

O'REILLY®

Head First

Python

A Learner's Guide to
the Fundamentals of
Python Programming

Paul Barry

**Early
Release**

RAW &
UNEDITED



Third Edition
Covers Python 3



A Brain-Friendly Guide

Head First Python

THIRD EDITION

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

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

Paul Barry

Head First Python

by Paul Barry

Copyright © 2023 Paul Barry. All rights reserved.

Printed in the United States of America.

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

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

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

Revision History for the Early Release

- 2023-02-08: First Release

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 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 pre-installed 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

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

Pick the download 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.

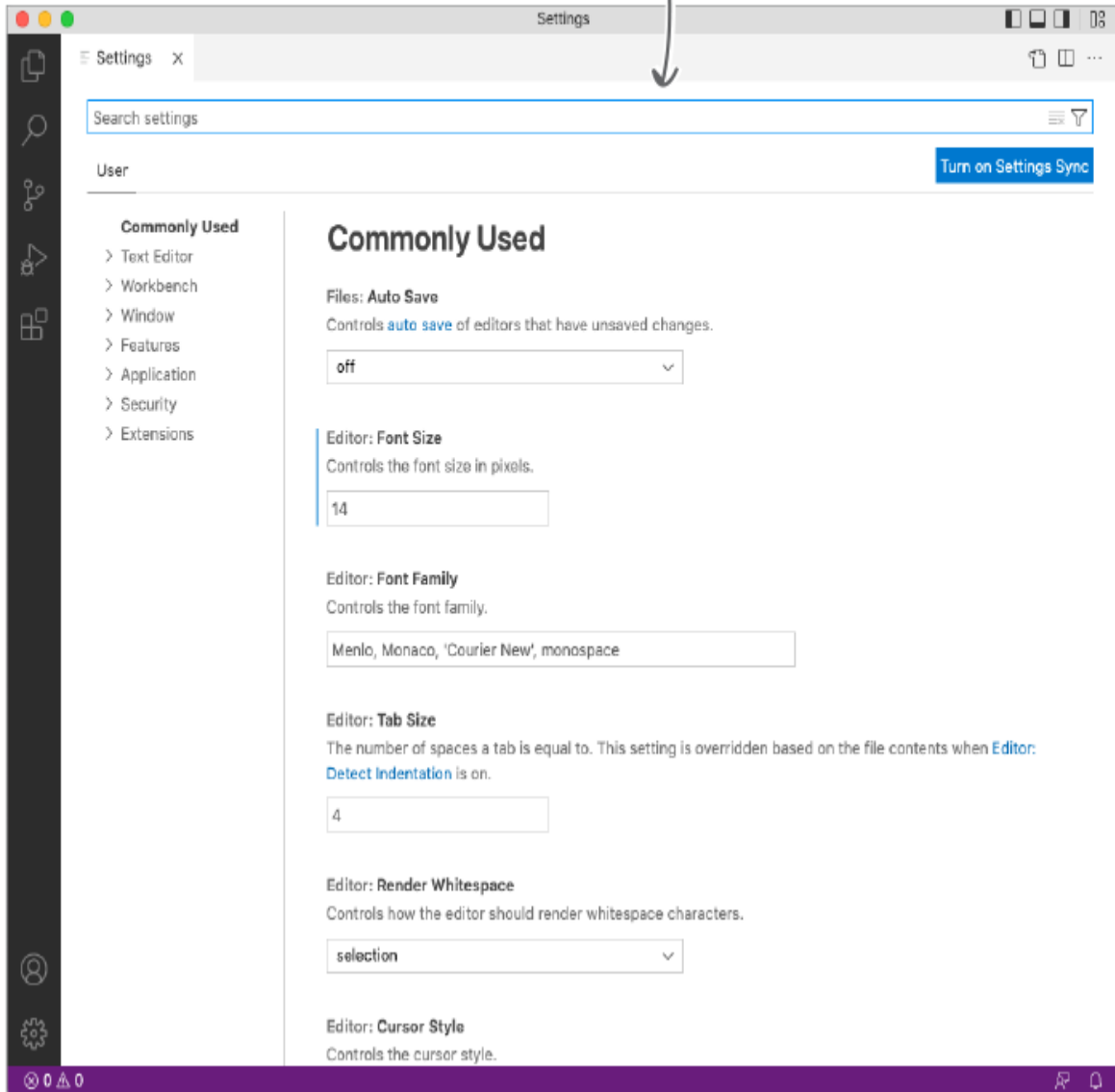
You should see something like this:

NOTE

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



Search for any setting by typing part or all of its name into this search box.



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.

← 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

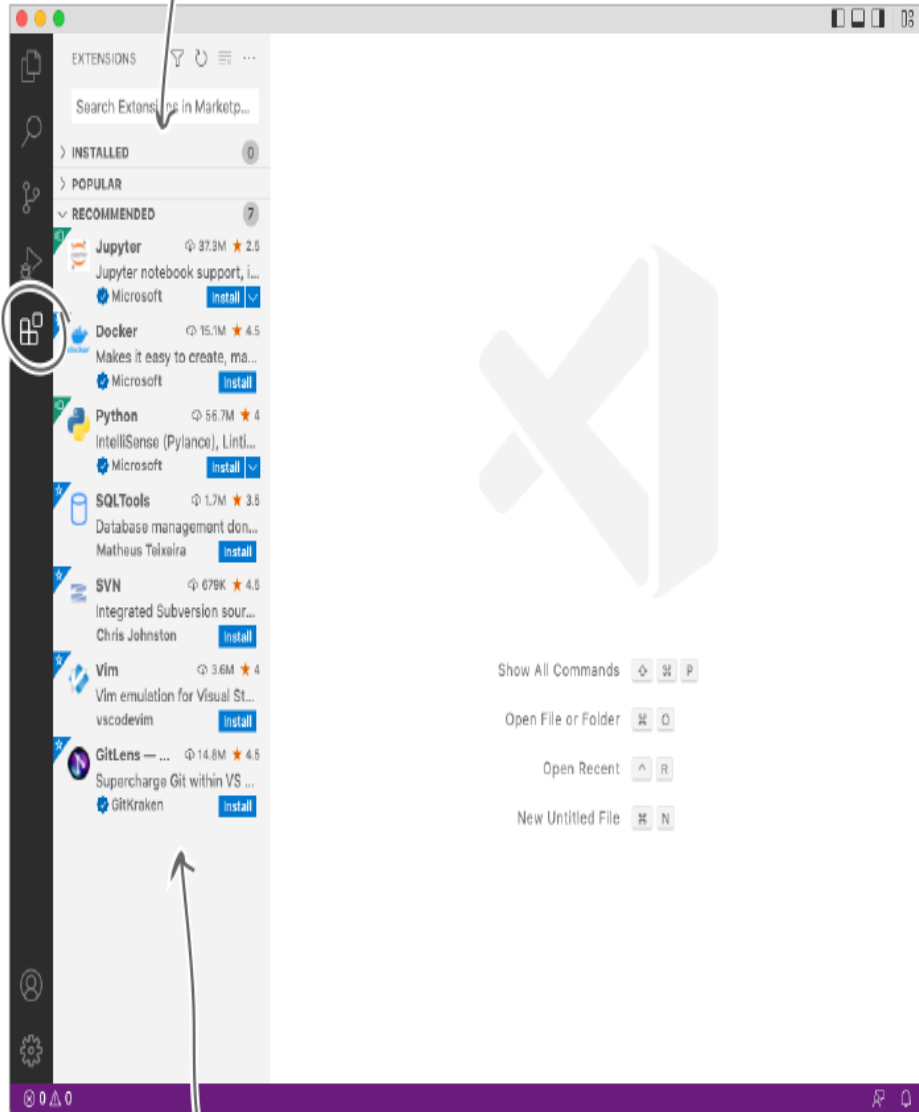
Whereas you are not obliged to copy the recommended editor setup, you absolutely have to install two VS Code extensions, namely **Python** and **Jupyter**.

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.

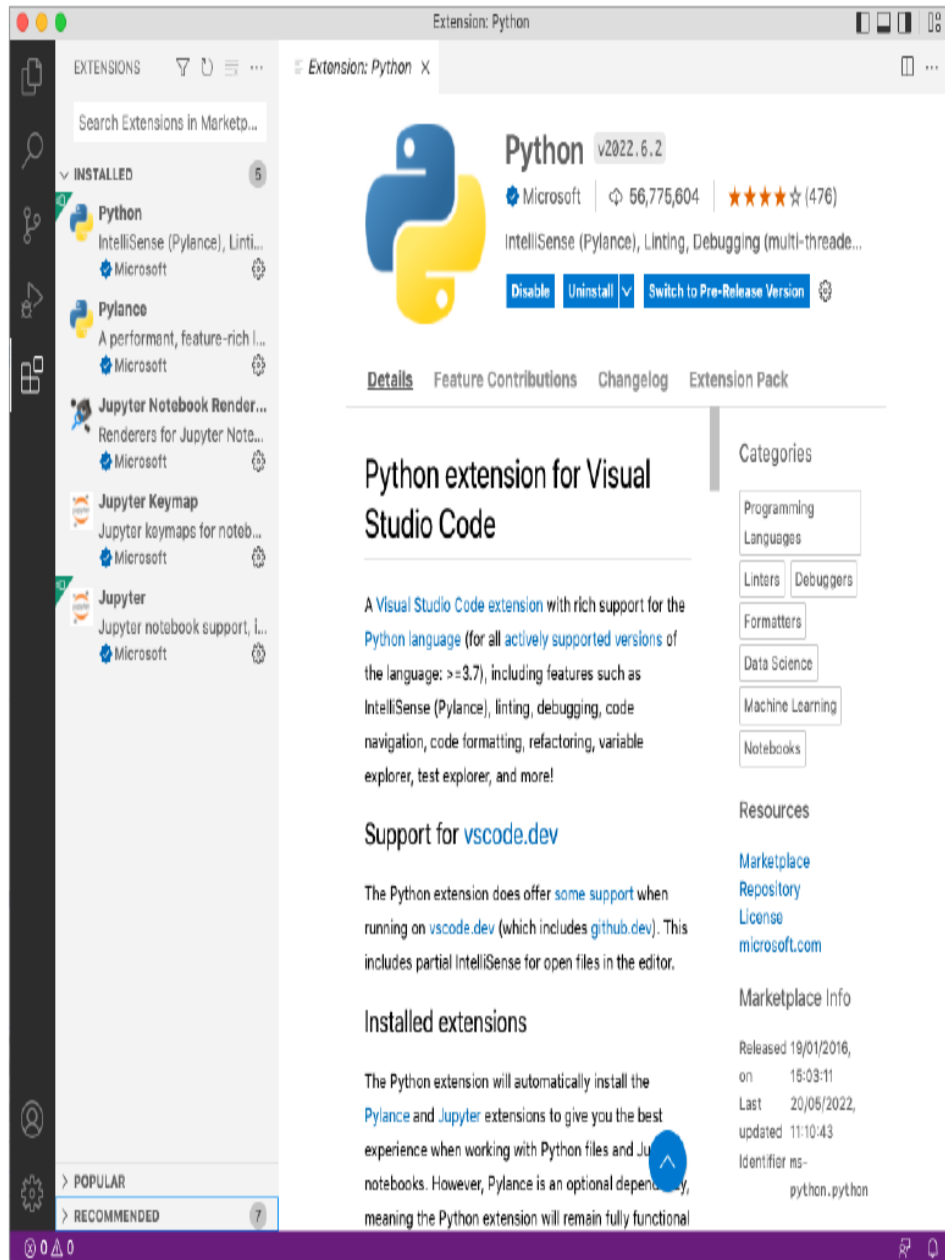


If you are seeing "Python" and "Jupyter" on your list, click the Install button to add each extension to VS Code. If you are not seeing these recommendations, use the search box to find them, then perform the install.

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).

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 the Introduction 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.



Existing programmers will recognise many pieces of Python. Python starts counting from zero. If you've programmed before, that's familiar. Same goes for if statements, loops, functions, modules, and classes – they're all here. But, Python does do some things in a *strange* way. For instance, statements typically end when a line ends, and code blocks (called **suites**) are signified via **whitespace** *not* curly-braces. This is, at first, well... *very weird*. Of course, this being *Head First*, you're going to jump right into all of this by **looking** at code, **running** code, and **experimenting** with code. *Look*, *run*, and *experiment* are cool words, as are quirky, friendly, clean, powerful, popular, easy, and fun, and every one of those words

describe Python. **Easy** and **fun**?!?! What's going on here? Isn't programming supposed to be *hard*?

Easy and fun
sounds good to me!



Cool, 'cause there's no time to waste.

We want the fun to start right away.

Additionally, we want you to confidently answer a *burning question* when you get to the end of this chapter. Specifically, this question: *Why Python?*

So, grab your pencil – yes, a *pencil* – and meet us at the top of the next page!

LOOK HOW EASY IT IS TO READ PYTHON

You don't know 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.

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

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

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

```
import random
```

```
def draw():
```

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

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

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

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

```
draw()
```

```
draw()
```

```
draw()
```

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

LOOK HOW EASY IT IS TO READ PYTHON

You don't know 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?

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

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

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

```
import random
```

```
def draw():
```

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

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

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

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

```
draw()
```

```
draw()
```

```
draw()
```

A variable called "suit" 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.

A library is included to provide support for random number generation.

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

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

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

Ditto.

Ditto.

THERE ARE NO DUMB QUESTIONS

Q: So Python uses the standard square bracket notation?

A: Yes, and no. Yes, the square bracket notation is similar in Python to how it is used in other programming languages. And, no, it's not what you'd call "standard" as Python extends what's possible with the notation in some rather cool ways. (You'll learn how soon).

Q: Doesn't Python call arrays by the name "list"?

A: Eh... yes, and no. Yes, Python uses the more generic name "list" to refer to the `suit`, `faces`, and `numbered` variable from the last example. And, no, lists are array-like, but not arrays. If you need an array, there's a library you can import called `array` which provides them.

Q: Seriously? There's no built-in array-type in Python?

A: No, there's the `list`, which is waaaay cooler. And, as we said in the last answer, the `array` library is just an import away should you really need it. But, we'll bet the farm you'll rarely need to use the `array` library once you see what lists can do.



Wrong. This code runs as-is.

In Python, variables are *not* pre-declared with typing information.

If this is the first time you've come across this, you may well regard this as *somewhat questionable*. But you should resist the urge to run for the hills,

screaming.

Here's what happens: When a variable is first used in Python code, it *must* be assigned a value, and that value (obviously) has a type, such as a number, a string, a list, or whatever. Think of it this way: Variables have no typing information associated with them, but the value they refer to does.

So... the code on the last page is ready to run, and it turns out it's easy to run, too.

Getting ready to run your code

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

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.

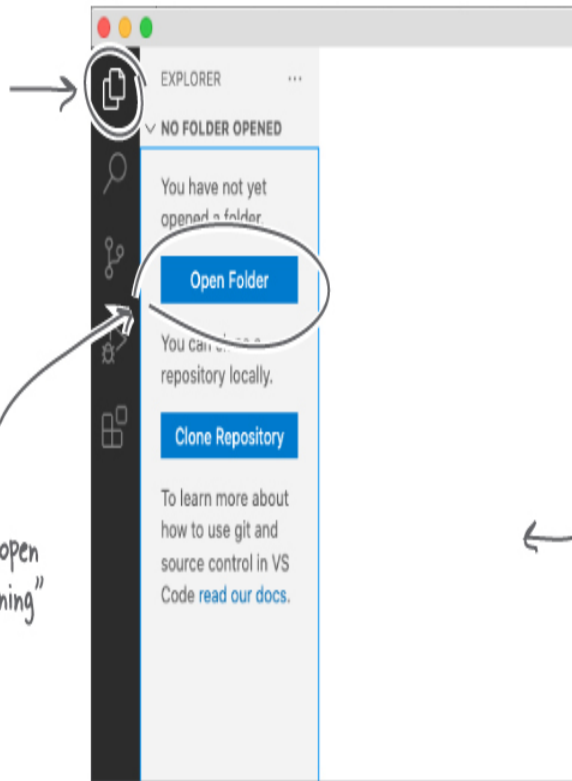
That's no biggie. Pop back to this book's *Intro* and work through the pages beginning at **Install the latest Python 3**. You'll find the instructions to install VS Code and some required extensions there.

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

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.

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

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



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

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

You'll use VS Code for everything

Whether you're creating a text file for your Python code, running Python code from a command prompt, running Python code line-by-line at a REPL, or debugging your Python code, you can do all of these activities within VS Code.

