

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

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

[REDACTED]

# **Head First Python**

THIRD EDITION

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

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

**Paul Barry**



Beijing • Boston • Farnham • Sebastopol • Tokyo

# **Head First Python**

by Paul Barry

Copyright © 2023 Paul Barry. All rights reserved.

Printed in the United States of America.

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

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

- Acquisitions Editor: Suzanne McQuade
- Development Editor: Melissa Potter
- Production Editor: Beth Kelly
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- August 2023: Third Edition

## **Revision History for the Early Release**

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

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

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

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05122-0

[FILL IN]

# Preface

---

## Install the latest Python 3

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

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



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

### Installing on Windows

The wonderful Python folk at *Microsoft* work hard to ensure the most-recent release of Python is always available to you via the *Windows Store* application. Open the Store, search for “Python”, select the most-recent version, then click the **Get** button. Watch patiently while the progress indicator moves from zero to 100% then, once the install completes, move to the next page – you’re ready.

### Installing on macOS

The latest Macs ship with older, out-of-date releases of Python. *Don’t use these*. Instead, head over to Python’s home on the web, <https://www.python.org/>, then click on the “Downloads” option. The latest

release of Python 3 should be going to download, as the Python site is smart enough to spot you’re connecting from a Mac. Once the download completes, run the installer that’s waiting for you in your Downloads folder. Click the **Next** button until there are no more **Next** buttons to click then, when the install is complete, move to the next page – you’re ready.

### NOTE

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

## Installing on Linux

The *Head First Coders* are a rag-tag team of techies whose job is to keep the *Head First Authors* on the straight and narrow (no mean feat). The coders love *Linux* and the *Ubuntu* distribution, so that’s discussed here.

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

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

## Python on its own is not enough

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

When it comes to creating Python code, you can use just about *any* programmer’s editor, but we’re recommending you use a specific one when

working through this book's material: Microsoft's *Visual Studio Code*, known the world over as **VS Code**.



## Install the latest Jupyter Notebook back-end

### NOTE

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

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

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

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



## Install the latest release of VS Code

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

<https://code.visualstudio.com/Download>

### NOTE

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

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

## Configure VS Code to your taste

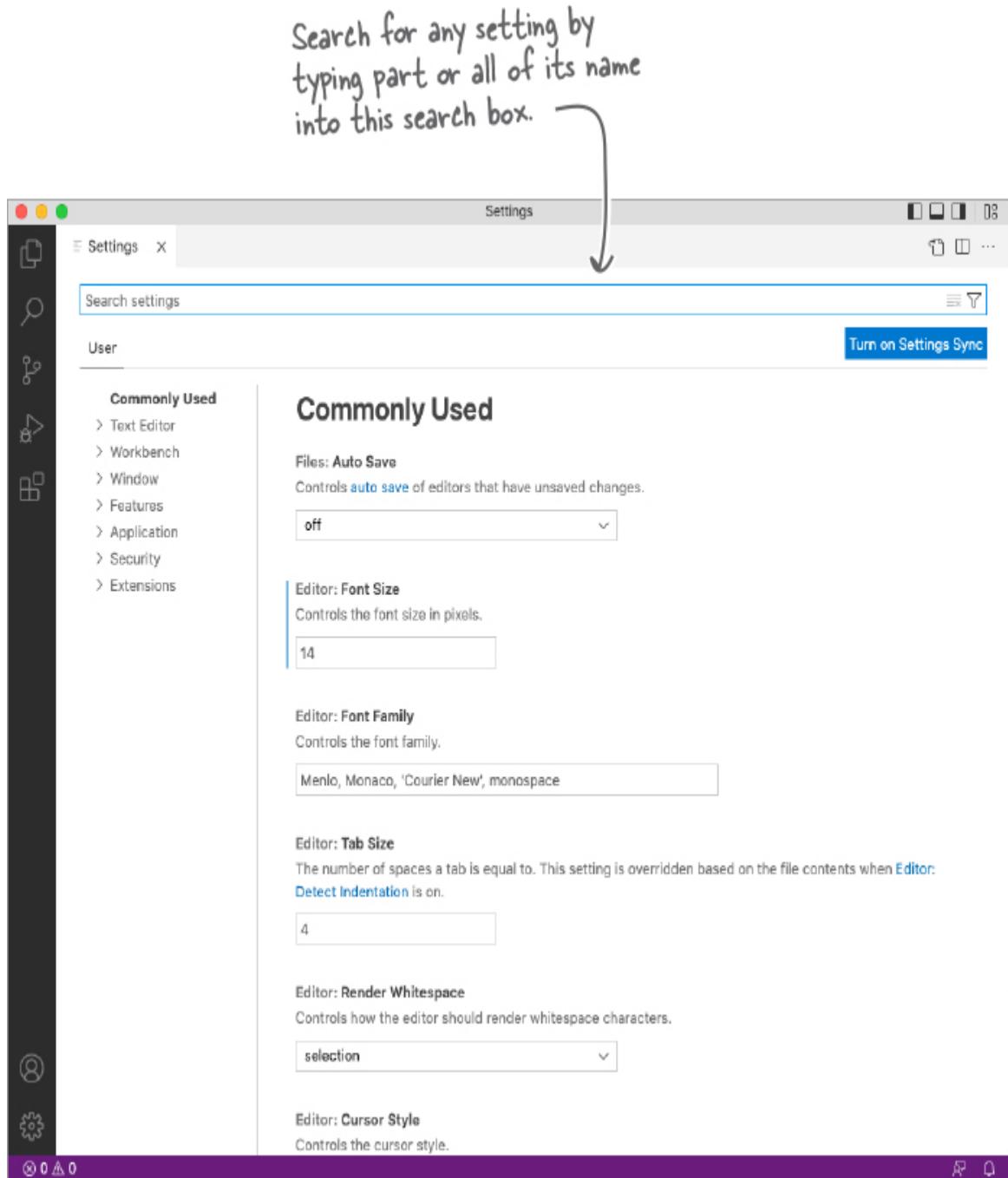


Go ahead and run VS Code for the first time. From the menu, select the **File**, then **Preferences**, then **Settings** to access the editor's settings preferences.

### NOTE

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

You should see something like this:



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

- The *indentation guides* are switched off.
- The editor's *appearance theme* is set to *Light*.

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

#### NOTE

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

## Add 2 required extensions to VS Code

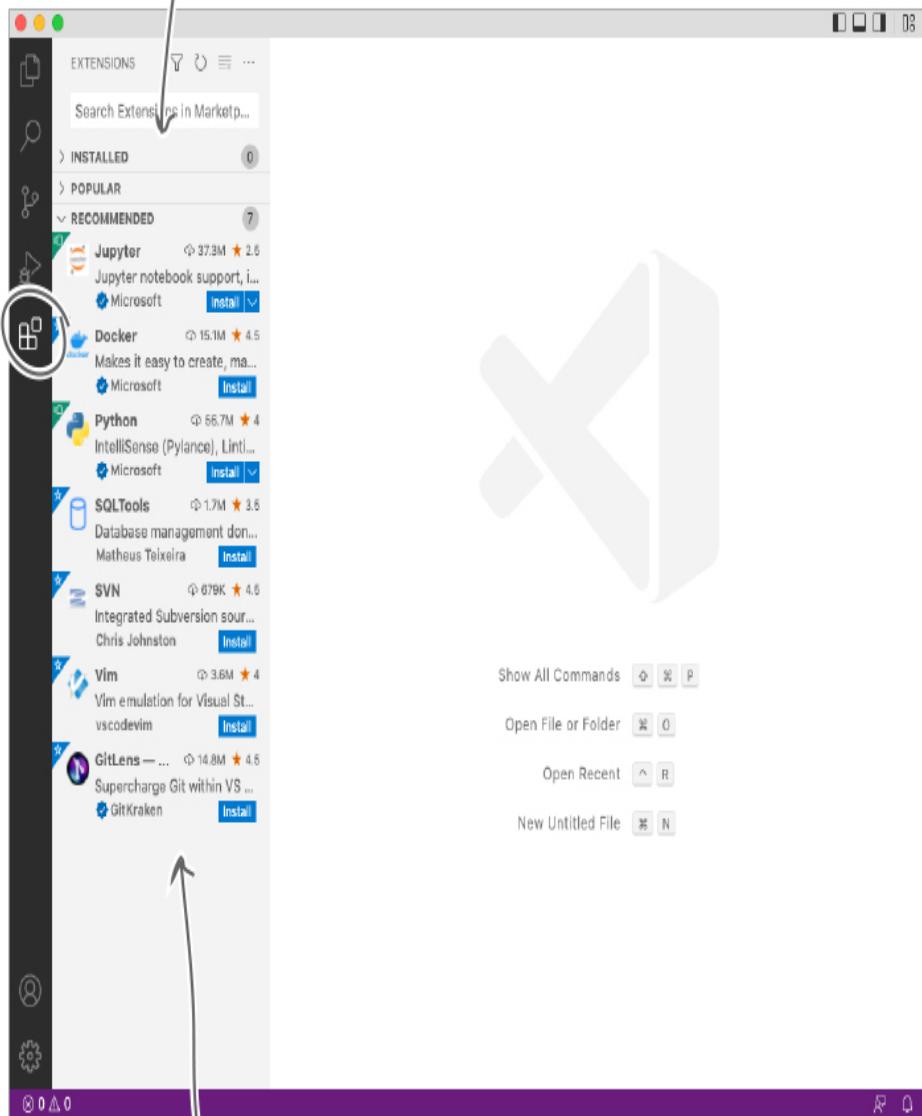


Whereas you are not obliged to copy the recommended editor setup, you absolutely have to install two VS Code extensions, namely **Python** and **Jupyter**.

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

The list of installed extensions is initially empty. But - look! - VS Code is making some recommendations.

Click on the Extensions icon to slide out the panel. You can search, select, and install extensions from here.

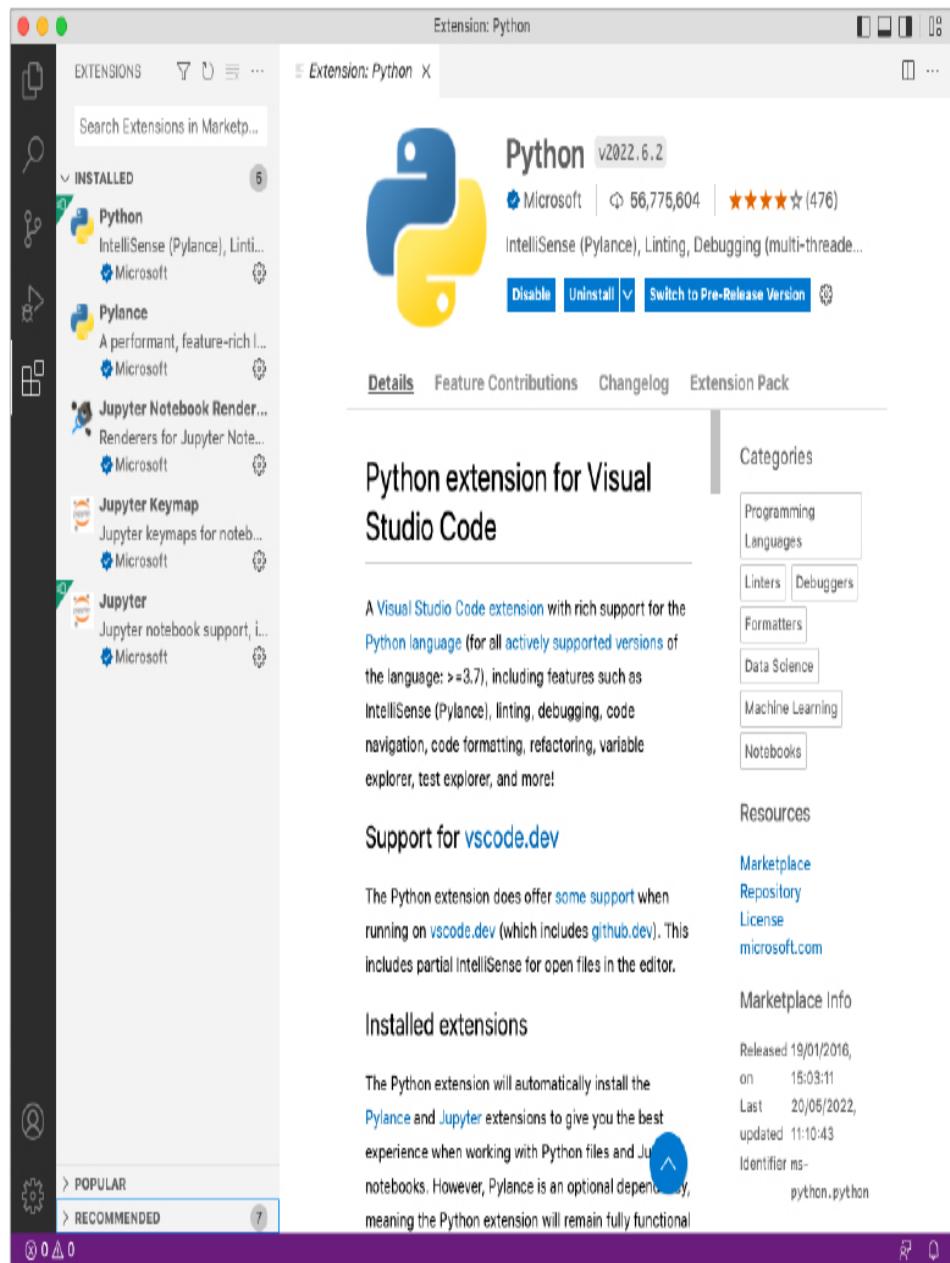


If you are seeing "Python" and "Jupyter" on your list, click the Install button to add each extension to VS Code. If you are not seeing these recommendations, use the search box to find them, then perform the install.

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

### NOTE

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

# Chapter 1. Why Python?: *Similar But Different*

---

## A NOTE FOR EARLY RELEASE READERS

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

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

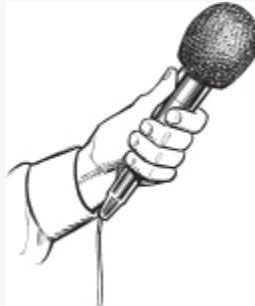
If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).



**Python starts counting from zero, which should sound familiar.** In fact, Python has a lot in **common** with other programming languages. There's **variables**, **loops**, **conditionals**, **functions**, and the like. In this, our opening chapter, we take you on a **high-level whistle-stop tour** of Python's basics, introducing the language without getting too much into the weeds. You'll learn how to **create** and **run** code within VS Code and Jupyter Notebook. You'll see how lots of programming functionality comes **built-in** to Python, which you'll **leverage** to get your job done. You'll also learn that although Python shares a lot of the ideas with other programming languages, how they manifest in your Python code can be, well, **different**. Now, don't get

the wrong idea here: we're talking different **good**, not different *bad*. Read on to learn more...

## IT'S A HEAD FIRST EXCLUSIVE



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 to most other programming languages. I do variables, loops, conditionals, functions, and so on, just like all the others.

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

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

**Head First:** Such as...?

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

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

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

**Head First:** And why do you think that is?

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

**Head First:** What else?

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

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

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

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

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

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

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

**Head First:** Py...P... what?!?

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

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

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

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

**Python:** Yes, that's it: *leverage*. My guiding philosophy is to ensure programmers only ever write new code when 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 talking 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 1960's British comedy troupe.

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

**Python:** I'm telling you not a word of a lie. Look it up (see: <https://docs.python.org/3/faq/general.xhtml#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!



### For sure. But, that's not all.

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

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

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

- ❶ Python code is easy to read.**
- ❷ Python comes with a Standard Library.**
- ❸ Python has practical, powerful, and generic built-in functions (BIFs).**
- ❹ Python comes with built-in data structures.**
- ❺ Python has the Python Package Index (PyPI).**
- ❻ Python doesn't take itself too seriously.**

#### NOTE

Don't underestimate the importance of this last one.

## **LOOK HOW EASY IT IS TO READ PYTHON**

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

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

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

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

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

```
import random
```

```
def draw():
```

```
    which_suit = random.choice(suits)
```

```
    which_type = random.choice([faces, numbered])
```

```
    which_card = random.choice(which_type)
```

```
    return which_card, "of", which_suit
```

```
draw()
```

```
draw()
```

```
draw()
```

## **LOOK HOW EASY IT IS TO READ PYTHON**

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

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

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

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

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

```
import random
```

A library is included to provide support for randomness.

```
def draw():
```

The empty parentheses are a bit of a give away. This looks like the start of a function.

```
    which_suit = random.choice(suits)  
  
    which_type = random.choice([faces, numbered])  
  
    which_card = random.choice(which_type)
```

Randomly select a suit and assign it to a new variable called "which\_suit".

```
    return which_card, "of", which_suit
```

Assign either the "faces" or "numbered" array to the "which\_type" variable. Do so randomly.

Randomly select a card from the "which\_type" array - it'll be either a face or a number value.

Return the randomly selected card, then word "of", then the randomly selected suit.

```
draw()
```

Invoke the "draw" function to randomly generate a drawn card from the deck.

```
draw()
```

Ditto.

```
draw()
```

Ditto.

## Getting ready to run some code

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

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

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



## 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 **Install the latest Python 3**. You'll find the instructions to install VS Code and some required extensions there.

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

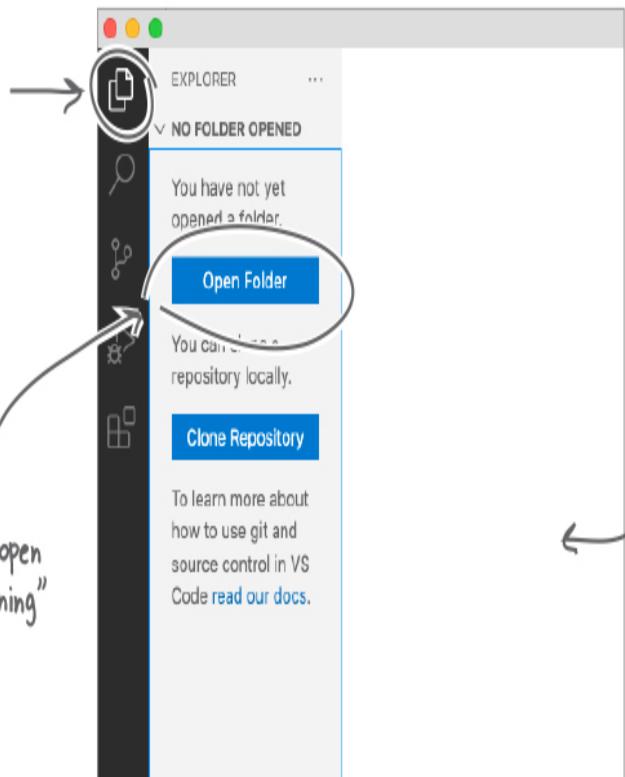
## NOTE

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



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.



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

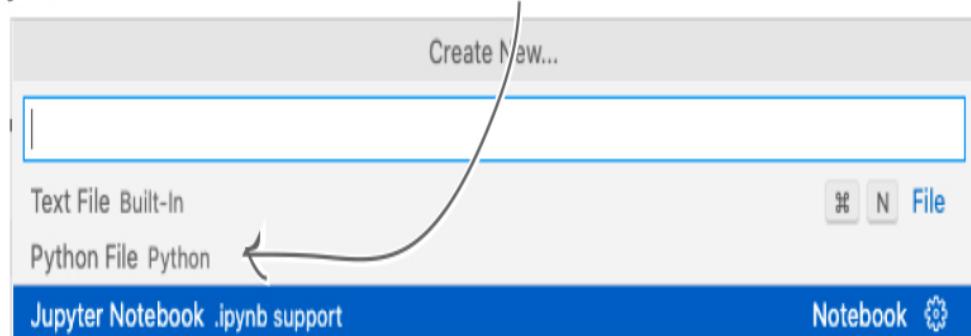
## Preparing for your first Jupyter experience

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:



1. Select this option if you want to create a new empty text file (but don't pick this option now).

2. Select this option to create a text file to specifically contain Python code (and, again, don't pick this option on this occasion).



3. Select this option to create a new "Jupyter Notebook". This is the option you'll want to choose each time you start a new notebook, and - yes! - choose this option now.

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



This "code cell" is your interactive window into Python, allowing you to enter **\*any\*** amount of Python code into the cell. Think of code cells as little text editors embedded within your notebook.

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

## TEST DRIVE



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

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

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



### No need. Just press Shift+Enter.

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

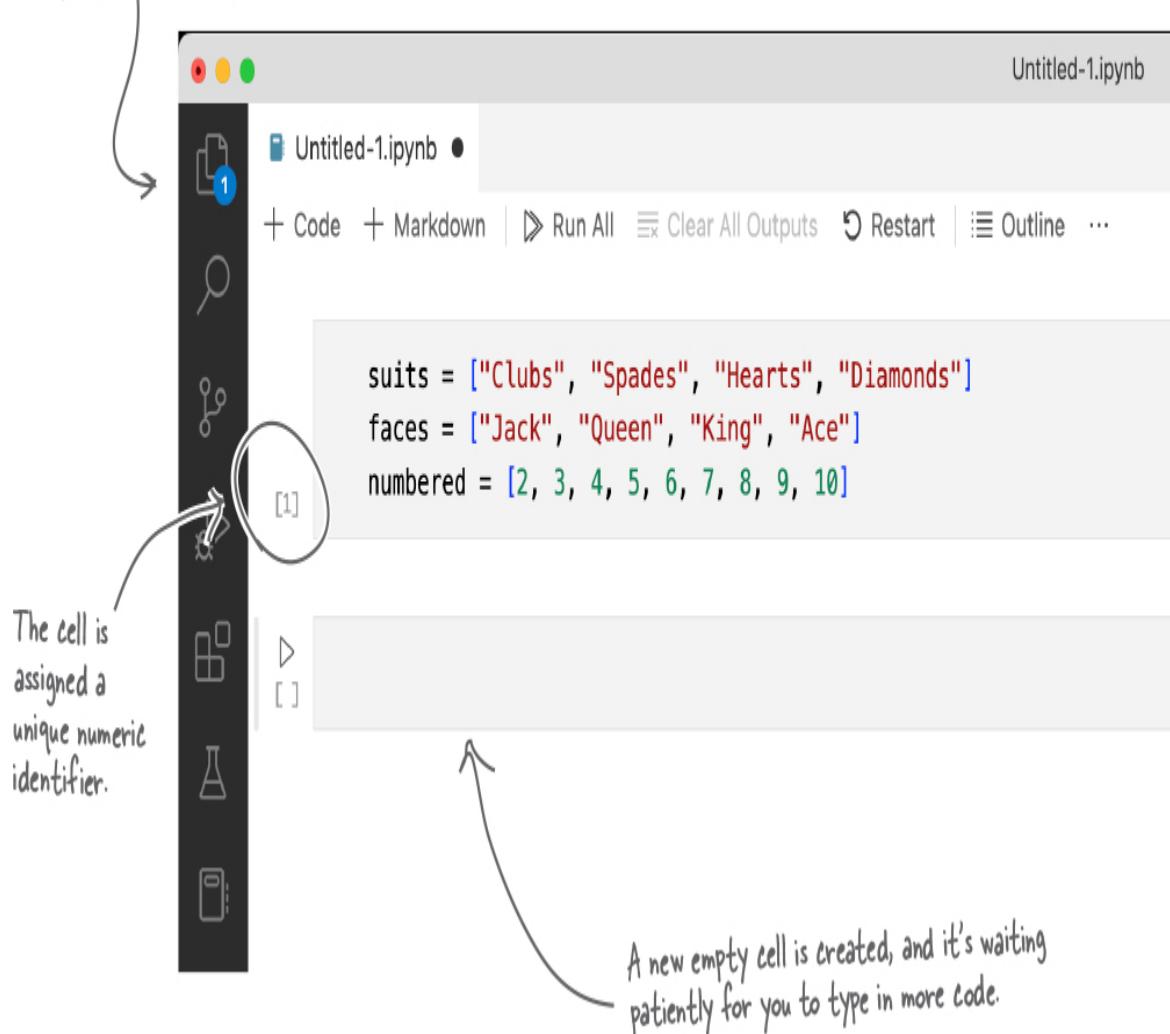
## NOTE

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

## Pressing Shift+Enter runs the cell’s code

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

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.



Maybe I'm missing something, but  
I don't think anything happened  
there. Did that code even run?



### The code ran, but produced no output.

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

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

## EXERCISE



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

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

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

*Note the use of  
indentation to denote  
the block of code* →

← *Do you recall  
what this  
code does?*

## THERE ARE NO DUMB QUESTIONS

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

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

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

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

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

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

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

## EXERCISE SOLUTION



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

Each executed cell is sequentially numbered.

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

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.

Grab the "random" module from Python's standard library.

```
[2] import random
```

Define the first version [3] of the "draw" function.

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

This notebook now has four code cells.



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

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

## TEST DRIVE



Let's draw some cards from your card deck.

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

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

[4] draw()

Each call to the "draw" function produces output, which VS Code displays on the screen. Note the three dots → ... ('Jack', 'of', 'Clubs') which are a visual clue.

[5]

draw()

Each call to the "draw" function returns a different random card. Cool.

[6]

draw()

... ('10', 'of', 'Clubs')

You should see three different card draws. If you're seeing \*exactly\* the same results as we are, then we only have one word for you: Freaky.

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

Besides Jupyter, there are other ways to run Python code, and you'll learn about some of them as you work through this book. However, using VS Code with the Jupyter extension is – in our view – the *perfect* way to read,

run, experiment, and play with Python code when first learning the language. So get ready to spend *a lot* of time in Jupyter and VS Code.

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

#### NOTE

Do this now!

## What if you want a bunch of cards?

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

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

The "for" loop, when combined with the "range" BIF, iterates a fixed number of times. In this case, this code loops FIVE times.

The "range" BIF returns a list of numbers up to, but not including, the provided number. On each iteration, the current number is assigned to the loop's variable, called "n" in this code.

```
for n in range(5):  
    card = draw()  
    print(f"{n} -> {card}")
```

Did you  
spot the  
indentation?

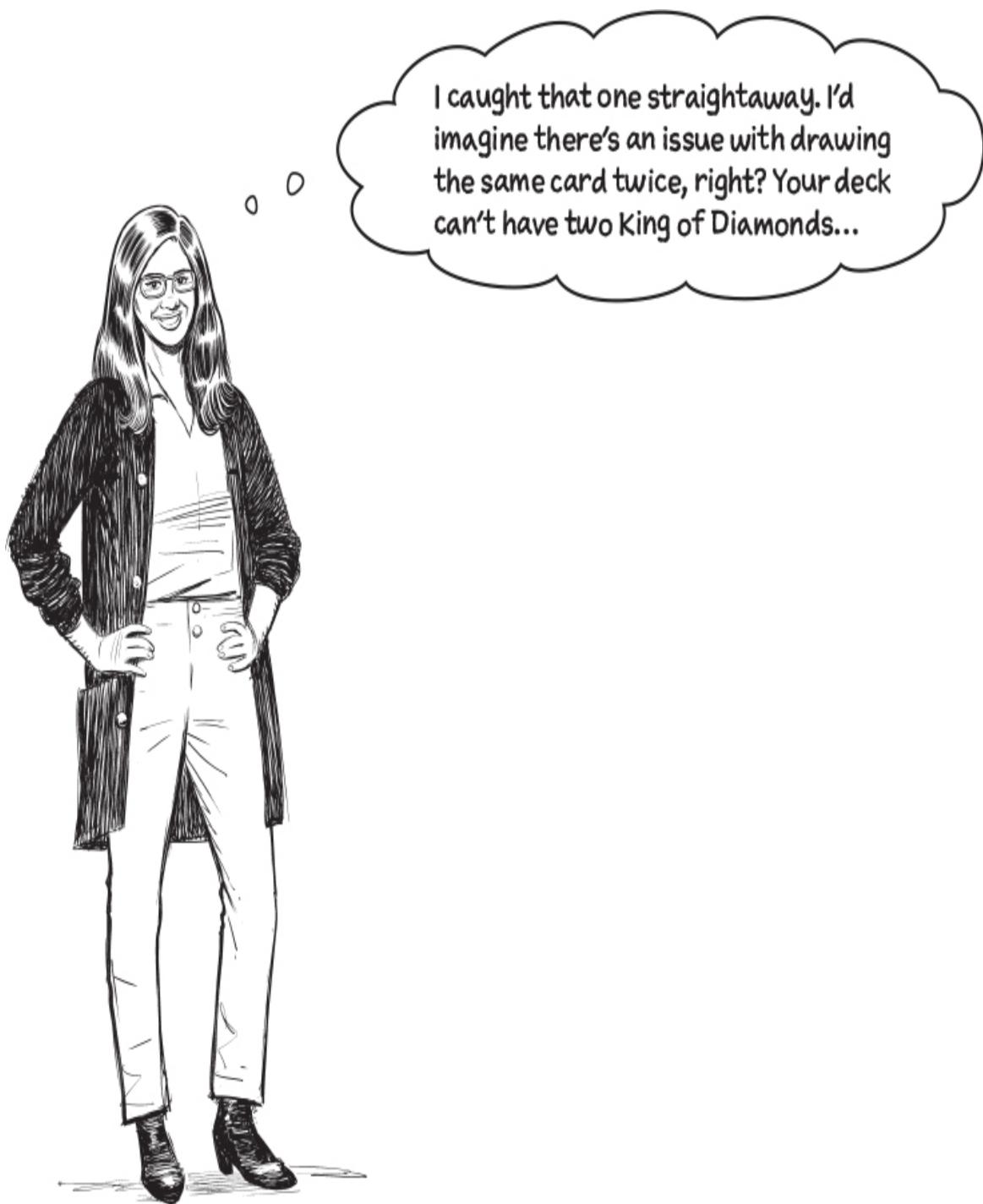
```
0 -> (6, 'of', 'Hearts')  
1 -> (7, 'of', 'Spades')  
2 -> ('King', 'of', 'Diamonds')  
3 -> ('King', 'of', 'Diamonds')  
4 -> (4, 'of', 'Spades')
```

Like all those other programming languages, Python starts counting from zero, with the numbers 0 through 4 confirming this loop's block executed FIVE times.

