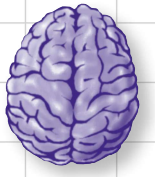
O'REILLY®

2nd Edition
Covers Python 3

Head First Python

A Brain-Friendly Guide



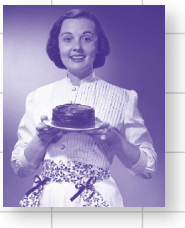
Load important Python concepts directly into your brain

Model data as lists, tuples, sets, and dictionaries

Don't get in a pickle: use DB-API instead

Objects? Decorators? Generators? They're all here.

Create a modern webapp with Flask



Share your code with modules

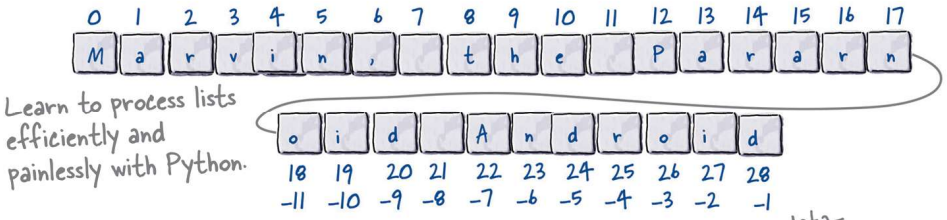
Paul Barry

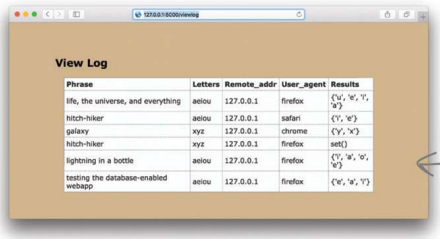
O'REILLY®

Python

What will you learn from this book?

Want to learn the Python language without slogging your way through how-to manuals? With *Head First Python*, you'll quickly grasp Python's fundamentals, working with the built-in data structures and functions. Then you'll move on to building your very own webapp, exploring database management, exception handling, and data wrangling. If you're intrigued by what you can do with context managers, decorators, comprehensions, and generators, it's all here. This second edition is a complete learning experience that will help you become a Python programmer in no time.