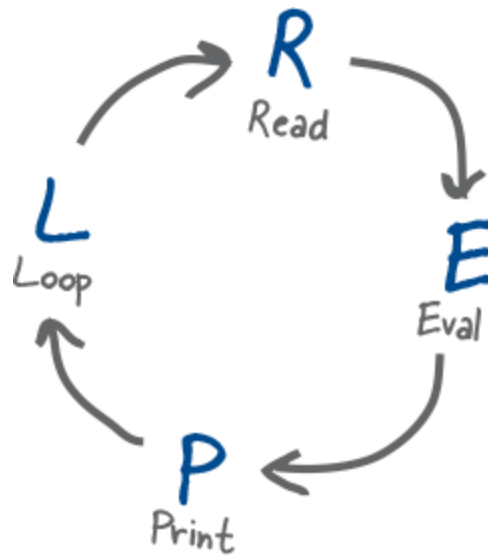


REPL stands for “read-eval-print loop”.

Pronounced “rep-ell”, a REPL is an **interactive environment** which supports the execution of a single line of code, displays it’s resulting output on screen, then iterates.

Your code is **Read** from an interactive prompt, then **Evaluated**. Any output is then **Printed** to the screen, before the system **Loops** back to get your next

line of code, then the process repeats. Ergo: REPL.

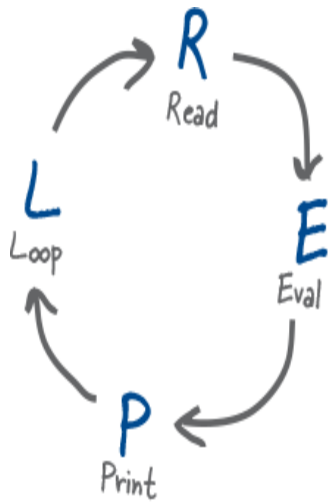


Python, Jupyter, and VS Code

Python has a basic built-in REPL which VS Code exposes to you and enhances through its extensions system. Remember that Jupyter extension you installed right after you installed VS Code in the *Intro*? Jupyter is a REPL, and then some.

It's useful to think of VS Code's Jupyter extension as a supercharged REPL, which you can't really appreciate until you experience it in action. But, don't worry, you will soon!

Before you do, note the following: Jupyter can create as many REPLs as you wish. Further, REPLs are not called "REPL" in the Jupyter universe. A Jupyter REPL is called a **notebook**.



==



Python uses the familiar "double equals" to test for equality, whereas as single equals sign signifies assignment. Again, this is similar to how many other programming languages work.



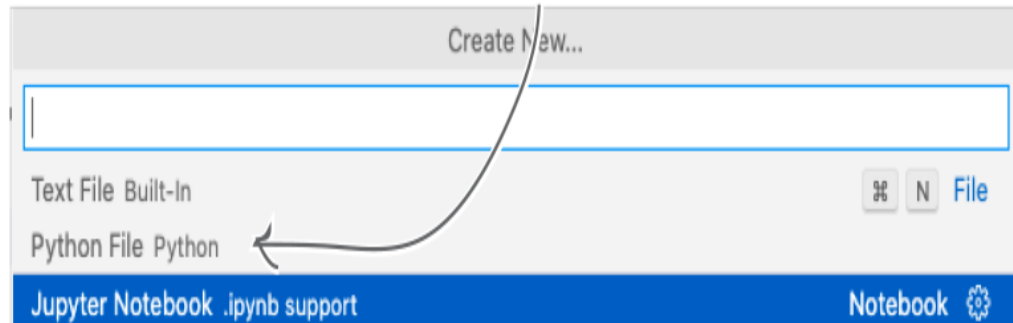
Now that you've seen how easy Python code is to read, and you understand what a REPL is, it's time to run some code!

Preparing for your first REPL 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:

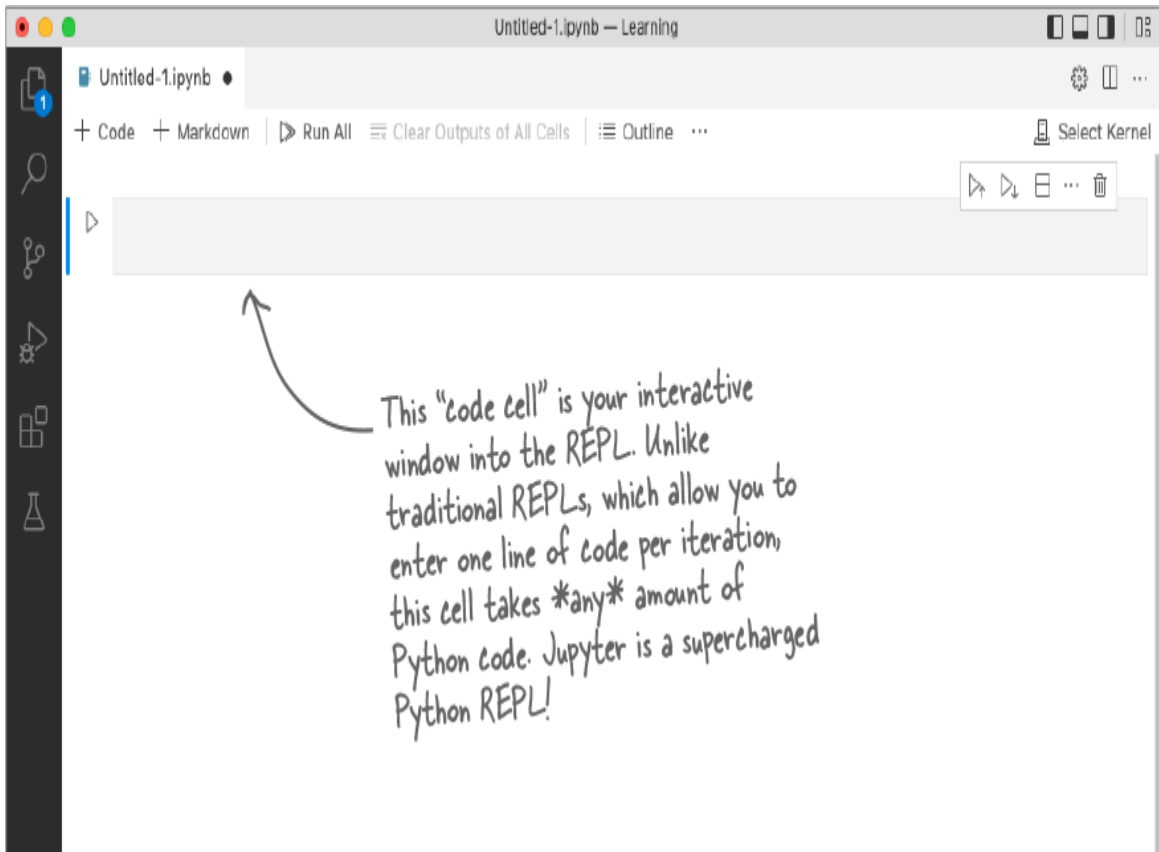
1. Select this option to create a new empty text file.

2. Select this option to create a text file to specifically contain Python code.



3. Select this option to create a new Jupyter REPL, aka a "Jupyter Notebook". This is the option you'll want to choose each time you start a new notebook, so choose this option now.

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



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:

It's useful to think of each code cell as a little embedded editor. You can type as much (or as little) Python code as you like in here.

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

Python code is easy to run!



Press Shift+Enter.

To run a code cell within a Jupyter notebook, press the **Shift** key *together* with the **Enter** key.

If all is well, three things can happen.

- **1 Your Python code executes (aka the R and E parts of REPL)**

Assuming, of course, what you typed is correct Python code. If you have syntax errors, you're told and your code **does not** run to completion. If your code runs successfully, the code cell is assigned a numeric identifier.

- **2 If your code produces output, it is displayed on screen (aka the P part of REPL)**

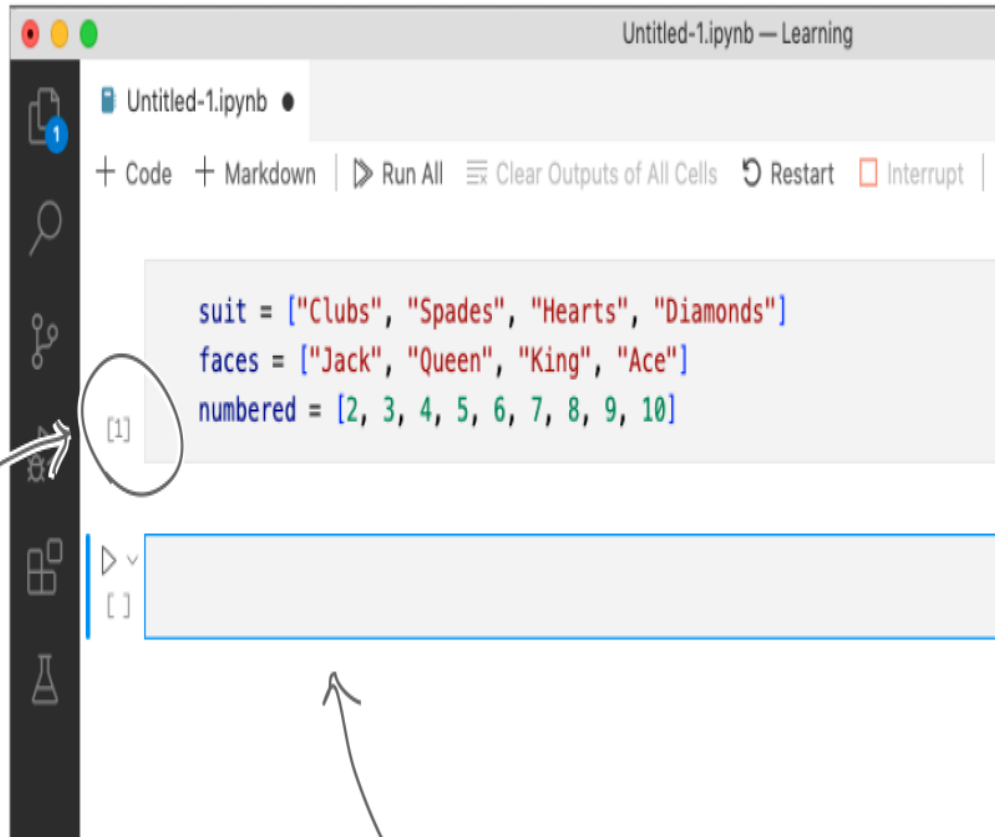
You'll see examples of this behavior in a little bit. Obviously, if your code produces no output you'll see nothing, which is what you'd expect with the cell shown above. That code defines three variables, so there's no output produced when this code runs.

- ③ **The notebook focus moves (aka the L part of REPL)**

When **Shift+Enter** works, the focus moves to the *next* cell in your notebook, assuming there is one. If there is no more cells in your notebook, a new empty code cell is created and the focus moves to it. This is what happens when you press **Shift+Enter** now. Go ahead: do it!

Pressing Shift+Enter runs your code cell

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



The cell is assigned a unique numeric identifier.

A new empty cell is created, and it's waiting patiently for you to type in more 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(suit)  
    which_type = random.choice([faces, numbered])  
    which_card = random.choice(which_type)  
    return which_card, "of", which_suit
```

Do you recall
what this
code does?

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]

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

[2]

```
import random
```

[3]

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

▶

[]

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

When you include the new empty cell at the bottom of your notebook, it now has four code cells.



You've yet to invoke draw.

Cell #3 defines your function but doesn't invoke it. This explains why cell #3 (as well as the other two) show no output: The definition of variables produces no output (cell #1) nor does the importation of a library (cell #2).

TEST DRIVE



Let's draw some cards from your card deck.

In your 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()
... ('Jack', 'of', 'Clubs')

[5] draw()
... ('Ace', 'of', 'Diamonds')

[6] draw()
... (10, 'of', 'Clubs')

Each call to the "draw" function produces output, which VS Code displays on the screen. Note the three dots which are a visual clue.

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

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.

Python code really is easy to run

Hopefully, you now agree. There are of course other ways to run Python code, and you'll learn about 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 notebooks within 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!

THERE ARE NO DUMB QUESTIONS

Q: The output from the draw function looks a little strange. What's the deal with those parens?

A: Technically, the draw function is returning a tuple, which is an immutable data structure built into Python. Don't worry what all this means for now, as you'll be learning lots about how Python works with data structures later in this book. And, yes, the output from the draw function could look more human-friendly, but at this stage in this book we're not interested in making your output look nice. Rather we're concentrating on showing you running Python code.

Q: “.ipynb” as a file extension? Kinda awkward, isn't it?

A: It stands for “Interactive PYthon NoteBook”, which is the format used by Jupyter to store your notebooks. Despite the weird filename extension, notebooks are cool in that they are text files based on a standard JSON format. You can treat an ipynb file like any other text file. In fact, if you know someone who can run a Jupyter notebook, you can share your ipynb file with them (perhaps including it as an email attachment?). Upon receipt, they can load your notebook into their Jupyter and work with their copy of your notebook as needed.

Q: What's the significance of those code cell numbers?

A: They are there mainly as a convenience, in that they are a visual clue as to what order your code cells executed in. They have nothing to do with Python. As such, and going forward, we'll only show the cell numbers when it makes sense to do so, as our goal is to get you to concentrate on the Python we're showing you, not the ins-and-outs of VS Code and Jupyter. For now, if you understand why you have to press **Shift+Enter**, then you're good to go.

Q: I just opened my Cards. ipynb file in VS Code and it's saved everything, including all the output! Shouldn't it just save my code?

A: No. Jupyter's format saves all the information from your notebook, including any generated output. This is why notebooks are saved as JSON. You can control what gets saved, so if you don't want the output saved, you don't have to save it. Having said, code saved without output is still saved as JSON, so things will look weird if you open an ipynb file in an editor which doesn't understand Jupyter, as you'll see the raw notebook JSON (which can look a tad intimidating).

Q: I have a Data Scientist friend and when I showed them my VS Code setup eyes were raised, and I was asked why I'm not running Jupyter inside my browser, like everyone else? What's the story?

A: Yes, our Data Scientist friends also love to run their notebooks in the browser-based Jupyter Notebook and Jupyter Lab environments (and we like both of those tools, too, BTW). However, we feel running notebook's within VS Code is a "better fit" to the way programmer's brains are wired. As you work through this book and gain more experience with VS Code, you'll see that the editor also has lots more to offer over what we've shown you so far.

Q: Can I use my VS Code produced notebooks with other Jupyter tools, for instance, inside a web browser?

A: Yes. If the tool you want to use understands the Jupyter notebook format, it can use anything produced by VS Code, as VS Code notebooks are 100% compatible with Jupyter's JSON standard.

Q: Can I use VS Code to manage my GIT stuff?

A: Yes, but getting into how you do that is beyond the scope of *Head First Python*, so you won't see us using GIT here. Of course, that's not to say we don't use GIT to manage the code in our projects: We do! Oh, BTW, If you're looking for an excellent GIT primer, check out *Head First GIT*.

Q: 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 a reference to Python’s preferred filename extension for code files, which is `.py`.

Q: [Coughs] Emmm... Is R a real programming language?

A: Oh, come on, now, let’s not go there. (This is a Python book, after all).

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 are going on a whistle-stop tour of some of Python’s standout language 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.

We're only joking. 😊

The goal for this chapter is to arm you with enough know-how to confidently answer the question: *Why Python?* To do that, you'll be *introduced* to Python language features on the pages which follow, albeit from a high-level.

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

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

Python ships with a rich standard library

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

When you hear programmers refer to Python as coming with “batteries included”, they are referring in part to the PSL. There's a lot to it:

<https://docs.python.org/3/library/index.html>.





In this book, “PSL” is shorthand for the “Python Standard Library”.

No. It’s quite an apt description.

The Python 3 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 3 *alone* is all you’ll need to get going, which means the standard install of Python 3 works “right out to 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.

The fact is some of the world’s largest websites run Python, some of the world’s biggest banks, too. And let’s not forget all those legions of data scientists solving all manner of problems every day with custom Python code. Python may indeed look *different*, but this does not mean it can’t get the job done. Python *is* fun to use, but this doesn’t mean it’s a toy. Far from it.

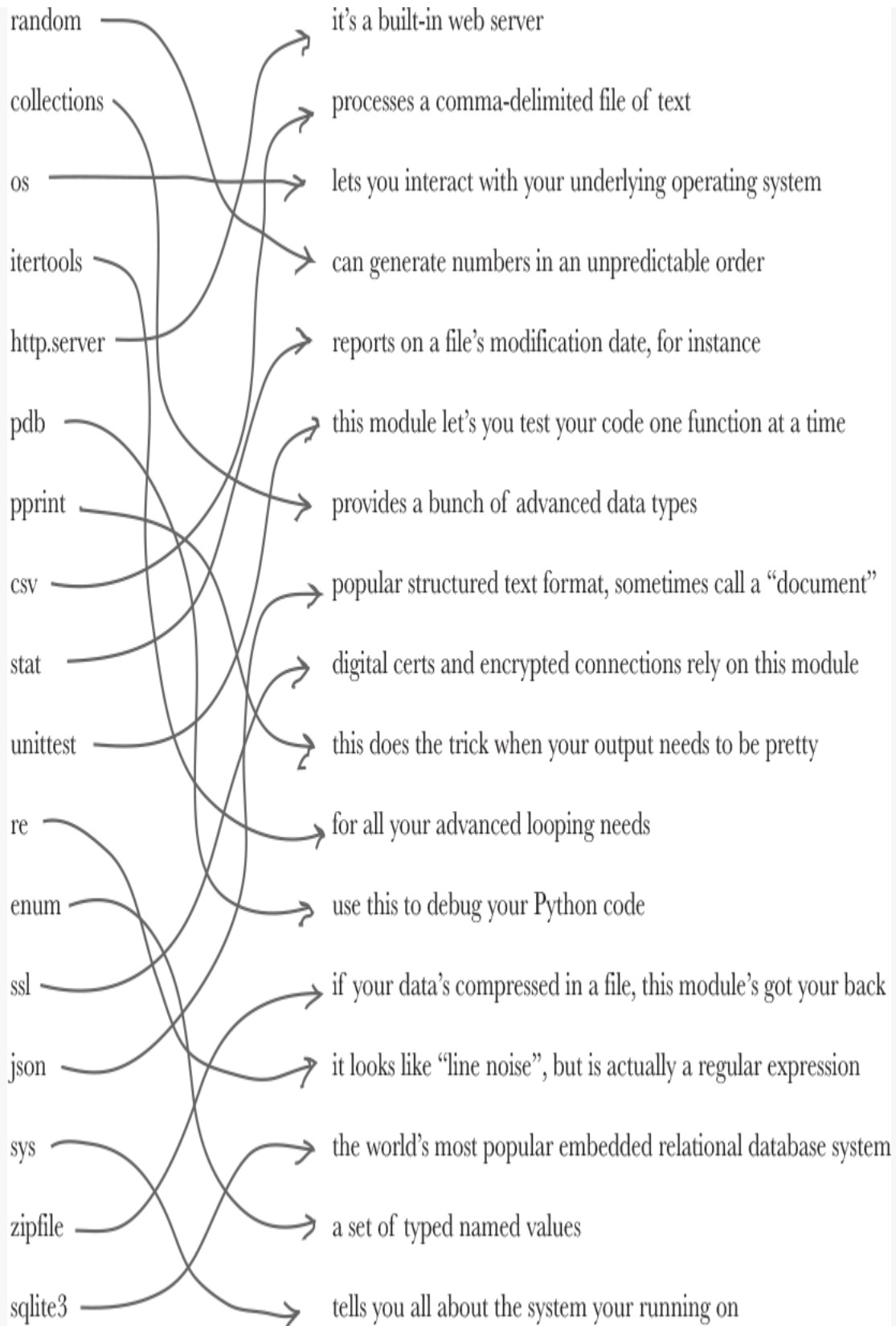
WHO DOES WHAT?

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

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





Wow! I've seen bowls
of spaghetti more
organized than that...



Yes: There's a lot going on there.

Recall the goal here 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.

- **① You've only scratched the surface**

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

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

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

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

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

TEST DRIVE



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:

