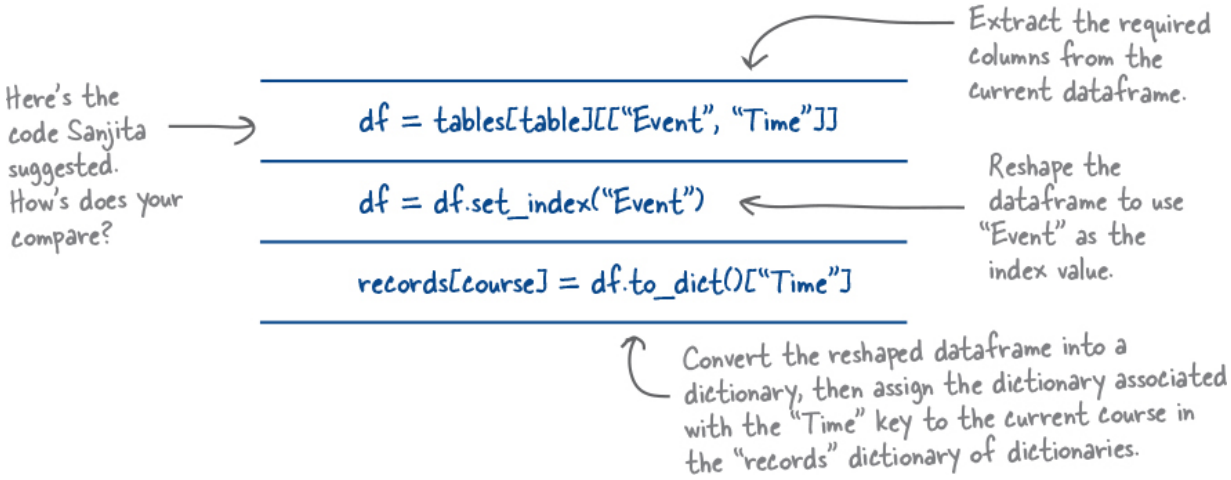
records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr") [1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

Your job was to convert the six lines of code from the outer **for** loop to process the `tables` list created by `pandas`, creating the

`records` dictionary of dictionaries from `pandas` dataframes instead of from the `gazpacho` soup:

```
RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

records = {}
for table, course in zip(RECORDS, COURSES):
```



TEST DRIVE



Let's take this `pandas`-based code for a spin, comparing the `records` dictionary of dictionaries produced by the `pandas` code to that produced by `gazpacho`.

```
RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC W

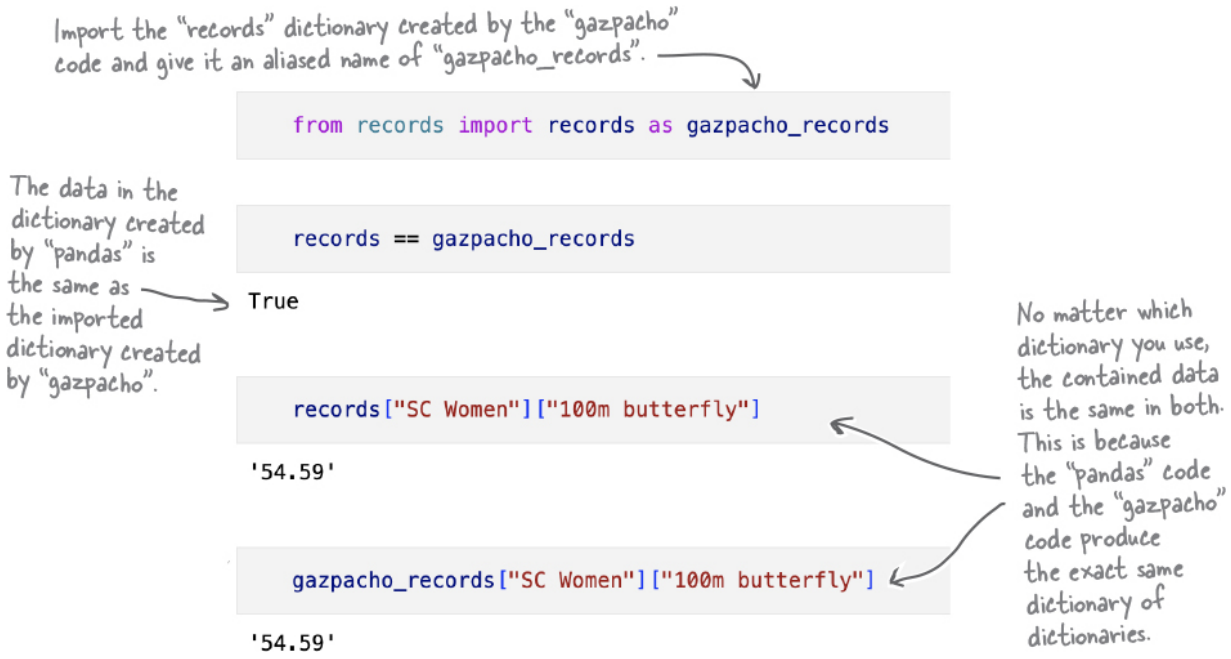
records = {}
for table, course in zip(RECORDS, COURSES):
    df = tables [table][["Event", "Time"]]
    df = df.set_index("Event")
    records[course] = df.to_dict()["Time"]
```

The **len** BIF confirms the latest `records` has four "outer" keys:

```
len(records)
4
```

No surprises here. The "records" dictionary created by "pandas" is the same size as that created by "gazpacho".

Let's import the `records` dictionary created by `gazpacho` into your notebook. This allows you to compare the data in the `pandas` dictionary against the data in the `gazpacho` dictionary:



Comparing gazpacho to pandas

With the `pandas` version of your `records`-producing code ready, let's compare the `gazpacho` code to it's equivalent `pandas` version (to see if Sanjita was right to chastise our lack of `pandas` use).

Here's the `gazpacho` code:

```
import gazpacho

URL = "https://en.wikipedia.org/wiki/List_of_world_records_in_swimming"

html = gazpacho.get(URL)
soup = gazpacho.Soup(html)
tables = soup.find("table")

RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr")[1:]:
        columns = row.find("td")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

This is the code from the previous chapter, which was lovingly hand-crafted to use "gazpacho", creating the "records" dictionary of dictionaries.

And here's the equivalent code written with `pandas`:

```
import pandas as pd

URL = "https://en.wikipedia.org/wiki/List_of_world_records_in_swimming"

tables = pd.read_html(URL)

RECORDS = (0, 1, 3, 4)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

records = {}
for table, course in zip(RECORDS, COURSES):
    df = tables[table][["Event", "Time"]]
    df = df.set_index("Event")
    records[course] = df.to_dict()["Time"]
```

And here's Sanjita's code which also creates "records", but uses "pandas" and takes fewer lines of code. (It's OK to be jealous).


See? Less code when you solve this problem with "pandas". What's not to like?

On the face of things, this looks like progress.

There is less `pandas` code than the `gazpacho` equivalent, but writing less code can *sometimes* have a hidden cost.

Let's explore what this means by calling in the experts…

Fireside Chats



Tonight's talk: **gazpacho and pandas sit down to discuss the low-down on HTML parsing with Python.**

pandas:	**gazpacho:**
Based on my back-of-a-napkin calculation, I make that a 33% reduction in the number of lines of code for the `pandas` solution. How's that?	
	It's certainly less code, but at what cost?
Cost? What cost?!? There's less code. There is no extra cost!	
	I'm not so sure. For one thing, you – and all the modules you depend on – add a fair amount of *overhead*. Whereas me, with no external dependencies, have none. I do what I do *without* the bloat.
I'm not going to be shamed by your cruel bloat jibe… Yes, I'm a **big** module with a collection of dependencies which take a	

moment of two to import and run but, look at all I do. I'm packed full of functionality. Everyone loves me.

But you can't even parse entire web pages…

I beg your pardon! Have you been sleeping for the last dozen pages or so? Did you not see that 33% reduction in code-size?

Fewer lines of code isn't always a good thing. What I'm getting at here is the fact that you can *only* parse HTML `<table>` tags. Everything else on the page is thrown by the wayside.

Eh… yes… but… eh…

I'm waiting…

All that other stuff isn't important. All the good stuff 's in `<table>`

tags!

So, if I want to extract all the `<img>` tags from a page, I can't use you, right?

Yes, that's true. I can't do that… unless the `<img>` tags are part of the `<table>`, then I can.

And if I want to extract all the `<a>` tags from any given HTML page, it's the same deal?

Emmm… I can only see them if they are included within the `<table>` tag. That's not bad, is it?

It *is* if that's what you want. For instance, if you need to access the all the `<a>` tags on the *entire* page, using your `read_html` function fails, as you can't see anything outside the `<table>` and `</table>` tags.

Please don't tell anyone.

Don't worry. Your secret is safe with me.

Look: you have to admit, my `<table>` processing abilities are pretty cool, aren't they?

Yes, if that's the *only* requirement you have. I can't compete. Although I can, with a few extra lines of code (and a loop or two), process `<table>` tags, too.

I knew you'd see the light, and you did say it: *you can't compete*. And let's not forget I automatically turn the data I find into dataframes, unlike the yucky soup you use.

There's nothing wrong with my soup! It represents *all* the HTML from a web page in a

parseable format which my `find` method can make mincemeat of, in short-order. It's all very tasty…

Oh, dear… enough with the food jokes!

Too much for you to stomach, eh?

Enough already!

Suddenly, there's a loud, angry knock at the door.

Yikes, who's that?!?
They don't sound at all happy.

Quick, hide. It's *Beautiful Soup*, and they've brought along the `requests` library!!



Absolutely. That's one take-away.

Which sounds like another food joke, doesn't it?

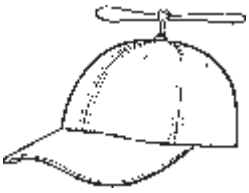
NOTE

Sorry. We couldn't resist.

In fact, there's nothing to stop you from using `gazpacho` and `pandas` *together*, leveraging the strengths of each when it makes sense to do so.

If all you need to do is quickly extract data from HTML tables, it's hard to beat `pandas`. If you need to access data from elsewhere in a HTML page, we think `gazpacho` is a great choice. But, as stated, it's OK to mix'n'match.

GEEK NOTE



It is beyond the scope of this book to discuss `pandas` in any great depth. If you'd like to know more, a quick *DuckDuckGo* search will reveal a wealth of how-to guides on the Internet. Our top-pick for starting out is the rather excellent **Getting started tutorials** included in the `pandas` documentation:

https://pandas.pydata.org/docs/getting_started/intro_tutorials/index.html

Take a few hours to work through those (in order) and you'll be well on your way.

For a more detailed coverage, we recommend *anything* by Wes McKinney, and the third edition of **Python for Data Analysis** is a must-have:

https://wesmckinney.com/book/

THERE ARE NO DUMB QUESTIONS

Q: Isn't `pandas` a data science thing?

A: On the surface, yes. The `pandas` library started life as a tool to perform financial analysis, then morphed into an general-purpose data analysis tool around about 2010, when it became an open source project. The rest, as they say, is history. At the time of writing, `pandas` is the premier data analysis tool within the Python data science space.

Q: Can `pandas` only be used by Data Scientists?

A: As shown in this chapter, the answer to that question is: no. If `pandas` is a good fit for your application, go ahead an use it as needed. Be advised that `pandas` (and its dependencies) are large libraries, which means they can sometimes take a moment or two to import. But, if you aren't in a hurry, `pandas` can be a great fit.

Q: So… should I refactor my `update_records.py` utility to use `pandas` instead of `gazpacho`?

A: You could if you wanted to. But, you know what they say: *if it works, don't fix it.*

Q: A lot of `pandas` code examples on the Internet start with the `import pandas as pd` line, but they also include another line:

`import numpy as np`. **Should I add that line too?**

A: The `numpy` library (pronounced "num-pie", as opposed to rhyming with *Humpty Dumpty*) provides a lightening fast implementation of a numerical array for Python. The `pandas` library is built on top of `numpy`, but you only need to import `numpy` into your code if you need to access its features specifically. As shown in this chapter, `pandas` works fine without a specific import of `numpy`.

Q: So, who uses `numpy`?

A: To be honest, `numpy` is used by lots of people to do lots of things. For instance, if you have a mountain of numerical data that needs processing, you're better off putting it into a `numpy` array over a Python list, as the former is way faster, more memory efficient, and built for that specific use-case. Additionally, if you have data arranged as a matrix, you'd be strongly advised to learn how `numpy` can help you process your matrix data. Over and above Data Scientists, `numpy` is a star in the Mathematical and General Science worlds, too, but *anyone* can use it.

Q: I guess `pandas` and Jupyter Notebook's were made for each other?

A: Yes and no. The *Jupyter Notebook* technology evolved out of a project called **iPython** which itself provides an upgrade to Python's

(rather basic) command-line interface (the `>>>` prompt). When the **iPython** folks were looking to add a nice GUI front-end to their technology, the opted for the web, creating *Jupyter Notebook* which, as well as running inside VS Code, originally started life running within any modern web browser. In fact, most people run their *Jupyter Notebooks* from their web browser, it's just us programmer-types who prefer to do so within an editor. The increase in the use of *Jupyter Notebooks* appears to go hand-in-hand with the increase in use of `pandas`, so both technologies are often considered to be tied-at-the-hip. But, as you've hopefully seen in this book (and, more specifically, this chapter), you can be effective in one without having to use the other. *Jupyter Notebook's* don't always need to use `pandas`, and `pandas` doesn't always need to run within a *Jupyter Notebook*. It all depends on what you're trying to do.

It was only the shortest of glimpses…

Of what's possible with the `pandas` library. Books, articles, businesses, fortunes, and projects exist in the world thanks to the `pandas` library. Due in no small part to the uptake of `pandas`, the growth of Python has mushroomed since 2010 within the data science community, and this is likely helping to fuel Python's growth elsewhere.

`pandas` is a Python super-power, and is a very useful library to have in your toolbox. It's not fair the data science crowd get to have all the fun with `pandas`, so our advice is to use it when you can (and have *fun* doing so, too).

NOTE

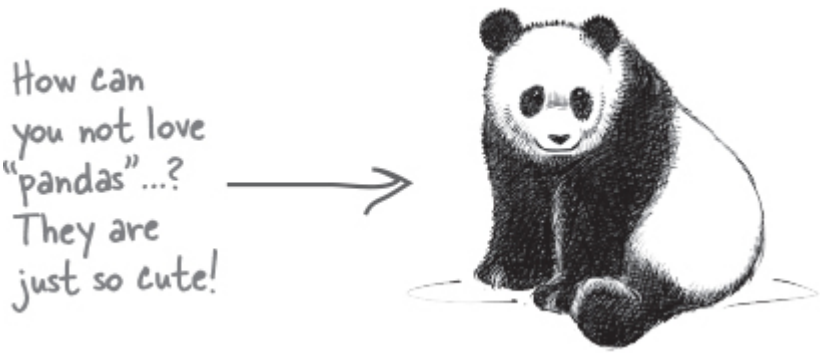
There's more to "pandas" than processing HTML tables and, as you explore what it can do, you'll see how "pandas" works hard to provide very powerful data maniplua-tion tools which you can exploit with very little code (and next to no loops, too).

Although this is the shortest chapter in this book, you're not getting out of here without a quick chapter review and a crossword puzzle. Enjoy both, and see you in the next chapter.

CHAPTER REVIEW BULLET POINTS

- `pandas` is the leading data analysis tool used by Python programmers. It's used *everywhere*.
- `pandas` provides excellent support for arranging your data in **tabular** form: data is arranged in row/ column format. The name given to this tabular form in `pandas` is **dataframe**.
- The `import pandas as pd` line of code may well be one of the most used within the Python data science space, and is right up their with `import numpy as np`. The `as` keyword allows you to specify a shorthand *alias* for a name in Python.
- The `read_html` function (which is part of `pandas`) grabs, reads, then converts any HTML page into a list of tables, with each slot containing a dataframe. It's a pretty cool function (assumming all you need to do is scrape `<table>` data).
- The `head` function lets you view the first five rows of data in any dataframe.
- The `to_dict` function (the *real* star of the show in this chapter) converts the data in a dataframe to a Python dictionary.
- If the default conversion isn't what you want, you can adjust the behavior of `to_dict` by passing in parameters. The `"records"` parameter almost did the trick in this chapter (but not quite).
- Each `pandas` dataframe supports the `set_index` function which lets you reshape your dataframe.

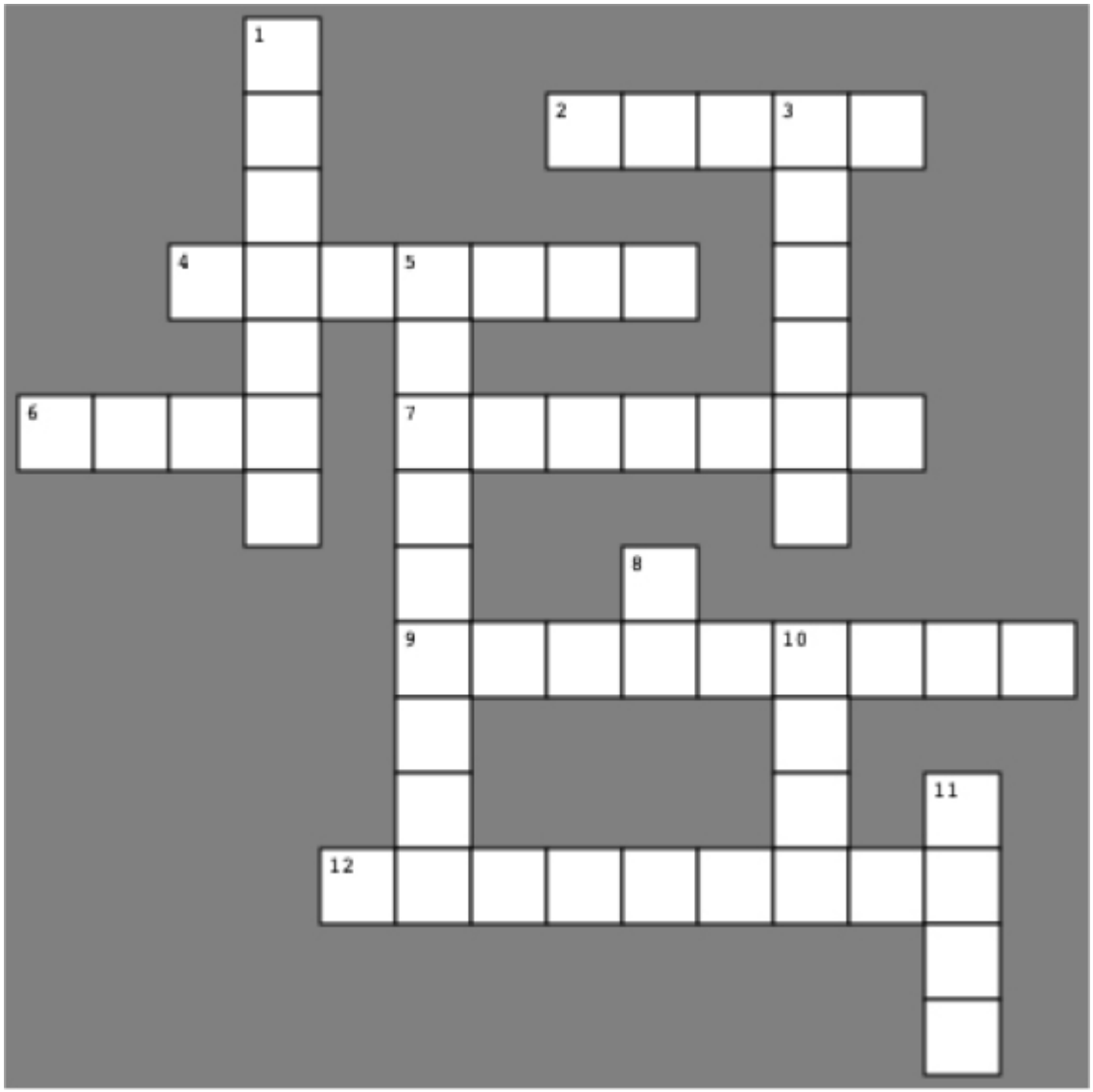
- Although web scraping code written with `pandas` is generally fewer lines of code than the equivalent written with `gazpacho`, it does not necessarily follow that the `pandas` code is “better”.



The Tabular Data Crossword



You’ll find all the answers to the clues in this chapter. The solutions are on the next page.



Across

2. This popular library's name does not rhyme with *Humpty Dumpty*.

4. A function to convert from 5 down to a Python dictionary.

6. This chapter's *Fireside Chat* may have had too many of these jokes.

7. It's got rows and columns, arranged as a table. It's _____.

9. This function's equivalent in `gazpacho` is to use `get`, then `Soup`, then `find`.

12. This function reshapes things.

Down

1. This parameter is "close, but no cigar".

3. Hugely popular, it's this chapter's super-power.

5. The row/column data structure central to three down.

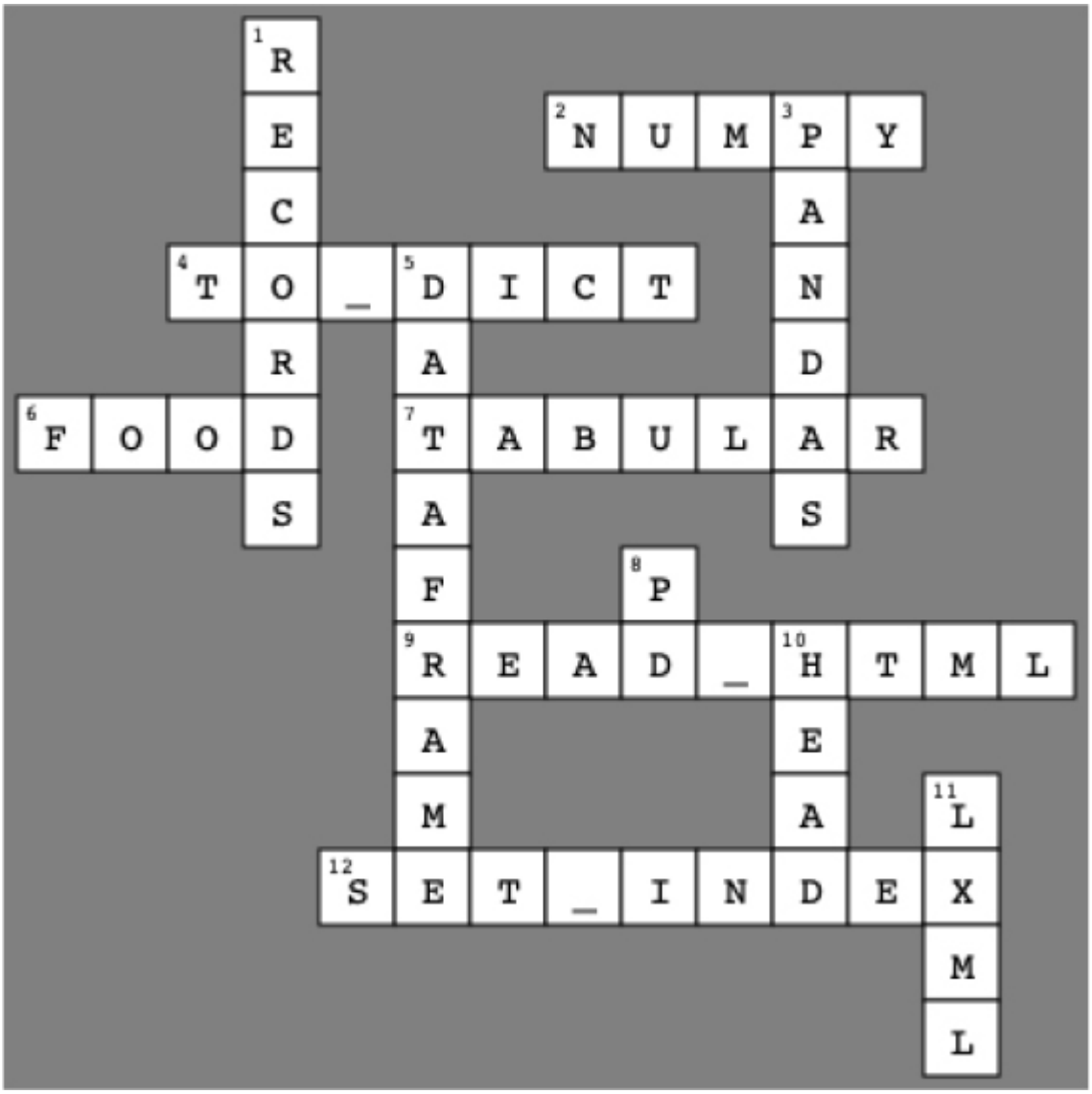
8. A two-letter alias for 3 down.

10. Let's you view the first five rows.

11. Sanjita made sure to also install this parsing library.

The Tabular Data Crossword Solution





Across

2. This popular library's name does not rhyme with *Humpty Dumpty*.

4. A function to convert from 5 down to a Python dictionary.

6. This chapter's *Fireside Chat* may have had too many of these jokes.

7. It's got rows and columns, arranged as a table. It's _____.

9. This function's equivalent in `gazpacho` is to use `get`, then `Soup`, then `find`.

12. This function reshapes things.

Down

1. This parameter is "close, but no cigar".

3. Hugely popular, it's this chapter's super-power.

5. The row/column data structure central to three down.

8. A two-letter alias for 3 down.

10. Let's you view the first five rows.

11. Sanjita made sure to also install this parsing library.

Chapter 11. Databases: *Getting Organized*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 10 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



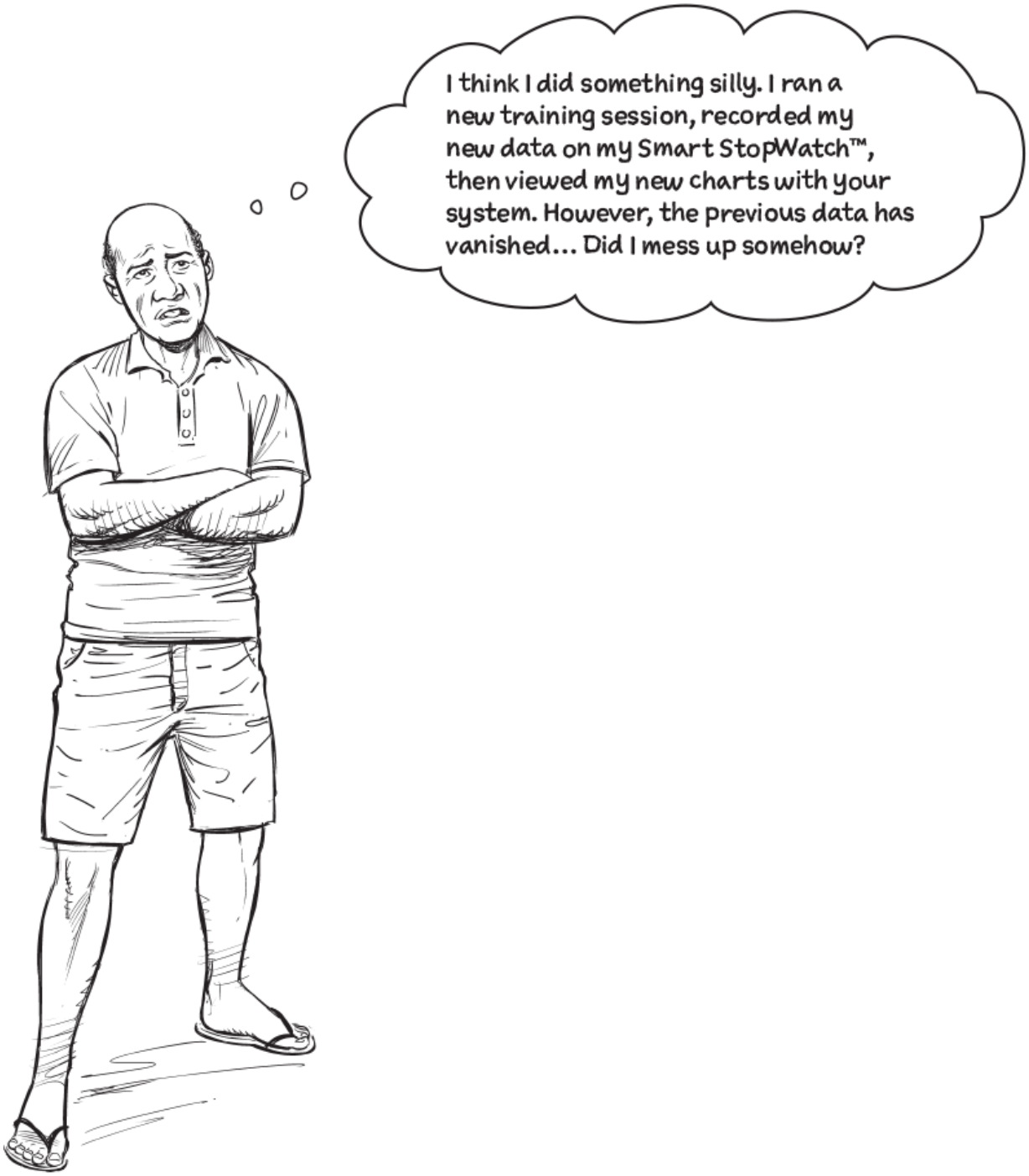
Sooner or later, your application's data needs to be managed.

And when you need to more appropriately **manage** your data, Python (on it's own) may not be enough. When this happens, you'll need to reach for your favorite **database** engine. To keep things… em, eh…

manageable, we're going to stick with database engines that support trusty ol' **SQL**. In this chapter, you'll not only **create** a database then add some **tables** to it, you'll also **insert**, **select**, and **delete** data from your database, performing all of these tasks with SQL queries orchestrated by your Python code.

The Coach has been in touch…

And he's a little confused.


I think I did something silly. I ran a new training session, recorded my new data on my Smart StopWatch™, then viewed my new charts with your system. However, the previous data has vanished... Did I mess up somehow?

Whoops.

When you take a look at the Coach's system, someone copied the new files from the latest swim session into the `swimdata` folder overwriting the existing data (where the filenames matched).

You assure the Coach he did nothing wrong, and you also confirm you still have a copy of the "old" data in your ZIP archive.

With a friendly "don't worry", you commit to adjusting the system so that older, historical data doesn't get wiped by any new stuff.

All you have to do now is work out how to do this…

Cubicle Conversation



Sam: OK, team, what are your thoughts on this?

Alex: Surely we can work with what we have?

Mara: How so?

Alex: All we need to do is invent a mechanism which swaps the dataset in the `swimdata` folder as needed. We ask the Coach which dataset he wants to work with, then take it from there.

Sam: That might work, but what happens when more than one person is using the system *at the same time*?

Mara: Does that make a difference?

Sam: Well… imagine the Coach has selected a dataset to work with, resulting in the system moving the selected dataset into the `swimdata` folder. Moments later, another user selects a *different* dataset, resulting in the system moving the user's dataset into the `swimdata` folder. The Coach still think he's working with the first dataset when in actual fact he's looking at the second. Whoops…

Alex: I hadn't figured on that. Can't we tell the Coach not to allow anyone else to use the system when he's online?

Sam: We could but, in the long run, that's not really a workable solution. Remember, the system is a webapp, so the basic assumption is

there are multiple users working on the system at once.

Mara: So we can't swap datasets in and out then?

Alex: I guess not. We'll need something else… but what?

Sam: We need something designed to handle multiple datasets so each user thinks they're the only user of the system. We need a system to handle our data…

Alex: Of course… we need a database, right?

Sam: Yes. Databases are designed to manage your application's data, as well as share it efficiently and safely among multiple users.

Mara: Is it easy to use a database with Python?

Sam: Yes. And, believe it or not, Python comes with the world's most popular SQL-based database technology already built-in. It's called **SQLite**3.



SQLite3 lets us get going quickly.

Starting with SQLite3 lets you build out your database functionality without having to install *anything*. As SQLite3 comes with Python, you

already have it. There's nothing to install, setup, configure, nor tweak.

Once you have your database code written and working *locally* on your computer using SQLite3, it's not too much work (as you'll see later) to move to one of those *big-iron* database technologies.

NOTE

And don't think for one second that SQLite3 isn't a "real" database technology. It the most widely deployed database technology in existence. See https://www.sqlite.org/mostdeployed.html to learn more.

Now… before you start writing any "database code", some thought has to be given over to deciding on a plan of attack.

RELAX



Don’t panic if all this database talk is giving you palpitations.

Stay calm. We aren’t expecting you to be a database expert. Nor do we expect you to know SQL in any detail.

We’ll provide you with the SQL you need to follow along but, if it all looks like gibberish to you, don’t fret. We’ll be sure to explain what we’re up to as you work through this material.

It pays to plan ahead…

The *Head First Coders* have been in touch, and are suggesting a four-point plan for this work.

1. **Decide on a structure for your data, then create your database tables**

You need to decide how the data in your database is going to be arranged. Once you have decided on this, you can create the necessary database and tables using Python and SQL.

2 Add your data values to your database tables

The Coach's system uses the data in the `swimdata` folder. You'll need to take the data from this folder and add it to the appropriate database tables. As in Task #1, Python and SQL are your go-to technologies here.

3 Extract the data you need from your database tables

At the moment, your webapp's code dips in and out of the `swimdata` folder as needed to grab the data required to do its thing. With Task #2 complete, you can write Python and SQL to grab the data your webapp needs from your database tables instead.

4 Adjust your existing webapp code to use the data in your database tables

With Task #3 done, you'll need to change your current webapp's code to grab its data from your database as opposed to from the `swimdata` folder.

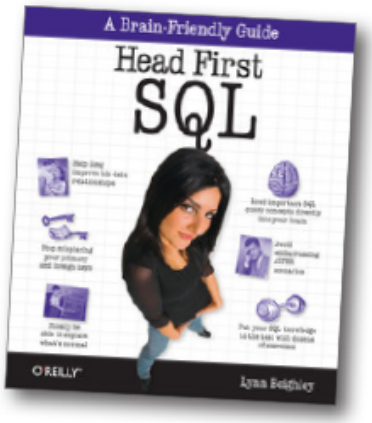
Of course, each of these tasks is likely a bit of work, and may include any number of sub-tasks, which might take some time… But, you did promise the Coach you'd fix this issue, and a promise is a promise, isn't it?

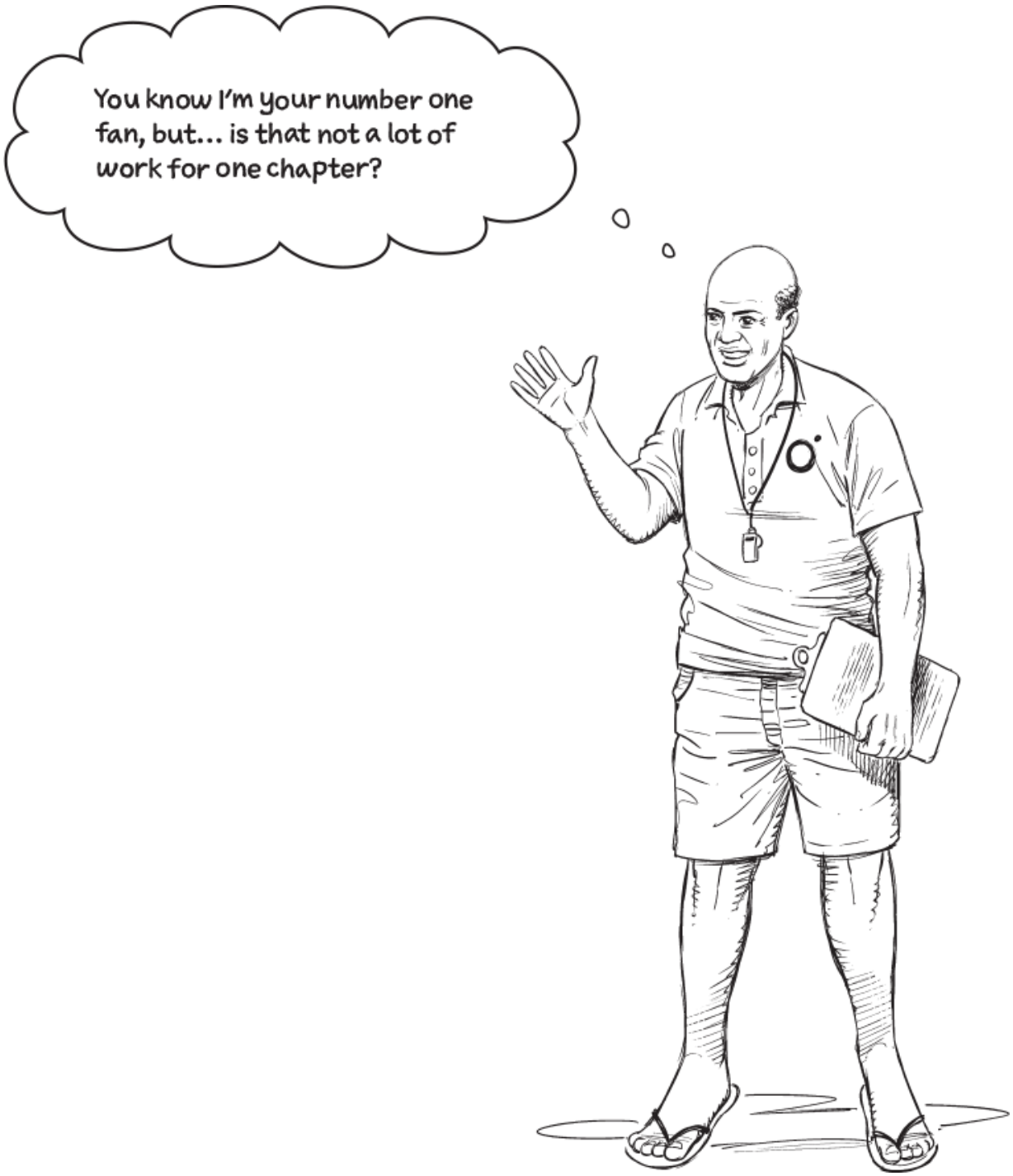
As you're about to see, by starting out with SQLite3 as your database engine, you can achieve these four tasks with a mixture of Python code and SQL, and you can experiment to your heart's content within a Jupyter Notebook while you work out the new code you need *before* adjusting your existing webapp.

NOTE

And remember: we'll provide all the SQL you need as you work through these tasks.

Looking for a quick primer on the essential SQL commands? You'll find lots of helpful material on the Internet. Despite everything that's out there, our favorite learning guide remains Lynn Beighley's excellent Head First SQL. It's an oldie, but it's a goodie. 😊





Don't worry, Coach, we've got this.

We know this sounds like a lot, but we're on top of it. In this and the next chapter (and with the help of a few additional notebooks) we'll have you back up and running again… or swimming again (see what we did there?).

Task #1: Decide on your database structure

Here's a reminder of what's required for Task #1:

1 **Decide on a structure for your data, then create your database tables**

You need to decide how the data in your database is going to be arranged. Once you have decided on this, you can create the necessary database and tables using Python and SQL.

We stated earlier that we don't expect you to be super comfortable with SQL, so let's call in the *Head First Coders* to offer up some advice here.

Cubicle Conversation



Sam: Can anyone remember what we decided to do last night?

Alex: Before or after those four beers?

Mara: It was good to unwind, wasn't it?

Sam: I seem to recall writing something down…

Alex: I'm surprised you remember anything!

Sam: Ha! Ha! I was in *total* control.

Mara: I remember deciding we needed *three* tables…

Alex: Ah, yes, it's coming back to me now: tables called "swimmers", "events", and "times", right?

Sam: Sounds good to me.

Mara: You really can't remember, can you?

Sam: Of course I can. Eh… was I singing at some point?

Mara: You know what they say: *what happens at after-work cocktails stays at after-work cocktails*.

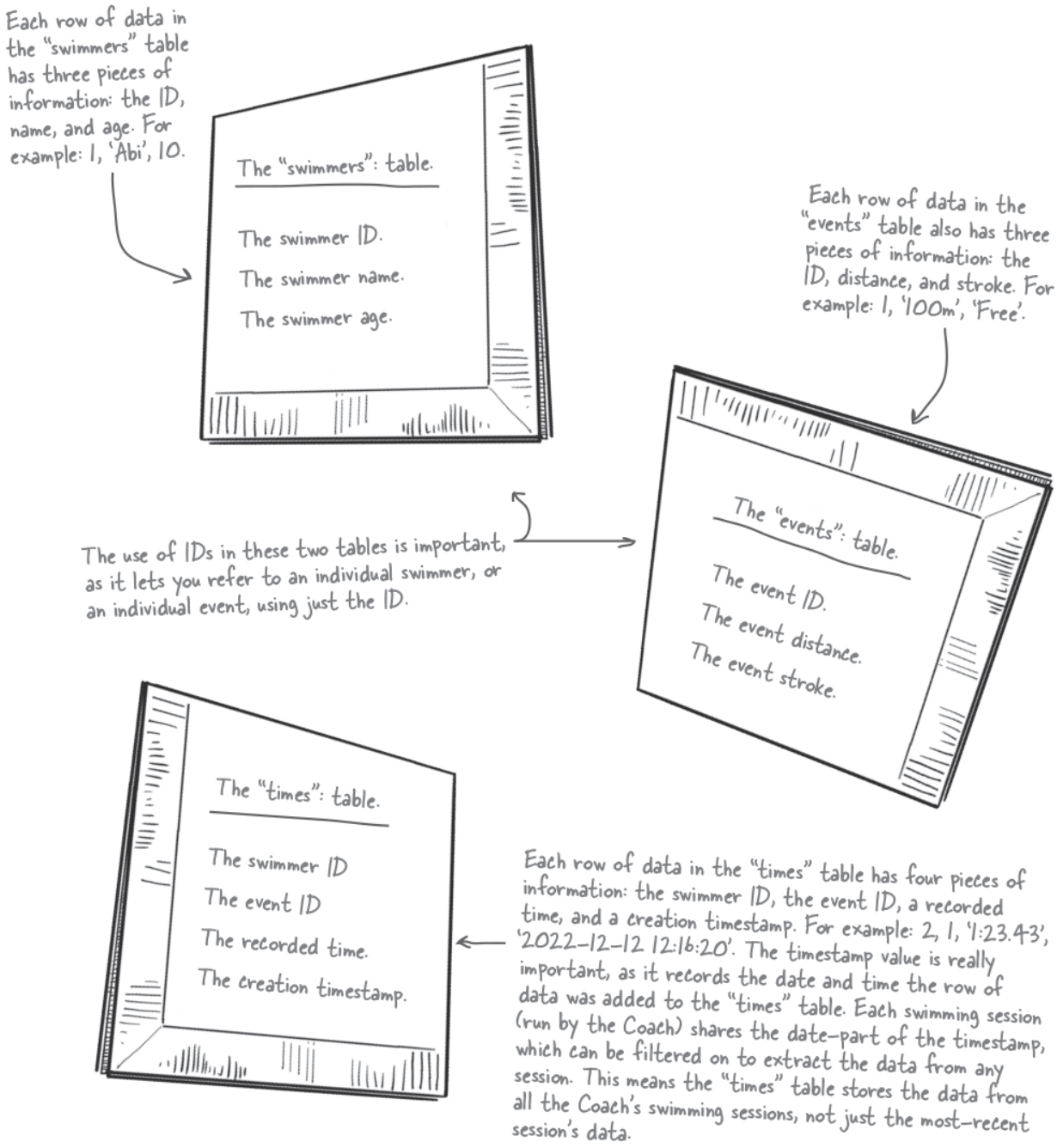
Alex: I'm telling you, we definitely wrote this down…

Sam [pulling something from his pocket]: Ah ha! I told you I was in total control. Look at these.

Alex: Brilliant! It's the database paper napkins. I *knew* we wrote everything down.

Mara: Our memories may be failing us, but thankfully we have our paper napkins!

Sam: For sure! Let's go through each of the napkins…now that my third cup of coffee is kicking in!



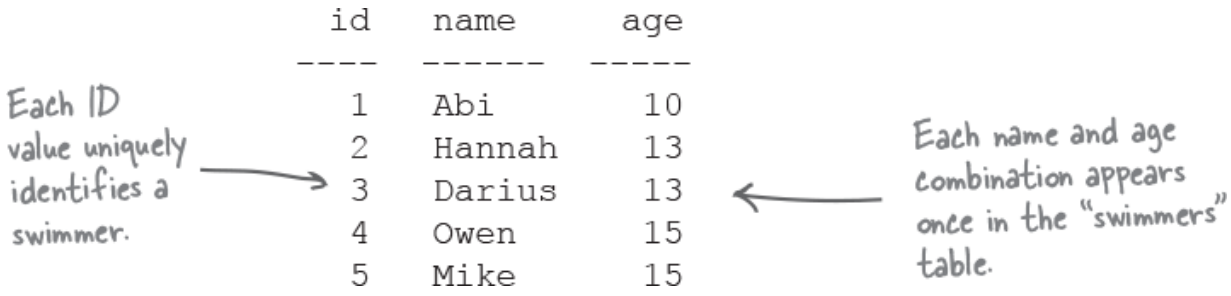
The napkin structure + data

The Boss further explains that the `swimmers` table contains a row of data for each of the Coach's young charges. In addition to the unique ID for each row, each swimmer can be identified by the combination of their name and age group.

Assuming the `swimmers` table is populated with each swimmer's data, here's what the first five rows of data might look like:

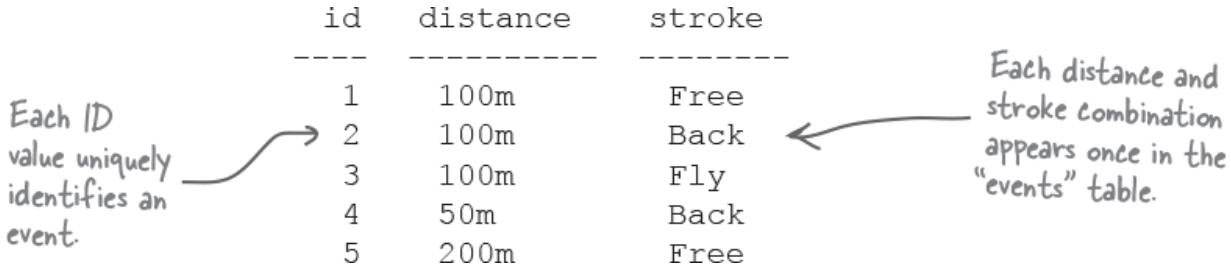
NOTE

You'll get to this in a little bit.



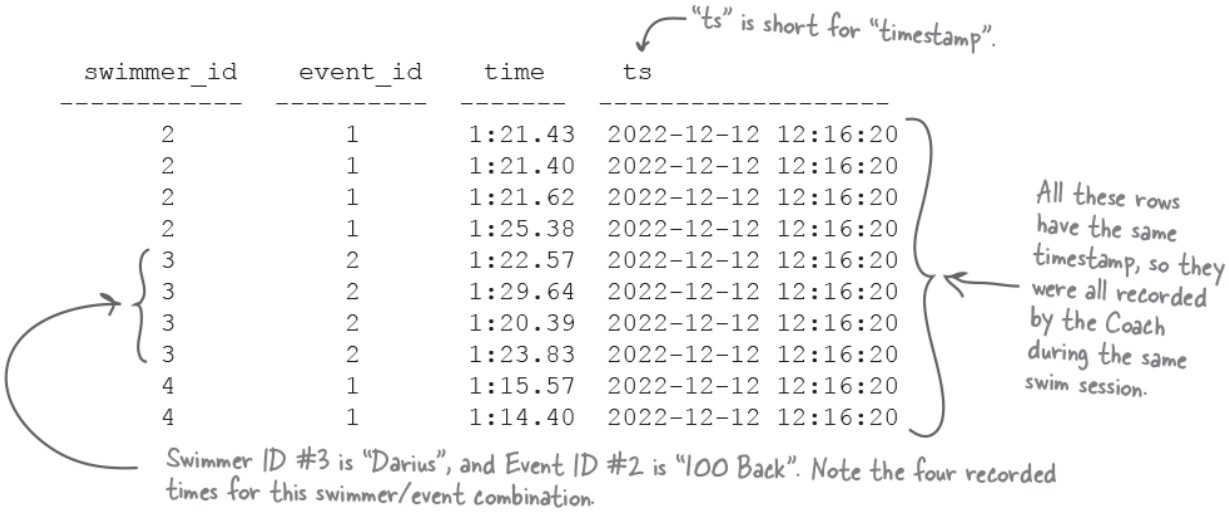
id	name	age
1	Abi	10
2	Hannah	13
3	Darius	13
4	Owen	15
5	Mike	15

Similarly, the `events` table, when populated with each event's data, might have the following data in its first five rows:



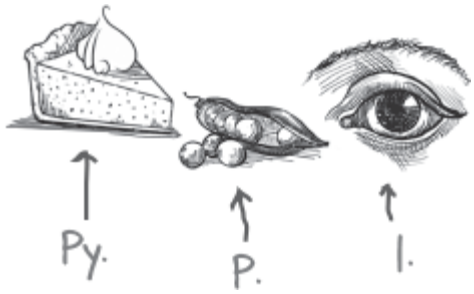
id	distance	stroke
1	100m	Free
2	100m	Back
3	100m	Fly
4	50m	Back
5	200m	Free

The `times` table, when populated with timing data, could have the following data in its first ten rows:



Installing the DBcm module from PyPI

With the tables (and their structure) identified, let's create a database to contain them. Additionally, let's also create the three tables: `swimmers`, `events`, and `times`.



Earlier, a big deal was made of the fact that SQLite3 comes built-into Python, and indeed it does. As such, you can follow the instructions detailed within the Python documentation, here:

https://docs.python.org/3/library/sqlite3.html

PSL is shorthand for the "Python Standard Library".

to use Python's DB-API technology, together with the `sqlite3` module from the PSL, to "talk" to SQLite3. As an approach, this certainly works. But, at *Head First Labs*, the coders prefer to interact with SQLite3 using PyPI's `DBcm` module, and that's their strong recommendation to you. So, let's install that module first, then use it to interact with SQLite3.

Do this to follow along…

To begin, create a new notebook in your `webapp` folder called `CreateDatabaseTables.ipynb`, then execute this command the first code cell:

NOTE

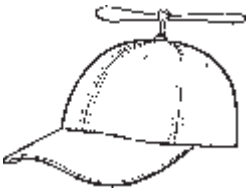
Earlier, we stated that nothing needed to be installed to use SQLite3 with Python… yet here we are, rather cheekily suggesting you install the "DBcm" module to simplify your interactions with SQLite3. Shocking, isn't it?!?