"print" is another BIF (built-in function), which is used to display a message on screen. In this code, the message is a formatted string made up from the values of "n" and "card".

As these cards are selected at random by the "draw" function, you'll likely see five different cards when you run this loop.

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



**Yes, the loop code drew the same card twice.**

Of course, the assumption here is that a drawn card is not put back into the deck of cards. So... unless you're playing with "magic cards" (or someone's

up to no good), the fact that this code randomly selects the same card more than once isn't likely what's required.

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

## BRAIN POWER



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

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

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

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

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

Lists start and end with a square bracket.

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

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**:

`draw()`

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

→ (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 modelling a deck of cards. Some other data structure is needed here. But, which one?

## NOTE

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

## Model your deck of cards with a...

Although Python's **dictionary** data structure is *hugely* popular, it primarily provides for lookup functionality (which features *lots* later in this book). Despite this, and like a list and tuple, a dictionary isn't the best fit for modelling your deck of cards.

Which leaves you with **set**.

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

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

Start by assigning  
an empty set to  
a variable called  
"deck".

This is another BIF  
which creates sets.

```
deck = set()  
for suit in suits:  
    for card in faces + numbered:
```

Cycle through each of  
the four suit values,  
then cycle though all  
of the possible card  
values, Jack, Queen,  
King, Ace, then 2  
through 10.

```
        deck.add(f"{card} of {suit}")
```

Did you notice  
that Python's  
"+" operator is  
being used to  
concatenate the  
two lists?

The formatted  
string is being added  
to the "deck" set.



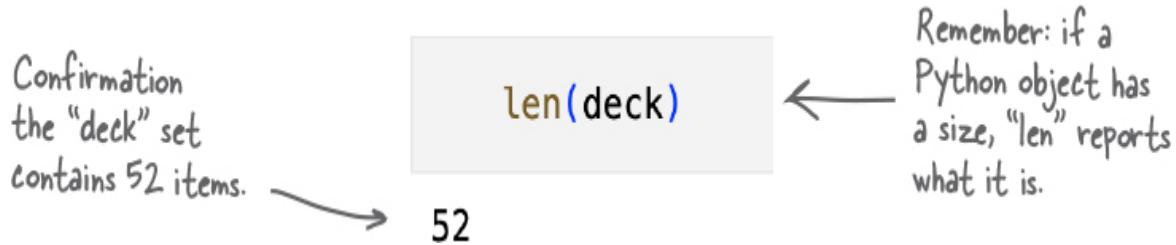
That shouldn't come as a surprise.

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

## Use BIFs to learn more about your variables

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

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



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

```
print(deck)
```

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

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

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

## Chaining BIFs to get more done

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

The **sorted** BIF is often combined (or *chained*) with the **print** BIF to display an ordered version of some data. Care is needed, though, as the ordered data returned by **sorted** is always a list, regardless of what's being

sorted. This means using **sorted** on a set does *not* produce a sorted set, it produces a *sorted list* made up from the data in the original, *unordered* set. The original set is unchanged, in that it's still a set, which is still unordered.

**Remember:** “**BIF**” is shorthand for “*built-in function*”.

Read this chain from the inside-out. Here, the "deck" set is first turned into a sorted list, which is then given to "print" for display on your screen.

```
print(sorted(deck))
```

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

Compare this output to the output on the previous page. This is a list (note the square brackets), not a set, as the "sorted" BIF \*always\* returns a list.

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

## The print dir combo mambo

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

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



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']
```

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

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

## Getting help with dir's output

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

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



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.remove)
```

Help on built-in function remove:

remove(...) method of builtins.set instance

Remove an element from a set; it must be a member.

This line of  
output is the  
important bit  
here.

If the element is not a member, raise a KeyError.

It's also possible to view this documentation within VS Code by hovering over the word "remove" 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).

## This feels like a deck of cards now

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

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

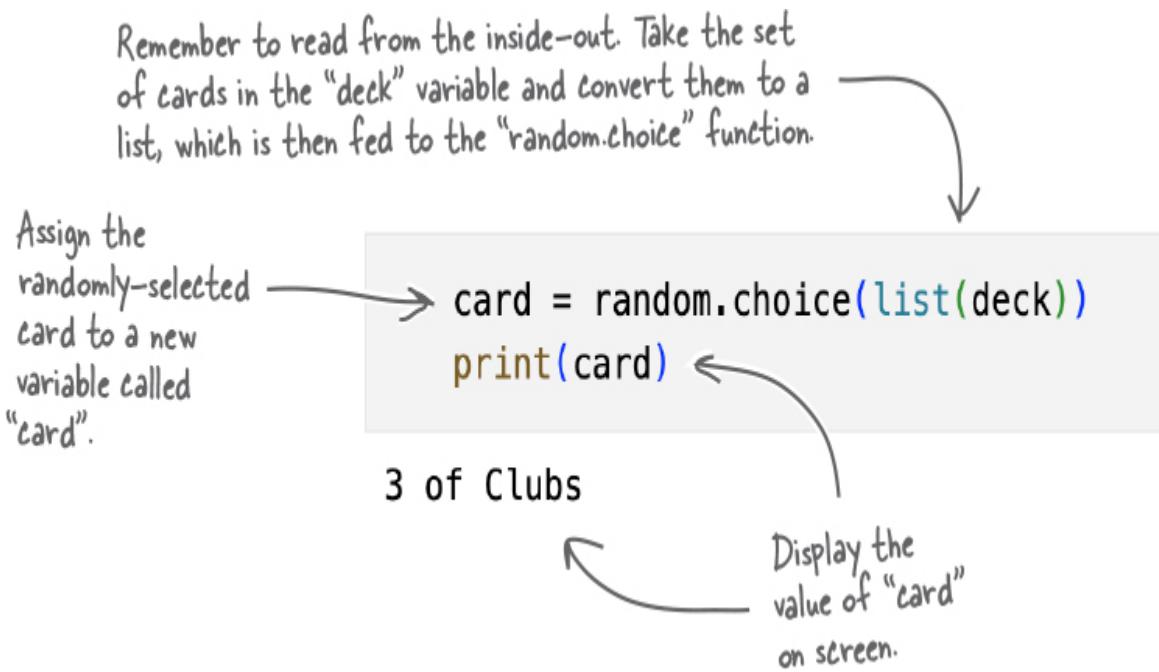
### NOTE

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

## NOTE

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

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



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

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 → 51  
each successful “remove”.

## What exactly is “card”?

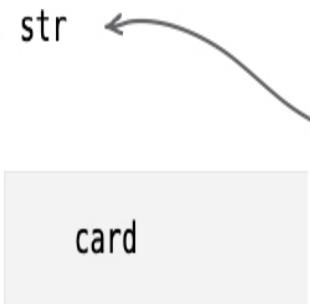
If you're wondering what the `card` variable is, there's another BIF, called `type`, which reports the type of the value currently assigned to `card`:



The “type” BIF  
asks the question:  
“What is?”

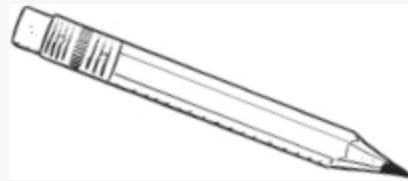
```
type(card)
```

The "card" variable refers to a string. To see the value of "card", type the variable name into a new notebook cell, then press "Shift+Enter".



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

## SHARPEN YOUR PENCIL



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

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

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

---

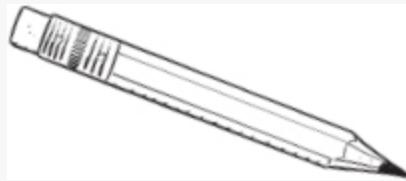
---

---

---

---

## SHARPEN YOUR PENCIL SOLUTION



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

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

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

Define a function called "draw". → def draw():

Randomly select a card. ← card = random.choice(list(deck))

Remove the drawn card from the deck. → deck.remove(card)

Return the selected card to the calling code. ← return card

## THERE ARE NO DUMB QUESTIONS

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

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

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

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

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

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

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

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

## TEST DRIVE



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

Define the function. → `def draw():`

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

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

Call the "draw" function, get back a card. → `draw()`

'4 of Hearts'

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

This code is copied from earlier. It builds → a complete set of cards.

```
deck = set()
for suit in suits:
    for card in faces + numbered:
        deck.add(f"{card} of {suit}")
```

Exactly what  
you'd expect.

`len(deck)`

52

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

Five randomly selected cards from the deck with no duplicates in sight.

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

King of Spades  
Queen of Clubs  
2 of Diamonds  
Queen of Spades  
8 of Hearts

A slight variation on the "for" loop used earlier. Rather than remember (and display) the number of each iteration, this loop uses Python's default variable (a single underscore) to ignore the iteration count..

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

No surprise here.

`len(deck)`

47

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

'2 of Diamonds' in deck

False

As the "2 of Diamonds" is no longer in the deck (the loop at the bottom of the previous page drew it), this statement evaluates to a boolean "False".

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

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

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

Draw another card. That one's taken.

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



### The parentheses are optional.

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

## But, wait! There's more...

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



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

As this is a *Head First* book, it's not enough we tell you what these are, we want you to *experience* them. So, in VS Code, close your **Cards** notebook,

then create a new notebook called `WhyPython.ipynb`. You'll work in this new notebook for the rest of this chapter.

### NOTE

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



**Yes. Every... single...word.**

Oh, we're only joking. 😊

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

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

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

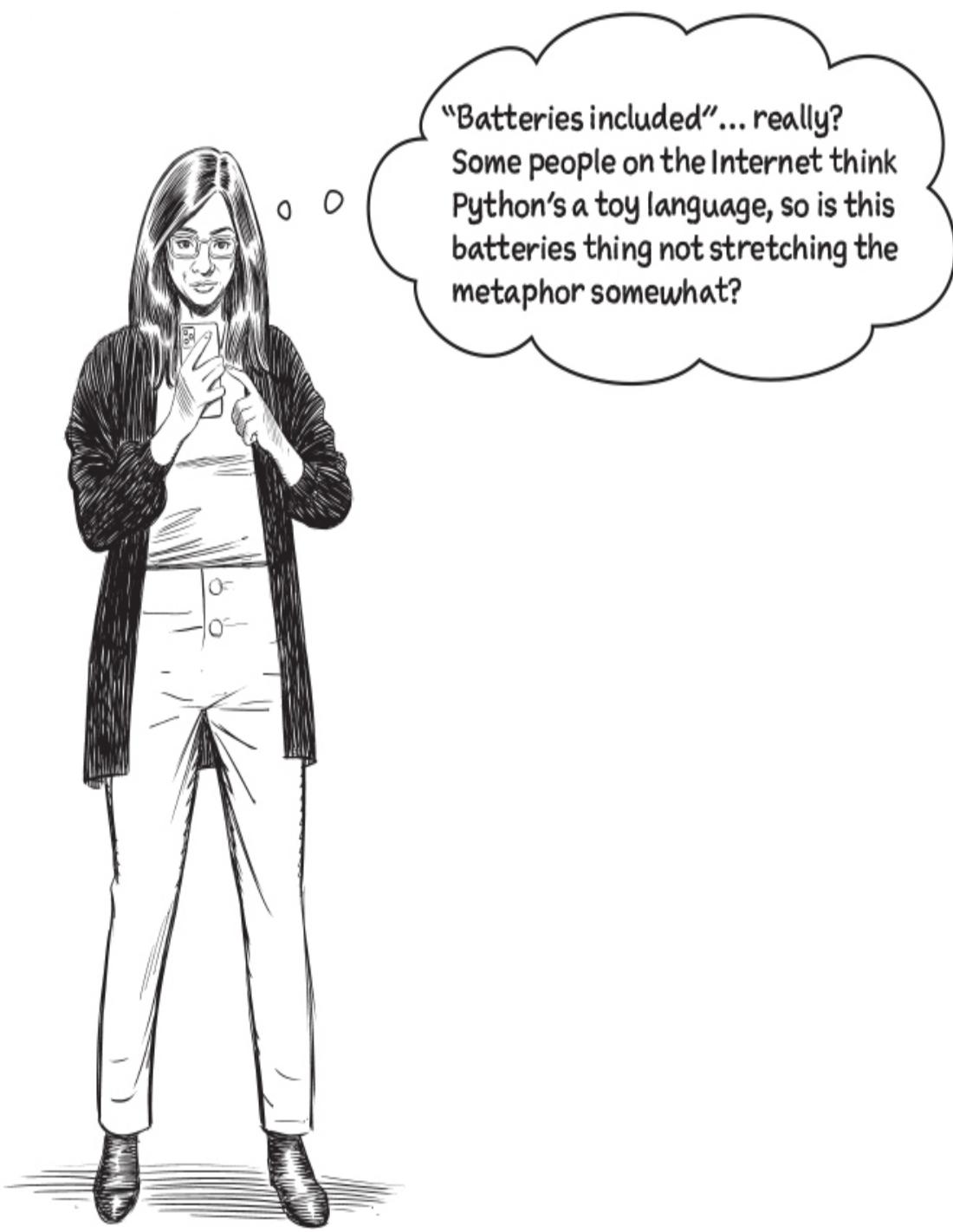
## Python ships with a rich standard library

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

When you hear programmers refer to Python as coming with “batteries included”, they are referring in part to the PSL. There’s a lot to it: <https://docs.python.org/3/library/index.xhtml>.



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



### **It's quite an apt description.**

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

which means the standard install of Python works “right out of the box” without the need for anything extra. Hence, *batteries included*.



## BTW: Python is not a “toy language”

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

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

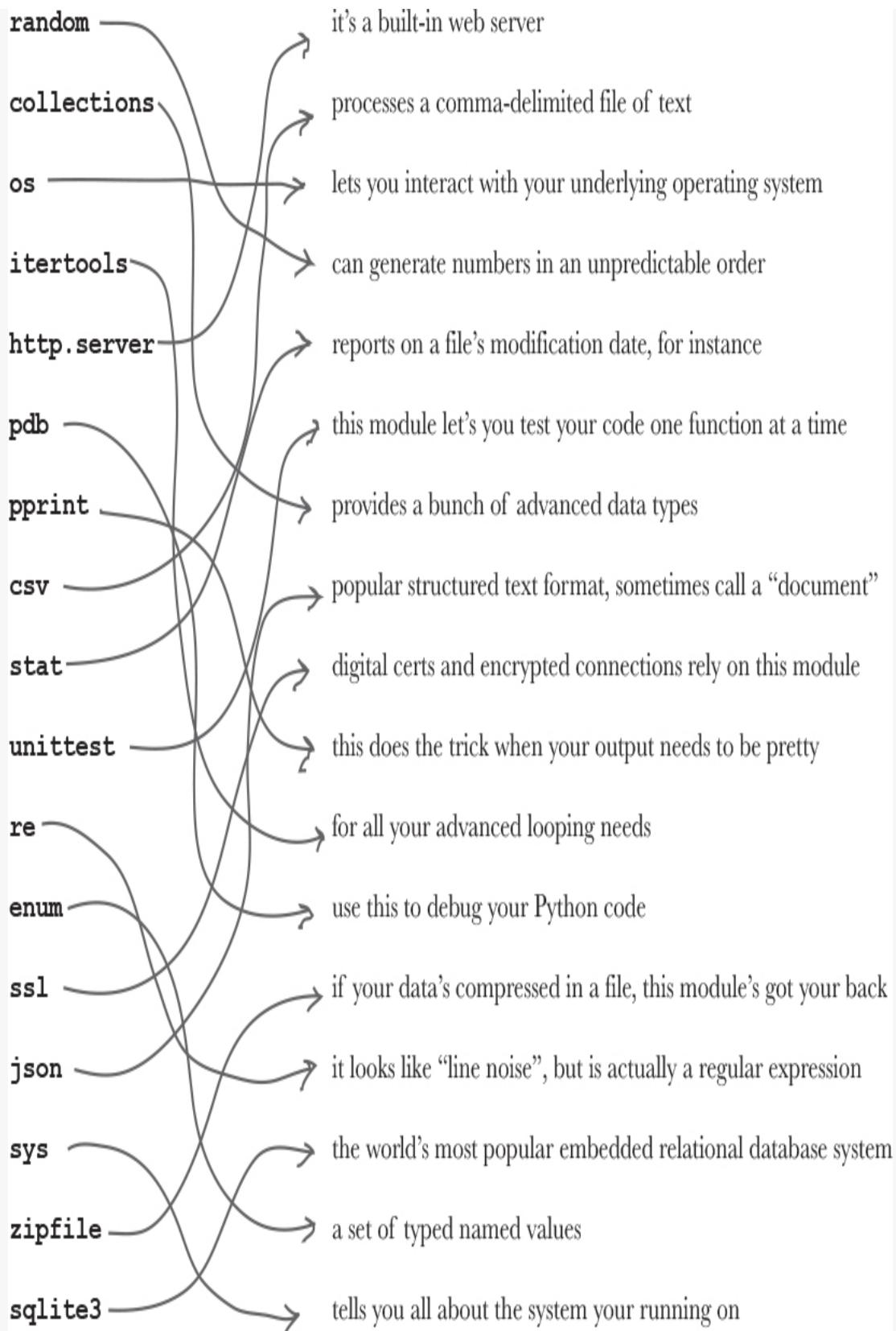
## **WHO DOES WHAT?**

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

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

## **WHO DOES WHAT? SOLUTION**

We know you've yet to look at the PSL in any great detail but, to give you a taste of what's included, we've devised a little test. Without taking a peek at the documentation referred to earlier, 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 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.

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

### NOTE

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

#### 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 (and you'll also find resources in the appendices for further exploring the PSL on your own).

#### ➋ The PSL represents a large body of tested code which you don't have to write, just use

As the PSL has existed for decades now, the modules it contains have been tested to destruction by legions of Python programmers *all over the globe*. Consequently, you can use PSL modules with confidence.

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

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

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

## TEST DRIVE



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



The PSL's **collections** module is also very popular, and contains a bunch of containerised data types. A particular favorite is the **Counter** class which automates frequency counting:

Import the module.

```
import collections
```

```
how_many = "How many f's are in this string? ffffffff"
```

A string which contains lots of "f" characters.

```
c = collections.Counter(how_many)
```

```
c["f"]
```

11

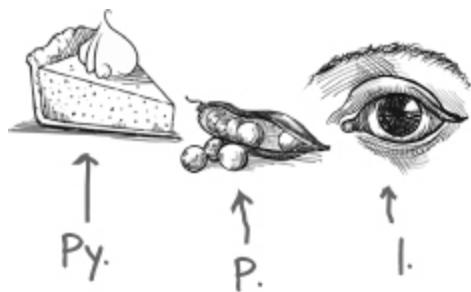
How many "f" characters?  
There's your answer.

Create a new Counter object from the characters in the "how\_many" string.

## 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 centralised repository of third-party modules, classes, and packages. It's called the *The Python Package Index* and lives here: <https://pypi.org/>.

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

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



[Help](#)   [Sponsors](#)   [Log in](#)   [Register](#)

# Find, install and publish Python packages with the Python Package Index

Search projects



[Or browse projects](#)

441,290 projects

4,186,105 releases

7,591,142 files

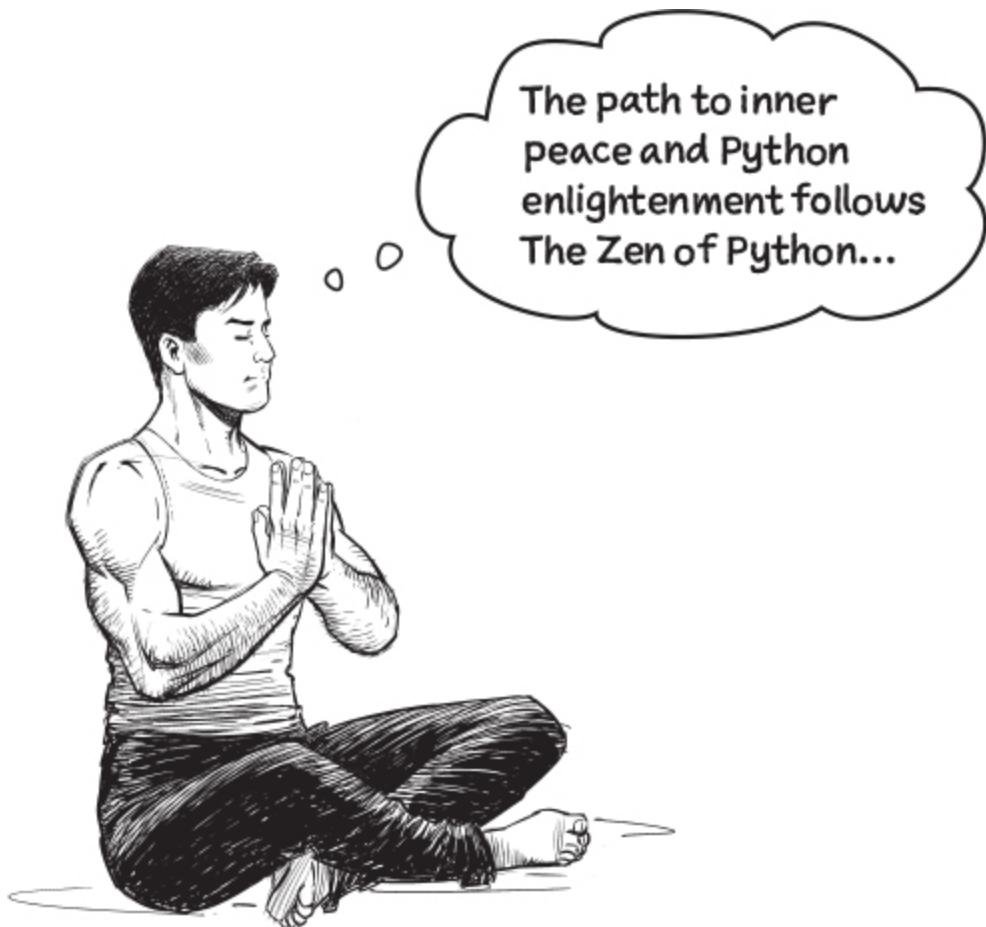
670,414 users



The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages](#).

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI](#).

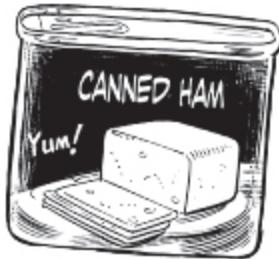


## OK. If it works for you, sure, follow your Zen!

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

```
import this
```

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.

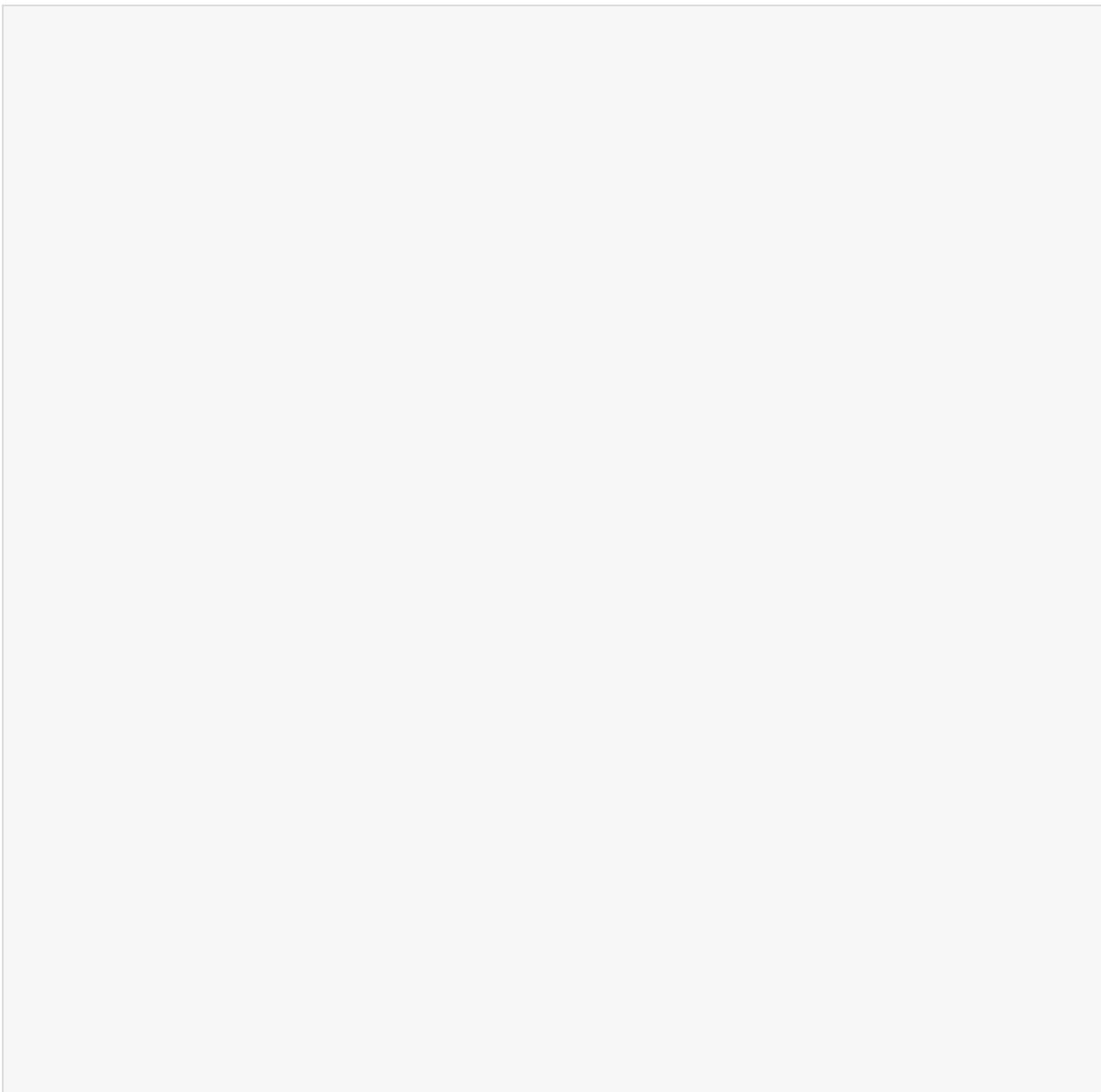


And we can help you with that.

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

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

See you in the next chapter, [Chapter 2](#), which is actually your *second* chapter as we starting counting from zero (just like Python).