Gain the confidence to read, understand, and explain complex lines of code (while maintaining your sanity).

```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First Python* uses a visually rich format to engage your mind, rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multi-sensory learning experience is designed for the way your brain really works.

"A Python book should be as much fun as the language is. With *Head First Python*, master teacher Paul Barry delivers a quick-paced, entertaining and engaging guide to the language that will leave you well prepared to write real-world Python code."

— Dr. Eric Freeman, computer scientist, technology educator, former CTO of Disney Online

"*Head First Python* is a great introduction to both the language and how to use Python in the real world.... If you're looking for a great introduction to Python, then this is the place to start."

— David Griffiths, author and Agile coach

Python

US $49.99 **CAN $57.99**

ISBN: 978-1-491-91953-8

twitter.com/headfirstlabs
facebook.com/HeadFirst

oreilly.com
headfirstlabs.com

Advance Praise for *Head First Python*, Second Edition

"A Python book should be as much fun as the language is. With Head First Python, master teacher Paul Barry delivers a quick-paced, entertaining and engaging guide to the language that will leave you well prepared to write real-world Python code."

— Dr. Eric Freeman, computer scientist, technology educator, and former CTO of Disney Online

"*Head First Python* is a great introduction to both the language and how to use Python in the real world. It's full of practical advice on coding for the web and databases, and it doesn't shy away from difficult subjects like collections and immutability. If you're looking for a great introduction to Python, then this is the place to start."

— David Griffiths, author and Agile coach

"With major changes and updates from the first edition, this edition of Head First Python is sure to become a favourite in the rapidly growing collection of great Python guides. The content is structured to deliver high impact to the reader, and is heavily focused on being productive as soon as possible. All the necessary topics are covered with great clarity, and the entertaining delivery makes this book a delight to read."

— Caleb Hattingh, author of *20 Python Libraries You Aren't Using (But Should)* and *Learning Cython*

"Here's a clear and clean entry into the Python pool. No bellyflops, and you'll go deeper than you expected to."

— Bill Lubanovic, author of Introducing Python

Praise for the first edition

"*Head First Python* is a great introduction to not just the Python language, but Python as it's used in the real world. The book goes beyond the syntax to teach you how to create applications for Android phones, Google's App Engine, and more."

— David Griffiths, author and Agile coach

"Where other books start with theory and progress to examples, *Head First Python* jumps right in with code and explains the theory as you read along. This is a much more effective learning environment, because it engages the reader to *do* from the very beginning. It was also just a joy to read. It was fun without being flippant and informative without being condescending. The breadth of examples and explanation covered the majority of what you'll use in your job every day. I'll recommend this book to anyone starting out on Python."

— Jeremy Jones, coauthor of *Python for Unix and Linux System Administration*

Praise for other Head First books

"Kathy and Bert's *Head First Java* transforms the printed page into the closest thing to a GUI you've ever seen. In a wry, hip manner, the authors make learning Java an engaging 'what're they gonna do next?' experience."

— Warren Keuffel, *Software Development Magazine*

"Beyond the engaging style that drags you forward from know-nothing into exalted Java warrior status, *Head First Java* covers a huge amount of practical matters that other texts leave as the dreaded 'exercise for the reader.…' It's clever, wry, hip and practical—there aren't a lot of textbooks that can make that claim and live up to it while also teaching you about object serialization and network launch protocols."

— Dr. Dan Russell, Director of User Sciences and Experience Research IBM Almaden Research Center (and teaches Artificial Intelligence at Stanford University)

"It's fast, irreverent, fun, and engaging. Be careful—you might actually learn something!"

— Ken Arnold, former Senior Engineer at Sun Microsystems Coauthor (with James Gosling, creator of Java), *The Java Programming Language*

"I feel like a thousand pounds of books have just been lifted off of my head."

— Ward Cunningham, inventor of the Wiki and founder of the Hillside Group

"Just the right tone for the geeked-out, casual-cool guru coder in all of us. The right reference for practical development strategies—gets my brain going without having to slog through a bunch of tired, stale professor-speak."

— Travis Kalanick, cofounder and CEO of Uber

"There are books you buy, books you keep, books you keep on your desk, and thanks to O'Reilly and the Head First crew, there is the penultimate category, Head First books. They're the ones that are dog-eared, mangled, and carried everywhere. *Head First SQL* is at the top of my stack. Heck, even the PDF I have for review is tattered and torn."