```
%pip install DBcm
```

A bunch of messages display on screen as `pip` does it thing. Not only does `DBcm` install, but you'll also see messages telling you the `mysql-connector-python` and `protobuf` modules are installed, too. DBcm uses these modules to also talk to MySQL and MariaDB database engines (which you'll get to in a later chapter).

For now, you're ready to talk to SQLite3 with `DBcm`. Let's learn now.

GEEK NOTE



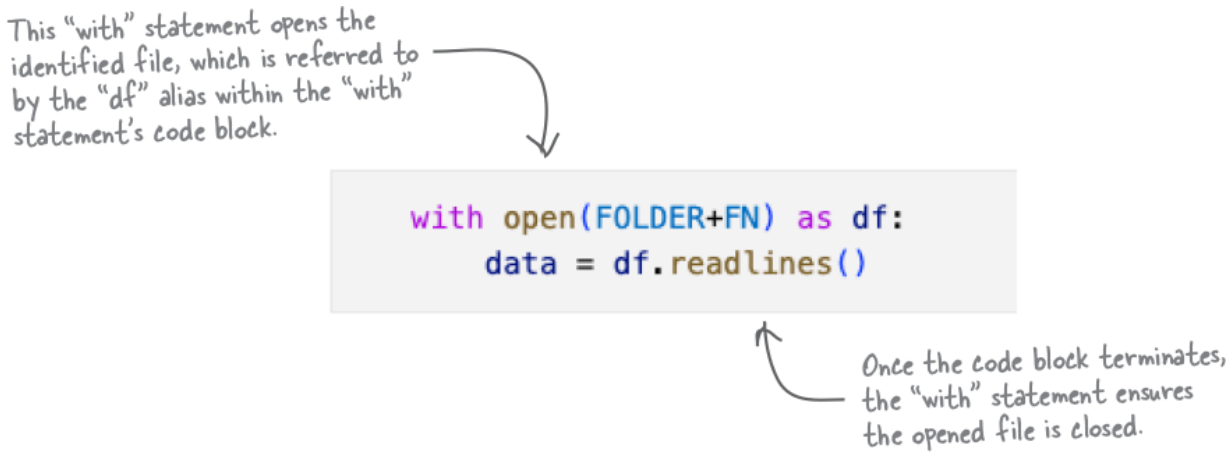
Details of how the `DBcm` module came into being are included in the second edition of *Head First Python*. Rather than repeat that material here, this edition concentrates on using `DBcm` to simplify interactions with the SQLite3 and/or MySQL-type database engines. You don't need to understand how `DBcm` works "under the hood" to take advantage of its functionality (as this and subsequent chapters demonstrate). However, if you're interested, the later chapters of this book's second edition dig into `DBcm`'s code (in detail) as the module is built.

Getting started with DBcm and SQLite3

The `DBcm` module builds on Python's `with` statement.

You first encountered the `with` statement back in Chapter 2, when the `Average.ipynb` notebook used `with` together with the **open**

BIF to manage the context within which your code interacted with a disk file. Here's the code you used then:



As you'll see, the `DBcm` module uses the `with` statement to talk to your database engine. Rather than opening then automatically closing a disk-file, `DBcm` opens a connection to your database engine, performs a database action, then automatically closes the connection, committing any database changes as needed.

To get going, import the `DBcm` module:



With the `DBcm` module imported, the next step is to provide the details of the database connection you wish to use. With SQLite3, all that's required here is the name of the file which contains your application's database. If the file already exists, SQLite3 uses it. If the file doesn't already exist, SQLite3 creates it for you.

The filename you use can be anything, but we like to use a name which clearly identifies the file as containing an SQLite3 database:

DBcm works alongside the "with" statement

Rather than working with the **open** BIF, the `DBcm` module provides the `UseDatabase` class which lets you talk – thanks to the **with** statement – to either an SQLite3 database file or a MySQL-like database engine (such as MySQL or MariaDB).

To demonstrate this mechanism, let's connect to the SQLite3 database engine associated with the filename identified by `db_details`, then issue the SQLite3 command `pragma table_list` to list the names of any tables available within the identified database.

NOTE

If you're familiar with MySQL/MariaDB, the "show tables" command does the same thing.

Here's the **with** statement which does just that, placing the outcome from the `pragma table_list` command into a variable called `results`:

Rather than using the "open" BIF, use DBcm's "UseDatabase" class instead, passing the name of the SQLite3 database as a parameter.

Similar to when using "open", an alias to the opened database connection is assigned to a variable, which is called "db" in this code.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute("pragma table_list")
    results = db.fetchall()
```

You can send a command to the database connection (identified by "db") using the "execute" function.

The "fetchall" function returns the output from the most-recently executed command as a list of tuples, which are assigned to a variable called "results" in this code.



That would make sense, wouldn't it?

But, to confirm this is indeed the case, let's take these last few lines of code for a *Test Drive*.

TEST DRIVE



With `DBcm` installed, you can run the code presented so far in this chapter in your notebook. The assumption, at the bottom of the last page, was that the `results` variable remains empty, which seems reasonable enough given that you've yet to create any tables. But, look what happens when the `results` variable is examined after the **with** statement runs:

```
import DBcm
```

```
db_details = "CoachDB.sqlite3"
```

```
with DBcm.UseDatabase(db_details) as db:
    db.execute("pragma table_list")
    results = db.fetchall()
```

```
results
```

```
[('main', 'sqlite_schema', 'table', 5, 0, 0),
 ('temp', 'sqlite_temp_schema', 'table', 5, 0, 0)]
```

We'll admit this output had us scratching our head... until, that is, we realized that SQLite3's "pragma table_list" also returns any tables which are internal to SQLite3. In this case, both "sqlite_schema" and "sqlite_temp_schema" are both examples of these internal tables, and you can safely ignore them.

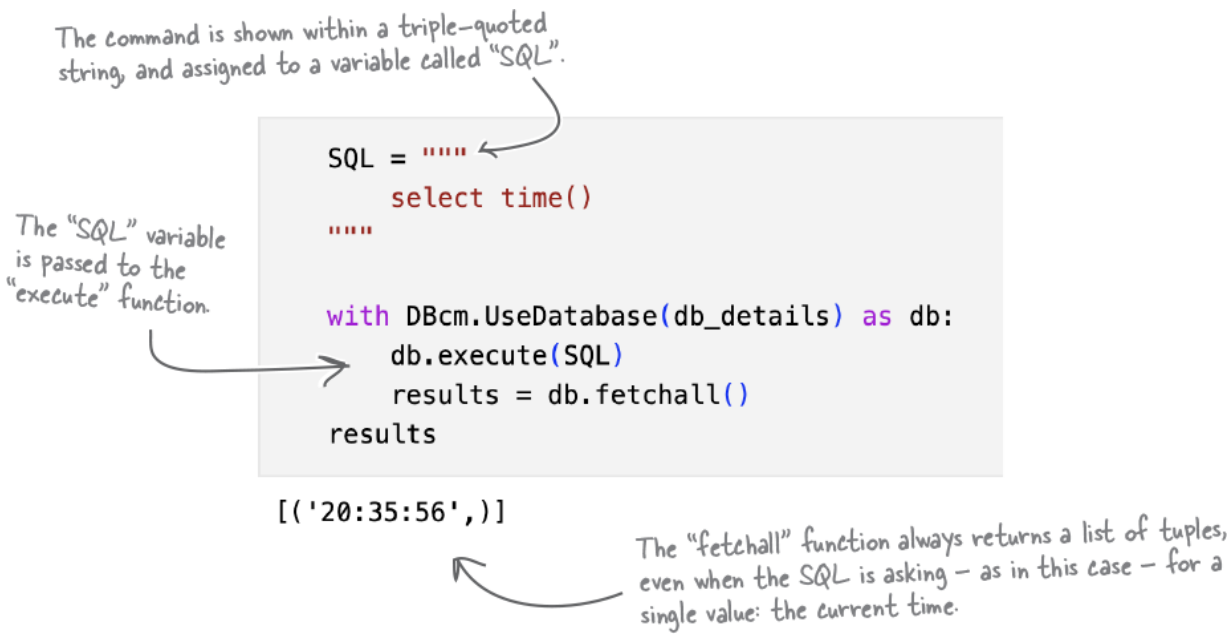
Look closely: "results" is a list of tuples.

If you are seeing something similar on your screen to what's shown above, then your `DBcm` module is working correctly, in that you are able to use the `execute` function to send a command to the database, then retrieve any outcome of the command executing using the `fetchall` function.

The two tables identified in the `results` list are used by SQLite3 to keep track of its internal state. Best to leave them alone, eh?

Use triple-quoted strings for your SQL

In the most-recent *Test Drive*, the command sent to the database was: `pragma table_list`. As the command was short, it made sense to include the entire command string as part of the `db.execute` call. However, as the command you send to your database gets longer (and more complex), it's often easier to read if the command is assigned to a variable, then shown within a triple-quoted string over multiple lines. Here's an example:



If you are sure a single value is being returned, you can use the `fetchone` function to return a single tuple from the database as op-

posed to a list of tuples (as is always the case with `fetchall`). Consider, for example, this version of the above code:

```
SQL = """
    select time()
"""

with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    result = db.fetchone()
result
```

Fetching one result, as opposed to all of them.

A single tuple of results is returned by "fetchone".

```
('20:53:44',)
```

THERE ARE NO DUMB QUESTIONS

Q: I'm with you so far, but for those commas included in the output from the displayed `results` and `result` variables. Those commas look decidedly weird. Is that an error?

A: No, it's not an error, but we agree that these do indeed look a little weird.

To understand what's going on, recall that `fetchall` returns a list of tuples. Two pages back, the output produced by the SQLite3 `pragma table_list` command is clearly a list of tuples. The returned list is enclosed in square brackets, with each value in the list separated from the next by a comma. In that example, the list values are themselves tuples, which are a collection of comma-separated values enclosed by parentheses.

Key point: lists contain square-bracket enclosed, comma-separated values, whereas tuples contain parentheses enclosed, comma-separated values.

When a list contains a single value, e.g., `[42]`, the square-brackets are enough for Python to successfully parse `[42]` as a single-element list.

All is good in the world until you consider what happens when a tuple contains a single value, for example, the number `42`. There's a

temptation to code this tuple as `(42)`, which won't work, as Python's rules state a tuple (to be a tuple) contains values separated from the next by a comma *even when there's only a single value in the tuple.* `(42)` has no comma, so Python parses it as the value `42`, which is likely not what you want if you were aiming for a single-element tuple. However, when a comma is added to give you this: `(42,)`, Python parses that as a single-element tuple. It does look very weird, but these are the rules…

The `fetchone` function, used at the bottom of the last page, returns a single result from the database. Instead of producing a list of tuples (as with `fetchall`), the `fetchone` function returns a single tuple. If that single tuple itself contains a single value, then the comma must be included for it to be a tuple.

Aren't you glad you asked? 😉

[We're off to have a wee lie down after that explanation].

Q: Just so I understand where all of this is heading, when we eventually get to using DBcm with MySQL or MariaDB, does all this code need to change?

A: No, but some edits will be required to the SQL that's used, as not all SQL dialects are created equally. What works with SQLite3 may not work with MySQL, nor with PostgreSQL, nor with Oracle, nor with

SQL Server, nor with… well, you get the idea. The vast majority of the SQL used by your application should work unchanged on any modern SQL-based database engine. However, they are edge-cases and incompatibilities to be wary off.

When you've worked out all the code (SQL and Python) that's needed to update the Coach's webapp to use SQLite3, you'll be required – in a later chapter – to deploy using MariaDB. You won't need to change any of your Python code as part of that exercise, but minor changes to your SQL will be needed, with emphasis on the word "minor" (so don't worry too much about this for now).

Q: So far all of the examples have returned some results output from the database engine, which `fetchall` or `fetchone` can grab as needed. What happens when the executed SQL doesn't produce any output, do `fetchall` and `fetchone` fail?

A: No, an empty list is returned in the case of `fetchall`, whereas Python's nothing value, `None`, is returned by the `fetchone` function.

Q: Is that it for fetch functions, or are there more?

A: There's also a `fetchmany` function which, based on the provided numeric parameter, returns that number of rows (i.e., tuples) from the

results returned from the database engine. The `fetchmany` function can be useful, for instance, when implementing pagination.

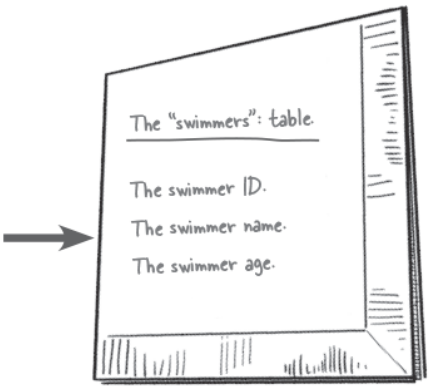
Not all SQL returns results

Of course, it's perfectly fine for an SQL command to produce no output. For instance, if you look at each of the `CREATE TABLE` commands that follow, none of them produce any output, although each of them adjusts the state of your database by adding a table.

Here are the SQL commands that create the `swimmers`, `events`, and `times` tables. Shown, too, are the "napkin structures" from earlier:

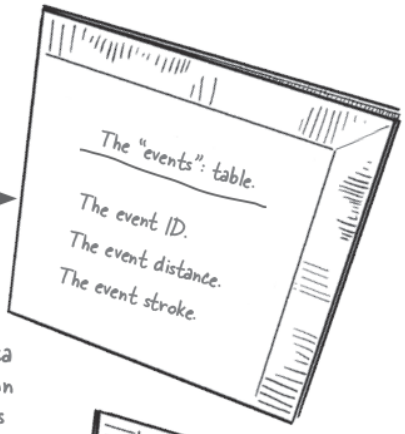
```
create table if not exists swimmers (
    id integer not null primary key autoincrement,
    name varchar(32) not null,
    age integer not null
)
```

This is SQLite3-specific SQL which likely won't work with other engines. Aren't standards wonderful?



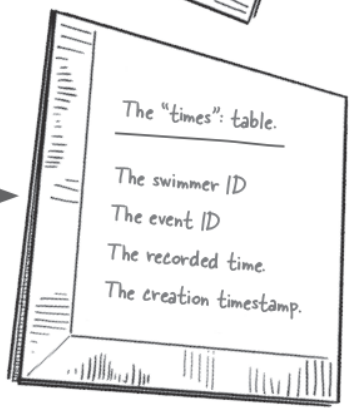

```
create table if not exists events (
    id integer not null primary key autoincrement,
    distance varchar(16) not null,
    stroke varchar(16) not null
)
```

The "not null" ensures data is provided for each column in every one of the table's rows.




```
create table if not exists times (
    swimmer_id integer not null,
    event_id integer not null,
    time varchar(16) not null,
    ts timestamp default current_timestamp
)
```

Whenever a row of data is added to the "times" table, the current timestamp is automatically generated.



EXERCISE



Here's the SQL which, when executed, creates the `swimmers` table:

```
create table if not exists swimmers (
    id integer not null primary key autoincrement
    name varchar(32) not null,
    age integer not null
)
```

Continuing to work within your `CreateDatabaseTables.ipynb` notebook, write code that assigns the above statement to a variable called `SQL`, then sends the query to your database for execution. Grab your pencil and write your code here:

_____

_____

_____

The above SQL produces no output. Can you think of a way to check that your SQL execution was successful? Write in your suggested strategy here:

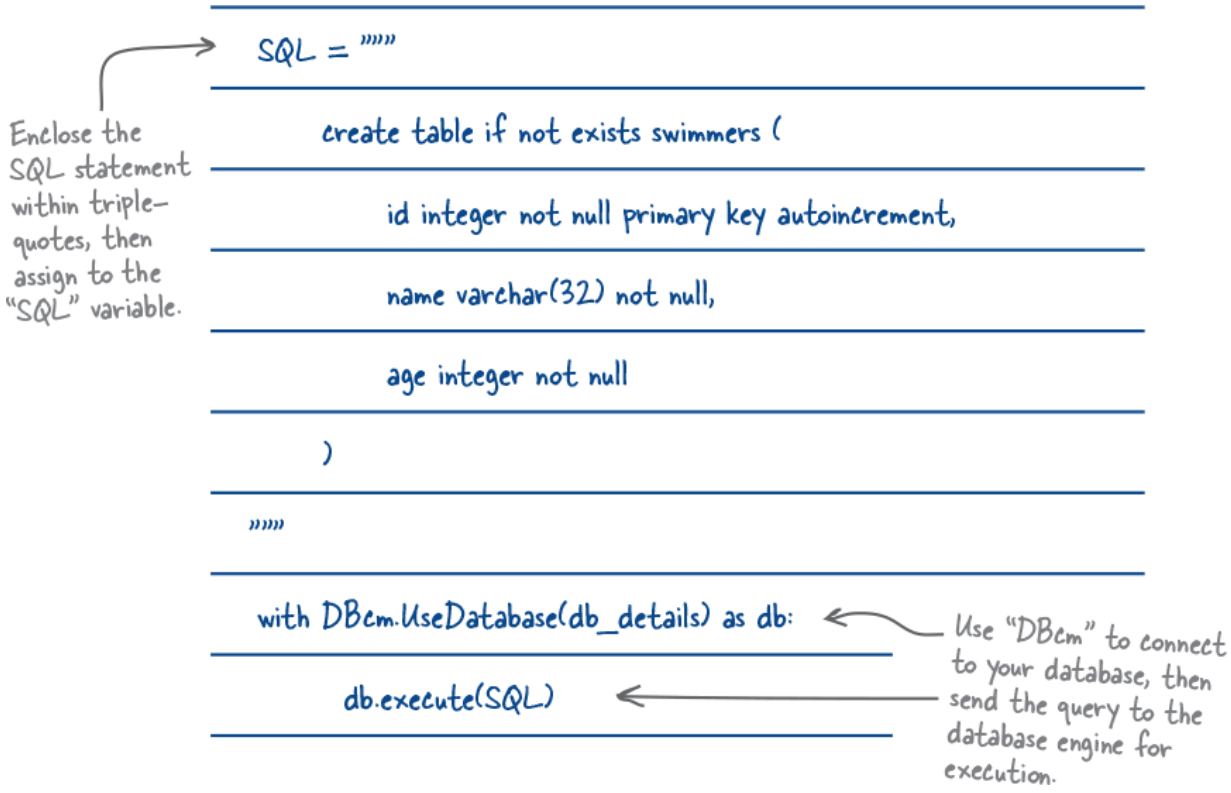
EXERCISE SOLUTION



Here's the SQL which, when executed, creates the `swimmers` table:

```
create table if not exists swimmers (
    id integer not null primary key autoincrement
    name varchar(32) not null,
    age integer not null
)
```

Continuing to work within your `CreateDatabaseTables.ipynb` notebook, you were to write code that assigns the above statement to a variable called `SQL`, then send the query to your database for execution. You were to write your code here:



The above SQL produced no output. You were to think of a way to check that your SQL execution was successful, then write in your suggested strategy. Here's what we thought to do:

Send "pragma tables_list" to the server, then view the returned results.

TEST DRIVE



Here's the code from the last *Exercise* which, when sent to your database engine, creates the `swimmers` table:

```
SQL = """
    create table if not exists swimmers (
        id integer not null primary key autoincrement,
        name varchar(32) not null,
        age integer not null
    )
"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
```

When you run the above code, no output is produced. The suggestion, at the bottom of the last page, was to run the `pragma tables_list` command (provided by SQLite3) to see if the list of tables in the database engine adjusts to indicate the changed state on the server. Let's send the command to the database engine and see:

This is the same code from earlier in the chapter.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute("pragma table_list")
    results = db.fetchall()
results
```

```
[('main', 'sqlite_sequence', 'table', 2, 0, 0),
 ('main', 'swimmers', 'table', 3, 0, 0),
 ('main', 'sqlite_schema', 'table', 5, 0, 0),
 ('temp', 'sqlite_temp_schema', 'table', 5, 0, 0)]
```

The information returned from the database engine has been updated to include data on the "swimmers" table. (There's also another internal SQLite3 table called "sqlite_sequence" which is best left alone). Note that the number "3" on the "swimmers" line indicates how many columns exist within that table (which is exactly what's expected here).

Create the "events" and "times" tables

Ready Bake Code



Now that you've created the `swimmers` table, it's not a big job to create the other two, namely the `events` and `times` tables. Here's some ready baked code which does this for you, together with another execution of `pragma tables_list` to confirm the three tables now exist within your database engine.

Rather than use multiple `with` statements here, a sequence of SQL statements are sent to the database engine within a single `with` block:

```
with DBcm.UseDatabase(db_details) as db:
    SQL_1 = """
        create table if not exists events (
            id integer not null primary key autoincrement,
            distance varchar(16) not null,
            stroke varchar(16) not null
        )
    """
    SQL_2 = """
        create table if not exists times (
            swimmer_id integer not null,
            event_id integer not null,
            time varchar(16) not null,
            ts timestamp default current_timestamp
        )
    """
    db.execute(SQL_1)
    db.execute(SQL_2)
```

Create two variables containing multi-line SQL statements.

Send each statement in sequence to your database engine.

```
    with DBcm.UseDatabase(db_details) as db:
        db.execute("pragma table_list")
        results = db.fetchall()
    results
```

```
[('main', 'times', 'table', 4, 0, 0),
 ('main', 'events', 'table', 3, 0, 0),
 ('main', 'sqlite_sequence', 'table', 2, 0, 0),
 ('main', 'swimmers', 'table', 3, 0, 0),
 ('main', 'sqlite_schema', 'table', 5, 0, 0),
 ('temp', 'sqlite_temp_schema', 'table', 5, 0, 0)]
```

Sending "pragma tables_list" to the database confirms the creation of the three tables: "swimmers", "events", and "times".

Your tables are ready (and Task #1 is done)

With the three tables created, Task #1 is done'n'dusted:

1 Decide on a structure for your data, then create your database tables

You need to decide how the data in your database is going to be arranged. Once you have decided on this, you can create the necessary database and tables using Python and SQL. ✔

Let's keep going, so go ahead and close the `CreateDatabaseTables.ipynb` notebook, then create a new notebook called `PopulateTables.ipynb`.

NOTE

Do this now.

Here's what's involved with Task #2:

2 Add your data values to your database tables

The Coach's system uses the data in the `swimdata` folder. You'll need to take the data from this folder and add it to the appropriate database tables. As in Task #1, Python and SQL are your go-to technologies here.



Yes. That's the plan: one table at a time.

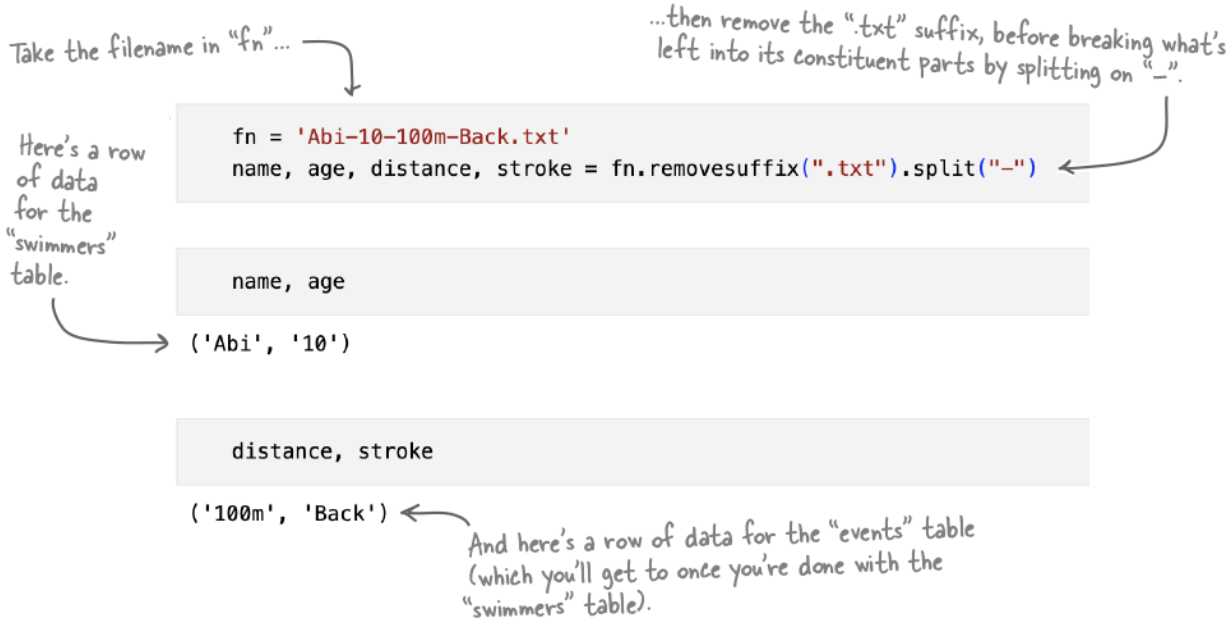
You'll start with the `swimmers` table, adding in a row of data identifying each of the Coach's swimmers.

Next up is the `events` table, where you'll create individual rows identifying each of the distance-stroke combinations.

Finally, once the `swimmers` and `events` tables are populated, it's time to tackle the `times` table, which is the most complex of the three population activities. Each row in the `times` table needs to identify a swimmer, an event, the recorded time in that event, as well as a timestamp identifying when the row was added to the table.

Determining the list of swimmer's files

Recall that the current version of the Coach's webapp works with the data in the `swimdata` folder. There's a bunch of files in that folder whose names conform to a very specific format, for example: `Abi-10-100m-Back.txt`. If this example filename is stored in a variable called `fn`, you already know that this code extracts the component parts:



For now, you're only interested in the `name` and `age` variables, as the values those variables contain can be used to populate the `swimmers` table. Of course, you'll want to determine these values for all the files in the `swimdata` folder. Once again, you've already used code to process all the files in your folder. These next few lines are taken from earlier in this book:



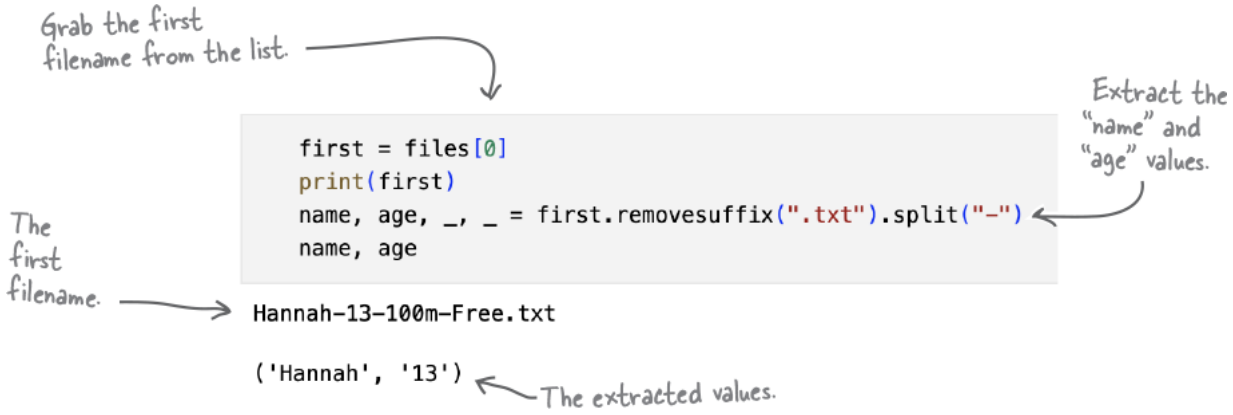
The `files` list contains all the swimmer's filenames from the `swimdata` folder. You can now take each file in `files`, extract the `name` and `age` values, then use the extracted values to add a row of data to your `swimmers` table.

Task #2: Adding data to a database table

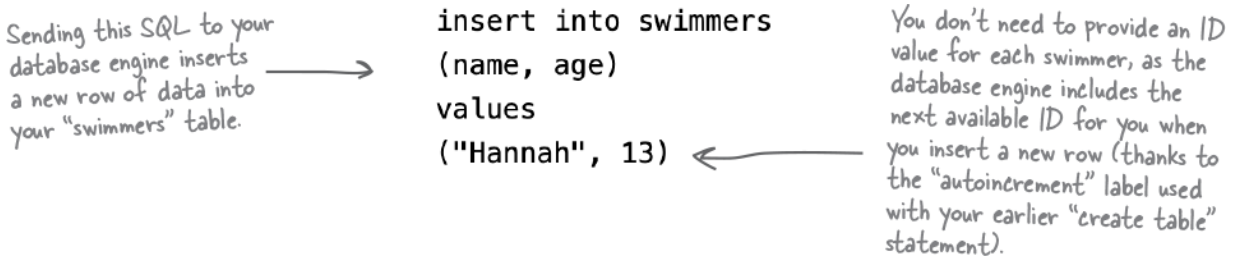
When it comes to adding data to an existing table, SQL provides the `INSERT` statement.

To get to the stage where you can add the swimmer's name and ages to the `swimmers` table, let work out how to do the `INSERT` for one swimmer. Then, once you can do it for one swimmer, you can repeat the process for all of them.

Working in your latest notebook, begin by assigning the first filename from the `files` list to a variable, then extract that filename's `name` and `age` values:



With the `name` and `age` values determined, your next step is to use the values in an `INSERT` statement, ultimately ending up with an SQL statement which look like this:



WATCH IT!



Don’t be tempted to use an f-string here!

If we were to ask you to create Python code to build the above INSERT statement as a string, you’d likely suggest using a triple-quoted f-string. Although this would work, it’s not the recommended approach when creating parameterized SQL statements. Instead, the Python database technologies provide **placeholders***, which do the same thing but in a much safer way.*



There's nothing wrong with using f-strings in most circumstances.

However, care is needed when it comes to programmatically building SQL statements.

There's an entire class of security vulnerabilities collectively known as *SQL Injection Attacks* which can be exploited when SQL statements are built using *string substitution* technologies, such as those employed internally by f-strings. To guard against these attacks, Python's database technologies support *parameter substitution using placeholders*. This mechanism negates any attempt to inject a potentially career-ending SQL attack into your application's code.

Stay safe with Python's SQL placeholders

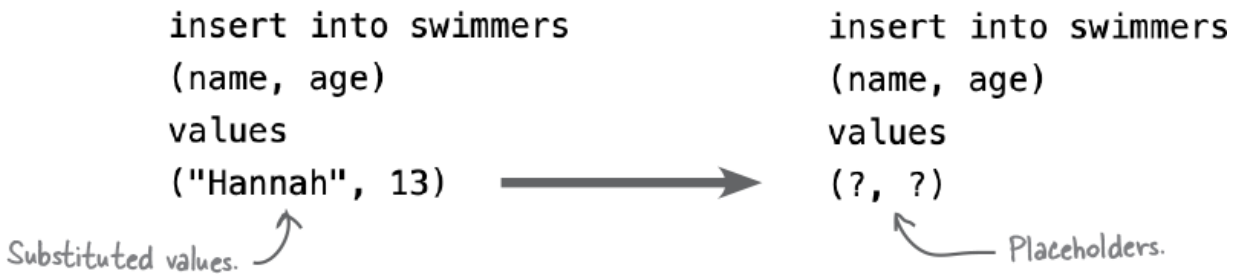
Rather than using *string* substitution, use Python's *parameter* substitution instead. Doing so is straightforward: When defining your SQL statement, insert a placeholder as opposed to a string substitution. In SQLite3, placeholders look like this:

```
?
```

That's not a typo. The placeholder symbol in SQLite3 is a single question mark, and your SQL statement can have as many place-

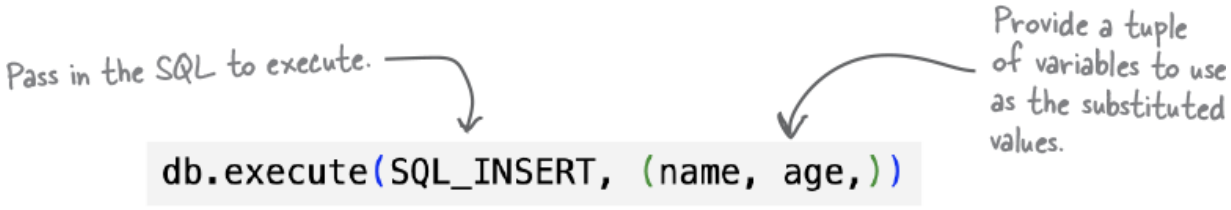
holders as required.

Here's the `INSERT` statement from earlier together with its placeholder equivalent:



Of course, when you eventually send your SQL statement to your waiting database engine, you'll want to replace the placeholders with the actual values to be used. This is accomplished by passing a tuple of values to the `execute` function, in addition to the actual SQL statement.

Assume the placeholder SQL shown above is assigned to a variable called `SQL_INSERT`. Additionally, assume the variables `name` and `age` exist and have been assigned values. The following line of code sends the SQL to your waiting database engine, substituting the values of the identified variables for the placeholders in a way which guards against *SQL Injection Attacks*:



All that remains is for you to sleep a little more soundly, safe in the knowledge that your use of placeholders has guarded your code, you, and your application's users from an *SQL Injection Attack*.

NOTE

Phew!

EXERCISE



Time for you to do a bit of work. There are five parts to this *Exercise*, with each requiring you to add code cells to your current notebook: `PopulateTables.ipynb`. Let's start off with a straightforward one.

1. Add code to import the `DBcm` module, then create the `db_details` variable assigning it the value `"CoachDB.sqlite3"`. Grab your pencil and write in the code you added here:

2 In your next code cell, create a variable called `SQL_INSERT`, then assign a triple-quoted string to it. Within the triple-quoted string, add the placeholder version of the INSERT statement from the previous page. Write in the code you used here:

3 Your next cell needs to use a **with** statement together with `DBcm` to connect to your SQLite3 database. A single line of code within your **with** statement's block executes the `SQL_INSERT` statement, passing in the `name` and `age` variables as placeholder substitutions. Write in the two lines of code you used here:

4 Now that you've added a row of data to the `swimmers` table, let's check the data was saved. Assign this SQL statement to your `SQL` variable: `select * from swimmers`, then create a **with** statement which sends the statement to your database, before retrieving the results and displaying them on screen. Write in the code you used in your next code cell here:

5 If all has gone well, your previous code cell confirms the addition (and storage) of a single row of data. Of course, you'll want to repeat the mechanism from the previous cell for all the filenames in the `swimdata` folder, not just the first one, and – you've guessed correctly – a loop is going to come in handy here. However, before you get to writing a loop, you'll need to remove the row you've just added to the `swimmers` table, as that was a quick test of the mechanism, and you'll want to start with an empty table when you get to writing your loop.

The SQL statement `delete from swimmers`, when sent to the database, removes all stored data from the `swimmers` table. In your next code cell, create code which send this SQL statement to SQLite3. Note that your table is emptied of data, but still exists (i.e., the table is not removed).

Write in the code you used here:

To confirm the deletion worked, re-run the code cell created for  to confirm the `swimmers` table is empty, that is: don't copy and paste, click on the cell then press **Ctrl+Enter** to re-run the cell's code.

EXERCISE SOLUTION



It was time for you to do a bit of work. There were five parts to this *Exercise*, with each part adding code cells to your current notebook: `PopulateTables.ipynb`. You started off with a straightforward one.

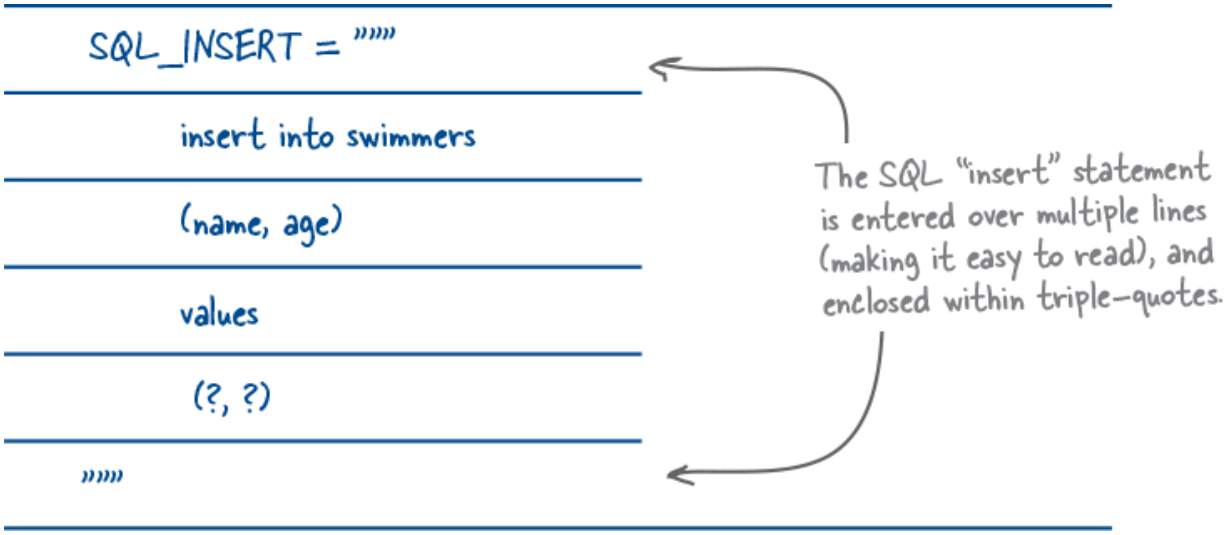
1 You were to add code to import the `DBcm` module, then create the `db_details` variable with the value `"CoachDB.sqlite3"`. You were to write in the code you added here:

```
import DBcm

db_details = "CoachDB.sqlite3"
```

This is as it was in your previous notebook.

2 In your next code cell, you were to create a variable called `SQL_INSERT`, then assign a string to it. Within the string, you were to add the placeholder version of the INSERT statement from the a few pages ago. You were to write in the code you used here:



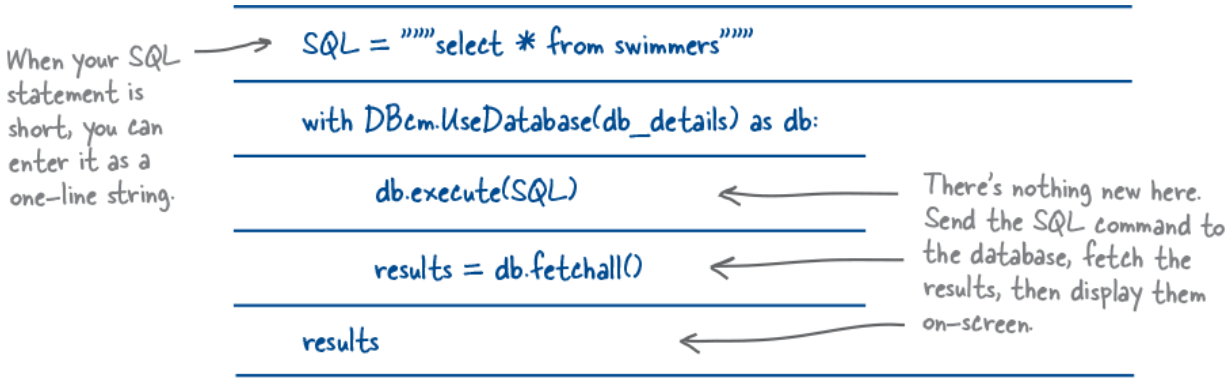
3 Your next cell needs was to use a **with** statement together with `DBcm` to connect to your SQLite3 database. A single line of code within your **with** statement was to execute the `SQL_INSERT` statement, passing in the `name` and `age` variables as placeholder substitutions. You were to write in your code here:

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_INSERT, (name, age,))
```

Connect to the database, then send the SQL statement together with the two parameter values.

4 Now that you'd added a row of data to the `swimmers` table, you were to check the data was saved by assigning this SQL statement to your `SQL` variable: `select * from swimmers`. Using a **with** statement, you were to send the statement to your database, before

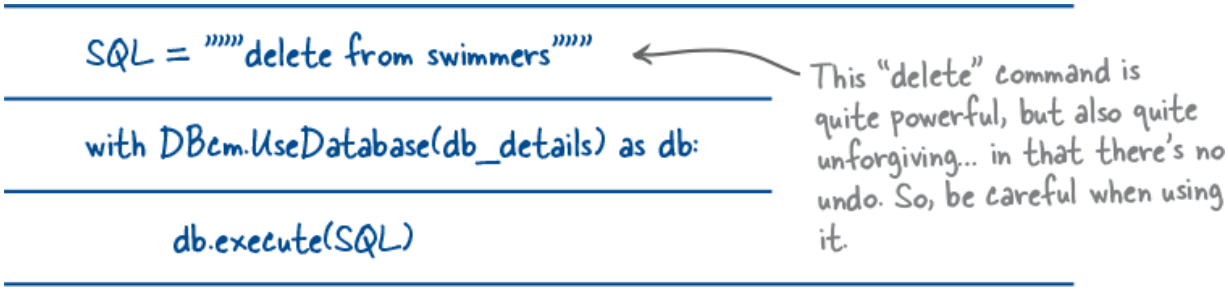
retrieving the results and displaying them on screen. You were to write in the code you added to your next code cell here:



5 If all went well, your previous code cell confirmed the addition (and storage) of a single row of data. Of course, you'll want to repeat the mechanism from the previous cell for all the filenames in the `swimdata` folder, not just the first one, and – you've guessed correctly – a loop is going to come in handy for that. However, before you get to writing a loop, you were to remove the row you've just added to the `swimmers` table, as that's a quick test of the mechanism. Doing so ensures you'll start with an empty table when you get to writing your loop.

The SQL statement `delete from swimmers`, when sent to the database, removes all stored data from the `swimmers` table. In your next code cell, you were to create code which send this SQL statement to SQLite3. Note that your table is emptied of data, but still exists (i.e., the table is not removed).

You were to write in the code you used here:



To confirm the deletion worked, re-run the code cell created for 4 to confirm the `swimmers` table is empty, that is: don't copy and paste, click on the cell then press **Ctrl+Enter** to re-run the cell's code.

NOTE

Let's run all this code in a "Test Drive" to confirm it works as expected.

TEST DRIVE



This code assumes `name` and `age` have been set to values extracted from the first filename in the `files` list (as read from the `swimdata` folder). Each chunk of code shown is entered into it's own cell in our notebook, except for the last chunk of code at the bottom of the next page, where you were asked to click on the cell from earlier, then re-run the code by pressing **Ctrl+Enter**.

As before, the "DBcm" module is imported, then the connection details for the database are defined.

```
import DBcm

db_details = "CoachDB.sqlite3"
```

The parameterized "insert" statement defines two placeholders for the actual "name" and "age" values to add as a row of data into the "swimmers" table.

```
SQL_INSERT = """
    insert into swimmers
    (name, age)
    values
    (?, ?)
"""
```

The "insert" statement is sent to the database, together with two required values (passed as a tuple).

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_INSERT, (name, age,))
```

The "insert" statement is the strong-silent type, in that there's no response from the database to indicate success. Generally, if there's an error (or a problem), you'll hear about it, whereas everything else results in silence...

```
SQL = """select * from swimmers"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchall()
results
```

```
[(1, 'Hannah', 13)]
```

When you ask the database to return all the rows of data in the "swimmers" table, not only do you get back the name and age values ("Hannah" and 13), but there's also an ID value of 1 returned. This value was automatically added by SQLite3 because you specified this column as "autoincrement". Neat, eh? Note that the next row of data will be numbered 2, then 3, and so on, until you run out of rows, data, disk space, or the will to live... 😉

```
SQL = """delete from swimmers"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
```



Removing all the rows from a specific table happens quietly within the database. To actually see what happened, you need to ask again to see all the rows of data.

```
SQL = """select * from swimmers"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchall()
results
```

```
[]
```



Click on your earlier cell, then press "Ctrl-Enter" to re-run the cell. Unlike at the top of this page, the "swimmers" table is now empty, which results in an empty list. This means there's no rows of data in the "swimmers" table, but the table still exists.



Yes, that sounds like a plan.

Let's quickly do just that and see what happens. Grab your pencil and see you on the next page.

EXERCISE



All of the filenames from the `swimdata` folder are in the `files` list in your notebook. You can loop through the `files` list assigning each individual filename to the `fn` loop variable, which can then have its ".txt" suffix removed, before being broken apart on the "-" symbol. This gives you the ability to populate the `name` and `age` variables, which can then be substituted into an `INSERT` statement and sent to your waiting database engine.

Grab your pencil (and your notebook) and work out the code to perform the above task, which should add the list of swimmers to your `swimmers` table. Write in the code you came up with in the space below.

```
import os

import DBcm

db_details = "CoachDB.sqlite3"

FOLDER = "swimdata/"

files = os.listdir(FOLDER)
files.remove(".DS_Store")

SQL_INSERT = """
    insert into swimmers
    (name, age)
    values
    (?, ?)
"""
```

To let you concentrate on the task at hand, we're showing you this code which sets up things for you, putting the list of swimmer filenames into the "files" list and defining the *INSERT* statement as a triple-quoted string. Your job is to write the loop code which adds all the swimmers details to the "swimmers" table.

EXERCISE SOLUTION



All of the filenames from the `swimdata` folder are in the `files` list in your notebook. You can loop through the `files` list assigning each individual filename to the `fn` loop variable, which can then have its ".txt" suffix removed, before being broken apart on the "-" symbol. This gives you the ability to populate the `name` and `age` variables, which can then be substituted into an `INSERT` statement and sent to your waiting database engine.

Grab your pencil (and your notebook) and work out the code to perform the above task, which should add the list of swimmers to your `swimmers` table. Write in the code you came up with in the space below.

```
import os

import DBcm

db_details = "CoachDB.sqlite3"

FOLDER = "swimdata/"

files = os.listdir(FOLDER)
files.remove(".DS_Store")

SQL_INSERT = """
    insert into swimmers
    (name, age)
    values
    (?, ?)
"""
```

To let you concentrate on the task at hand, we're showing you this code which sets up things for you, putting the list of swimmer filenames into the "files" list and defining the INSERT statement as a triple-quoted string. Your job was to write the loop code which adds all the swimmers details to the "swimmers" table.

Here's the code we came up with. How does it compare to yours?

```
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, _, _ = fn.removesuffix(".txt").split("-")
        db.execute(SQL_INSERT, (name, age, ))
```

BRAIN POWER



Take a moment to look at the way we wrote our code at the bottom of the previous page, with the **with** statement enclosing the **for** loop (and not the other way around). Can you think of a reason for why we did this?

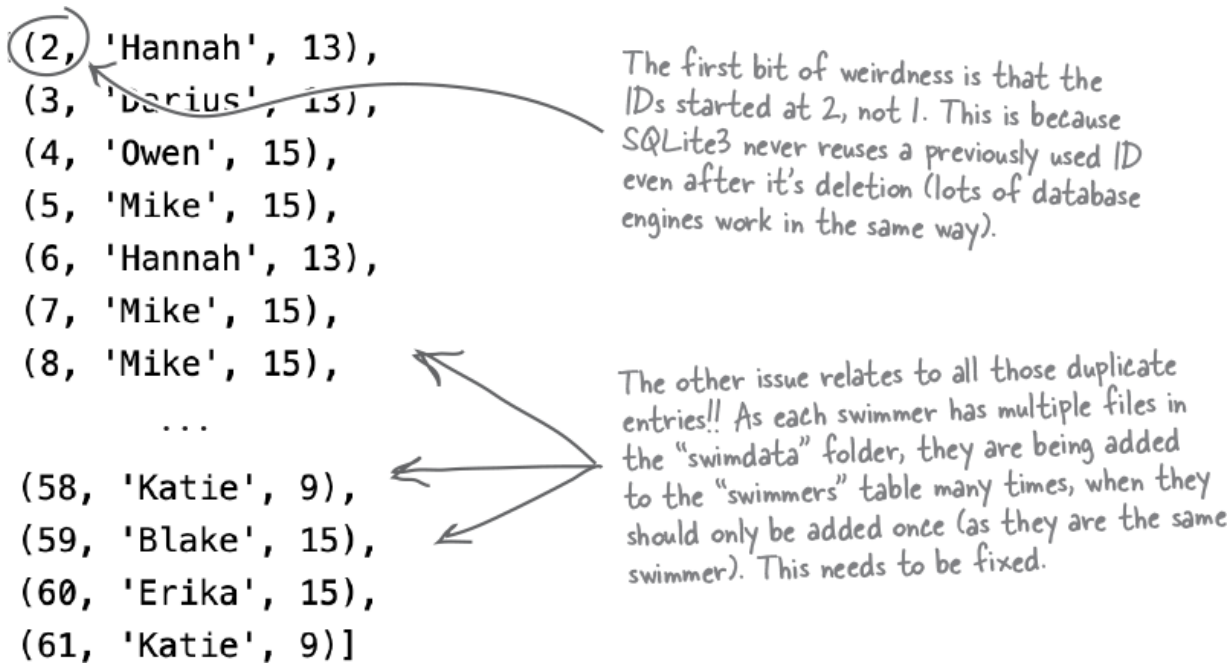
TEST DRIVE



Here's our code, which works, but maybe not in exactly the way we had hoped:

```
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, _, _ = fn.removesuffix(".txt")
        db.execute(SQL_INSERT, (name, age, ))
```

When we **Ctrl+Click** on the code cell which re-runs and displays all the data, we see this:


(2, 'Hannah', 13),
(3, 'Darius', 13),
(4, 'Owen', 15),
(5, 'Mike', 15),
(6, 'Hannah', 13),
(7, 'Mike', 15),
(8, 'Mike', 15),
...
(58, 'Katie', 9),
(59, 'Blake', 15),
(60, 'Erika', 15),
(61, 'Katie', 9)]
The first bit of weirdness is that the IDs started at 2, not 1. This is because SQLite3 never reuses a previously used ID even after it's deletion (lots of database engines work in the same way).
The other issue relates to all those duplicate entries!! As each swimmer has multiple files in the "swimdata" folder, they are being added to the "swimmers" table many times, when they should only be added once (as they are the same swimmer). This needs to be fixed.



Yes, we need to avoid duplicate entries.

It's the combination of the name value *and* the age value which ensures uniqueness, so you need to check against *both*.

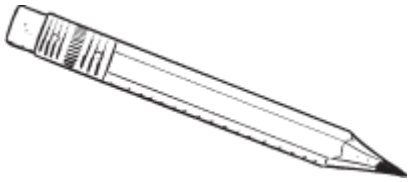
To understand what might go wrong, consider if, as well as the under 13 year old Darius, another – younger – swimmer *also* called Darius joins the club. The younger Darius swims in the under 10 age group, so to distinguish him from the under 13 Darius, you need to search on *both* name and age.

You also have to consider what might happen next year, when a bunch of swimmers move up to their next age group. As an example, Darius will swim in the under 14 age group, and you'll want to keep his under 14 data separate from his under 13 data, so – in that case – you'll want "both" swimmers in your database table.

NOTE

Things get even more complex if *another* Darius swimming in the under 13 age group joins the club. But, as this is a book on Python as opposed to a book on database application design, management, and storage, we're going to keep things simple for now and assume such a situation won't ever occur with the Coach's webapp (whereas it most likely will happen in the "real world").

SHARPEN YOUR PENCIL



Grab your pencil once more. Find below two parameterized SQL statements. The first, `SQL_SELECT`, returns all the data from the swimmers table while filtering on name and age values. You've seen the second statement before, `SQL_INSERT`, which add a rows of data to the `swimmers` table.

Your job is to adjust your code to insert a new row into the `swimmers` table when the data is new (i.e., it doesn't already exist within the table). As a hint, recall that `fetchall` returns an empty list when your SQL query returns no results. Work out the code you need, then write it into the space below.