## CHAPTER REVIEW BULLET POINTS

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

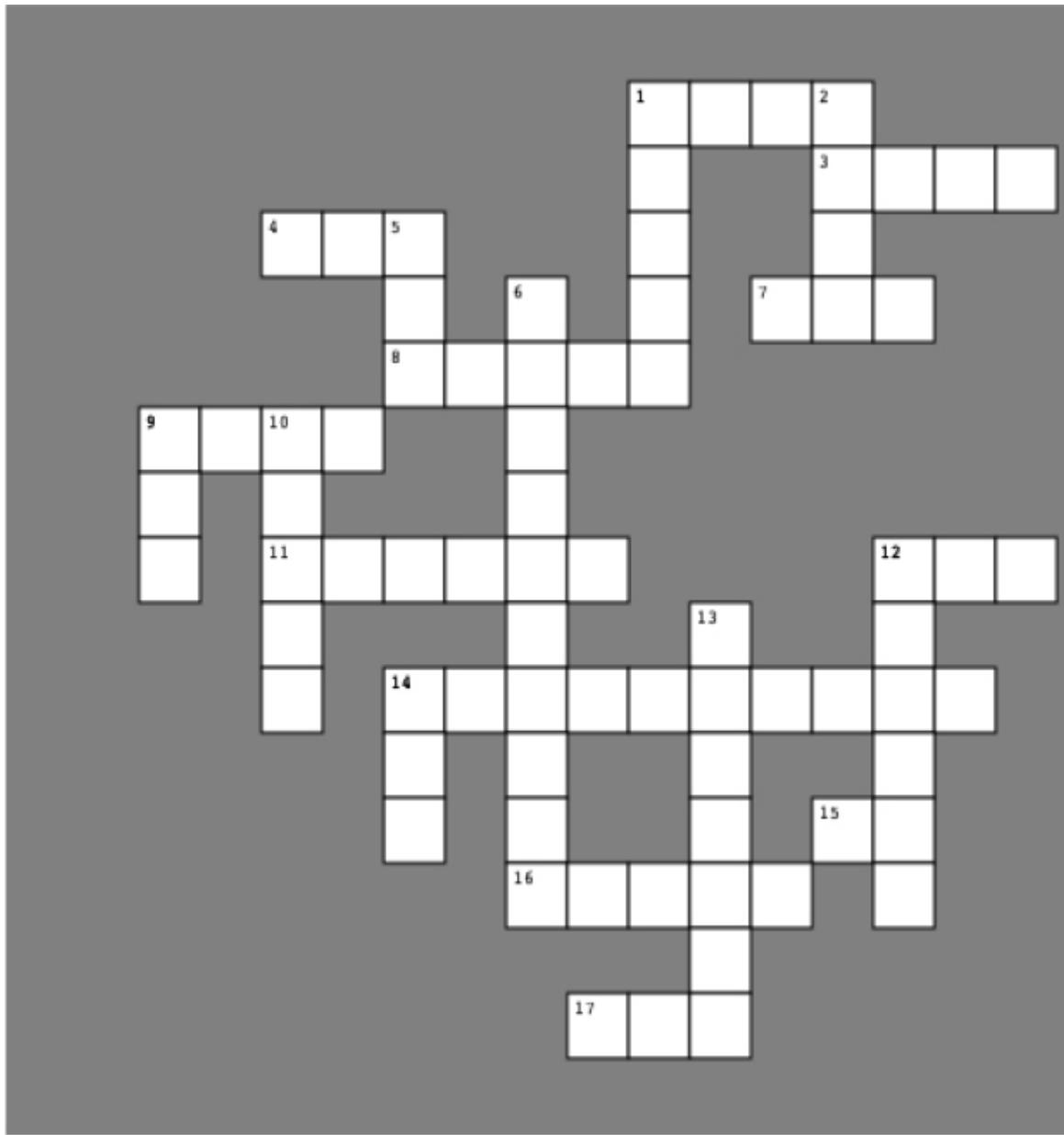
- A common idiom is to combine (or **chain**) the **dir** BIF with the **print** BIF creating (what we like to refer to as) the **print dir combo mambo**.
- Some of the attributes shown by the combo mambo refer to **methods** which can be applied to the object, for example `deck.remove`.
- Attributes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

- Python has two built-in **boolean** values: `True` and `False`.
- The **PSL** has nothing to do with coffee, but everything to do with the **Python Standard Library**. The PSL is a large collection of built-in modules (which come with Python) and can be used all over the place to do many useful things.
- If the PSL isn't enough for you, check out **PyPI**, the **Python Package Index**, an online repository of shareable Python modules. It's often the case some of the code you need has already been written and uploaded to PyPI as a shareable module. Feel free to "leverage" as needed.
- There are other useful **keyboard shortcuts** which you can use when working within Jupyter. Our nine essential shortcuts are coming up after this chapter's crossword solution (in three pages time).

## The Card Deck Crossword



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



## Across

2. A built-in function which tells you what something is.
3. Objects surrounded by [ and ].
4. Shorthand for built-in function.
7. Reports on an object's size.
8. Generates collection of numbers.

9. The Python Package Index.
11. Includes a module in your code.
12. Objects surrounded by { and }.
14. This chapter's missing Big 4.
15. This operator can find things.
16. Use together with Shift to run.
17. Enlightenment, Python-style.

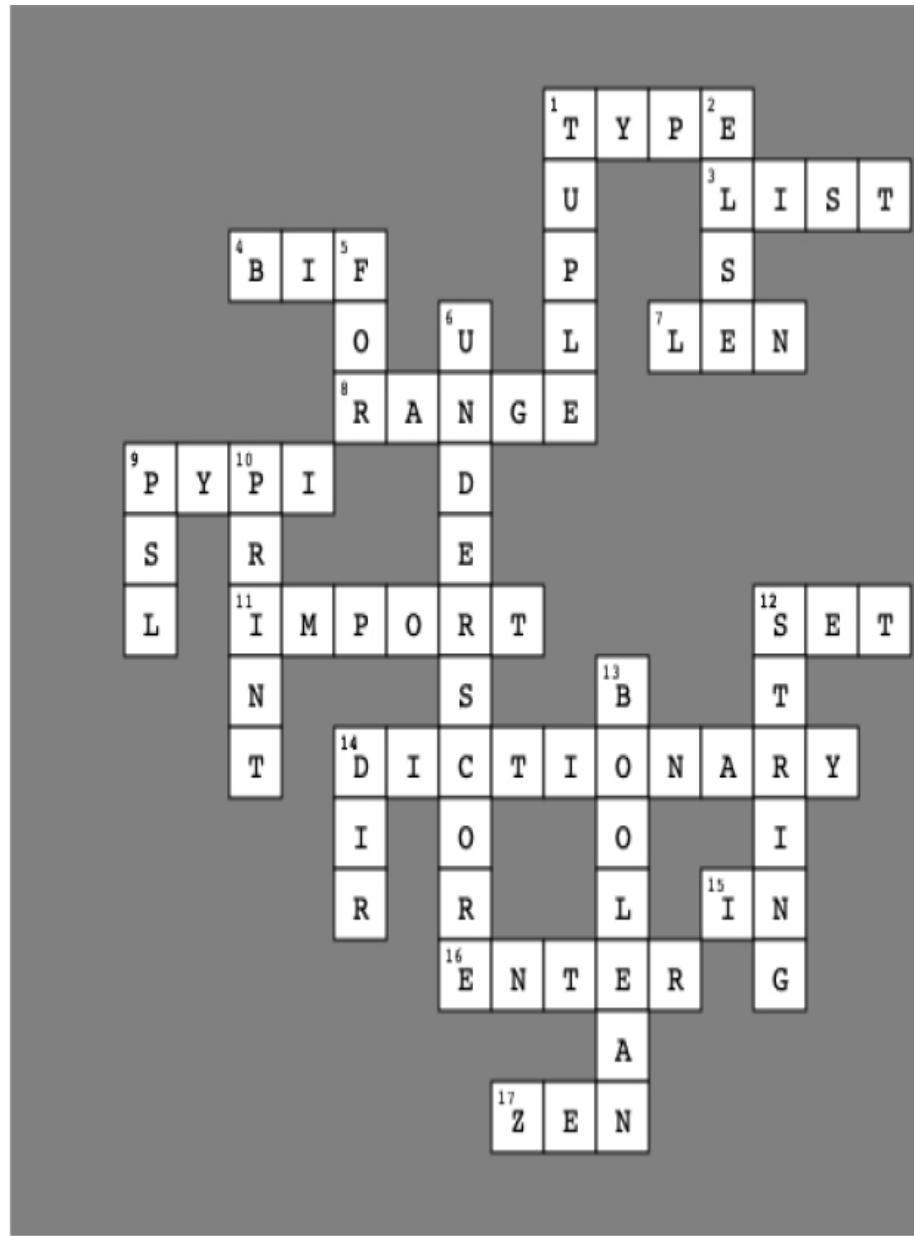
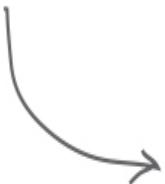
## Down

1. Objects surrounded by ( and ), and it is one of the Big 4, too.
2. The optional part of an `if` statement.
5. Loops a specific number of times.
6. The name given to Python's default variable.
9. It's not a *Pumpkin Skinny Latte*, but shares the same acronym
10. Displays to screen.
12. It's what the `card` variable is.
13. It's either `True` or `False`.
14. Makes up the *combo mambo*.

## The Card Deck Crossword Solution



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



## Across

2. A built-in function which tells you what something is.
3. Objects surrounded by [ and ].
4. Shorthand for built-in function.
7. Reports on an object's size.
8. Generates collection of numbers.

9. The Python Package Index.
11. Includes a module in your code.
12. Objects surrounded by { and }.
14. This chapter's missing Big 4.
15. This operator can find things.
16. Use together with Shift to run.
17. Enlightenment, Python-style.

## Down

1. Objects surrounded by ( and ), and it is one of the Big 4, too.
2. The optional part of an `if` statement.
5. Loops a specific number of times.
6. The name given to Python's default variable.
9. It's not a *Pumpkin Skinny Latte*, but shares the same acronym
10. Displays to screen.
12. It's what the `card` variable is.
13. It's either `True` or `False`.
14. Makes up the *combo mambo*.

## Just when you thought you were done...

Go grab your scissors, as here's a handy cut-out chart of the Jupyter Notebook keyboard shortcuts we view as *essential*. You'll get to use all of these 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 which currently has the focus.
Esc then V	Paste a previously copied (or cut) cell below the currently focused cell.
Esc then X	Cut the currently focused cell from the notebook.
Z	Undo the last cut (there's no need to press the ESC key here).

## **NOTE**

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

# Chapter 2. Diving in: *Let's Make a Splash*

---

## A NOTE FOR EARLY RELEASE READERS

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

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

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



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

Hi. I'm told you're the person to talk to about coding? I'm currently struggling with my Swimmer Performance Analysis System, and I wonder if you might be up for helping me replace it with something that takes less time? I can't grow my club membership until I sort out this issue...



## This sounds interesting.

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

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

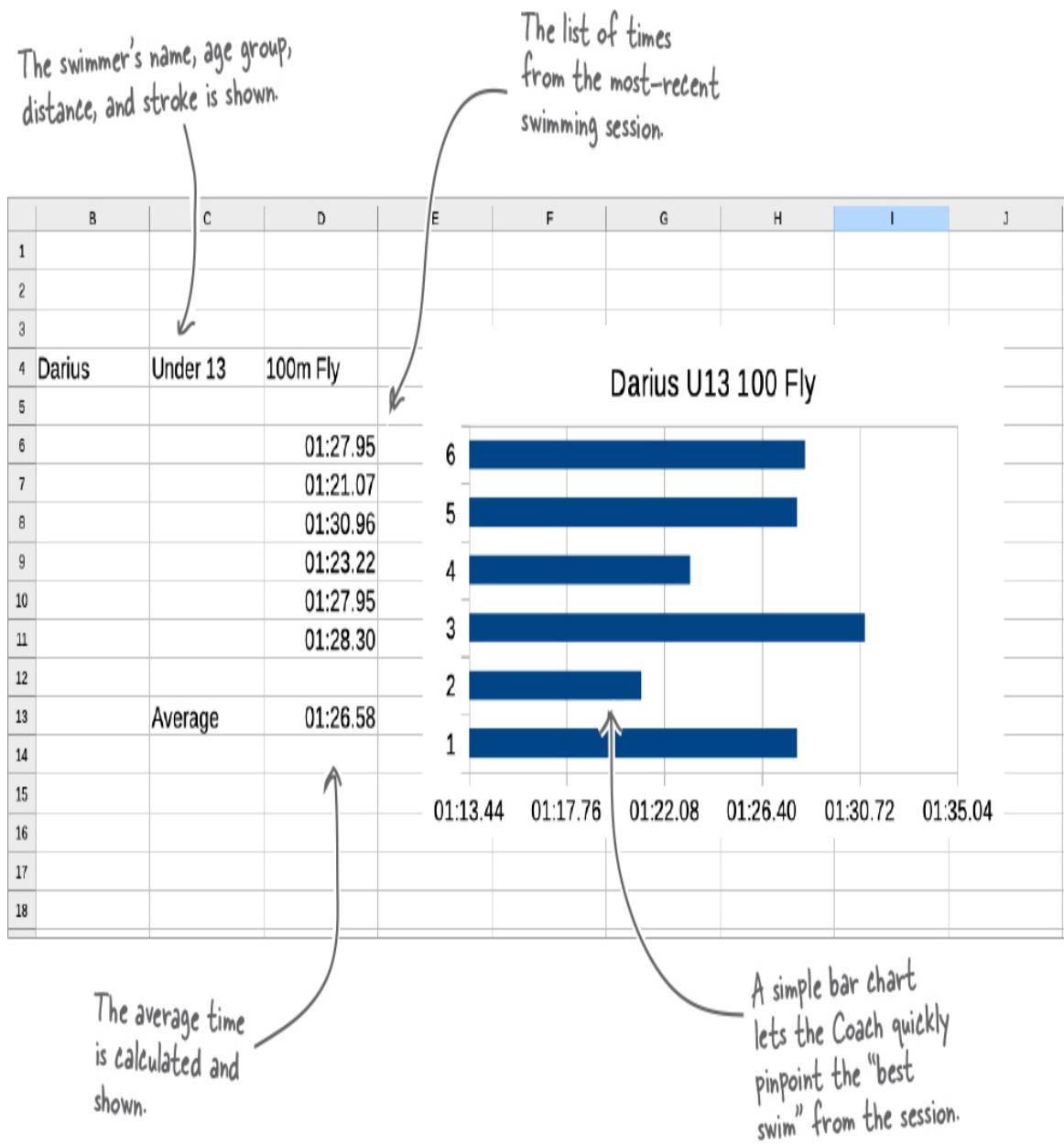
For now, this analysis is straightforward. The times for the session are averaged, and a simple bar chart allows for a quick visual check of the swimmer's performance. The Coach readily admits to being a computer neophyte, stating that he's much better at Coaching young swimmers than "mucking about with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



## How do things work right now?

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

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



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

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

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

## Fortune decides to smile on you

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



The filename encodes the swimmer's name, their age group, the distance swam, and the stroke.

This is the data file as shown in VS Code.

Note: this is a text file, so those times look like numbers but aren't. They are a sequence of characters.

```
☰ 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
```

For some reason, this second line is always blank. Probably safe to ignore it.

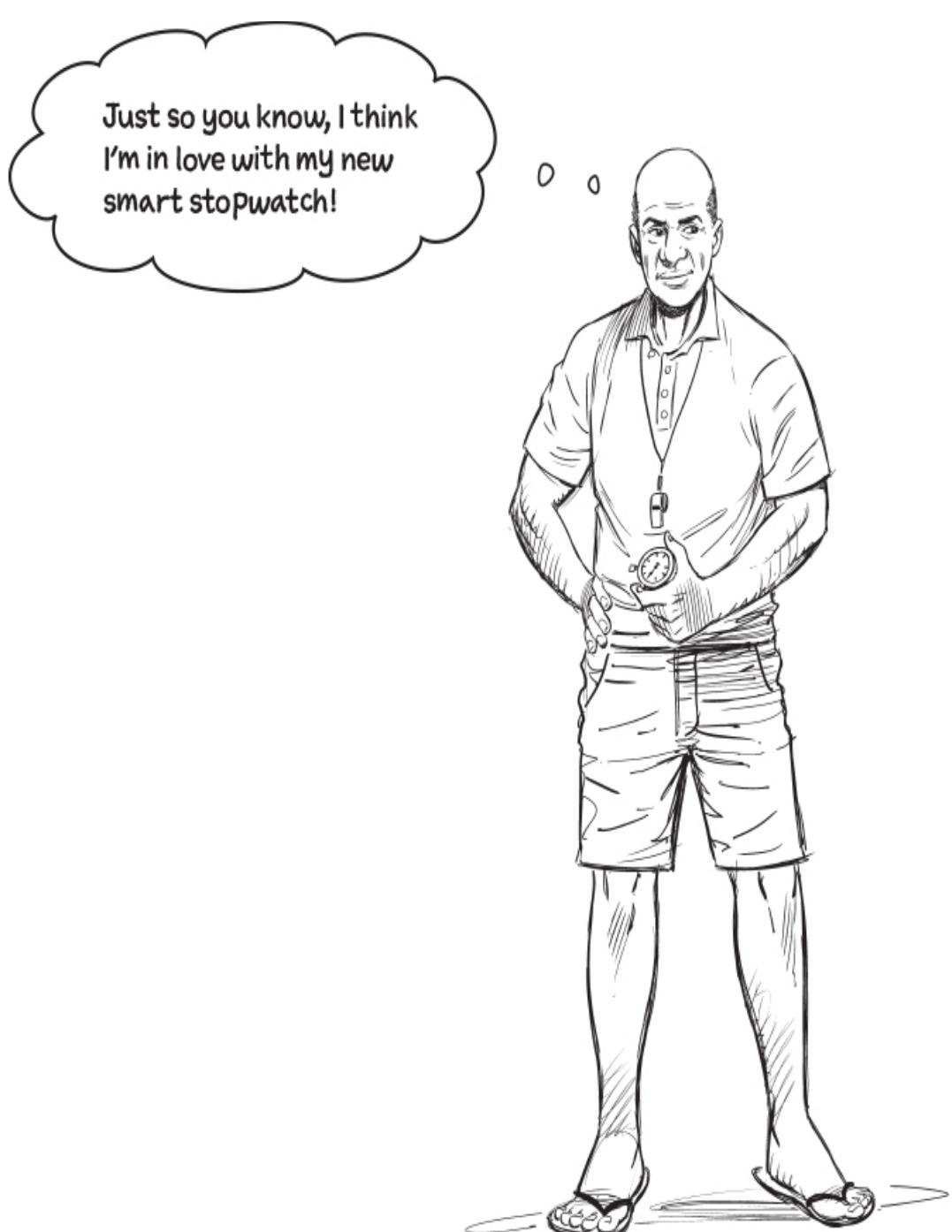
The first line in the file shows all the recorded times, separated by commas.

## 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.*



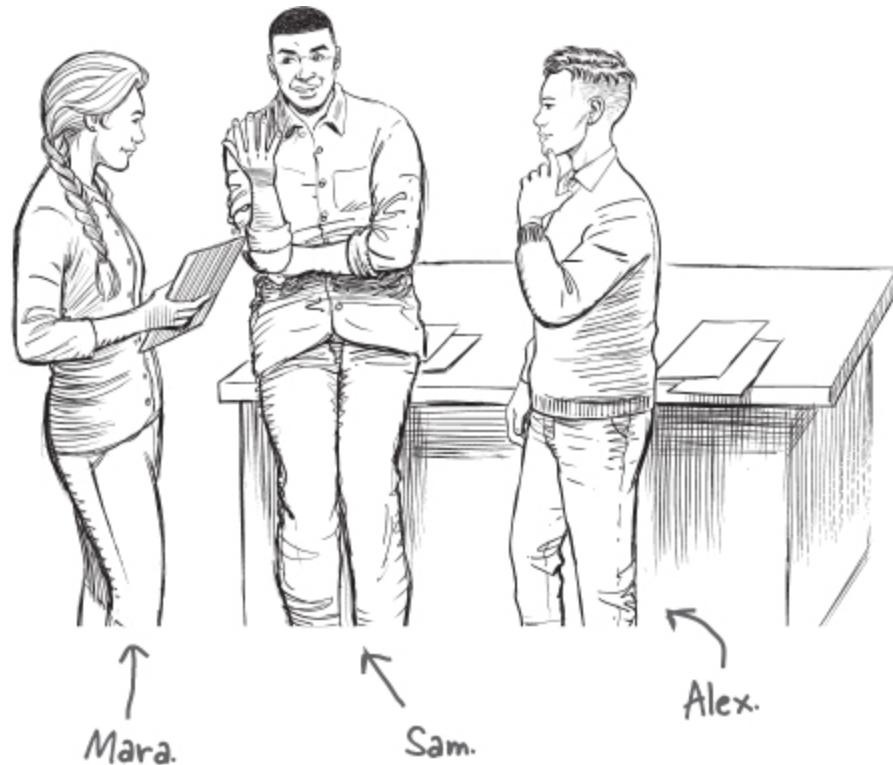
Just so you know, I think  
I'm in love with my new  
smart stopwatch!

We may have created a  
monster. The Coach is  
so excited by his new  
watch that he's using  
it for his very next  
training session.

**The previous page's data looks promising.**

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

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



## Cubicle Conversation

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

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

**Alex:** How so?

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

**Mara:** What do you mean by “converted”?

**Alex:** That was my question, too.

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

**Alex:** I’ll add “conversion” to the to-do list.

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

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

**Alex:** That’s assuming we can read the filename?

**Sam:** Isn’t that a given?

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

**Alex:** This is getting a little complex...

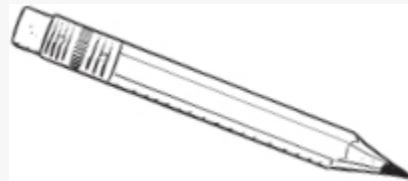
**Sam:** Not if we take things bit-by-bit.

**Mara:** We just need a plan of action.

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

**Sam:** I can recommend a great book... 😊

## SHARPEN YOUR PENCIL



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

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

### ① Extract data from the file's name

---

---

---

---

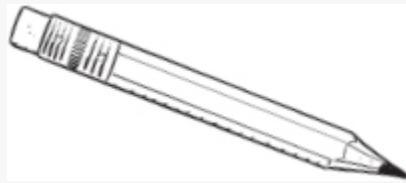
---

---

### ② Process the data in the file



## SHARPEN YOUR PENCIL SOLUTION



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

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

### ① Extract data from the file's name

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

### ② Process the data in the file

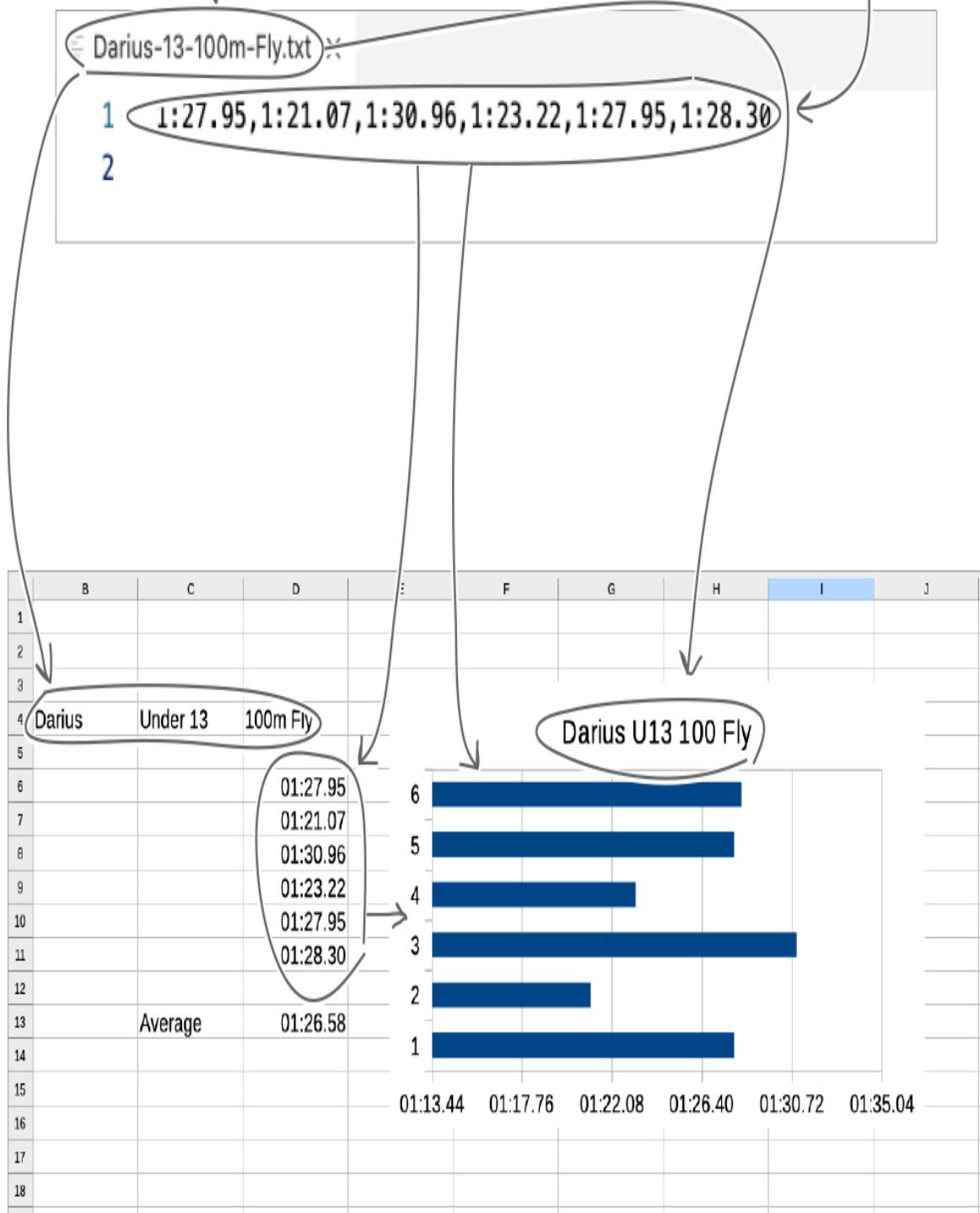
- a. Read the lines from the file
- b. Ignore the second line
- c. Break the first line apart by “,” to produce a list of times
- d. Take each of the times and convert them to a number from the “mins:secs.hundredths” format
- e. Calculate the average time, then convert it back to the “mins:secs.hundredths”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

## **The file and the spreadsheet are “related”**

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

The data embedded in the file's name appears twice in the spreadsheet (as header information).

The file's data appears in the spreadsheet twice: once in the column (which supports the average calculation) and, again, in the bar chart (where the time values correspond to the width of the bar).



## Our first task: Extract the filename's data

For now, the plan is to concentrate on one file, specifically the file which contains the data for the 100m Fly times for Darius, who is swimming in the under 13 age group.

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

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

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



### A string is not really a string...

The value to the right of the assignment operator (=) in the above line of code certainly *looks* like a string. After all, it's a sequence of characters

enclosed within quotes which, in most other programming languages, is the very *definition* of a string. This is not the case in Python.

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

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

*In Python, everything is an object.*



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

## A string is an object with attributes

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

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', 'casifold', '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']
```



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.



**RELAX**



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

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

## THERE ARE NO DUMB QUESTIONS

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

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

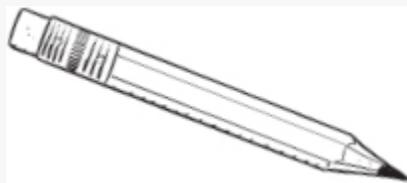
**Q: Is there a way I can learn more about what a particular method does?**

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

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

**A:** Yes, it is. Most Python programmers shorten “double-underscore add double-underscore” to simply “dunder add”. So, if you hear someone refer to 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()` \_\_\_\_\_  
\_\_\_\_\_



**No problem. Great question, by the way.**

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

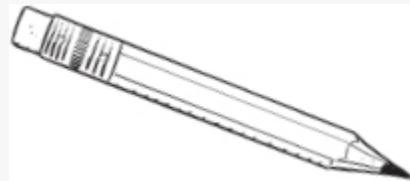
This is a little different from the BIFs which are invoked like functions. For instance, `len(fn)` returns the size of the object referred to by the `fn`

variable.

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

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

## SHARPEN YOUR PENCIL SOLUTION



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

`fn.upper()`

Return a copy of the value of what "fn"

refers to in all UPPERCASE lettering.

`fn.lower()`

Return a copy of the value of what "fn"

refers to in all lowercase lettering.

Here's what we saw on screen:



```
fn.upper()
```

Remember  
to press  
"Shift+Enter"  
to run each  
cell.



The "upper"  
method returns a → 'DARIUS-13-100M-FLY.TXT'  
copy of the string  
in all UPPERCASE.

```
fn.lower()
```

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

'darius-13-100m-fly.txt' ←

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

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

# Extract the swimmer's data from the filename

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

a. Read the filename



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

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

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

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

```
'capitalize', 'casefold', 'center', 'count', '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'
```

## NOTE

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



**Sounds interesting.**

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

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

Read the method's documentation!

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

Regrettably, Python can't read the documentation for you, so you'll still have to do that bit yourself. But the `help` BIF 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.

```
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 to 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 part of the docstring discusses what the parameters mean. Note, with this method, both parameters have default values.

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.