```
import random
random.randint(1, 100)
```

32

```
random.randint(1, 100)
```

51

Import the module. →

← Generate random integers.

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

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

c = collections.Counter(how_many)

c["f"]
```

11

Import the module.

A string which contains lots of "f" characters.

Create a new Counter object from the characters in the "how_many" string.

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

Only write the code you need

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

The PSL comes packed with powerful built-in functions

As the phrase *built-in function* is a bit of a mouthful, everyone uses BIF instead. The BIFs are part of the PSL, and you can use them anywhere in

your code. The full list is here:

<https://docs.python.org/3/library/functions.html>, and you'll see a bunch of BIFs in action in this book. For now, we'll look at four: **len**, **print**, **type**, and **dir**.

“BIF” is shorthand for “builtin function”.



Ha, ha, very funny...

Let's get one thing straight: When it comes to making jokey references to popular culture, *that's our department*, okay?

Seriously, though, naming can sometimes get a little silly around Python folk. For a case in point, check out this page's *Geek Note*, below.

GEEK NOTE



What's in a name?

Lots, when it comes to Python, which is **not** named after a type of snake! Instead, Python is named in honor of *Monty Python's Flying Circus*, a classic British comedy TV series and movie franchise featuring *John Cleese, Michael Palin, Graham Chapman, Terry Gilliam, Terry Jones, and Eric Idle*. Who knew? For more on the origin of Python's name, see: <https://docs.python.org/3/faq/general.html#why-is-it-called-python>.

BIFs provide practical, generic functionality

Let's get to know some BIFs.

If you are working with something in Python which has an associated size, the **len** BIF returns its length. Take a string as a for instance. Here's one, which you can go ahead and type into your notebook in VS Code (remember: press **Shift+Enter** to run the code in your cell):

```
msg = "Hello from Head First Python"
```

A new variable named "msg".

A friendly string assigned to your new variable.

Two things happen here: Whatever's to the right of the assignment operator (=) is created in Python's memory as a new object, then a *reference* to the object is assigned to the variable name to the left of the assignment operator.

Go ahead and type the name of your latest variable into your next cell, then press **Shift+Enter** again:

We know it's almost too exciting, but when you type a variable name into a cell, then press Shift+Enter, Jupyter displays the value the variable refers to as output. In this case, the string is displayed.

```
msg
```

'Hello from Head First Python'



I surrounded that string with double-quotes, but Jupyter's displayed it with single quotes. Is this something I should worry about?

No. Not really.

When entering strings you can use *either* just so long as they match, and most Python programmers prefer to use double-quotes. That said, the Python interpreter favors displaying strings with single-quotes, and that's what Jupyter is showing you here: The interpreter's representation of the string.

EXERCISE



When we stated that *we* were going to look at four BIFs, we actually meant *you* were. The four BIFs we picked – **len**, **print**, **type**, and **dir** – see a lot of usage, especially when paired with Python’s various REPL mechanisms.

Here’s what we’d like you to do. For each of the lines of code shown below, type the line into its own code cell in your WhyPython notebook, remembering to press **Shift+Enter** to run each cell. Then, in the spaces provided and using your trusty pencil, write down what you think the BIF is doing. (Our answers are on the next page).

1

```
len(msg)
```

Write your
answer
here.



2

```
print(msg)
```

3

```
type(msg)
```

4

```
print(dir(msg))
```

EXERCISE SOLUTION



When we stated that *we* were going to look at four BIFs, we actually meant *you* were. The four BIFs we picked – **len**, **print**, **type**, and **dir** – see a lot of usage, especially when paired with REPLs.

Here's what we asked you to do: For each of the lines of code shown below, you were to type the line into its own code cell in your WhyPython notebook, remembering to press **Shift+Enter** to run each cell. Then, in the spaces provided and using your trusty pencil, you were to write down what you thought the BIF is doing.

Here's what we saw, together with annotations on what's going on. Does this match what you think each BIF does?

1

```
len(msg)
```

The "len" BIF reports the length of a variable. In this case, "len" is telling you that the "msg" variable contains 28 characters.

28

2

```
print(msg)
```

The "print" BIF displays values on screen and, unlike Jupyter, there are no quotes shown here, as the string is printed to your display.

Hello from Head First Python

3

```
type(msg)
```

The "type" BIF reports on the type of the value referred to by the "msg" variable. In this case, the "msg" variable refers to a string (known as a "str" in Python).

str

4

```
print(dir(msg))
```

This last line of code, which uses the "dir" BIF needs a bit more room, so you'll have to wait until the next page for an explanation.

```
['_add_', '_class_', '_contains_', '_doc_', '_eq_', '_format_', '_ge_', ...
'_getitem_', '_getnewargs_', '_gt_', '_init_subclass_', '_iter_', '_le_',
'_mul_', '_ne_', '_new_', '_reduce_'
:]
```

There's a lot more output in both directions.

The print dir combo mambo

Don't worry. The first time we saw the output from the **print dir** combination, we were a little perplexed, too. And... we certainly didn't erupt into spontaneous dancing.

The **print** BIF in this line of code is used to ensure the output from the **dir** BIF displays *across the screen* as opposed to down the screen (as it very quickly scrolls out of sight).

What's actually appearing is generated by the **dir** BIF, which displays the list of valid attributes associated with the Python object passed as an argument. In this case, the argument to **dir** is the `msg` variable (which the **type** BIF just confirmed is a string):



↑
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 leading and trailing double-underscore characters.

```
print(dir(msg))
```

```
['_add_', '_class_', '_contains_', '_delattr_', '_dir_',  
'_doc_', '_eq_', '_format_', '_ge_', '_getattr_',  
'_getitem_', '_getnewargs_', '_gt_', '_hash_', '_init_',  
'_init_subclass_', '_iter_', '_le_', '_len_', '_lt_', '_mod_',  
'_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_',  
'_rmod_', '_rmul_', '_setattr_', '_sizeof_', '_str_',  
'_subclasshook_', 'capitalize', 'casefold', 'center', 'count', 'encode',  
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',  
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',  
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', '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 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 any Python REPL, of which Jupyter is just one example. 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 `msg` variable to see what they do, a more sensible approach is to read the documentation associated with the method...



We all love searching through the docs, right?

Like most of you, our eyes are also glazing over at the thoughts of this. We agree with the sentiment here: Who has time to lookup, find, and read documentation? Thankfully, Python makes the *lookup* and *find* bits easy, thanks to another BIF called **help**.

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

```
help(msg.upper)
```

Help on built-in function upper:

upper() method of builtins.str instance

Return a copy of the string converted to uppercase.

The last line of output is the important bit here.

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

There's built-in functionality everywhere!

Not only do you have the PSL and the BIFs built-in, but – as demonstrated when you performed the **print dir** combo on your `msg` variable – there's a ton of functionality packed inside each and every Python object:

```
print(dir(msg))
```

```
['_add_', '_class_', '_contains_', '_delattr_', '_dir_',  
'_doc_', '_eq_', '_format_', '_ge_', '_getattr_',  
'_getitem_', '_getnewargs_', '_gt_', '_hash_', '_init_',  
'_init_subclass_', '_iter_', '_le_', '_len_', '_lt_', '_mod_',  
'_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_',  
'_rmod_', '_rmul_', '_setattr_', '_sizeof_', '_str_',  
'_subclasshook_', 'capitalize', 'casefold', 'center', 'count', 'encode',  
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',  
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',  
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', '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']
```

Look at all of the functionality packed into strings! Again, this is code you don't have to write yourself, you just call it as needed. (And, as before, don't worry about all those double-underscore entries for now).

In addition to strings, Python has built-in data type support for numbers which can be either integers (`int`) or floating point numbers (`float`). There's also the standard boolean data values: `True` and `False`.

Although numbers and booleans are *also* Python objects which you can use the **print dir** combo against, this tends to happen less in practice as

numbers and booleans are simple scalar values which are, as a result, well understood. This is not the case with a string which is a more complicated beast, as evidenced by the number of built-in methods available to you. This is also the case with Python's built-in data structures: **list**, **tuple**, **dictionary**, and **set**.

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, the briefest of overviews is presented here. You'll see each of the big 4 data structures in detail later in this book (when we're sure you'll learn to love them, too). For now, return to your WhyPython notebook, and follow along with the next four *Test Drives* in VS Code.

TEST DRIVE



You met a list earlier in the card deck code. Recall the `suit` variable:

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

A list of 4 strings, which each string representing the name of one of the four card suits.

```
type(suit)
```

list ← The "type" BIF confirms you have a list.

```
len(suit)
```

4 ← The "len" BIF tells you there are 4 items in the list.

```
print(dir(suit))
```

```
['_add_', '_class_', '_class_getitem_', '_contains_',  
'_delattr_', '_delitem_', '_dir_', '_doc_', '_eq_', '_format_',  
'_ge_', '_getattr_', '_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']
```

Continuing to ignore all those double-underscore entries for now, the available methods included with every list is shown. There are nowhere near as many methods as for strings, but these do look useful, don't they?

TEST DRIVE



Lists are really useful for lots of reasons, but mostly due to the fact they can mutate: As your code runs, lists can shrink and grow as needed. If what you are trying to model with your data does not require mutability, you may wish to consider using a tuple which – keeping things simple, for now – can be thought of as a list which cannot mutate.

On the surface, using a tuple instead of a list doesn't look all that different.

Whereas lists surround data items with square brackets, tuples surround their data items with parentheses.

```
suit = ("Clubs", "Spades", "Hearts", "Diamonds")
```

```
type(suit)
```

tuple ← The "type" BIF confirms the variable is now a tuple.

```
len(suit)
```

4 ← The type of the variable has changed, but it still contains 4 items.

```
print(dir(suit))
```

```
['_add_', '_class_', '_class_getitem_', '_contains_',  
'_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_',  
'_getattr_', '_getitem_', '_getnewargs_', '_gt_', '_hash_',  
'_init_', '_init_subclass_', '_iter_', '_le_', '_len_', '_lt_',  
'_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_',  
'_rmul_', '_setattr_', '_sizeof_', '_str_', '_subclasshook_',  
'count', 'index']
```

You may be looking at these two tuple methods names and thinking: "It that it!?" Based on this, it appears tuples don't do much, but that's not the impression you should take from this Test Drive. Tuples don't "do much" because they cannot mutate. It can often be useful to think of the "suit" variable (on this page) as a **CONSTANT** list. And, don't worry, you'll see lots more tuple use-cases as you work your way through Head First Python.

TEST DRIVE



Lists and tuples are all over Python, but everybody's favorite built-in data structure is the dictionary (`dict`), which is a mapping data structure associating *keys* with *values*. Here's a simple example which associates a handful of student names with alphabetic grades.

Note the use of curly braces here which, in Python, surround data (as opposed to blocks of code).

```
grades = {  
    "Joe": "A",  
    "Ronald": "C",  
    "Jimmy": "B",  
    "Donald": "D",  
    "Richard": "F",  
}
```

The student names on the left are associated with alphabetic grades on the right. Another way to think about this is to consider the grades are "mapped" to their associated names. Each mapping can be thought of as a row of data.

```
type(grades)
```

dict ← The "type" BIF reports "grades" is a dictionary data structure.

```
len(grades)
```

5 ← You'd be forgiven for expecting the "len" BIF to report 10 here, as there are ten strings in the "grades" dictionary, but "len" reports the number of rows instead (which makes more sense).

```
print(dir(grades))
```

```
['_class_', '_class_getitem_', '_contains_', '_delattr_',  
'_delitem_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_',  
'_getattr_', '_getitem_', '_gt_', '_hash_', '_init_',  
'_init_subclass_', '_ior_', '_iter_', '_le_', '_len_', '_lt_',  
'_ne_', '_new_', '_or_', '_reduce_', '_reduce_ex_', '_repr_',  
'_reversed_', '_ror_', '_setattr_', '_setitem_', '_sizeof_',  
'_str_', '_subclasshook_', 'clear', 'copy', 'fromkeys', 'get', 'items',  
'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

Like lists, Python's dictionaries come with a bunch of built-in methods. There are a couple of these methods you'll use all the time. Dictionaries are cool.

TEST DRIVE



The last of The Big 4 is the set, which is just like the sets you learned about in Math class. When a bunch of objects are assigned to a set, duplicates are removed, and it's this characteristic which most Python programmers exploit. Of course, sets can be do so much more.

A string with lots of duplicated characters.

```
three = "sets sets sets"
```

```
unique = set(three)
```

Remove the duplicate characters by turning the string into a set.

```
print(unique)
```

```
{'e', 's', 't'}
```

You're left with a unique set of characters. Note, again, the use of curly braces to surround data.

```
type(unique)
```

set ← Not that you needed convincing, but the "type" BIF confirms you're dealing with a set.

```
len(unique)
```

4 ← There are 4 characters in the set.

```
print(dir(unique))
```

```
['_and_', '_class_', '_class_getitem_', '_contains_',  
'_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_',  
'_getattr_', '_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']
```

Despite the fact most Python programmers typically reach for sets when they need to remove duplicates, there's also a bunch of super useful set manipulation methods built-in, too.

Python has powerful built-in operators

Like other programming languages, Python comes with a large collection of operators. There's the usual suspects, such as `==`, `>`, `!=`, `<`, and so on. But, Python has a few extras which can be incredibly useful, especially when combined with the built-in data types/structures.

One such operator is **in**, which performs *membership testing*. Let's see **in** at work against some of the variables from earlier in this chapter.

Rather than force you to flip back to recall values referred to earlier, here's the value of four of this chapter's variables, displayed on screen thanks to the "print" BIF.

```
print(msg)
print(suit)
print(grades)
print(unique)
```

A string.

```
Hello from Head First Python
('Clubs', 'Spades', 'Hearts', 'Diamonds')
{'Joe': 'A', 'Ronald': 'C', 'Jimmy': 'B', 'Donald': 'D', 'Richard': 'F'}
{'e', 's', ' ', 't'}
```

A tuple.

A dictionary.

A set.

The **in** operator knows all about the built-in data types/structures. The **print** BIF, once again, can be put to good use to help you see the **in** operator

doing its stuff. As you run these five calls to **print** in your notebook, note the absence of any loop code:

```
print("bad" in msg)
print("Spades" in suit)
print("Jimmy" in grades)
print("B" in grades)
print("f" in unique)
```

Even though the variables on the right of the "in" operator refer to variables which contain multiple things, you are able to search each variable for a specific value (shown to the left of "in") without having to write any loop code. Result!