```
import os

import DBcm

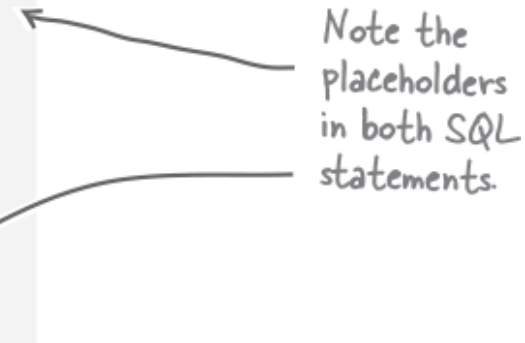
db_details = "CoachDB.sqlite3"

FOLDER = "swimdata/"

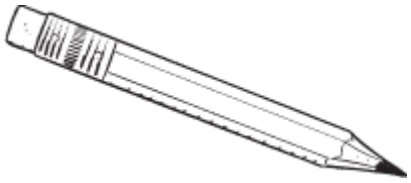
files = os.listdir(FOLDER)
files.remove(".DS_Store")

SQL_SELECT = """
    select * from swimmers
    where name = ? and age = ?
"""

SQL_INSERT = """
    insert into swimmers
    (name, age)
    values
    (?, ?)
"""
```



SHARPEN YOUR PENCIL SOLUTION



You were to grab your pencil once more, then review the two provided parameterized SQL statements shown below. The first, `SQL_SELECT`, returns all rows data from the swimmers table while filtering on `name` and `age`, while the second, `SQL_INSERT`, adds a row of data to the `swimmers` table. Your job was to adjust your code to insert a row into the `swimmers` table when the data is new (i.e., it doesn't already exist within the table). You were to recall that `fetchall` returns an empty list when your SQL query returns no results. You were to work out the code you needed, then write it into the space below.

```
import os

import DBcm

db_details = "CoachDB.sqlite3"

FOLDER = "swimdata/"

files = os.listdir(FOLDER)
files.remove(".DS_Store")

SQL_SELECT = """
    select * from swimmers
    where name = ? and age = ?
"""

SQL_INSERT = """
    insert into swimmers
    (name, age)
    values
    (?, ?)
"""
```

The trick to understanding this code is to recall that a non-empty list in Python evaluates to TRUE, whereas an empty list evaluates to FALSE. Look how this code exploits this behavior in its "if" statement.

```
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, _, _ = fn.removesuffix(".txt").split("_")
        db.execute(SQL_SELECT, (name, age,))
        if db.fetchall():
            continue
        db.execute(SQL_INSERT, (name, age,))
```

When called, the "fetchall" function returns a list of tuples when data exists, or an empty list when the data cannot be found.

The "continue" statement causes the current loop iteration to terminate early.

Data is only inserted when it cannot be found.

TEST DRIVE



Let's see what difference your latest code makes to your database insertions. If you haven't done so already, select the code cell which runs `delete from swimmers` and re-run it (by pressing **Ctrl+Enter**). This empties the `swimmers` table of all data prior to running your latest code. When you're ready, return to the code cell which contains your latest code, then press **Shift+Enter**. Then return to the cell which runs `select * from swimmers`, pressing **Ctrl+Enter** to re-run that code. You should see a much shorter list of swimmers returned from the database, as shown below:

```
    with DBcm.UseDatabase(db_details) as db:
        for fn in files:
            name, age, _, _ = fn.removesuffix(".txt").split("-")
            db.execute(SQL_SELECT, (name, age,))
            if db.fetchall():
                continue
            db.execute(SQL_INSERT, (name, age,))
```

The latest version of your insertion code for the "swimmers" table.

```
[(62, 'Hannah', 13),
 (63, 'Darius', 13),
 (64, 'Owen', 15),
 (65, 'Mike', 15),
 (66, 'Abi', 10),
 (67, 'Ruth', 13),
 (68, 'Tasmin', 15),
 (69, 'Erika', 15),
 (70, 'Maria', 9),
 (71, 'Elba', 14),
 (72, 'Ali', 12),
 (73, 'Chris', 17),
 (74, 'Aurora', 13),
 (75, 'Katie', 9),
 (76, 'Alison', 14),
 (77, 'Emma', 13),
 (78, 'Calvin', 9),
 (79, 'Blake', 15),
 (80, 'Bill', 18),
 (81, 'Dave', 17),
 (82, 'Lizzie', 14),
 (83, 'Carl', 15)]
```

The data retrieved from the database when you ask it to return all the rows from the "swimmers" table. Note that there are no duplicated entries in these results. Also, note, that the ID values start at 62, as the previous 61 rows have been deleted. This might look weird, but isn't really, as the most important thing is that each row has a unique ID.

Let’s repeat this process for the events

Now that you have code that adds rows of data to the `swimmers` table, it’s not a huge effort to produce similar code for your `events` table.



That sounds like a good idea, but may be overkill.

If you take a moment to look at the code which adds unique rows of data to the `swimmers` table, it’s not hard to see what has to change to adjust this code to do a similar thing for the `events` table. And, of course, the *Software Engineering* part of your brain immediately

thinks “reusable function”. After all, if the code only needs to be adjusted slightly, abstracting those differences away into a function which you then pass in parameter values to (as needed) sounds like a good idea.

Job done.

Although, in this case, it isn’t (trust us, we tried). It is possible to create a function here, but the effort required quickly turns into a tangled mess which can be hard to read and reason about. So, despite what your inner software engineer is telling you, let’s reach for copy’n’paste here…

EXERCISE



Below you'll find the code which adds data to your `swimmers` table, but with all the references to the `swimmers` table removed (i.e., blanked out). In the indicated spaces (which are *underlined*) write in the replacement values if you were converting this copy'n'pasted code to work with your `events` table.

```
import os

import DBcm

db_details = "CoachDB.sqlite3"

FOLDER = "swimdata/"

files = os.listdir(FOLDER)
files.remove(".DS_Store")

SQL_SELECT = """
    select * from ______
    where ______ = ? and ______ = ?
"""

SQL_INSERT = """
    insert into ______
    ( ______, ______ )
    values
    (?, ?)
"""

with DBcm.UseDatabase(db_details) as db:
    for fn in files:

        _, _, ______, ______ = fn.removesuffix(".
        db.execute(SQL_SELECT, (______,______,))
```

```
        if db.fetchall():
            continue
        db.execute(SQL_INSERT, (______,______,))
```

Hint: Use the **Esc then C** key combination, followed by **Esc then V** to copy'n'paste your `swimmers` code cell, then make your suggested changes to your new code cell based on what you've entered above.

EXERCISE SOLUTION



After using **Esc then C**, followed by **Esc then V** to copy'n'paste your `swimmers` code cell, you were to adjust the copied code, entering values in the indicated spaces (blanked out and underlined).

Unlike the previous code which adds data to your `swimmers` table, you were to write in the replacement values as if you were converting this copy'n'pasted code to work with your `events` table. Here's the changes we made:

```
import os

import DBcm

db_details = "CoachDB.sqlite3"

FOLDER = "swimdata/"

files = os.listdir(FOLDER)
files.remove(".DS_Store")

SQL_SELECT = """
    select * from events
    where distance = ? and stroke = ?
"""

SQL_INSERT = """
    insert into events
    ( distance , stroke )
    values
    (?, ?)
"""

with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        _, _, distance , stroke = fn.removesuffix(".txt").split("-")
        db.execute(SQL_SELECT, ( distance , stroke ,))
        if db.fetchall():
            continue
        db.execute(SQL_INSERT, ( distance , stroke ,))
```

Replace all the references to the "swimmers" table with the table named "events".

Instead of the "name" value, this code refers to the "distance" value instead.

Instead of the "age" value, this code refers to the "stroke" value instead.

There's a subtle change here. Did you spot it?

TEST DRIVE



Here's the copy'n'pasted code with the `events` table amendments applied. Press **Shift+Enter** to execute it:

```
SQL_SELECT = """
    select * from events
    where distance = ? and stroke = ?
"""

SQL_INSERT = """
    insert into events
    (distance, stroke)
    values
    (?, ?)

"""

with DBcm.UseDatabase(db_details) as db:
    for fn in files:

        _, _, distance, stroke = fn.removesuffix(
        db.execute(SQL_SELECT, (distance, stroke,
```

```
            if db.fetchall():
                continue
            db.execute(SQL_INSERT, (distance, stroke,
```

A small adjustment to the code which displays all the data from `swimmers` table confirms all is OK:

```
SQL = """select * from events"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchall()
results
```

```
[(1, '100m', 'Free'),
 (2, '100m', 'Back'),
 (3, '100m', 'Fly'),
 (4, '50m', 'Back'),
 (5, '200m', 'Free'),
 (6, '200m', 'Back'),
 (7, '50m', 'Free'),
 (8, '50m', 'Breast'),
 (9, '200m', 'IM'),
 (10, '100m', 'Breast'),
 (11, '400m', 'Free'),
 (12, '50m', 'Fly'),
 (13, '200m', 'Breast')]
```

There are no duplicate event entries as your copy'n'pasted code already checks for that. Unlike the "swimmers" data, the IDs in the "events" table start at 1 as you didn't have to experiment with inserts beforehand. Although you may have already noticed, note that database engines start counting from one, not zero (presumably in an effort to keep everyone's life interesting).

All that's lef t is your times table…

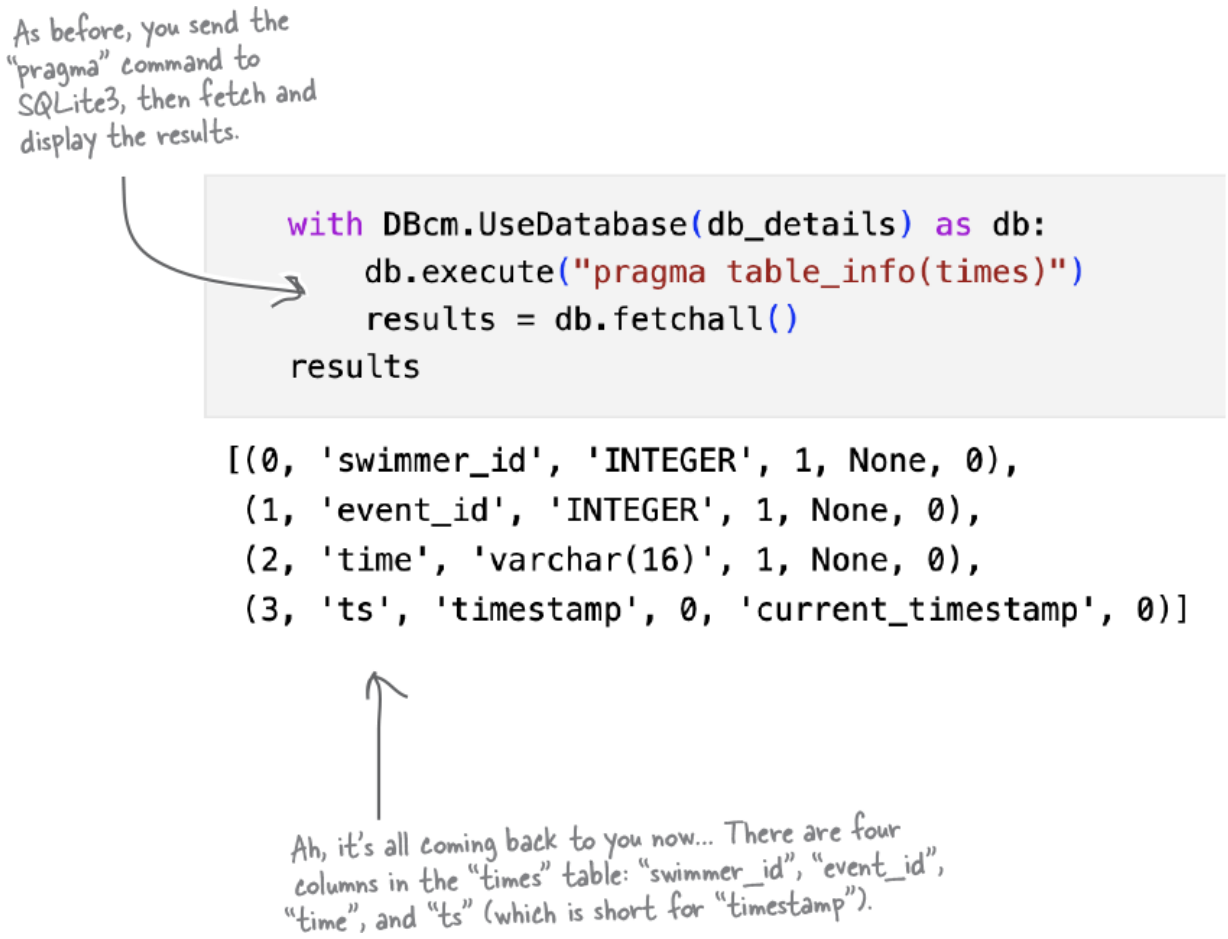
With the `swimmers` and `events` tables populated with data, all that remains is to populate the `times` table with data.

Rather than have you flip back through this chapter in search of the `CREATE TABLE` command for `times`, let's ask SQLite3 to fess-up what it knows about this table. There's another `pragma` command, `table_info`, which can help here.

Run the `pragma table_info` command in your next code cell to be reminded of the `times` table's structure:

NOTE

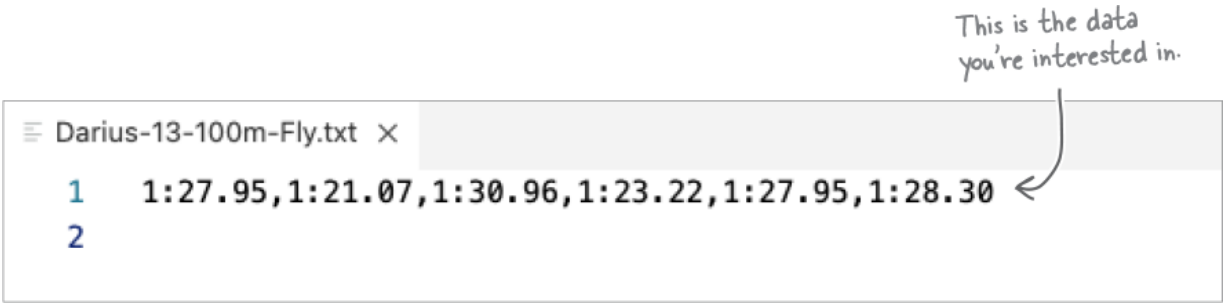
MySQL/MariaDB has a similar command called "describe".



The values to use for `swimmer_id` and `event_id` relate to the ID columns in the `swimmers` and `events` tables, respectively, so they should be easy to find. The `ts` value is automatically entered by the SQLite3 database engine when a row of data is added to `times`, so there are no worries there (as this is handled for you). But… where does the `time` value come from?

The times are in the swimmer's files…

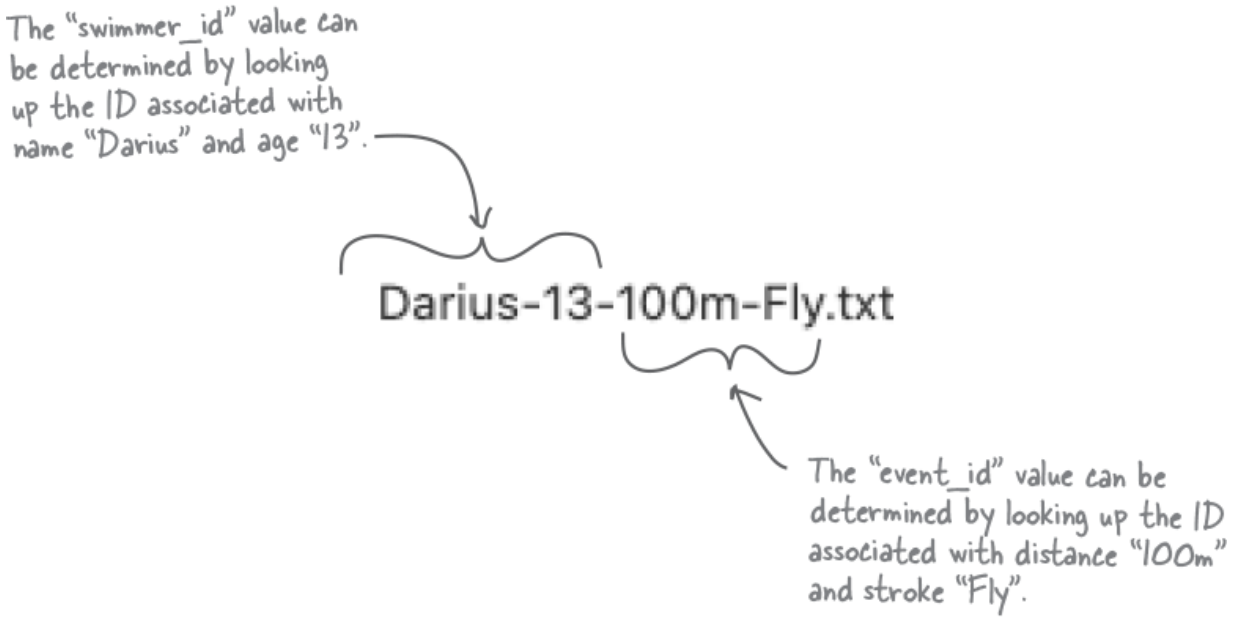
Recall the contents of any one of the swimmer's files. Here's the contents of one of the files associated with Darius:



Consider what you've got here.

There's the list of (in this example) six recorded swim times. These times are the data which needs to be added to the `times` table, one row per time shown.

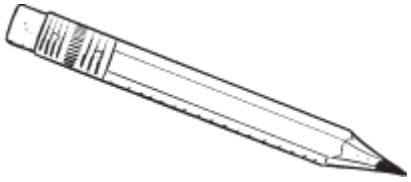
For each row, the `swimmer_id` and `event_id` values need to be determined from the information contained in the filename:



Once the two IDs are found, it's merely a matter of taking the file's contents (that string of times), breaking them apart on the comma symbol, then inserting a new row for each of the times values.

Let's do that right now.

SHARPEN YOUR PENCIL



Here's a reminder of some of the code which has already executed in your notebook. First up is the code which grabs the list of filenames

from your `swimdata` folder:

```
import os

FOLDER = "swimdata/"

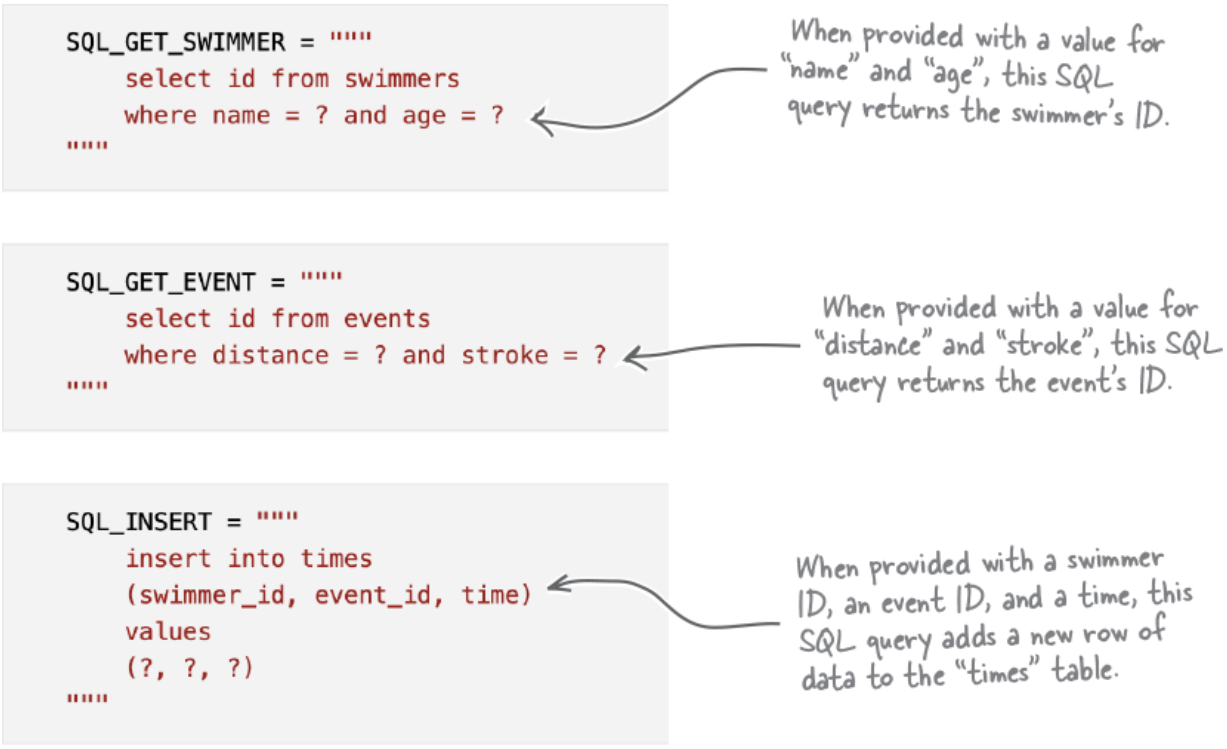
files = os.listdir(FOLDER)
files.remove(".DS_Store")
```

In addition to the above code, you've also imported the `DBcm` module and identified your SQLite3 database file:

```
import DBcm

db_details = "CoachDB.sqlite3"
```

Recall, too, that you've written and executed code which populates the `swimmers` and `events` tables with data. With that in mind, take a look at three new SQL statements which, when provided with values for each of their placeholders, are going to help you populate the `times` table with data:



Now, grab your pencil.

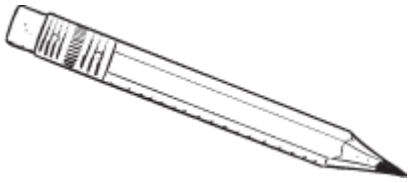
Here's a **with** statement which opens a connection to your database file. The first line of code in the **with** statement's code block loops over all the filenames in your `files` list, then (within the loop's code block) values for `name`, `age`, `distance`, and `stroke` are determined:

```
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, distance, stroke = fn.removesu
```

Your task is to complete the loop's code block to arrange for the population of the `times` table with data. The SQL queries shown on the previous page should help you here. Experiment as needed in your notebook then, when you think your code is working correctly, write it in here:

Hint: as well as the `fetchall` method (which hands you all the rows returned by a query), Python's `sqlite3` library also supports `fetchone` and `fetchmany` (which work differently). The `sqlite3` documentation has all the details.

SHARPEN YOUR PENCIL SOLUTION



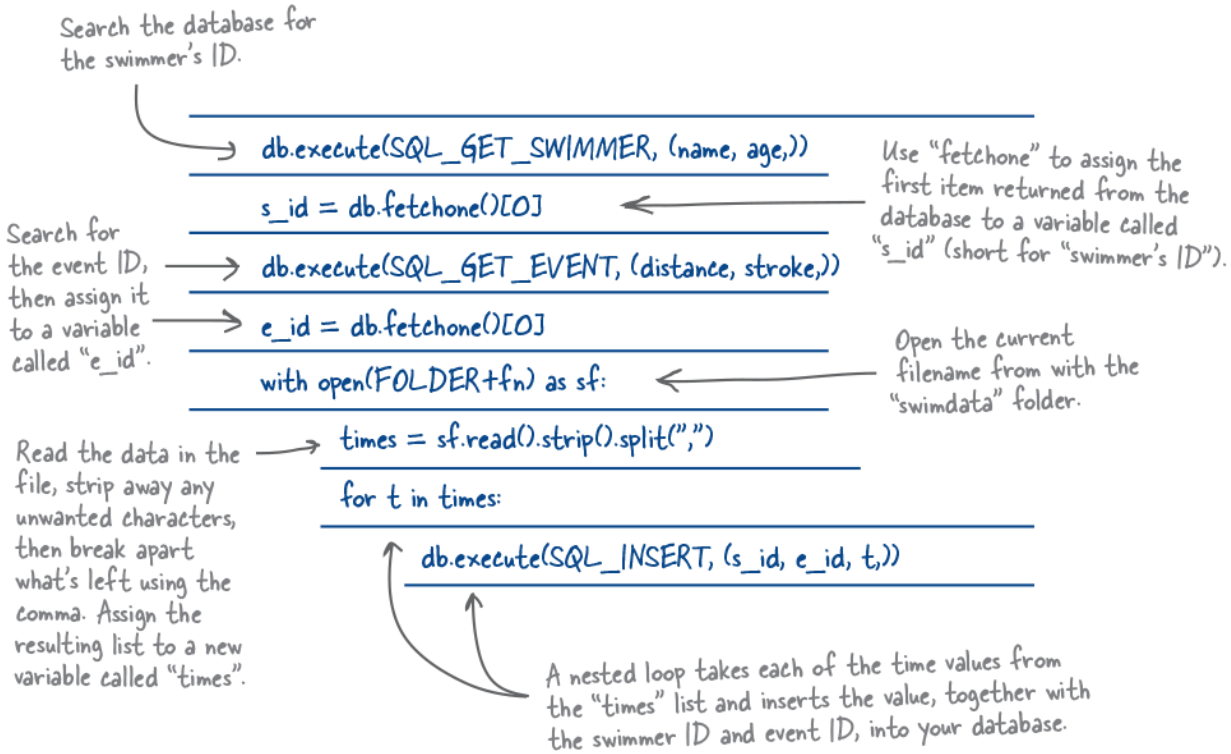
You were to grab your pencil.

You were given a **with** statement which opens a connection to your database file. The first line of code in the **with** statement's code block loops over all the filenames in your `files` list, then (within the loop's code block) values for `name`, `age`, `distance`, and `stroke` are determined:

```
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, distance, stroke = fn.removesu
```

Your task was to complete the loop's code block to arrange for the population of the `times` table with data. The provided SQL queries would help you here. You were to experiment as needed in your note-book then, when you thought your code is working correctly, you were to write it in here:

Here's the code we came up with. How does this compare to your code?



TEST DRIVE



When you run your `times` updating code, there's no output, as the code generates no messages (if all is well). To see what happened, begin by asking your database to tell you how many rows are now in your `times` table:

```
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, distance, stroke = fn.removesuffix(".txt").split("-")
        db.execute(SQL_GET_SWIMMER, (name, age,))
        s_id = db.fetchone()[0]
        db.execute(SQL_GET_EVENT, (distance, stroke,))
        e_id = db.fetchone()[0]
        with open(FOLDER+fn) as sf:
            times = sf.read().strip().split(",")
            for t in times:
                db.execute(SQL_INSERT, (s_id, e_id, t,))
```

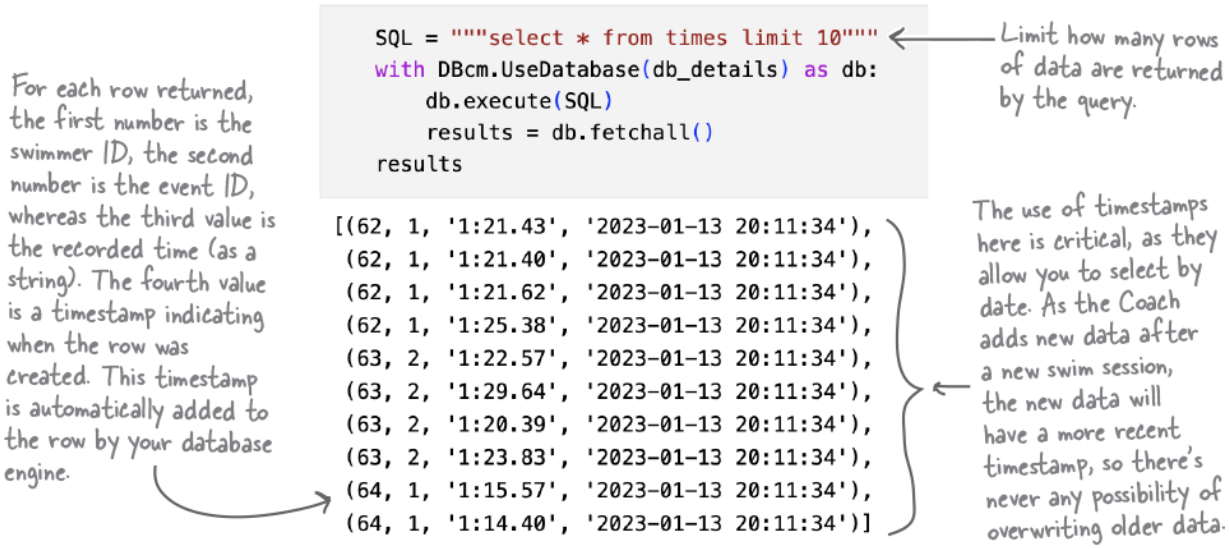
```
SQL = """select count(*) from times"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchone()[0]
results
```

The use of "count(*)" tells your database to report how many rows of data are returned. Note how "fetchone" is being used to good effect here, as you can be sure "count(*)" only ever returns a single row of data.

```
338
```

There are 338 individual times in the Coach's data.

You can read the first ten rows of data from your `times` table to get an idea of what has happened here:



THERE ARE NO DUMB QUESTIONS

Q: So, let me get this straight. To use DBcm I have to know a bit of SQL?

A: Yes, you do, and we'd suggest this is not such a bad thing. The *Head First Coders* are of the opinion that you should know when to use and exploit Python where it makes sense, as well as use and exploit SQL where it makes sense.

For instance, let's say you have a table with one hundred thousand rows of data. And let's further suggest you want to select a subset of the rows. Although you could grab all the data with `select *` then use Python to apply a filter, you'd be ill-advised to choose this path. Better to add a `where` clause to your query so that the database engine performs the filtering before the data gets to your Python code. Database engines exist to do this type of thing efficiently and without any fuss, so let them do it. Don't make work for yourself by writing Python code which performs a task your database engine provides.

Equally, say you need to transform some data returned by your database engine. Although you might be able to work out how to perform the transformation with some fancy SQL, it might be better to perform the transformation with Python code (especially if the transformation is in any way complex). Python is likely the better choice here.

The bottom line: when working with databases, take the time to learn a little bit of SQL.

Q: Should I be concerned that the data in my swimdata has created 338 rows of data in the times table? That seems like a lot…

A: In the big scheme of things, it's not really an awful lot of data. If you were to process all the files in the `swimdata` folder and count the number of individual times in each file while calculating a running total, you'd end up with 338 times spread across the sixty or so files.

And, anyway, nowadays diskspace is *cheap.* 😉



Almost…

There's just one final thing to do.

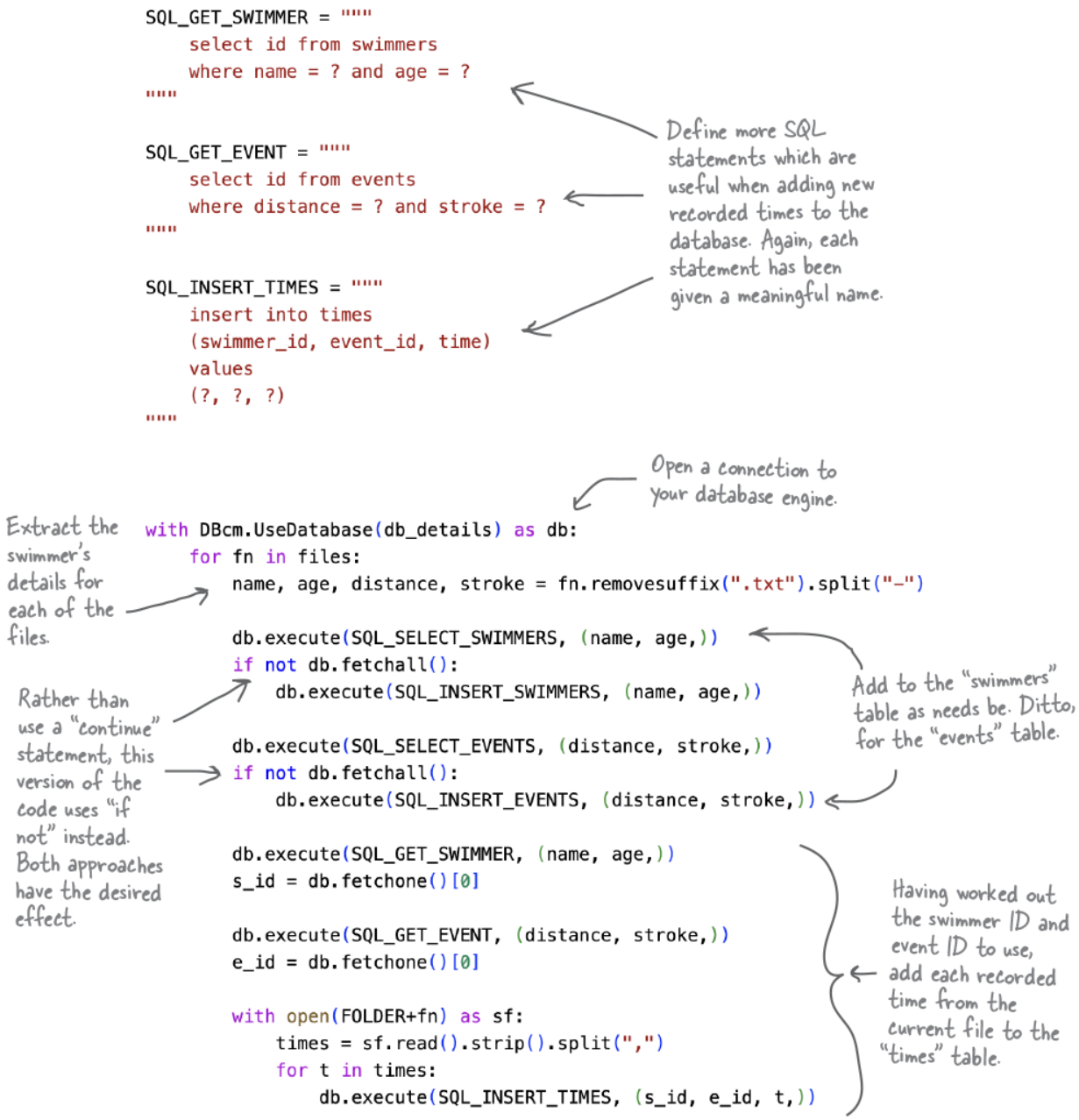
Right now, all the code that updates your three database tables is in your most-recent notebook. Let's not have a situation where the notebook has to be re-executed every time the Coach runs a training session and generates fresh data. Let's create a standalone utility which can be executed independently of your notebook.

A database update utilit y, 1 of 2

As this has been a large chapter, you are likely relieved that we've created the required utility for you. The utility is called `update_tables.py` and is included in this book's download materials. The vast majority of the code in the utility has been taken from your `PopulateTables.ipynb` notebook, and the utility's code is shown below and on the next page.



This is the rest of the `update_tables.py` code:



Task #2 is (finally) done

It's been a long time coming, but Task #2 is complete:

1 Decide on a structure for your data, then create your database tables

You need to decide how the data in your database is going to be arranged. Once you have decided on this, you can create the necessary database and tables using Python and SQL. ✔

2 Add your data values to your database tables

The Coach's system uses the data in the `swimdata` folder. You'll need to take the data from this folder and add it to the appropriate database tables. As in Task #1, Python and SQL are your go-to technologies here. ✔

We did warn you that the four tasks identified near the beginning of this chapter were a bit of work. What you've done so far as been great, but it's time to take a break. Task #3 and Task #4 are being saved for your next chapter:

3 Extract the data you need from your database tables

At the moment, your webapp's code dips in and out of the `swimdata` folder as needed to grab the data required to do its thing. With Task #2 complete, you can write Python and SQL to grab the data your webapp needs from your database tables instead.

4 Adjust your existing webapp code to use the data in your database tables

With Task #3 done, you'll need to change your current webapp's code to grab its data from your database as opposed to from the `swimdata` folder.

Take a moment to review this chapter's bullet points on the next page, then relax with your favorite brew while you tackle this chapter's crossword.

See you in the next chapter (after you've taken some time away from your keyboard).

CHAPTER REVIEW BULLET POINTS

- When it comes to managing the data associated with your applications, not much beats a **database**.
- When you install Python, you get **SQLite3** for free.
- SQLite3 is a single-process, file-based, database management **engine** which requires very little admin to run. All of your interactions with the engine can be executed from within Python code. This makes using SQLite3 an excellent choice when starting out, as you can experiment with ease.
- Although Python ships with the SQLite3 database engine and the `sqlite3` library, the **DBcm** PyPI module, once installed, allows your database interactions take advantage of the Python **with** statement.
- When used with **with**, DBcm lets you connect to an underlying database engine (either SQLite3 or MySQL/MariaDB), then execute an arbitrary number of SQL queries/statements. When the code block associated with your **with** statement ends, a "commit" is sent to the engine, and your database connection is automatically closed.
- If you use a filename with DBcm's **UseDatabase** class, SQLite3 is used. As you'll learn in a later chapter, a dictionary of credential information can also be provided to UseDatabase, allowing you to use DBcm with an existing MySQL/MariaDB installation.

- The Python DB-API provides a number of **methods** for interacting with your chosen database engine, including execute, fetchall, fetchmany, and fetchone.
- The **execute** methods sends an SQL query/statement to your chosen database engine.
- The **fetchall** method returns results as a list of tuples (when at least one row is returned) or an empty list (when no results are returned from your database engine).
- The **fetchmany** method operates like fetchall, but for the fact that you can specify a maximum number of rows to return. Repeated calls to **fetchmany** allows you to implement **pagination** (should you need it)..
- The **fetchone** method return one, and only one, tuple of results from your database engine. Like with **fetchall** and **fetchmany**, the results can be empty.
- Depending on the database engine you're using, you may be able to request additional information about the database and the tables your are working with. For example, SQLite3 provides the **pragma table_list** and **pragma table_info** commands, whereas MySQL/ MariaDB provide the equivalent **show tables** and **describe** commands.
- Despite SQL being a very established database technology, each database engine implements its own **dialect**. This can make life interesting…

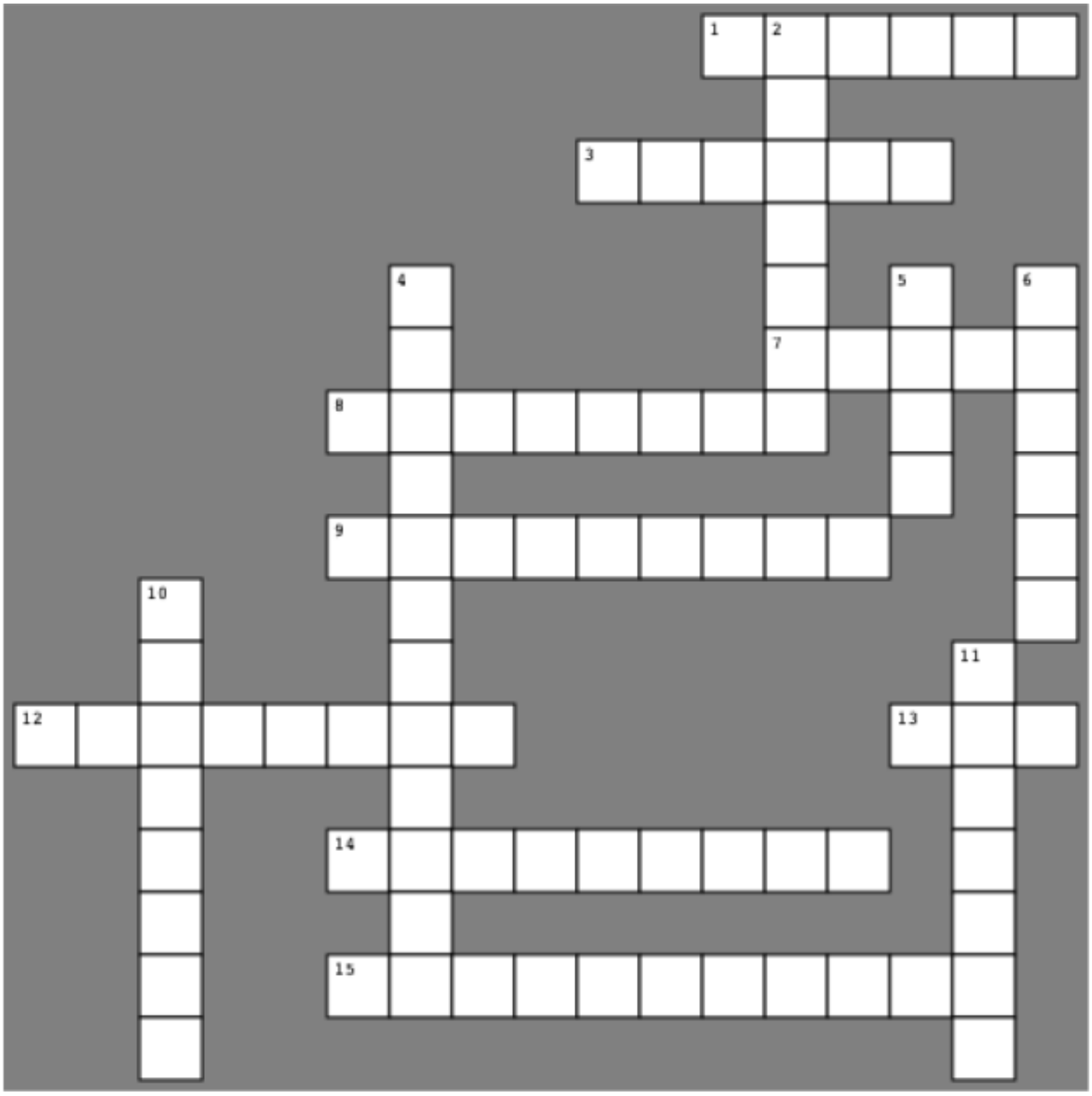
- When using SQL queries/statements in your Python code, the recommendation is to put them within **multi-line triple-quoted strings**. Of course, this is only a **recommendation**, so you don't have to do this. However, other coders reading your code may well thank you when you do (and, remember, that "other programmer" may be you in six months time).
- Python database engines support the notion of parameterized queries/statements. The use of **placeholders** can protect your code from all manner of SQL Injection Attacks, and the use of placeholders is strongly encouraged. Using f-strings to achieve the same effect (i.e., parameterized queries/statements) might lead to a world of pain… just don't try and say we didn't warm you.
- Most placeholder technologies expect data to be passed into SQL queries using a **tuple**, even when there's only one parameter. This can lead to weird looking tuples. Remember `(42,)`, which looks weird, but works.
- When executing multiple SQL queries/statements against your database (perhaps within a tight loop), it's better to place your loop within a database-connecting **with** statement than to constantly connect/execute/disconnect to your database within a tight loop. Your database engine (and, by extension, your database sys-admins) may well thank you when you do.

- When using SQL, the **select** statement lets your read data, the **insert** statement lets you add data, and the **delete** statement removes data from a table. There's also an **update** statement which can change data (but you've yet to need to perform any changes, so **update** has yet to make an appearance in this book).

The Database Crossword



You'll find all the answers to the clues in this chapter. The solutions are, as always, on the next page.



Across

1. Use this command to read managed data.

2. Use this command to add managed data.

7. A place for data: rows and columns.

8. Can have any number of 7 across.

9. The DB-API command which returns a specific list of rows of data.

12. The DB-API command which returns all rows of data (everything).

13. Sometimes pronounced “sequel”.

14. Use with a file object to grab all the data from an opened file (as a list of lines).

15. Used with 5 down to establish a connection.

Down

2. Use this command to run a query.

4. Using these helps guard against nasty Injection Attacks (you sleep better, too).

5. A custom module built on top of a PSL module, which allows you to run queries using a **with** statement (see 11 down).

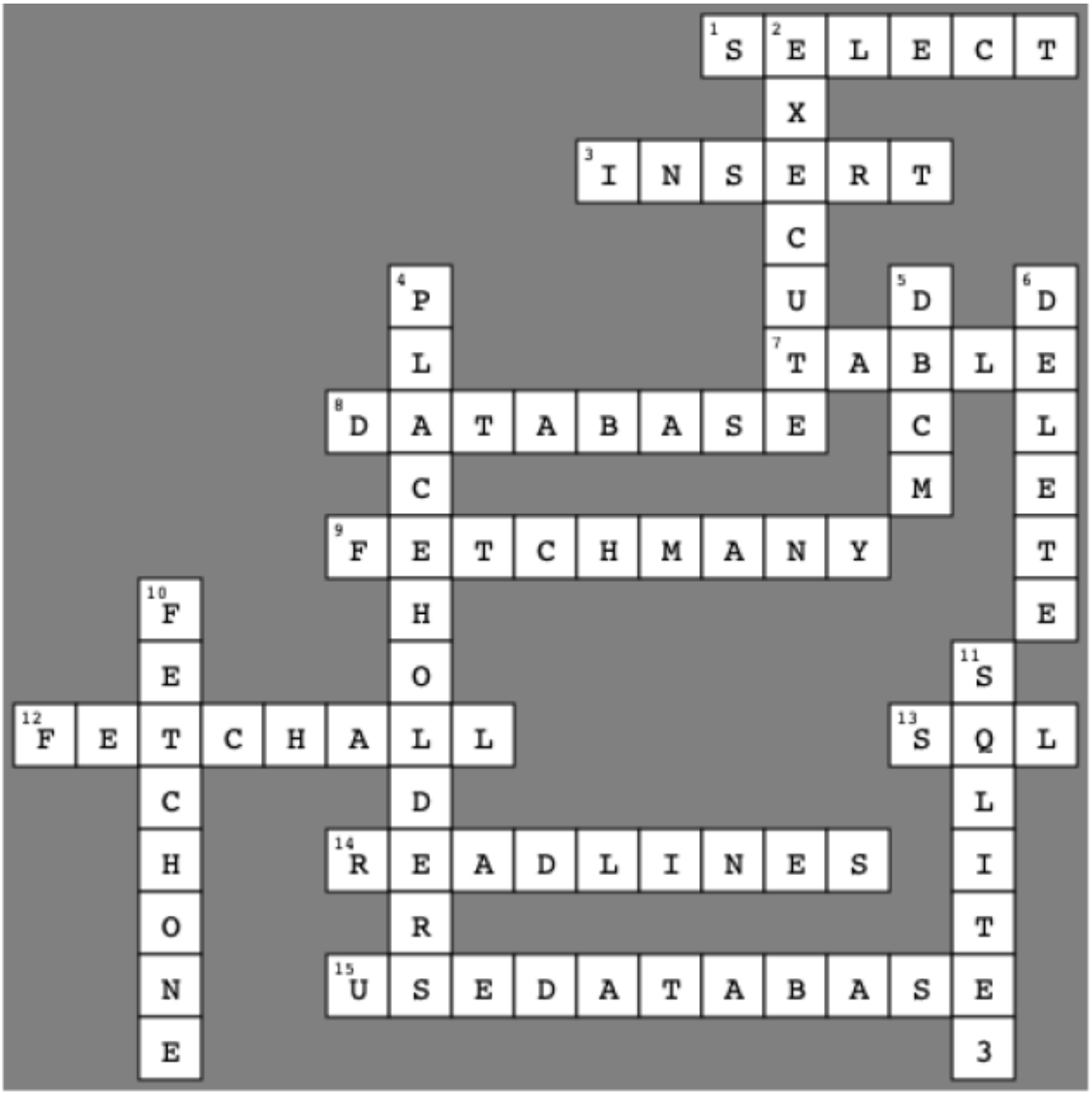
6. Use this command to remove managed data.

10. This DB-API command returns a single row of data.

11. This engine is pre-installed when you install Python, but what's its name?

The Database Crossword Solution





Across

1. Use this command to read managed data.

2. Use this command to add managed data.

7. A place for data: rows and columns.

8. Can have any number of 7 across.

9. The DB-API command which returns a specific list of rows of data.

12. The DB-API command which returns all rows of data (everything).

13. Sometimes pronounced “sequel”.

14. Use with a file object to grab all the data from an opened file (as a list of lines).

15. Used with 5 down to establish a connection.

Down

2. Use this command to run a query.

4. Using these helps guard against nasty Injection Attacks (you sleep better, too).

5. A custom module built on top of a PSL module, which allows you to run queries using a **with** statement (see 11 down).

6. Use this command to remove managed data.

10. This DB-API command returns a single row of data.

11. This engine is pre-installed when you install Python, but what's its name?

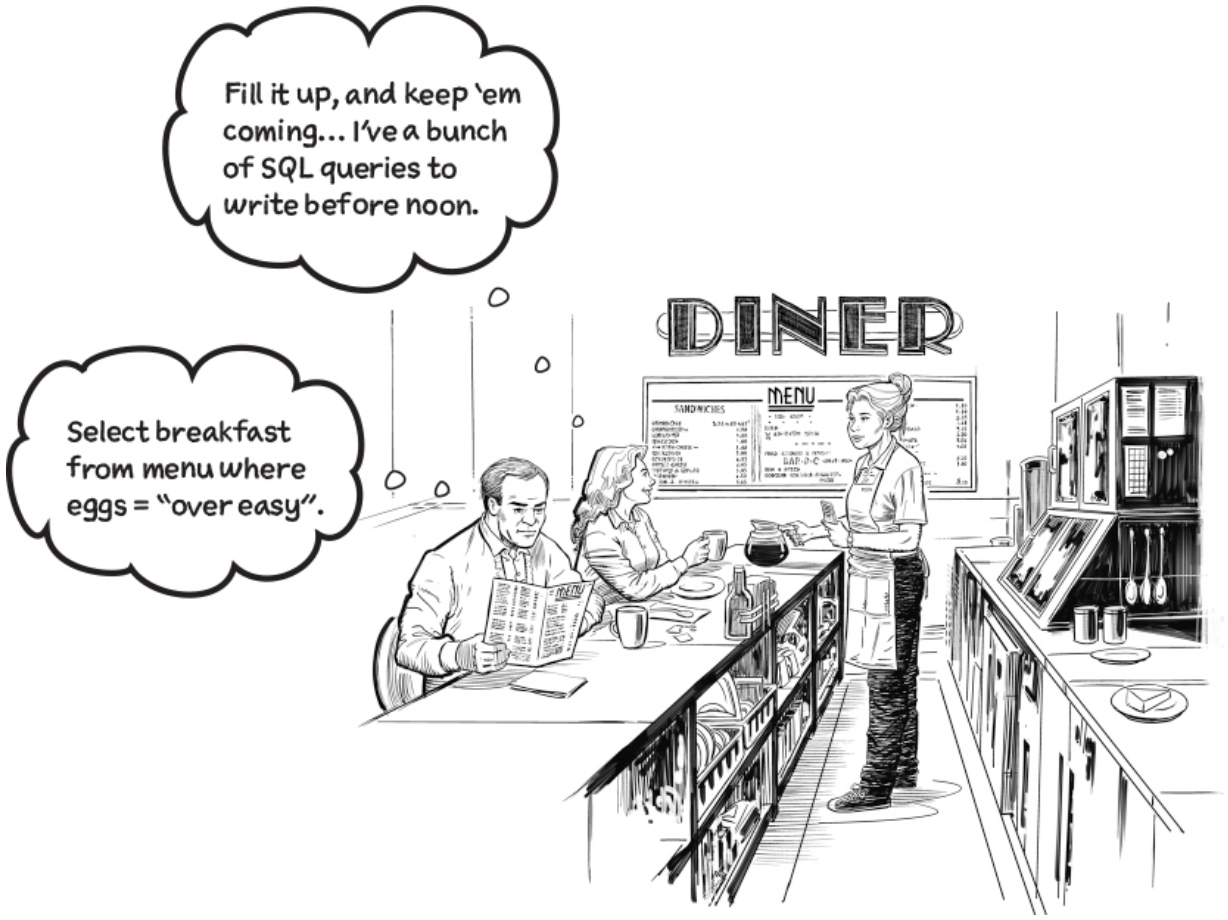
Chapter 12. List Comprehensions: *Database Integrations*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

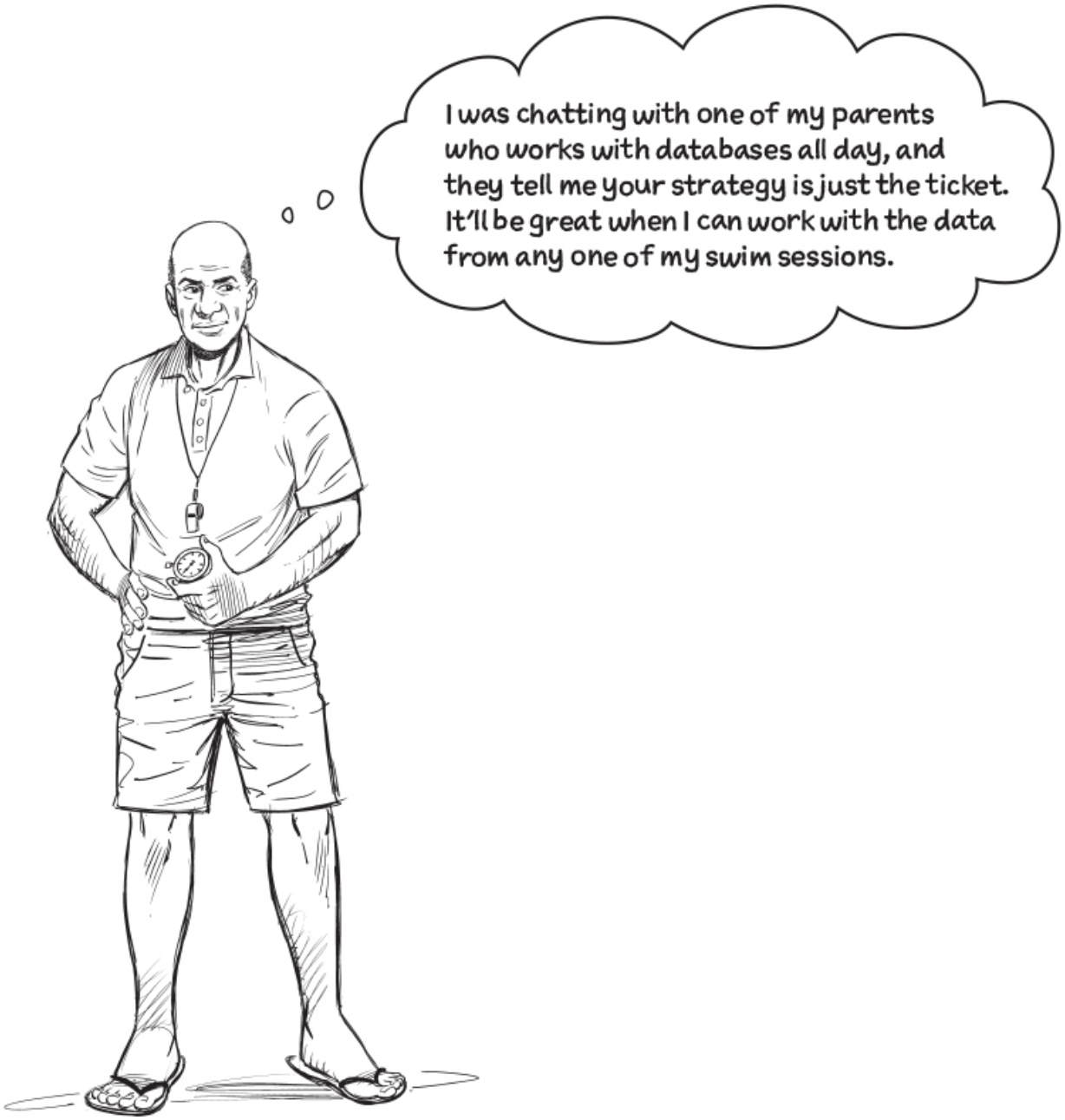
This will be Chapter 11 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



With your database tables ready, it's time to integrate.

Your webapp can gain the **flexibility** the Coach requires by using the datasets in your database tables, and in this chapter you create a module of **utilities** which lets your webapp **exploit** your database engine. And, in a never ending quest to do more with less code, you'll learn how to read and write **list comprehensions**, which are a genuine Python superpower. You'll also **reuse** a lot of your pre-existing code in new and interesting ways, so let's get going. There's lots of **integration** work to do.



We're looking forward to that, too.

To help us get there, the *Head First Coders*, concerned that your SQL skills might be a bit rusty, have sent over a file of SQL queries for you to consider using with your webapp.

The *Coders* suggested you'll need to do a bit of extra work to use these queries to tackle Task #3 and #4, but they think the queries will help extract the data your webapp needs.

Recall the next two tasks, then take a look at the queries (copied from `queries.py`) on the next page.

3 Extract the data you need from your database tables

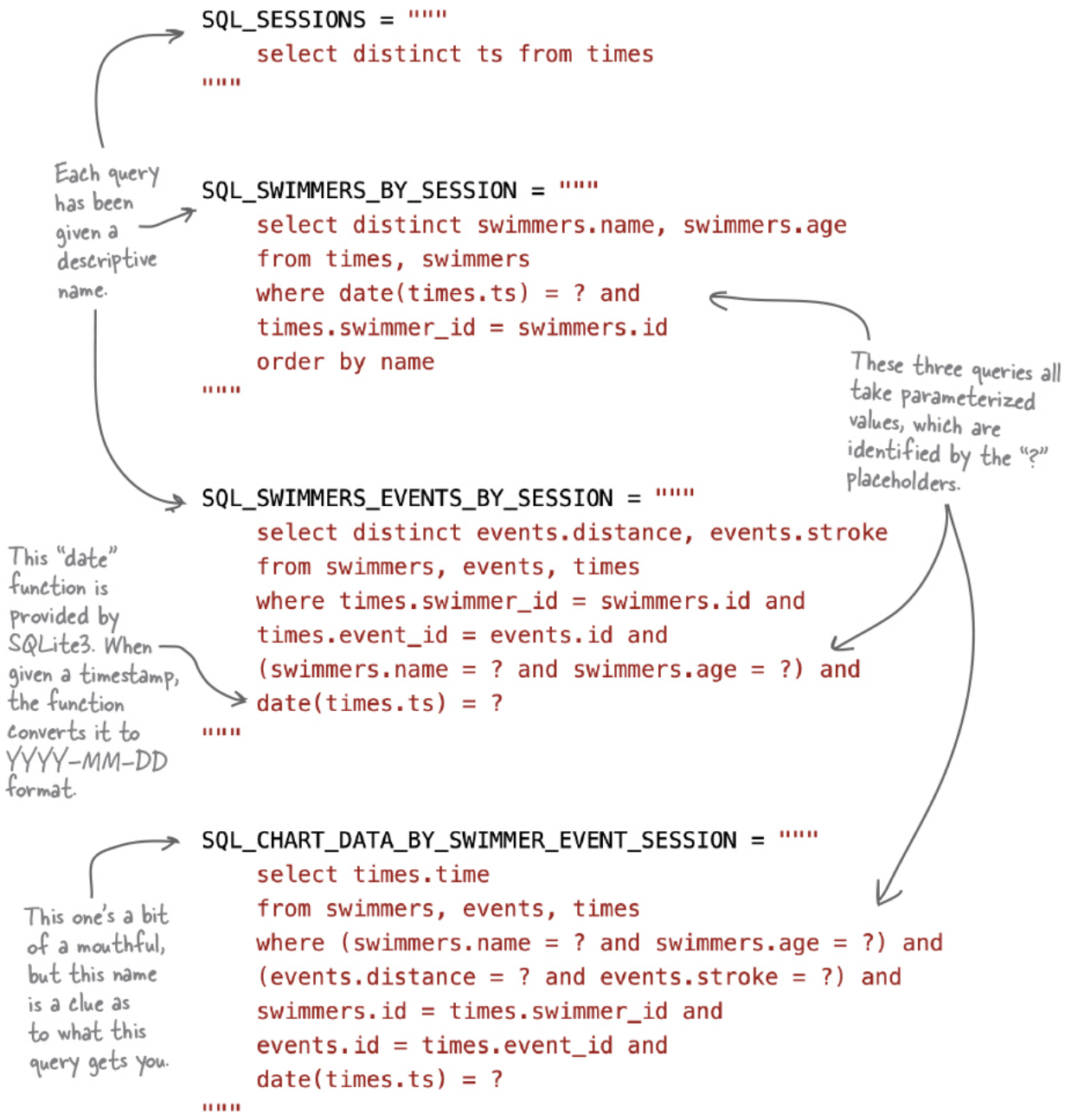
At the moment, your webapp's code dips in and out of the `swimdata` folder as needed to grab the data required to do its thing. With Task #2 complete, you can write Python and SQL to grab the data your webapp needs from your database tables instead.

4 Adjust your existing webapp code to use the data in your database tables

With Task #3 done, you'll need to change your current webapp's code to grab its data from your database as opposed to from the `swimdata` folder.

Four queries to grab the data you need

As this book doesn't expect you to be an SQL expert (although it is OK if you are), here's the content of the `queries.py` file as sent over by the *Coders*. Pay particular attention to the names given to the queries, which indicates the data you can expect back from the database when any one of the queries is used:



Let’s explore the queries in a new notebook

In order to make sure everyone knows what to expect from these queries, let’s create yet another notebook within which to experiment. Go ahead and use VS Code do this now, saving your newly created notebook as `TryQueries.py` in your `webapp` folder. To start things off, import the `queries` module, then pull out the *combo mambo*:



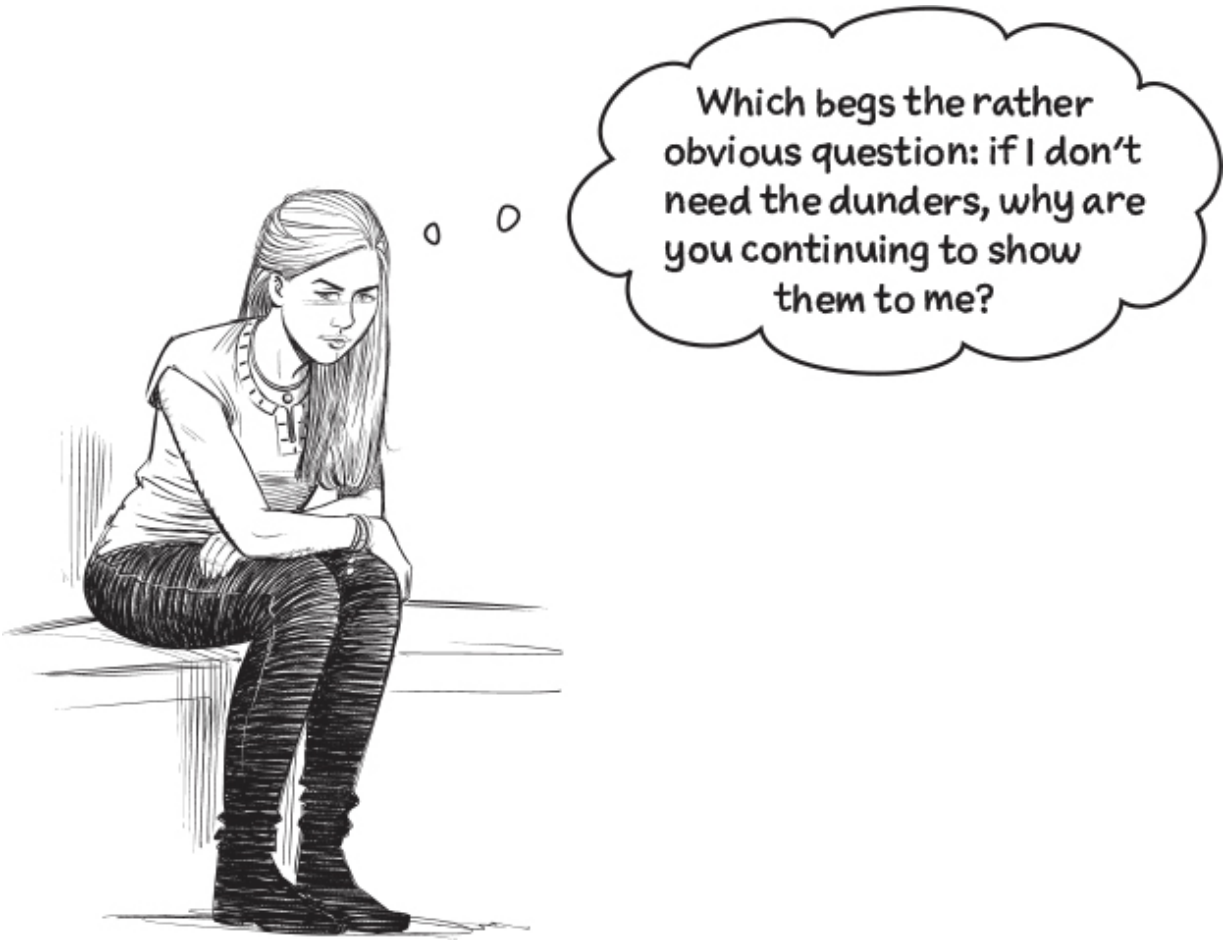
```
import queries