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

## TEST DRIVE



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

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

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

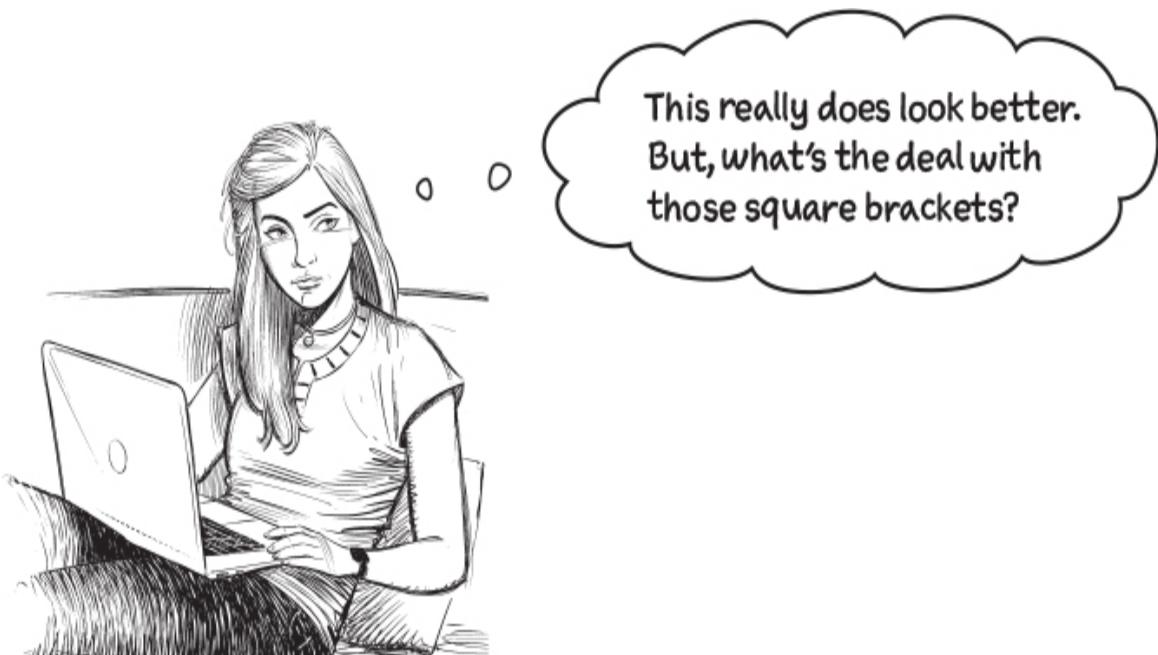
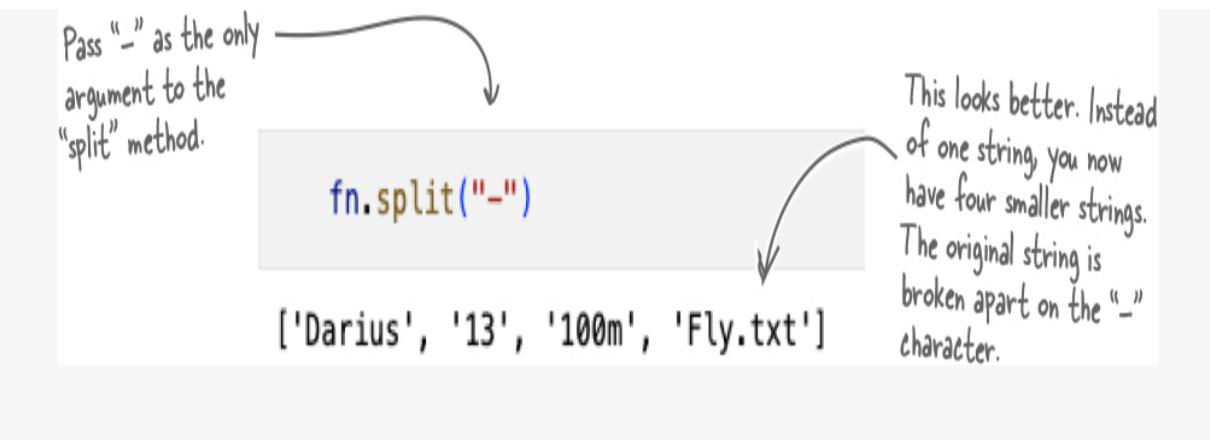
`fn.split()`

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

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

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

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



You did read the `“split”` method’s documentation, didn’t you? The answer’s right there...

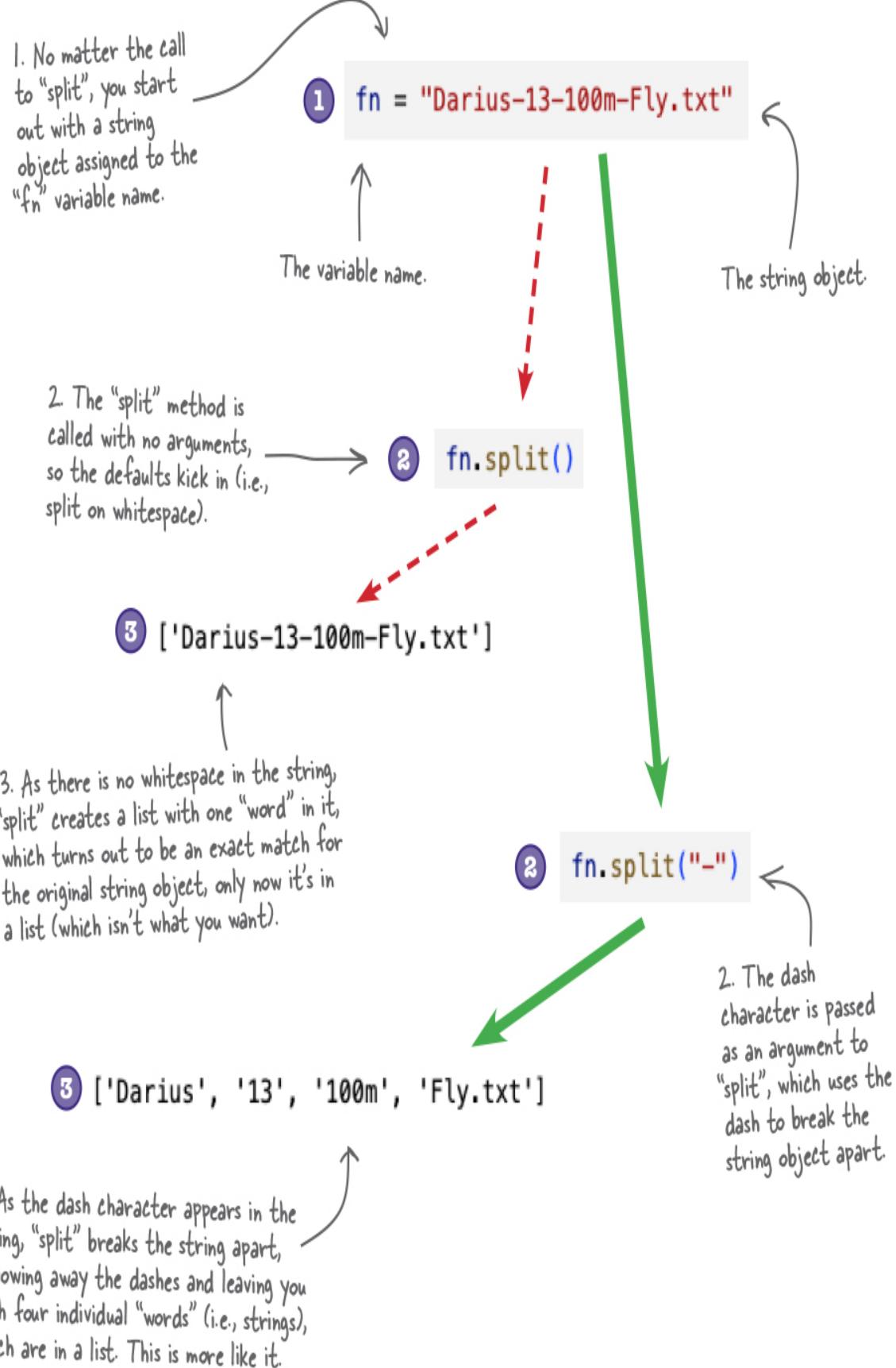
## The `split` method returns a list of words...

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

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

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

easy as 1-2-3:



## Is it time for another checkmark?

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

a. Read the filename ✓

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

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

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

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

['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?



**Emmm, maybe...**

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

## EXERCISE



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

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

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

1

fn.split("13")

---

---

2

fn.split("-1")

---

---

3

fn.split(",")

---

---

4

fn.split("-.")

---

---

5

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

---

---



## EXERCISE SOLUTION



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

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

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("-1")`

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

The string is split in  
three this time, as "-1"

appears twice in the  
original string.

→ ['Darius', '3', '00m-Fly.txt'] ←



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

As with all calls to  
"split", the output 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.

→ ['Darius-13-100m-Fly', 'txt']

4

`fn.split("-.")`

Break the string apart based on where “-.” is found, i.e,

where the character “-” and the “.” appear together.

`['Darius-13-100m-Fly.txt']` ←

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

5

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

Break the string apart based on where “-” is found, then

Yikes!

do another split on the “.” character (i.e., dot)...??

← Nope. It doesn't do this!

`AttributeError`

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

Traceback (most recent call last)

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



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

## Read error message from the bottom-up

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

```
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 arrow indicates where in your code the error occurred.

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

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 chaining method calls**

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

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

As Python techniques go,  
chaining method calls like this  
is allowed, but care is needed.

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 here?



I think I see what's going on here. The first method on the chain splits the original string on “-” creating a list, which is then passed to the second method on the chain, “split”, which is expecting a string, but got a list.  
Right?

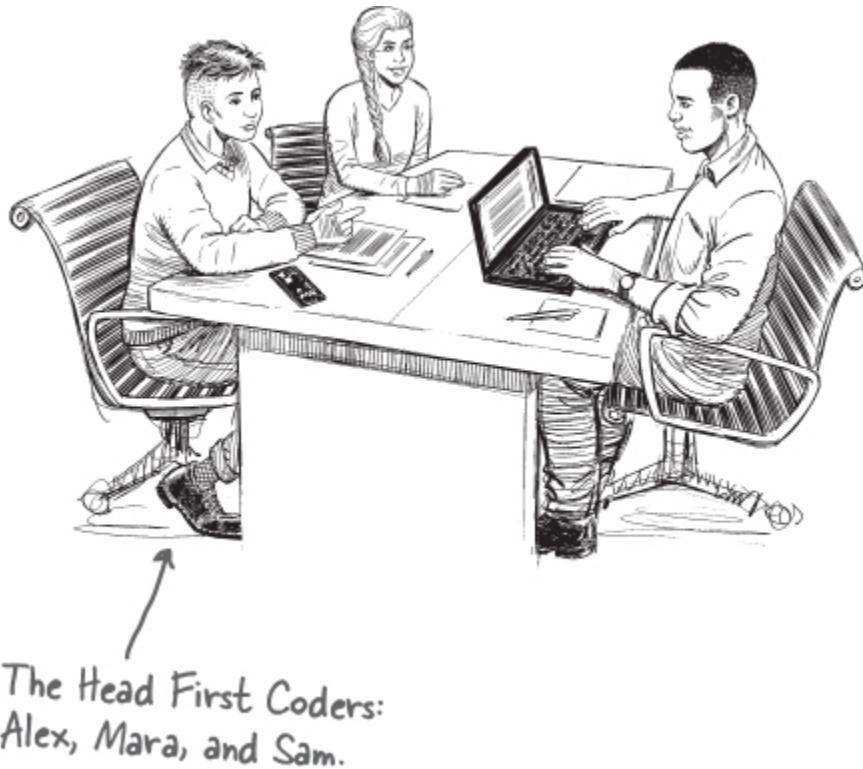
### Yes, that's exactly what's happening.

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?

# Fixing broken chains

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



## Cubicle Conversation

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

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

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

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

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

**Alex:** So, we're out of luck?

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

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

**Alex:** How do I do that again?

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

## BRAIN POWER



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

```
'capitalize', 'casefold', 'center', 'count', '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'
```

## Strings can do more than just split

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



`help(fn.rstrip)`

This isn't  
really  
what you  
want.

Help on built-in function `rstrip`:

`rstrip(chars=None, /)` method of `builtins.str` instance

Return a copy of the string with trailing whitespace removed.

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

But, this statement shows a bit more promise.

## TEST DRIVE



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

Things start off fine, as your first three tests produce exactly what you expect: The ".txt" extension is gone. Result!

```
"Fly.txt".rstrip(".txt")
```

'Fly'

Yes, you can invoke a method against a literal string. The string does not have to be referred to by a variable.

```
"Free.txt".rstrip(".txt")
```

'Free'

```
"Back.txt".rstrip(".txt")
```

'Back'

The "rstrip" method removes the characters specified from the RIGHT of the string, assuming they are found. As the "t" at the end of "Breast" is in ".txt", it is removed too.

But look at this. The "t" at the end of the word "Breast" is gone too. Bummer!

```
"Breast.txt".rstrip(".txt")
```

'Breas'

## Let's try another string method

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

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

This one has an interesting name.

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

```
help(fn.removeprefix)
```

Help on built-in function removeprefix:

This sounds  
more like it.

removeprefix(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.

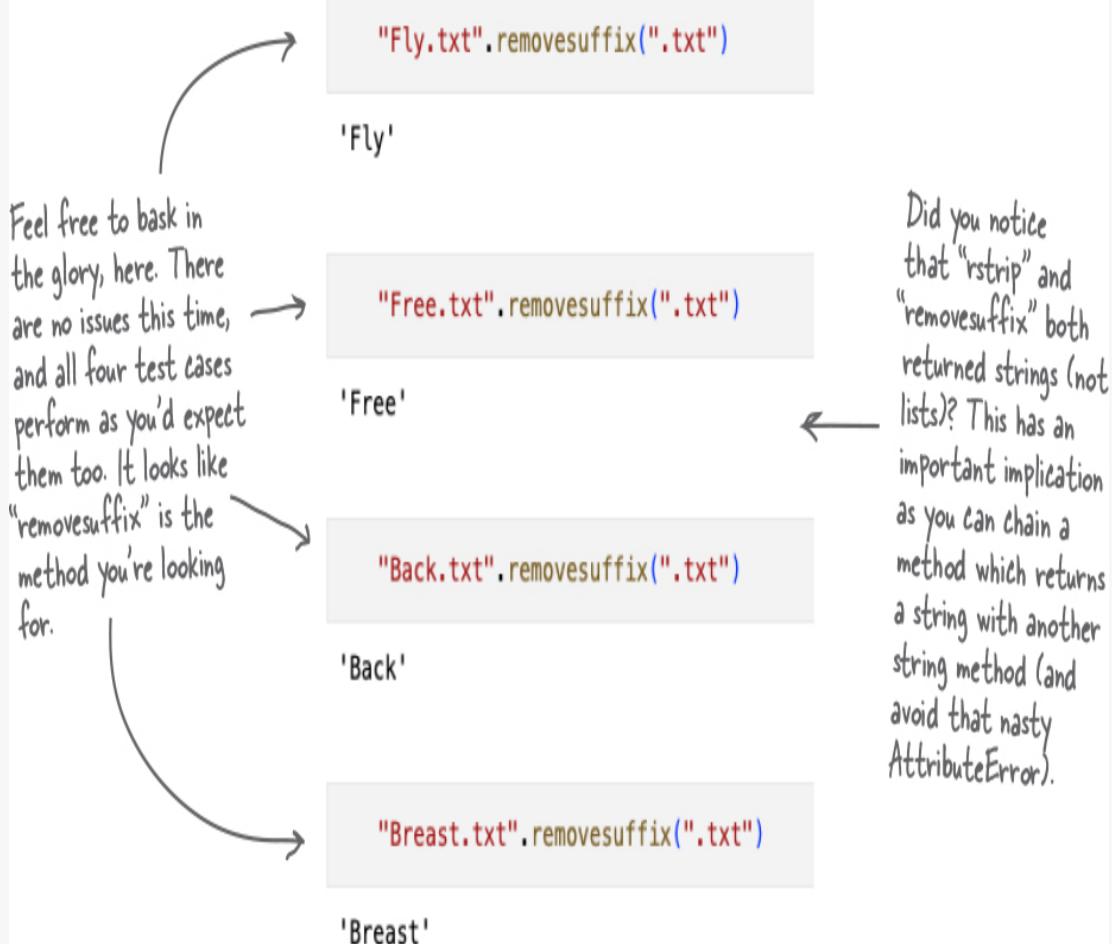
This part of the "removeprefix" method's documentation reads a bit like gobbledegook, especially that "[::-len(suffix)]" bit, right? For now, don't worry about what this paragraph means. You'll learn how this notation works in a later chapter (when, hopefully, these does will then make sense). And, anyway, 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



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



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

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

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

```
['Darius', '13', '100m', 'Fly'] ← This is looking good!
```

a. Read the filename ✓

It's time  
for a  
checkmark. →

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

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

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

into variables (so they can be used later)

## THERE ARE NO DUMB QUESTIONS

**Q:** Can method chains be of any length?

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

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

**A:** The short answer is no. The longer answer is also no, in that you cannot change a string object in Python once it's defined. Strings in Python are immutable (they cannot mutate or change once they exist). This behavior is by design, and sometimes strikes programmers from other languages as strange. Python treats strings as a basic data type, like numbers. Numbers are also immutable. Once 42 exists in your code it cannot be mutated nor changed. It's always 42. Same thing with strings. If a string has the value "Marvin", it'll remain that value until one of two things happen: (1) your code terminates or (2) the Universe ends.

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

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

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

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

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

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

## All that remains is to create some variables

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

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

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

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

The use of square brackets is  
your clue that this is a list. → ['Darius', '13', '100m', 'Fly']

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

If Python's lists really are like the  
arrays found in other programming  
languages, then surely they support  
the square bracket notation?



**Indeed they do.**

When working with lists, it is possible to use the familiar square bracket notation. And, as in most other programming languages, Python starts counting from zero, so [0] refers to the first element in the list, [1] the second, [2] the third, and so on.

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

### NOTE

Don't forget to follow along!

## TEST DRIVE



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

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

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

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

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

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

`parts`

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

Although it's a bit of an oversimplification, you can often think of Python lists as being like arrays... so, knowing this, you can use the **square bracket notation** to assign each individual data value to its own variable:

Four new variables  
are created.

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

As expected, the square bracket notation is used to pick out the individual data values from the list. (Remember, like a lot of other languages, Python counts from zero).

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

Type a variable name into an empty cell, press Shift+Enter, then – Ta da! – the referred-to value appears.

swimmer

'Darius'

We know. It's almost too exciting...

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

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

a. Read the filename



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



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

into variables (so they can be used later)



Is this  
checkmark  
premature?

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

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

Put the filename  
in a variable.



Break the  
filename apart.



Extract the  
relevant data items.



```
age = parts[1]
```

```
distance = parts[2]
```

```
stroke = parts[3]
```

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



I'm not trying to throw a spanner  
in the works here, but I wonder if that  
"parts" variable is really needed? Can  
your code do without it?

## The `parts` variable feels kind of integral.

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

So, can you do without `parts`?

### NOTE

And if the “`parts`” variable is not needed, does this mean it’s spare?!? (Sorry).

## Can you do without the `parts` list?

Short answer: yes.

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

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

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

  
This call to “`split`” at the end  
of the chain produces a list.

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

Assign the list to the "parts" name.

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

```
swimmer = parts[0]
```

```
age = parts[1]
```

```
distance = parts[2]
```

```
stroke = parts[3]
```

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

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

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

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

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

## We can think of three reasons

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

### ① The latest code is slow

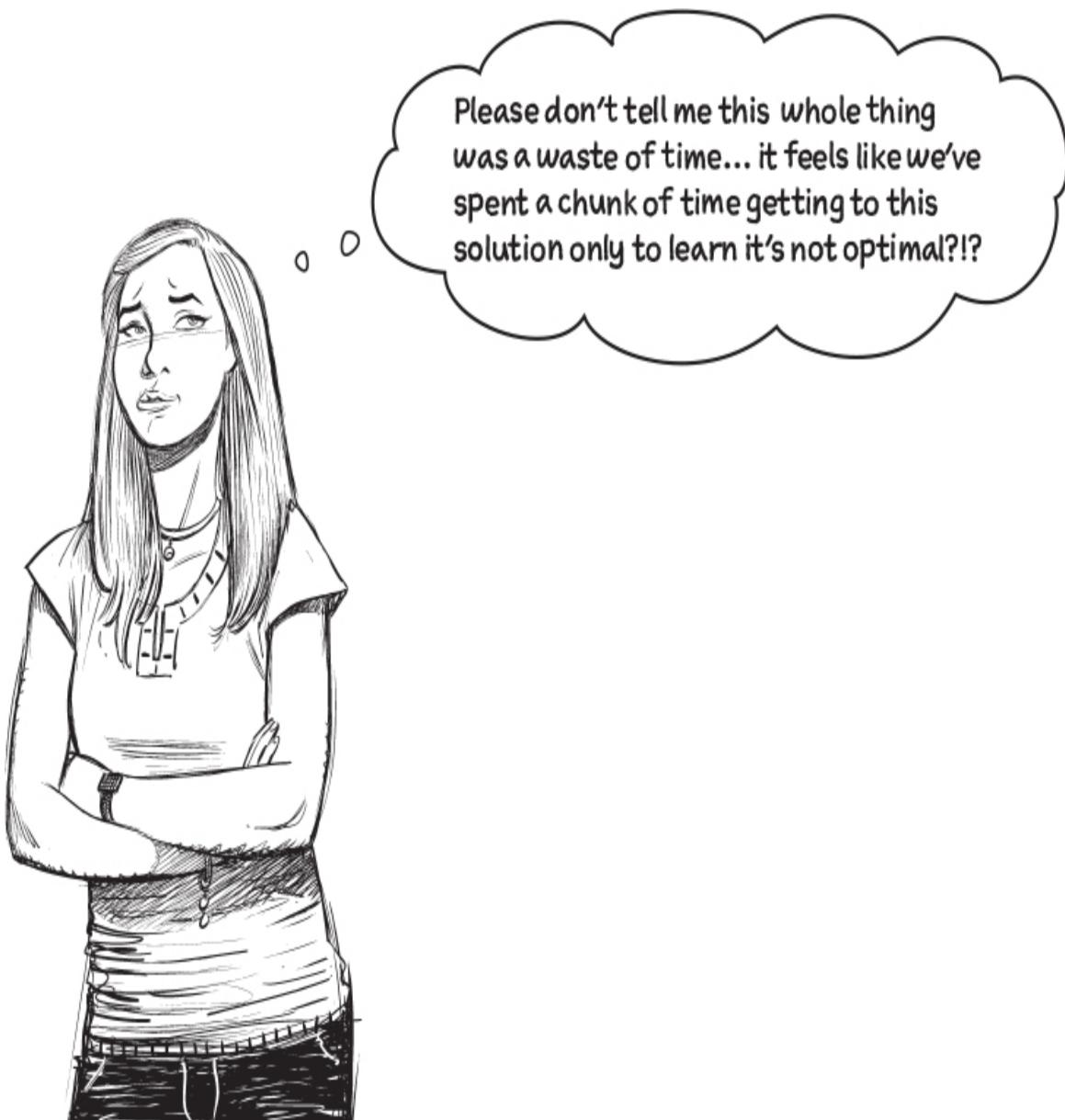
The original code generated the list *once*, assigned it to the **parts** list, then used the list as needed. This is efficient. The latest code generates the list *four times*, which is hugely inefficient.

### ② The latest code is harder to read

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

### ③ **The latest code is a maintenance nightmare**

If you're asked to change the suffix from ".txt" to say, ".py", it's an easy change when working with the original code (as it's a single edit). With the latest code, you have to apply the edit four times (multiple edits) which can be fraught with danger.



### Let's not be too hasty.

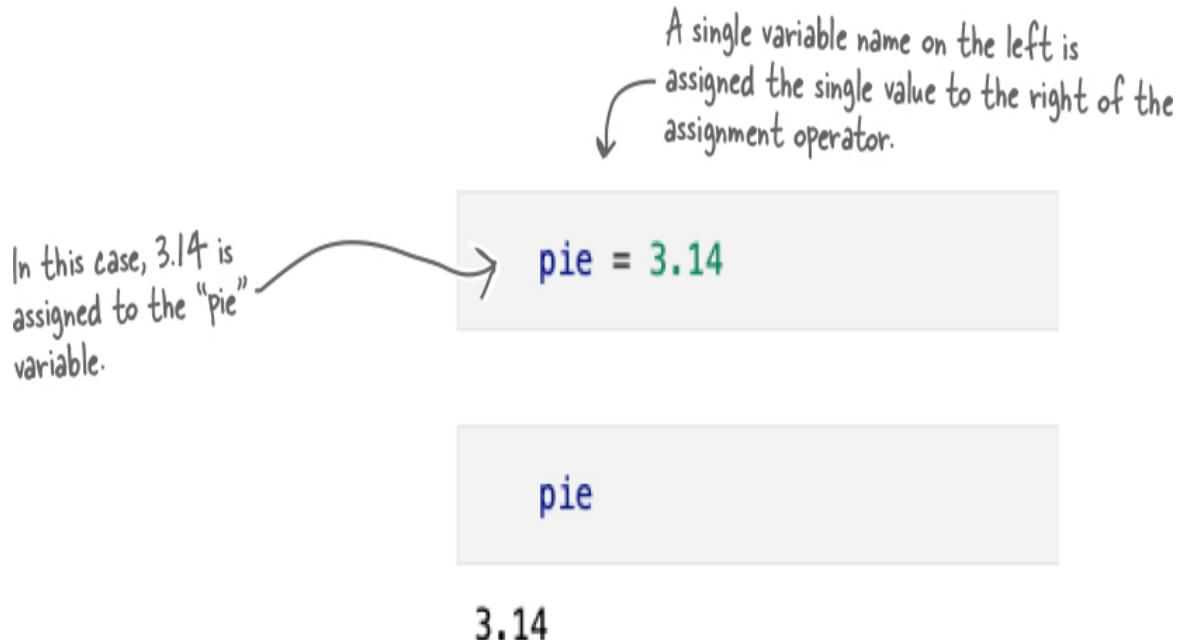
Yes, the latest code is more trouble than it's worth, but the idea of removing the `parts` variable from your code still has merit, as it's of no use once the assignment to `swimmer`, `age`, `distance`, and `stroke` have occurred.

There may be another way to do this...

## Multiple assignment (aka unpacking)

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

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



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



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...

pie

3.14

meaning

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

42

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



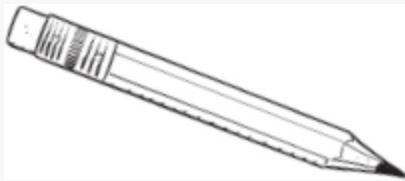
So, let me get this straight. Does this mean if I generate a list on the right of the assignment operator, I can assign the list's data values to an equivalent number of variable names on the left?

## **Yes, that's what this means.**

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

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

## SHARPEN YOUR PENCIL



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

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

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

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

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

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

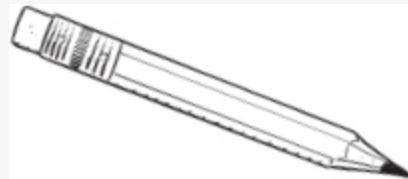
There's a line of code  
missing from here. Add it  
into the space provided.

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

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

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

## SHARPEN YOUR PENCIL SOLUTION



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

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

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

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

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

Darius  
13  
100m  
Fly

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

```
fn = "Abi-10-50m-Back.txt" ← 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. And, did you notice there's not a "parts" list in sight?