False
True
True
False
False

← Only the "Spades" and "Jimmy" searches succeed. You might have thought "B" would be found in the "grades" dictionary, but "in" only looks for matches against the dictionary's keys (not the values).

THERE ARE NO DUMB QUESTIONS

Q: The `suit` variable started out as a list, then you assigned a tuple to it, which surely changed its type, right? Why didn't Python complain?

A: On the surface it looks like `suit`'s type changed, but it didn't. The type of the object `suit` refers to changed. Think of variables in Python as **object references**. As a variable in Python can refer to any object of any type, it follows that the type of the object a variable refers to can change as your program runs. This is what is meant by “dynamic typing”, in that the type your variable refers to is bound at run-time, not compile-time. Some programmer view such an arrangement as evil at work. Python programmers do not share this view. To keep things straight in your head, remember that Python variables are object references which refer to values which themselves have type. There's a double look-up here: First, Python looks up the variable's name then, second, Python accesses the object the name refers to. (This might sound absurd, but it works surprisingly well).

Q: Surely there's more to the big 4 built-in data structures than what's presented in those four Test Drives?

A: Of course there is! In fact, knowing when to effectively use the correct built-in data structure is what often separates the better Python programmers from the pack. You'll see lots of uses of every one of the big 4 in this book and, by the time you're done, you'll be wielding all four of them like a real pro.

Q: What if none of the big 4 built-in data structures fit my needs? Can I create my own?

A: Yes, you can. But, let's not get ahead of ourselves here. When the time comes for you to craft a custom data structure as a 100% perfect fit for your application needs, we'll walk you through the process. All we'll say now is it's a class act!



Pretty much, yes.

If you think the PSL is cool, just wait until you learn about *PyPI*. Flip the page for a quick intro.

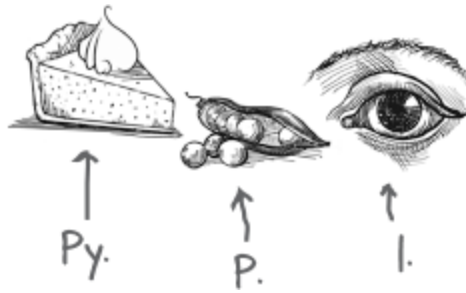
Python's package ecosystem is to die for

Being not content with what's already included in the PSL, the Python community supports a 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.





The image shows a browser window displaying the PyPI website. The browser's address bar shows the URL `https://pypi.org`. The page has a blue header with the PyPI logo on the left and navigation links for `Help`, `Sponsors`, `Login`, and `Register` on the right. The main content area features a large blue background with the text `Find, install and publish Python packages with the Python Package Index`. Below this is a search bar with the placeholder text `Search projects` and a magnifying glass icon. Underneath the search bar is a link that says `Or browse projects`. A light gray bar below the search area displays statistics: `369,706 projects`, `3,387,280 releases`, `5,907,835 files`, and `586,392 users`. The bottom section of the page has a white background and contains the `python Package Index` logo on the left. To the right of the logo, there is a paragraph: `The Python Package Index (PyPI) is a repository of software for the Python programming language.` Below this paragraph are two lines of text, each starting with `PyPI helps you find and install software developed and shared by the Python community.` The first line ends with `Learn about installing packages`. The second line ends with `Learn how to package your Python code for PyPI`.

PyPI - The Python Package Index

Help Sponsors Login Register

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

Search projects

Or [browse projects](#)

369,706 projects 3,387,280 releases 5,907,835 files 586,392 users

python™
Package Index

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

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

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



Emmm... maybe someone needs to lay off the incense, eh?

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. 😊

The Python REPL 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
```

The Zen of Python. Be sure to give it a read!

```
import antigravity
```

Make sure you're connected to the Internet before running this code.



It's still early days on your Python journey, but here's (for now) why we think why.

- 1 Python code is easy to read.**
- 2 Python code is easy to run.**
- 3 Python encourages experimentation and learning via its REPL.**
- 4 Python includes the Python Standard Library (PSL).**
- 5 The PSL provides practical, powerful, and generic built-in functions (BIFs).**

NOTE

Who can forget the “combo mambo”?

- 6 The PSL has “The Big 4” data structures: lists, tuples, dictionaries, and sets.**
- 7 Python has powerful built-in operators (like “in”).**

NOTE

Just think of all the code included in 4 through 8 that you don't have to write, maintain, nor test!

- 8 Python has the Python Package Index (PyPI).**
- 9 Python is fun!**

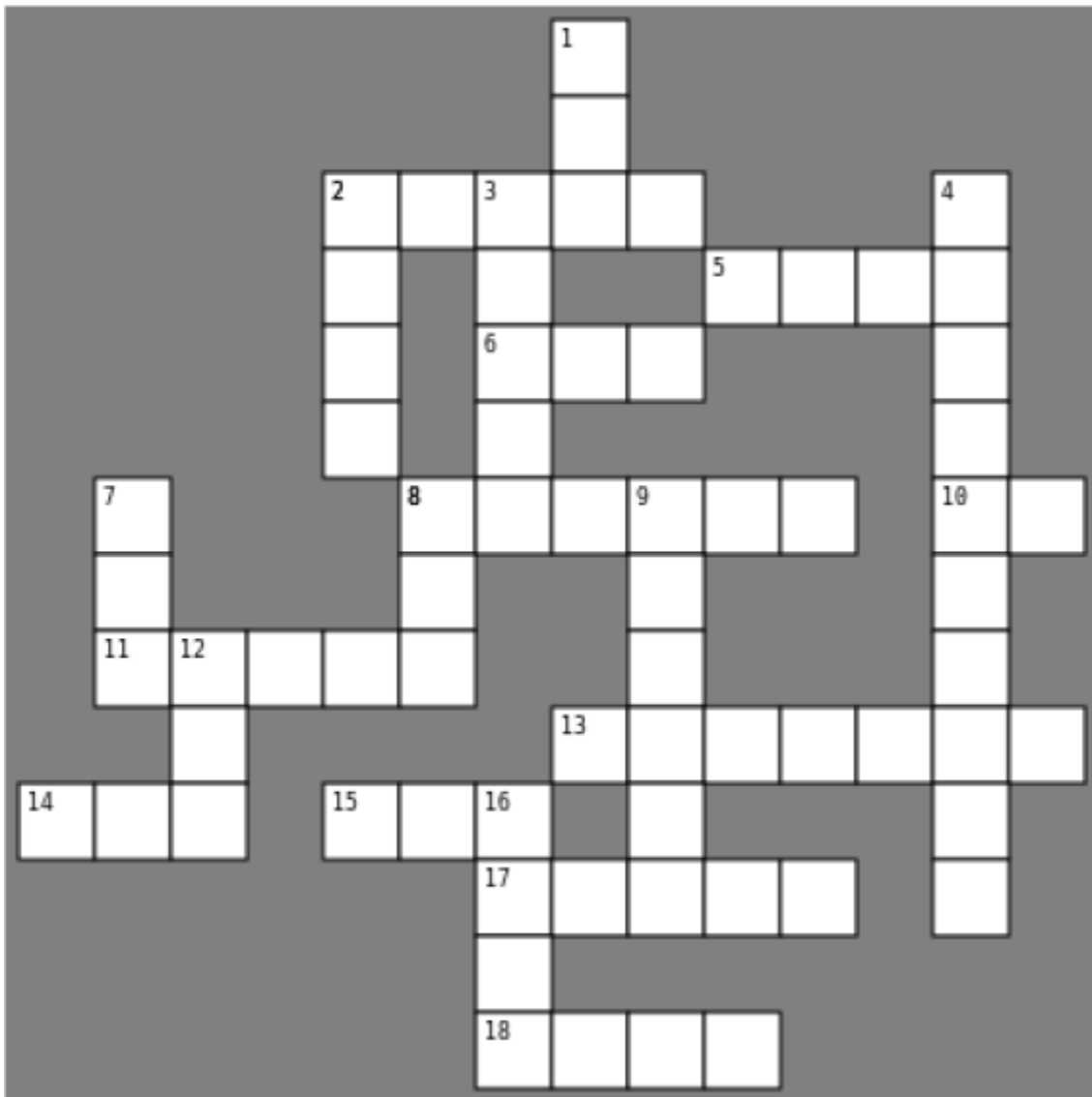
NOTE

Don't underestimate the importance of this last one.

The Opening 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. Like a constant list, immutable.
- 5. A worldwide repository of Python modules.
- 6. Basic type for whole numbers.
- 8. It's as long as this, a piece of?
- 10. A powerful little operator, good at finding things.
- 11. A number with a decimal point.
- 13. Can be either True or False.

14. Enlightenment, Python-style.
15. 2nd part of the combo mambo.
17. Use together with Shift to run.
18. Like an array, only more.

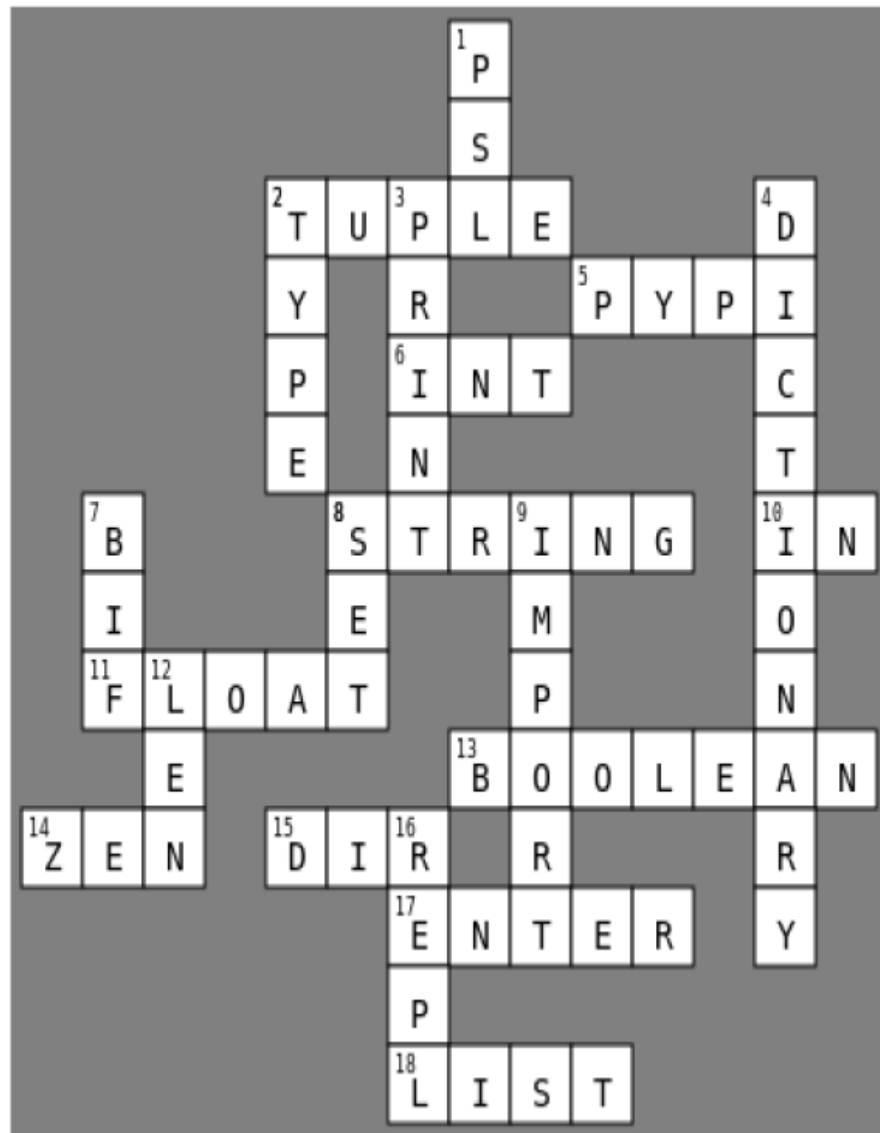
Down

1. Shorthand for the Python Standard Library.
2. This built-in function tells you what any object is.
3. Displays objects on screen.
4. The big 4's mapping technology.
7. Built-in function abbreviation.
8. There's no duplicates here.
9. Brings a library into your code.
12. Size does matter to this function.
16. Read-Eval-Print Loop.

The Opening Crossword Solution



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



Across

2. Like a constant list, immutable.
5. A worldwide repository of Python modules.
6. Basic type for whole numbers.
8. It's as long as this, a piece of?
10. A powerful little operator, good at finding things.
11. A number with a decimal point.
13. Can be either True or False.

14. Enlightenment, Python-style.
15. 2nd part of the combo mambo.
17. Use together with Shift to run.
18. Like an array, only more.

Down

1. Shorthand for the Python Standard Library.
2. This built-in function tells you what any object is.
3. Displays objects on screen.
4. The big 4's mapping technology.
7. Built-in function abbreviation.
8. There's no duplicates here.
9. Brings a library into your code.
12. Size does matter to this function.
16. Read-Eval-Print Loop.

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, and they all work in VS Code. For now, **Shift+Enter** is the most important combination:

Notebook key combinations:

Shift+Enter	Execute the current code cell, then move the focus to the next cell (creating a new empty cell when at the bottom of the notebook).
--------------------	---

Ctrl+Enter	Execute the current code cell, but don't move the focus.
-------------------	--

Alt+Enter	Execute the current code cell, then insert a new empty cell below the executed one. Move the focus to the new cell.
------------------	---

Notebook key sequences:

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

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

Other useful notebook key sequences:

Esc then C

Take a copy of the cell which currently has the focus.

Esc then V

Paste a previously copied (or cut) cell below the currently focused cell.

Esc then X

Cut the currently focused cell from the notebook.

Z

Undo the last cut (there's no need to press the ESC key here).

NOTE

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

Chapter 1. Diving in: *Hit the Ground Running!*

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

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



The best way to learn a new language is to write some code. Which is exactly what you'll do in this chapter. 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 to **automate** an existing manual process with Python. Along the way, you'll be introduced to Python **strings**, learning how work, how they are stored in Python's memory, as well as how to manipulate them to your heart's desire. You'll also (briefly) meet Python's **list** technology, learn how **variables** work, as well as discover how to read Python's **error messages** without going crazy... all while solving a real problem with real Python code. Let's get started!

Once upon a time, there was a swim coach...

A friend of a friend has been in touch. They heard you know how to code, and they know an underage swim coach they think you can help.

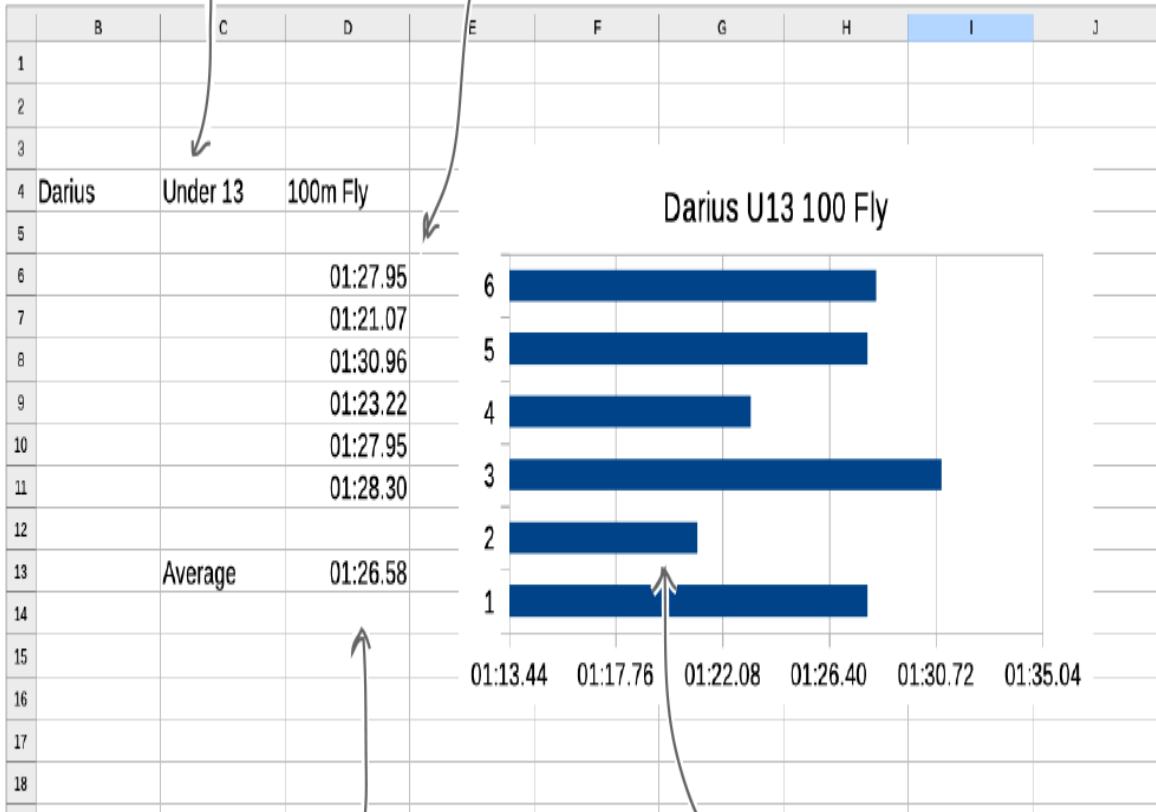
When it comes to monitoring the progress of their swimmers, the Coach in question does everything *manually*. During a training session, the Coach records each swimmers training times on their trusty clipboard then, later at home, *manually* types each swimmer's data into their favorite spreadsheet program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

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



The swimmer's name, age group, distance, and stroke is shown.

The list of times from the most-recent swimming session.



The average time is calculated and shown.

A simple bar chart lets the Coach quickly pinpoint the "best swim" from the session.

On the face of things, this doesn't look at all bad. Until, that is, you consider the Coach has over 60 such sheets to create after each training session, as swimmers can swim different distances in multiple strokes. That's a lot of work, and, as you can imagine, this process is *painfully* slow...



Automation is what you do!

You've been looking for a project you can cut your Python teeth on, and this work might just fit the bill.

Let's think about possible approaches for a moment.



That sounds simple enough, but...

That won't work. Yes, the Coach could bring a laptop to the pool, but they're way too busy running the swim session.

Stopping every minute or so to work at a laptop screen is a non-runner, and that's before considering that water, splashes, and laptops don't mix too

well.



That idea breaks too many rules.

The pool has a strict policy: *No photographic nor recording devices on-deck*. Privacy laws forbid any poolside device which can take a photo, so nobody at any swim session carries a smartphone.

So, using a smartphone is also a no-go.

Fortune decides to smile on you!

I think I'm in love...
with my new
smart stopwatch!

o
o



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.