— Bill Sawyer, ATG Curriculum Manager, Oracle

"This book's admirable clarity, humor and substantial doses of clever make it the sort of book that helps even nonprogrammers think well about problem-solving."

— Cory Doctorow, co-editor of Boing Boing Author, *Down and Out in the Magic Kingdom* and *Someone Comes to Town, Someone Leaves Town*

Praise for other Head First books

"I received the book yesterday and started to read it…and I couldn't stop. This is definitely très 'cool.' It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

— Erich Gamma, IBM Distinguished Engineer, and coauthor of *Design Patterns*

"One of the funniest and smartest books on software design I've ever read."

— Aaron LaBerge, VP Technology, ESPN.com

"What used to be a long trial and error learning process has now been reduced neatly into an engaging paperback."

— Mike Davidson, CEO, Newsvine, Inc.

"Elegant design is at the core of every chapter here, each concept conveyed with equal doses of pragmatism and wit."

— Ken Goldstein, Executive Vice President, Disney Online

"I ♥ *Head First HTML with CSS & XHTML*—it teaches you everything you need to learn in a 'fun-coated' format."

— Sally Applin, UI Designer and Artist

"Usually when reading through a book or article on design patterns, I'd have to occasionally stick myself in the eye with something just to make sure I was paying attention. Not with this book. Odd as it may sound, this book makes learning about design patterns fun.

"While other books on design patterns are saying 'Bueller…Bueller…Bueller…' this book is on the float belting out 'Shake it up, baby!'"

— Eric Wuehler

"I literally love this book. In fact, I kissed this book in front of my wife."

— Satish Kumar

Other related books from O'Reilly

Learning Python

Programming Python

Python in a Nutshell

Python Cookbook

Fluent Python

Other books in O'Reilly's Head First series

Head First Ajax

Head First Android Development

Head First C

Head First C#, Third Edition

Head First Data Analysis

Head First HTML and CSS, Second Edition

Head First HTML5 Programming

Head First iPhone and iPad Development, Third Edition

Head First JavaScript Programming

Head First jQuery

Head First Networking

Head First PHP & MySQL

Head First PMP, Third Edition

Head First Programming

Head First Python, Second Edition

Head First Ruby

Head First Servlets and JSP, Second Edition

Head First Software Development

Head First SQL

Head First Statistics

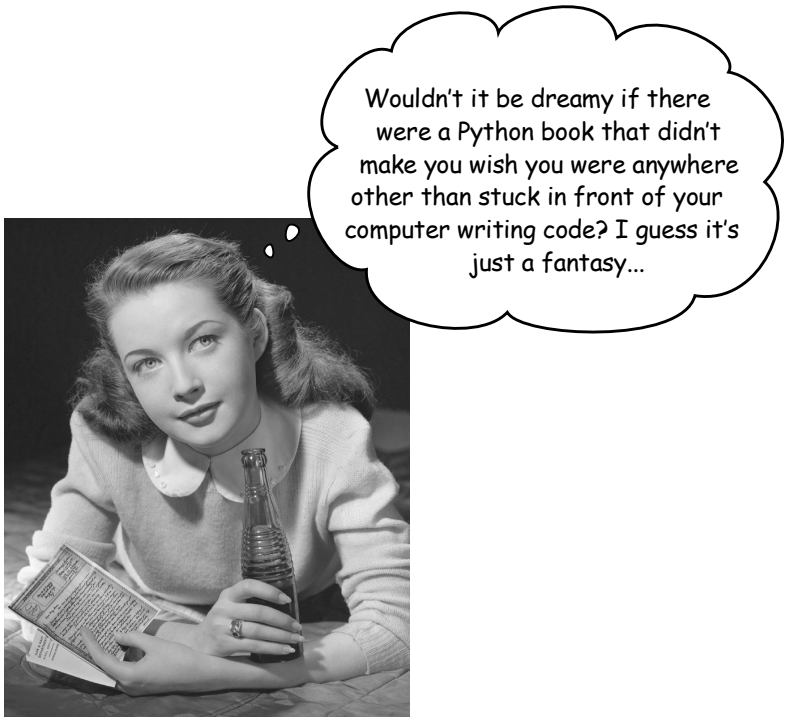
Head First Web Design

Head First WordPress

For a full list of titles, go to *headfirstlabs.com/books.php*.

Head First Python

Second Edition



Paul Barry

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First Python, Second Edition

by Paul Barry

Printed in the United States of America.

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

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

Series Creators: Kathy Sierra, Bert Bates

Editor: Dawn Schanafelt

Cover Designer: Randy Comer

Production Editor: Melanie Yarbrough

Proofreader: Rachel Monaghan

Indexer: Lucie Haskins

Page Viewers: Deirdre, Joseph, Aaron, and Aideen

Printing History:

November 2010: First edition.
November 2016: Second edition.

No weblogs were inappropriately searched in the making of this book, and the photos on this page (as well as the one on the author page) were supplied by *Aideen Barry*.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-1-491-91953-8

[M]

I continue to dedicate this book to all those generous people in the Python community who continue to help make Python what it is today.

And to all those that made learning Python and its technologies just complex enough that people need a book like *this* to learn it.

Author of Head First Python, 2nd Edition

While out walking, Paul pauses to discuss the correct pronunciation of the word "tuple" with his long-suffering wife.

This is Deirdre's usual reaction. ☺

Paul Barry lives and works in *Carlow*, *Ireland*, which is a small town of 35,000 people or so, located just over 80km southwest of the nation's capital: *Dublin*.

Paul 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. Paul is married to Deirdre, and they have three children (two of whom are now in college).

The Python programming language (and its related technologies) has formed an integral part of Paul's undergraduate courses since the 2007 academic year.

Paul is the author (or coauthor) of four other technical books: two on Python and two on *Perl*. In the past, he's written a heap of material for *Linux Journal Magazine*, where he was a contributing editor.

Paul was raised in *Belfast*, *Northern Ireland*, which may go some of the way toward explaining his take on things as well as his funny accent (unless, of course, you're also from "The North," in which case Paul's outlook and accent are *perfectly normal*).

Find Paul on *Twitter* (*@barrypj*), as well as at his home on the Web: *http://paulbarry.itcarlow.ie*.

Table of Contents (Summary)

Table of Contents (the real thing)

Intro

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

the basics

1 Getting Started Quickly

Get going with Python programming as quickly as possible.

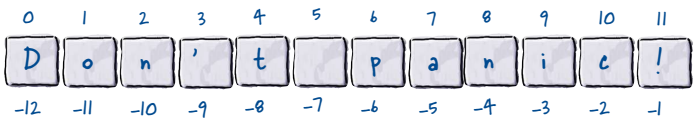