## Task #1 is done!

Recall the list of sub-tasks once more:

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

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

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

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



**Yes, they are. We are on our way!**

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

## **Task #2: Process the data in the file**

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

1. Read the lines from the file

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

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



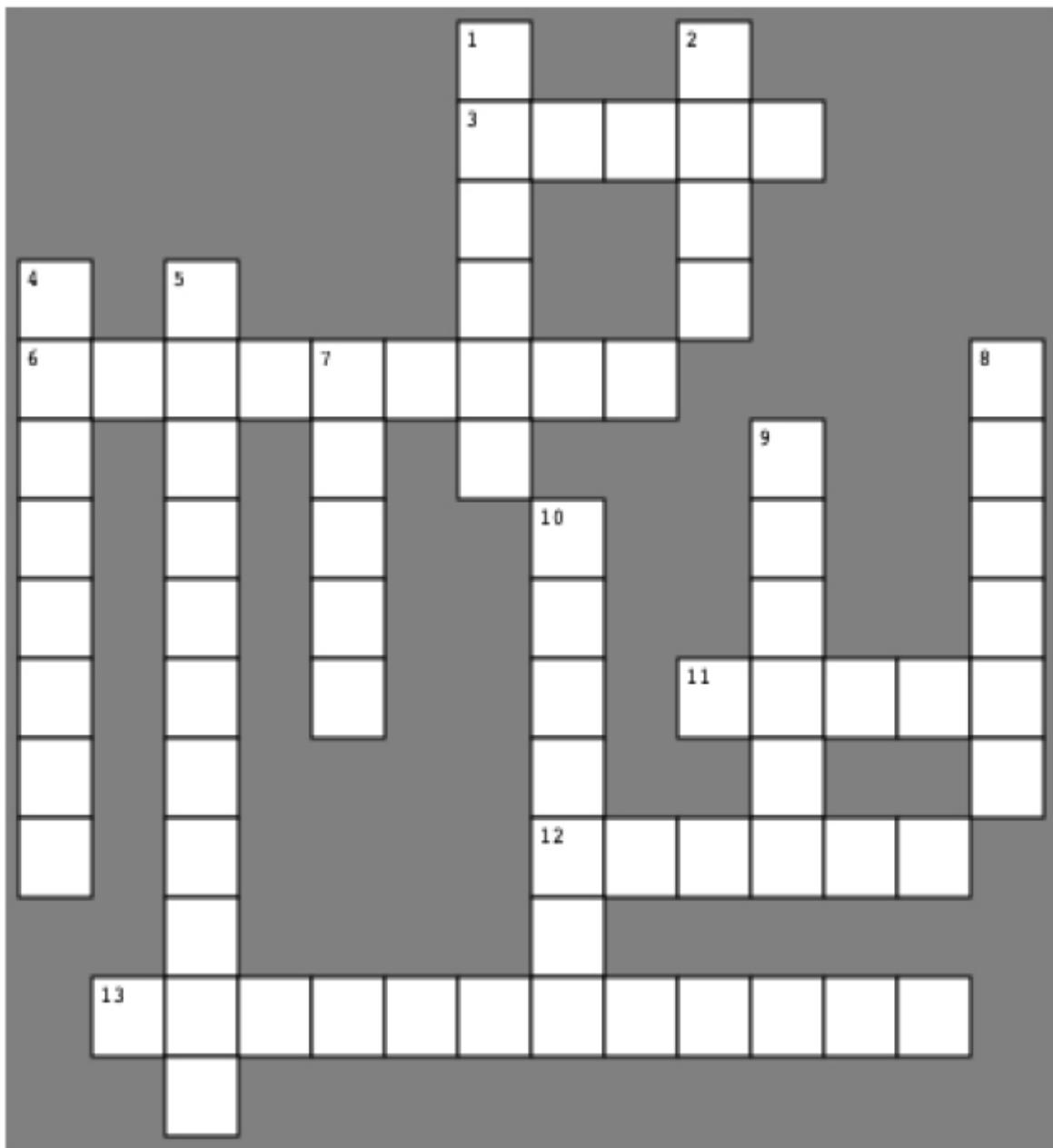
## CHAPTER REVIEW BULLET POINTS

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

## The String Crossword



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



## **Across**

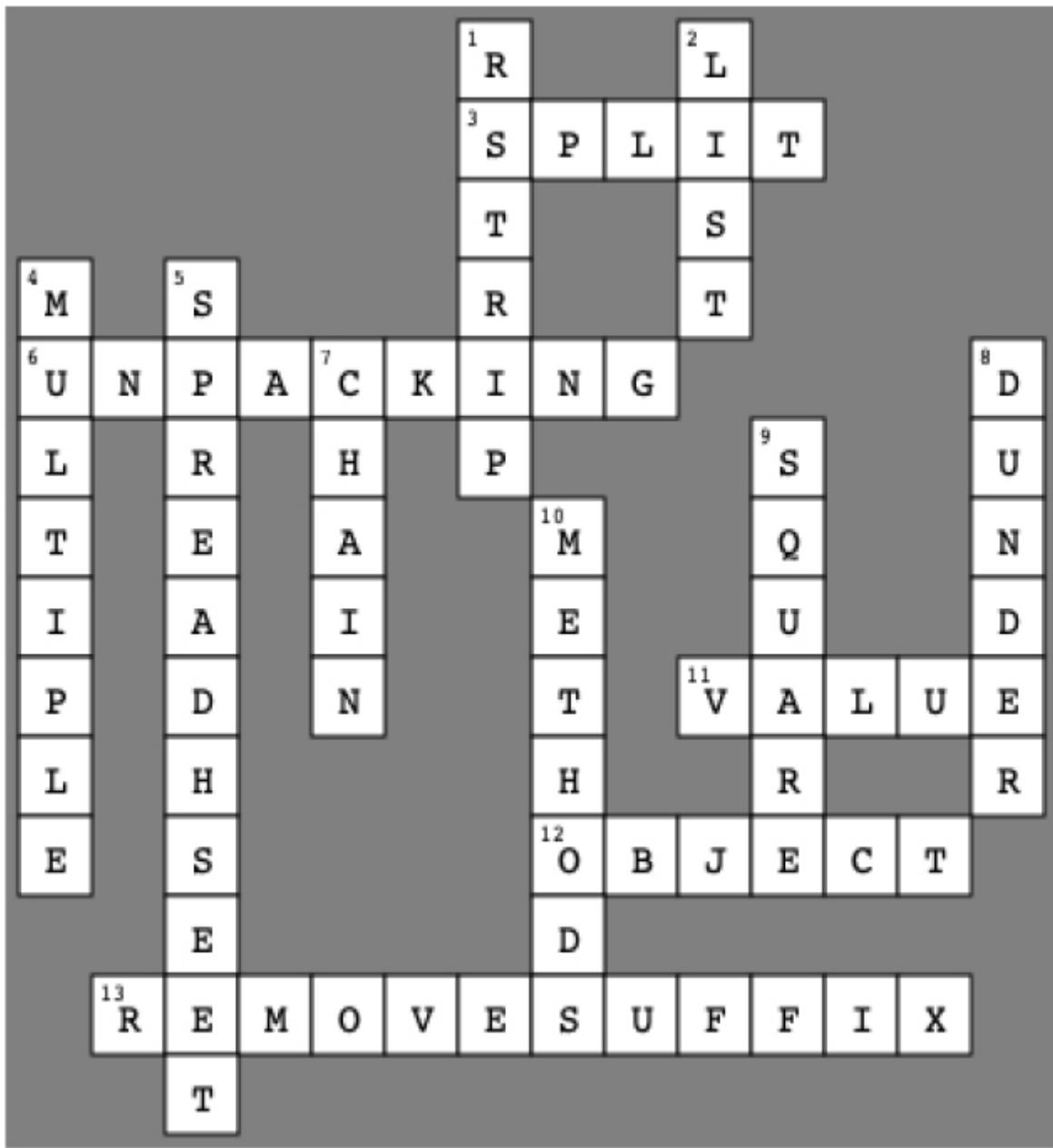
3. Comes in handy when breaking apart strings.
6. Another name for multiple assignment.
11. To be a variable, it has to have one of these.
12. Everything is one of these.
13. Can be used to get rid of a filename's extension.

## **Down**

1. Removes a set of characters from the end of a string.
2. Like an array in other languages.
4. More than one.
5. The swim coach mucks about with one of these.
7. Ball and \_\_\_\_\_.
8. Short for “double underscore”.
9. Our favorite brackets.
10. The plural name given to an object’s built-in functions.

## **The String Crossword Solution**





## Across

- 3. Comes in handy when breaking apart strings.
  - 6. Another name for multiple assignment.
  - 11. To be a variable, it has to have one of these.
  - 12. Everything is one of these.
  - 13. Can be used to get rid of a filename's extension.

## **Down**

1. Removes a set of characters from the end of a string.
2. Like an array in other languages.
4. More than one.
5. The swim coach mucks about with one of these.
7. Ball and \_\_\_\_\_.
8. Short for “double underscore”.
9. Our favorite brackets.
10. The plural name given to an object’s built-in functions.

# Chapter 3. Lists of Numbers: *Processing List Data*

---

## A NOTE FOR EARLY RELEASE READERS

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

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

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



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

make a new best friend (your very own Python **BFF**). You've had your fill of coffee and pie, so it's time to roll up your sleeves and get back to work.

## Task #2: Process the data in the file

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

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



### **It's time to get to work.**

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

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

## **Grab a copy of the Coach's data**

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

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

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

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

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

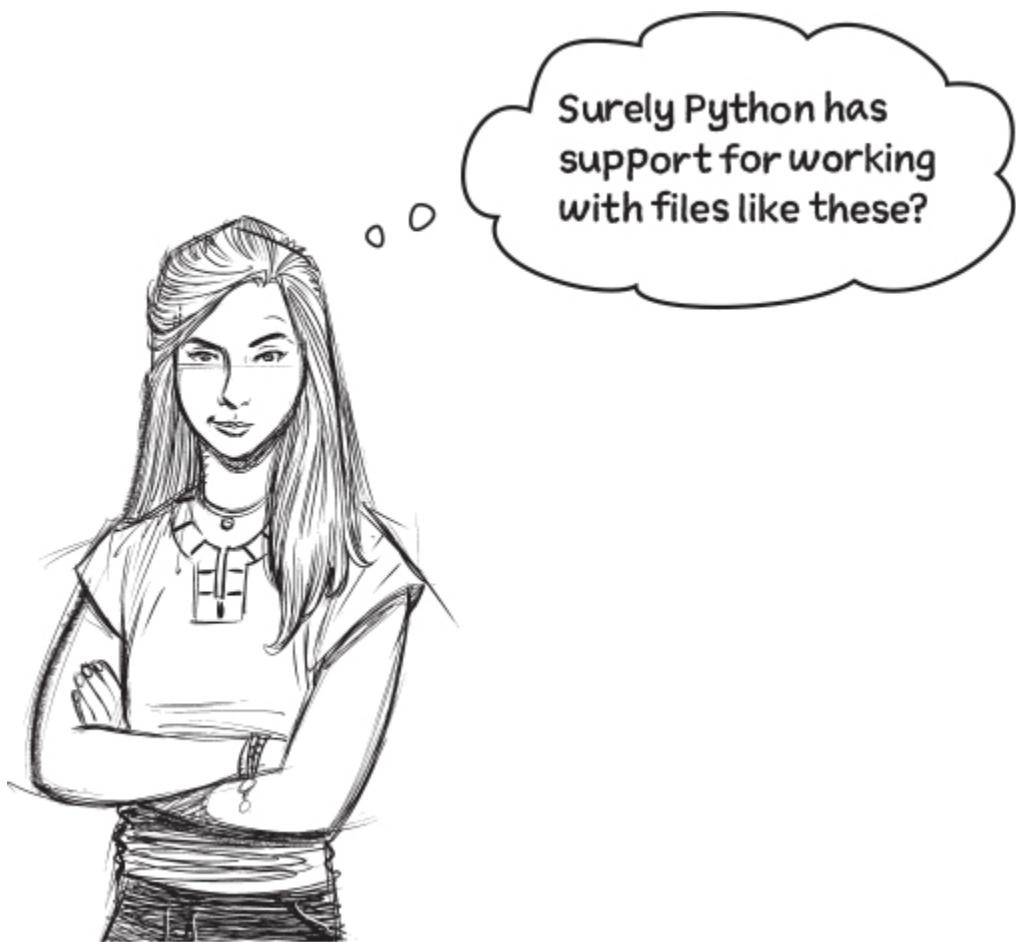
Each file in the `swimdata` folder contains the recorded times for one swimmer's attempts at a specific youth distance/stroke pairing. Recall the data file from the start of the previous chapter which shows Darius's under-13 times for the 100m fly:

This is the data  
you're interested in.

Darius-13-100m-Fly.txt 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** which 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 open BIF works with files

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



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

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



## NOTE

As always, follow along.

## TEST DRIVE

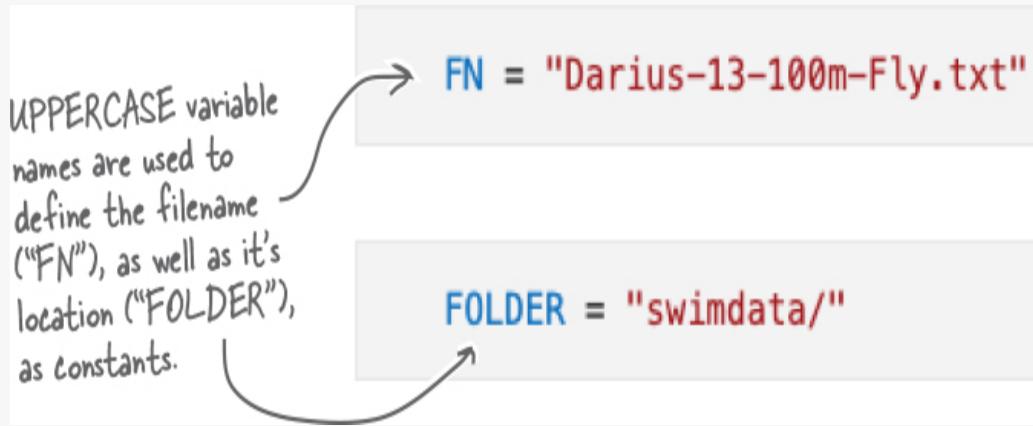


Let's see the **with** statement together with the **open** BIF at work, so you can get in on the action.

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

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

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

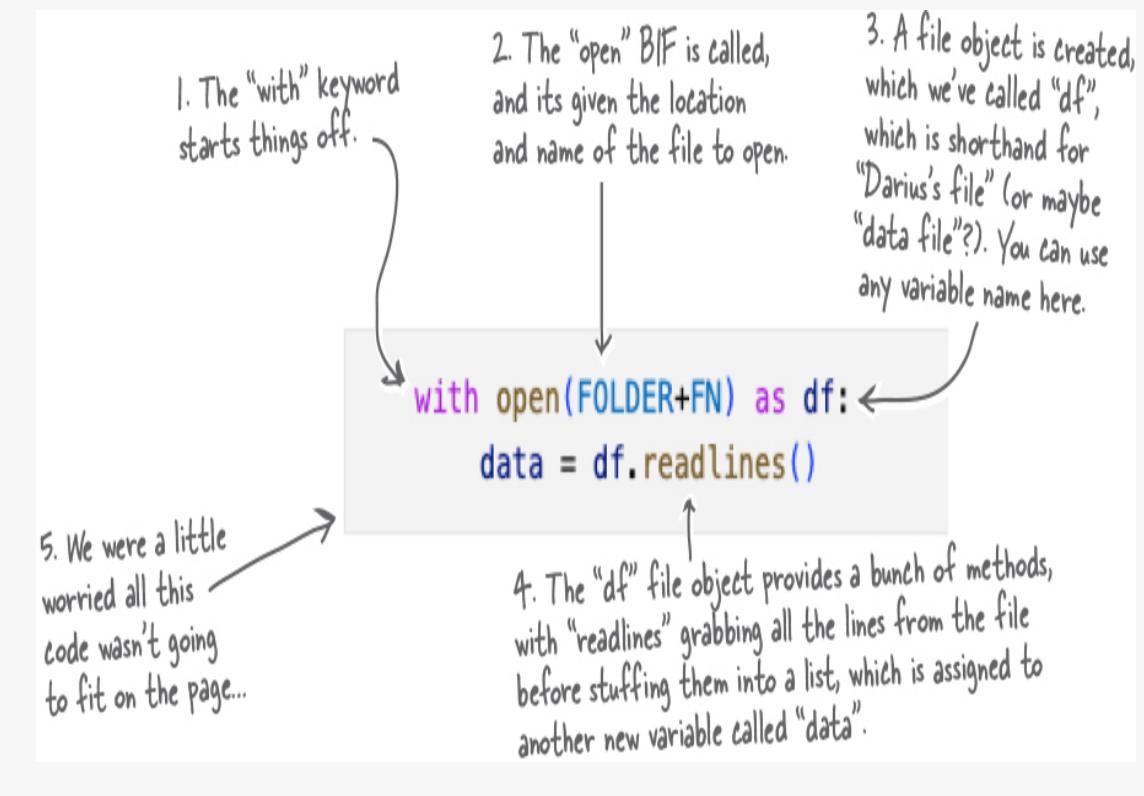


Although referred to as “constants”, Python doesn’t support the notion of constant values, so it’s a convention within the Python programming community to use UPPERCASE variable names to signal to other programmers that the values are constant (and should not be changed).

And, yes, eagle-eyed readers will have spotted that we – rather blatantly – disregarded (!! ) this convention in the previous chapter with “fn”.

This is a *shocking* use of a lowercase variable name when an uppercase constant name should’ve been used. We got away with this because Python doesn’t enforce this naming convention, as it’s not a language rule. That said, from now on, we’ll be sure to follow it.

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



## Not much code, but there’s lots happening...

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

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

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



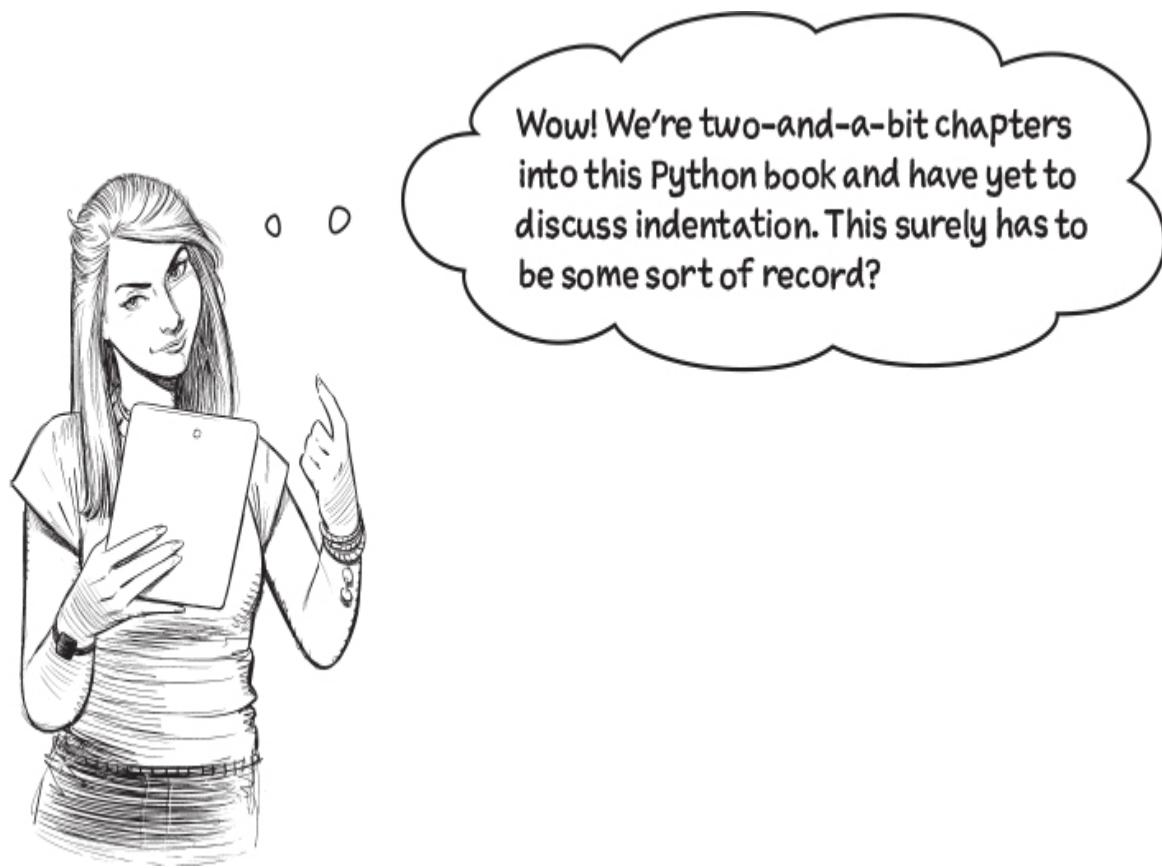
Let's highlight three important take-aways:

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

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

#### NOTE

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



## **It not that we don't want to talk about indentation.**

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

### **❸ The `with` statement closes the file *after* its code block runs.**

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

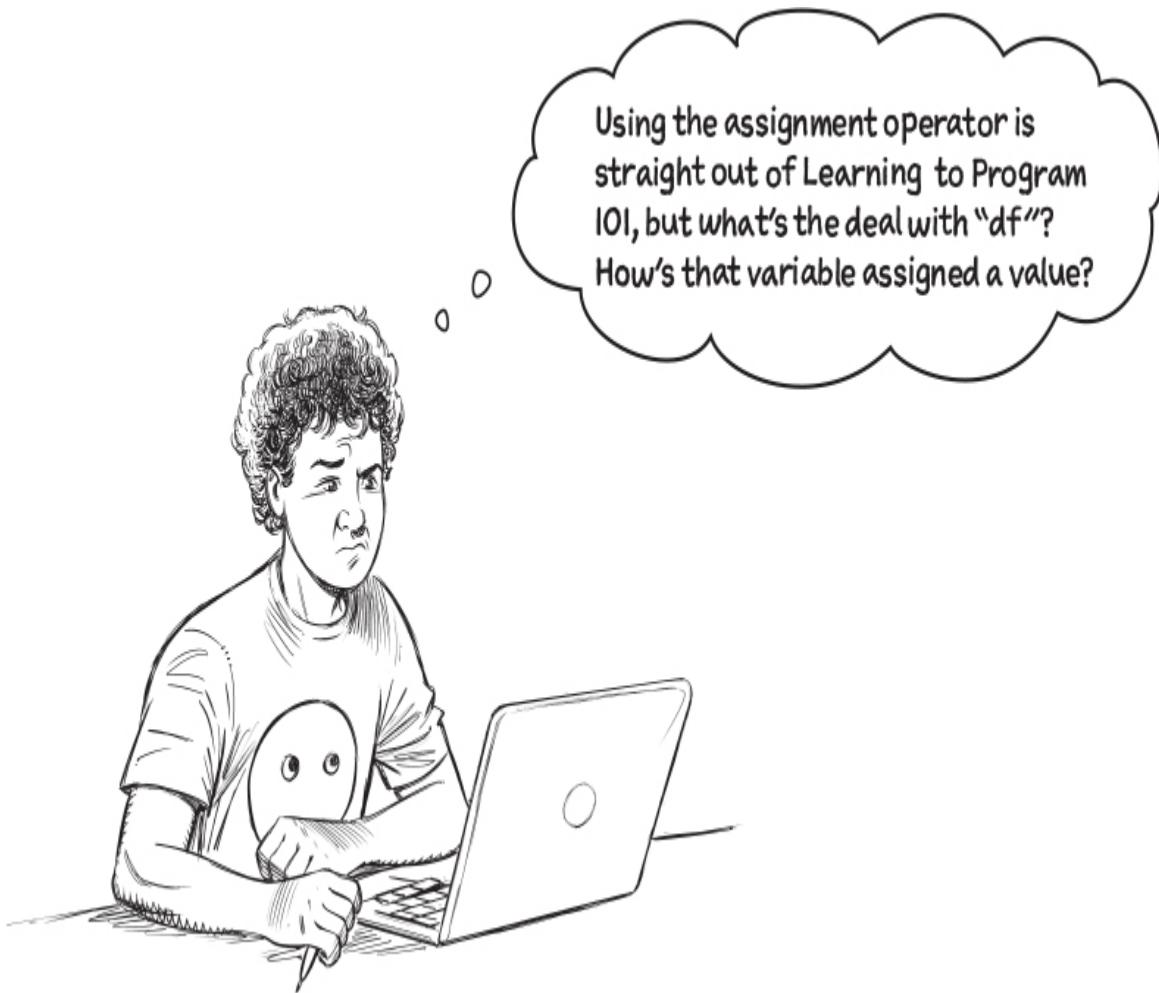
### **❹ Two variables are created by the code: `df` and `data`.**

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

the code block ends, although the `df` variable now refers to a *closed* file object.

## Variables are created dynamically, as needed

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



**The key word is “as”.**

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

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

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

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

`type(df)`

`_io.TextIOWrapper`

The "print dir" combo  
mambo  
strikes again!

`print(dir(df))`

[`'__CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__',  
 '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__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',  
 'fileno', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',  
 'readlines', 'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',  
 'writelines']`

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

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

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

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

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

`type(data)`

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

### NOTE

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

```
data
```

```
→ ['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 "data" is a list 'cause that's what the "type" B/F reported...

... but, this list's data values look a little weird. Until, that is, you realize you're looking at a list which 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.

```
data[0]
```

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

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

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

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



**Yes, with some help from “with”.**

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

Not only has your list been created, assigned to your `data` variable *and* populated with the data contained within the swimmer's file, but that `with`

statement has managed to complete the first two subtasks for Task #2. How *cool* is that?

Take a look (on the next page).

## Work has started on Task #2

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

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

← The code.

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

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

← The data.

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

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

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

## Your new best friend, Python's colon

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

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

A code block in Python ends when the indentation ends.

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

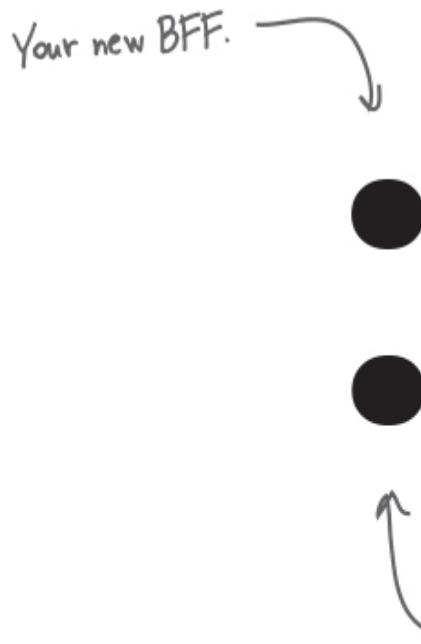
The use of the colon is critical here (which is why it's your new best friend). Like in real life, if you forget your best friend, bad things happen. If you forget the colon at the end of that line, Python refuses to run your code!

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

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

#### NOTE

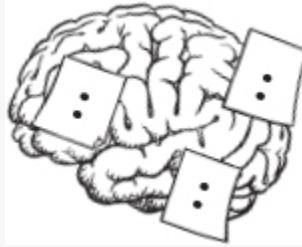
We think this is weird, too.



Your new BFF.

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

## MAKE IT STICK



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

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

## THERE ARE NO DUMB QUESTIONS

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

**A:** Yes. If you are used to using curly braces to indicate the start and end of a block of code, you have likely had a need to embed a block of code within another block of code, ending up with curly braces *inside* curly braces. It's the same thing with Python, except the level of indentation increases. Visually, it is easy to work out where the block of code starts and ends. It starts on the line right after the colon, and it ends with the block's indentation ends. For an example, recall the short example of a nested `for` loop from [Chapter 1](#).

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

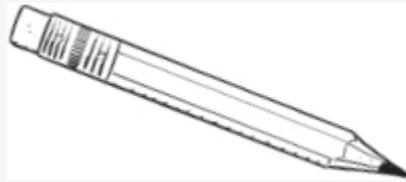
**A:** To be honest, it doesn't really matter whether you use four spaces, two spaces, or any number of spaces to indicate indentation. The tab character can also be used. What does matter is that whatever you do, your usage needs to be **consistent**. If you use four spaces on one line of code, you can't use two spaces on the next, nor can you mix'n'match spaces with the tab character. The Python parser gets confused when you do mix'n'match and raises an **IndentationError** or **TabError**. Note: if you *consistently* use four spaces, two spaces, or the tab character to indicate your level of indentation, you'll never see either of those errors.

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

Consistency is good.

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

## SHARPEN YOUR PENCIL



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

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

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

---

---



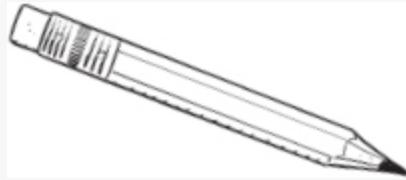
### Yes, to both questions.

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

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

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

## SHARPEN YOUR PENCIL SOLUTION



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

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

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

This code strips then splits the value in the first slot of the existing "data" list.

data[0].strip().split(",")

---

times = data[0].strip().split(",")

---

The result of the chain is assigned to a variable called "times".

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

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

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

The list is assigned to the "times" variable.