```
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 best to ignore it.

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

An important point here is that this line of data is a line of text, not a collection of (what looks like) numbers. So, bear that in mind.

As luck would have it, 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, it'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.

As an example of one of these files, here's the contents of the file which contains matching data to the spreadsheet page shown for Darius from a few pages back:

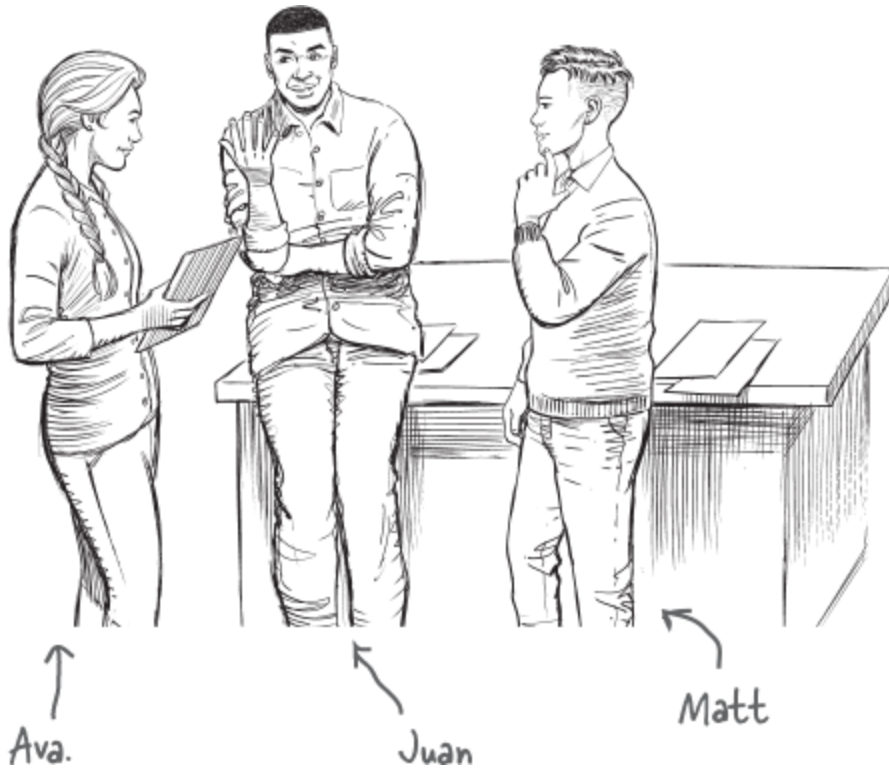
This data looks promising. If you can work out how to process this file, you can then do the same thing for any number of files which are formatted similarly.

As always, the big question is: *Where do you start?*



* You heard it *here* first.

Cubicle Conversation



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

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

Matt: How so?

Juan: 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.

Ava: What do you mean by "converted"?

Matt: That was my question, too.

Juan: A value like "1:27.95" represents, I'd imagine, 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.

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

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

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

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

Juan: Isn't that a given?

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

Matt: This is getting a little complex...

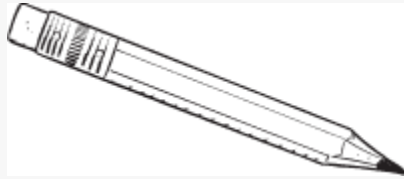
Juan: Not if we take things bit-by-bit.

Ava: We just need a plan of action.

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

Juan: I can recommend a great book... 😊

SHARPEN YOUR PENCIL



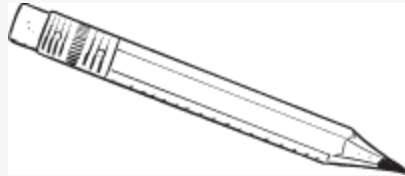
From the conversation on the last page, it looks like there are two main tasks identified at this stage: (1) extract data from the filename, and (2) process the swim times data in the file.

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 over the 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

Did you forget something? What about producing my bar chart? That's important to me!



A worried swim coach. →

Don't worry, we didn't forget.

Let's put that requirement on the long-finger for now, so that we can concentrate on processing the file's name and its data automatically with Python.

Once done, we'll return to the problem of automatically creating the bar chart.

Pinky promise.

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

For now, the plan is to concentrate on a single 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.

Create a new variable called "fn", then set it to refer to a string which matches the file's name.

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

A string (or is something else?).

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

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, what's shown is sequence of characters enclosed within quotes which, in most other programming languages, is the very definition of a string. *Not so in Python.*

The value to the right of assignment operator is a **string object**, which – you might well think – isn't that far off what a string is. However, the difference in Python is both subtle *and* important.



Objects contain more than their value.

In Python, each object contains whatever value it refers to (which for strings is a sequence of characters enclosed in quotes) as well as a collection of *methods* that can be executed against the value. This is an important distinction, so let's consider how this works.

Let's see what happens when Python runs this line of code:



Behind the Scenes, 1 of 2

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

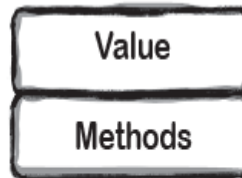
This looks simple enough. A string on the right of the assignment operator is assigned to a variable name on the left. How hard can this be?

- **1** First things first: the above line of code is a simple assignment statement. It's what the Python interpreter does in the background that can often raise some eyebrows. What you're about to be shown on this and the next page may well feel like you're heading

down a rabbit-hole, but bear with us for now. This is illuminating. Here goes...

Python starts with whatever's to the right of the assignment operator, spots what looks like a string, so creates a new, empty object in memory:

A new object is created in memory. It has two parts: a value-part and a methods-part.



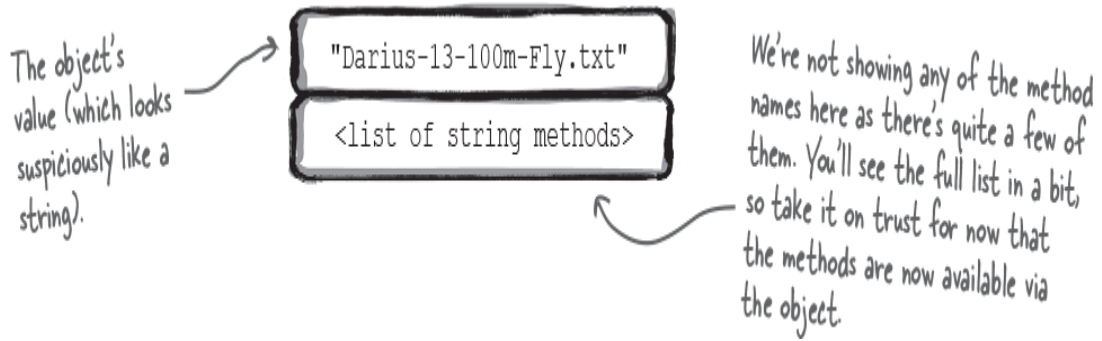
So far, so good.

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

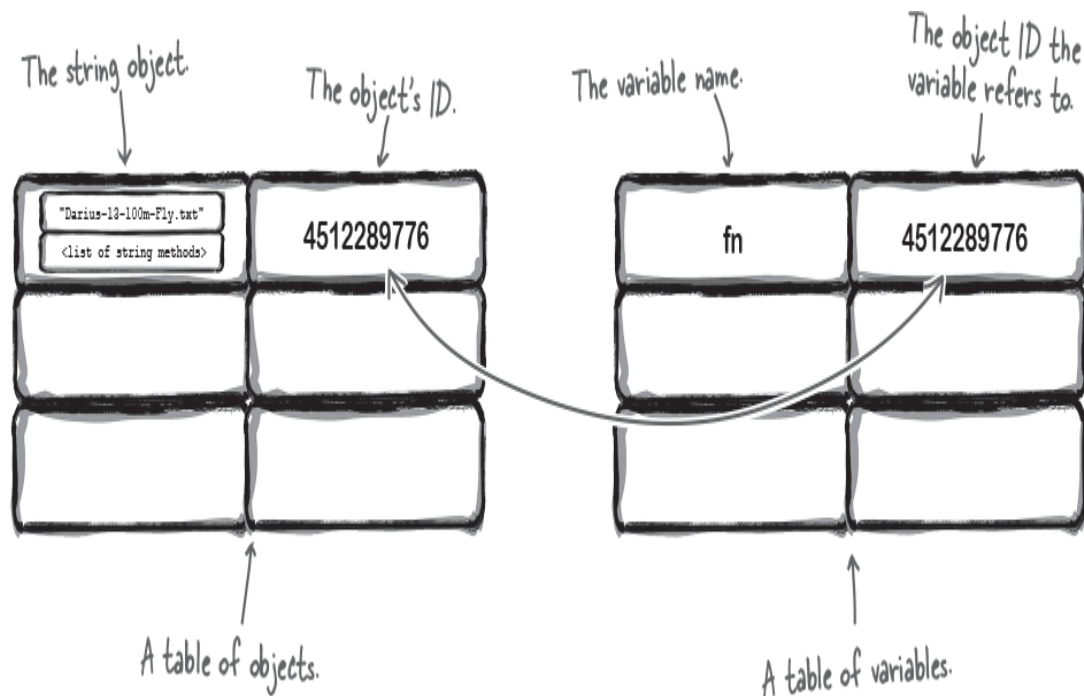
Behind the Scenes, 2 of 2



- ② The sequence of characters enclosed within quotes is assigned to the value-part of the new object, then the list of string methods built into Python are made accessible through the methods-part:



- 3** The string object now exists in memory and, consequently, has a memory address associated with it, known as its ID. This ID is then associated with the variable name (which is found to the left of the assignment operator, `fn` in the above line of code). It's useful to imagine Python maintains two tables in support of this arrangement:





So let me see if I've got this right. When I use the "fn" variable in my code, Python first looks up the ID it refers to then uses the ID to look up the referred to object?

Yes. Sounds a bit bananas, doesn't it?

When most programmers first have this explained to them, they immediately suggest such an arrangement is *very inefficient*. Others simply break into a sweat.

However, although there is a *double-lookup* occurring here, the fact variables refer to object IDs (not actual values) is one of the things that gives Python a lot of its power. We'll be sure to point out where this makes a big difference as you meet exemplars in this book.

Oh, one more thing: Pythonistas rarely refer to object IDs. Instead, they call them **object references**.

GEEK NOTE



It's important to note that Python **never** requires you to perform the double-lookup, as it's all handled automatically for you whenever you use a variable.

And, although Python provides an **id** BIF which given a variable name returns the variable's object reference (i.e., its memory address), you should **never** use nor rely on the value returned. Remembering to not use the **id** BIF is really, really important. Let Python handle memory for you, as you've more than enough to worry about trying to write the code you need to build your application.

Let's get back to that string...

Now that you know what goes on behind the scenes when a string's object reference is assigned to a variable name, let's return to the list of methods associated with any string.

The good news is you already know how to list any object's methods: 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.

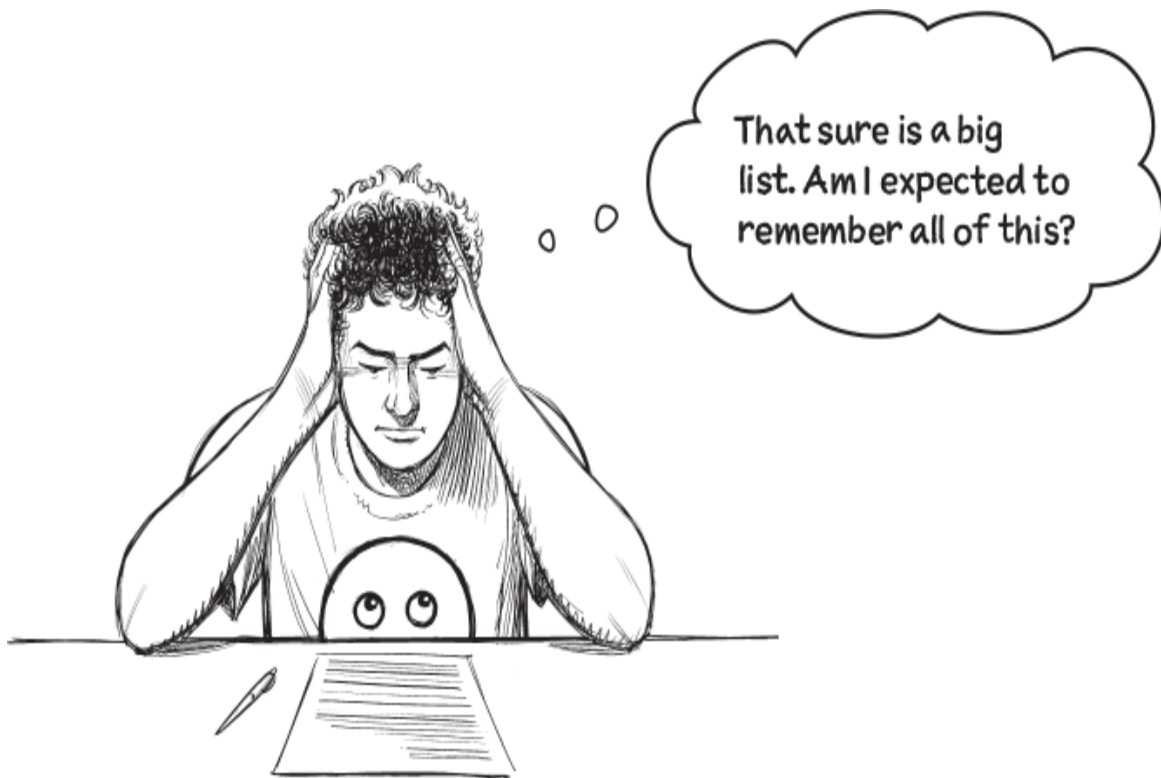


```
print(dir(fn))
```

```
['_add_', '_class_', '_contains_', '_delattr_', '_dir_', '_doc_',  
'_eq_', '_format_', '_ge_', '_getattr_', '_getitem_',  
'_getnewargs_', '_gt_', '_hash_', '_init_', '_init_subclass_', '_iter_',  
'_le_', '_len_', '_lt_', '_mod_', '_mul_', '_ne_', '_new_',  
'_reduce_', '_reduce_ex_', '_repr_', '_rmod_', '_rmul_', '_setattr_',  
'_sizeof_', '_str_', '_subclasshook_', 'capitalize', 'casefold', 'center',  
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',  
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',  
'isnumeric', 'isprintable', 'isspace', 'istitle', '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 methods 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?



RELAX



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

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

THERE ARE NO DUMB QUESTIONS

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

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

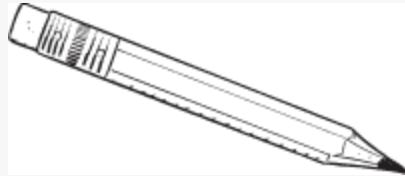
Q: Is there a way I can learn more about what a particular method does?

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

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

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

SHARPEN YOUR PENCIL



Let's try out 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 – using a pencil – 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.

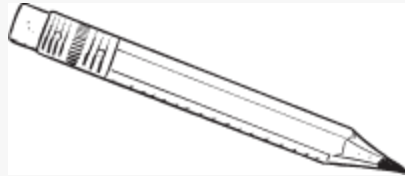
This is Python's **dot** operator, which allows you to invoke a method on an object. This means `fn.upper()` calls the **upper** method on the string referenced by the `fn` variable.

This is a little different to 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 try out 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 – using a pencil – 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()
```

The "upper" method returns a copy of the string in all UPPERCASE.