print(dir(queries))
```

```
['SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION', 'SQL_SESSIONS', 'SQL_SWIMMERS_BY_SESSION',
'SQL_SWIMMERS_EVENTS_BY_SESSION', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__']
```

In addition to the usual collection of dunders, each of the query names are listed, too.

As always, the suggestion is to *ignore* the dunders reported by the *combo mambo*. It's not that we've anything against the dunders, it's just that we've yet to need them in this book… and if there's one thing a Head First book tries hard to do, it's to only explain something when it is really needed, and you *still* don't need the dunders.



Good question. Why indeed?

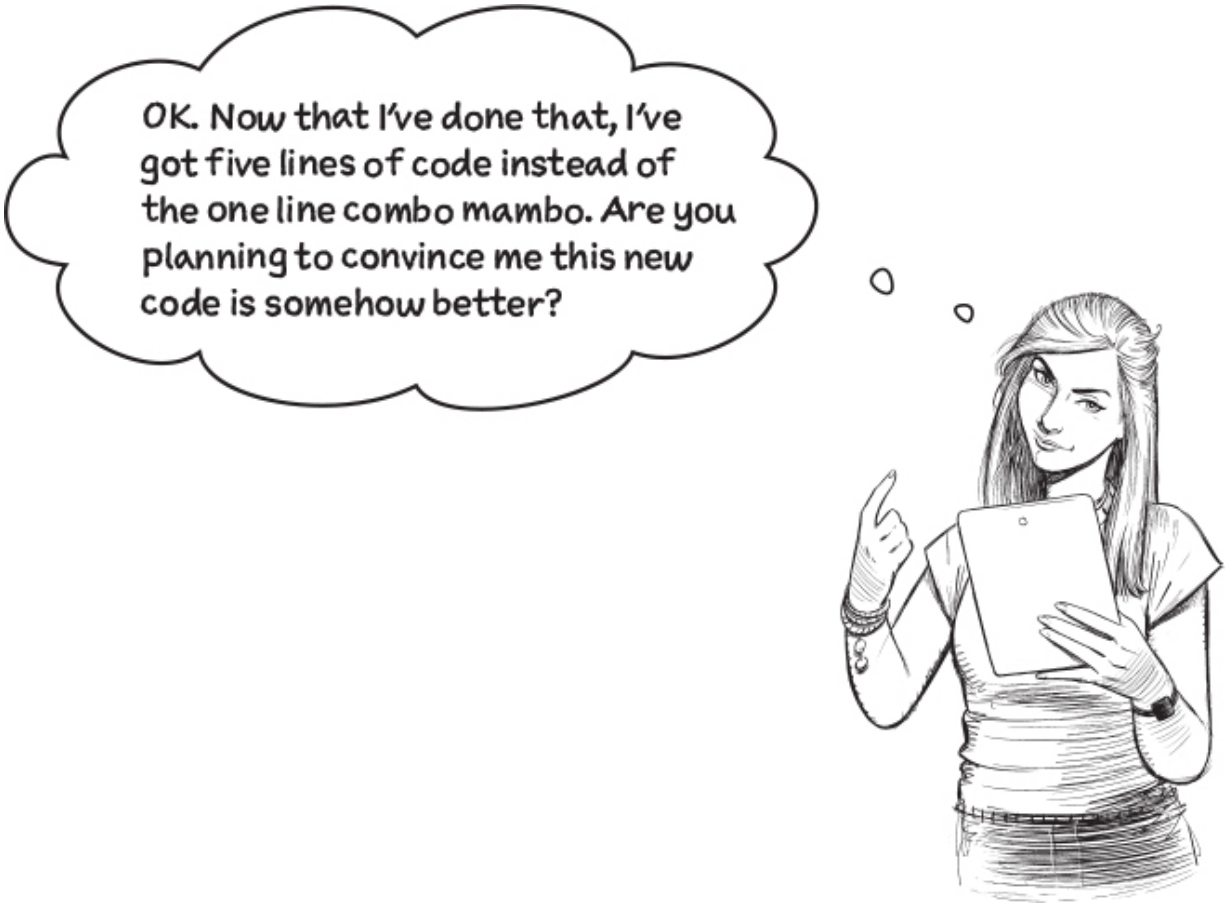
Let's create a little bit of code which only shows the non-dunders when you use the **print dir** *combo mambo*.

EXERCISE



Let's experiment in a notebook cell. Create a empty list called `statements`, then write the code for a **for** loop which iterates over each of the items in the list produced by `dir(queries)`. Append the current item to the `statements` list, but only if it starts with something other than two underscore characters, i.e., "`__`". When your loop terminates, display the `statements` list on screen.

Write the code you created to do this task in the space below.



Bear with. There's method to our madness.

Yes, you've likely created a number of lines of code doing this exercise, just like those shown over the page. But, as you are about to

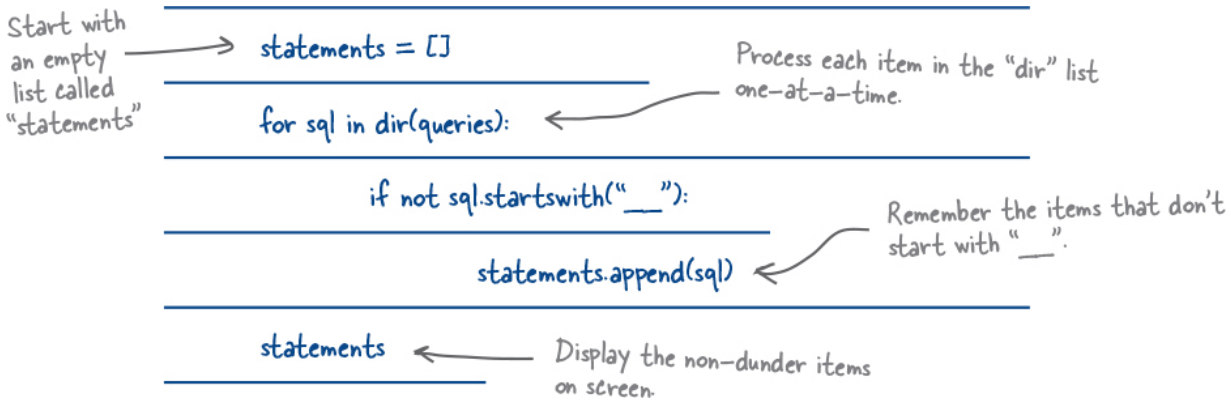
see, we’re not done yet.

EXERCISE SOLUTION



You were to experiment in a notebook cell. Creating a empty list called `statements`, you were then to write the code for a **for** loop which iterates over each of the items in the list produced by `dir(queries)`, appending the current item to the `statements` list as you go. However, you were only to do this if the item started with something other than two underscore characters, i.e., "`__`". When your loop terminated, you were to display the `statements` list on screen.

Here's the code we came up with:



TEST DRIVE



Your code works, albeit using quite a bit more code than your much-loved **print dir** *combo mambo*:

```
    statements = []
    for sql in dir(queries):
        if not sql.startswith("__"):
            statements.append(sql)
    statements
```

```
['SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION',
 'SQL_SESSIONS',
 'SQL_SWIMMERS_BY_SESSION',
 'SQL_SWIMMERS_EVENTS_BY_SESSION']
```

Look! No dunders here.

Five lines of loop code become one



Now, before everyone throws up their hands in despair, it is possible to reduce those five lines of code to **one**. Take a look:

```
    [sql for sql in dir(queries) if not sql.startswith("__")]
```

```
['SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION',
 'SQL_SESSIONS',
 'SQL_SWIMMERS_BY_SESSION',
 'SQL_SWIMMERS_EVENTS_BY_SESSION']
```

This output is identical to the output from the five lines of code, but has been produced here with a single line of code. And, BTW, there are no dunders here, either.



It's not magic. It's a list comprehension.

In Python, a *comprehension* is a way to generate a collection of values using a compact form of the five lines of loop code from the previ-

ous page. That code generated a new list, called `statements`, containing the entries produced by the **dir** BIF which aren't dunders.

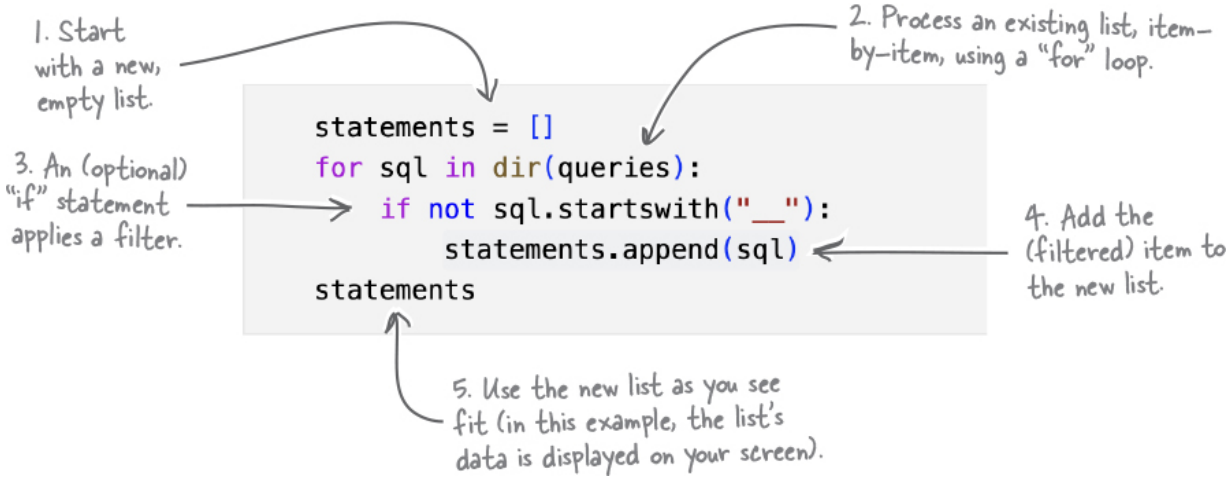
The **list comprehension** at the top of this page does *exactly* the same thing: it generates a list of entries from the **dir** BIF which aren't dunders.

Unlike the five lines of loop code, the list comprehension doesn't create the `statements` list, although there's nothing stopping you assigning the generated list to a variable if you want to. Both the five lines of code and the single-line list comprehension produce the same output. The former has to have its generated list assigned to a variable so you can view the results, whereas the list comprehension doesn't.

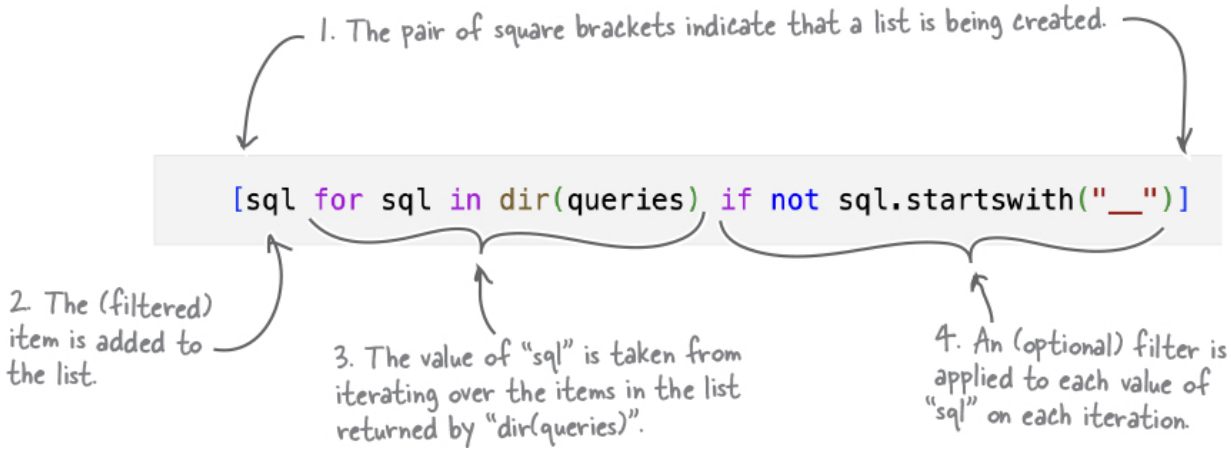
Getting from fives lines of code to one…



Take a look at the five lines of **for** loop code and notice that they follow a very specific structure (or pattern):



On the face of things, the one-line list comprehension doesn't appear to have much in common with the five-line **for** loop, but take a closer look:



Reading from left-to-right, the list comprehension can be described as follows:

> *"Remember the value of* `sql` *for each of the values of* `sql` *in the* `dir(queries)` *list, but only remember the value if the filtering condition holds".*

As shown earlier, the output produced by the five lines of code and the single line of code is *identical*.

A non-dunder combo mambo



A slight change to the list comprehension lets you use it with *any* Python object:

The original, all-inclusive "print dir" combo mambo.

```
print(dir(list))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

```
[x for x in dir(list) if not x.startswith("__")]
```

In this list comprehension we're using "x" as our variable name, but you can use any name.

```
['append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

The list of "non-dunders", thanks to the list comprehension..

```
print([x for x in dir(list) if not x.startswith("__")])
```

```
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

The list comprehension in this cell is passed as a parameter to the "print" BIF. The filtered list is generated by the comprehension, then the resulting list is displayed across the screen by "print".

That most-recent code cell also demonstrates a major advantage a list comprehension has over its equivalent **for** loop (other than the five-line to one-line pay-off, that is). Comprehensions can be used when it's *syntactically impossible* to use an equivalent five-line **for** loop. You cannot code a "regular" **for** loop as a parameter to a func-

tion (like **print**). However, you can – as you've just seen – use a list comprehension as an argument value.

NOTE

This can be surprisingly useful.



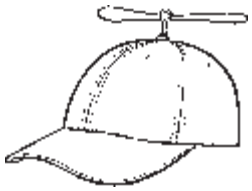
List comprehensions are used everywhere.

Python programmers *love* list comprehensions, and they pop up all over the place in Python code.

As you're about to see, they can be very handy when a quick transformation is required of the data returned from your database engine (among many other uses), so think of the previous three pages as your heads-up.

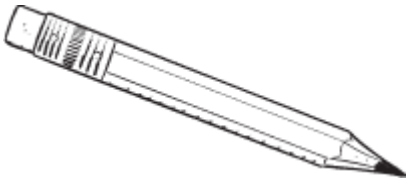
You'll use more list comprehensions later in this chapter.

GEEK NOTE



In addition to supporting the creation of list comprehensions, Python also allows for the creation of **dictionary** and **set comprehensions**. Tuple comprehensions make no sense, as tuples are immutable so cannot be "generated" on-thefly. If you try to create a "tuple comprehension", Python won't complain. Instead, Python creates an *iterable generator* which may or may not be what you were expecting. (See the Python docs for a bit more on **generators**).

SHARPEN YOUR PENCIL

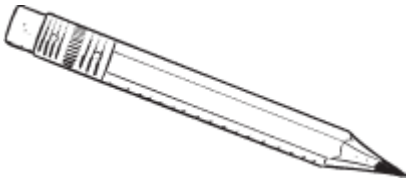


Return to your `TryQueries.ipynb` notebook, then type in the four lines of code shown below.



In the space provided, type in the code you'd use to send the `SQL_SESSIONS` query to your database engine, then display any returned results:

SHARPEN YOUR PENCIL SOLUTION



You were to return to your `TryQueries.ipynb` notebook, then type in the four lines of code shown below.



In the space provided, you were to type in the code you'd use to send the `SQL_SESSIONS` query to your database engine, then display any returned results. Here's the code we used. How's does yours compare?

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SESSIONS)
    results = db.fetchall()
results
```

Continuing to use the "with" statement pattern from the previous chapter, the query is sent to the database, all the rows returned are fetched, then displayed on screen.

TEST DRIVE



Running the code from the previous *Sharpen* confirms the existence of one set of stored data in your database:

```
    with DBcm.UseDatabase(db_details) as db:
        db.execute(SQL_SESSIONS)
        results = db.fetchall()
    results
```

```
[('2023-01-13 20:11:34',)]
```

The output you see will likely display a different date to us, as the date is derived from when you inserted your data.

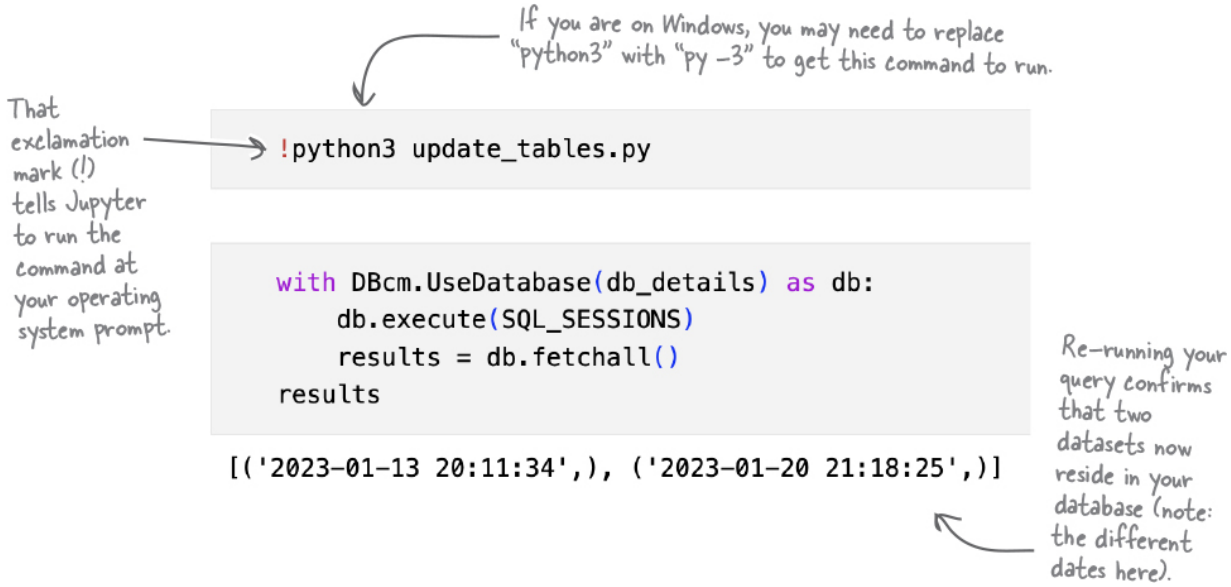
As luck would have it, the Coach has just sent over the data from his most-recent swim session, explaining that some of his usual attendees were off at a competition, so his latest dataset isn't as large as last time. He also tells you a new swimmer joined the club: a seven year old called… wait for it… Darius.

The most recent data is in a file called `swimdata2.zip`. Remove the files in your current `swimdata` folder (as you've already added that data to your database tables), then unzip the contents of

`swimdata2.zip` into your empty `swimdata` folder, resulting in a collection of new files in `swimdata`.

Recall how, at the end of the previous chapter, you created a database utility which, when executed, takes the datafiles from your `swimdata` folder and uses them to populate your database tables. Let's run your utility now to update your database with the latest swim session times.

Staying within your `TryQueries.py` notebook, execute an operating system command to run your utility, then rerun the cell of code shown above to confirm that your database now contains two datasets:



One query down, three to go…

The three other queries can be tried out in much the same way as you just did with the first one. The main difference being, of course, that each query expects parameterized values. Let's work through the three remaining queries now, to see how they work.

Be sure to **follow along** with this material in your `TryQueries.ipynb` notebook.

Grab (and display) the next query from the "queries.py" module.

```
from queries import SQL_SWIMMERS_BY_SESSION

print(SQL_SWIMMERS_BY_SESSION)
```

```
select distinct swimmers.name, swimmers.age
from times, swimmers
where date(times.ts) = ? and
times.swimmer_id = swimmers.id
order by name
```

This query expects to receive a date value, as it filters the returned swimmer's names and ages on the date (which is to be provided in YYYY-MM-DD format).

The usual pattern runs your latest query.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SWIMMERS_BY_SESSION, ("2023-01-13",))
    results = db.fetchall()
results
```

There are two things to note about how the date value is provided. Firstly, the time part of the timestamp is not required here and, secondly, the single parameter to the SQL query is provided as a single-element tuple. Note the trailing comma at the end of the tuple which tells Python this single item is meant to be in a tuple. (And, yes, it looks weird to us too, but that's how single-element tuples roll...).

When you run the code in the cell shown at the bottom of the previous page (being sure to update the date value to match the data in your database), here's the list of swimmer's from that first swim session.

```
[('Abi', 10),
 ('Ali', 12),
 ('Alison', 14),
 ('Aurora', 13),
 ('Bill', 18),
 ('Blake', 15),
 ('Calvin', 9),
 ('Carl', 15),
 ('Chris', 17),
 ('Darius', 13),
 ('Dave', 17),
 ('Elba', 14),
 ('Emma', 13),
 ('Erika', 15),
 ('Hannah', 13),
 ('Katie', 9),
 ('Lizzie', 14),
 ('Maria', 9),
 ('Mike', 15),
 ('Owen', 15),
 ('Ruth', 13),
 ('Tasmin', 15)]
```

Remember: the "fetchall" method returns either an empty list (no results) or a list of tuples (as shown here).

Adjusting the parameterized value to refer to the second swim session results in a much smaller list of results (as expected).

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SWIMMERS_BY_SESSION, ("2023-01-20",))
    results = db.fetchall()
results
```

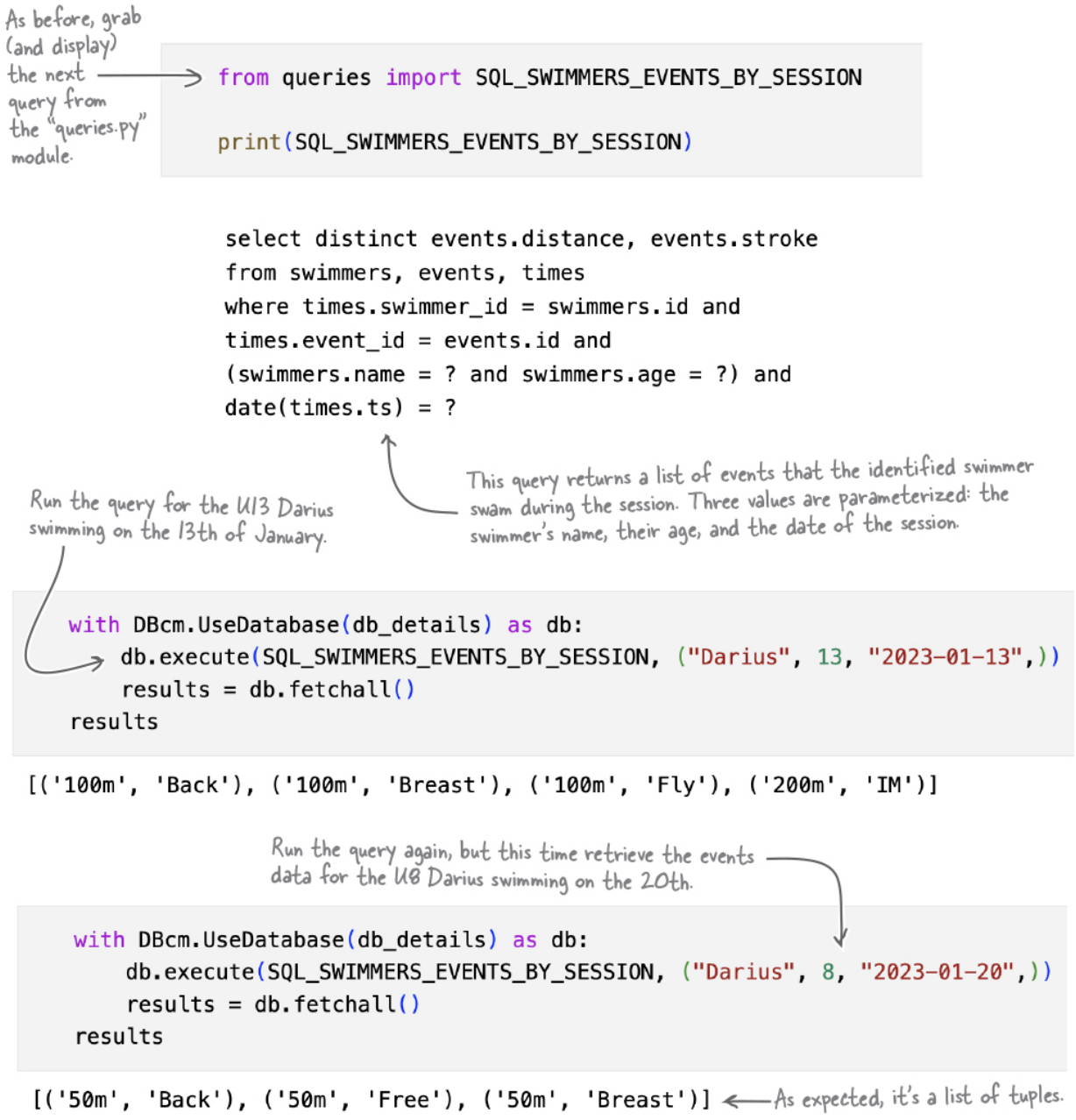
A smaller list of tuples this time (due to most swimmers being off at a competition).

```
[('Abi', 10),
 ('Blake', 15),
 ('Darius', 13),
 ('Darius', 8),
 ('Dave', 17),
 ('Katie', 9),
 ('Maria', 9),
 ('Owen', 15)]
```

There are now two different swimmers called "Darius". And don't confuse these numbers with the swimmer's actual age. They refer to the age group each swimmer swims in. So, the younger Darius is 7 years old, but swims in the Under 8 age group.

Two queries down, t wo to go…

Let's keep going with the remaining queries. Continue to **follow along** within your notebook.



The last, but not least (query)…

The fourth query retrieves the timing data for a swimmer of a certain age, swimming during a certain session, while swimming a specific event:

Grab and display the query.

```
from queries import SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION

print(SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION)
```

```
select times.time
from swimmers, events, times
where (swimmers.name = ? and swimmers.age = ?) and
(events.distance = ? and events.stroke = ?) and
swimmers.id = times.swimmer_id and
events.id = times.event_id and
date(times.ts) = ?
```

Five (!!!) parameterized values need to be provided to this query: the details of the swimmer (two values), the details of the event (another two values), and the date of the swim session.

It's a bit of a squeeze, but five values are provided for each of the parameterized placeholders in this query. As always, the values are passed into the query as a tuple.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION, ("Darius", 13, "100m", "Fly", "2023-01-13",))
    results = db.fetchall()
results
```

```
[('1:27.95',),
 ('1:21.07',),
 ('1:30.96',),
 ('1:23.22',),
 ('1:27.95',),
 ('1:28.30',)]
```

This list (of tuples) of swim times should look familiar to you. They are the exact same times as presented to you in the Coach's spreadsheet at the very start of Chapter #1.



Yes, maybe we could do that.

However, it might also be a good idea to create a module of data-access functions which abstract away the inner workings of the `TryQueries.ipynb` notebook.

By creating a module of database utility functions you can centralized all of your database code in one place. This saves you from littering your existing webapp code with a bunch of `DBcm` **with** statements. Further, by having your database utility functions in one place, you make it easier to swap out one database engine for another (as you'll see, later, when you make the jump from SQLite3 to MariaDB).

Let's create a module of database utilities.

THERE ARE NO DUMB QUESTIONS

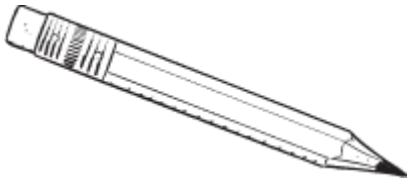
Q: Do I have to learn how to read and write list comprehensions?

A: No, as a **for** loop can generally accomplish the same thing. However, list comprehensions are *very* popular, so you are likely to come across them a lot when working with another programmer's code, so we think learning about them is very much worth it.

Q: I already have MariaDB installed. Can I go ahead and use it instead of SQLite3?

A: There are a small number of incompatibilities with how Python interacts with MariaDB, so *hold that thought* until we get to discuss them in the next chapter. For now, stick with SQLite3.

SHARPEN YOUR PENCIL



Time for you to flex your function-creating muscles.

Create a new file (**not** a notebook) in your `webapp` folder called `data_utils.py` and add the following lines of code to your new file:

```
import DBcm

db_details = "CoachDB.sqlite3"

from queries import *
```

This is yet another variant of the "import" statement. The "*" is shorthand for "everything" and, when used in this context, imports all the names defined in the named module to the current namespace. Using "*" allows you to refer to objects in the imported module without having to prefix the object's name with the modules name and a dot. This means, for instance, you can use "SQL_SESSIONS" to refer to the first query as opposed to having to use "queries.SQL_SESSIONS".

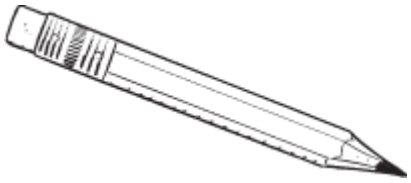
Here is the code you used in your `TryQueries.ipynb` notebook to run the third query. Your task is to turn this code into a parameterized function which can be called with three parameter values which are

slotted into the placeholders as needed. The function needs to return the results from the query to your calling code.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SWIMMERS_EVENTS_BY_SESSION, ('
    results = db.fetchall()
results
```

Write the code for your function here:

SHARPEN YOUR PENCIL SOLUTION



It was time for you to flex your function-creating muscles. You were to create a new file (**not** a notebook) in your `webapp` folder called `data_utils.py` and add the following lines of code to your new file:

```
import DBcm

db_details = "CoachDB.sqlite3"

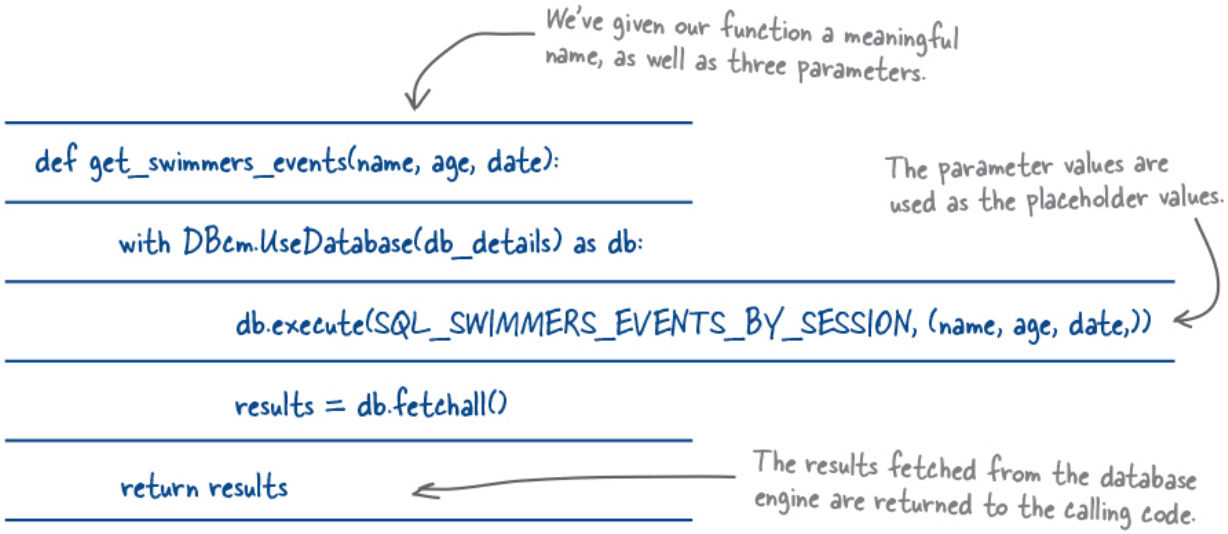
from queries import *
```

This is yet another variant of the "import" statement. The "*" is shorthand for "everything" and, when used in this context, imports all the names defined in the named module to the current namespace. Using "*" allows you to refer to objects in the imported module without having to prefix the object's name with the modules name and a dot. This means, for instance, you can use "SQL_SESSIONS" to refer to the first query as opposed to having to use "queries.SQL_SESSIONS".

Here is the code you used in your `TryQueries.ipynb` notebook to run the third query. Your task was to turn this code into a parameterized function which can be called with three parameter values which are slotted into the placeholders as needed. The function needs to return the results from the query to your calling code.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SWIMMERS_EVENTS_BY_SESSION, ('
    results = db.fetchall()
results
```

Here's the code we came up with:





Yes, the goal is for it to be easy.

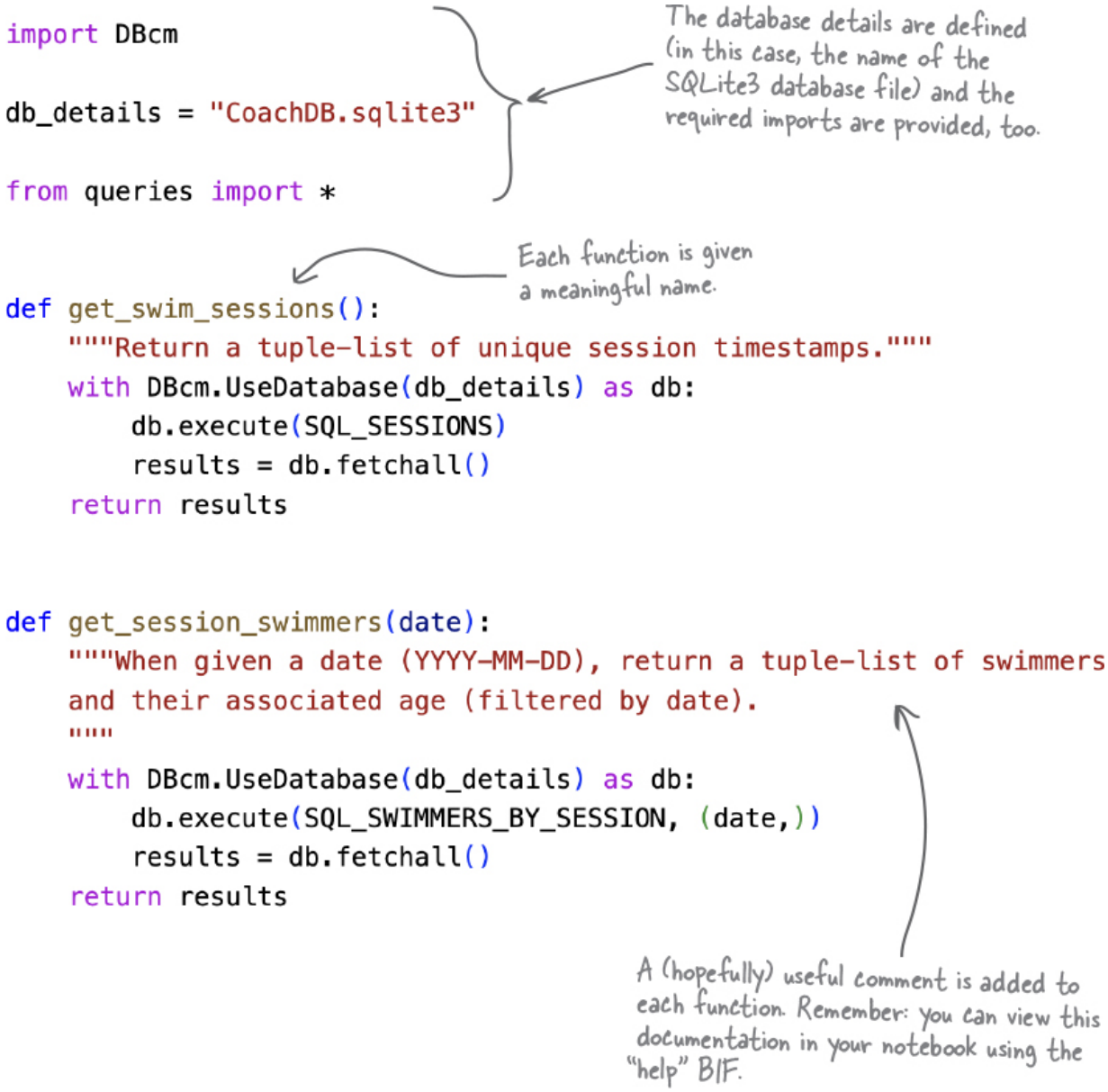
It *should* be a straightforward task to take your code from a Jupyter notebook and turn it into a callable function, as shown.

There's no skullduggery here. This really is a straightforward task.

Rather than launch straight into a *Test Drive* to showcase your new function in action, instead we're going to save you some time and present you with the code to our `data_utils.py` module (over the page). As you'll see, we've repeated the work from your most-recent *Sharpen* to create four callable functions, one for each of the SQL queries you've been working with.

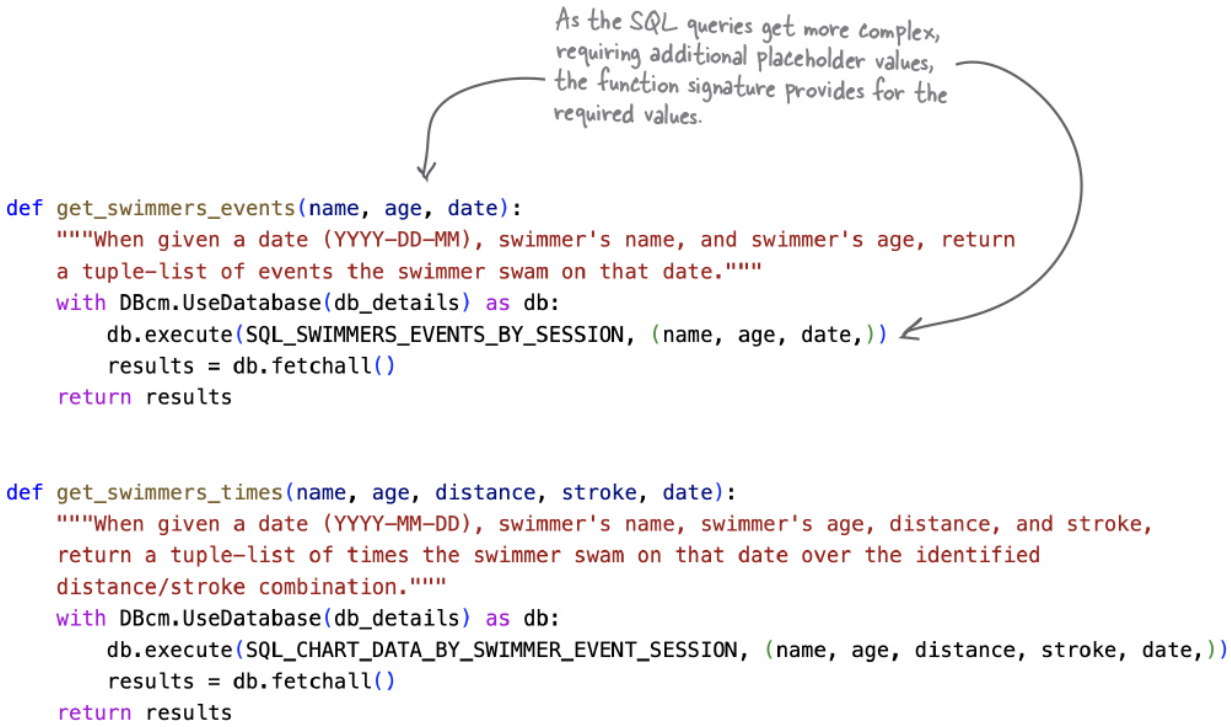
The database utilities code, 1 of 2

Here's the first half of the `data_utils.py` code we created to turn the four database-accessing **with** statements into callable functions. We've added comments to each of the functions (to help keep everyone informed).