```
times = data[0].strip().split(",")
```

```
times
```

You now have a copy of the swimmer's timing data in the "times" list, ready for further processing.

## That was almost too easy

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

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

This chain removes the suffix then splits the sting on "-".

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

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

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

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

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

## Pause to review this task's code

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

These three lines  
of code sure do  
pack a punch.

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

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

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

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

Both do the same thing.



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

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

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

## Converting a time string into a time value

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

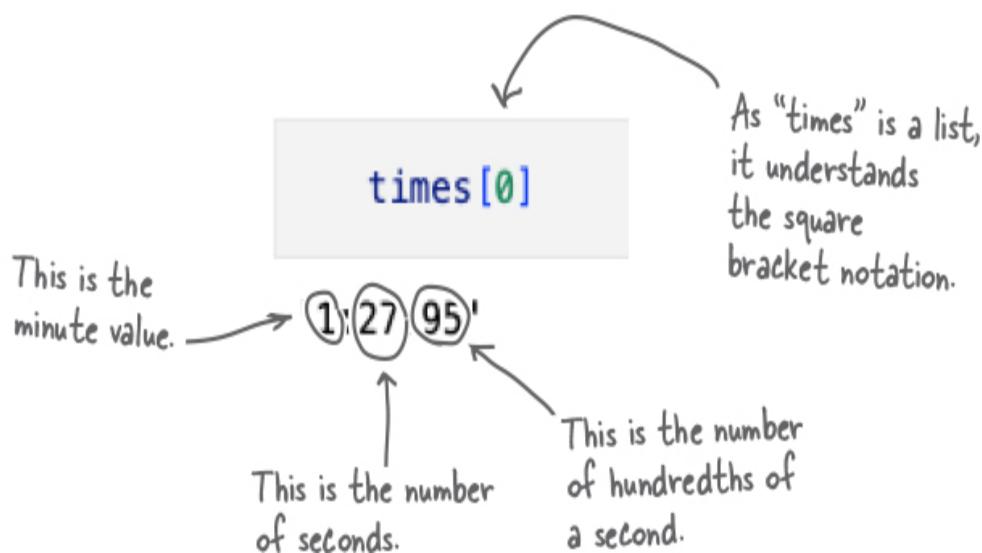
times

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

This is exactly what you want, but  
for the fact this is a list of strings,  
not a list of times.

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

Let's take a closer look at just 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.



## BRAIN POWER



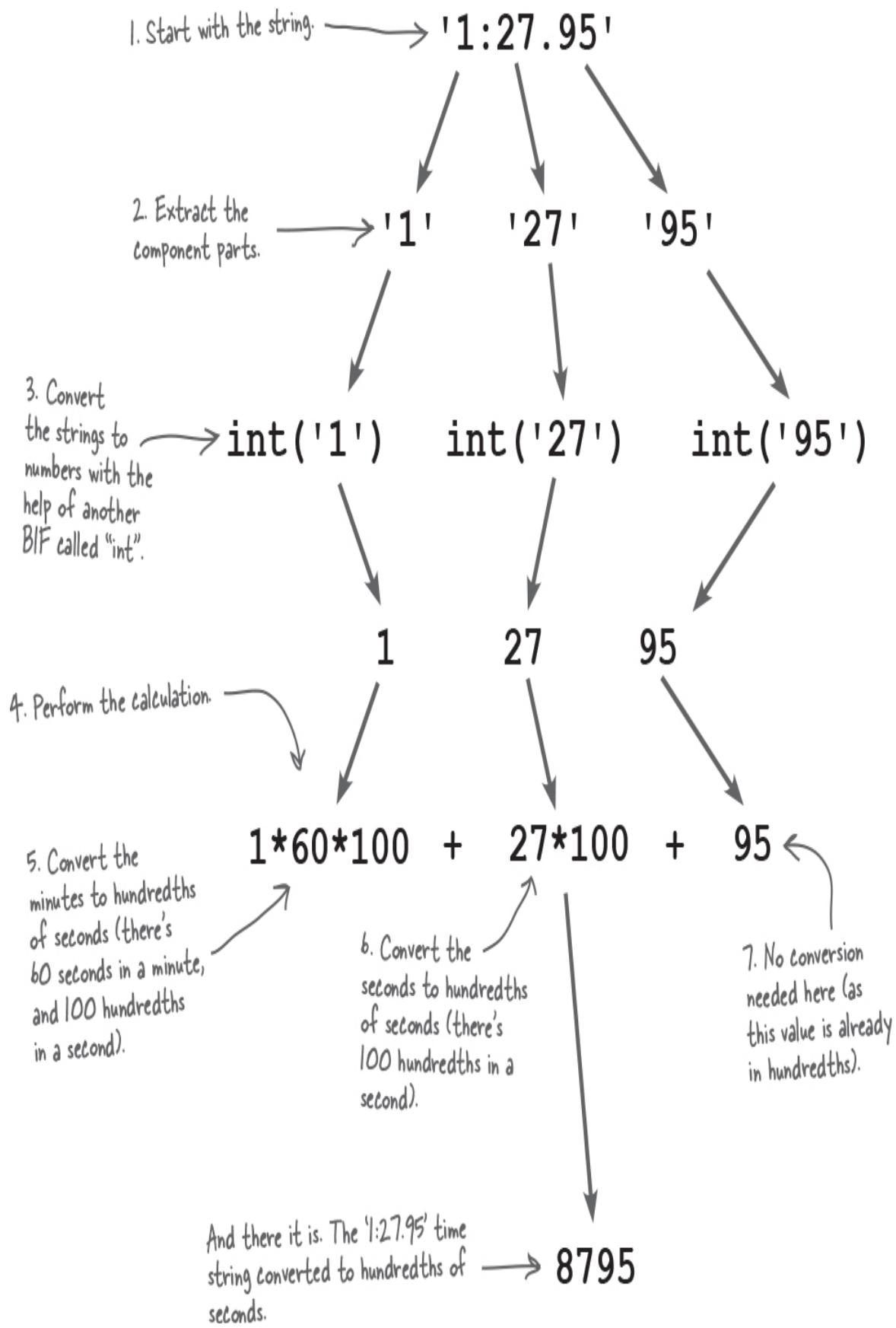
Assuming you can extract the three numbers you need from the string, can you think of a calculation which converts the string into a numeric value?

### NOTE

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

## Convert the times to hundredths of seconds

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



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

## To hundredths of seconds with Python

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



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

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

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

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(".")

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.

```
# 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 "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.



### If you can convert one swim time...

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

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

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

"for" is a  
Python keyword.

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

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

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

1:27.95

1:21.07

1:30.96

1:23.22

1:27.95

1:28.30



## Cool, isn't it?

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

There's always a temptation to use the **len** BIF to work out how big your list is before it's looped over, but with **for** this is an unnecessary step. The **for** loop starts with the first value in the list, takes each value in order,

processes the value, then moves onto the next. When the list is exhausted, the **for** loop terminates.

This is the sort of magic we love.

## THERE ARE NO DUMB QUESTIONS

**Q:** Didn't we see the "for" loop in Chapter 1?

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

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

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

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

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

## EXERCISE



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

---

---

---

---

---

---

---

---

---

---

---

---

---

---



**Python does indeed support `while`.**

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

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

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

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

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

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

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

1. You need to initialise a counter, which we've called "n" is this code.

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

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

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

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

4. And if you forget to increment the value of "n", 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).

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

## EXERCISE SOLUTION



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

Loop over  
all the  
swim times. → `for t in times:`

---

Extract  
the  
Component →  `minutes, rest = t.split(":")`

---

parts. →  `seconds, hundredths = rest.split(".")`

---

Display the  
calculated  
output. →  `print(t, " -> ", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))`

## TEST DRIVE



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

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

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

↑  
Whoo hoo! Looks good.

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

## You're motoring now!

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

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

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



### **Either approach works.**

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

What do you think?



## Let's keep a copy of the conversions

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

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

### Creating a new, empty list

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

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

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.

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

len(converts)

... and it's empty. → 0

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

## Displaying a list of your list's methods

It's combo mambo time!

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



Simply pass in your list's name.

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

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

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

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

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

Help on built-in function append:

append(object, /) method of builtins.list instance

Append object to the end of the list.

Oooooh. Interesting.

Ah ha!

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

#### NOTE

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

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



This is all great. But, I'm a little concerned you haven't declared how big your "converts" list is going to be, and you also haven't said what type of data it's going to hold... Do I need to be worried here?

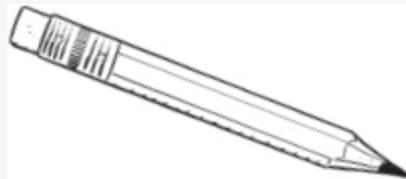
**No, you do not need to worry.**

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

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

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

## SHARPEN YOUR PENCIL



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

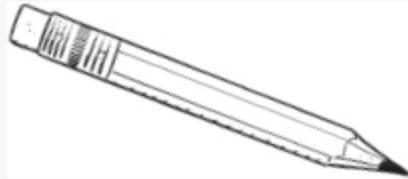
```
for t in times:  
    minutes, rest = t.split(":")  
    seconds, hundredths = rest.split(".")  
    print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

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



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

## SHARPEN YOUR PENCIL SOLUTION



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

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

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

Here's the code we came up with:

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

converts = []

for t in times:

    minutes, rest = t.split(":")

    seconds, hundredths = rest.split(".")

    converts.append(int(minutes)\*60\*100 + int(seconds)\*100 + int(hundredths))

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

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

## TEST DRIVE



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

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

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

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

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

times

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

converts

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

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

## It's time to calculate the average

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

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

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



That's the average in hundredths of seconds. I guess we need to show this as a time-string, right?

**Yes, as mins:secs.hundredths.**

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

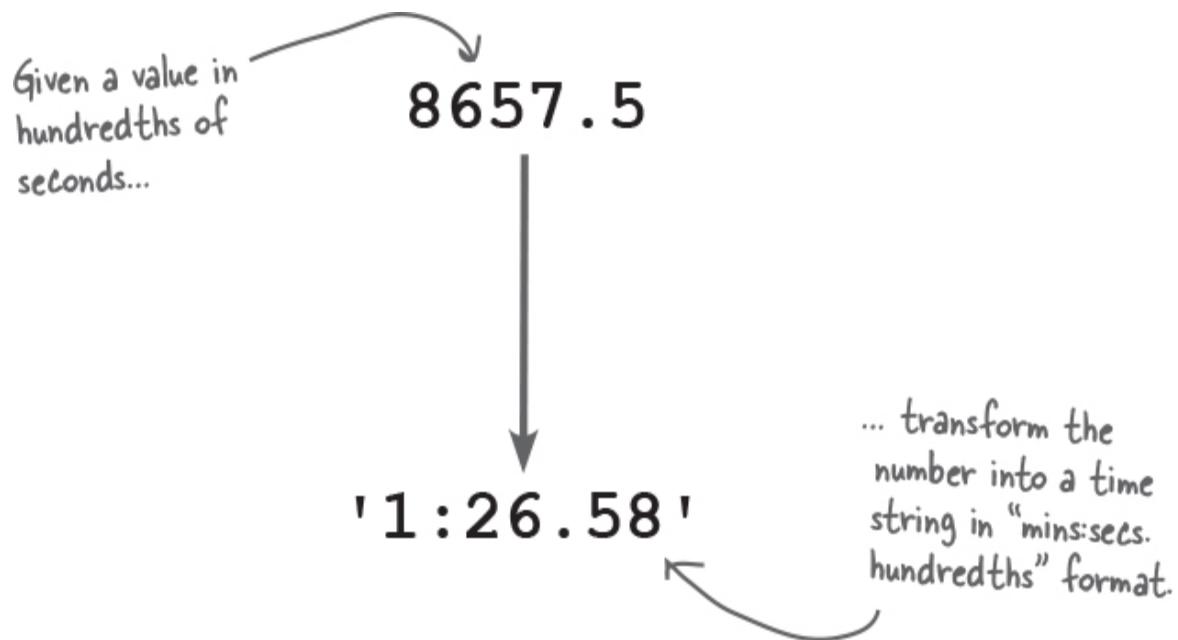
It can't be that hard, can it?

## 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)
```

86.58

Seconds.

Hundredths.

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

## ② Break the rounded average into it's 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 Maths is straightforward.

$$\text{seconds} = \text{mins\_secs} - \text{minutes}*60$$

seconds

26

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

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

minutes, seconds, hundredths

(1, 26, '58') ← The three calculated values...

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

'1:26.58' ← ... are used to build a swim time string in the required format. And, yes, the + operator works with strings as well as numbers, performing concatenation.

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

average

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

'1:26.58'

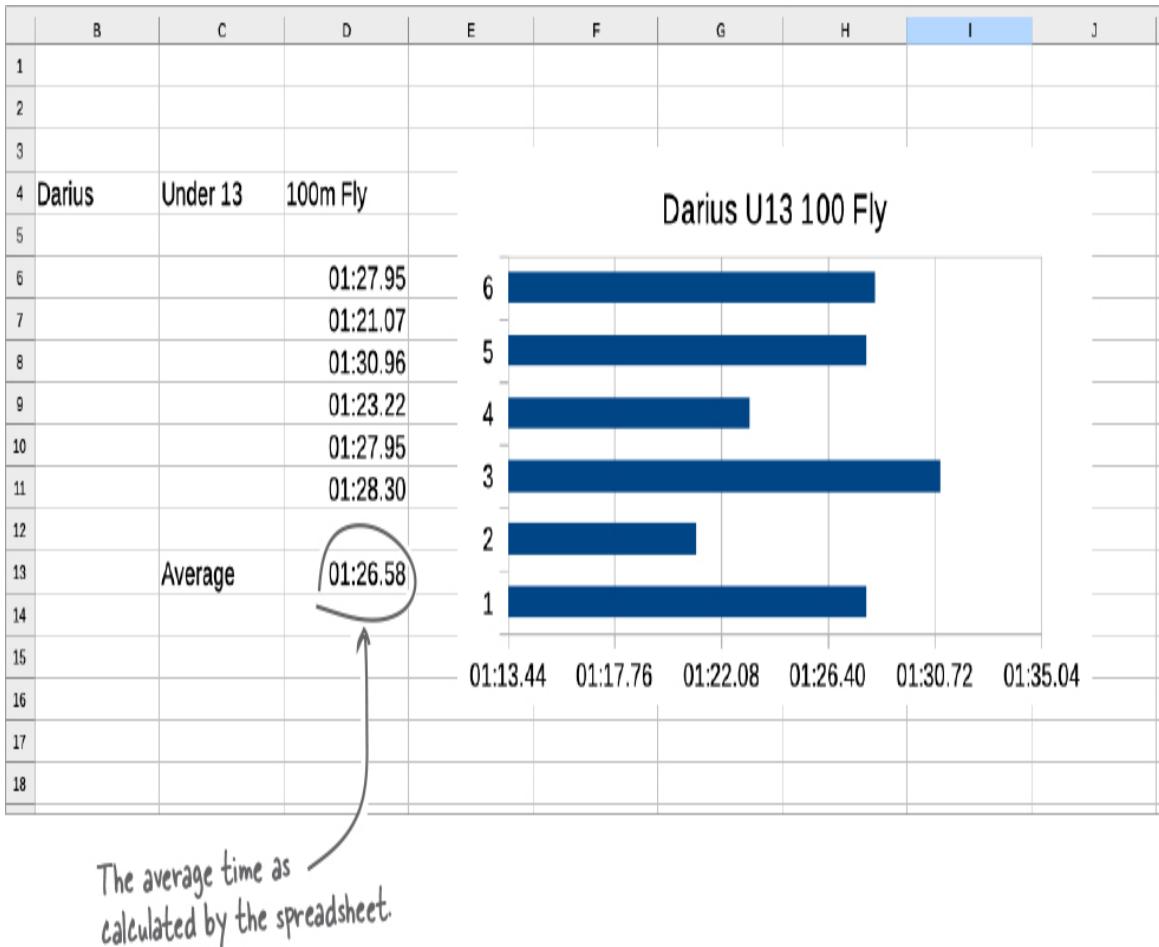


### **Yes, and it's easier than you think.**

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 been a while since your last checkmark...

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

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

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

## EXERCISE



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

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

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

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

## EXERCISE SOLUTION



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

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

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

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

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

We started with the code that defines the two constant values, identifying the filename to use as well as its location.

Next up was that powerful line of code from the end of the previous chapter which extracted the values which identifies the swimmer, their age, the distance classification and the stroke swam.

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

In the original code, this was called "fn" (i.e., written in lowercase). As it is meant to be used as a constant, we tweaked this code to show it in UPPERCASE.

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

The "with" statement opens the data file, reads the lines in the file into a list, then closes the file (automatically for you). A quick strip-split combo on the first line of data from the file creates another list, called "times", which contains this swimmer's swim time strings.

The strings in "times" are converted from the "mins:secs.hundredths" format into numeric hundredths of seconds, ending up in yet another list called "converts". The "for" loop, together with the "append" method, makes this as easy as it can be.

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

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

['1:27.95', '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 sub-task (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!

### ① Extract data from the file's name

a. Read the filename ✓

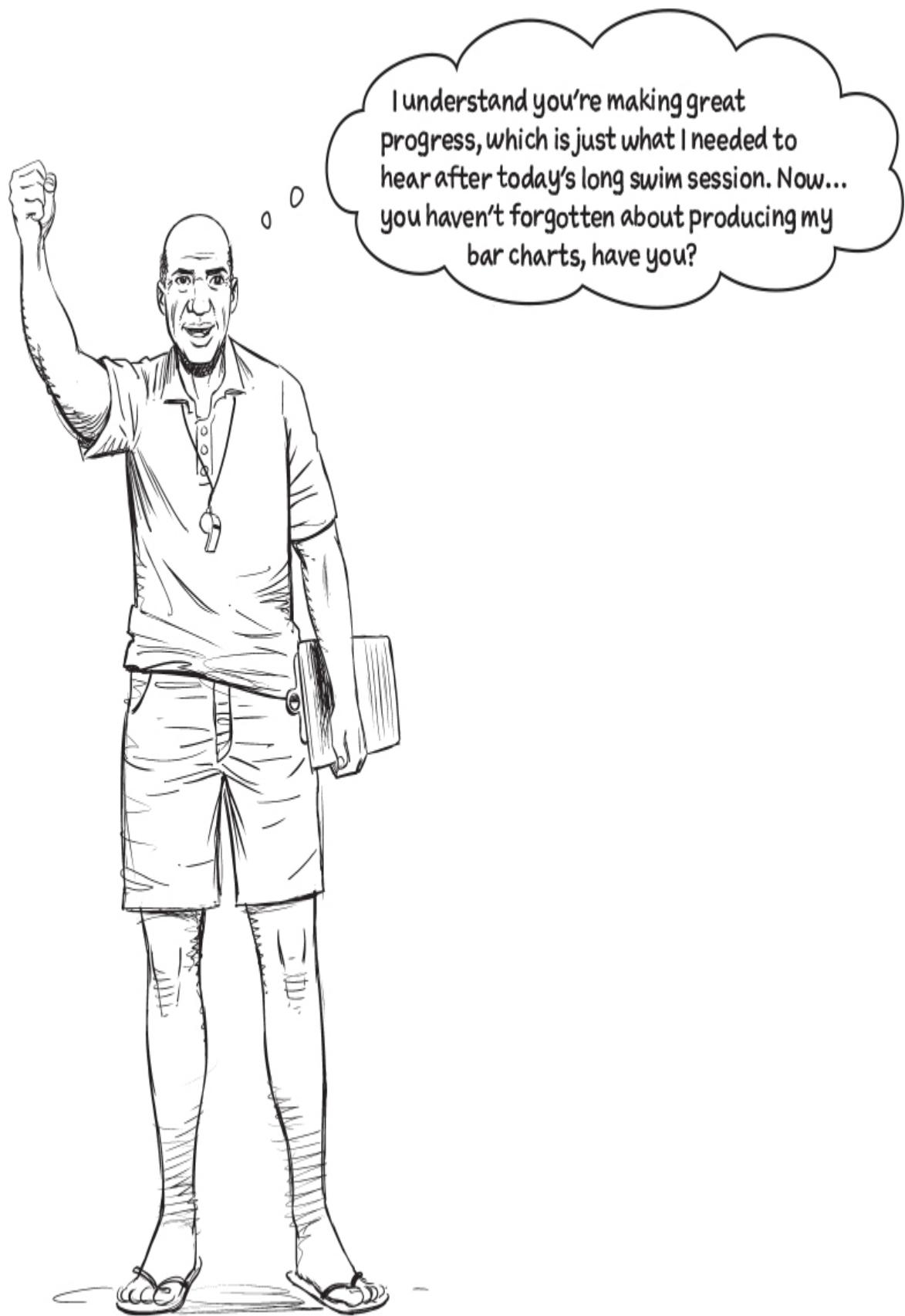
b. Break the filename apart by the “-” character ✓

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

### ② Process the data in the file

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

Of course, getting to this point doesn't necessarily mean you're done...



I understand you're making great progress, which is just what I needed to hear after today's long swim session. Now... you haven't forgotten about producing my bar charts, have you?

## No, we haven't forgotten.

The next chapter lays the groundwork for getting to the point where you can tackle the charting requirement, so we've ask the Coach to bear with us.

At the moment, your code only works with the data for one specific data file. There are another 60+ files in the Coach's dataset. It would be nice if there was a way to use this code with *any* of them on demand, and as needed.

Doing so is something you can mull over on your way to the next chapter when we'll work through a solution to this problem *together*.

For now, let's conclude this chapter with another summary and a super-topical crossword puzzle. Enjoy!

## CHAPTER REVIEW BULLET POINTS

- Python has no real notion of a **constant** value, so a programming *convention* has emerged whereby constants are assigned to UPPERCASE variable names.
- The **with** statement is used to manage the context within which its block of code runs.
- When used with the **open** BIF, the **with** statement *automatically* arranges to close any still opened file once the **with**'s block of code terminates.
- Additionally, the **with** statement assigns the opened file to a **file object** which is used as shorthand for the opened file.
- The file object has lots of methods built-in, including **readlines** which... emm, eh... reads lines. The lines are returned as a **list**.
- The rest of the file object methods can be displayed with the **print dir** combo mambo.
- By default, the **open** BIF opens a named file for reading. The file is assumed to contain **textual data**.
- The colon, in addition to being your **BFF**, indicates to Python that an **indented** block is about to start on the very next line of code.
- The **.strip().split()** chain is very popular. It used in lots of places, by lots of Python programmers.
- Python has all the usual **operators** (e.g., `+`, `-`, `*`, `/`), and then some (e.g., `//` and `:=`).
- The `//` operator performs **floor division** which ensures, among other things, that the result of the division is always a whole number.
- Although you are well within your rights to *love* the **for** and **while** loops equally, the **for** loop sees a lot more use in the community,

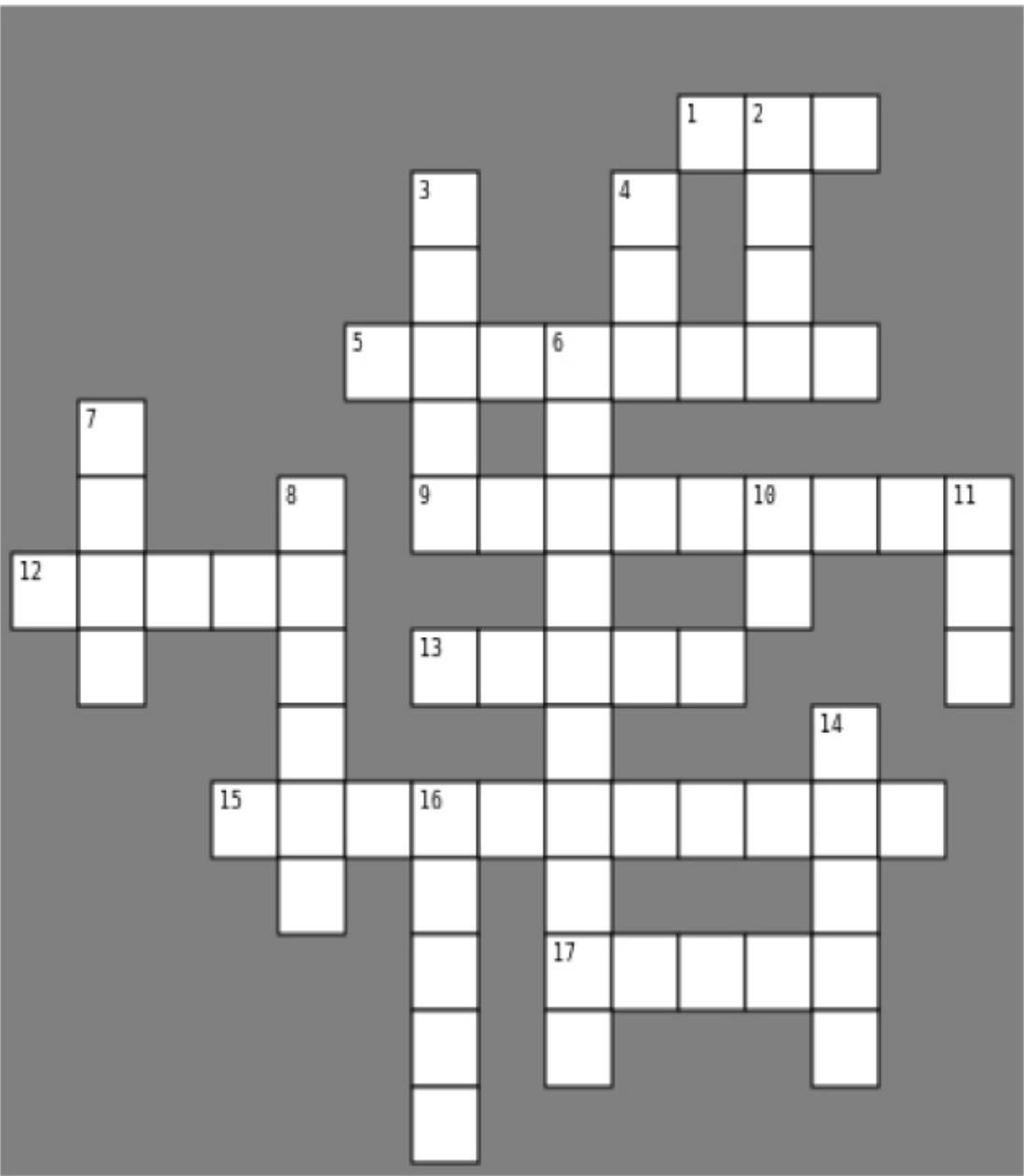
and it's typically reached for *first* whenever a Python programmer needs to iterate.

- Python's **list** is a very popular built-in data structure, which understands the familiar square bracket notation.
- An empty list looks like this: [ ].
- Items in a list are enclosed in **square brackets** and separated from each other by a **comma**. Knowing this, lists easy to spot.
- Lists come with lots of built-in functionality, but the *star of this show* (in that chapter) is the **append** method.
- The **int** BIF converts its single argument to an integer, if it can. The value "42" converts without issue, whereas "forty-two" sends **int** into meltdown.
- The **PSL** (not the coffee!!) provides the **statistics** module which, among other things, provides a handy **mean** function. The **mean** function has everything to do with calculating averages, as opposed to not being nice to be around...
- The **round** BIF gets rid of unwanted decimal places.
- The **str** BIF converts its argument to a string.
- The Coach is really **happy** with your progress to date, as should you be. There's a lot of **good work** in this chapter.

## THE LISTS CROSSWORD



*All of the answers to the clues are found in this chapter's pages, and the solution is on the next page. Got for it!*



## Across

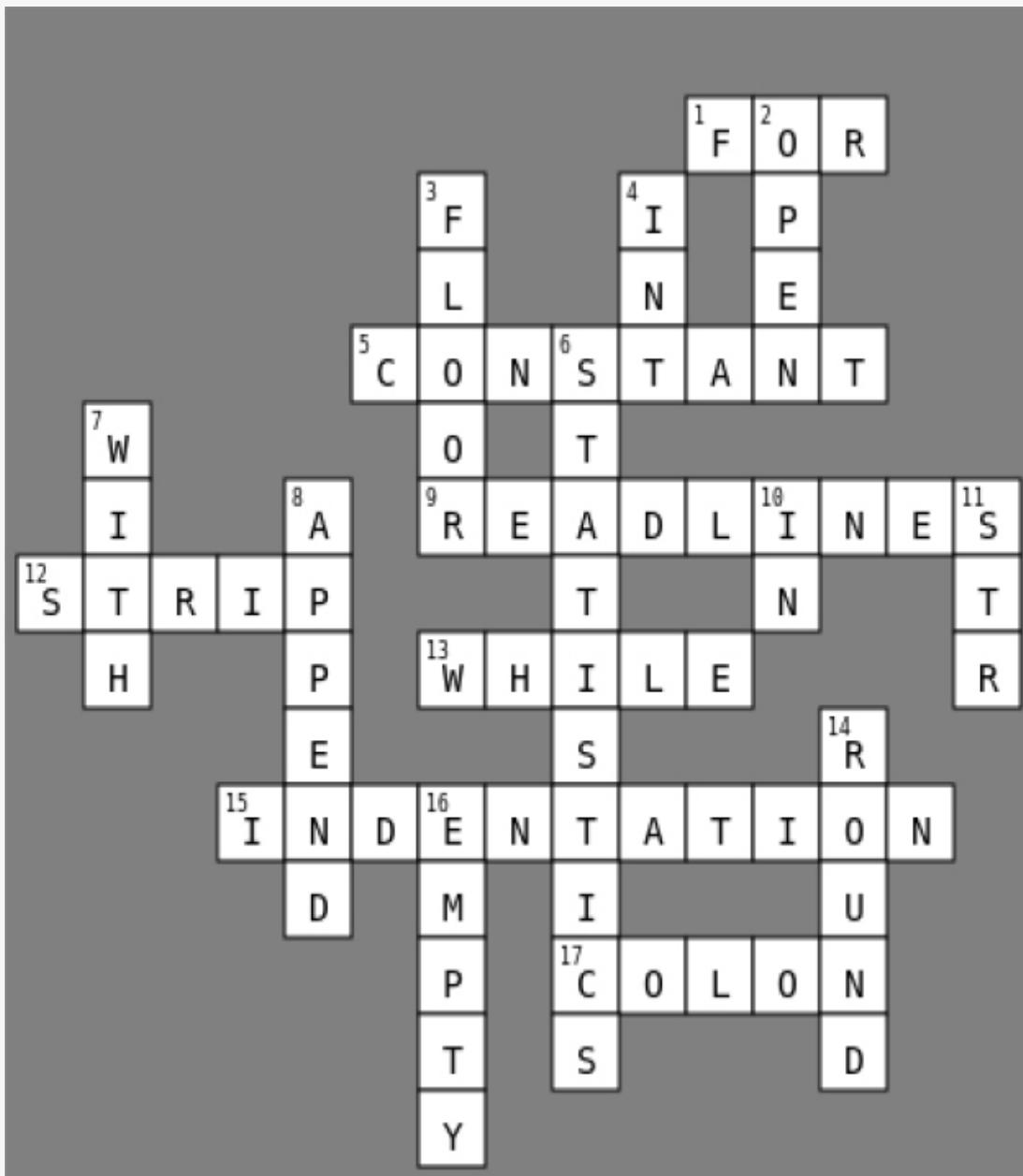
1. Python programmer's favorite looping construct.
5. When a variable name is in UPPERCASE, it's meant to be treated as one of these.
9. This method creates a list from your file's data.
12. Part of a famous combo, when paired with **split**.