'DARIUS-13-100M-FLY.TXT'

```
fn.lower()
```

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

'darius-13-100m-fly.txt'

```
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'

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



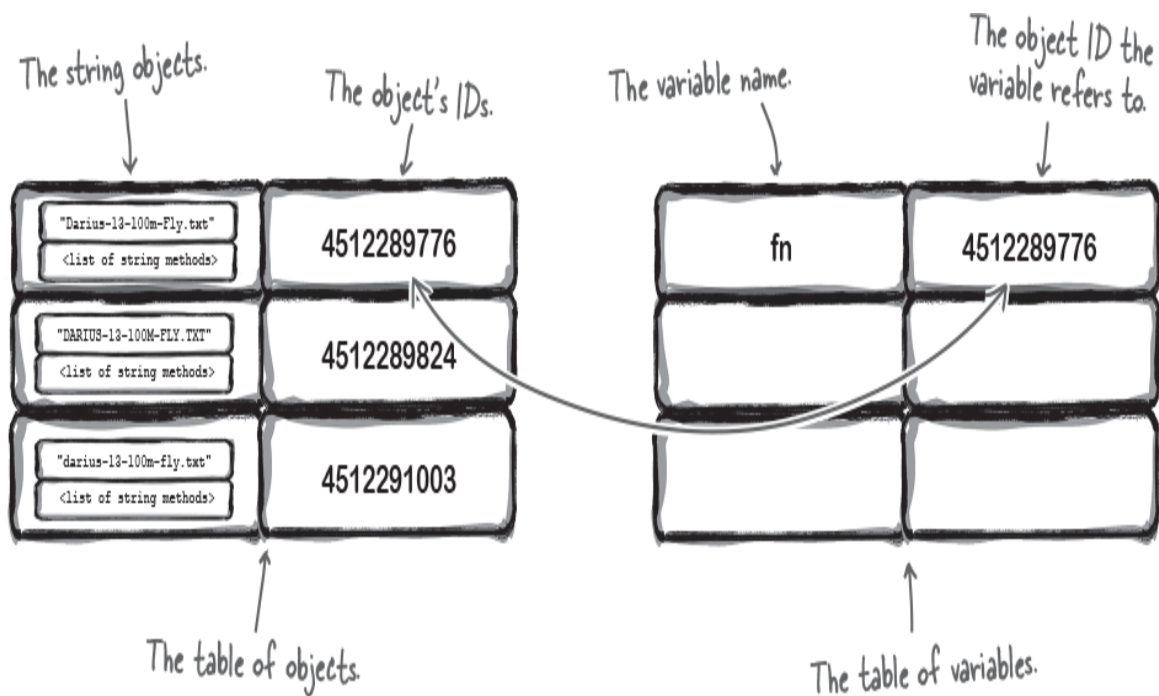


Yes, that's right.

The values returned by the **upper** and **lower** methods are both *new* string objects, which have a value-part and a methods-part, as well as unique object references (or IDs, if you prefer).

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

Returning to the diagram we used earlier to introduce object references, here's what it looks like as a result of the code from the *Sharpen* executing:



The three string objects are in the table of objects, with each assigned their own object reference. Sadly, this state of affairs doesn't last very long. As the two most-recent string objects aren't assigned to a variable name, there are no actual references to them. The next time Python's memory management technology runs, the *unreferenced* objects are garbage collected, and effectively disappear. Cue the sad music...

You're still on Task #1

Recall the three sub-tasks identified earlier for Task #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)

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'
```

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.

You have a choice: You can run **split** and see what happens, or you can read **split**'s documentation.

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

Read the method's documentation!

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

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

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

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 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 docs 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:

Pass "-" as the only argument to the "split" method.

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

This looks better. Instead of one string, you now have four smaller strings. The original string is broken apart on the "-" character.

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



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*.

No matter the call to "split", you start out with a string object assigned to the "fn" variable name.

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

The variable name.

The string object.

The "split" method is called with no arguments, so the defaults kick in (i.e., split on whitespace).

`fn.split()` 2

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

As there is no whitespace in the string, "split" creates a list with one "word" in it, which turns out to be an exact match for the original string object, only now it's in a list!

4 `fn.split("-")`

The dash character is passed as an argument to "split", which uses the dash to break the string object apart.

5 `['Darius', '13', '100m', 'Fly.txt']`

As the dash character appears in the string, "split" breaks the string apart, throwing away the dashes and leaving you with four individual "words" (i.e., strings), which are in a list.

Is it time for another tickmark?

It's tempting to look at your list of sub-tasks, grab your pen, then put a satisfying tick mark 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 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. 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 character "1" and the "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', '100m-Fly.txt'] ←

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

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

3

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

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

the dot character..

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

→ ['Darius-13-100m-Fly', 'txt']

4

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

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

```
['Darius-13-100m-Fly.txt'] ←
```

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

5

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

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

the original string in a list (as per the default behavior).

Nope. It doesn't do this!

Yikes!
↓

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

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

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

How to understand Python's error messages

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.



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**.

But... 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 “.”:

```
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?



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**.

But... now you know this, how do you fix it?

Fixing broken chains

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

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

Lists are important in Python, but I'm not so sure this is a list issue, despite that traceback.

I agree. We're trying to manipulate the string, not the list. Do strings come with anything else which might help?



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 from earlier. 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** 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 show 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'

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

'Free'

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

'Back'

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

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

'Breas'

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

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.

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.removesuffix)
```

Help on built-in function removesuffix:

```
removesuffix(suffix, /) method of builtins.str instance
```

```
Return a str with the given suffix string removed if present.
```

*This sounds
more like it.*

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

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

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

Did you notice that `rstrip` and `removesuffix` both returned strings (not lists)? This has an important implication when using either of these methods in a chain.

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

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

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

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

Now that you've identified the method you need, you can create the method chain needed to extract 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 four data you need.

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

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

a. Read the filename ✓

It's time for a tickmark. →

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 be changed. 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` (after all, 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.

You're nearly done with Task #1

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 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 lists 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.

TEST DRIVE



NOTE

Don't forget to follow along!

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

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

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

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

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

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

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

```
parts
```

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

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

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

Four new variables are created.

As expected, the square bracket notation is used to pick out the individual data values from the list. (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:

```
swimmer
```

'Darius'

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

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

It looks like Task #1 is complete!

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) ✓

Considering the code from your last *Test Drive*, you’re now 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, although you’re coming up on forty pages for this chapter, the amount of code you need to copy isn’t overwhelming, is it?

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

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

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

Put the filename in a variable. →

Break the filename apart. →

Extract the relevant data items. →

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?

↑
This is Nina, BTW.

The `parts` variable feels kinda integral.

That said, we get where Nina's coming from, 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.

But, can you do without `parts`?

NOTE

So... 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.

As the **split** method produces a list, 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 doing 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.



First off: rude. Secondly: not so fast.

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 the `swimmer`, `age`, `distance`, and `stroke` have occurred.

Perhaps there is another way?

Multiple assignment (aka unpacking)

Although it's a language idea which 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:

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

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

```
pie = 3.14
```

```
pie
```

3.14

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

```
pie, meaning = 3.14, 42
```

```
pie
```

3.14

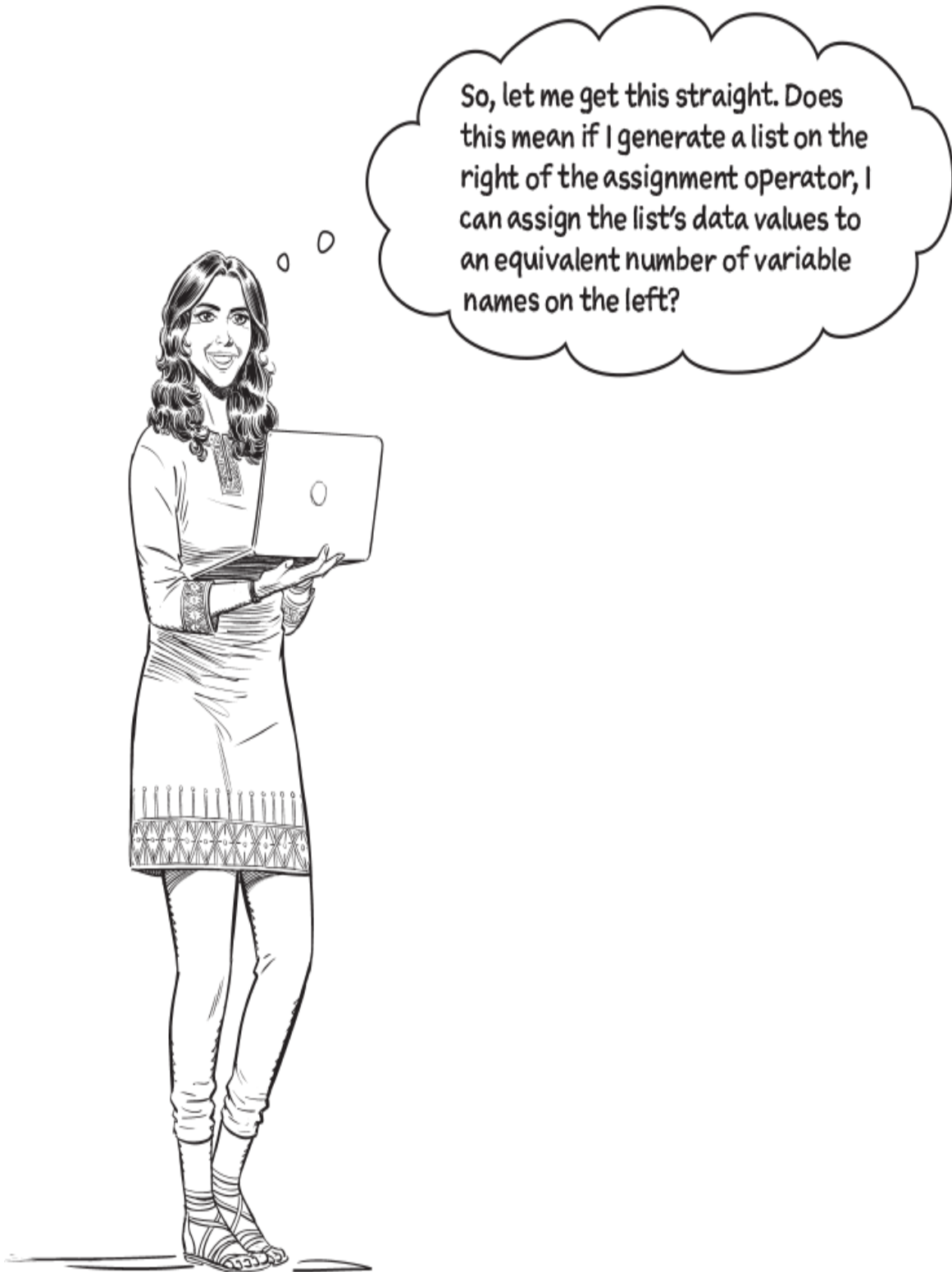
```
meaning
```

42

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

Not 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). And, Python treats the data values on the right as if they are a list, but does not require you to enclose the literal data values within square brackets (as shown above).



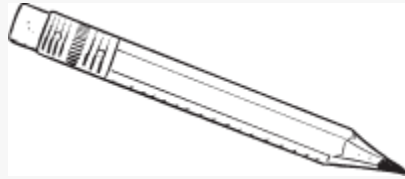
So, let me get this straight. Does this mean if I generate a list on the right of the assignment operator, I can assign the list's data values to an equivalent number of variable names on the left?

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 *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.

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

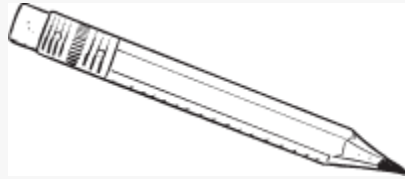
```
fn = "Abi-10-50m-Back.txt"

_____

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 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)

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("-")
```

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





Great. Thanks!

Of course, there's still a bit of work to do. Let's remind everyone what Task #2 is (over the page).

Task #2: Process the data in the file

At first glance, it looks like there's a bit of work here:

1. [Read the lines from the file](#)

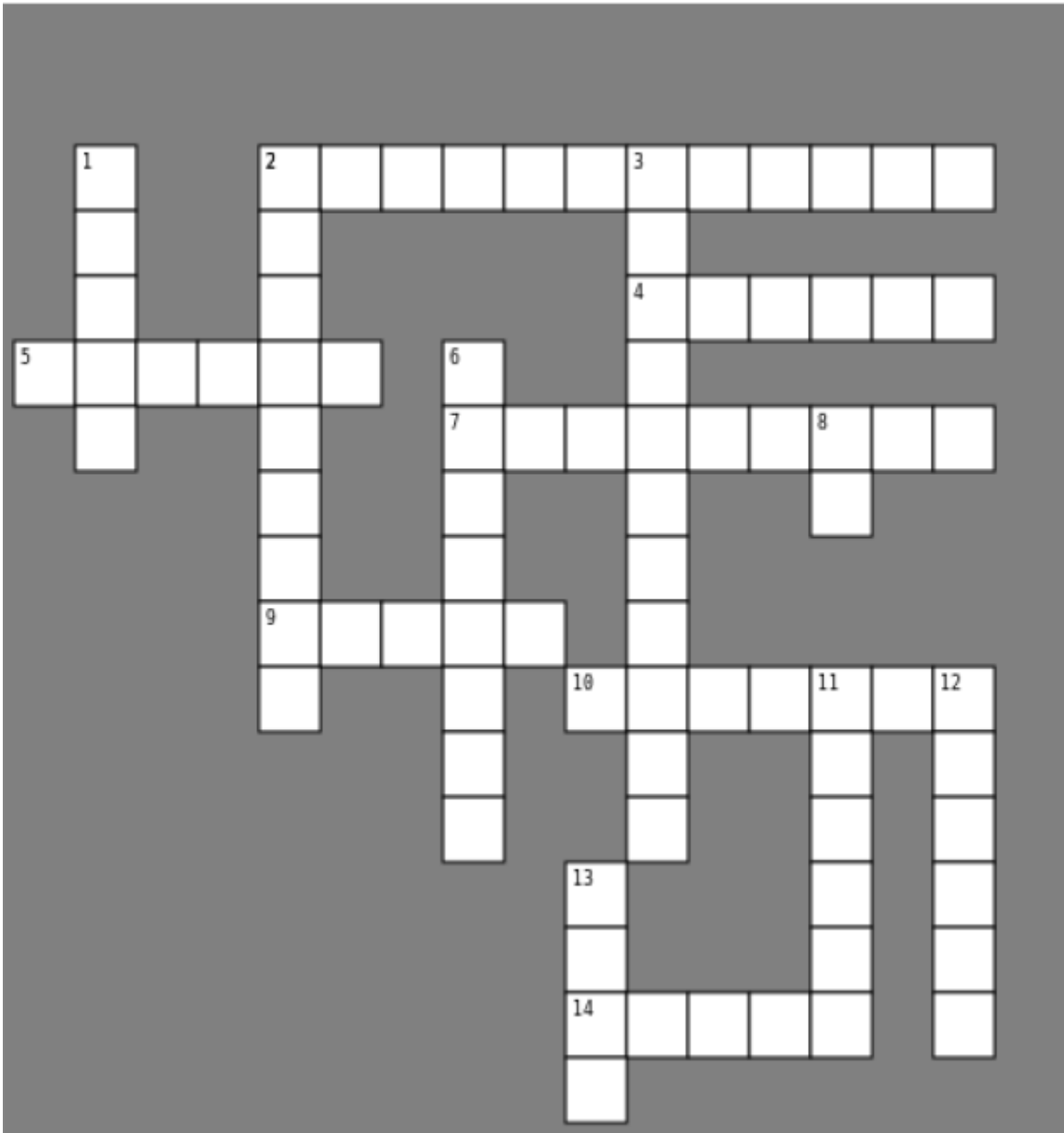
2. Ignore the second line
3. Break the first line apart by “,” to produce a list of times
4. Take each of the times and convert them to a number from the “mins:secs.hundredths” format
5. Calculate the average time, then convert it back to the “mins:secs.hundredths” format (for display purposes)
6. Display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

So much, in fact, that we’ve made an *Executive Decision* and decided to hold off on starting this work until your next chapter. Before getting to that, though, you’ve just enough time to do two things: make a cup of your favorite brew, and, sip your beverage while working through this chapter’s crossword.

The Unpacking 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

- 2. Can be used to get rid of a filename's extension.
- 4. Removes a set of characters from the end of a string.
- 5. Short for "double underscore".
- 7. Another name for multiple assignment.
- 9. Ball and _____.