The database utilities code, 2 of 2

Here's the second half of the `data_utils.py` code we created to turn the four database-accessing **with** statements into callable functions.




```
def get_swimmers_events(name, age, date):
    """When given a date (YYYY-DD-MM), swimmer's name, and swimmer's age, return
    a tuple-list of events the swimmer swam on that date."""
    with DBcm.UseDatabase(db_details) as db:
        db.execute(SQL_SWIMMERS_EVENTS_BY_SESSION, (name, age, date,))
        results = db.fetchall()
    return results


def get_swimmers_times(name, age, distance, stroke, date):
    """When given a date (YYYY-MM-DD), swimmer's name, swimmer's age, distance, and stroke,
    return a tuple-list of times the swimmer swam on that date over the identified
    distance/stroke combination."""
    with DBcm.UseDatabase(db_details) as db:
        db.execute(SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION, (name, age, distance, stroke, date,))
        results = db.fetchall()
    return results
```

Using a data module supports future refactoring activities

An advantage of placing all of your data-access functions within their own module is that it allows you to future-proof your webapp against changes to the underlying data access mechanisms.

If, down the road, you decide to change how your data is stored – perhaps using a MariaDB database engine as opposed to SQLite3 – you can limit your coding changes to your data module, being sure to keep the existing function signatures unchanged. As far as your application is concerned, your data utility functions are called and the

data is returned. Your application shouldn’t care what underlying database engine you’re using.

TEST DRIVE



Let's take your new `data_utils.py` module for a spin to confirm it performs the way you want it to.

Create another Jupyter notebook with VS Code called `TestDataUtils.ipynb`. Let's begin by importing the module then using the list comprehension version of the combo mambo to display the module's non-dunders:

```
import data_utils
```

```
print([x for x in dir(data_utils) if not x.startswith("__")])
```

```
['DBcm', 'SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION', 'SQL_SESSIONS', 'SQL_SWIMMERS_BY_SESSION',
'SQL_SWIMMERS_EVENTS_BY_SESSION', 'db_details', 'get_session_swimmers', 'get_swim_sessions',
'get_swimmers_events', 'get_swimmers_times']
```

This might be a bigger list of names than you were expecting. Note that the "data_utils.py" module imports your "queries.py" module, so the non-dunder names from *both* modules are reported here.

With your module imported, you can use the **help** BIF to learn a little about any of your functions:

```
help(data_utils.get_swim_sessions)
```

```
Help on function get_swim_sessions in module data_utils:

get_swim_sessions()
    Return a tuple-list of unique session timestamps.
```

Read the docs from the bottom-up.

Let's run the `get_swim_sessions` function:

```
data_utils.get_swim_sessions()
```

```
[('2023-01-13 20:11:34',), ('2023-01-20 21:18:25',)]
```

The tuple-list of results confirms your database contains two datasets, added on two separate dates (which are represented as two distinct timestamps values).

The next function returns the tuple-list of swimmers who swam during a particular session, which (in this case) is the second swim session from your database:

```
data_utils.get_session_swimmers("2023-01-20")
```

```
[('Abi', 10),
 ('Blake', 15),
 ('Darius', 13),
 ('Darius', 8),
 ('Dave', 17),
 ('Katie', 9),
 ('Maria', 9),
 ('Owen', 15)]
```

A much smaller tuple-list of swimmers (as expected).

Let's take a look at the events the seven year old Darius swam during that session:

```
data_utils.get_swimmers_events("Darius", 8, "2023-01-20")
```

```
[('50m', 'Back'), ('50m', 'Free'), ('50m', 'Breast')]
```

The tuple-list of the events the younger Darius swam during that session.

And, finally, let's see the swim times seven year old Darius recorded during his 50m Free training dips:

```
data_utils.get_swimmers_times("Darius", 8, "50m", "Free", "2023-01-20")
```

```
[('39.42',), ('36.13',), ('37.66',), ('39.07',)]
```

The tuple-list of times recorded for Darius which, for a seven year old, are very promising indeed. We think the younger Darius might be a swimmer to watch out for in the future!

With the conclusion of this *Test Drive*, you now have a module of database functions you can integrate into your existing webapp code. Which means, of course, it's time for another tickmark…

It's nearly time for the database integration

Let's remind ourselves of what's involved in Task #3 :

3 Extract the data you need from your database tables

At the moment, your webapp's code dips in and out of the `swimdata` folder as needed to grab the data required to do its thing. With Task #2 complete, you can write Python and SQL to grab the data your webapp needs from your database tables instead.

We've been making great progress, but…



Maybe. That's an excellent point.

Tuple-lists are always returned from the **fetchall** method, and are passed back to the code which called your database function untouched.

Depending on what you have planned for the data, it might be an idea to perform a bit of a transformation on the tuple-lists.

But, which transformations are required here and where in your code should they occur?

Cubicle Conversation



Mara: Does it really matter where we perform these transformations?

Sam: I think it might.

Alex: Really? I vote we perform the transformations inside `data_utils.py` then move on.

Sam: We could do that, but then any other programmer who uses the `data_utils.py` module is stuck with whatever transformations we decide on, and they are out of luck if they actually want a tuple-list.

Mara: How likely is that to happen, though?

Sam: Well…you never do know, do you? It might well be the case that nobody else needs to use the data, but, it's impossible to know. One thing is sure, applications have a habit of never being finished, and they grow in features over time, so more use-cases for the data can't be dismissed.

Alex: So, we're OK with placing an additional burden on the user of our database module to perform their own transformations?

Sam: Yes, as only the user's of the data module know what they want to do with the data. We can't possible know, can we?

Mara: I guess not. So, to be clear, you're suggesting we keep the `data_utils.py` module as-is, but leaving it up to programmers us-

ing the module to decide on the transformations they need?

Sam: Yes.

Alex: Please tell me there's an easy way to perform these transformations…

Sam: Of course there is, and you've already seen them in action. This is a perfect job for *list comprehensions*.

EXERCISE



Continue to work within your `TestDataUtils.ipynb` notebook. This next cell assigns the results returned from your first data-access function to a new variable called `data`, producing the output as shown:

```
data = data_utils.get_swim_sessions()
data
```

```
[('2023-01-13 20:11:34',), ('2023-01-20 21:18:25',)]
```

In your next notebook cell, craft a list comprehension to transform the above output into a list which looks like this:

```
['2023-01-13', '2023-01-20']
```

Write the list comprehension you used in the space below:

Hint: if you are struggling to get started with your list comprehension, begin by writing a "regular" **for** loop to do the transformation, then convert the multi-line **for** loop into a single-line list comprehension.

Here's the results from your second data-access function, which is once again assigned to the `data` variable then displayed on screen:

```
data = data_utils.get_session_swimmers("2023-01-20")
print(data)
```

```
[('Abi', 10), ('Blake', 15), ('Darius', 13), ('Darius', 8), ('Dave', 17), ('Katie', 9), ('Maria', 9), ('Owen', 15)]
```

Continuing to work in your next code cell, craft a list comprehension which produces this output:

```
['Abi-10', 'Blake-15', 'Darius-13', 'Darius-8',
```

Write the list comprehension you used in this space:

Hint: consider using an f-string to perform the transformation required by your list comprehension.

Let's move onto the third function. As on the previous page, assign the results from the function call to the `data` variable, then display `data` on screen:

```
data = data_utils.get_swimmers_events("Darius", 8, "2023-01-20")
data
```

```
[('50m', 'Back'), ('50m', 'Free'), ('50m', 'Breast')]
```

Here's what your next list comprehension needs to transform `data` into:

```
['50m Back', '50m Free', '50m Breast']
```

Write the list comprehension you'd used to perform the above transformation here:

Hint: another f-string is what you need here (above), as well as with the final comprehension (below).

And, finally, the fourth data-access function produces this output:

```
data = data_utils.get_swimmers_times("Darius", 8, "50m", "Free", "2023-01-20")
data
```

```
[('39.42',), ('36.13',), ('37.66',), ('39.07',)]
```

Here's what this version of `data` needs to be transformed into:

```
['39.42', '36.13', '37.66', '39.07']
```

In the space below, write the list comprehension you'd use to do this final transformation:

NOTE

When you're ready, compare your list comprehensions with ours (on the next page).

EXERCISE SOLUTION



Continue to work within your `TestDataUtils.ipynb` notebook. This next cell assigns the results returned from your first data-access function to a new variable called `data`, producing the output as shown:

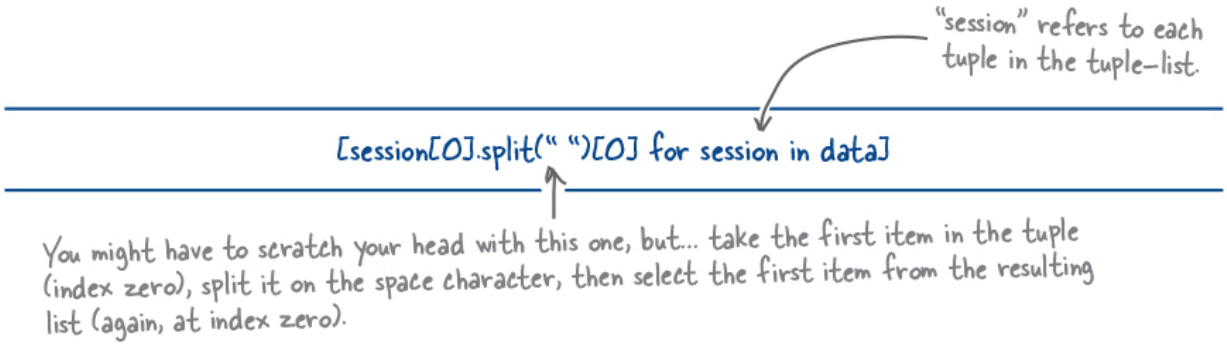
```
data = data_utils.get_swim_sessions()
data
```

```
[('2023-01-13 20:11:34',), ('2023-01-20 21:18:25',)]
```

In your next notebook cell, craft a list comprehension to transform the above output into a list which looks like this:

```
['2023-01-13', '2023-01-20']
```

You were to write the list comprehension you used in the space below:



Here's the results from your second data-access function, which is once again assigned to the `data` variable then displayed on screen:

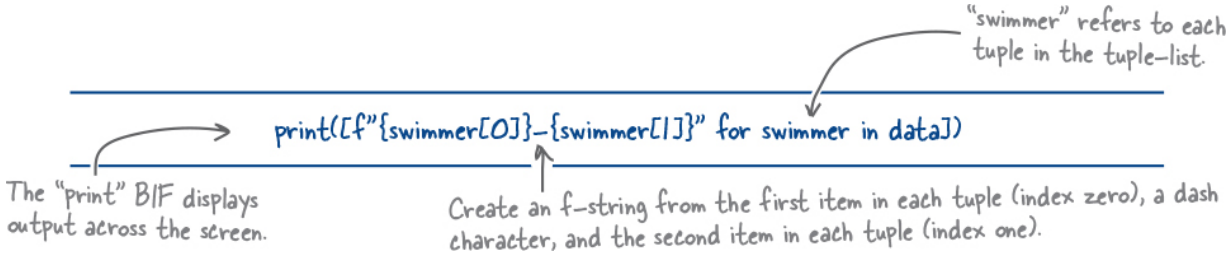
```
data = data_utils.get_session_swimmers("2023-01-20")
print(data)
```

```
[('Abi', 10), ('Blake', 15), ('Darius', 13), ('Darius', 8), ('Dave', 17), ('Katie', 9), ('Maria', 9), ('Owen', 15)]
```

Continuing to work in your next code cell, craft a list comprehension which produces this output:

```
['Abi-10', 'Blake-15', 'Darius-13', 'Darius-8',
```

As before, you were to write the list comprehension you used in this space:



Let's move onto the third function. As on the previous page, assign the results from the function call to the `data` variable, then display `data` on screen:

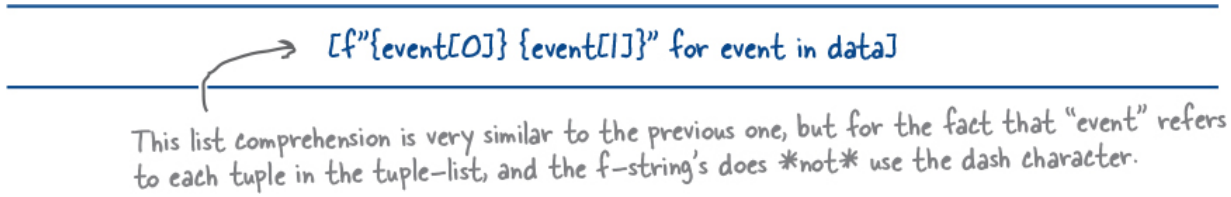
```
data = data_utils.get_swimmers_events("Darius", 8, "2023-01-20")
data
```

```
[('50m', 'Back'), ('50m', 'Free'), ('50m', 'Breast')]
```

Here's what your next list comprehension needs to transform `data` into:

```
['50m Back', '50m Free', '50m Breast']
```

You were to, once again, write the list comprehension you'd used to perform the above transformation here:



And, finally, the fourth data-access function produces this output:

```
data = data_utils.get_swimmers_times("Darius", 8, "50m", "Free", "2023-01-20")
data
```

```
[('39.42',), ('36.13',), ('37.66',), ('39.07',)]
```

Here's what this version of `data` needs to be transformed into:

```
['39.42', '36.13', '37.66', '39.07']
```

In the space below, you were to write the list comprehension you'd use to do this final transformation:

```
[time[0] for time in data]
```

We kept the most straightforward list comprehension to the end. All that happens here is that each tuple (in "time") has it's first item (index zero) retrieved.

TEST DRIVE



Let’s confirm the lists produced by each of your list comprehensions transformations are as expected.

```
data = data_utils.get_swim_sessions()
data
```

```
[('2023-01-13 20:11:34',), ('2023-01-20 21:18:25',)]
```

```
[session[0].split(" ")[0] for session in data]
```

```
['2023-01-13', '2023-01-20']
```

You might have to read this one more than once. You take the first item from each tuple, split it, then select the "date" part.

The list comprehension at the start of this chapter contained a "if" condition. As stated then, the "filter-" part is optional. None of the four list comprehensions associated with this Test Drive require a filter (and that's OK).

```
data = data_utils.get_session_swimmers("2023-01-20")
print(data)
```

```
[('Abi', 10), ('Blake', 15), ('Darius', 13), ('Darius', 8), ('Dave', 17),
('Katie', 9), ('Maria', 9), ('Owen', 15)]
```

```
print([f"{swimmer[0]}-{swimmer[1]}" for swimmer in data])
```

```
['Abi-10', 'Blake-15', 'Darius-13', 'Darius-8', 'Dave-17', 'Katie-9', 'Maria-9',
'Owen-15']
```

The f-string performs the transformation here, picking out the two values from each tuple, then inserting them into an appropriately formatted string.

```
data = data_utils.get_swimmers_events("Darius", 8, "2023-01-20")
data
```

```
[('50m', 'Back'), ('50m', 'Free'), ('50m', 'Breast')]
```

```
[f"{event[0]} {event[1]}" for event in data]
```

```
['50m Back', '50m Free', '50m Breast']
```

This transformation follows the same pattern as the previous list comprehension (without the dash character, that is).

```
data = data_utils.get_swimmers_times("Darius", 8, "50m", "Free", "2023-01-20")
data
```

```
[('39.42',), ('36.13',), ('37.66',), ('39.07',)]
```

```
[time[0] for time in data]
```

```
['39.42', '36.13', '37.66', '39.07']
```

You can think of this list comprehension as asking for the data item from each tuple in the tuple-list. It's not really a transformation, more of a data item grab (and that's a perfectly fine use of Python's list comprehension technology).

It's time to integrate your database code!

Finally, we're onto Task #4.

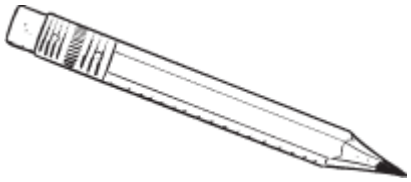
This is what this (and the previous chapter) has been building up to: integrating your data access functions into your existing webapp code so that the Coach can work with any number of swim session datasets. Here's what's required of Task #4:

4 Adjust your existing webapp code to use the data in your database tables

With Task #3 done, you'll need to change your current webapp's code to grab its data from your database as opposed to from the `swimdata` folder.

It's time to remind yourself of the current state of your webapp's code, so you can work out where the changes are to be applied.

SHARPEN YOUR PENCIL



Here's the code to your webapp's `app.py` file (on this and the next page).Take as long as you need to read this code, then grab your pencil and encircle those parts of the code which you think need to change in order to use the functions in `data_utils.py`.

```
from flask import Flask, session, render_template

import os
import swimclub


app = Flask(__name__)
app.secret_key = "You will never guess..."


@aapp.get("/")
def index():
    return render_template(
        "index.html",
        title="Welcome to Swimclub",
    )
```

```
def populate_data():
    swim_files = os.listdir(swimclub.FOLDER)
    swim_files.remove(".DS_Store")
    session["swimmers"] = {}
    for file in swim_files:
        name, *_ = swimclub.get_swim_data(file)
        if name not in  session["swimmers"]:
            session["swimmers"][name] = []
        session["swimmers"][name].append(file)


@app.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]),
    )


@app.post("/showfiles")
def display_swimmer_files():
```

```
    if "swimmers" not in session:
        populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"][name],
    )


@app.post("/showbarchart")
def show_bar_chart():
    file_id = request.form["file"]
    location = swimclub.produce_bar_chart(file_id
    return render_template(location.split("/")[-1

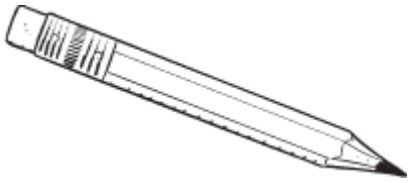

if __name__ == "__main__":
    app.run(debug=True)
```

BRAIN POWER



Other than replacing your webapp's existing data-access mechanisms with your new database-enabled functions, can you think of any other changes that are required to your webapp's code in order to support the Coach's multiple datasets?

SHARPEN YOUR PENCIL SOLUTION



You were shown the code to your webapp's `app.py` file, then asked to take as long as was required to read the code. Grabbing your pencil, you were then to encircle those parts of the code which you thought need to change in order to use the functions in `data_utils.py`.

```
from flask import Flask, session, render_template, request

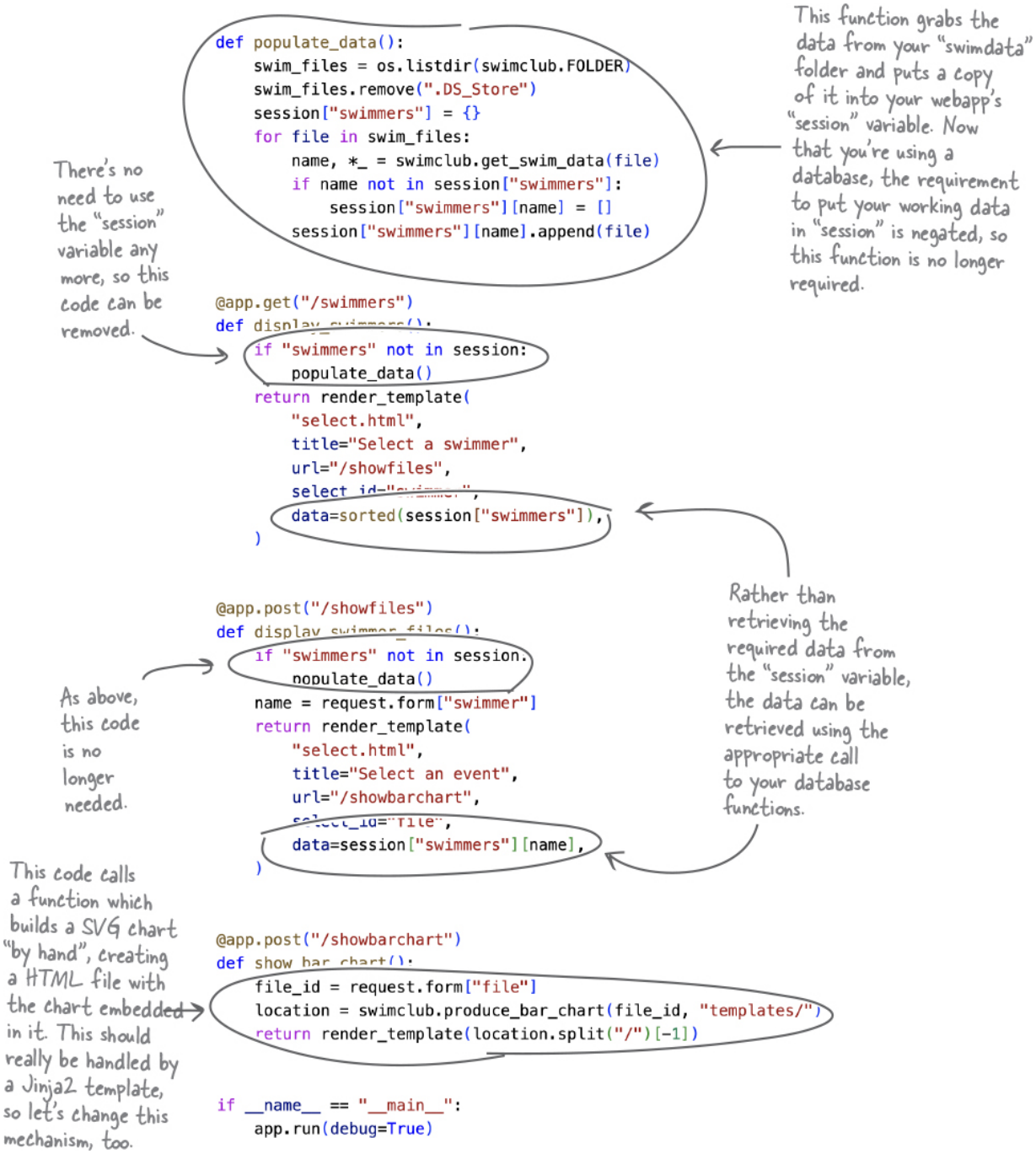
import os
import swimclub

app = Flask(__name__)
app.secret_key = "You will never guess..."


@app.get("/")
def index():
    return render_template(
        "index.html",
        title="Welcome to Swimclub",
    )
```

These imports will likely change, as you are no longer grabbing the webapp's data from your "swimdata" folder. Instead, you're using the data stored in your SQLite3 database engine.

No other changes are required to the code on this page. However, do you think another URL is needed here? Do you need to add new code to allow the Coach to select which session dataset to use?

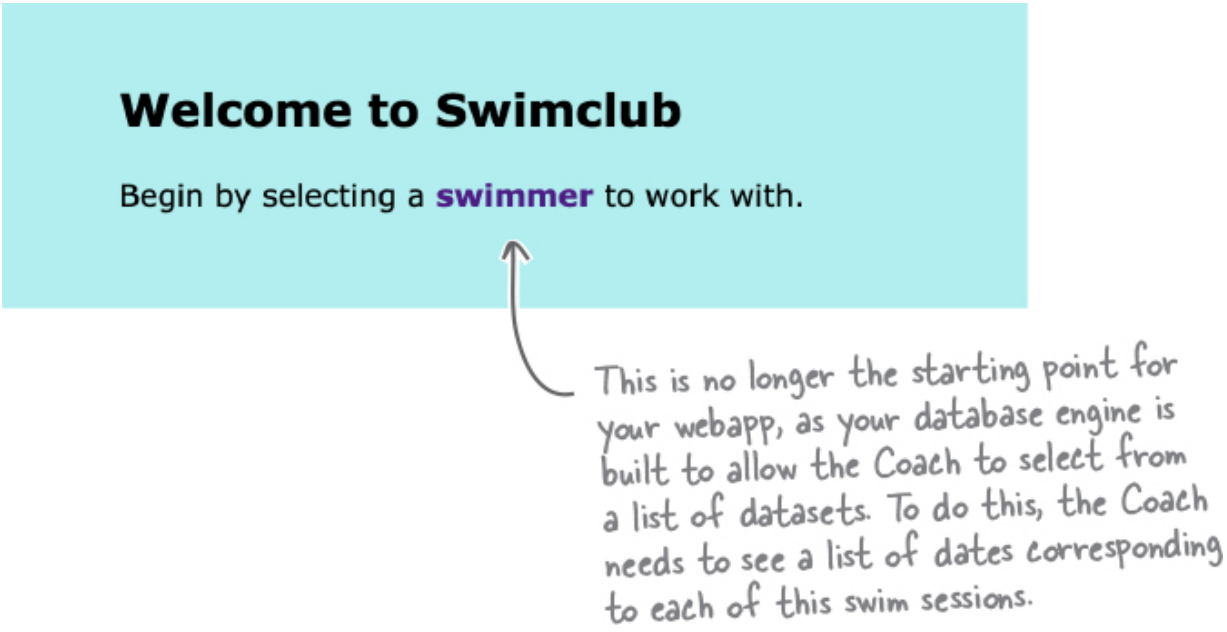

def populate_data():
swim_files = os.listdir(swimclub.FOLDER)
swim_files.remove(".DS_Store")
session["swimmers"] = {}
for file in swim_files:
name, *_ = swimclub.get_swim_data(file)
if name not in session["swimmers"]:
session["swimmers"][name] = []
session["swimmers"][name].append(file)
This function grabs the data from your "swimdata" folder and puts a copy of it into your webapp's "session" variable. Now that you're using a database, the requirement to put your working data in "session" is negated, so this function is no longer required.
There's no need to use the "session" variable any more, so this code can be removed.
@app.get("/swimmers")
def display_swimmers():
if "swimmers" not in session:
populate_data()
return render_template(
"select.html",
title="Select a swimmer",
url="/showfiles",
select_id="swimmer",
data=sorted(session["swimmers"]),
)
@app.post("/showfiles")
def display_swimmer_files():
if "swimmers" not in session:
populate_data()
name = request.form["swimmer"]
return render_template(
"select.html",
title="Select an event",
url="/showbarchart",
select_id="file",
data=session["swimmers"][name],
)
Rather than retrieving the required data from the "session" variable, the data can be retrieved using the appropriate call to your database functions.
As above, this code is no longer needed.
This code calls a function which builds a SVG chart "by hand", creating a HTML file with the chart embedded in it. This should really be handled by a Jinja2 template, so let's change this mechanism, too.
@app.post("/showbarchart")
def show_bar_chart():
file_id = request.form["file"]
location = swimclub.produce_bar_chart(file_id, "templates/")
return render_template(location.split("/")[-1])
if __name__ == "__main__":
app.run(debug=True)

Updating your existing webapp's code

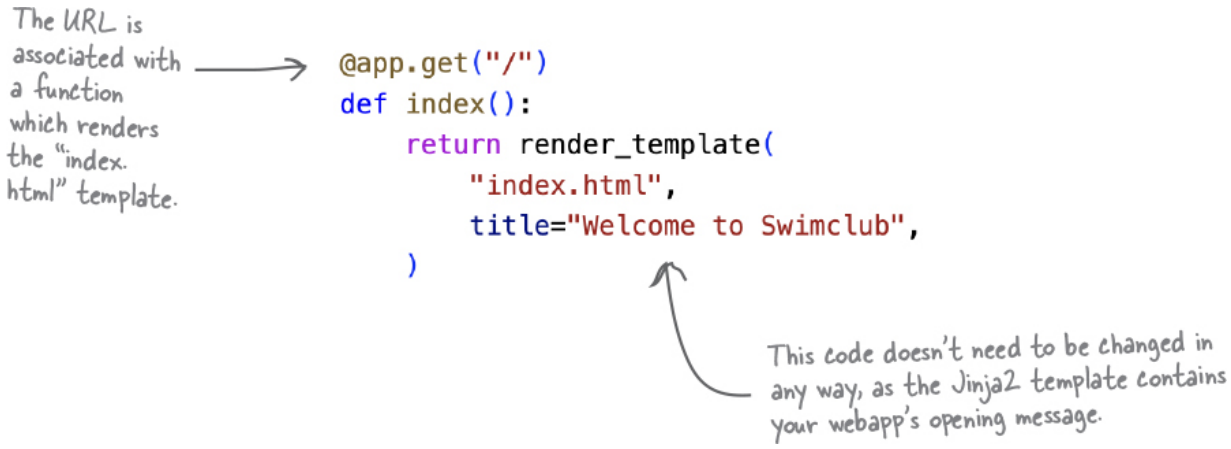
Let's fire-up the Coach's webapp to remind yourself of your user's flow through the screens.

To run your webapp, open `app.py` in VS Code, then select **Run** then **Run Without Debugging** from the menu system. A terminal opens in VS Code and your webapp starts. You'll see the URL your webapp is running on in the terminal.

When you're ready, visit the web address with your favorite browser. You'll be greeted with a web page which contains this content:



Here's the code which displays the above content:



Let's take a look at the `index.html` template to determine what needs to change.

Review your template(s) for changes…

Take a look at `index.html`. It's fairly obvious that this template needs to change so that the Coach is advised to begin by selecting a swim session, *not* a swimmer:

```
{% extends "base.html" %}

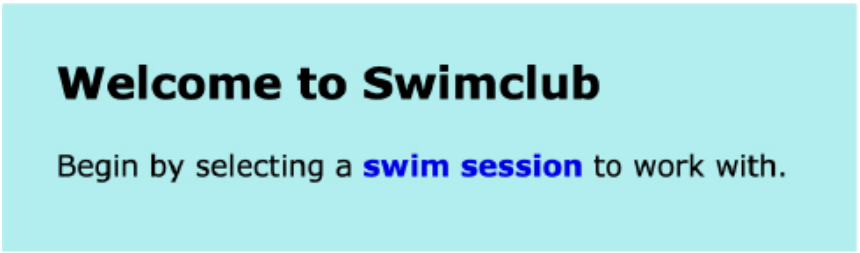
{% block body %}
<p>
    Begin by selecting a <a href="/swimmers">swimmer</a> to work with.
</p>
{% endblock %}
```

This is no longer your webapp's starting point.

It's a trivial edit (to the template). Adjust your `index.html` template so that the `<p>` content looks like this:

```
Begin by selecting a <a href="/swims">swim sessio
```

If you save your template then reload your webapp's opening page in your browser, it should come as no surprise that the page updates:

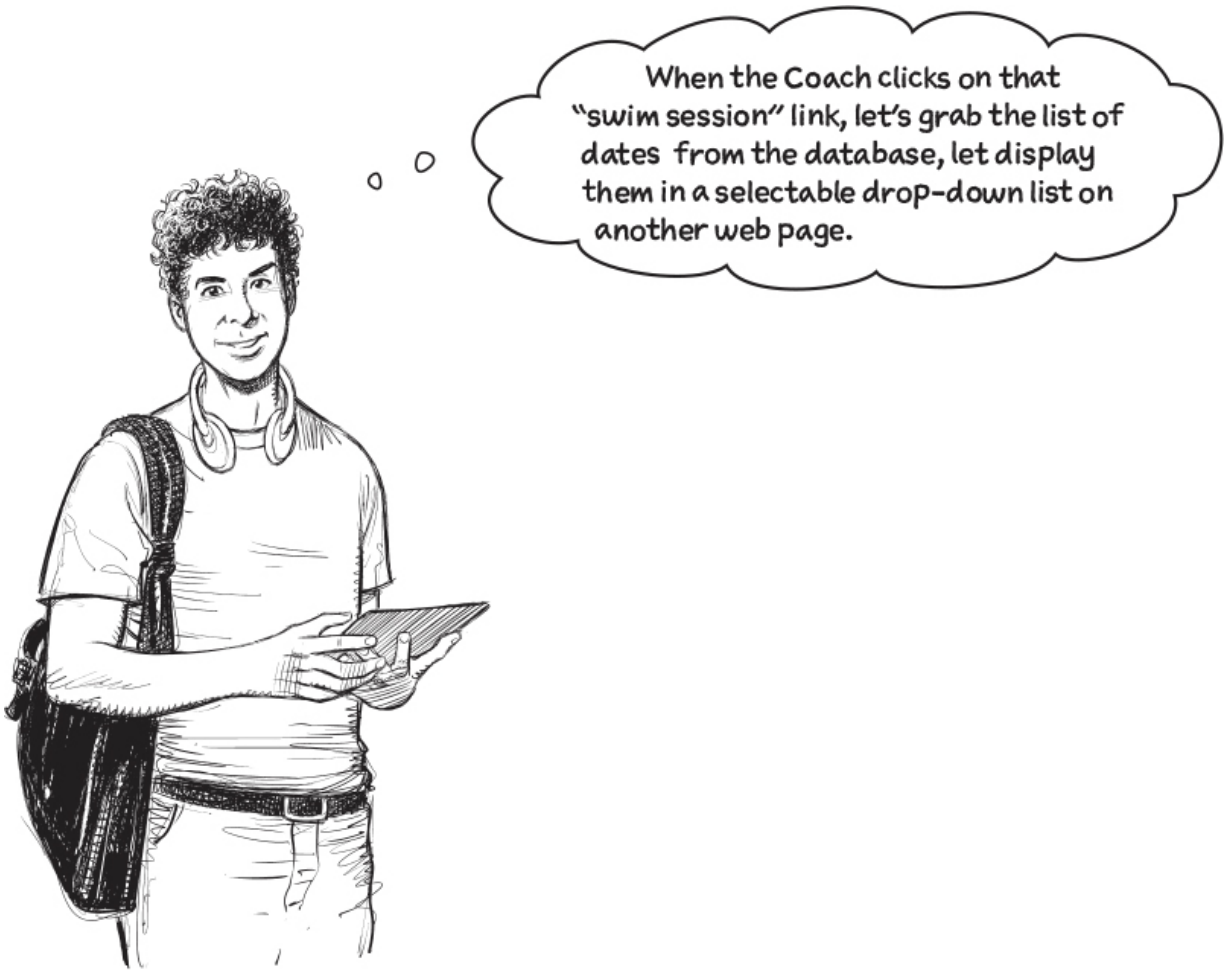


Nothing to get too excited about here, as your template update worked as expected.

We are *deliberately* taking our time here, as it has been a while since you looked at your webapp code. Rather than force you to skip back

to the chapter that originally covered this material, we want you to concentrate on the job at hand, so we're approaching these edits baby-step-by-baby-step *on purpose*.

Now… ask yourself, what does the `/swim` URL need to do?

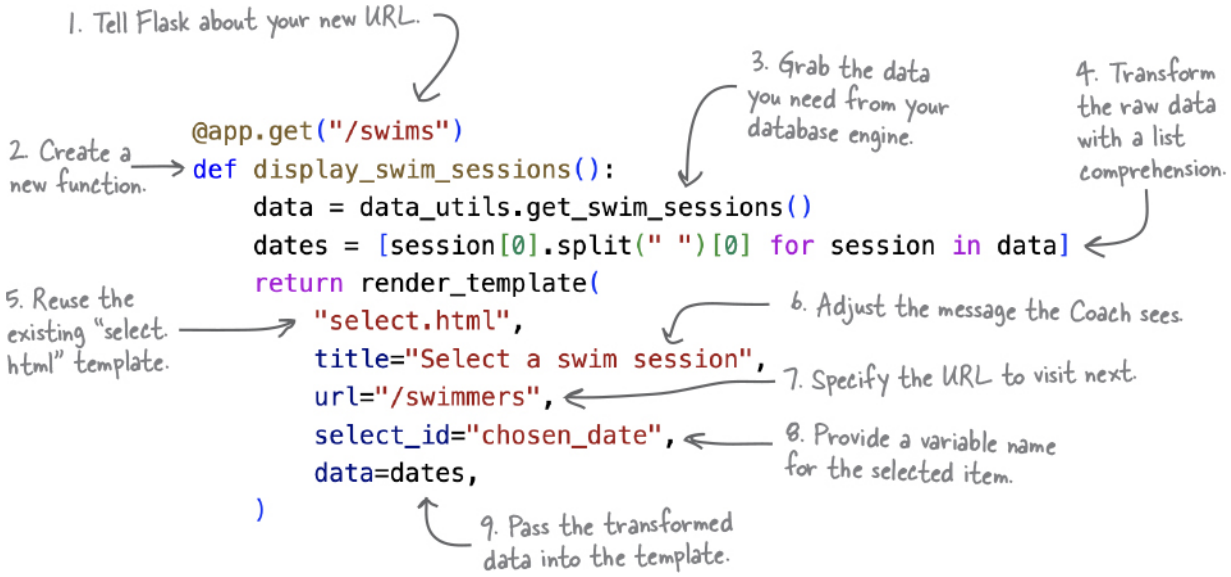


That sounds like a good plan.

Take a quick look at your webapp code which responds to a click of the `/swimmers` URL:

```
@app.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]),
    )
```

New code is required to handle the `/swims` URL. Let's base it on the `/swimmers` URL code. Following the suggested steps, as outlined above, code like this does the trick:



TEST DRIVE



Go ahead and add the code from the bottom of the previous page to your `app.py` file. Additionally, add this **import** statement to the top of `app.py` so the calls to your database utilities run without error:

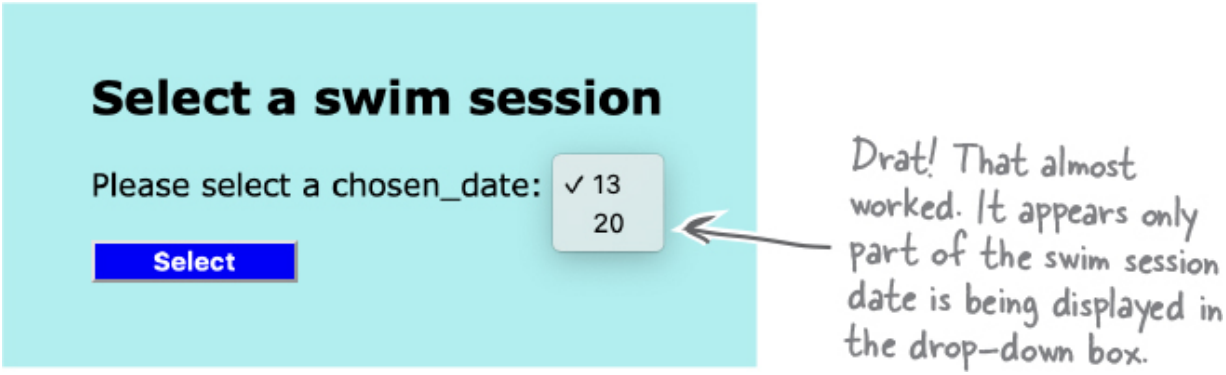
```
import data_utils
```

Save your updated `app.py` code within VS Code. If you've typed everything correctly, your webapp should *reload automatically* and you'll see a message similar to this in your VS Code terminal:

```
* Detected change in '/Users/barryp/Desktop/THIRD/Learning/webapp/app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 115-518-560
```

The latest version of your webapp's code is now active (and we're hoping, by now, this is all starting to come back to you).

Regrettably, when you click on the "swim session" link on your webapp's homepage, you're in for a bit of a surprise. Here's what we saw on screen:



The question is: what's causing this error?



Let's take a look and see…

So… what's the deal with your template?

Here's the current Jinja2 code for `select.html` which is used a few times by the Coach's webapp.

The question posed is: *what's wrong with this template?*

```
{% extends "base.html" %}

{% block body %}

<form method="POST" action="{{ url }}">

    <label for="{{ select_id }}">Please select a {{ select_id }}:</label>

    <select name="{{ select_id }}" id="{{ select_id }}">

        {% for name in data %}
            <option value="{{ name }}">{{ name.removesuffix(".txt").split("-", 2)[-1] }}</option>
        {% endfor %}

    </select>

    <p>
        <input type="submit" value="Select">
    </p>

</form>

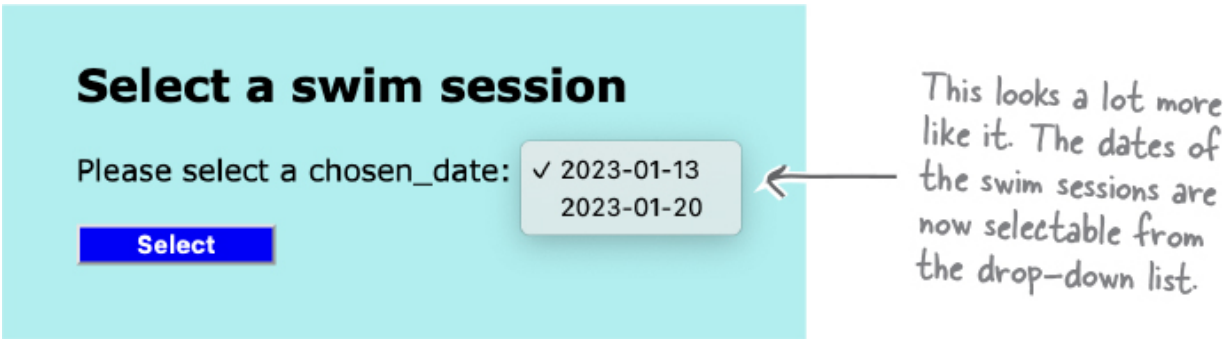
{% endblock %}
```

It didn't take too long for us to spot the problem. This manipulation was added to the template to process the filenames in the "swimdata" folder. Is this still needed?

Now that you're updating your webapp to grab the data it needs from your database engine, the need for this sort of filename-manipulation "hack" shown above is removed. Adjust that `<option>` line so that it looks like this:

```
<option value="{{ name }}">{{ name }}</option>
```

Go ahead and save everything in VS Code, confirm your webapp has reloaded, then generate your drop-down menu once more:



EXERCISE



Here is the code for the `display_swimmers` function, taken from your current version of `app.py`:

```
@app.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
```

```
        populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]),
```

Clicking on the **Select** button at the bottom of the previous page *posts* the selected date to the `/swimmers` URL, with the posted value assigned to the `chosen_date` variable. Adjust the code above to receive the posted value, then use `chosen_date` to filter the list of swimmer names displayed on screen. As the value of `chosen_date` is likely to be used in future functions, be sure to save the date to Flask's `session` variable.

Hint: if you have forgotten how to retrieve a form value, take a quick look at the `display_swimmer_file` code.


Write in the code you used here.

EXERCISE SOLUTION

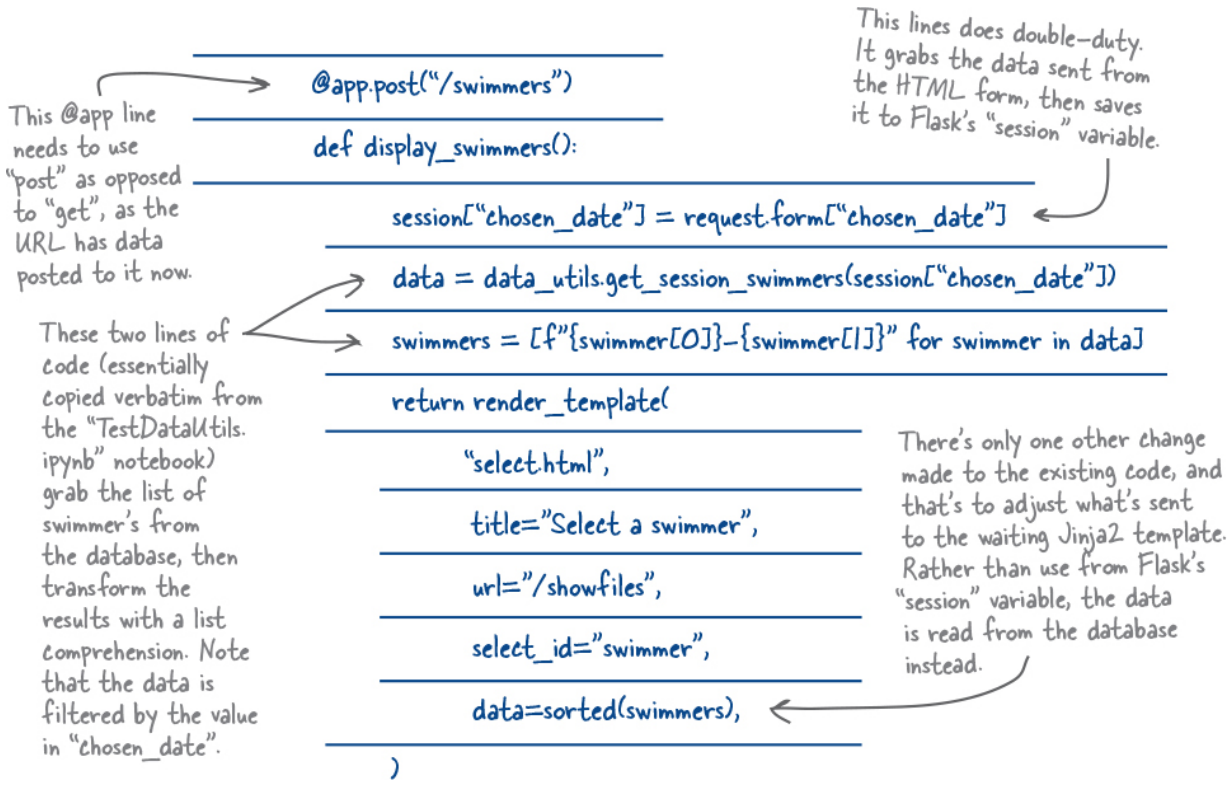


Here is the code for the `display_swimmers` function, taken from your current version of `app.py`:

```
@app.get("/swimmers")
def display_swimmers():
    if "swimmers" not in session:
        populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]),
    )
```

Clicking on the **Select** button at the bottom of the previous page *posts* the selected date to the `/swimmers` URL, with the posted value assigned to the `chosen_date` form variable. You were to adjust the code above to receive the posted value, then use c`hosen_date` to filter the list of swimmer names displayed on screen. As the value

of `chosen_date` is likely to be used in future functions, you were asked save the date to Flask's `session` variable. Here's the code we used for all of this:



TEST DRIVE

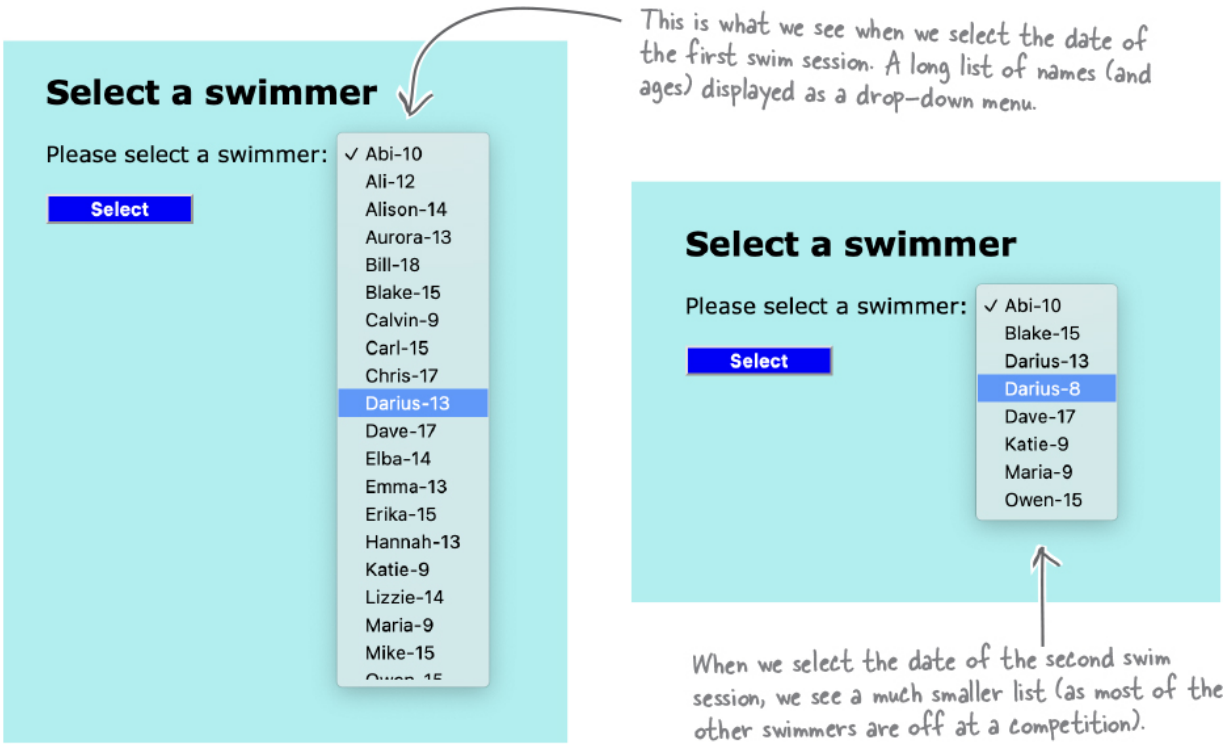


When you save your latest version of `app.py`, your webapp reloads.

Here's the code to the updated `display_swimmers` function which filters the list of swimmers by date:

```
@app.post("/swimmers")
def display_swimmers():
    session["chosen_date"] = request.form["chosen
    data = data_utils.get_session_swimmers(sessio
    swimmers = [f"{swimmer[0]}-{swimmer[1]}" for
    return render_template(
        "select-html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(swimmers),
)
```

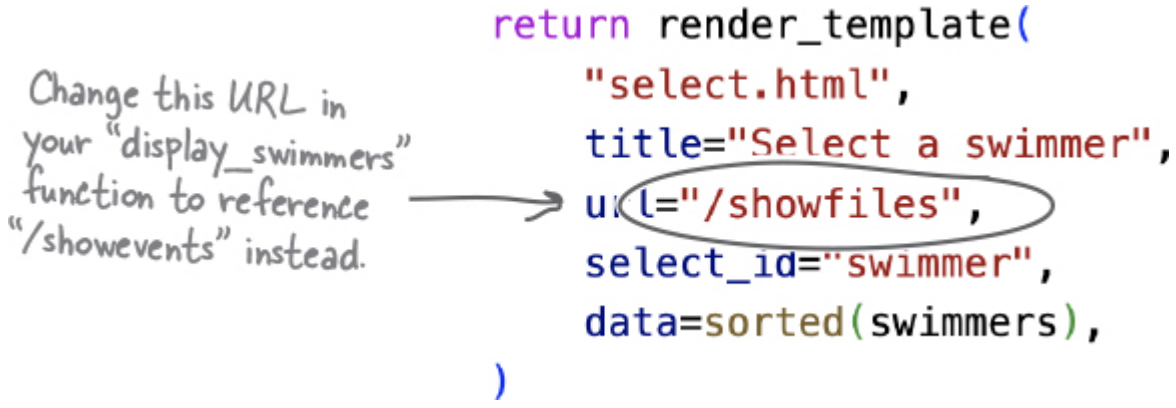
The swim session chosen impacts the list of swimmers the Coach sees in his browser:



Let's display a list of events…

The next URL your webapp interacts with is `/showfiles`. As URL names go, this one made sense when your webapp processed the files in your `swimdata` folder. Now that your webapp is grabbing its data from your database engine, this name need to change. Let's use `/showevents` instead.