13. The less-used looping construct.
15. Another name for whitespace when used with code blocks.
17. Your new BFF.

## **Down**

2. Another powerful combo when used with 7 down.
3. // performs \_\_\_\_\_ division.
4. A numeric conversion BIF.
6. A module loved by Maths-heads.
7. The recommended statement to use when opening files.
8. This method grows lists.
10. A small keyword which you'll learn more about in a later chapter.
11. A string-creating BIF.
14. A BIF to control decimal places.
16. This [ ] signifies an \_\_\_\_\_ list.

## THE LISTS CROSSWORD SOLUTION



Across

1. Python programmer's favorite looping construct.
5. When a variable name is in UPPERCASE, it's meant to be treated as one of these.
9. This method creates a list from your file's data.
12. Part of a famous combo, when paired with **split**.
13. The less-used looping construct.
15. Another name for whitespace when used with code blocks.
17. Your new BFF.

## **Down**

2. Another powerful combo when used with 7 down.
3. // performs \_\_\_\_\_ division.
4. A numeric conversion BIF.
6. A module loved by Maths-heads.
7. The recommended statement to use when opening files.
8. This method grows lists.
10. A small keyword which you'll learn more about in a later chapter.
11. A string-creating BIF.
14. A BIF to control decimal places.
16. This [ ] signifies an \_\_\_\_\_ list.

# **Chapter 4. List of Files: *Functions, Modules & Files***

---

## **A NOTE FOR EARLY RELEASE READERS**

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

This will be Chapter 3 of the final book. Please note that the GitHub repo will be made active later on.

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



**Your code can't live in a notebook forever. It wants to be free.** And when it comes to freeing your code and **sharing** it with others, a bespoke **function** is the first step, followed shortly thereafter by a **module**, which lets you organize and share your code. In this chapter, you'll create a function directly from the code you've written so far, and in the process create a **shareable** module, too. You'll immediately put your module to work as you process the Coach's swim data with **for** loops, **if** statements, conditional tests, and the **PSL** (Python's *Standard Library*). You'll learn

how to **comment** your functions, too (which is always a *good idea*). There's lots to be done, so let's get to it!



## Cubicle Conversation

**Sam:** I've updated the Coach on the progress-to-date.

**Alex:** And is he happy?

**Sam:** To a point, yes. He's thrilled things have started. However, as you can imagine, he's only really interested in the final product, which for the Coach is the bar chart.

**Alex:** Which should be easy enough to do now that the most-recent notebook produces the data we need, right?

**Mara:** Well... sort of.

**Alex:** How come?

**Mara:** The current notebook, `Times.ipynb`, produces data for Darius swimming the 100m Fly in the Under 13 age group. But, there's a need to perform the conversions and the average calculation for *any* swimmer's file.

**Alex:** Sure that's easy: just replace the filename at the top of the notebook with another filename, then press the *Run All* button and – voila! – you've got your data.

**Mara:** And you think the Coach will be happy to do that?

**Alex:** Errr... I hadn't thought about how the Coach is going to run this stuff.

**Sam:** We are heading in the right direction, in that we do need a mechanism which works with any swimmer's filename. If that can be produced, we can get on with then creating code for the bar chart.

**Alex:** So we have a ways to go yet...

**Mara:** Yes, but not far. As you already mentioned, all the code we need is in the `Times.ipynb` notebook...

**Alex:** ...which you don't want to give to the Coach...

**Mara:** ...well, not it it's current form.

**Alex:** Then how?

**Sam:** We need a way to package-up the code so it can be used with any filename and accessed outside of the notebook...

**Alex:** Ah, but of course: we need a function!

**Sam:** Which gets us part of the way.

**Mara:** If the function is put inside a Python module it can be shared in lots of places.

**Alex:** Sounds good to me. Where do we start?

**Mara:** Let's start by turning the existing notebook code into a function that we can call, then share.

# How to create a function in Python

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

## ➊ Think up a nice, meaningful name

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

## ➋ Decide on the number and names of any parameters

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

## 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.

### NOTE

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 names 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 get_swim_data(fn):
```

3

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

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

## Be sure to add a comment to your function

Full disclosure: we told a little white lie... we've led you up the garden path to believe all you need to do is copy your code into your function, suitably

indented. But, you should *also* add a comment to the start of your function.

You've already seen how a single `#` character switches on a single-line comment in your code. When you need to add a multi-line comment, you can enclose your comment in **triple-quotes** which, believe it or not, is *also* a string object.

### NOTE

A string can span many lines when you surround it with triple-quotes.

When Python encounters a string object which isn't assigned to a variable name, it's simply ignored, which makes using a triple-quoted string *perfect* for multi-line comments. Here's an example:

```
def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""

```

The comment (which future  
you will thank you for).



Note the triple-quotes  
surrounding this text.



The function's signature.





### Great question (and well spotted).

The **tuple** is one of Python's built-in data structures, which you briefly met in [Chapter 1](#). Back then we over-generalized a little, suggesting tuples were quite like lists. We'll have more to say about tuples later in this book. For now, recall what one looks like:

(3, 'of', 'Clubs') ←  
A tuple  
(representing a  
playing card).

Now, what's your second question?

I'm not sure if I've missed something, but...  
that single parameter has lost me a bit.  
Obviously, its name is "fn", but what's its  
type? Did you forget to declare it?



### That's another great observation.

No, we haven't forgotten to declare the parameter's type. In fact, in Python you don't need to declare the types of your parameters, as every function parameter can be of *any* type.

Python is a *dynamically-typed* language. This means, among other things, that typing decisions are not made until run-time, which is very unlike those

*stodgy* statically-typed languages where everything must be known up-front, at compile-time.

## NOTE

Depending on your programming background, this might strike you as controversial. Our advise is not to let this worry you. This page’s “Geek Note” discusses a Python 3 feature which can be thought of as meeting the statically-typed die-hards half-way.

## GEEK NOTE



Recent versions of Python 3 have introduced the notion of **type hints**. This extra syntax adds annotations to your function’s signature to *indicate* the types expected. As an example, here’s how you’d add type hints to your `get_swin_data` function to indicate a string is expected as input, with a tuple returned:

```
def get_swin_data(fn: str) -> tuple:
```

These annotations are “hints” and are otherwise **ignored** by Python: there’s nothing checking at run-time to ensure a string is actually sent as input to the function. This means the use of type hints is *optional*. Python is dynamically-typed, after all. Some programmers believe type hints are the best thing since sliced bread, whereas others are... less enthusiastic. We’re on the fence and, consequently, use type hints in this book sparingly (if at all).

## Create a file for your function's code

There's nothing stopping you defining any function within a Jupyter Notebook code cell. In fact, when you are trying to work out what code you need to create, experimenting in a notebook *first* is our recommended approach.



However, all the code you need to add to your `get_swim_data` function already exists in your `Times.ipynb` notebook, so there's no need for any further experimentation. Instead, let's use VS Code to create a new Python file called `swimclub.py`.

### NOTE

In VS Code, select “File”, then “New File”, then select “Python File” as the type. When an empty editing window appears, save the file as “`swimclub.py`”. (And be sure to save the file into your existing “Learning” folder).

Here's the code we copied into `swimclub.py`, which we've applied a few minor tweaks to (note the annotations):

```

import statistics
FOLDER = "swimdata/"

def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
        times = data[0].strip().split(",")
        converts = []
        for t in times:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
            converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
        average = statistics.mean(converts)
        mins_secs, hundredths = str(round(average / 100, 2)).split(".")
        mins_secs = int(mins_secs)
        minutes = mins_secs // 60
        seconds = mins_secs - minutes * 60
        average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

```

The code relies on this PSL module, so be sure to import it.

Define any required constant values (note: the "FN" constant is *\*not\** needed here as the filename is passed as a parameter to the function).

"Python File" as the type.  
When an empty editing window appears, save the file as "swimclub.py". (And be sure to save the file into your existing "Learning" folder).

There are minor edits here. Rather than using the constant "FN", we're referring to the parameter from the function's signature.

All of this code is indented under that "def" statement.

Another small edit was to change this line's assigned-to variable name to "average\_str". The "average" name was used earlier to store the numeric average, and we decided to remember both the number and the converted string in their own variables.



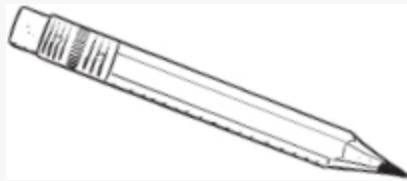
**Yes, that's it exactly.**

There's just a small addition to apply to the function's code to ensure it's really useful.

## Functions can return results to their calling code

In addition to accepting data *on the way in*, function's can deliver data to the code which invoked them (aka data *on the way out*). Arguments sent into a function are assigned to the parameter names defined in the function's signature, whereas any results are sent back to the calling code with a **return** statement.

## SHARPEN YOUR PENCIL

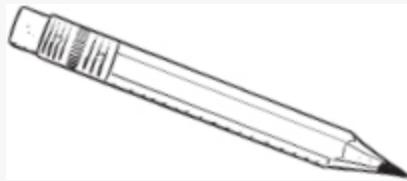


Take a moment to read the code on the previous page then, in the space below, write in the **return** statement you'd add to the end of your function in order to send values back to the function's caller.

You'll find our suggested **return** statement on the next page, but do have a go at creating this single line of code yourself before flipping the page:

---

## SHARPEN YOUR PENCIL SOLUTION



You were to take a moment to read the code from two pages back then, in the space below, write in the **return** statement you'd add to the end of your function in order to send values back to the function's caller.

Here's the **return** statement we'd use. How does yours compare?

---

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

A collection of values are returned from the function to the calling code. Note the absence of parentheses around that list of variable names (Python doesn't require them).

## Update and save your code before continuing...

Before moving onto the next page, be sure to add the following line of code as the last line in your `get_swim_data` function within your `swimclub.py` file. Be careful to match the indentation of this line of code with the indentation used for all the other code in your function:

Use VS Code to add this line of  
code to the end of your function,  
then save your file.



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

### WATCH IT!



#### Remember to save your code often when using the VS Code editor

*When you use a Jupyter Notebook, you execute your code immediately using Shift+Enter without having to save anything beforehand.*

*However, when you use an editor to create Python code (as you just did with VS Code and your `swimclub.py` file), nothing runs until you specifically tell Python to run your code. When you do, the latest saved version of your code is executed, so be sure to save early and often when working within an editor.*

## Use modules to share code

If you look at the code in your `swimclub.py` file, it consists of a single **import** statement, a single constant definition, and a single function. Once you move code into its own file, it becomes is a Python *module*. Congratulations: you've created your first Python module, called `swimclub`.



**That's pretty much it.**

Things can get a lot more complex but, as you've just seen, creating a **shareable module** in Python is as simple as popping some code in a file and giving it a name. It's so easy, there ought to be a law against it. .. 😊

## EXERCISE



With the code in `swimclub.py` complete, you were to create another new notebook in your `Learning` folder, giving it the name `Files.ipynb`.

In your new notebook's first cell, you were to type this line of code, being sure to type **Shift+Enter** to execute it:

```
import swimclub
```

When importing any module, you never include the ".py" extension.

This imports any constants and functions defined in the `swimclub` module from your `swimclub.py` file. In the space below, you were to provide the line of code which calls the `get_swim_data` function from this module, passing the name of Darius's file from the previous chapter:

Write your line of code here. As usual, the answer is over the page.

## EXERCISE SOLUTION



With the code in `swimclub.py` complete, you were to create another new notebook in your **Learning** folder, giving it the name **Files.ipynb**.

In your new notebook's first cell, you were to type this line of code, being sure to type **Shift+Enter** to execute it:

```
import swimclub
```

This imports any constants and functions defined in the `swimclub` module from your `swimclub.py` file. In the space below, you were to provide the line of code which calls the `get_swim_data` function from this module, passing the name of Darius's file from the previous chapter:

This is the line of code we used. We prefix the name of the function with the name of the module, then pass the name of the file as the argument value.



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



### This is a fully qualified name.

When you refer to your function with “module DOT function”, you are qualifying the name of your function with the name of the module which contains it. This is very common in practice, although there are other common importing techniques. You’ll see examples of these as you continue to work through this book.

## TEST DRIVE



If you haven't done so already, press **Shift+Enter** on each of your notebook's code cells to execute them. The first cell produces no output as all that happens is the `swimclub` module is imported. The **import** statement runs the code in the identified module from top-to-bottom. The second cell run your functiona and produces output as shown below:

```
import swimclub
```

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

```
('Darius',  
 '13',  
 '100m',  
 'Fly',  
 8657.5,  
 '1:26.58',  
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'])
```

The data extracted from the file name.

The average (as a numeric and a time string).

The list of time strings.



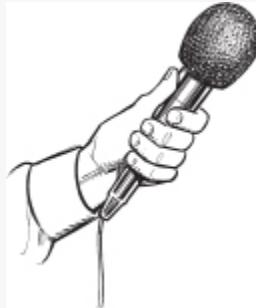
Something's not right here. What's with those parentheses around all the returned data values?

### Good eye. Well spotted, too.

This may not be the explanation you're expecting here, but those parentheses are meant to be there.

Let's dig into this a little so you can appreciate what's going on.

## TODAY'S INTERVIEW



Chatting with Python's **function**.

**Head First:** Thank you for taking time out of your busy day to chat to us.

**Function:** No problem.

**Head First:** How come you're so busy?

**Function:** I'm used everywhere, all the time.

**Head First:** And you'll work with anything?

**Function:** If you're referring to the data I'll accept then, yes, I'll happily accept anything you give me.

**Head First:** Care to expand?

**Function:** Sure. You can send me any number of argument values which I'll happily match up with my parameters, and all you need to do is ensure you send me the correct number. If I've got two parameters, be sure to send me two argument values.

**Head First:** What happens if I send, say, one argument or three instead?

**Function:** I get cranky.

**Head First:** I see. It's like that, is it?

**Function:** Yes. I'm pretty strict on that sort of thing. Unless, of course, one of your two example parameters happens to be declared as *optional*.

**Head First:** What happens then?

**Function:** Continuing on with my example of a two-parameter function... if, for instance, the second 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 which has been decided upon by the programmer who created me.

**Head First:** That sounds kind of complex.

**Function:** It's not really and – to be honest – it's not something every function needs, but it's part of me if you need it. I'm pretty flexible.

**Head First:** And what about returning values? Is it the same deal? Can any number of values be returned?

**Function:** No.

**Head First:** Seriously? No? That's all you're going to say on this?

**Function:** Well... I thought 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 `get_swim_data` function call on the previous page. There are *seven* results returned.

**Function:** No there isn't, there's only ONE.

**Head First:** What the...

**Function:** If you look closely, you'll notice those parentheses surrounding the seven data values, right?

**Head First:** Yes, but...

**Function:** No “buts” about it. Those parentheses are a single tuple which happens to contain the seven individual data values. As I said,

only ONE result is returned, either a single result on it's own, or a single tuple result with multiple values contained therein.

**Head First:** But the code doesn't convert the seven return values into a tuple.

**Function:** Yeah, the code didn't, *but I did*. I do it automatically when I see a programmer attempting to return more than ONE result. You can thank me later.

**Head First:** No, I'll thank you now. That's important to know. Thanks for the chat!

**Function:** I'm always happy to clear things up. Bye!

## Functions return a tuple when required

When you call a function which looks like it returns multiple results, it isn't. Instead, you get back a single tuple containing a collection of results (regardless of how many there are).

This looks like seven objects are being returned from the function. But this is not allowed as functions can only ever return one result. So Python bundles the returned objects into a single tuple.

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

```
('Darius',
```

```
'13',
```

```
'100m',
```

```
'Fly',
```

```
8657.5,
```

```
'1:26.58',
```

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

Tuples surround their  
objects with parentheses  
(unlike lists, which use  
square brackets).



**That's a great suggestion.**

Not that we're suggesting there's a bit of mind reading going on here, but it is a little spooky we had the same idea...

## BEHIND THE SCENES



### Up close and personal with Python's tuple.

Python's docs state that a **tuple** is an *immutable sequence*.

#### Immutable?

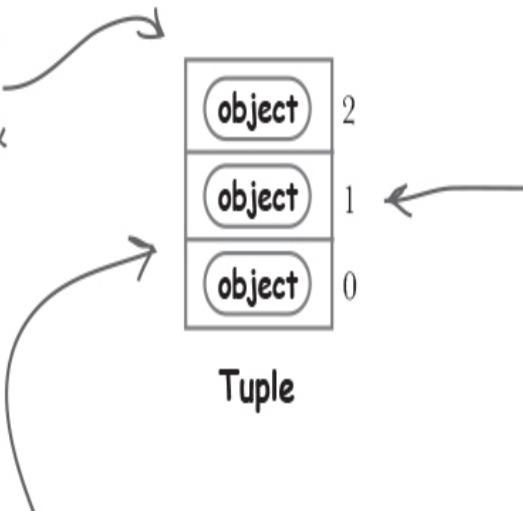
You've already met two *immutable* data types: numbers and strings. Both share the characteristic that once a value is created in code, the value **cannot** be changed. For instance, 42 is always 42: it cannot change, it's *immutable*. The same thing goes for strings in Python: "hello" is always "hello": once created, it cannot change, it's *immutable*.

Python's **tuple** takes this idea of immutability and applies it to a collection of data values. It can sometimes help to think of a tuple as a *constant list*. Once values are assigned to a tuple, the tuple cannot change, it's *immutable*.

#### Sequence?

If you can use the square-bracket notation to access items (or slots) from a collection, then you're working with a *sequence*. Python's list is the most familiar sequence type, but others do exist, including strings and... **tuple**. In addition to supporting the use of square brackets, sequences also maintain the *order* of the items they contain.

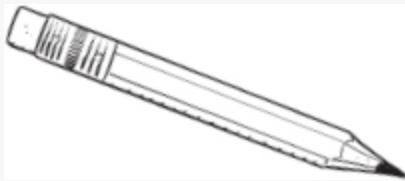
If we were asked to draw a picture of what a tuple might look like in memory, it would look like this.



The slots are numbered sequentially (from zero) so the square bracket notation can then be used to index into the tuple.

As tuples are immutable, once defined, they cannot change. So a tuple always has a fixed number of slots.

## SHARPEN YOUR PENCIL



Assume the following line of code has executed in a new, empty code cell in your notebook (hint: go ahead and execute this line of code in your notebook so you can experiment as needed):

```
data = swimclub.get_swim_data("Darius-13-100m-Fly.txt")
```

Using the square bracket notation, extract the list of swim times from the `data` tuple, assigning them to a list called `times`. Then display the contents of the `times` list. Write the code you used here:

---

---

---

Using Python's unpacking technology (aka *multiple assignment*), extract the list of swim times from the `data` tuple a second time, assigning them to a list called `times2`. Display the contents of the `times2` list. Write the code you used here:

---

---

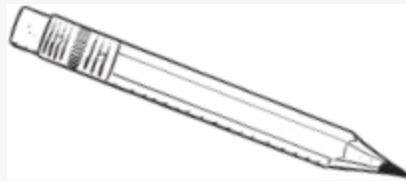
---

## THERE ARE NO DUMB QUESTIONS

**Q:** I just did a “print dir” on my data tuple and there’s only two methods that aren’t dunder. Is it the case that you can’t do anything useful with tuples?

**A:** No, that’s not the case. Now, that said, if you compare the output from the `print dir` combo mambo for a list to a tuple, it certainly appears that tuples are inferior to lists as they have next to no methods associated with them. But, recall: tuples are immutable, so you can never change a tuple once it’s assigned data (this fact alone reduces the number of methods you need with tuples). As you work through this book you’ll see examples of where it makes sense to use a tuple over a list, and vice versa. Bottom line: you need both.

## SHARPEN YOUR PENCIL SOLUTION



You were to assume the following line of code has executed in a new, empty code cell in your notebook:

```
data = swimclub.get_swim_data("Darius-13-100m-Fly.txt")
```

Using the square bracket notation, you were to extract the list of swim times from the `data` tuple, assigning them to a list called `times`. You then were to display the contents of the `times` list on screen:

Within the returned tuple,  
the list of swim times are  
in the tuple's seventh slot,  
so you use `6` inside of  
square brackets to select  
the data you need.

times = data[6]

times

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 the list of swim times from the `data` tuple a second time, assigning them to a list called `times2`. You then were to display the contents of the `times2` list on screen:

Using the unpacking powers of multiple assignment, the data values from the seven slots in the tuple are assigned to individual variable names (including "times2").

---

```
swimmer, age, distance, stroke, average, average_str, times2 = data
```

---

```
times2
```

---

The contents of "times2" can be displayed on screen just as it was with "times" above.

Isn't there an easier way to do this using multiple assignment? Can't I just ignore those unused values and target "times" without having to create all those unneeded variables?



### There's often a “better” way.

The code on the last page works and does what you expect it to. (You did try it out in your notebook, didn't you?!?). It is possible to make both examples more *Pythonic* with a couple of small changes, which you'll explore over the page.

## NOTE

“Pythonic”: when code is written to take advantage of what Python has to offer, as opposed to Python code which clearly looks like it been written to conform to some other programming language’s idea of “the right way”.

The hope is, once these changes are applied, your code won’t feel quite so unwieldy.

## BEHIND THE SCENES



### Multiple assignment, default variables, and Pythonic code, 1 of 2.

With the "data" variable created (by calling your "get\_swim\_data" function), the good ol' square bracket notation can pick out the data you need:

```
times = data[6]  
times
```

The item in  
the seventh  
slot of the  
"data" tuple.

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

The list of swim times is in the tuple's last slot, so it's  
more Pythonic (and convenient) to index from the other  
end using negative indexing with square brackets.

```
times = data[-1]  
times
```

The item in the  
last slot of the  
"data" tuple  
(which is the  
same item in  
"data[6]").

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

If you want to avoid square brackets altogether, the use of multiple assignment assigns each of the tuple's items to their own variable name.

```
swimmer, age, distance, stroke, average, average_str, times2 = data  
times2
```

"times2" is the same list of swim times.

If all you need is this list of swim times, it's more Pythonic to replace those unused individual variable names with the default variable

```
, , , , , times2 = data  
times2
```

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

"times2" is still the same list of swim times.

## BEHIND THE SCENES



### Multiple assignment, default variables, and Pythonic code, 2 of 2.

If you want to remember the rest of the data, but not as individual variables, it's more Pythonic to rewrite the previous code to pack the rest of the data into a different list (called "the\_rest" here). Note the variable's name is prefixed with \* which enables this behavior.

```
*the_rest, times3 = data  
times3
```

['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'] ← Still the same list of swim times.

```
the_rest
```

['Darius', '13', '100m', 'Fly', 8657.5, '1:26.58'] ←

The rest of the data values from "data" in a new list called "the\_rest".

If you do not want to keep the rest of the data in its own list variable, you can assign it (in one go) to the default variable. This is also a very Pythonic thing to do. Note that in this form of multiple assignment (as in the previous code, above) it doesn't matter how many other items are in the data list. The last item is always assigned to "times4", with whatever's left assigned to the variable.

```
*_, times4 = data  
times4
```

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

The rest  
of "data"  
is in the  
default  
variable.

```
['Darius', '13', '100m', 'Fly', 8657.5, '1:26.58']
```

## WATCH IT!



**Be careful: don't rely on the default variable.**

*The use of the default variable is very Pythonic. However, care is needed. Even though you can assign values to the default variable (the single underscore), you can't rely on the value staying assigned to \_ for any length of time. Python (and Jupyter) uses \_ behind the scenes in lots of places, so its value can change... and you won't know. Yikes.*

*If you need to remember a value, assign it to a variable with a proper variable name. Don't rely on the default variable.*

I'm told your code now works with any of my swimmer's data files? That's great to hear. I can't wait to use your system to help me identify those swimmers who aren't giving me 100%. What's up next?



### A list of filenames would be nice.

Your `get_swim_data` function, part of the `swimclub` module, takes any swimmer's filename and returns a tuple of results to you.

What's needed now is the full list of filenames which you should be able to get from your underlying operating system. As you can imagine,

Python has you covered.

## Let's determine the list of filenames

When it comes to working with your operating system (whether you're on *Windows*, *macOS*, or *Linux*), the PSL has you covered. The `os` module lets your Python code talk to your operating system in an platform-independent way, and you'll now use the `os` module to grab a list of the files in the `swimdata` folder.

Be sure to follow along in your `Files.ipynb` notebook.

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.

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 `swim_files` list on screen:



Type this into an empty  
code cell, then press  
"Shift+Enter".

```
print(swim_files)
```

```
['Hannah-13-100m-Free.txt', 'Darius-13-100m-Back.txt', 'Owen-15-100m-Free.txt', 'Mike-15-100m-Free.txt',  
'Hannah-13-100m-Back.txt', 'Mike-15-100m-Back.txt', 'Mike-15-100m-Fly.txt', 'Abi-10-50m-Back.txt', 'Ruth-  
13-200m-Free.txt', '.DS_Store', 'Tasmin-15-100m-Back.txt', 'Erika-15-100m-Free.txt', 'Ruth-13-200m-  
Back.txt', 'Abi-10-50m-Free.txt', 'Maria-9-50m-Free.txt', 'Elba-14-100m-Free.txt', 'Tasmin-15-100m-  
Free.txt', 'Abi-10-100m-Back.txt', 'Abi-10-50m-Breast.txt', 'Mike-15-200m-IM.txt', 'Ali-12-100m-Back.txt',  
'Ruth-13-100m-Back.txt', 'Chris-17-100m-Back.txt', 'Ali-12-100m-Free.txt', 'Darius-13-100m-Breast.txt',  
'Ruth-13-100m-Free.txt', 'Aurora-13-50m-Free.txt', 'Katie-9-100m-Breast.txt', 'Alison-14-100m-Breast.txt',  
'Ruth-13-400m-Free.txt', 'Emma-13-100m-Free.txt', 'Calvin-9-50m-Fly.txt', 'Darius-13-100m-Fly.txt', 'Mike-  
15-200m-Free.txt', 'Emma-13-100m-Breast.txt', 'Tasmin-15-100m-Breast.txt', 'Blake-15-100m-Free.txt', 'Abi-  
10-100m-Breast.txt', 'Chris-17-100m-Breast.txt', 'Blake-15-100m-Back.txt', 'Katie-9-100m-IM.txt', 'Bill-18-  
200m-Back.txt', 'Darius-13-200m-IM.txt', 'Dave-17-100m-Free.txt', 'Alison-14-100m-Free.txt', 'Lizzie-14-  
100m-Free.txt', 'Katie-9-50m-Fly.txt', 'Katie-9-50m-Breast.txt', 'Katie-9-50m-Back.txt', 'Lizzie-14-100m-  
Back.txt', 'Tasmin-15-200m-Breast.txt', 'Katie-9-50m-Free.txt', 'Dave-17-200m-Back.txt', 'Erika-15-200m-  
Breast.txt', 'Calvin-9-50m-Back.txt', 'Calvin-9-50m-Free.txt', 'Carl-15-100m-Back.txt', 'Bill-18-100m-  
Back.txt', 'Katie-9-100m-Free.txt', 'Blake-15-100m-Fly.txt', 'Erika-15-100m-Breast.txt', 'Katie-9-100m-  
Back.txt']
```

The 61 data values  
currently in the  
"swim\_files" list.



**What a great idea.**

Let's use the combo mambo to see what's built into lists.

## What can you do to lists?

Here's the `print dir` combo mambo output for your `swim_files` list:

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

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 which can help here. The **sorted** BIF returns a ordered copy of your list's data, all the while leaving the existing list ordering intact. There's no undo here, either, as your original list is unchanged.*

What's this then?