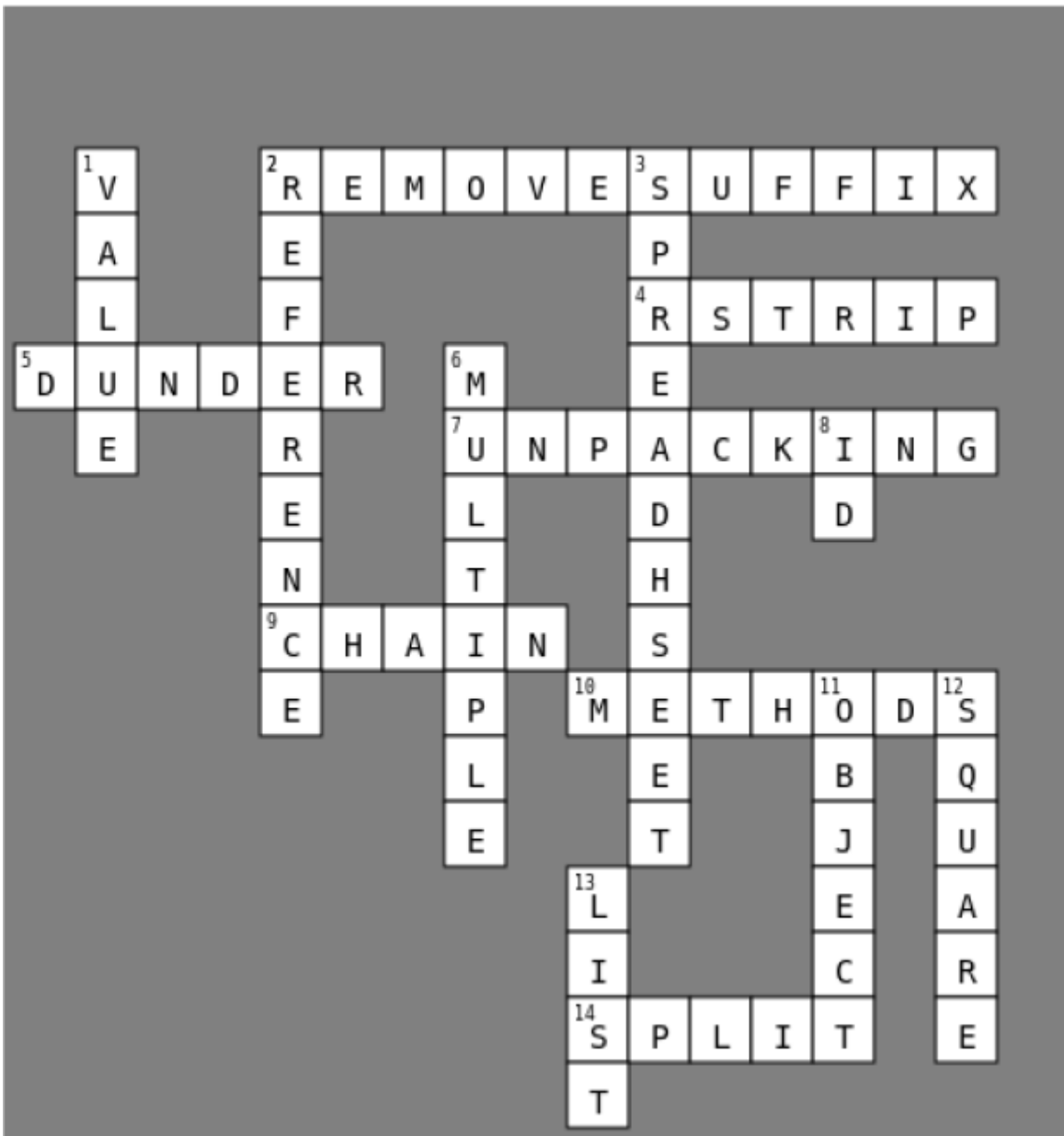
10. The plural name given to an object's built-in functions.
14. Comes in handy when breaking apart strings.

Down

1. Every object in Python has 10 across, in addition to this.
2. It's not an object identifier, it's an object _____.
3. The swim coach mucks about with one of these.
6. More than one.
8. Never rely on the value returned by this BIF.
11. Everything is one of these.
12. Our favorite brackets.
13. This is sort of like an array in other programming languages.

The Unpacking Crossword Solution





Across

- 2. Can be used to get rid of a filename's extension.
- 4. Removes a set of characters from the end of a string.
- 5. Short for "double underscore".
- 7. Another name for multiple assignment.
- 9. Ball and _____.

10. The plural name given to an object's built-in functions.
14. Comes in handy when breaking apart strings.

Down

1. Every object in Python has 10 across, in addition to this.
2. It's not an object identifier, it's an object _____.
3. The swim coach mucks about with one of these.
6. More than one.
8. Never rely on the value returned by this BIF.
11. Everything is one of these.
12. Our favorite brackets.
13. This is sort of like an array in other programming languages.

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

A NOTE FOR EARLY RELEASE READERS

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

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

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



The more code you write, the better you get. It's that simple. In this chapter, you continue to write *real* code to solve a *real* problem. In order to keep coding your solution, you need to learn how to **read** data from a **file**. More times than enough, the data read from files ends up in a **list**, one of Python's most-used **built-in data structures**. As well as learning how to create lists from file data, you'll also learn how to create lists from scratch, **growing the list** as needs be while your code runs. You'll also process your lists using one of Python's most popular (and loved) looping constructs, the

for loop, **converting** values from one data format to another. So, roll up your sleeves and let's get stuck in!

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 62 individual data files packaged as a ZIP archive. Grab a copy from here::

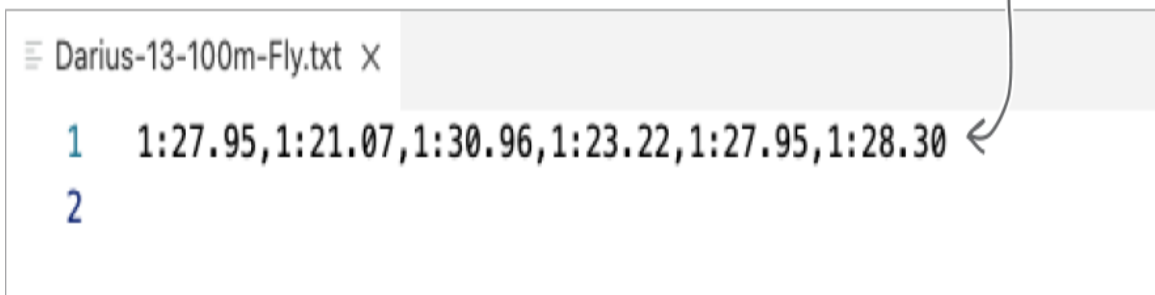
<https://python.itcarlow.ie/ed3/Learning/swimdata.zip>

NOTE

Don't worry, those 62 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 underage 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:



```
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 it's own, but it shines when combined with Python's... em, eh... **with** statement.

The open BIF works with files

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

You can call **open** directly in your code, opening a named file, processing its data, then closing the file when you're done. This open-process-close

pattern is very common, regardless of the programming language you use. In fact, Python has a language statement which makes working with the open-process-close pattern especially convenient: the **with** statement.

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

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

If it saves me from having to remember to do so, then YES. I'm forever forgetting...



TEST DRIVE



NOTE

As always, follow along.

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

For the code which 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:

UPPERCASE variable names are used to define the filename ("FN"), as well as it's location ("FOLDER"), as constants.

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

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

Although referred to as “constants”, Python doesn’t actually support the notion of constant values, so it is 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 when we named our filename variable “fn”. This is, of course, a shocking use of a lowercase variable name for a constant value! Just to be clear, we won’t tell if you won’t tell... and we promise to conform to this convention from here on in.

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 close the file:

The diagram shows a central code block with five numbered annotations explaining its parts:

- 1. The “with” keyword starts things off. (Arrow points to `with`)
- 2. The “open” B/F is called, and its given the location and name of the file to open. (Arrow points to `open(FOLDER+FN)`)
- 3. A file object is created, which we’ve called “df”, which is shorthand for “Darius’s file” (or maybe “data file”?). You can use any variable name here. (Arrow points to `as df:`)
- 4. The “df” file object provides a bunch of methods, with “readlines” grabbing all the lines from the file before converting them to a list, which is assigned to another new variable called “data”. (Arrow points to `df.readlines()`)
- 5. We were a little worried all this code wasn’t going to fit on the page... (Arrow points to the entire code block)

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

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.



Wow! We're two-and-a-bit chapters into this Python book and have yet to discuss indentation. This surely has to be some sort of record?

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

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

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

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

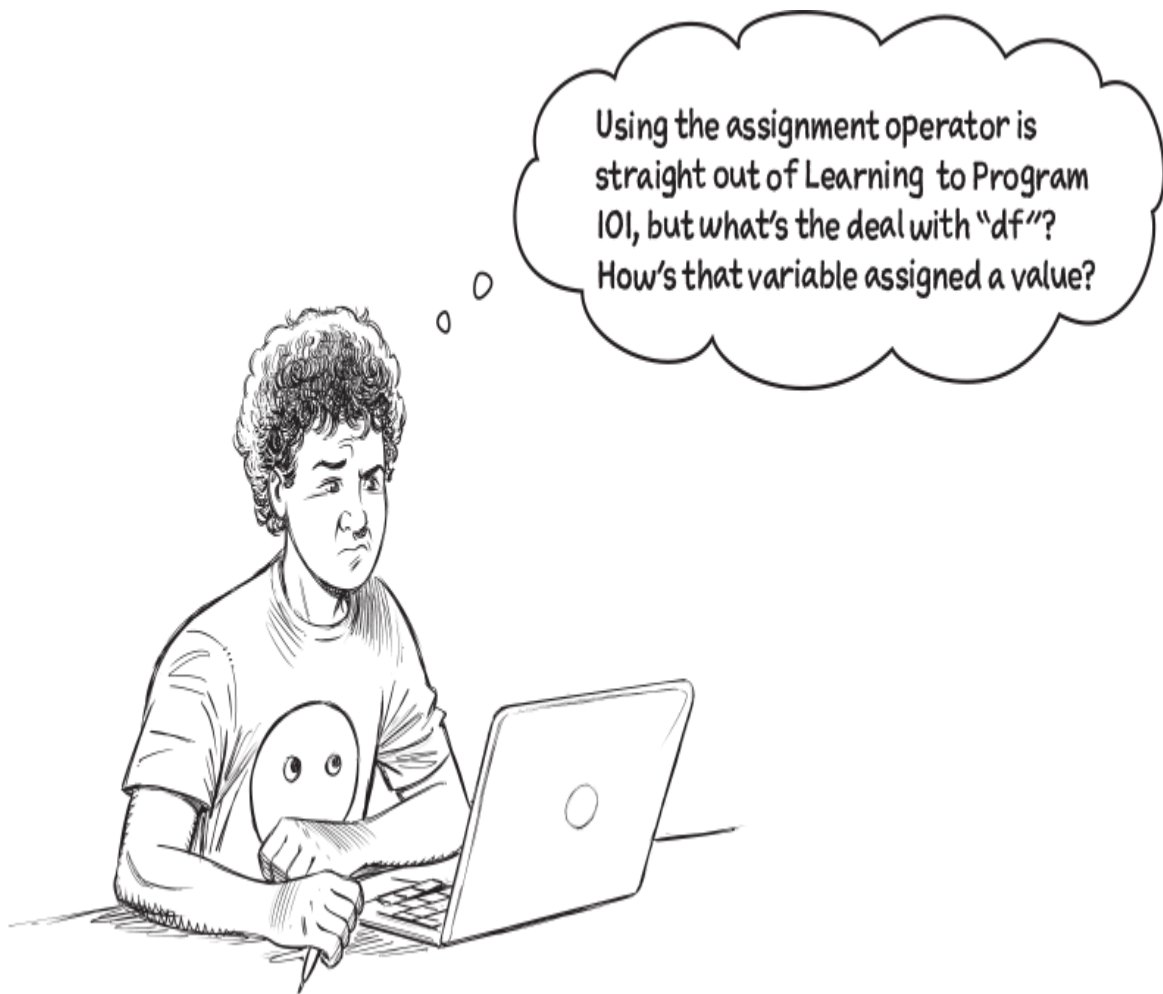
3 Two variables are created by the code: `df` and `data`.

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

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

Variables are created dynamically, as needed

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



The key word is “as”.

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

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

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

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

```
type(df)
```

```
io.TextIOWrapper
```

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

The "print dir" combo mambo strikes again!

```
print(dir(df))
```

```
['_CHUNK_SIZE', '_class_', '_del_', '_delattr_', '_dict_', '_dir_', '_doc_', '_enter_', '_eq_', '_exit_', '_format_', '_ge_', '_getattr_', '_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', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'reconfigure', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'write_through', 'writelines']
```

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

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!

```
type(data)
```

list

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.

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:

```
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 your looking at a list. Of course, you already know "data" is a list 'cause that's what the "type" BIF reported...

... but, this list's data values look a little weird. Until, that is, you realize you're looking at a list 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 single-line code block, a lot's happening here. Not only has your `data` list been created and populated with the data contained within the swimmer's file, but those two lines of code have managed to complete the first two sub-tasks for Task #2.

Take a look (over the 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 tick marks 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 correct, *whitespace*). In fact, in Python, curly braces delimited data, *not* code.

A code block in Python ends when the indentation ends.

In the **with** statement, the block contains a single 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* code 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, your code refuses to run!

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".**



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 forget to add it to the end of the line when introducing a new indented block. The Python interpreter complains **LOUDLY** when you do, as it's never (ever!) nice to forget you BFF.

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 level ends.

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?

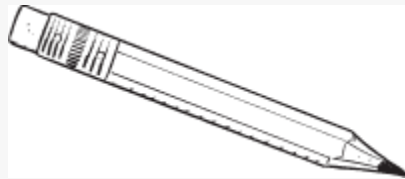
A: To be honest, it doesn't really matter whether you use four spaces, two spaces, or any number of spaces to indicate indentation. The tab character can also be used. What does matter is that whatever you do, your usage needs to be **consistent**. If you use four spaces on one line of code, you can't 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 Pythonistas configure their text editors to automatically replace a press of the tab key with four spaces. Also, be warned: if you are sharing code with other Python programmers and you haven't used four spaces to consistently indent your code, you better be ready to explain why.

Consistency is good.

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

SHARPEN YOUR PENCIL



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

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

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



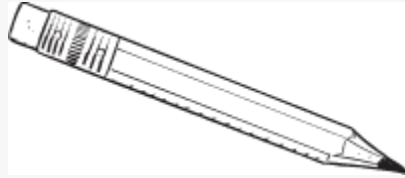
Yes, to both questions.

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

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

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

SHARPEN YOUR PENCIL SOLUTION



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

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

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

```
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
```

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

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

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:

Looking good.

```
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(",")
```

Both do the same thing.

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



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

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

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

Converting a time string into a time value

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

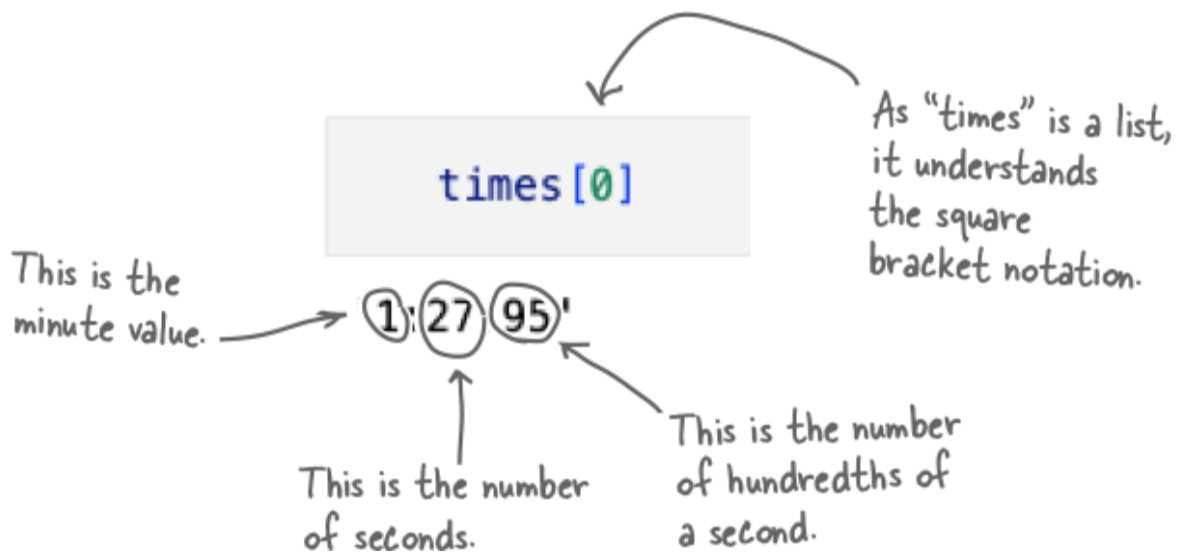
```
times
```

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

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. Assuming you can extract the numeric values from the string, converting the swim time to a number representing the time as hundredths of seconds should work. Here's how to do this:

1. Start with the string. → '1:27.95'

2. Extract the component parts. → '1' '27' '95'

3. Convert the strings to numbers with the help of another BIF called "int". → int('1') int('27') int('95')

4. Perform the calculation.

1 * 60 * 100 + 27 * 100 + 95

5. Convert the minutes to hundredths of seconds (there's 60 seconds in a minute, and 100 hundredths in a second).

6. Convert the seconds to hundredths of seconds (there's 100 hundredths in a second).

7. No conversion needed here (as this value is already in hundredths).

And there it is. The '1:27.95' time string converted to hundredths of seconds. → 8795

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

Swim times to hundredths of seconds

Ready Bake Code



The strategy described on the previous page can be turned into Python code without too much difficulty.