Before you get to the next function in your webapp (`display_swimmer_files`), let's make a change to the previous one:





We think this is a worthwhile change.

Making a small change to this URL *now* ensures your code more closely matches what's actually happening. And don't forget this is not the last time you're likely to interact with `app.py`. The last thing you'll want in three months time (when you or some other programmer is reading this code) is for someone to be asking the question: *Why is this URL called /showfiles when it displays event data?*

EXERCISE



Here is the code for the `display_swimmer_files` function from `app.py`:

```
@app.post("/showfiles")
def display_swimmer_files():
    if "swimmers" not in session:
        populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"] [name],
    )
```

We've made a few changes for you, below. We adjusted the URL to be `/showevents`, then changed the function's name to `display_swimmer_events`. The next line of code uses multiple

assignment to assign the selected swimmer data from your HTML form into the Flask `session` variable:

```
@app.post("/showevents")
def display_swimmer_events():
    session["swimmer"], session["age"] = request.form["swimmer"].split("-")
```



EXERCISE SOLUTION

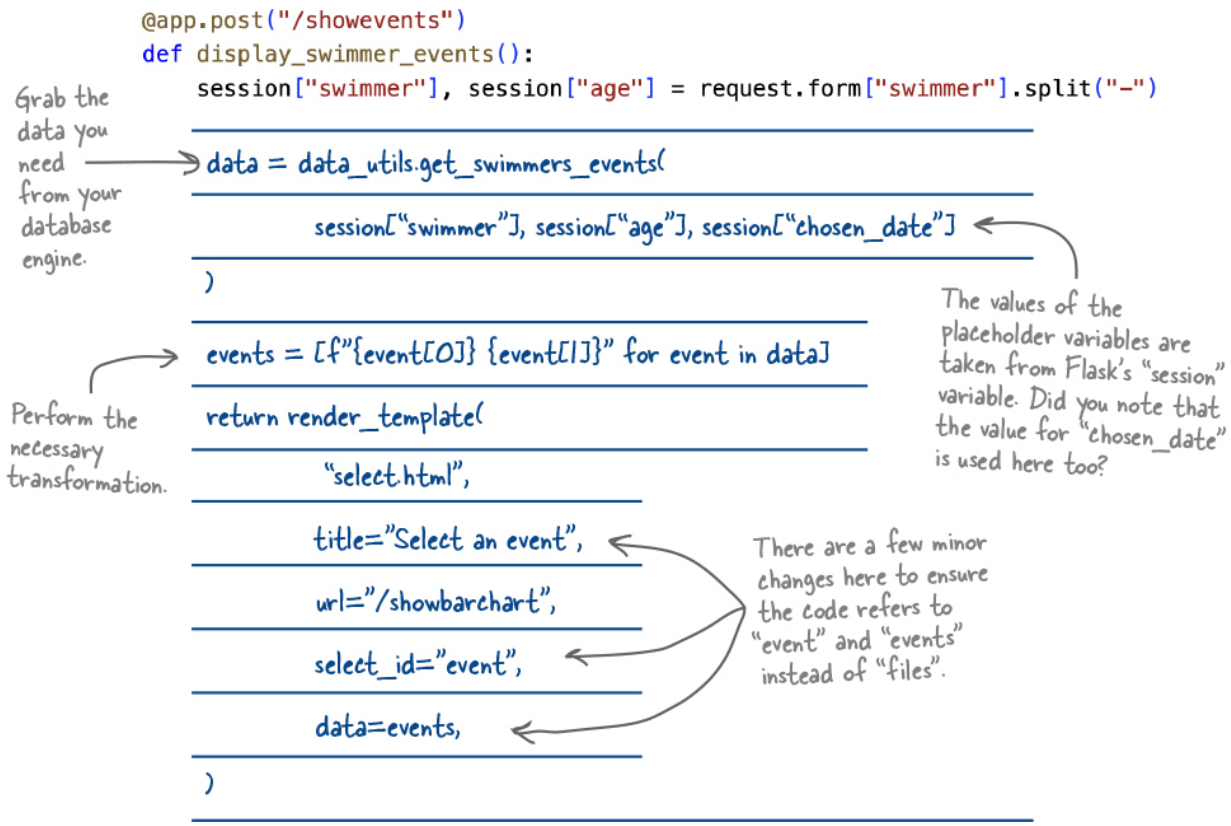


Here is the code for the `display_swimmer_files` function from `app.py`:

```
@app.post("/showfiles")
def display_swimmer_files():
    if "swimmers" not in session:
        populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"][name],
    )
```

We adjusted the URL to refer to `/showevents`, then changed the function's name to `display_swimmer_events`. The next line of code used multiple assignment to assign the selected swimmer data

from your HTML form into the Flask `session` variable. You were to provide the rest of this function's code in the space provided:



TEST DRIVE

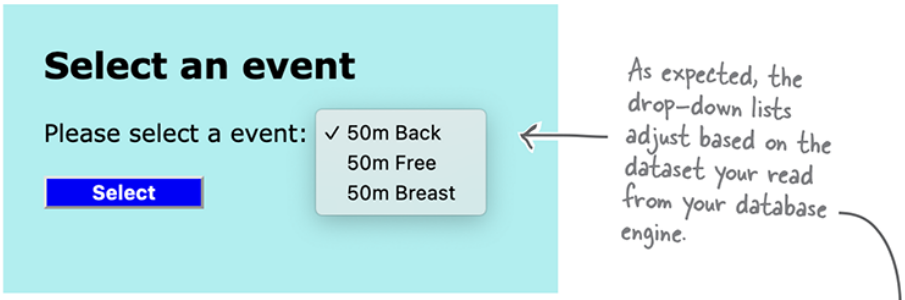


With the most-recent changes to your `app.py` code applied and saved, your webapp reloads.

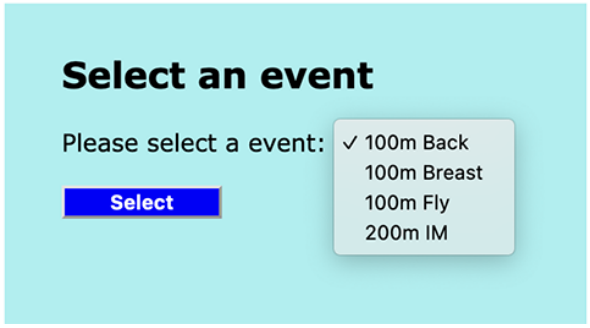
The code to your latest function should look like this:

```
@app. post("/showevents")
def display_swimmer_events():
    session["swimmer"], session["age"] = request.
    data = data_utils.get_swimmers_events(
        session ["swimmer"], session ["age"], ses
    )
    events = [f"{event [0]} {event [1]}" for ever
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="event",
        data=events,
    )
```

If you select the second swim session from the list of dates, then choose the seven year old Darius from the list of swimmers, your database-driven webapp displays these list of events for the younger Darius:



Use your browser's *Back* button to return to your webapp's home page, select the first training session, then select the older Darius from the list of swimmers. You should now see this drop-down list:



All that's lef t is to draw the bar chart…

Here's the code to the `show_bar_chart` function which, in the current version of the Coach's webapp, grabs the data it needs from a

swimmer's file, then calls the `produce_bar_chart` function from the `swimclub` module to do all the heavy lifting:

```
@app.post("/showbarchart")
def show_bar_chart():
    file_id = request.form["file"]
    location = swimclub.produce_bar_chart(file_id, "templates/")
    return render_template(location.split("/")[-1])
```

Only five lines of code here. How hard can it be to adjust this to grab the data it needs from your database engine?

Recall that you already have some code which grabs and transforms the raw data in your database into a list of times:

Grab the raw data...

```
data = data_utils.get_swimmers_times("Darius", 8, "50m", "Free", "2023-01-20")
data
```

```
[('39.42',), ('36.13',), ('37.66',), ('39.07',)]
```

```
[time[0] for time in data]
```

```
['39.42', '36.13', '37.66', '39.07']
```

...and turn it into a list of times.

If you consider the five argument values the `get_swimmers_times` function requires, three of them are already

saved in Flask's `session` variable, namely `chosen_date`, `swimmer`, and `age`. The other two arguments, the *distance* and *stroke* values, are available to you in the submitted data from your HTML form, which this line of code extracts and assigns to the shown variables:

```
distance, stroke = request.form["event"].split(" ")
```

Your old friends, multiple assignment and "split", help you out once more.



Yes, you have all you need to extract the data from your database.

However, on it's own, the data isn't enough.

Your `swimclub` module, which you started working on back in Chapter 3, abstracts away additional processing applied to your list of swim times . Recall that the `produce_bar_chart` function (with some help from the `get_swim_data` function) performs a bunch of conversions, lookups, and manipulations to the swim times before "manually" crafting a HTML webpage containing a SVG bar chart which is saved to disk before being rendered by your webapp.

Now, there's an argument to be made for continuing to rely on the `swimclub` module to perform this work. However, doing so would mean continuing to rely on the fact that the data in your `swimdata` folder doesn't change… a situation which has already created problems for the Coach. You need to use the data in your database to draw your chart by sending any and all required data to a new Jinja2 template, performing the work previously performed by `produce_bar_chart`. To solves the Coach's issue, you need to stop relying on the existence of all those files. Of course, that's not to say you should throw out *all* the code in `swimclub.py`, as it is likely some of it can be salvaged.

On the next page, we present the current code for `swimclub.py` and – in the provided annotations – indicate which chunks we think are candidates for reuse.

Reviewing the most-recent swimclub.py code

```
import statistics

import hfpy_utils
from records import records


CHARTS = "charts/"

FOLDER = "swimdata/"


def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    scaled = []
    for t in times:
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = f"{(average / 100):.2f}".split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = f"{minutes}:{seconds}.{hundredths}"
    return swimmer, age, distance, stroke, average, average_str, times, converts


def produce_bar_chart(fn, location=CHARTS):
    """Given the name of a swimmer's file, produce a HTML/SVG-based bar chart, saving
    it to the "location" folder. Return the path to the bar chart file."""

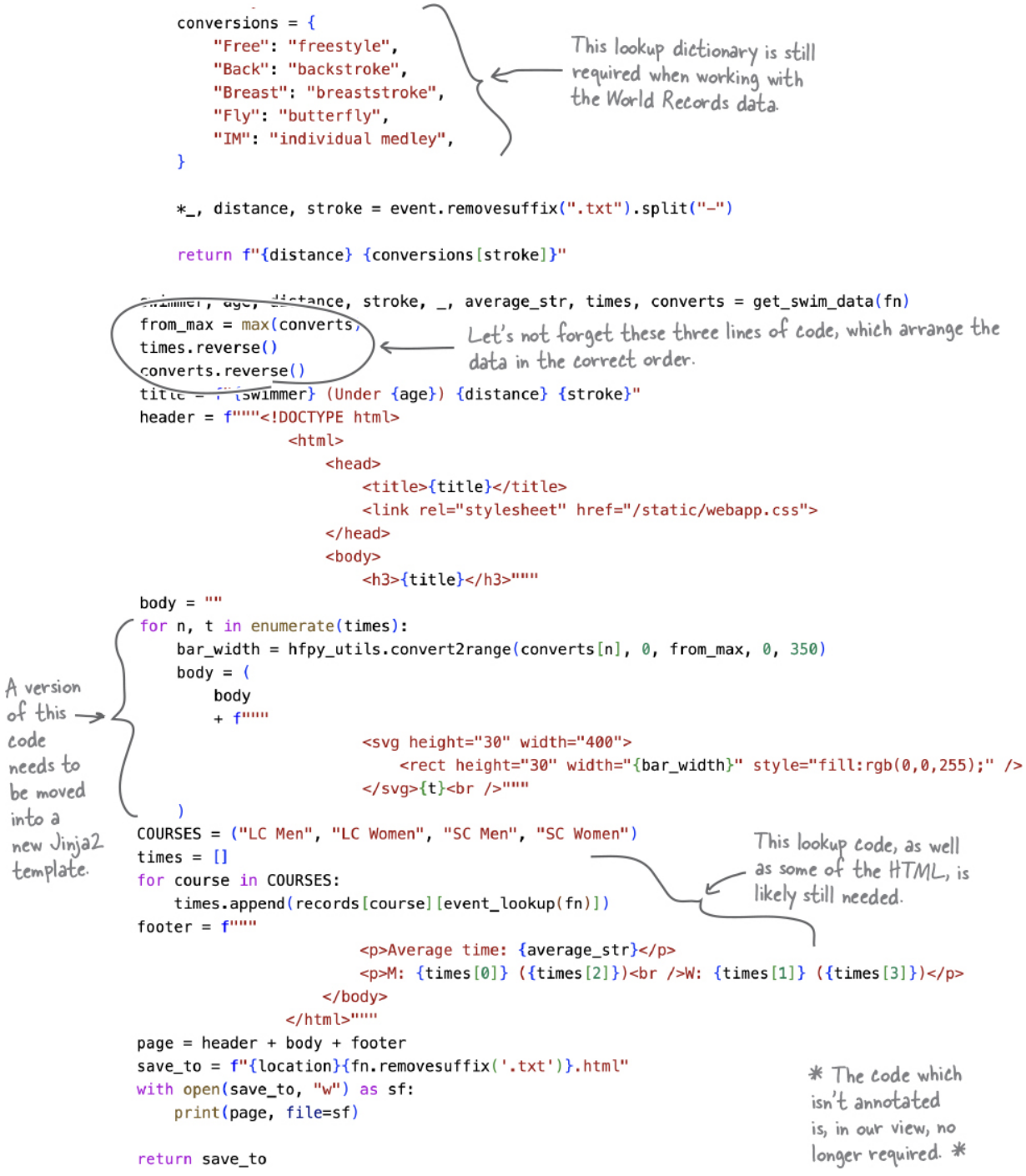
    def event_lookup(event):
        """Given an event descriptor (the name of a swimmer's file), convert
        the descriptor into a lookup key which can be used with the "records"
        dictonary."""
```

You'll need to decide which of these imports are still needed.

This code converts the times as strings into times as hundredths of seconds.

The bar chart reports the average swim time, and the calculations are performed by this code.

Recall that this is an example of an nested function (which is only visible within its enclosing function).



Meet the SVG-generating Jinja2 template

Here's a Jinja2 template which, when provided with the appropriate data values, generates a bar chart much like the code from `swimclub.py` did:

```
{% extends "base.html" %}

{% block body %}

    {% for row in data %}
        <svg height="30" width="400">
            <rect height="30" width="{{ row[1] }}" style="fill:■rgb(0,0,255);" />
        </svg>{{ row[0] }}<br />
    {% endfor %}
    <p>Average time: {{ average }}</p>
    M: {{worlds[0]}} ({{worlds[2]}})<br />W: {{worlds[1]}} ({{worlds[3]}})

{% endblock %}
```

Jinja2's "for" statement (which is modelled on Python's "for" loop) does most of the work here, adding an SVG tag for each of your chart's bars.

Remember: using Jinja2's double curly-braces lets you insert any valid Python expression into your template.

The first thing to note is that most of the code from `swimclub.py` which created the complete HTML page is *gone*. Remember: the `base.html` template, which the above template inherits from, contains all of the shared HTML, including the header and footer HTML. All the inheriting templates have to provide is the content of their HTML page's body.

Go ahead a create a new file in your `templates` folder, called `chart.html`, then add the template code shown above to the file.

As you do so, note the variable names surrounded by double curly braces, {{ and }}:

NOTE

Do this now.

- `row` – a two item list which takes its values from `data`,
- `data` – a list of two-tuples which contain the time string and it's scaled numeric equivalent,
- `average` – the "stringified" representation of the average time value, and
- `worlds` – a four-item list containing the four swimming world records associated with the current event.

All but the first of these values, `row`, needs to be passed into the template from your `app.py code`. All that's required now is to re-use the code from `swimclub.py` to produce these values. Before getting to that code, take a moment to consider an important Python maxim.

Code is read more than it's written.

It's a simple statement, with a huge implication: like it or not, as a programmer, you'll spend more time *reading* code than *writing* code, and Python is optimized to make your code (and that of other Python programmers) *easy to read*.

With this in mind, over the page, we present the "reused" code from `swimclub.py` which has been repackaged as a new Python module called `convert_utils.py`. As you *read* this code, note that the vast majority of it is copied verbatim from `swimclub.py`, with a couple of exceptions (which are noted).

NOTE

This module is included as part of this book's download material.

The converions module

You've seen most of this code before. Here's the `convert_utils.py` code:

```
import hfpy_utils
from records import records

import statistics


CONVERSIONS = {
    "Free": "freestyle",
    "Back": "backstroke",
    "Breast": "breaststroke",
    "Fly": "butterfly",
    "IM": "individual medley",
}

COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

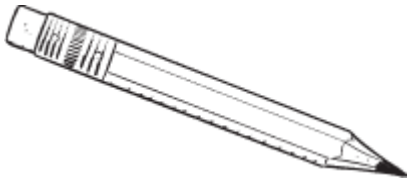

def get_worlds(distance, stroke):
    """Given an event distance and stroke, return the list of the four
    world records at that distance and stroke."""
    return [records[course][f"{distance} {CONVERSIONS[stroke]}"] for course in COURSES]


def perform_conversions(times):
    """Given a list of swim times as strings, return the average time as a string,
    as well as the reversed list of times and a list of scaled swim times as
    numbers (floats)."""
    converts = []
    for t in times:
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = f"{(average / 100):.2f}".split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = f"{minutes}:{seconds}.{hundredths}"
    converts.reverse()
    times.reverse()
    from_max = max(converts)
    scaled = [hfpy_utils.convert2range(n, 0, from_max, 0, 350) for n in converts]
    return average_str, times, scaled
```

This module introduces two new functions. This one condenses the world record lookup code into a single line of code (thanks to the use of a list comprehension).

This is the only new line of code, which is another example of using a list comprehension to perform a transformation of existing data. This comprehension scales the values in the "converts" list.

SHARPEN YOUR PENCIL



Now that the `chart.html` template exists, as well as the `convert_utils.py` module, let's complete the code for `app.py`. Here is the first half of the updated `show_bar_chart` function which determines the selected event's details, then grabs the recorded swim times for the identified swimmer (on the date in question) from you database:

The "event" data is retrieved from the submitted HTML form's data.

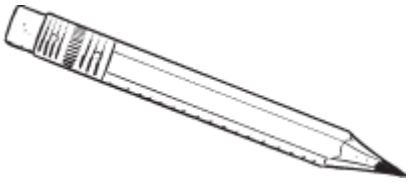
```
@app.post("/showbarchart")
def show_bar_chart():
    distance, stroke = request.form["event"].split(" ")
    data = data_utils.get_swimmers_times(
        session["swimmer"],
        session["age"],
        distance,
        stroke,
        session["chosen_date"],
    )
    times = [time[0] for time in data]
```

The five argument values to the "get_swimmers_times" function are shown over multiple lines to make this call easier to read.

Your job is to provide the rest of the code for this function which needs to perform the necessary conversions on the `times` list, work out the world record times to use, create an appropriate page title, then terminate the function with a call which returns a rendered tem-

plate of the `chart.html` Jinja2 template. Write in the code you created here:

SHARPEN YOUR PENCIL SOLUTION



Now that the `chart.html` template exists, as well as the `convert_utils.py` module, it was time to complete the code for `app.py`. You were given the first half of the updated `show_bar_chart` function which determines the selected event's details, then grabs the recorded swim times for the identified swimmer on the date in question:

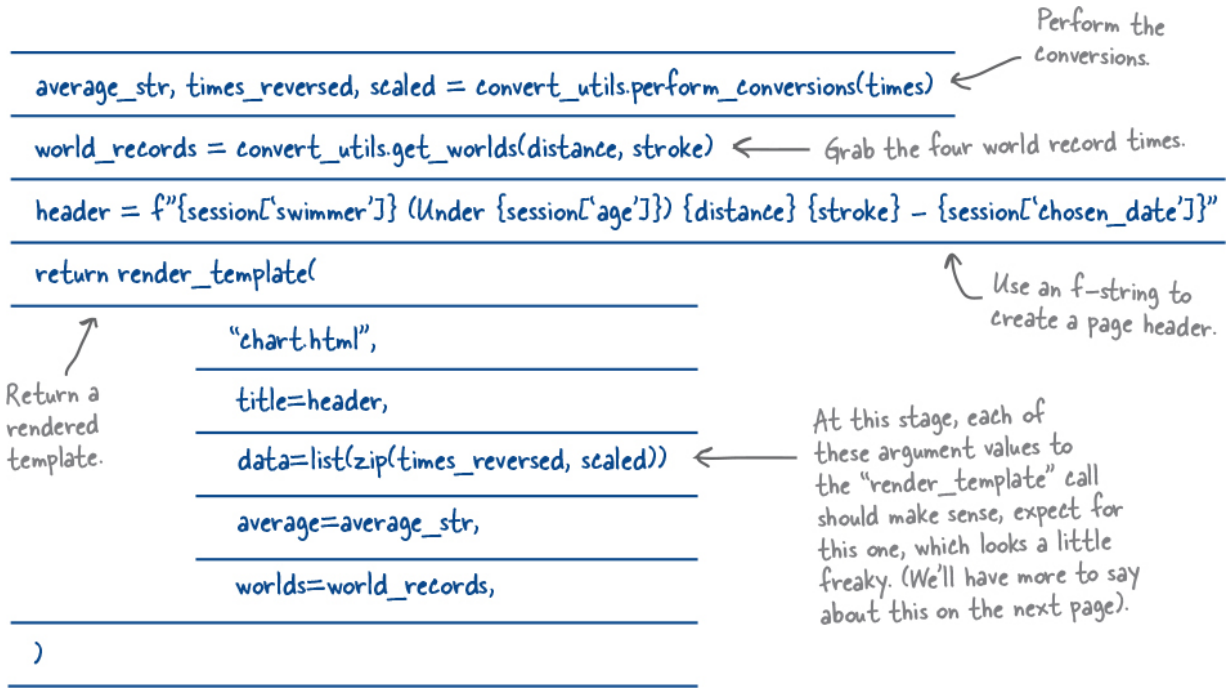
```
@app.post("/showbarchart")
def show_bar_chart():
    distance, stroke = request.form["event"].split(" ")
    data = data_utils.get_swimmers_times(
        session["swimmer"],
        session["age"],
        distance,
        stroke,
        session["chosen_date"],
    )
    times = [time[0] for time in data]
```

The "event" data is retrieved from the submitted HTML form's data.

The five argument values to the "get_swimmers_times" function are shown over multiple lines to make this call easier to read.

Your job was to provide the rest of the code for this function which needs to perform the necessary conversions on the `times` list, work out the world record times to use, create an appropriate page title, then end the function with a call which returns a rendered template of

the `chart.html` Jinja2 template. Here is our code. How does your compare?



list zip… what?!?

That call to **list** then **zip** against the `times_reversed` and `scaled` lists likely has you scratching your head. You've seen this technique before… way back when you created your `records.py` module. Despite this, let's look at a small Jupyter notebook, called `TheZipBIF.ipynb`, which shows what's going on here in a bit more detail:

NOTE

Note: Head First “zipper” art to go here…

Two list of value are defined and assigned to named variables.

```
times = ['39.42', '36.13', '37.66', '39.07']
```

```
scaled = [346.89, 334.37, 320.79, 350.0]
```

```
zip(times, scaled)
```

```
<zip at 0x1055bdac0>
```

Things look a little strange when “zip” is called on the two lists. but is a little less strange when the output from “zip” is turned into a list.

```
list(zip(times, scaled))
```

```
[('39.42', 346.89), ('36.13', 334.37), ('37.66', 320.79), ('39.07', 350.0)]
```

```
for row in list(zip(times, scaled)):
    print(row)
```

```
('39.42', 346.89)
('36.13', 334.37)
('37.66', 320.79)
('39.07', 350.0)
```

Displaying the data produced by the two zipped lists confirms what’s going on here: the two lists are zipped together with one item-at-a-time taken from each of the lists. Note how each two value pairing is returned from “zip” as a tuple.

TEST DRIVE

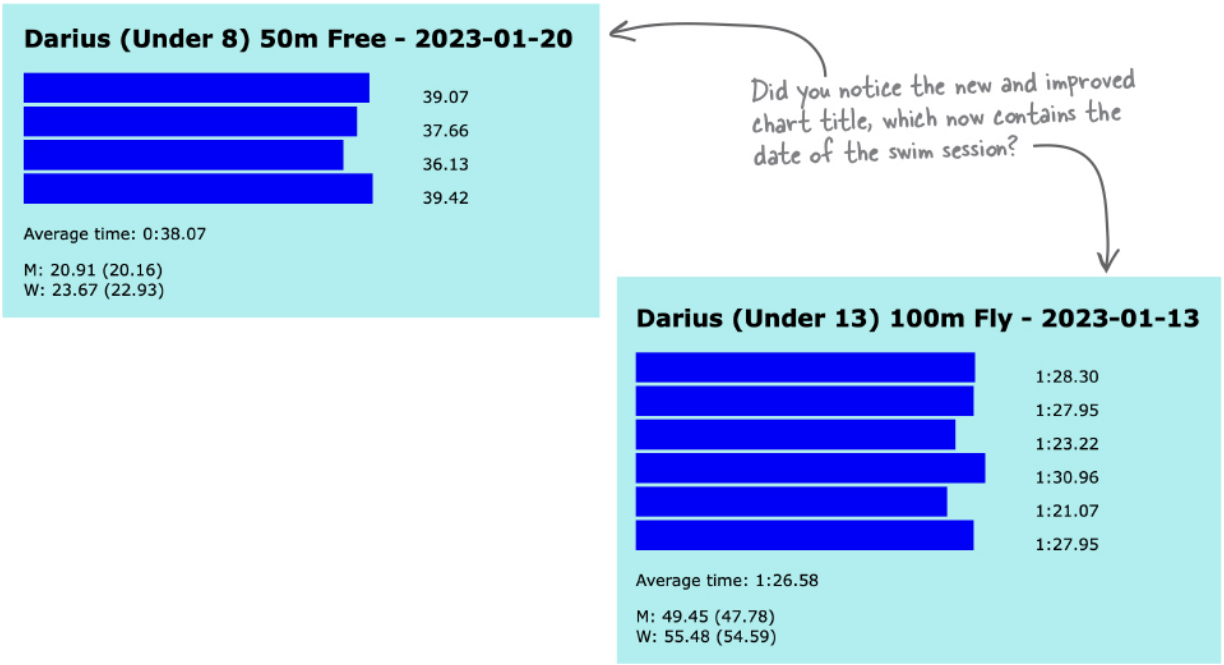


Here's the code for your updated `show_bar_chart` function which, once added to `app.py`, completes the edits for Task #4:

```
@app.post("/showbarchart")
def show_bar_chart():
    distance, stroke = request.form["event"].spli
    data = data_utils.get_swimmers_times(
        session["swimmer"],
        session["age"],
        distance,
        stroke,
        session["chosen_date"],
    )
    times = [time[0] for time in data]
    average_str, times_reversed, scaled =  conver
    world_records = convert_utils.get_worlds(dist
    header = f"{session['swimmer']} (Under {sessi
    return render_template(
        "chart.html",
        title=header,
        data=list(zip(times_reversed, scaled)),
```

```
            average=average_str,
            worlds=world_records,
        )
```

If you select the 50m Free option for the younger Darius, you should see this bar chart:



The chart for the older Darius swimming the 100m Fly is exactly as it was when first created earlier in this book, so all appears to be working well:

Your database integrations are complete!

This is great work!

It's been a lot of work to get here, but your webapp can now work with any amount of data generated by any amount of swim sessions. You're done for now, and it's time for your final tick-mark:

1️⃣ Decide on a structure for your data, then create your database tables

You need to decide how the data in your database is going to be arranged. Once you have decided on this, you can create the necessary database and tables using Python and SQL. ✔️

2️⃣ Add your data values to your database tables

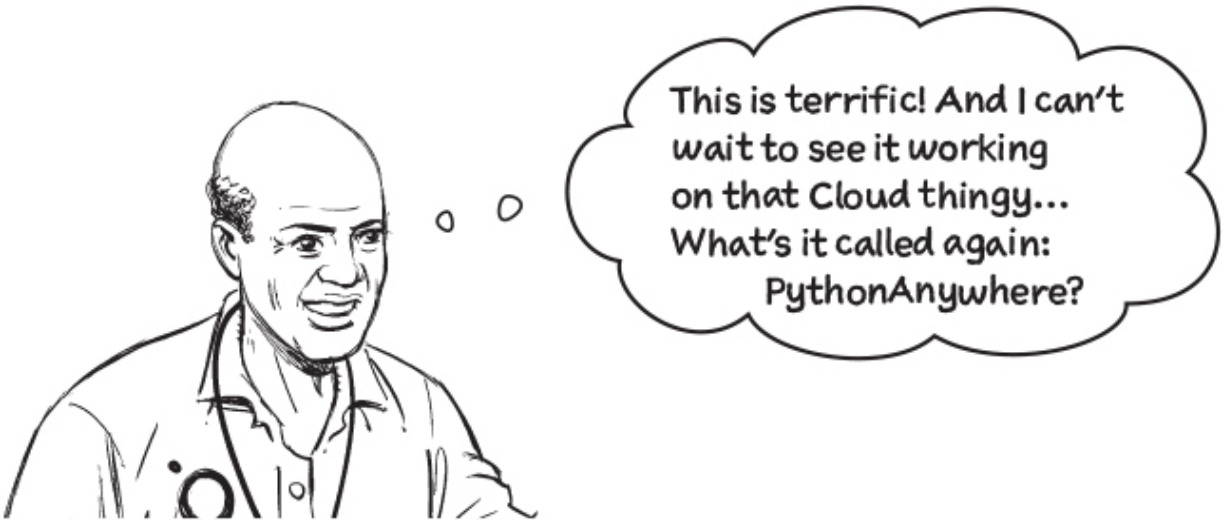
The Coach's system uses the data in the `swimdata` folder. You'll need to take the data from this folder and add it to the appropriate database tables. As in Task #1, Python and SQL are your go-to technologies here.✔️

3️⃣ Extract the data you need from your database tables

At the moment, your webapp's code dips in and out of the `swimdata` folder as needed to grab the data required to do its thing. With Task #2 complete, you can write Python and SQL to grab the data your webapp needs from your database tables instead.✔

4 Adjust your existing webapp code to use the data in your database tables

With Task #3 done, you'll need to change your current webapp's code to grab its data from your database as opposed to from the `swimdata` folder.✔



Although there's a temptation to share this version of your webapp with the Coach *right now*, be aware that this code currently runs on your local computer, not on the Coach's. To make your latest webapp

ready for the Coach, it should really be deployed to *PythonAnywhere*. This is (surprisingly) straightforward, and you'll do it in the next, and final, chapter of this book.

Before then, there's this chapter's bullet-list summary, together with your end-of-chapter crossword to ponder and complete.

CHAPTER REVIEW BULLET POINTS

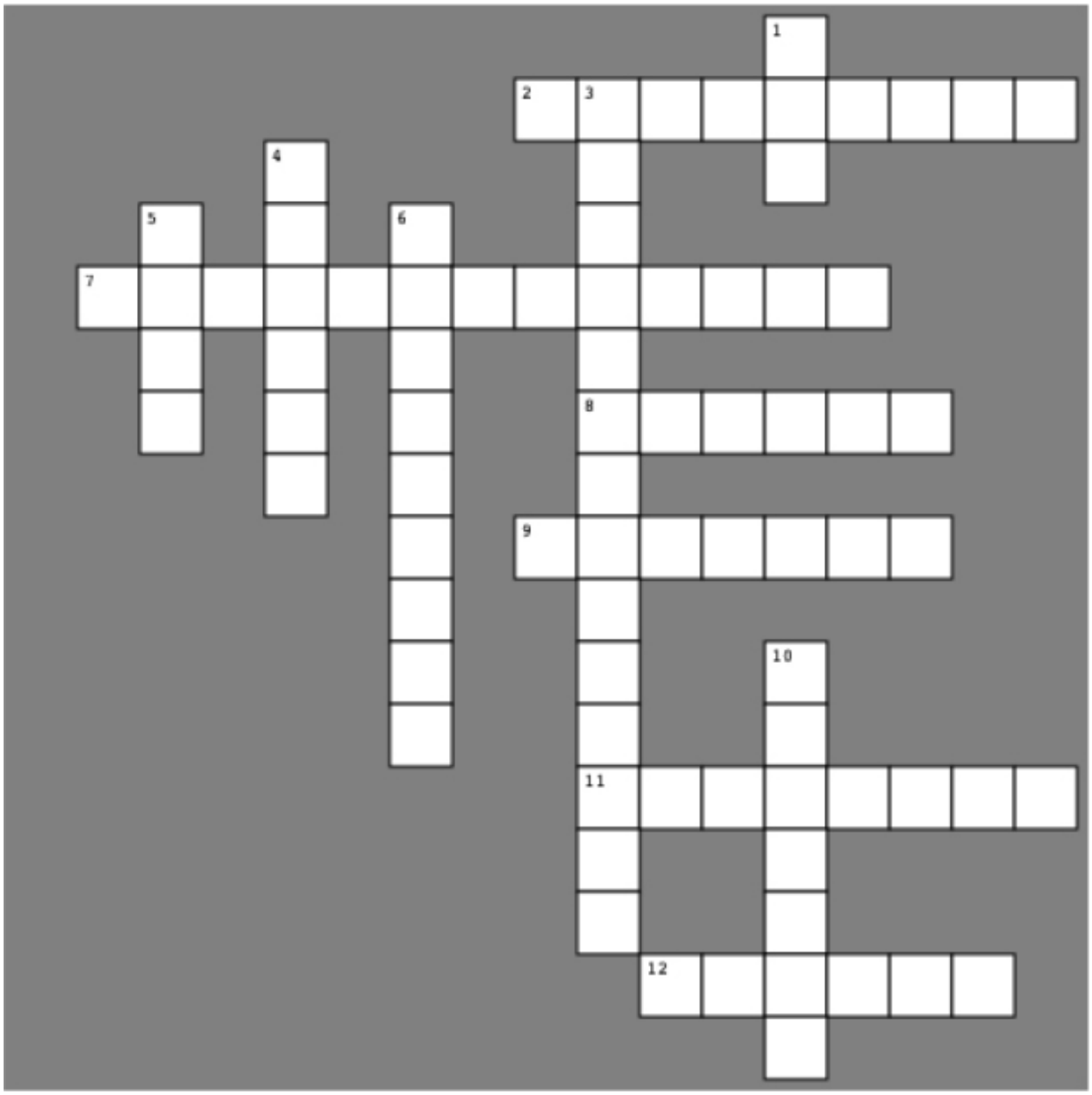
- This chapter was chock-full of code which **integrated** the data in your database engine with the Coach's webapp.
- Despite all of your database integration code, there are only two takeaways from all of this chapter's material we really want you to remember, and it's these two words: **list** and **comprehension**.
- A **list comprehension** lets you rewrite a multi-line **for** loop as a single line. Not only is the resulting single line of code less code (duh!), it's also **faster** than its multi-line equivalent, and list comprehensions can appear almost anywhere in your code (unlike the multi-line loop equivalent). This makes list comprehensions very, very cool.
- Python programmers the world over **love** list comprehensions. Consequently, you'll come across them being used in lot of places in other programmer's programs.
- It is possible to create **set** and **dictionary** comprehensions, too, but you did not need them in this chapter, so they did not feature. However, if you understand how a list comprehension works, you'll have no problem applying what you know to set and dictionary comprehensions.
- There's no such thing as a tuple comprehension (so don't even go there).
- Comprehensions are typically one of two types: **transformational** and **filtering**.

- Transformational comprehensions (also called **mappings**) convert the values in one list to a new list of converted values.
- Filtering comprehensions are easy to spot: they have an **if** condition which must hold in order for a value to be appended to a target list.
- It's perfectly acceptable to have a compehension which is both transformational *and* filtering (although we're not sure what to call a list comprehension which does double-duty like this).
- Did we mention that Python programmers *love* list comprehensions?

The Integration Crossword



You'll find all the answers to the clues in this chapter. The solutions are on the next page.



Across

2. Two new modules of _____ were created in this chapter (one called `data` and the other called `convert`).

7. One of the two words you have to takeaway from this chapter, and it's not "list".

8. An **if** statement can be used to apply this to 7 across.

9. Code is read more than it's _____.

11. In a Jinja2 template, the `extends` command does this.

12. Prior to this chapter, these were referred to as "files".

Down

1. Given two lists, this BIF creates a list of tuple pairs.

3. Convert one list of values into another list of values with a _____.

4. The `fetchall` database method (when is returns data) always returns a list of these.

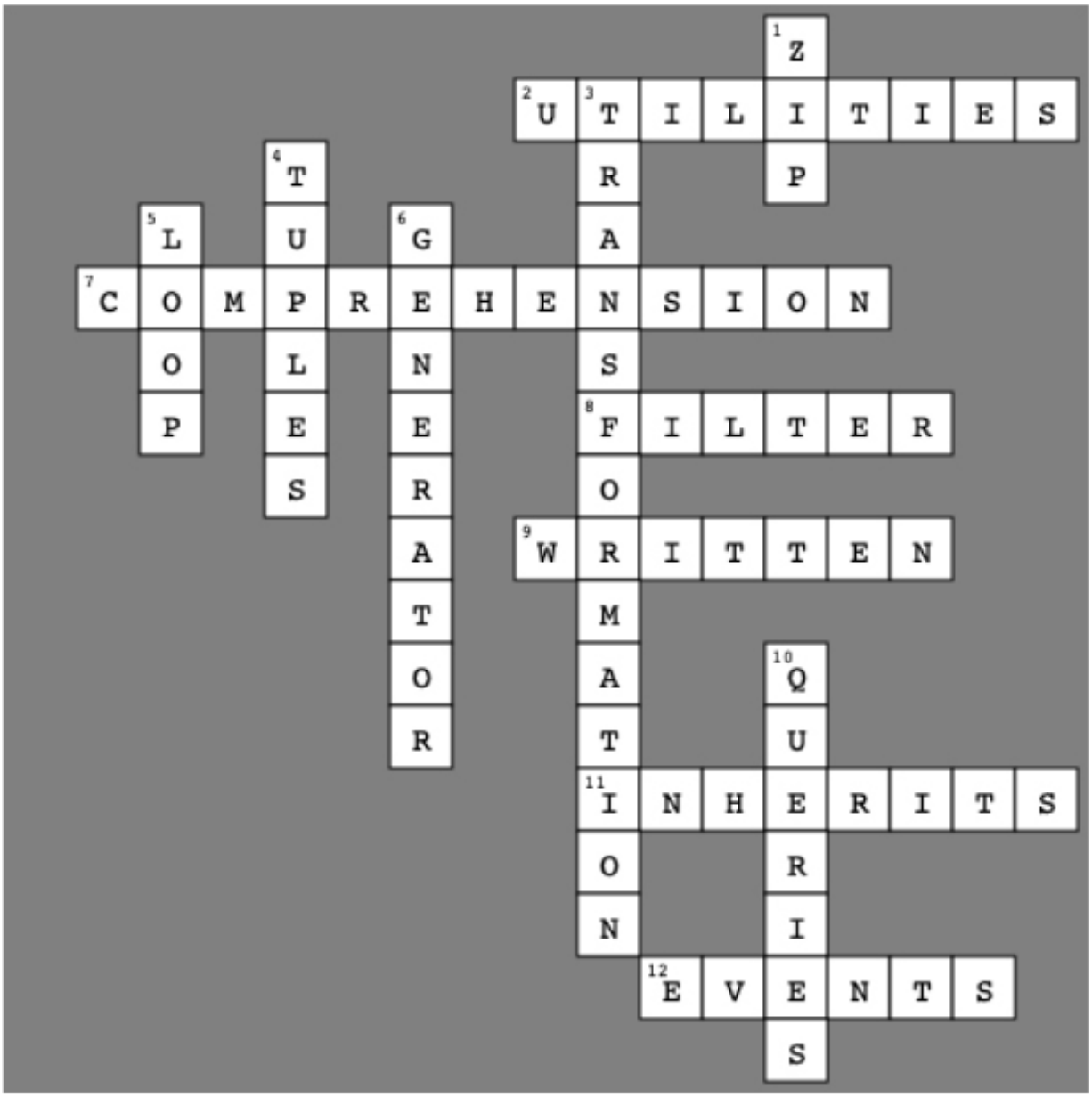
5. 7 across rewrites a multi-line **for** _____ as a single line of code.

6. This chapter's standout technology works with lists, sets, and dictionaries. But try it with a tuple and you'll get one of these.

10. Another name for a bunch of SQL statements.

The Integration Crossword Solution





Across

2. Two new modules of _____ were created in this chapter (one called `data` and the other called `convert`).

7. One of the two words you have to takeaway from this chapter, and it's not "list".

8. An **if** statement can be used to apply this to 7 across.

9. Code is read more than it's _____.

11. In a Jinja2 template, the `extends` command does this.

12. Prior to this chapter, these were referred to as "files".

Down

1. Given two lists, this BIF creates a list of tuple pairs.

3. Convert one list of values into another list of values with a _____.

4. The `fetchall` database method (when is returns data) always returns a list of these.

5. 7 across rewrites a multi-line **for** _____ as a single line of code.

6. This chapter's standout technology works with lists, sets, and dictionaries. But try it with a tuple and you'll get one of these.

10. Another name for a bunch of SQL statements.

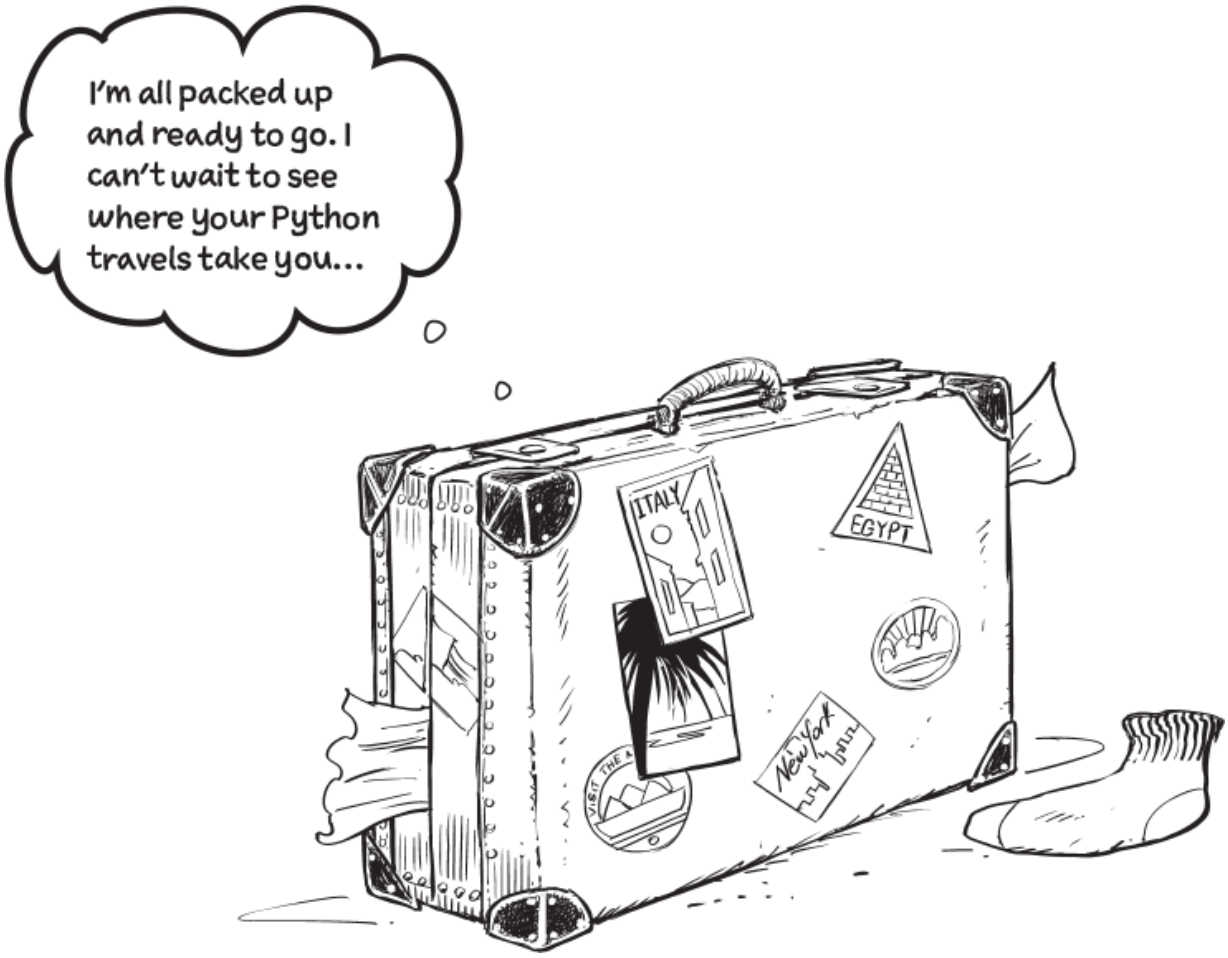
Chapter 13. Deployment Revisited: *The Finishing Touches…*

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 12 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.



You're coming to the end of the start of your Python journey.

In this, the final chapter of this book, you adjust your webapp to use **MariaDB** as your database backend as opposed to SQLite3, then tweak things to deploy the latest version of your webapp on *PythonAnywhere*. In doing so, the Coach gets access to a system which supports **any** number of swimmers attending **any** number of swim sessions. There's not a lot of new Python in the chapter, as you spend most of your time adjusting your existing code to work with

MariaDB and *PythonAnywhere*. Your Python code never exists in **isolation**: it **interacts** with it's environment and the systems it depends on.



We promise: we're almost done.

The webapp code has been updated to support as many training sessions as the Coach can throw at it. Once a training session is selected, the system adapts to only show the data from that session. This is *exactly* what the Coach wants.

All that remains is to replace the webapp currently running on *PythonAnywhere* with your new code. But, it is a matter of simply copying over your existing cloud code?



Cubicle Conversation

Sam: This database integration work is great, but I'm a little concerned about using SQLite3 on *PythonAnywhere*.

Alex: How come? Doesn't SQLite3 work on the cloud?

Sam: It does, yes, but is not really designed to be deployed there.

Mara: I heard SQLite3's more suited to applications that support a single user, as opposed to something that's designed to have many users interacting with it at the same time.

Alex: Why is that an issue?

Mara: It may not be, but you never know in which way your webapp will grow. It might be better to make the jump to a multi-user database management system *now*, before things get too complex.

Sam: Agreed.

Alex: OK… I guess. But, which database management system should we move to?

Mara: As our existing code uses `DBcm`, we should stick to either MySQL or MariaDB, which `DBcm` supports *in addition to* SQLite3. If we continue to use `DBcm` our code changes should be minimal.

Alex: So which do we go with: MySQL or MariaDB?

Sam: In truth, it doesn't really matter which of the two we pick. However, we prefer MariaDB over MySQL.

Alex: How do they differ?

Mara: They don't really, as MariaDB is a *clone* of MySQL.

Alex: But why do both exist if they are the same thing?

Sam: Don't ask.

Mara: That's a long story, which isn't really all that relevant to what we're doing now.

Alex: So the plan is to port our webapp to MariaDB on our machines, then move everything to *PythonAnywhere*?

Mara: Yes.

Alex: And *PythonAnywhere* runs MariaDB, right?

Sam: No, its runs MySQL.

Alex: What??!?

Sam: I know this sounds strange, but don't let it worry you. As you are about to see, MariaDB and MySQL are *essentially* the same thing.

Mara: And it's a personal preference we like to use MariaDB on our computers. The fact that MySQL is used by *PythonAnywhere* won't make a difference.

Alex: I can't imagine anything will go wrong…

Sam: As Mara said earlier, our use of `DBcm` hides a lot of the details from us, so I'm not anticipating any serious issues. There are a few minor SQL incompatibilities that we need to be aware of, but none of these are *showstoppers*, so let's get to it!



We're not throwing away anything.

The last two chapters adapted the webapp to work with data stored in an SQL-based database system, with `DBcm` managing the database

connection.

Although SQLite3 is a great database engine (which, you'll recall, comes with Python), it may not be the best choice when your system needs to support multiple simultaneous connections, i.e., many users. In this case, something designed with that goal in mind (like MariaDB) is often a better choice, and that's why we're moving to it for the Coach's webapp.

NOTE

The Coach may well be our main user, but he's not the only one. We are anticipating the swimmers and their folks want to see some bar charts, too.

THERE ARE NO DUMB QUESTIONS

Q: I already have MySQL installed. Can I use that here?

A: Of course. MariaDB is *our* preference, but this does not mean it has to be *your* preference. Everything we demonstrate with MariaDB in the pages which follow works *unchanged* with MySQ, so you are free to use whichever you want.

Q: Can I install both MariaDB and MySQL on the same computer?

A: Technically, yes. However, we'd caution against doing so. The reason has to do with how MySQL-based database systems communicate with your code (and the rest of the world). Specifically, both MariaDB *and* MySQL use protocol port 3306 by default to communicate, and things get a little hairy when both servers try to grab that port. Although it's possible to configure the server to use another port, our experience is more one of heartache than happiness when you do. Our advice: use either MariaDB or MySQL on your computer, *but not both*.

Q: What about PosgreSQL?

A: No, sorry, not if you want to use `DBcm`. Only MySQL/MariaDB and SQLite3 are supported at the moment.

Migrating to MariaDB

Before deploying on *PythonAnywhere*, let's get your database-enabled version of the Coach's webapp to work locally with MariaDB. Once working locally, your webapp can be deployed to *PythonAnywhere* running MySQL.