In this chapter, we introduce the basics of programming in Python, and we do this in typical *Head First* style: by jumping right in. After just a few pages, you'll have run your first sample program. By the end of the chapter, you'll not only be able to run the sample program, but you'll understand its code too (and more besides). Along the way, you'll learn about a few of the things that make **Python** the programming language it is.

list data

2 Working with Data

All programs process data, and Python programs are no exception.

In fact, take a look around: *data is everywhere*. A lot of, if not most, programming is all about data: *acquiring* data, *processing* data, *understanding* data. To work with data effectively, you need somewhere to *put* your data when processing it. Python shines in this regard, thanks (in no small part) to its inclusion of a handful of *widely applicable* data structures: **lists**, **dictionaries**, **tuples**, and **sets**. In this chapter, we'll preview all four, before spending the majority of this chapter digging deeper into **lists** (and we'll deep-dive into the other three in the next chapter). We're covering these data structures early, as most of what you'll likely do with Python will revolve around working with data.



structured data

3 Working with Structured Data

Python's list data structure is great, but it isn't a data panacea. When you have *truly* structured data (and using a list to store it may not be the best choice), Python comes to your rescue with its built-in **dictionary**. Out of the box, the dictionary lets you store and manipulate any collection of *key/value pairs*. We look long and hard at Python's dictionary in this chapter, and—along the way—meet **set** and **tuple**, too. Together with the **list** (which we met in the previous chapter), the dictionary, set, and tuple data structures provide a set of built-in data tools that help to make Python and data a powerful combination.

code reuse

4 Functions and Modules

Reusing code is key to building a maintainable system.

And when it comes to reusing code in Python, it all starts and ends with the humble **function**. Take some lines of code, give them a name, and you've got a function (which can be reused). Take a collection of functions and package them as a file, and you've got a **module** (which can also be reused). It's true what they say: *it's good to share*, and by the end of this chapter, you'll be well on your way to **sharing** and **reusing** your code, thanks to an understanding of how Python's functions and modules work.

module

building a webapp

5 Getting Real

At this stage, you know enough Python to be dangerous.

With this book's first four chapters behind you, you're now in a position to productively use Python within any number of application areas (even though there's still lots of Python to learn). Rather than explore the long list of what these application areas are, in this and subsequent chapters, we're going to structure our learning around the development of a web-hosted application, which is an area where Python is especially strong. Along the way, you'll learn a bit more about Python.

storing and manipulating data

6

Where to Put Your Data

Sooner or later, you'll need to safely store your data somewhere.

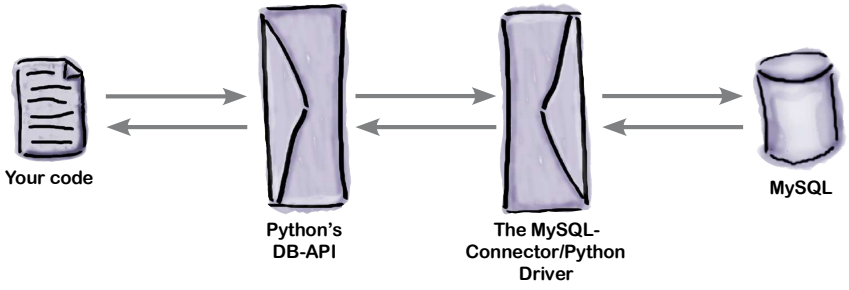
And when it comes to **storing data**, Python has you covered. In this chapter, you'll learn about storing and retrieving data from *text files*, which—as storage mechanisms go—may feel a bit simplistic, but is nevertheless used in many problem areas. As well as storing and retrieving your data from files, you'll also learn some tricks of the trade when it comes to manipulating data. We're saving the "serious stuff" (storing data in a database) until the next chapter, but there's plenty to keep us busy for now when working with files.

Form Data	Remote_addr	User_agent	Results
ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526 .106 Safari/537.36	{'e', 'i'}

using a database

7 Putting Python's DB-API to Use

Storing data in a relational database system is handy. In this chapter, you'll learn how to write code that interacts with the popular **MySQL** database technology, using a generic database API called **DB-API**. The DB-API (which comes standard with every Python install) allows you to write code that is easily transferred from one database product to the next... assuming your database talks SQL. Although we'll be using MySQL, there's nothing stopping you from using your DB-API code with your favorite relational database, whatever it may be. Let's see what's involved in using a relational database with Python. There's not a lot of new Python in this chapter, but using Python to talk to databases is a **big deal**, so it's well worth learning.



a little bit of class

8 Abstracting Behavior and State

Classes let you bundle code behavior and state together.

In this chapter, you're setting your webapp aside while you learn about creating Python **classes.** You're doing this in order to get to the point where you can create a context manager with the help of a Python class. As creating and using classes is such a useful thing to know about anyway, we're dedicating this chapter to them. We won't cover everything about classes, but we'll touch on all the bits you'll need to understand in order to confidently create the context manager your webapp is waiting for.

countfromby.py - /Users/paul/Desktop/_NewBook/ch07/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr
```