In the code which follows, some of the annotations from the previous page are converted to single-line comments which start with the `#` character and continue to the end of the line (and are, obviously, ignored by the Python interpreter).

VS Code, like most other editors, displays comments in a different color to your code.

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

This is a single-line comment.

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

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

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

If you type this code into a new code cell in your notebook, then press **Shift+Enter**, 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 awarded for guessing you need to employ a loop here.

Python's favorite looping mechanism: for

Like most programming languages, Python provides many ways to loop, with the **for** loop being a favorite of many Pythonistas. Let's look at a simple loop which takes each of the swim times 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

↖
As when using "with" earlier, this "for" loop has a single-line code block (suitably indented).



Cool, isn't it?

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

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

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

This is the sort of magic we love.

EXERCISE



Now that you've seen the **for** loop in action, take a moment to experiment in your notebook to combine the *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:

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

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

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

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

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).

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

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

EXERCISE SOLUTION



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

```
Loop over  
all the  
swim times. → for t in times:  
  
Extract  
the  
component  
parts. → { minutes, rest = t.split(":")  
           seconds, hundredths = rest.split(".")  
  
Display the  
calculated  
output. → print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

TEST DRIVE



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

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

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

↑
Whoa 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

Although, 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 haven't forgotten about the Coach's

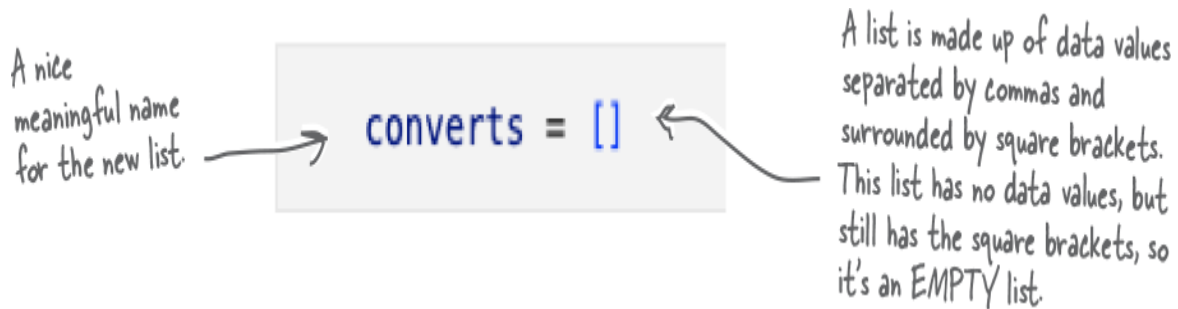
requirement to produce bar charts, so we're guessing these converted values will be needed at least once more., so it's likely best if we put them in another list while we perform the conversions.

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

Creating a new, empty list

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

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



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

```
type(converts)
```

It's a list... → list

```
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_', '_getattr_', '_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 dunder.

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.

Ooooooh. 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. 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?



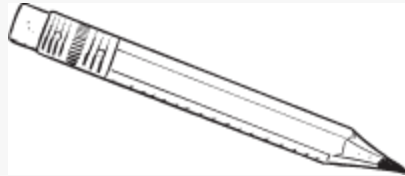
No, you do not need to worry.

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:

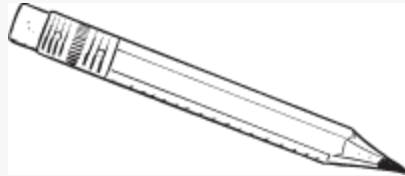
```
for t in times:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
    print(t, "->", int(minutes)*60*100 + int(seconds)*100 + int(hundredths))
```

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 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. Write your code in the space below (and, when you're ready, check you 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 which 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:

Don't forget the PSL – it's full of cool code.

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.



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

An experienced Python programmer knows enough to apply a few “tricks” to the problem of converting your hundredths of seconds back into the `mins:secs.hundredths` string format. You'll learn about these techniques later in this book, as showing them to you now would 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've walked through the five steps to perform the conversion. Here's what you're trying to do:

Be sure to follow along on your computer.

Given a value in hundredths of seconds...

8657.5

'1:26.58'

... transform the number into a time string in "mins:secs.hundredths" format.

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

Let's create a new variable to store the average.

```
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 its component parts.

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

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

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

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

3 Calculate the number of minutes.

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

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

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

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

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

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

```
minutes
```

1

And there's how many whole minutes there are in 66 seconds.

4 Calculate the number of seconds.

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

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

```
seconds
```

26

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

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

minutes, seconds, hundredths

(1, 26, '58') ← The three calculated values...

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

'1:26.58' ← ... are used to build a swim time string in the required format. And, yes, the + operator works with strings as well as numbers, performing concatenation.

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

average

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

'1:26.58'

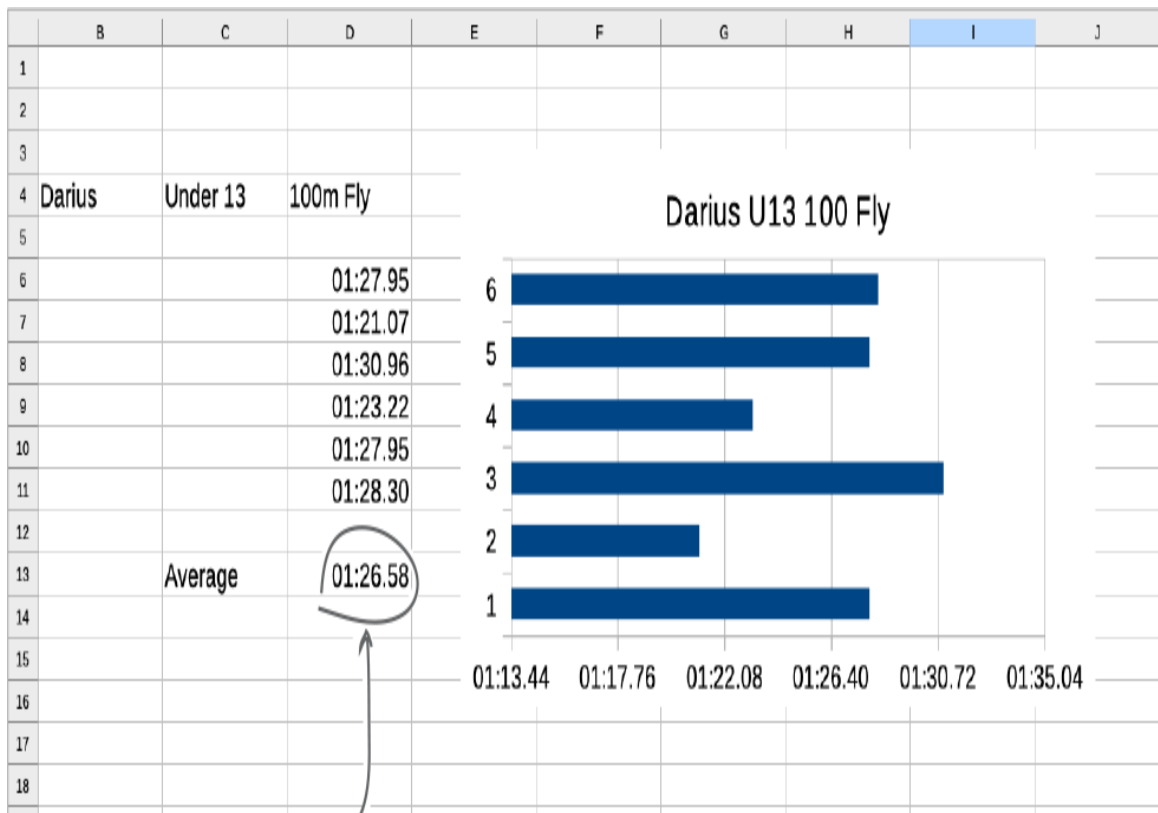


Yes, and it's easier than you think.

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.



The average time as calculated by the spreadsheet.

It's been a while since your last tick mark...

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 which 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 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 a doddle, 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 tick marks 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...

While you did all the work, I've just had the most amazing cup of coffee... Anyway, the code you have so far is great! But, can this code be adjusted to work with any data file? That would make this code really useful.



If something can be done once, it can be done again, and again, and again...

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.

Before getting to that point, it look's like someone else has another question.



I understand you're making great progress, which is just what I needed to hear after today's swim session. Now... you haven't forgotten about producing my bar charts, have you?

No, we haven't forgotten.

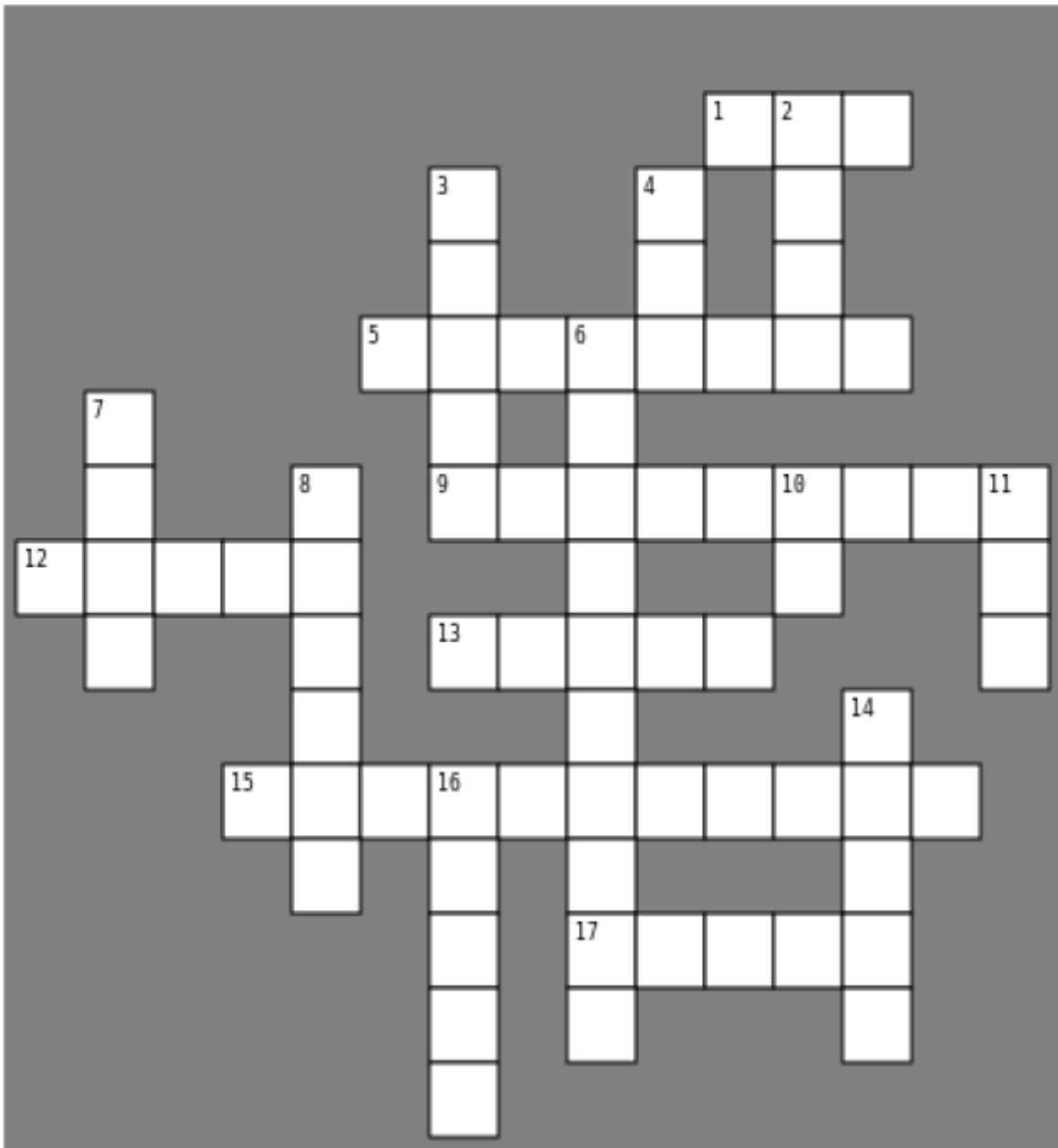
The next chapter lays the groundwork for getting to the point where you can tackle the graphing requirement, so bear with.

For now, let's conclude this chapter with another topical crossword puzzle. Enjoy!

The Listers 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**.

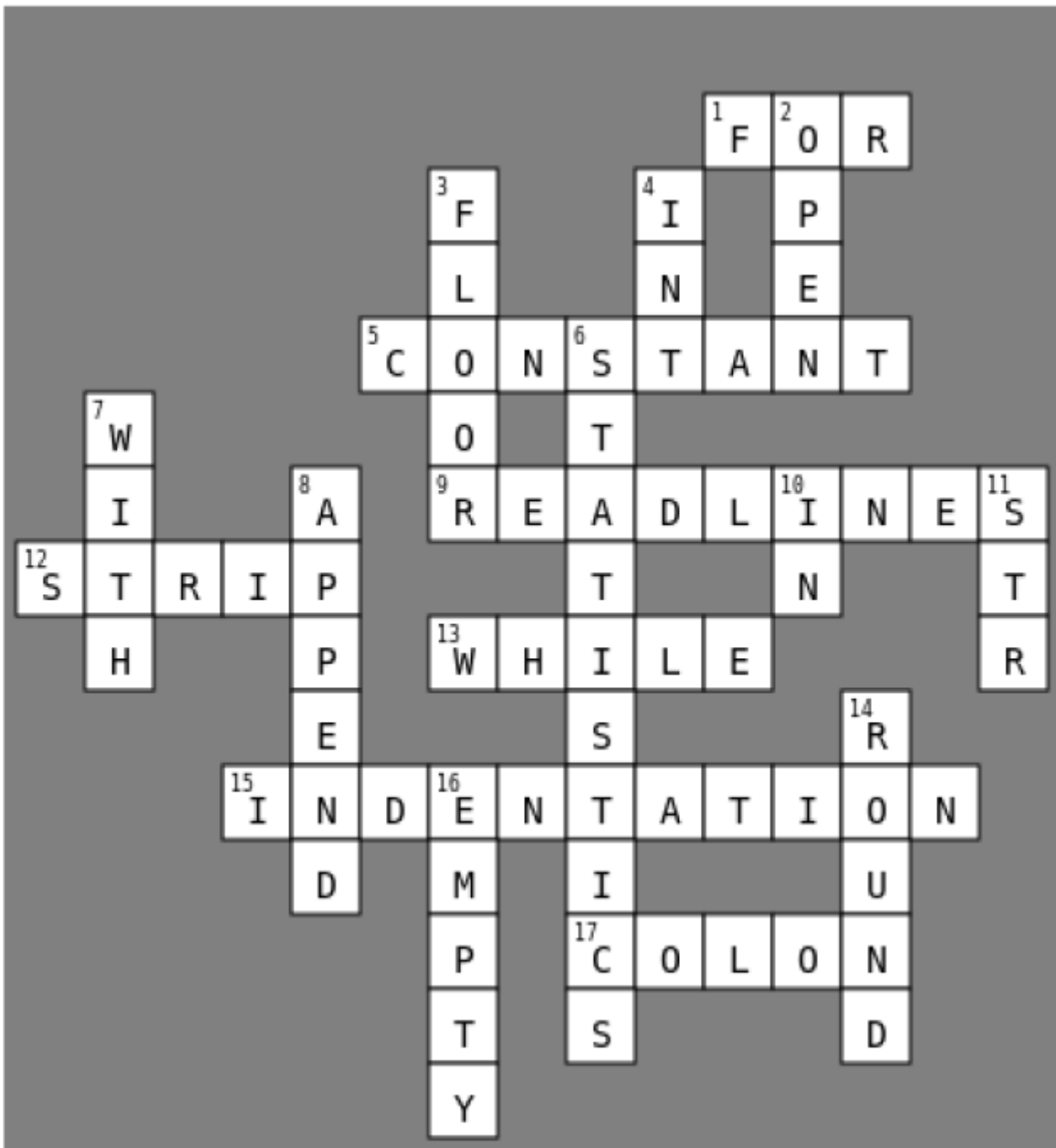
13. The less-used looping construct.
15. Another name for whitespace when used with code blocks.
17. Your new BFF.

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 Listers 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.

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.