Before continuing: take as long as is needed to find and install the latest version of MariaDB on your computer. We'll wait while you do this…

NOTE

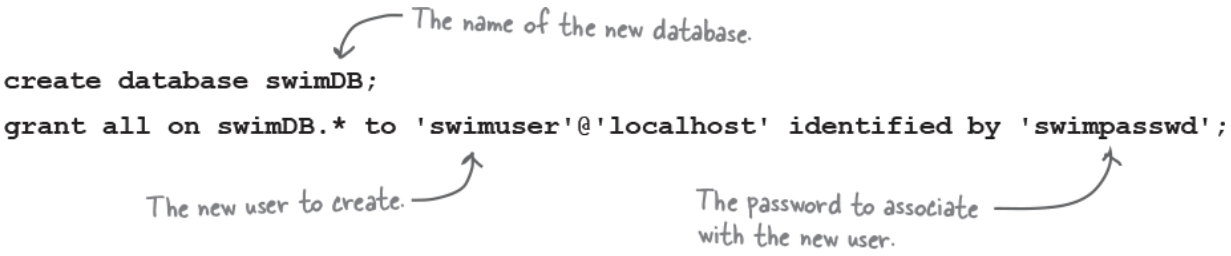
Grab the server from https://mariadb.org/ and follow the installation instructions for your operating system.

Configuring MariaDB for the Coach's webapp

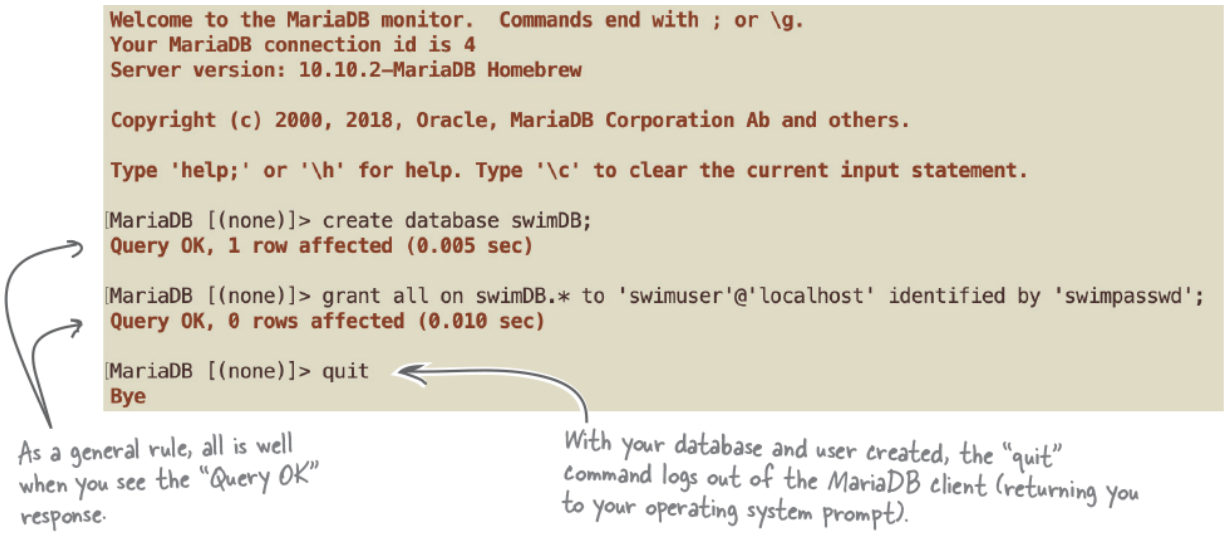
Once MariaDB is installed, log into the server from the command-line as MariaDB's admin user. This is an operating system-dependent activity, so note that the admin user may be known as *superuser* or *root* (depending on your chosen OS).

As soon as you're logged in as admin, use the following commands to create a new database called `swimDB`, as well as a user called

`swimuser` to manage access to it:



Here's what appeared on our screen during the above interaction:



Moving the Coach's data to MariaDB

With your database ready, and your database user created, your next step is to take the data currently residing in your SQLite3 database and move it lock stock and barrel into MariaDB.

There are two parts to this task: (1) *reusing* your table definitions and (2) *copying* your existing data from each of your exiting tables (in SQLite3) into your new tables (in MariaDB).

RELAX



Don't freak out: this sounds like a lot of work, but isn't.

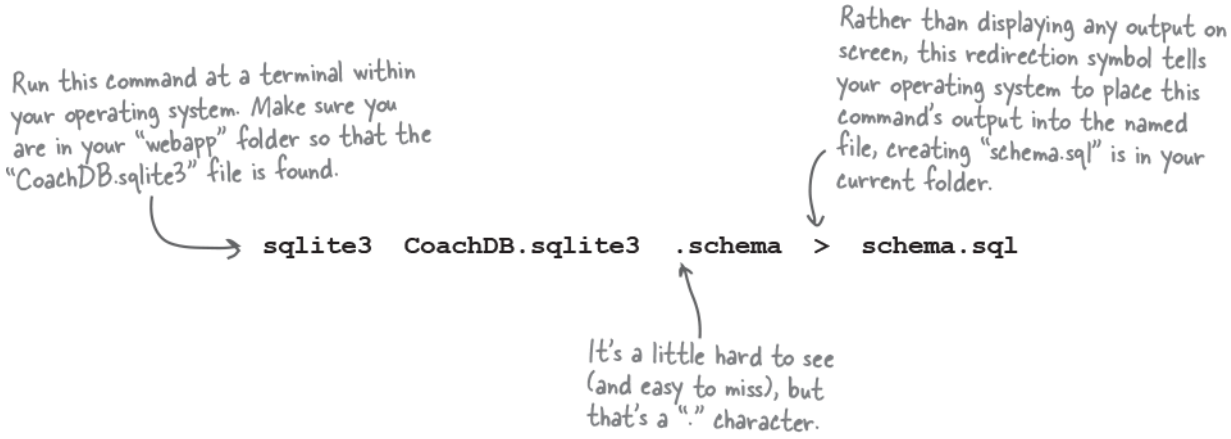
All the work you did getting the Coach's webapp to work with SQLite3 is about to be *reused* with MariaDB (with only minor edits).

Reusing your tables, 1 of 2

You may well recall (from a couple of chapters back) creating your SQLite3 tables using custom Python code in your `CreateDatabaseTables.ipynb` notebook. Although it's a reasonable idea to reuse that code with MariaDB, you're not going to do

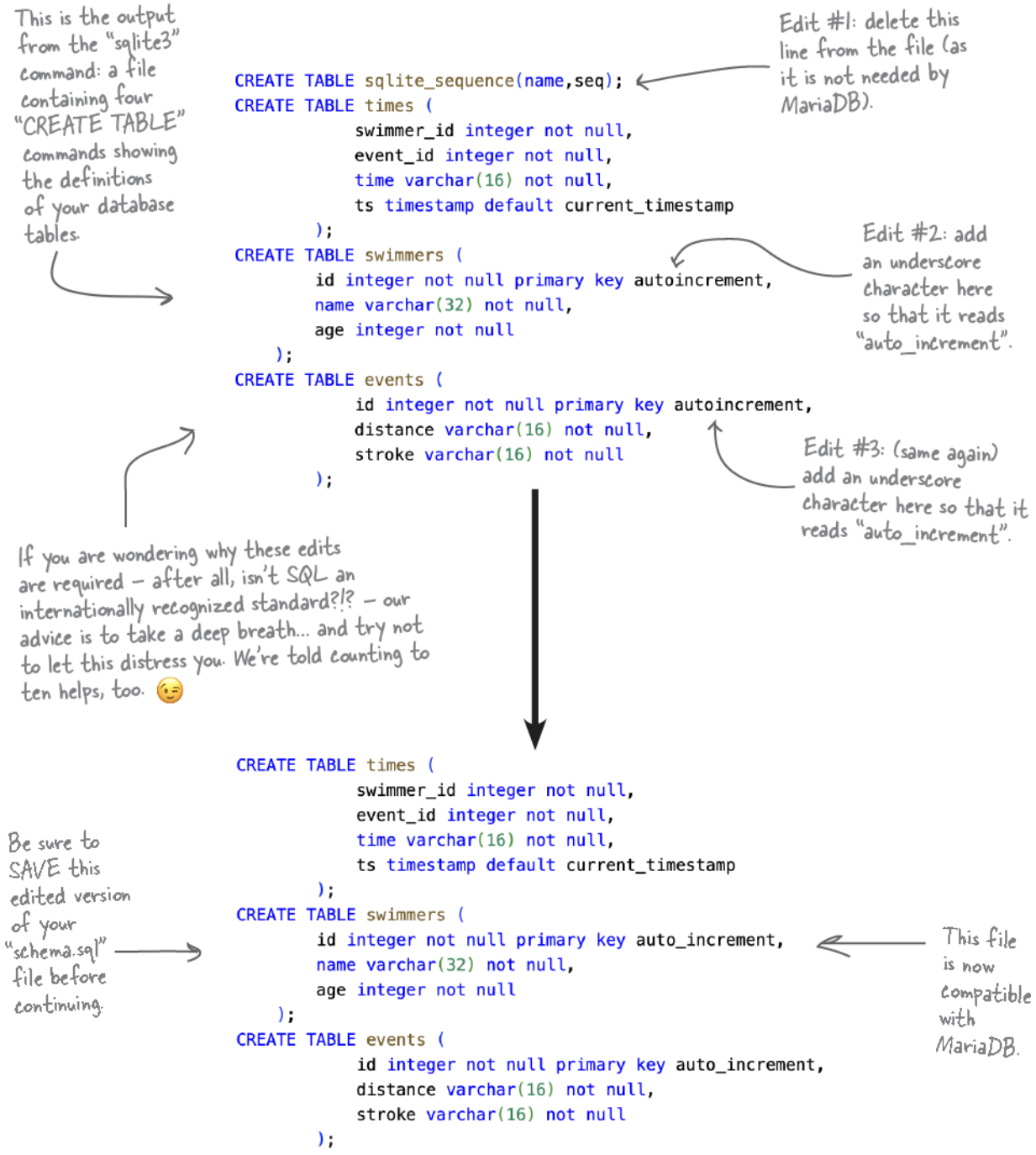
so here. Instead, let's ask SQLite3 to fess-up the table definitions you need, so that you can then "feed" them to MariaDB.

SQLite3 comes bundled with Python, so the `sqlite3` executable is installed on your system when you first installed Python. You can use it to export the definitions of the three tables in your SQLite3 database with this command:



The upshot of running this command is that `schema.sql` comes into existence in your webapp folder. Use VS Code to open `schema.sql`, the join us at the top of the next page to look at what you've got (and make a few minor changes).

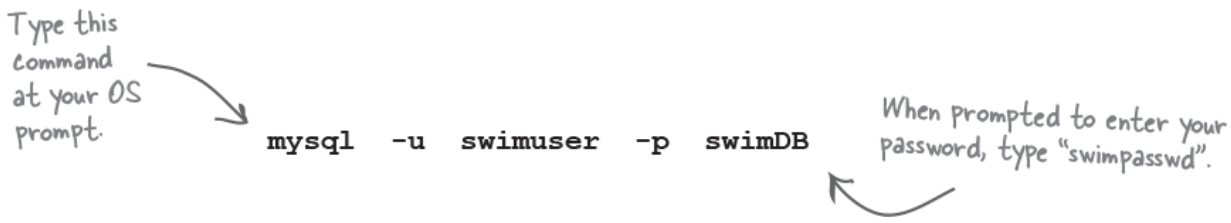
Apply three edit to "schema.sql"



Reusing your tables, 2 of 2

With the three edits applied to `schema.sql` and *saved*, let's use the `schema.sql` file to create the necessary tables within your `swimDB` database running on MariaDB.

Staying within your operating system's terminal (within your `webapp` folder), log into the MariaDB client using the username and password set earlier:



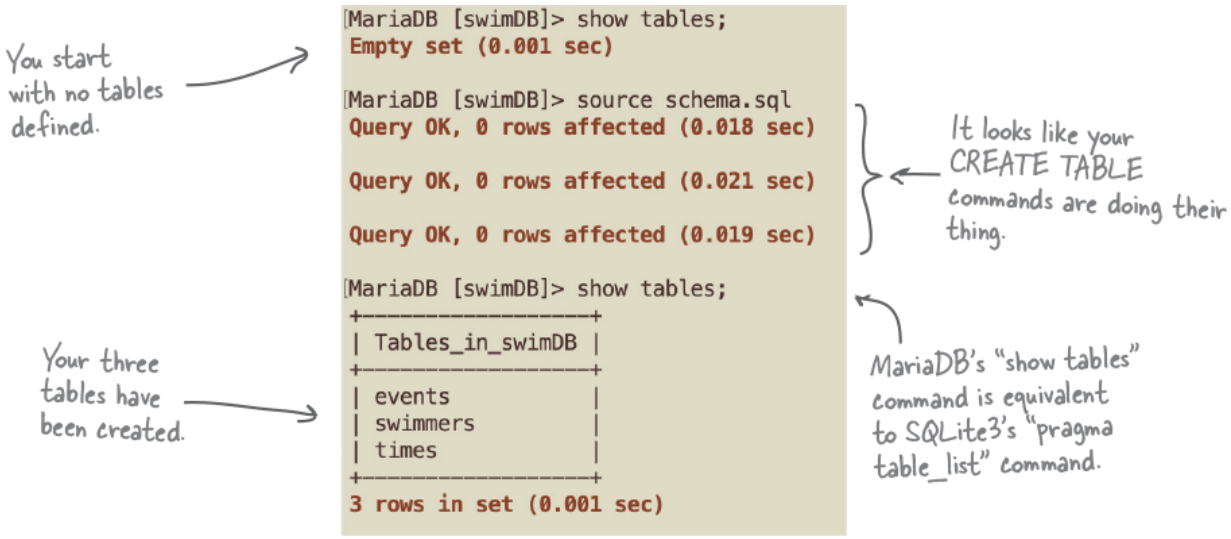
Assuming you've entered your password correctly, you'll be brought back to the MariaDB prompt, where you're going to issue three commands.

The first command, `show tables`, displays a list of tables defined in your database. As you've yet to create any in MariaDB, this list is initially empty.

The second command, `source schema.sql`, reads and runs the SQL queries contained within the `schema.sql` file. As you have three `CREATE TABLE` commands defined in that file, you should expect to see three *Query OK* messages on screen as each of your tables are created.

The third (and final) command, `show tables`, repeats the first command to confirm your three tables have been successfully created:

> **MariaDB is a MySQL clone, so all the tools and commands which work with MySQL also work with MariaDB.**



Let's check your tables are defined correctly

Recall from your work with SQLite3 that you used the `pragma table_info` command to ask your database engine to display details of each of your database tables. MariaDB's `describe` command does the same thing, as shown below:

Looking good.

```
MariaDB [swimDB]> describe swimmers;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(32) | NO   |     | NULL    |                |
| age   | int(11)     | NO   |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.007 sec)
```

Also looking good.

```
MariaDB [swimDB]> describe events;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | int(11)     | NO   | PRI | NULL    | auto_increment |
| distance | varchar(16) | NO   |     | NULL    |                |
| stroke   | varchar(16) | NO   |     | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
3 rows in set (0.005 sec)
```

That's three for three. Your three tables now exist within your MariaDB database.

```
MariaDB [swimDB]> describe times;
+------------+-------------+------+-----+---------------------+-------+
| Field      | Type        | Null | Key | Default             | Extra |
+------------+-------------+------+-----+---------------------+-------+
| swimmer_id | int(11)     | NO   |     | NULL                |       |
| event_id   | int(11)     | NO   |     | NULL                |       |
| time       | varchar(16) | NO   |     | NULL                |       |
| ts         | timestamp   | YES  |     | current_timestamp() |       |
+------------+-------------+------+-----+---------------------+-------+
4 rows in set (0.005 sec)
```

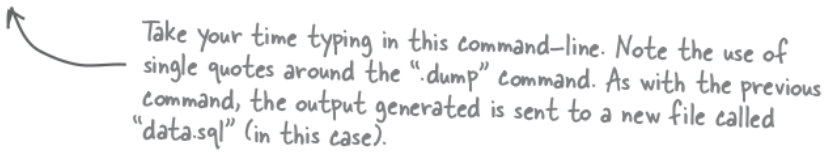
With your tables ready, your next task is to take a copy of the data from your SQLite3 database tables and copy it into your MariaDB database tables. As with your table definitions, the command-line `sqlite3` executable is your go-to tool here.

Copying your existing data to MariaDB

You are likely looking at your MariaDB command prompt in your operating system's terminal window. Leave that window untouched for now, then open a second terminal, again within your `webapp` folder. Let's use your second terminal to generate a file containing the data in each of your existing SQLite3 database tables.

It's possible to do this with a single `sqlite3` command-line:

```
sqlite3 CoachDB.sqlite3 '.dump swimmers events times --data-only' > data.sql
```



With the `data.sql` file created, return to your MariaDB prompt (in your other terminal) and issue a second `source` command, `source data.sql`, to execute the SQL queries contained within your `data.sql` file (which has the effect of importing all your data):

As "data.sql" contains lots of SQL queries, those "Query OK" messages quickly scroll off your screen.

```
MariaDB [swimDB]> source data.sql;
Query OK, 1 row affected (0.002 sec)

Query OK, 1 row affected (0.000 sec)

Query OK, 1 row affected (0.000 sec)

Query OK, 1 row affected (0.000 sec)

Query OK, 1 row affected (0.000 sec)

Query OK, 1 row affected (0.000 sec)
```

There are lots more messages displayed here...

Use these three SQL queries to quickly check how many rows of data were added to each of your tables by that "source" command.

```
MariaDB [swimDB]> select count(*) from swimmers;
+----------+
| count(*) |
+----------+
|       23 |
+----------+
1 row in set (0.003 sec)

MariaDB [swimDB]> select count(*) from events;
+----------+
| count(*) |
+----------+
|       14 |
+----------+
1 row in set (0.001 sec)

MariaDB [swimDB]> select count(*) from times;
+----------+
| count(*) |
+----------+
|      467 |
+----------+
1 row in set (0.001 sec)
```

These row count totals are all looking good.


The data on it's own is not enough. The query code needs to be tweaked to run with MariaDB instead of SQLite3, right?

Yes, but the changes are minor.

Recall you've already had to adjust your SQL `CREATE TABLE` commands, which worked perfectly with SQLite3 but needed *minor* adjustments when ported to MariaDB. Similar *minor* incompatibilities also present when it comes to how placeholders work with MariaDB. Whereas Python's SQLite3 driver uses `?` as the placeholder character, Python's MariaDB driver uses `%s` instead.

Fell free to roll your eyes and let out a long, exasperated sigh…

Granted, it's not a big edit, but it's still something you have to worry about and fix. Let's do that now.

Make your queries compatible with MariaDB

Open `queries.py` in VS Code, then replace all your `?` placeholders with `%s`. Save this updated version of your `queries.py` file, after double-checking it contains code equivalent to that shown here:

```
SQL_SESSIONS = """
    select distinct ts from times
"""


SQL_SWIMMERS_BY_SESSION = """
    select distinct swimmers.name, swimmers.age
    from times, swimmers
    where date(times.ts) = %s and
    times.swimmer_id = swimmers.id
    order by name
"""
```



Changing the placeholders isn't a big edit, but – if we're being honest – is a little bit annoying to have to make them. (Don't these database folk ever talk to each other?)

```
SQL_SWIMMERS_EVENTS_BY_SESSION = """
    select distinct events.distance, events.stroke
    from swimmers, events, times
    where times.swimmer_id = swimmers.id and
    times.event_id = events.id and
    (swimmers.name = %s and swimmers.age = %s) and
    date(times.ts) = %s
"""


SQL_CHART_DATA_BY_SWIMMER_EVENT_SESSION = """
    select times.time
    from swimmers, events, times
    where (swimmers.name = %s and swimmers.age = %s) and
    (events.distance = %s and events.stroke = %s) and
    swimmers.id = times.swimmer_id and
    events.id = times.event_id and
    date(times.ts) = %s
"""
```

What was ? is now %s.

And the edits don't end there…

Two other edits are also required, one to `data_utils.py`, and another to `app.py`.

At the moment, `data_utils.py` assigns the name of your SQLite3 database file to the `db_details` variable. When using `DBcm` with MariaDB, the `db_details` variable is assigned a dictionary of connection/credential information. Again, this isn't a big change, but is an important one.

Use VS Code to adjust the top of your `data_utils.py` so that it looks like this:



Another incompatibility has to do with how SQLite3 and MariaDB handle timestamps. Whereas SQLite3 returns timestamps to your Python code as a string, MariaDB returns timestamps as a `datetime` object, so another adjustment is needed.

NOTE

Note: The PSL contains a handy library called “datetime” which knows all about Python dates.

Use VS Code to adjust the top of the `display_swim_sessions` function within `app.py` to look like this:

```
@app.get("/swims")
def display_swim_sessions():
    data = data_utils.get_swim_sessions()
    ## dates = [session[0].split(" ")[0] for session in data]  # SQLite3.
    dates = [session[0].isoformat().split("T")[0] for session in data]  # MySQL/MariaDB.
```

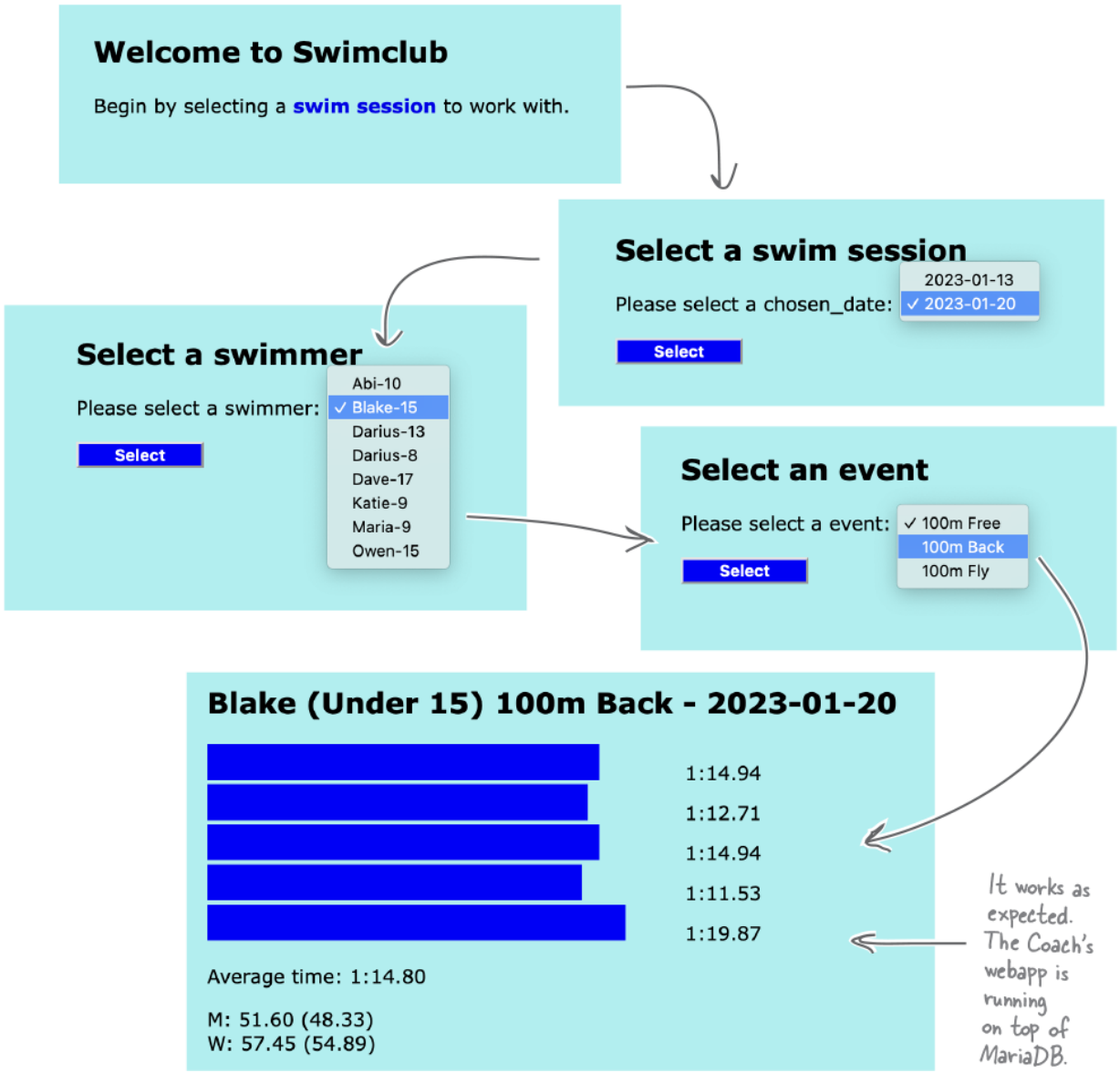
Again, we're commenting-out a line code here. Think of the double-hashes as a reminder to return to this code at some later date and, perhaps, to consider deleting the line of code if it's no longer needed.

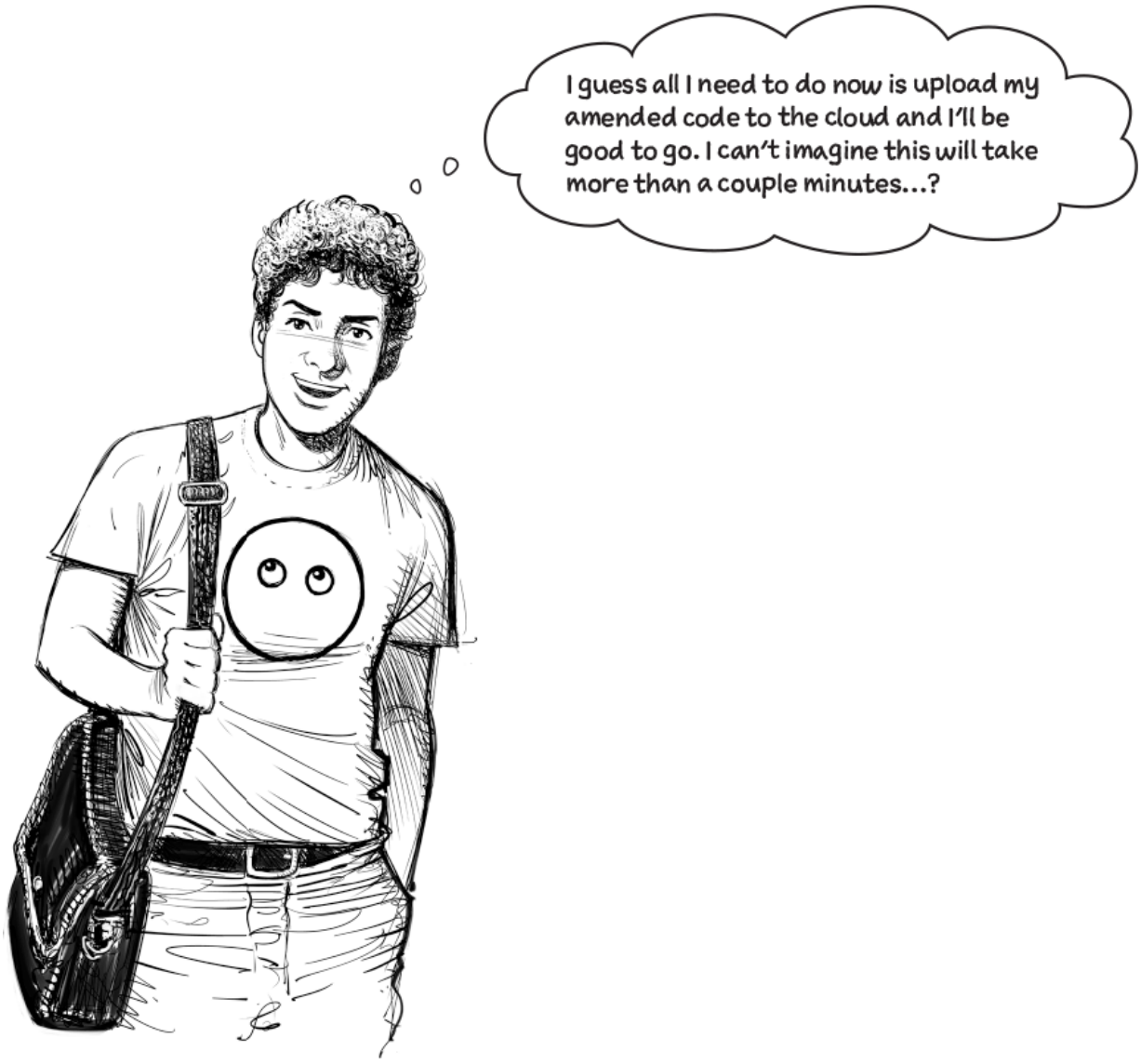
MariaDB returns Python datetimes. This code converts the datetime to a standardized string format (recommended by the ISO), then splits the string on "T" to extract just the date-part of the timestamp.

TEST DRIVE



OK. Take a deep breath…You've installed MariaDB on your computer, created a new database and given a user exclusive rights to your database. You've created the three tables you need within MariaDB, and populated those tables with your existing rows of data from your SQLite3 database. You've also edited (and saved!) three of your webapp's files: `queries.py`, `data_utils.py`, and `app.py`. With your `app.py` code loaded into VS Code, choose **Run**, then **Run Without Debugging** to take your MariaDB-compatible webapp for a spin:


Welcome to Swimclub
Begin by selecting a swim session to work with.
Select a swim session
Please select a chosen_date:
2023-01-13
✓ 2023-01-20
Select
Select a swimmer
Please select a swimmer:
Abi-10
✓ Blake-15
Darius-13
Darius-8
Dave-17
Katie-9
Maria-9
Owen-15
Select
Select an event
Please select a event:
✓ 100m Free
100m Back
100m Fly
Select
Blake (Under 15) 100m Back - 2023-01-20
1:14.94
1:12.71
1:14.94
1:11.53
1:19.87
Average time: 1:14.80
M: 51.60 (48.33)
W: 57.45 (54.89)
It works as expected. The Coach's webapp is running on top of MariaDB.



Yes, a code upload is the first step.

But, there's a little more to this than meets the eye. We are getting closer to being done here, but we'll need more than just the updated code. When it comes to deploying your latest webapp to *PythonAnywhere*, you still need to *create* a database, *define* your tables, and *populate* your tables with your data *on the cloud*.

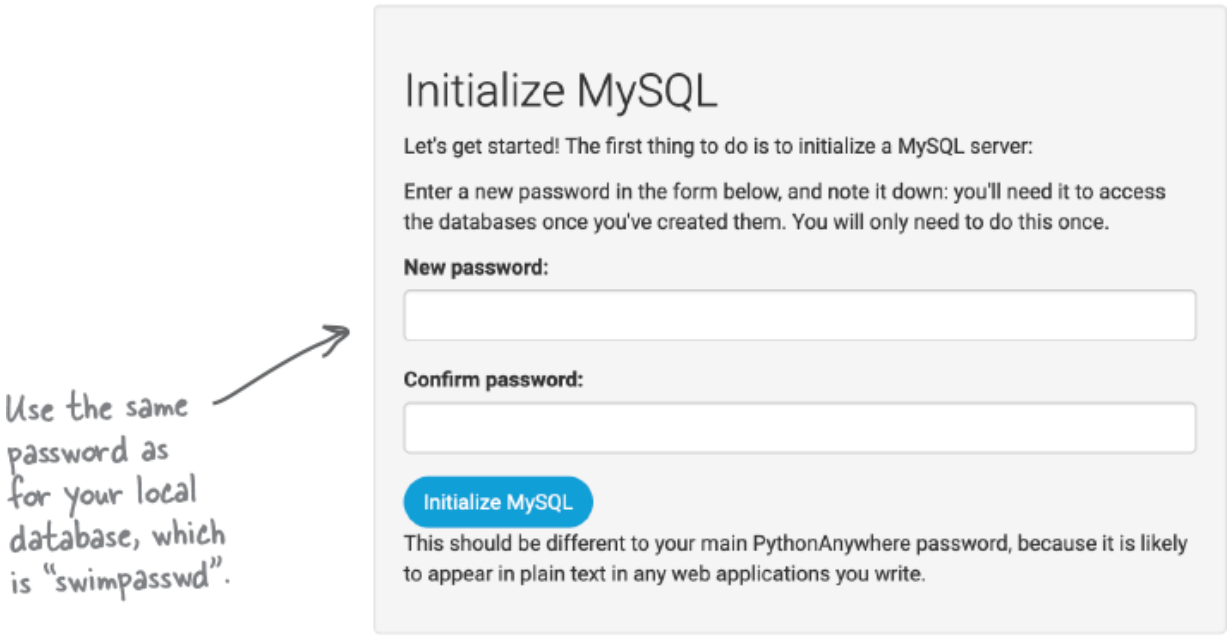
At the moment, all your webapp's data resides in your local MariaDB database (on your computer). That said, and as you are about to discover, the deployment process is straightforward. Let's get to it.

Create a new database on PythonAnywhere

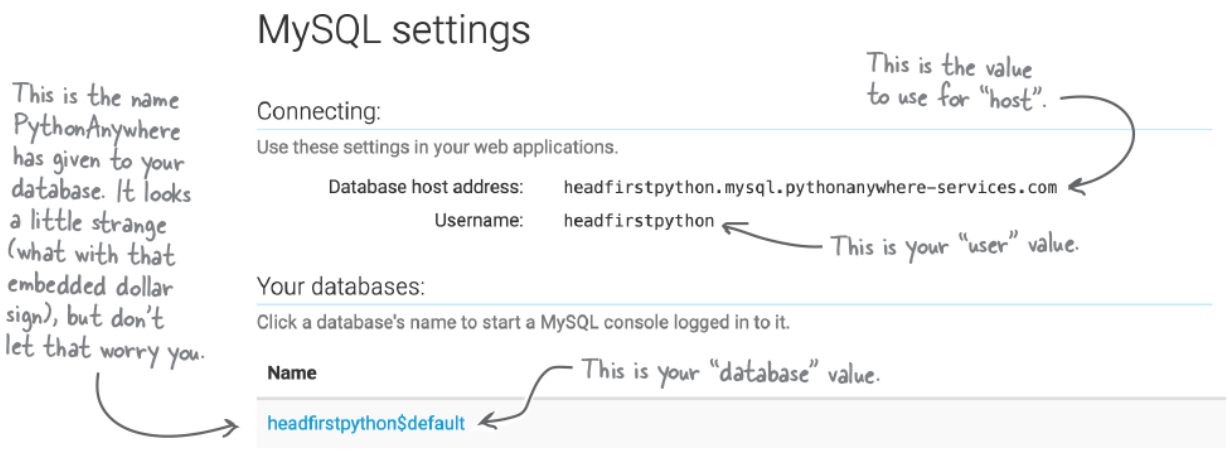
Let's configure your *PythonAnywhere* account to run the MariaDB-enabled version of your webapp.



To get going, log into your *PythonAnywhere* account then click on the **Databases** tab. *PythonAnywhere* uses MySQL (not MariaDB), but that's OK as MariaDB is designed to be compatible with MySQL to a very high degree. Enter the password you want to use with your database into the input boxes displayed on the **Databases** tab:



With your password entered, click on the big blue button. After a moment or two, your browser screen refreshes to confirm your database now exists. You already know the password to use, and you're now provided with the three other pieces of credential information needed to establish a connection to your database from your code:



Adjust your database credentials dictionary

Now that you know the four pieces of *PythonAnywhere* database credential information, you may be tempted to adjust your `db_details` dictionary within your `data_utils`.py code to use these cloud-specific values. However, if you do that, your code won't run on your local machine any more…

You could make a mental note to adjust your code depending on what you're doing, but – trust us – you're going to forget at some point.

As luck would have it, the PSL can help here thanks to its included `platform` module. Here's code which exploits `platform`, allowing you maintain a single copy of your code which selects the credentials to use based on the computing platform it's running on: either your local computer or *PythonAnywhere*.

NOTE

Once again, the standard library comes to our rescue.

Edit “data_utils.py” to support multiple connections

Let’s use VS Code to adjust the top of `data_utils.py` so that it looks like the code shown below, which selects the database credentials after determining the platform your code is running on, either *locally* or on *PythonAnywhere*.



Copying everything to the cloud

You’ve created your database on *PythonAnywhere*, and you’ve adjusted your `data_utils.py` code *one last time* to support the selection of the correct database credentials to use (based on which

platform your code thinks it's running on). You're now ready to copy your code and data to *PythonAnywhere*.

Preparing your code and data for upload

First up: preparing your code. Use your operating system's ZIP utility to compress your `webapp` folder, giving the compressed file the name `webapp-update. zip`.

Now for your data.

You could upload the files your created earlier in this chapter, `schema.sql` and `data.sql`, then use the `source` command on *PythonAnywhere* to run these commands on your cloud-hosted MySQL server. However, the fact you're using MariaDB locally, twinned with the fact that MariaDB and MySQL are designed to be compatible, lets you use a MySQL tool to migrate all the details of a database on one machine to another *using a single command*.

The `mysqldump` command (included with MySQL/MariaDB) lets you create a copy of not just your the data contained in any database, but also its table definitions. The file produced contains *both* the database table definitions *and* the data.

NOTE

A typical use of "mysqldump" is to create backups of running databases.

This command can be used to "dump" everything from your local `swimDB` database into a file called `db.sql`:

```
sudo mysqldump swimdb > db.sql
```

This command should work unchanged if you are using Linux or macOS. Windows users are unlikely to have access to the "sudo" command, so should check the Windows-specific MariaDB documentation to determine how best to run "mysqldump" on Windows.

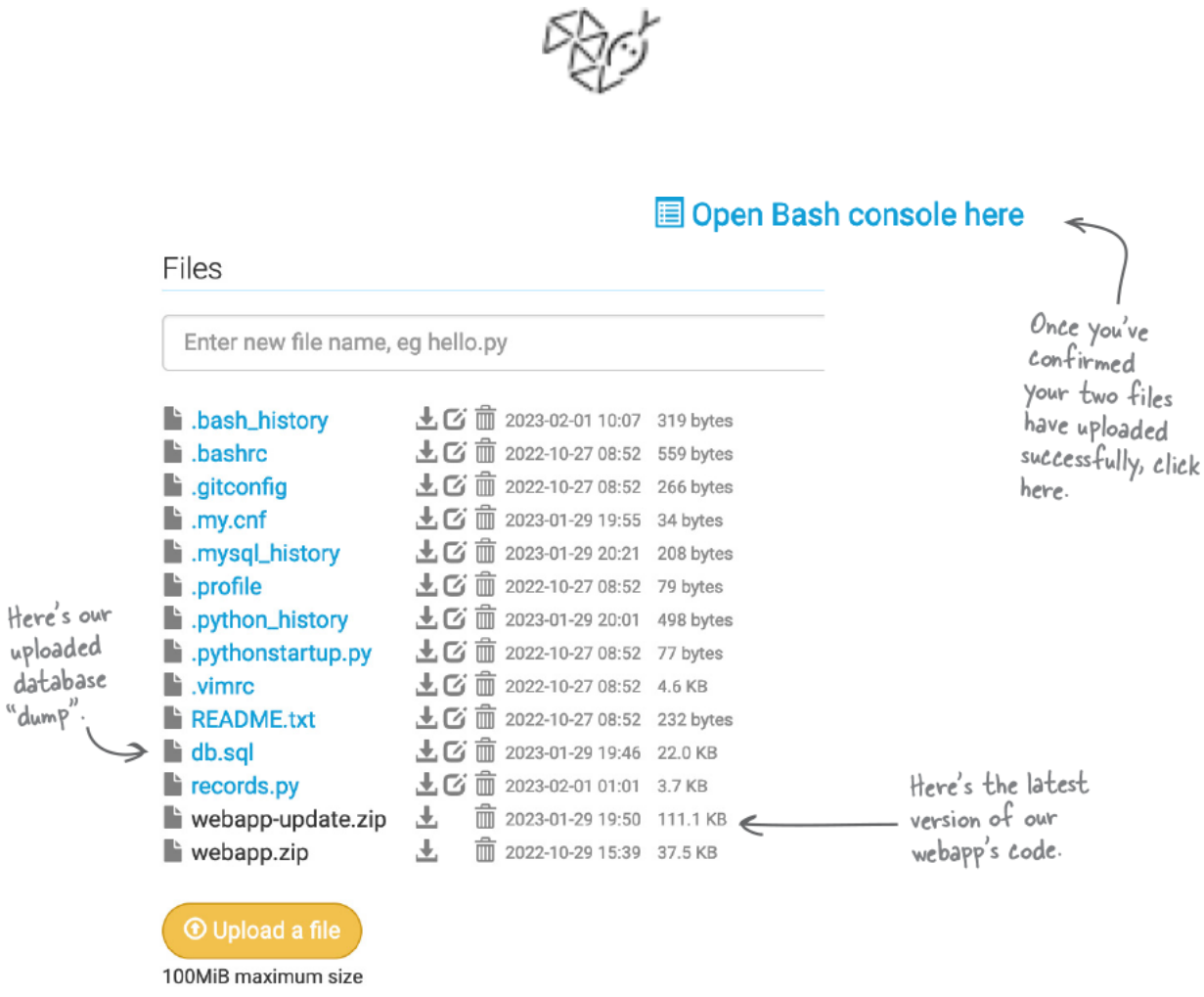
You now have your code in the `webapp-update.zip` file, and all the details of your database in the `db.sql` file. It's time to pop them up on *PythonAnywhere*…



This could not be any easier: simply use the "Upload a file" button on the *PythonAnywhere* **Files** tab to copy your two files to the cloud.

Update your webapp with your latest code

With `webapp-update.zip` and `db.sql` uploaded, click on the "Open Bash console here" link:



Once your Bash console opens, type `unzip webapp-update.zip` to extract your uploaded code:

```
19:52 ~ $ unzip webapp-update.zip
Archive:  webapp-update.zip
  inflating: webapp/update_tables.py
  inflating: __MACOSX/webapp/._update_tables.py
  inflating: webapp/app-pre-database.py
  inflating: __MACOSX/webapp/._app-pre-database.py
  inflating: webapp/db.sql
  inflating: webapp/CreateDatabaseTables.ipynb
  inflating: __MACOSX/webapp/._CreateDatabaseTables.ipynb
  inflating: webapp/data.sql
  inflating: webapp/.DS_Store
  inflating: webapp/ReusedCode.ipynb
  inflating: __MACOSX/webapp/._ReusedCode.ipynb
replace webapp/WebappSupport.ipynb? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

PythonAnywhere's "unzip" command overwrites the files in your existing "webapp" folder. Be sure to enter uppercase "A" when prompted to confirm this is what you mean to do.

Just a few more steps…

You're nearly there.

You've upload your latest webapp code and unzipped it to your *PythonAnywhere* `webapp` folder. However, before you run it, you need to install the `DBcm` module as your webapp code now depends on this module being available. It's not installed on *PythonAnywhere* by default, but can be easily added.



Continuing to work at the Bash console from the last page, ask Python's **pip** command to install `DBcm`:

```
python3 -m pip install DBcm
```

This may well be the first time you've seen this command since this book's Introduction, as you've used the %pip command within your notebooks to do the same thing. The command-line installs a named package from PyPI into your Python environment.

Running this command produces a slew of status messages. If you see a message near the bottom of the output telling you "DBcm" has been successfully installed, then you're good.



Yes, your code's ready…

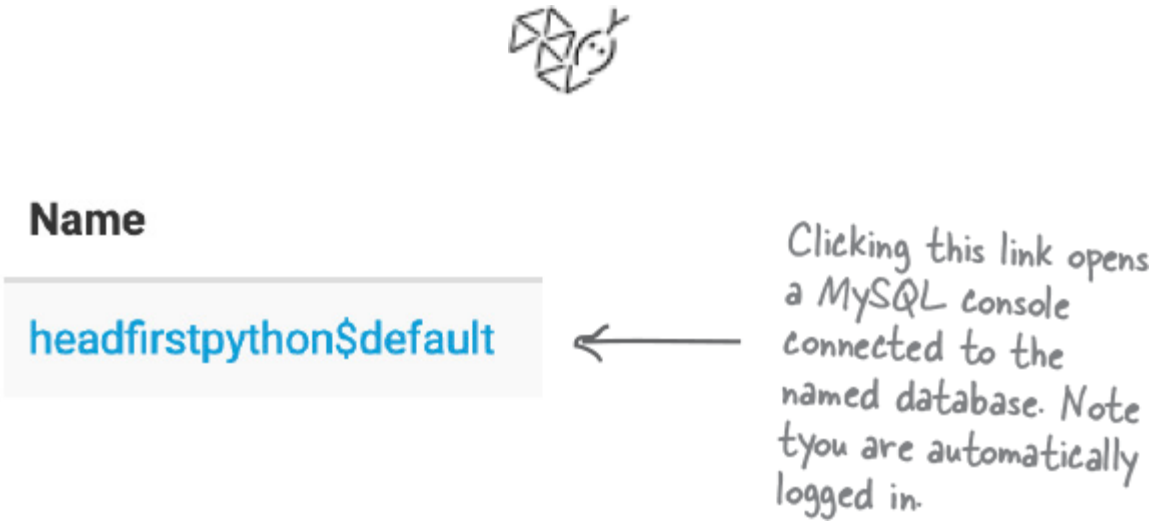
But not your *data*.

You've successfully uploaded your database information to *PythonAnywhere*, but you've yet to suck in all into MySQL.

Let's do that now.

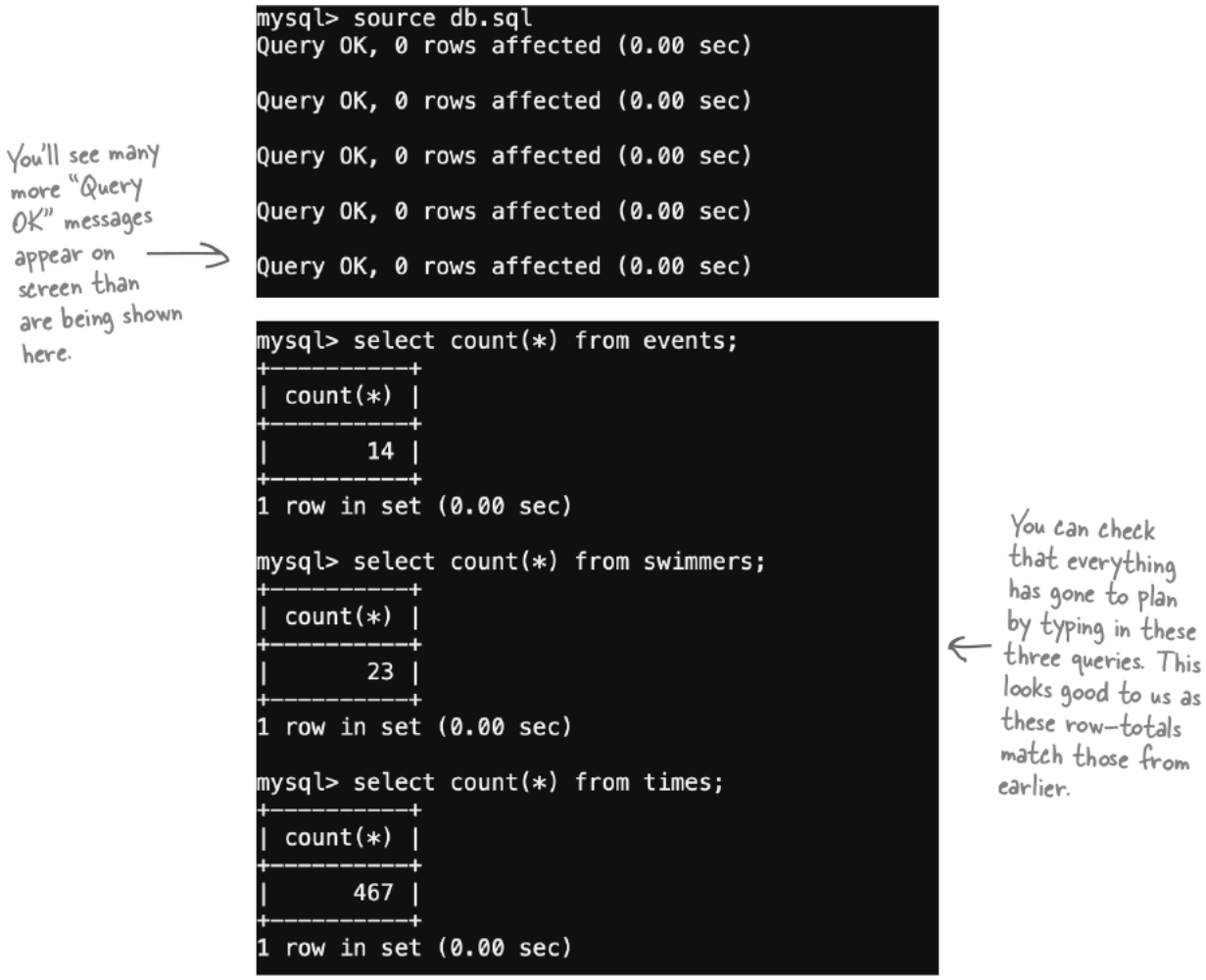
Populate your cloud database with data

Return to your **Databases** tab on *PythonAnywhere*, then click on the blue link associated with your recently created database:



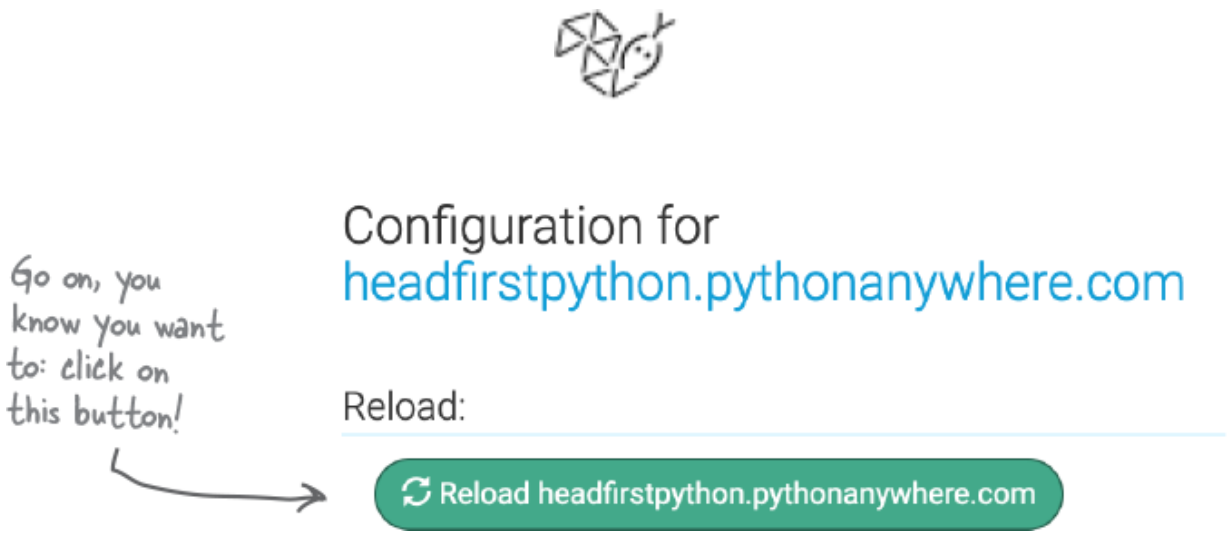
As with your local MariaDB database engine, the `source` command is your friend here. Enter `source db.sql` to run all the SQL com-

mands and queries in that file. When the process completes, your three tables have been created *and* populated with a copy of your local data:



It's time for a PythonAnywhere Test Drive

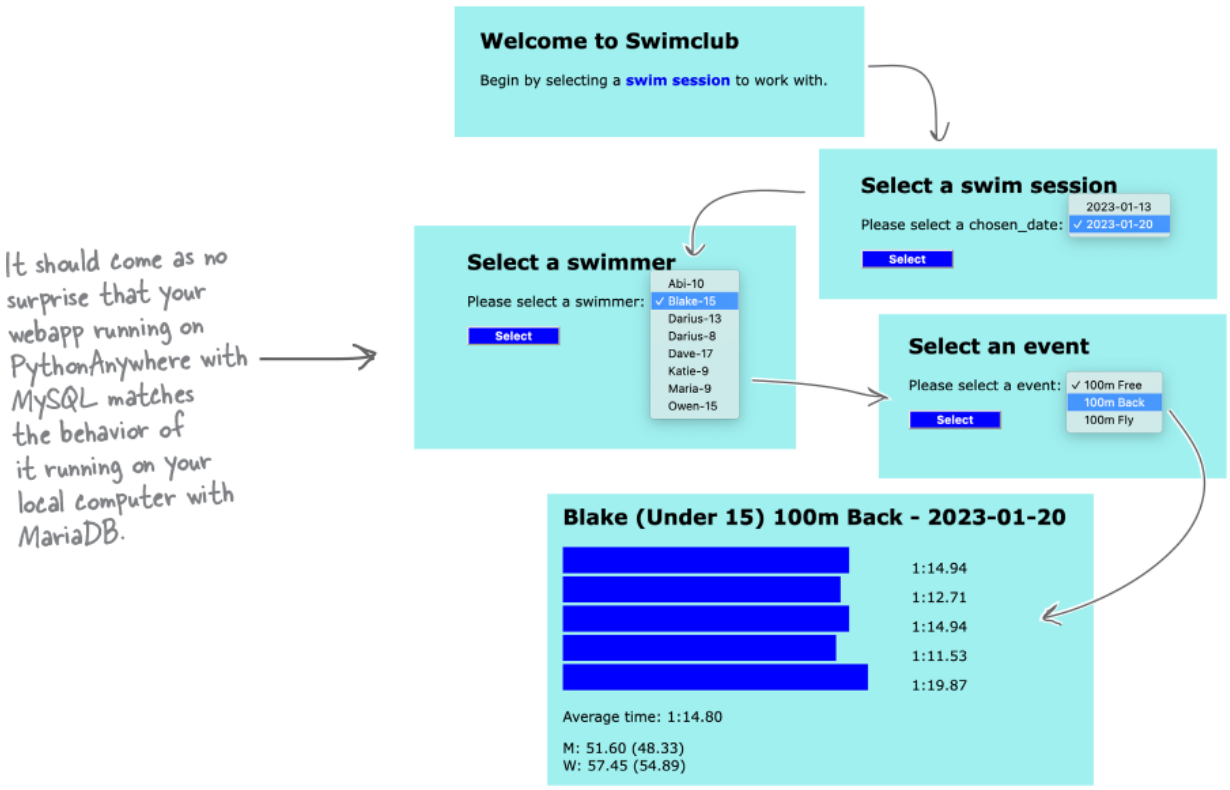
Your updated code is loaded into *PythonAnywhere*, and a copy of your local database is running on *PythonAnywhere*, too. All that's left to do is take this updated version of your webapp for a spin. Before you do, return to the *PythonAnyhere* **Web** tab, then click on the big green button to restart your webapp:



TEST DRIVE



With your webapp reloaded, click in your webapp's blue link to take it for a spin.





That's great news. But… perhaps you haven't been so lucky?

If something went wrong, you may have been presented with a terse error message, something which looks like one of these:



Now: don't panic if you are seeing either of these messages. Flip the page to learn what to do if something like this happens to you.

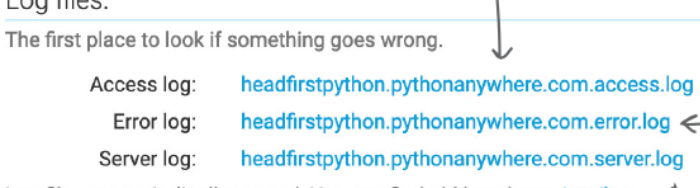
Is something wrong with PythonAnywhere?

Nine times out of ten, when your webapp refuses to run on *PythonAnywhere*, it's something that you've done (or forgotten to do).



Your first port of call should always be to return to the **Web** tab on *PythonAnywhere* and scroll down until you see the section describing your webapp's log files:

The "Access log" contains information on successful interactions with your server.



The "Error log" contains information on requests to your webapp which have resulted in errors (producing those plain-as-Jane error messages from the previous page). Looking at this file should be your first port of call when something breaks.

The "Server log" file contains information on PythonAnywhere's backend technology and, although it may be less relevant to the running of your webapp's code, does often contain messages which might just make you chuckle. If you know the issue can't possibly be with your code, check this log file for issues with PythonAnywhere.