```
print(sorted(swim_files))
```

```
['.DS_Store', 'Abi-10-100m-Back.txt', 'Abi-10-100m-Breast.txt', 'Abi-10-50m-Back.txt', 'Abi-10-50m-Breast.txt', 'Abi-10-50m-Free.txt', 'Ali-12-100m-Back.txt', 'Ali-12-100m-Free.txt', 'Alison-14-100m-Breast.txt', 'Alison-14-100m-Free.txt', 'Aurora-13-50m-Free.txt', 'Bill-18-100m-Back.txt', 'Bill-18-200m-Back.txt', 'Blake-15-100m-Back.txt', 'Blake-15-100m-Fly.txt', 'Blake-15-100m-Free.txt', 'Calvin-9-50m-Back.txt', 'Calvin-9-50m-Fly.txt', 'Calvin-9-50m-Free.txt', 'Carl-15-100m-Back.txt', 'Chris-17-100m-Back.txt', 'Chris-17-100m-Breast.txt', 'Darius-13-100m-Back.txt', 'Darius-13-100m-Breast.txt', 'Darius-13-100m-Fly.txt', 'Darius-13-200m-IM.txt', 'Dave-17-100m-Free.txt', 'Dave-17-200m-Back.txt', 'Elba-14-100m-Free.txt', 'Emma-13-100m-Breast.txt', 'Emma-13-100m-Free.txt', 'Erika-15-100m-Breast.txt', 'Erika-15-100m-Free.txt', 'Erika-15-200m-Breast.txt', 'Hannah-13-100m-Back.txt', 'Hannah-13-100m-Free.txt', 'Katie-9-100m-Back.txt', 'Katie-9-100m-Breast.txt', 'Katie-9-100m-Free.txt', 'Katie-9-100m-IM.txt', 'Katie-9-50m-Back.txt', 'Katie-9-50m-Breast.txt', 'Katie-9-50m-Fly.txt', 'Katie-9-50m-Free.txt', 'Lizzie-14-100m-Back.txt', 'Lizzie-14-100m-Free.txt', 'Maria-9-50m-Free.txt', 'Mike-15-100m-Back.txt', 'Mike-15-100m-Fly.txt', 'Mike-15-100m-Free.txt', 'Mike-15-200m-Free.txt', 'Mike-15-200m-IM.txt', 'Owen-15-100m-Free.txt', 'Ruth-13-100m-Back.txt', 'Ruth-13-100m-Free.txt', 'Ruth-13-200m-Back.txt', 'Ruth-13-200m-Free.txt', 'Ruth-13-400m-Free.txt', 'Tasmin-15-100m-Back.txt', 'Tasmin-15-100m-Breast.txt', 'Tasmin-15-100m-Free.txt', 'Tasmin-15-200m-Breast.txt']
```



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 always a concern.

Before moving on, it's important to *remove* that unwanted filename from the `swim_files` list.

## EXERCISE



Here's the list of built-in methods available when working with lists:

```
'append', 'clear', 'copy', 'count', 'extend', '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:

---

---

## RELAX



Don't stress yourself if, as a result of completing the above exercise, you identify more than one way to accomplish the assigned task. This is OK, as sometimes the same outcome can be accomplished in many different ways. There's rarely an absolutely right way to do something. As with most things, you should concentrate on getting your code to do what you need it to do *first* before attempting any sort of optimization.



**There sure is.**

You'll get to see most of the list built-in methods in action as you progress through this book. However, for now, concentrate on just enough to complete the exercise.

## EXERCISE SOLUTION



You were shown the list of built-in methods available when working with lists:

```
'append', 'clear', 'copy', 'count', 'extend', '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 always going to be there.

**Q:** Can't we avoid this issue by asking the Coach to use something other than a Mac for this?

**A:** We asked, and the Coach told us he'd rather eat glass...

**Q:** I can see the relationship between tuples and lists, in that they are somewhat similar. I guess a list is also a sequence, but it is immutable?

**A:** Lists are indeed a sequence, but – no – lists are not immutable. Rather, lists are *mutable*, in that they can dynamically change as your code runs (unlike tuples).

**Q:** I guess lists are way more useful than tuples, right?

**A:** It depends. Both lists and tuples have their uses. Overall, lists see more action than tuples, in that lists support many more use-cases. But it's not the case that lists are “better” or “more useful” than tuples: lists and tuples are designed for different purposes.

## TEST DRIVE



Of the eleven methods built-into every Python list, the one which caught our eye was **remove**.

Here's what the **help** BIF reported about **remove**:

```
help(swim_files.remove)
```

Help on built-in function remove:

remove(value, \_) method of builtins.list instance

Remove first occurrence of value.

This method  
looks like what  
we need.

Raises ValueError if the value is not present.

Sure enough, when the **remove** method is invoked against the **swim\_files** list (with **.DS\_Store** given as the filename ear-marked for the chop) the list *shrinks* from 61 items to 60:

Identify the file  
to remove from  
the list.

```
swim_files.remove('.DS_Store')
```

Don't worry. The  
filename is removed  
from your list, but  
still exists on your  
hard-drive.

Check the length  
of the list after  
the removal.

```
60
```

This is more like it. The  
number of filenames in the  
"swim\_files" list now matches  
what we were expecting.



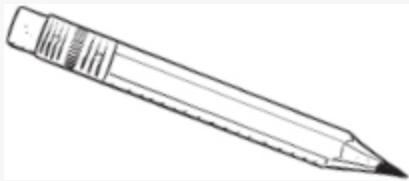
Now that we've got a list of 60 filenames, what's the plan? Are we ready to create some bar charts for the Coach yet?!?

## **That would be nice, wouldn't it?**

We could throw caution to the wind and dive into creating some bar charts, but it might be too soon for that.

Your `get_swim_data` function has worked so far, but can you be sure it'll work for *any* swimmer's file? Let's spend a moment ensuring our `get_swim_data` function works as expected no matter the data file it's presented with.

## SHARPEN YOUR PENCIL



Let's see if you can confirm your `get_swim_data` function works with any swimmer's file. Write a **for** loop which processes the 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 `get_swim_data` function with the current filename. You do not need to display the data returned from your function, but you do need to call it.

Write the code you used here and note down anything you learned from running your **for** loop in the space below:

Put your  
code here.



---

---

---

---

---

---

---

---

---

Put your  
notes here.



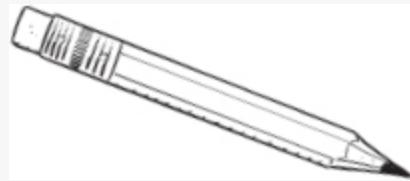
---

---

---

---

## SHARPEN YOUR PENCIL SOLUTION



You were to try out your `get_swim_data` function on all the files in your `swim_files` list..

You were to write a `for` loop which processes the filenames in `swim_files` one at a time. On each iteration, you were to display the name of the file currently being processed, then arrange to call your `get_swim_data` function with the current filename. You didn't need to display the data returned from your function, but you do need to call it.

Here's the code which we came up with after experimenting in our `Files.ipynb` notebook, together with the output generated by the code:

The "for" loop cycles through the data in the "swim\_files" list, displaying the name of the current file being processed, then calls the "get\_swim\_data" function. If something goes wrong, the last filename displayed is the one which caused the issue.

```
for s in swim_files:  
    print("Processing:", s)  
    swimclub.get_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

None of the files in this collection generated an error.

This is the name of the file which resulted in an error.

```
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)  
=> 3     swimclub.get_swim_data(s)  
  
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 X
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 which 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 pre-process each data file to make sure there's no missing minutes, perhaps by prefixing a zero and a colon when the minutes are missing? That way, we won't have to change any code.

**Mara:** That would work, but...

**Sam:** ...it would be messy. Also, I'm not too keen to preprocessing all the files, as the vast majority won't need to be changed, which feels like it might be wasteful.

**Mara:** And although, as a strategy, we wouldn't have to change any existing code, we would have to create the code to do the pre-processing, perhaps as a separate utility.

**Sam:** Recall, too, that the data is in a fixed format, and that it's generated by the Coach's smart stopwatch. We really shouldn't mess with the data, so let's leave it as is.

**Alex:** So, we're looking at changing our `get_swim_data` function, then?

**Mara:** Yes, I think that's a better strategy.

**Sam:** Me, too.

**Alex:** So, what do we need to do?

**Mara:** We need to identify where in our code the changes need to be made...

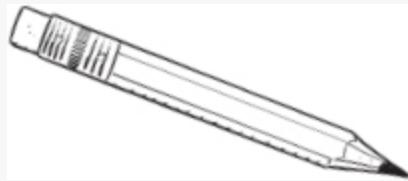
**Sam:** ...and what those code changes need to be.

**Alex:** OK, sounds good. So we're going to take a closer look at our `get_swim_data` function so we can decide what code needs to change?

**Mara:** Yes, then we can use an `if` statement to make a decision based on whether or not the swim time currently being processed has a minute value.



## SHARPEN YOUR PENCIL



Here's the most recent version of the code from your `swimclub.py` module.

Grab your pencil and encircle the area of the code you think needs to incorporate an `if` statement:

```
import statistics

FOLDER = "swimdata/"

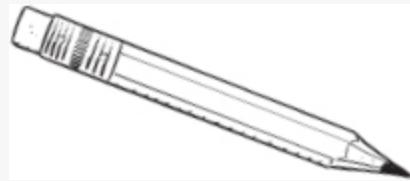
def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

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

## NOTE

Flip the page to see our selection. →

## SHARPEN YOUR PENCIL SOLUTION



You were shown the most recent version of the code from your `swimclub.py` module.

Grabbing your pencil, you were asked to encircle the area of the code you think needs to incorporate an `if` statement:

```

import statistics

FOLDER = "swimdata/"

def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
        times = data[0].strip().split(",")
        converts = []
        for t in times:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
            converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

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

```

This is where we think you need to put the "if" statement, as you need to decide which of the two swim time formats you're currently processing.

## Decisions, decisions, decisions

That's what **if** statements do, day-in and day-out: they make decisions.



**Yes, that is what's needed here.**

Let's take a closer look at the two possible swim time formats.

First up, here is one of the times recorded for Darius in his file:

'1:30.96'

And here's a time taken from Abi's data:

'43.35'

It's easy to spot the difference: *Abi's data doesn't show any minutes*. With this in mind, it's possible to come up with a condition to check when making a decision. Can you work out what it is? (Hint: consider your BFF, the colon).

## Let's look for the colon "in" the string

If the colon appears in any swim time string, then the time has a minute value. Although strings come with lots of built-in methods, including methods which can perform a search, let's not use any of these here. As searching is such a common requirement, Python provides the **in** operator. You've seen **in** before, with **for**:

### NOTE

The "find" and "index" string methods both perform searching.

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**:

Use "in" to check for the existence of a substring within a larger string.

"ell" in "Hello there!"

True

← "True" signifies success!

"fell" in "Hello there!"

False

← "False" means failure.

Search for a value within a list of values.

42 in ["Forty two", 42, "42"]

True

This one might have you scratching your head... but you're searching for the string "two" in the first slot of the list. Recall, [0] is the first slot.

"two" in ["Forty two", 42, "42"][0]

True

Does the list on the left of the "in" keyword appear within the list on the right?

[1, 2, 3] in ['a', 'b', 'c', [1, 2, 3], 'd', 'e', 'f']

True

[1, 2] in ['a', 'b', 'c', [1, 2, 3], 'd', 'e', 'f']

False

Wow. The “in” keyword really is powerful, as I was able to perform those six searches without once having to write a custom loop. You just gotta love it!



We love the `in` keyword, too.

It's a Python superpower.

## EXERCISE



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 which checks to see when the colon appears in the current swim time (which is stored in a variable named `t`)? Fill-in the blank space below:

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

What condition goes in here?

This code block runs when the condition evaluates to "True".

## EXERCISE SOLUTION



Here's the general structure of a simple **if** statement:

```
if <some condition>:  
    Code to execute when <some condition> is True  
else:  
    Code to execute when <some condition> is False
```

Now that you know about the **in** keyword, you were asked to come up with a conditional statement which checks to see when the colon appears in the current swim time (which is stored in a variable named **t**)? You were to fill-in the blank space below:

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!



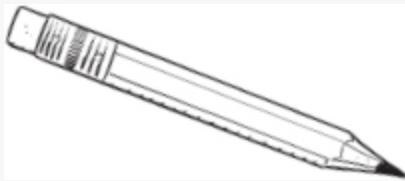
### Be sure to write your if statements in a Pythonic way

*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(".")
```

## SHARPEN YOUR PENCIL



There's one final task, and that's to work out the code which runs when the condition evaluates to **False**. Here's your **if** statement so far. Can you come up with the code which needs to be added to the **else** code block? Experiment in your notebook, then add the code below. As always, our solution is over the page.

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

Add code in →  
here to execute  
when the "else"  
block runs.

## THERE ARE NO DUMB QUESTIONS

**Q:** I'm guessing "True" and "False" are Python's boolean values? Can I use 1 and 0 in their place?

**A:** Sort of. Within a *boolean context*, 1 evaluates to **True** and 0 evaluates to **False**. That said, Python programmers rarely use 1 and 0 in this way. This has to do with the fact that any value in Python can be used within a boolean context.

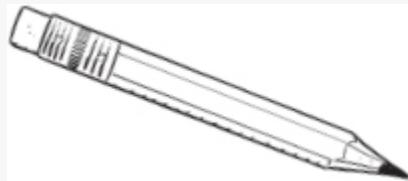
**Q:** Is there a way to check what a value evaluates to within a boolean context?

**A:** Yes, there's a BIF for that. It's called **bool**. You can use it interactively (or in your code) to check any value's boolean evaluation.

**Q:** What about using "true" and "false", i.e., all lowercase? How do those values evaluate?

**A:** Not the way you might expect. Using **true** and **false** is a bad idea, as case is significant. **True** is a boolean value, as is **False**, whereas **true** and **false** are not. They're valid variable names, *not* booleans. And as variables, they exist, which means they'll always evaluate to **True** (how can something that exists be anything other than **True**?). And, yes, we agree that something called **false** evaluating to **True** is a little on the weird side. Life lesson: don't ever use lowercase **true** nor **false** as variable names.

## SHARPEN YOUR PENCIL SOLUTION



You had one final task, and that was to work out the code which runs when the condition evaluates to **False**. You were given your **if** statement so far, and were to come up with the code which needs to be added to the **else** code block. Here's what we used. Is your code similar, the same, or entirely different?

```
if ":" in t:  
    minutes, rest = t.split(":")  
    seconds, hundredths = rest.split(".")  
  
else:  
    minutes = 0  
    seconds, hundredths = t.split(".")
```

When the colon is not found in the time string, the "minutes" value is set to zero.

When there are no minutes recorded, you need to break apart the time string (stored in the "t" variable) on the "." character. This gives you the "seconds" and "hundredths" values.



### **Absolutely, after a bit of housekeeping.**

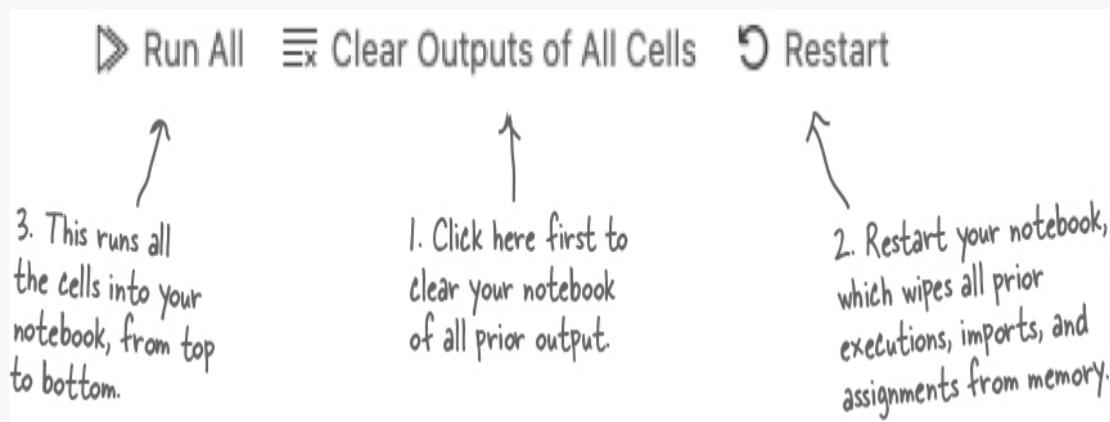
Be sure to add the above code to your `get_swim_data` function within your `swimclub.py` module, and don't forget to **save** your file.

With that done, return to your notebook and follow along with the *Test Drive* on the next page.

## TEST DRIVE



In order to use the latest version of your `swimclub` module, you need to reload it. The easiest way to do this is to clear all of your notebook's cell, restart your notebook, then run every cell. At the top of VS Code, notice the following options, which you should click on in the order shown (1, then 2, and then 3):



Once you complete the three steps above, your `for` loop produces (hopefully) 60 lines of output. This time, hopefully, there is no runtime error:

```
for s in swim_files:  
    ...print("Processing:", s)  
    ...swimclub.get_swim_data(s)
```

This time, Abi's  
file is processed  
without  
generating an  
error. Whoo hoo!



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

:

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**:

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, and "n" holds the current enumeration.

```
for n, s in enumerate(swim_files, 1):  
    print(n, "Processing:", s)  
    swimclub.get_swim_data(s)
```

The "enumerate" BIF defaults to zero as its starting value so, in this case, we've asked it to start counting from one.

```
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  
10 Processing: Tasmin-15-100m-Back.txt
```

:

```
58 Processing: Blake-15-100m-Fly.txt  
59 Processing: Erika-15-100m-Breast.txt  
60 Processing: Katie-9-100m-Back.txt
```

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:** This may be nitpicking, but why did we do a restart on our notebook? Surely typing `import swimclub` into an empty cell reloads the previously imported module's newest code?

**A:** (We hope you're sitting down...) No, it doesn't. The Python interpreter implements an aggressive caching scheme for imported modules which, for all intents and purposes, effectively forbids the re-importation of an already imported module even if the module's code has changed in the meantime. Although there are some techniques for overriding this default module caching behavior, in our experience, the safest thing to do is to *always* restart a notebook after a change to a module's code. This way you are guaranteed to be running the code you think you're running.

```

import statistics

FOLDER = "swimdata/"

def get_swim_data(fn):
    """Given the name of a swimmer's file, extract all the required data, then
    return it to the caller as a tuple."""
    swimmer, age, distance, stroke = fn.removesuffix(".txt").split("-")
    with open(FOLDER + fn) as df:
        data = df.readlines()
    times = data[0].strip().split(",")
    converts = []
    for t in times:
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average_str = str(minutes) + ":" + str(seconds) + "." + hundredths

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

```

Here is the most-recent version of our "swimclub.py" code. Make sure your code is the same as this before moving on.

## The Coach's code is taking shape...

Your `swimclub` module is now ready. Given the name of a file which contains a collection of swim time strings, your new module can produce usable data. The Coach is expecting to see some bar charts created from this data, so let's dive into implementing that functionality in the next chapter.

As always, you can move on after you've reviewed the chapter summary, then tried your hand at this chapter's crossword.



## CHAPTER REVIEW BULLET POINTS

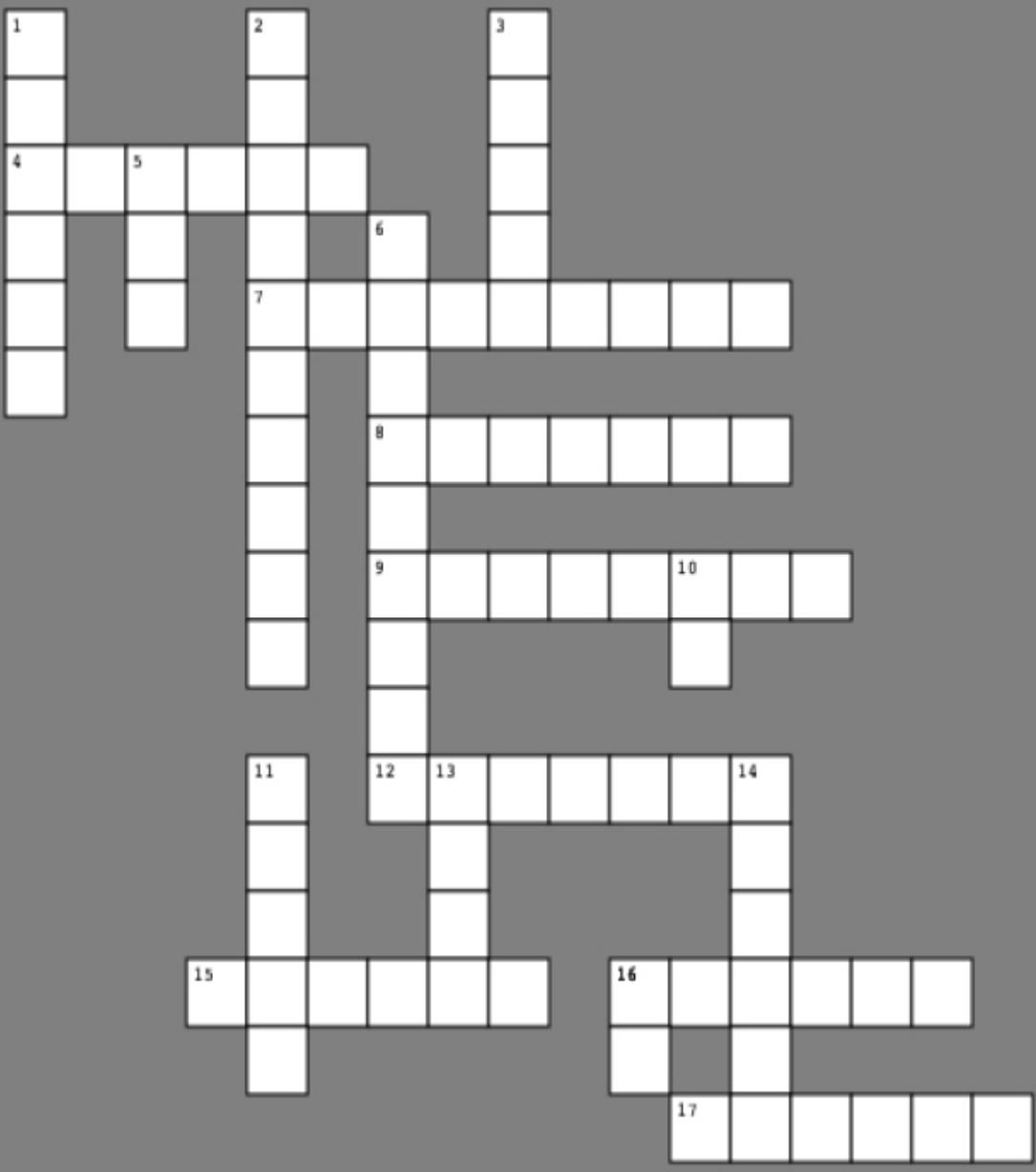
- The **def** keyword defines a new, bespoke function.
- **Triple-quoted** strings are often used to add a multi-line comments to code, and are often found at the top of a function's code block (right after the function signature).
- When you put code in it's own file (with a `.py` extension), you create a **module**.
- The **import** statements lets you reuse a module, e.g., `import swimclub`.
- Use a **fully qualified name** to invoke a function from a module, e.g., `swimclub.get_swim_data`.
- The **return** statement allows a bespoke function to return a result.
- If a function tries to return more than one result, the collection of returned values are **bundled together** as a single **tuple**. This is due to the fact that Python function's only ever return a single result.
- A tuple is an **immutable sequence** data structure. Once a tuple is assigned values, the tuple cannot change.
- Lists are like tuples, expect for the fact that lists are **mutable**.
- The **os** module (included as part of the PSL) lets your Python code talk to your underlying operating system.
- Although lists come with a handy **sort** method, be careful using it as the ordering is applied *in-place*. If you want to keep any list's current order, use the **sorted** BIF instead.
- Lists come built-in with lots of methods (not just **sort**), including the useful **remove** method.
- The **in** operator is one of our favorites, and should be one of yours, too. It's great at searching (aka *checking for membership*).

- When you need to make a decision, nothing beats the **if else** combo.
- An often overlooked, but truly wonderful, BIF is **enumerate**. It can be used to number the iterations of any **for** loop.

## THE MODULE CROSSWORD



*The answers to the clues are found in this chapter's pages, and the solution is, as always, on the next page.*



### Across

4. A place to put 9 across.
7. A BIF which provides numbers on iteration.
8. From `os`: list a folder's content.
9. A named chunk of code.
12. The underscore is the \_\_\_\_\_ variable.

15. BIF for applying order to your data (while leaving the current ordering untouched).

16. Loads code from a file.

17. Can start the last line of 9 across.

## Down

1. Finds and deletes a value from a list.

2. What you get when you've not enough values to unpack.

3. It's like an immutable list.

5. This keyword introduces 9 across.

6. Be explicit with a fully \_\_\_\_\_ name.

10. Used when making decisions.

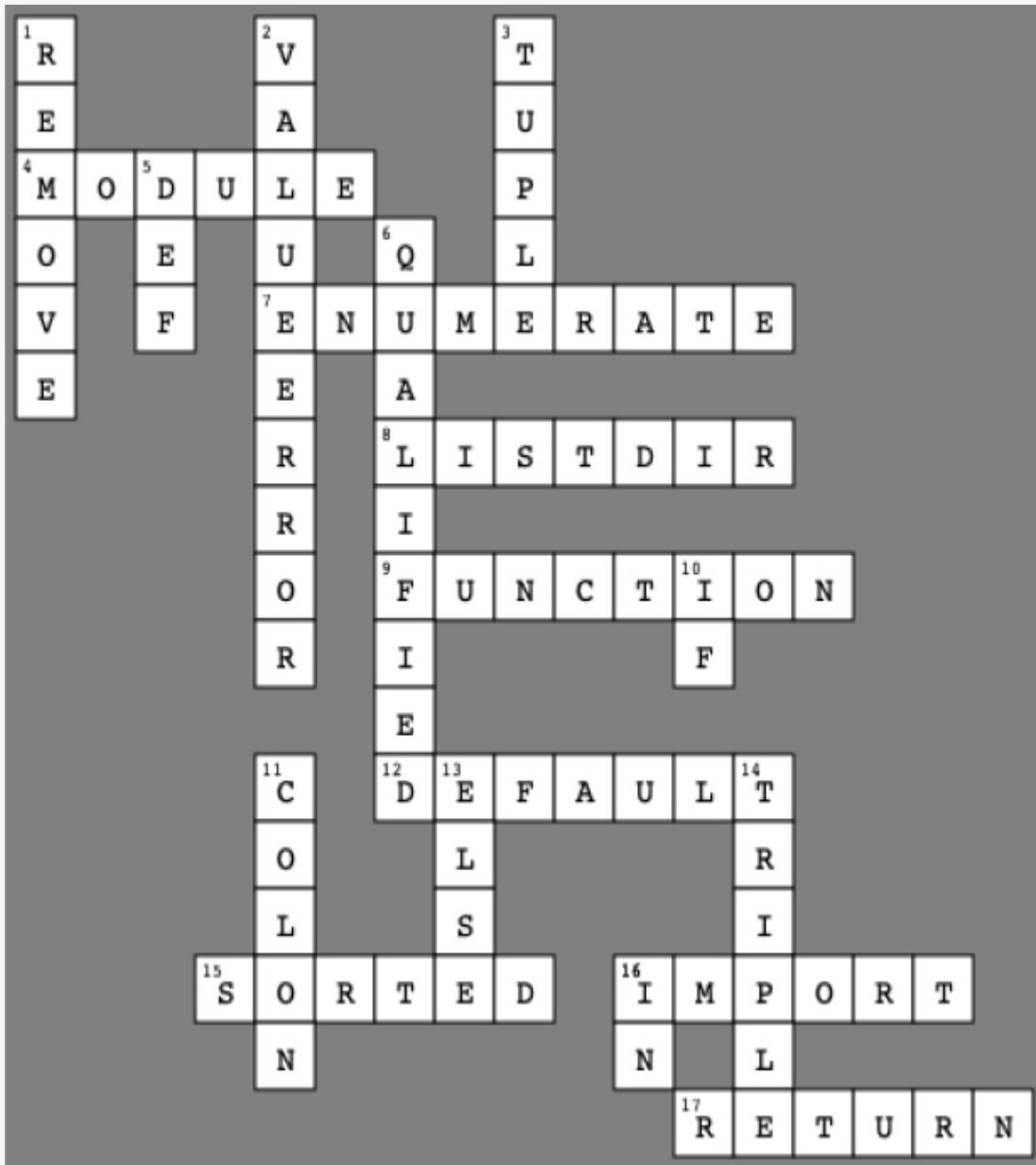
11. Your new BFF.

13. The false part of 10 down.

14. \_\_\_\_\_ quotes, looks like: """.

16. A powerful little operator.

# THE MODULE CROSSWORD SOLUTION



## Across

4. A place to put 9 across.
7. A BIF which provides numbers on iteration.
8. From `os`: list a folder's content.
9. A named chunk of code.
12. The underscore is the \_\_\_\_\_ variable.
15. BIF for applying order to your data (while leaving the current ordering untouched).
16. Loads code from a file.
17. Can start the last line of 9 across.

## Down

1. Finds and deletes a value from a list.
2. What you get when you've not enough values to unpack.
3. It's like an immutable list.
5. This keyword introduces 9 across.
6. Be explicit with a fully \_\_\_\_\_ name.
10. Used when making decisions.
11. Your new BFF.
13. The false part of 10 down.
14. \_\_\_\_\_ quotes, looks like: """.
16. A powerful little operator.

## About the Author

**Paul Barry** has a B.Sc. in Information Systems, as well as an M.Sc. in Computing. He also has a postgraduate qualification in Learning and Teaching. Paul has worked at The Institute of Technology, Carlow since 1995, and lectured there since 1997. Prior to becoming involved in teaching, Paul spent a decade in the IT industry working in Ireland and Canada, with the majority of his work within a healthcare setting.