```
During handling of the above exception, another exception occurred:
**NO MATCH**
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 2077, in wsgi_app
    response = self.full_dispatch_request()
  File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1525, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1523, in full_dispatch_request
    rv = self.dispatch_request()
  File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1509, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)
  File "/home/headfirstpython/webapp/app.py", line 23, in display_swim_sessions
    data = data_utils.get_swim_sessions()
  File "/home/headfirstpython/webapp/data_utils.py", line 17, in get_swim_sessions
    with DBcm.UseDatabase(db_details) as db:
  File "/home/headfirstpython/.local/lib/python3.10/site-packages/DBcm.py", line 96, in __enter__
    self.conn = mysql.connector.connect(**self.configuration)
  File "/usr/local/lib/python3.10/site-packages/mysql/connector/__init__.py", line 272, in connect
    return CMySQLConnection(*args, **kwargs)
  File "/usr/local/lib/python3.10/site-packages/mysql/connector/connection_cext.py", line 94, in __init__
    self.connect(**kwargs)
  File "/usr/local/lib/python3.10/site-packages/mysql/connector/abstracts.py", line 1052, in connect
    self._open_connection()
  File "/usr/local/lib/python3.10/site-packages/mysql/connector/connection_cext.py", line 251, in _open_connection
    raise errors.get_mysql_exception(msg=exc.msg, errno=exc.errno,
mysql.connector.errors.DatabaseError: 2003 (HY000): Can't connect to MySQL server on 'localhost:3306' (111)
```

Here's an example entry in the "Error.log" file. Note the very last line, which is a pretty big clue as to what's broken.



Cubicle Conversation

Alex: I don't want to jinx us or anything, but are we done?

Mara: Is any software-based system *ever* really finished?

Sam: I guess there's always more to add. But I think for now, we have a system which does what the Coach needs it to do. Recall that the Coach needed a system which lets him grow his club membership by dramatically reducing the amount of time he spent, in his words, "mucking around with spreadsheets".

Alex: I guess we've come a long way, haven't we?

Sam: Yes, we have. And the code we've created is nicely modularized.

Alex: And it's easy to read.

Mara: Which is a given when you use Python, right?

Sam: If you ask me, we're in pretty good shape. If the Coach comes back to us with any more "feature ideas", we'll be more than ready for them.

Alex: I guess we never really addressed the mechanism required to automatically copy the data from the Coach's *Smart Stopwatch* into the cloud, did we?

Mara: Isn't that why we have those DevOps folks?!?

Sam: Ha! Ha! Yes, indeed. It's always nice to make those sort of things *someone else's problem*…

Alex: While we get stuck into our next project?

Sam: Actually, I was thinking of heading to the beach.

Alex: Do we have time for a break?

Sam: Of course we do!

Mara: You've gone from knowing next to nothing about Python to deploying a cloud-based, database-driven webapp which solves a real-world problem. Along the way you've also learned how Python handles data, how web scraping works, as well as how powerful technologies like pandas can be exploited to help gets things done. And you've been exposed to lots of Python code.

Alex: I guess we've been busy, haven't we?

Sam: All the more reason for some R&R.

Alex: OK, then. I'll just grab my board, then I'm ready to go.

Mara: Because it's surf time!

Sam: Let's hit the waves, folks!

The Coach is a happy chappy!

If your latest code is running fine on the cloud, anyone with access to your system sees your latest updates. This includes the Coach…



Shucks, Coach: we were happy to help.

It always great to work on a real-world problem while learning a new programming language.

And, with that said, this book concludes.

All that remains is one last (and *short*) chapter review, as well a final crossword for you to try, which is – oh my! – a really *difficult* one. Then also there's a bunch of appendices you won't want to skip. And, of course, let's not forget the Index, there's that too, as well as the rather wonderful Back Cover, and…

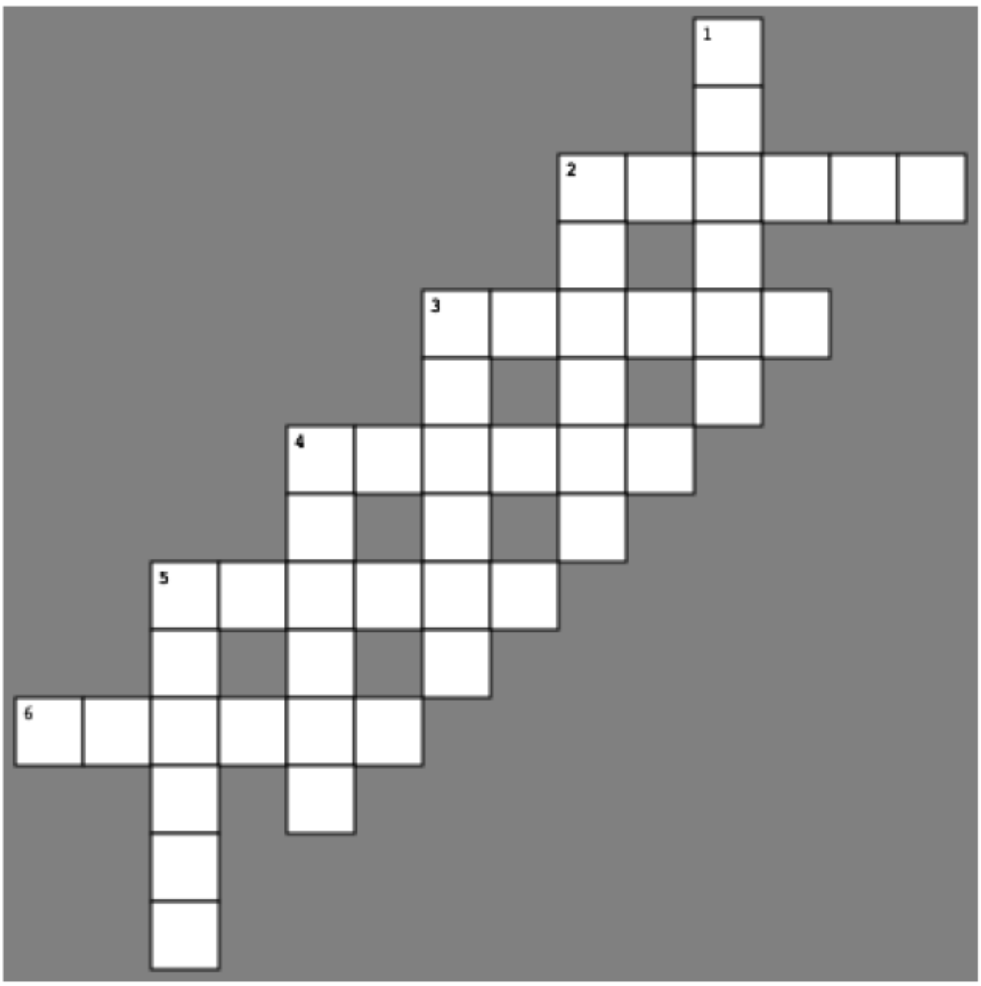
CHAPTER REVIEW BULLET POINTS

- Moving from SQLite3 to MariaDB is not hard, but you do need to get the details right (recall the `autoincrement` to `auto_increment` adjustments).
- Despite SQLite3 and MariaDB using SQL, their dialects differ in subtle ways. Additionally, placeholders in MariaDB use `%s`, whereas SQLite3 uses `?`, so care is needed here.
- The only other changes to the code revolved around setting the correct database credentials (in the `db_details` variable). The rest of the code worked with no changes thanks to your use of `DBcm`.

The Final Crossword



You'll find all the answers to the clues in this book (maybe). The solutions are on the next page.



What a pretty pattern. You'd almost think someone did this on purpose...

Across

2. Your favorite programming language.

3. The main topic of this book.

4. A soothsaying spirit or demon (from classical antiquities).

5. Doesn't kill using venom, but squeezes the life out of its prey.

6. The subject of XKCD 353.

Down

1. As of early 2023, it's now one of the world's most popular programming languages.

2. Data scientists love two languages: R and _____.

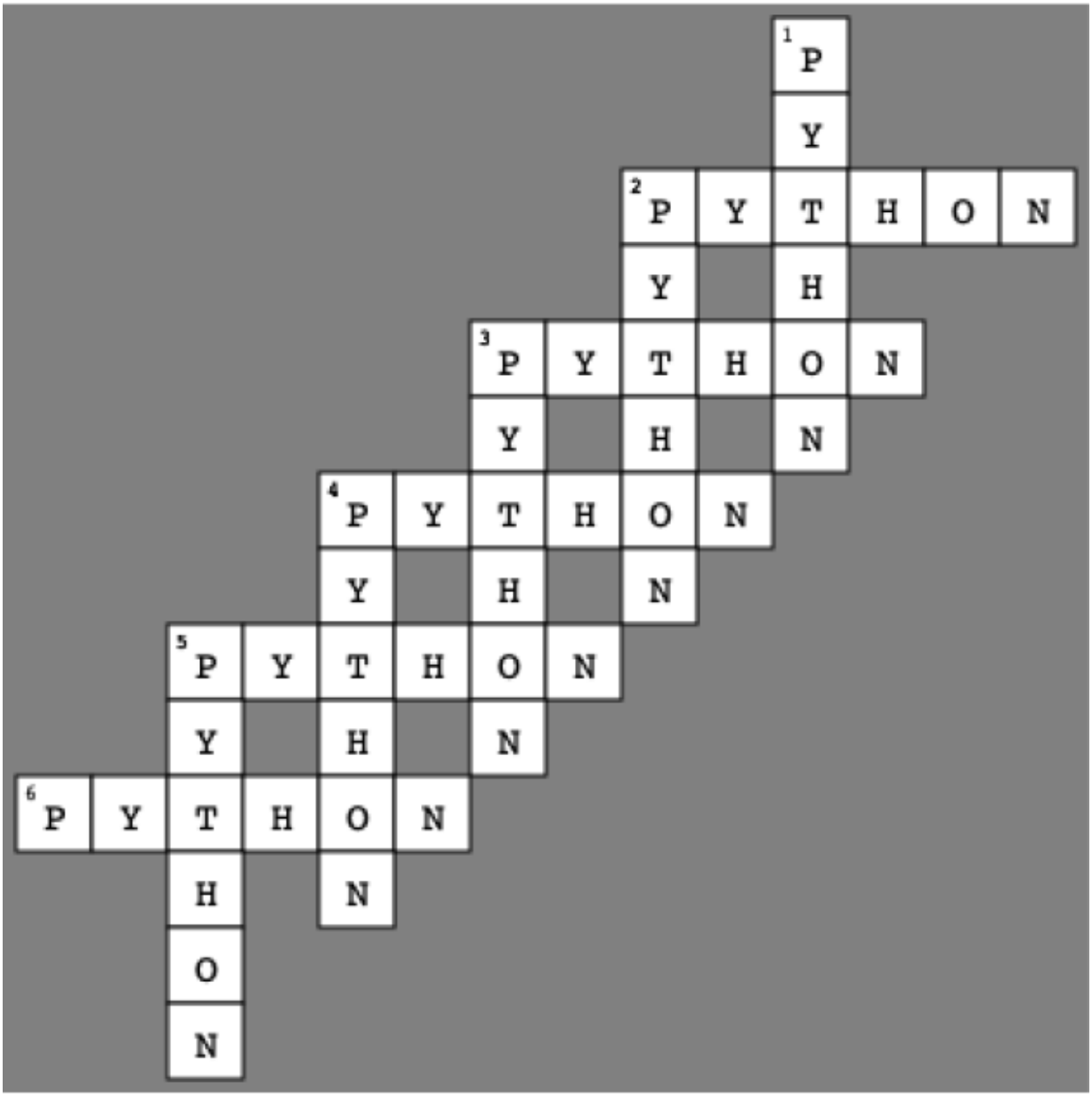
3. As of early 2023, the latest version of this programming language is 3.11.

4. This programming language was named after a famous British comedy troupe (which, it appears, really loved SPAM).

5. It may well be all the programming language you ever need.

The Final Crossword Solution





Across

2. Your favorite programming language.

3. The main topic of this book.

4. A soothsaying spirit or demon (from classical antiquities).

5. Doesn't kill using venom, but squeezes the life out of its prey.

6. The subject of XKCD 353.

Down

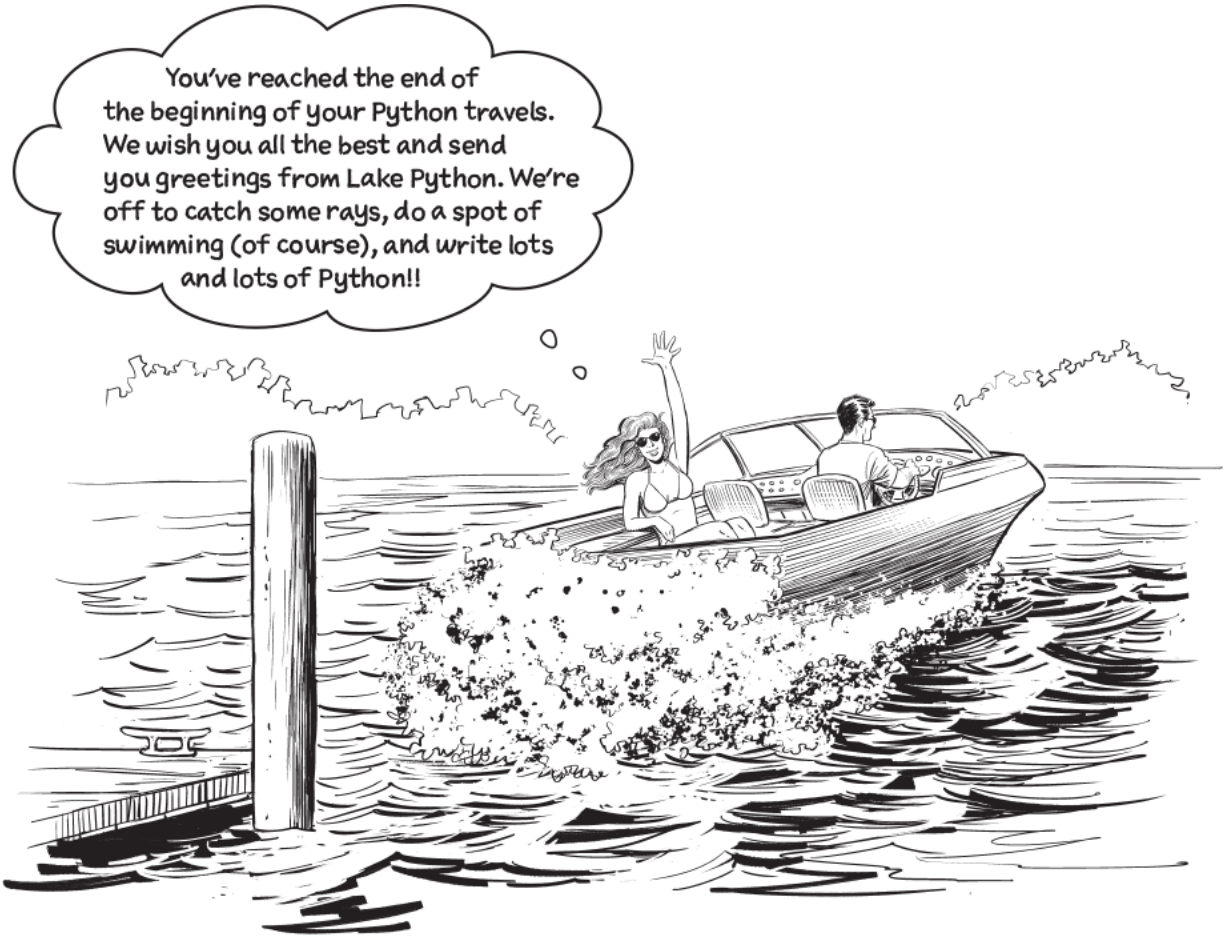
1. As of early 2023, it's now one of the world's most popular programming languages.

2. Data scientists love two languages: R and _____.

3. As of early 2023, the latest version of this programming language is 3.11.

4. This programming language was named after a famous British comedy troupe (which, it appears, really loved SPAM).

5. It may well be all the programming language you ever need.



The next time you've a problem to solve, create a new Jupyter notebook in VS Code, then start typing. We wish you every success as you continue to work with, and learn about, Python. Just don't forget to read the appendices...

Appendix A. The Top Ten Things We Didn't Cover



We firmly believe that it's important to know when to stop.

Especially when your author is from Ireland, a nation famed for producing individuals with a gift for not quite knowing when to stop talking.  And we love talking about Python, our favorite programming language. In this appendix we present the top ten things that, given another six hundred pages or so, we'd happily dive *Head First* into telling you all about. There's information on classes, exception handling, testing, a walrus (Seriously, a walrus? Yes: a walrus), switches, decorators, context managers, concurrency, type hints, virtual environments, and programmer's tools. As we said, there's always more to talk about. So, go ahead and flip the page. Enjoy the next ten pages!

1. Classes



Depending on your programming background, you might find this book's exclusion of how to create a **class** in Python a *curiosity*, an *affront*, or otherwise *hardly noteworthy*.

In some programming languages it's next to impossible to do anything useful until you define a **class**. As we hope this book demonstrates, this is *not* the case with Python. If you are moving to Python from a class-based language, consider this question: *As you worked though this book, did you even miss the* `class` *keyword*?

It's not that we're against classes…

It's more that we're against the idea that using classes for *everything* is where you should start.

Based on the code you've worked on in this edition of *Head First Python*, you'd be forgiven for thinking Python doesn't support the creation of custom classes. Of course, Python does, and they support inheritance (including the multiple type), instances, attributes, methods, private variables, and the like. If *object oriented programming* is your thing, Python has you covered.

And if object oriented programming *isn't* your thing, Python has you covered, too.

But, what if you can't do without a custom class?

When we think we need to define a new class, we work though a series of questions in an effort to convince ourselves we actually need one. Here's what we ask:

1️⃣ Will a dictionary do instead?

If all you need is a way to bundle named data values together, use a dictionary not a class. You'll be much happier.

2️⃣ What about defining a class using the built-in dataclass?

The PSL contains a module called `dataclass`. This module exists to make it easy to create a custom class in Python with the minimum of effort.

3️⃣ What about using the third-party attrs package?

PyPI contains the very highly regarded `attrs`, which can be considered to be a superset of the built-in `dataclass`. Like the latter, it works hard to help you avoid writing all the boilerplate code required to create your own custom class.

If you end up answering *no*, *no*, and *no* to these three questions, feel free to create your own custom class. The Python documentation (especially Chapter 9 of *The Python Tutorial*) is a good starting point.

2. Exceptions

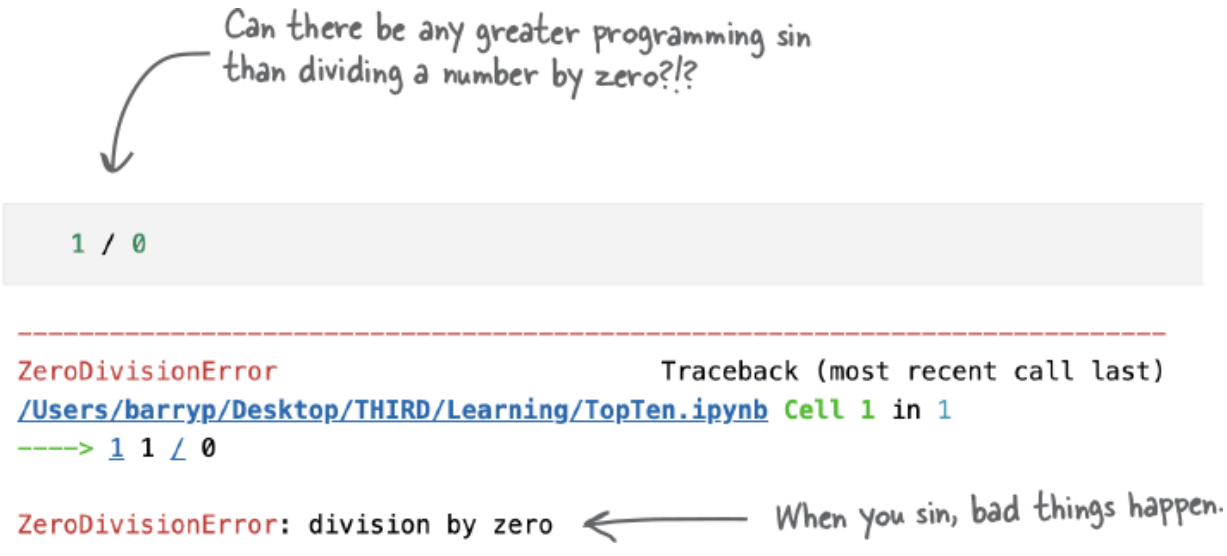


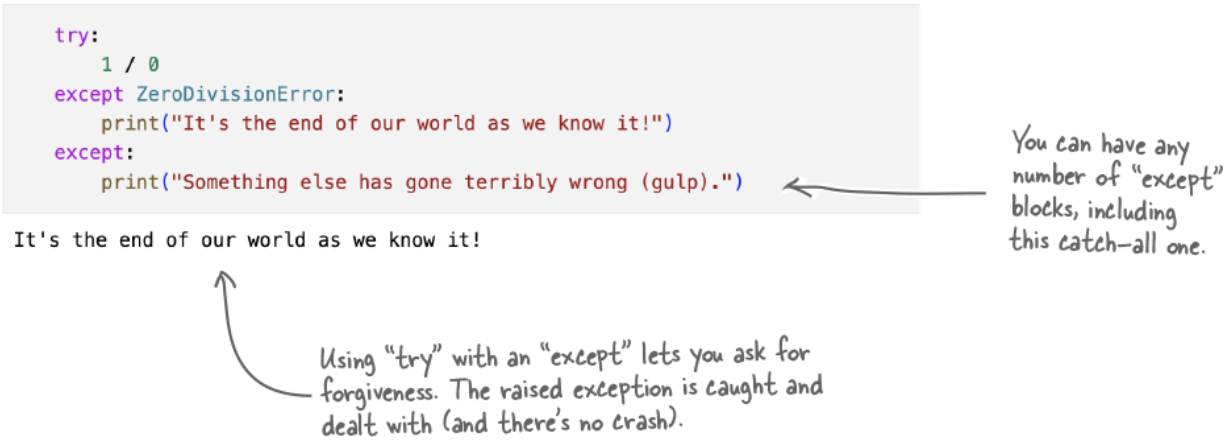
Throughout this book, we (rather naively, to be honest) assumed our code ran without experiencing any *unexpected* run-time errors. Which in The Real World™ is what always happens, isn't it?

When things go wrong, Python *raises an exception*. When faced with a run-time exception in this book, we side-stepped dealing with it by *coding around it*. In effect, we made sure the exception couldn't occur, so no longer had to worry about it being raised. Problem *solved*. Which, as a strategy works… right up until a run-time exception is raised that you weren't expecting. When that happens, things break, and they break badly.

The "solution" is never to hope for the best, ignore run-time exceptions, and keep going. A better strategy is to *catch* the raised exception *before* it stops your code. You can do this with Python's `try…except…` mechanism.

Here's an example of `try` in action:





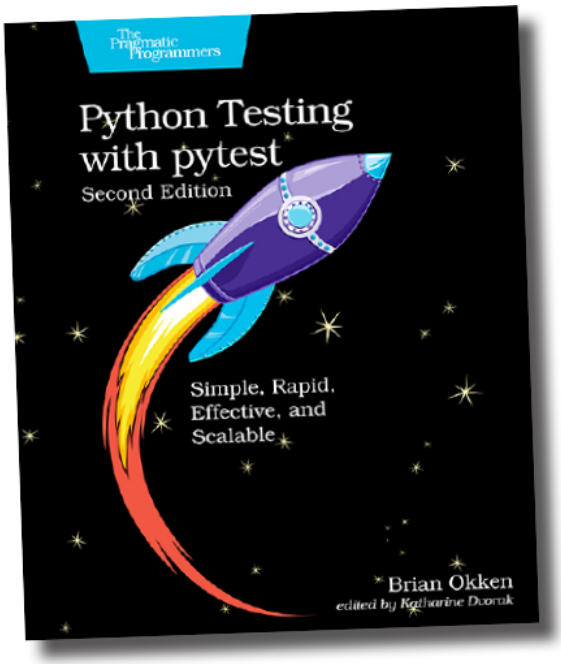
3. Testing

Following on from the previous page, another useful strategy for *avoiding* bugs in your code is to use automated testing to ensure your code runs the way you expect it to.

Python comes with the build-in `unittest` library which is modelled on a similar testing technology originally developed for *Java*. Although part of the PSL, `unittest` is not the recommended automated testing tool nowadays, having been supplanted in many Python programmer's eyes (and toolsets) by `pytest`.

If you are writing code which you hope to deploy to production, then you should be writing tests (using Python) for that code, too. When it comes to learning `pytest`, we recommend anything by *Brian Okken*.

Our favorite "pytest:" book which, if you are serious about testing your Python code, should be your favorite too.



Brian is also one half of the "Python Bytes" podcasting team. Brian, together with Michael Kennedy, help keep everyone up-to-date with a weekly review of all things Python. Michael also hosts the "Talk Python To Me" podcast, which is also worth subscribing to. As if that's not enough (where does he find the time?!?!), Michael runs the "Talk Python Training" site which offers all manner of online training courses to the Python programming community.

4. The Walrus Operator

A recent addition to Python is the notion of an **assignment expression** in the guise of the `:=` operator, known the world over as the *walrus* operator. What the walrus operator lets you do is combine an assignment and expression together, typically replacing two lines of code with one. Here's a quick (and totally contrived) example:

NOTE

The name “walrus” has nothing to do with The Beatles (sadly) and everything to do with how the operator is viewed. If you look at the operator, then tilt your head to the left, you end up looking at something which has two eyes and two big teeth. It looks sort of like a walrus…



The walrus operator lets you *combine* the assignment and expression lines of code:



The assignment expression combination. The use of walrus operator lets you assign to the variable then test against its value in one go, and on one line.

```
if (x := random.randint(0, 10)) < 5:
    print(f"{x} is less than five!")
else:
    print(f"{x} is five or greater.")
```

```
4 is less than five!
```

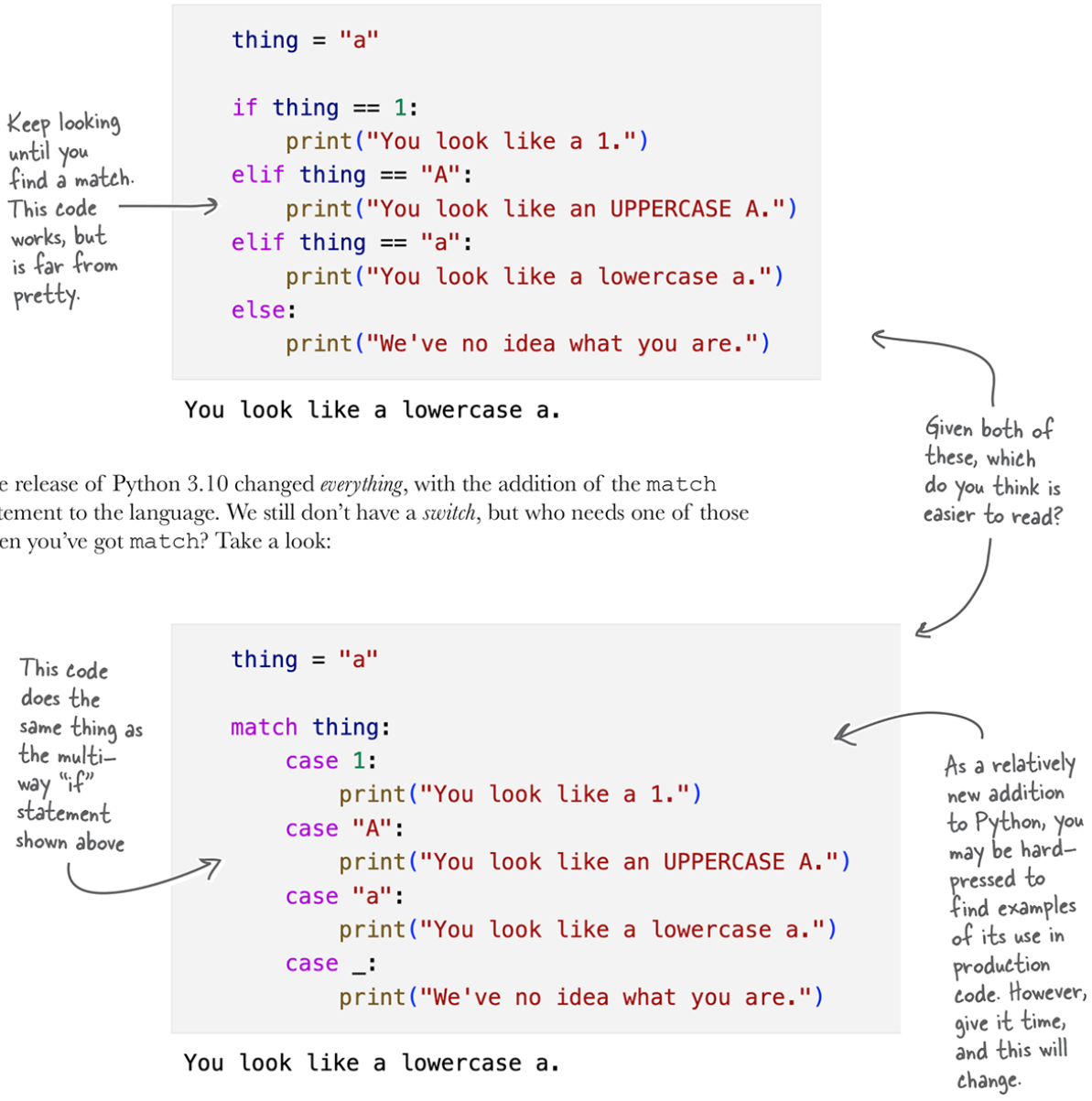
If you're struggling to think if a use-case for the walrus, we find it comes into its own when used with a filtering list comprehension.

Big deal, eh? Well… the addition of the walrus operator to Python caused quite a stir. Things got heated. Battle lines were drawn. An established dictator was *overthrown*. It all got very unfriendly and un-Pythonlike for a while, until everyone (thankfully) calmed down. *Our advice is simple*: if you like the look of the walrus operator, go ahead and use it. If not, then don't. *It's no biggie, either way.*

5. Where's the Switch? What Switch?

A common question when existing programmers first encounter Python is to ask: *Does Python have a switch statement?* For close to thirty years, every Python programmer answered in the negative, then pointed to a multi-way **if** statement (which sort of does the same thing):





The release of Python 3.10 changed *everything*, with the addition of the `match` statement to the language. We still don't have a *switch*, but who needs one of those when you've got `match`? Take a look:

6. Advanced Language Features

In the previous edition of this book, we went out of our way to show you how to create your own custom **decorator**, as well as your own

custom **context manager**. In this edition, the emphasis is less on creating and more on *using* these language features.

You used a decorator when writing Flask code for the Coach's we-bapp, for example:

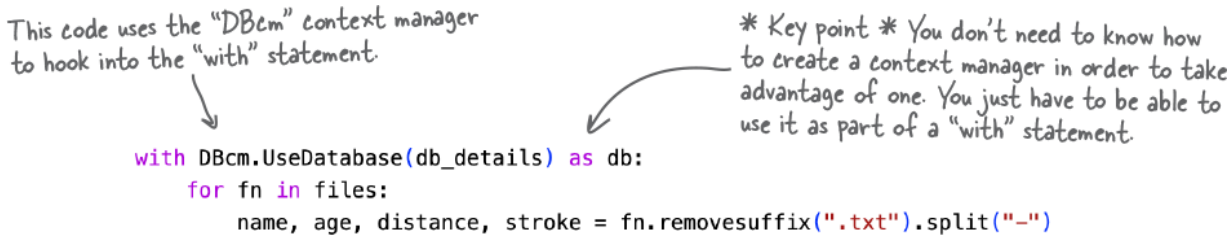


A function decorator allows you to wrap one of your functions with the code from another function, transforming it in some way. Typically, the decorator returns a function to *overwrite* yours.



Now… we've just re-read the previous paragraph and, even though we wrote it, that description still scares the life out of us. In our experience, most Python programmers never need to create a decorator (which can be *hard* to get right), but do need to know how to use them (which is always *easy*). The same goes for **context managers**.

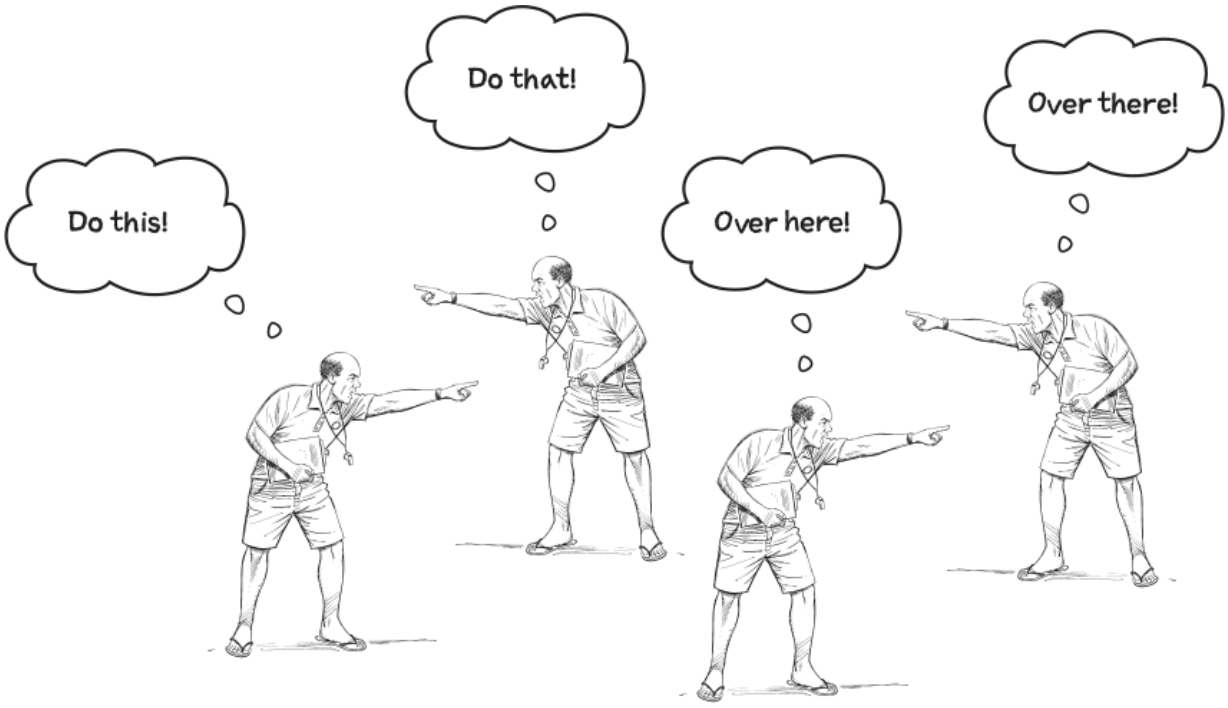
Python's context manager technology is linked to the **with** statement, and you've seen plenty of **with** statements in this book. Most recently, you've used the **with** statement with `DBcm`, creating code like this:



Creating custom decorators and context managers aren't the only advanced feature we sidestepped in this book. There's also the likes of **generators**, **abstract types**, and **metaclasses**. For those readers yearning to learn more about these and other advanced Python language features, take a look at the last page of this appendix for our recommended next steps.

7. Concurrency

Sometimes you need to do more than one thing at the same time in your code. And, just like real-life juggling, it can be hard to keep track of things.



There's just too many coaches to keep track of…

Python supports a number of concurrent programming technologies. If you find yourself needing to code a solution to a problem where different parts of your code needs to *appear* to run in at the same time, you have choices.

NOTE

Just to be clear, not everyone needs to do this sort of thing. It is an advanced technique.

For starters, there's *threads*, and the PSL's built-in `threading` module is where you'll want to start. If fine-grained threading is not what you need, there's the (also built-in) `multiprocessing` module which operates at the *process-level*. And then there's Python built-in support for asynchronous programming using the (relatively new) **async** and **await** keywords, as well as the `asyncio` module. There's also a bunch of third-party modules on PyPI.

WATCH IT!

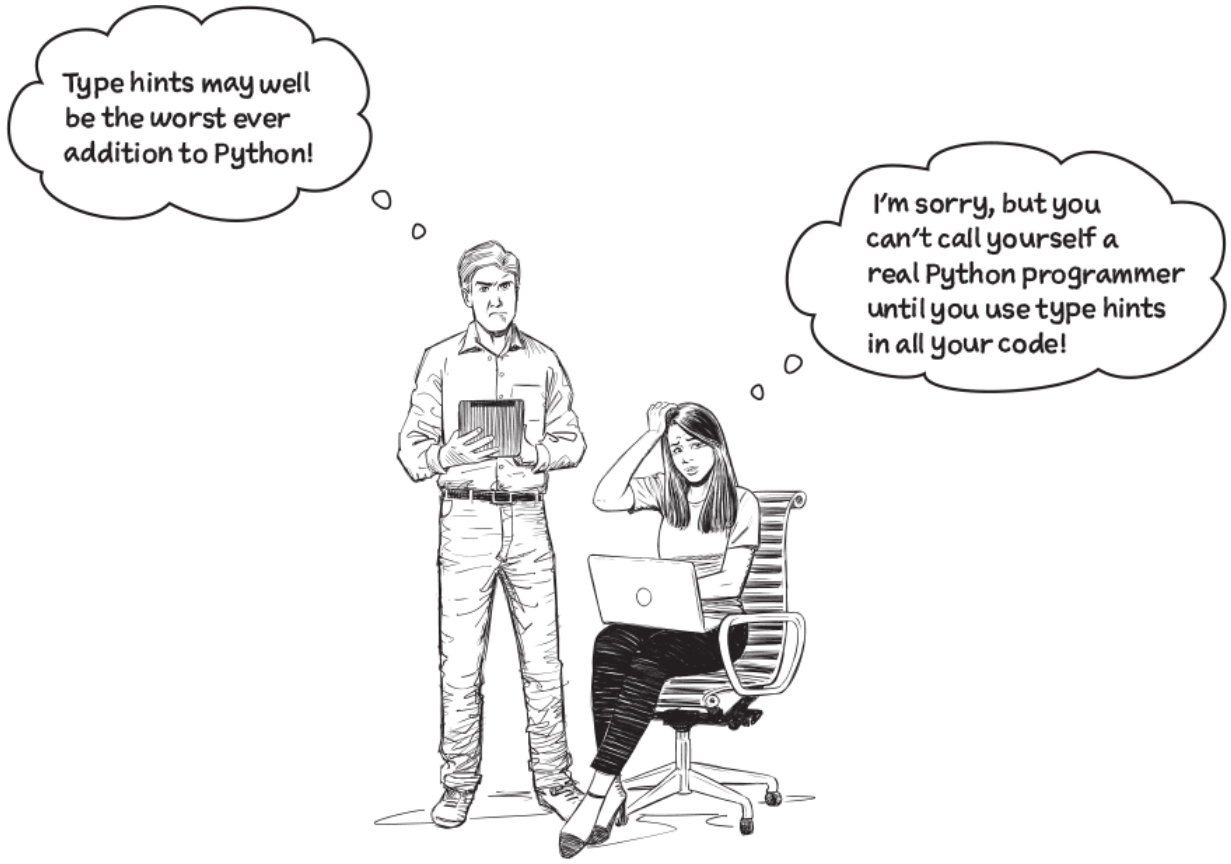


There's been a fair bit of hoopla surrounding the addition of **async** *and* **await** *to Python. So much so, in fact, that you could be forgiven for believing every Python programmer needs to master asynchronous programming for fear of being left behind. This could not be more further from the truth. This type of programming is advanced, and isn't for everybody. If you need it, you'll know.*

---

8. Type Hints

If you want to start a fight at a Python conference, make a wildly provocative public statement about the use of type hints in everyone's code.



Type hints allow you to add an *optional* notation to your function signatures to *hint at* the argument types expected, as well as specify your function's return type.

NOTE

The key word here is "optional".

If your function expects a string as it's sole parameter, and then returns a list to the caller, type hints let you specify this in your code. Other programmers can then read you code, spot the type hints, then (hopefully) avoid sending something other than a string as an argument, nor expect to get back something other than a list.

Some third-party developer tools (with PyPI's `mypy` the most famous) can preprocess your code to ensure the calls to your functions conform to their type hinted signature, complaining loudly when an infraction is spotted. This can be useful to know.

Critically, Python *ignores* all type hints. If you send a dictionary to a function expecting a string, Python happily accepts it, sends the dictionary to your function then – like everyone else – crosses its fingers and hopes for the best. There's no mechanism built into Python to check a call against your type hints (and there likely *never* will be). Type hints are *optional*, after all.

9. Virtual Environments



If you randomly search for and find an online tutorial for any significant Python feature, chances are the tutorial will begin by asking you to create a new virtual environment *before* you get going. Sigh.

Now, don't get the wrong idea here: we don't consider this to be bad advice, more *misplaced*. The problem is that you need to know a bit about how Python works to take advantage of what a virtual environ-

ment offers and, if you're a newbie, you may not be quite there yet. Further, when things go wrong when first setting up a new virtual environment, confusing reigns. Additionally, most tutorials present their advice in such a way that it feels like a virtual environment *has* to be created in order to do *anything* with the Python feature being discussed, when this is clearly *not the case*.

So… what is a *virtual environment* and why might you want to use one?

A virtual environment is a *named clean* installation of Python. Here "clean" means that the only things installed in your virtual environment are Python itself together with the PSL. The "named" bit refers to the fact that each virtual environment is assigned an individual name, and it should come as no surprise that you can have as many virtual environments on your computer as your disk space allows (and you can think up unique names for).

Now – and this is a *key insight* – if you install a package from PyPI into a virtual environment, it is *only* available in *that* named virtual environment. It's not even available in the "main" Python installed on your computer. The virtual environment effectively hides and compartmentalizes a version of Python for you.

As you might have already guessed, it's often regarded as good practice to create a virtual environment for each of your Python projects. This then allows you to only install the packages your project needs, safe in the knowledge that you aren't messing anything up for anything else. Why this is important becomes clear when you consider that a virtual environment can install any release/version of Python as well as any release/version of a PyPI package. So if you have a project that needs Flask v1 to run, and another project which needs Flask v2, simply pop each of your projects into their own virtual environment. This then gives you the flexibility to install whichever version of Flask into each environment *as needed*.

NOTE

You may well be asking yourself "Why didn't we create a virtual environment for the Coach's webapp?". That's a valid question. Recall, however, that this book's goal is to give you a grounding in programming Python, not in using any particular project management tool (which also explains why we didn't show you our use of GIT).

As with most things in Python, you don't have to use virtual environments if you don't want to. Their use is *optional*. It this way, virtual environments are similar to a tool like GIT. You don't have to use GIT with all your projects… if you don't want to. It's up to you. Same goes for virtual environments. It's up to you to decide when to use one.

With all that said and to take a look, start with the PSL's `venv` module.

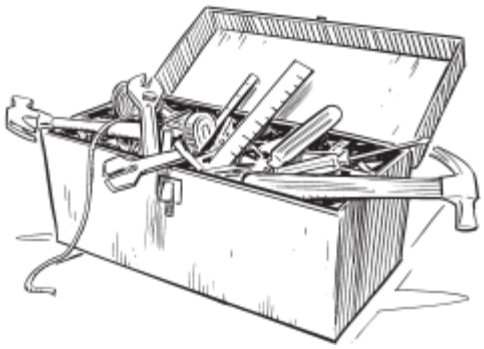
NOTE

When talking about this stuff, many Python programmers shorten "virtual environment" to "venv" (which is also the name of the built-in module).

---

10. Tools

There are a host of tools designed to make your life as a Python developer as productive as possible. Throughout this book, *VS Code* together with *Jupyter Notebook* provide an excellent learning environment, and we encourage you to continue to rely on notebooks when starting a new project, regardless of your goals.



Programmer's code editors (and IDEs)

It's rare indeed to encounter a code editor which does not have great support for Python, and regardless of the editor you class as your favorite, you will likely enjoy code syntax highlighting and much more besides. Direct support for notebooks, like that provided by *VS Code*, is less universal, but is becoming an increasingly popular plugin for modern editors.



If you favor a full-featured IDE, you have choices. The IDE which you'll hear mentioned most often is *PyCharm*, but don't forget about the very capable *Wing Python IDE*. Both of these IDEs (with a bit of tweaking) support notebooks.



Code formatters

If you ask a bunch of programmers of most any other programming language how they format their code, you'll receive a bunch of suggestions (potentially a different suggestion from every one of them). Ask a bunch of Python programmers the same question, and they likely all reply: "Just use **black**". Billing itself as *The Uncompromising Code Formatter*, **black** reformats your code to an agreed standard, no questions asked and no quarter given. You can add **black** to your Python environment from right inside a notebook with the `%pip install black` command. If you do this within *VS Code* you'll get access to a right-clickable context-sensitive pop-up menu which lets you apply **black**'s code formatting standard to an individual code cell or to your entire notebook. Sweet.



There are other code formatters available should **black** not meet your needs. Start your search for what's available on PyPI.

Taking notebooks to the next level

If you enjoy working with *Jupyter Notebook* and want to take it to the next level, check out the **nbdev** project, which lets you create, test, document, and distribute Python packages from within your notebook

environment. And if you like what they do, and are that way inclined, offer to create a groovy logo for the project.





Yes, lots. 😉

Take a moment to review the `TopTen.ipynb` notebook included which this book's download materials. In it, you'll find the code from this appendix as well as clickable links to the stuff referred to over the last ten pages.

Other than that suggestion, we urge you to code, code, then code some more with Python. And we wish you all the fun in the world while you do so.

To *dig deep* into Python as a language, there's only one book in our view which fits the bill: the wonderful *Fluent Python* by Luciano Ramalho (now in its second edition).

This book has lots packed into its 1000+ pages. It's not really a book you "read", it's more a book you "study". It's a wonderful resource for anyone wanting to know most everything there is to know about Python. It does assume you know the fundamentals of the language, which thanks to the book you just finished working through, you do!



About the Author

Paul Barry has a B.Sc. in Information Systems, as well as an M.Sc. in Computing. He also has a postgraduate qualification in Learning and Teaching. Paul has worked at The Institute of Technology, Carlow since 1995, and lectured there since 1997. Prior to becoming involved in teaching, Paul spent a decade in the IT industry working in Ireland and Canada, with the majority of his work within a healthcare setting.