Ln: 2 Col: 0

the context management protocol

9 Hooking into Python's with Statements

It's time to take what you've just learned and put it to work.

Chapter 7 discussed using a **relational database** with Python, while Chapter 8 provided an introduction to using **classes** in your Python code. In this chapter, both of these techniques are combined to produce a **context manager** that lets us extend the `with` statement to work with relational database systems. In this chapter, you'll hook into the `with` statement by creating a new class, which conforms to Python's **context management protocol**.

File Edit Window Help Checking our log DB

```
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase                   | letters | ip        | browser_string | results              |
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker              | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2016-03-09 13:42:15 | galaxy                   | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}           |
|  4 | 2016-03-09 13:43:07 | hitch-hiker              | xyz     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

function decorators

10 Wrapping Functions

When it comes to augmenting your code, Chapter 9's context management protocol is not the only game in town. Python also lets you use function **decorators**, a technique whereby you can add code to an existing function *without* having to change any of the existing function's code. If you think this sounds like some sort of black art, don't despair: it's nothing of the sort. However, as coding techniques go, creating a function decorator is often considered to be on the harder side by many Python programmers, and thus is not used as often as it should be. In this chapter, our plan is to show you that, despite being an advanced technique, creating and using your own decorators is not that hard.

exception handling

11 What to Do When Things Go Wrong

Things go wrong, all the time—no matter how good your code is.

You've successfully executed all of the examples in this book, and you're likely confident all of the code presented thus far works. But does this mean the code is robust? Probably not. Writing code based on the assumption that nothing bad ever happens is (at best) naive. At worst, it's dangerous, as unforeseen things do (and will) happen. It's much better if you're wary while coding, as opposed to trusting. Care is needed to ensure your code does what you want it to, as well as reacts properly when things go south.

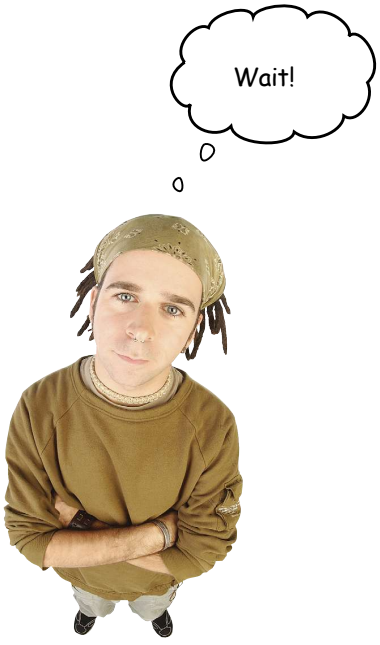

```
   ...
Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |     +-- FloatingPointError
      |     +-- OverflowError
      |     +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
            ...
```

a little bit of threading

11 3/4 Dealing with Waiting

Your code can sometimes take a long time to execute.

Depending on who notices, this may or may not be an issue. If some code takes 30 seconds to do its thing "behind the scenes," the wait may not be an issue. However, if your user is waiting for your application to respond, and it takes 30 seconds, everyone notices. What you should do to fix this problem depends on what you're trying to do (and who's doing the waiting). In this short chapter, we'll briefly discuss some options, then look at one solution to the issue at hand: *what happens if something takes too long?*



advanced iteration

12 Looping Like Crazy

It's often amazing how much time our programs spend in loops.

This isn't a surprise, as most programs exist to perform something quickly a whole heap of times. When it comes to optimizing loops, there are two approaches: (1) improve the loop syntax (to make it easier to specify a loop), and (2) improve how loops execute (to make them go faster). Early in the lifetime of Python 2 (that is, a *long, long* time ago), the language designers added a single language feature that implements both approaches, and it goes by a rather strange name: **comprehension**.

installation

a

Installing Python

First things first: let's get Python installed on your computer.

Whether you're running on *Windows*, *Mac OS X*, or *Linux*, Python's got you covered. How you install it on each of these platforms is specific to how things work on each of these operating systems (we know...a shocker, eh?), and the Python community works hard to provide installers that target all the popular systems. In this short appendix, you'll be guided through installing Python on your computer.

pythonanywhere

b

Deploying Your Webapp

At the end of Chapter 5, we claimed that deploying your webapp to the cloud was only 10 minutes away. It's now time to make good on that promise. In this appendix, we are going to take you through the process of deploying your webapp on *PythonAnywhere*, going from zero to deployed in about 10 minutes. *PythonAnywhere* is a favorite among the Python programming community, and it's not hard to see why: it works exactly as you'd expect it to, has great support for Python (and Flask), and—best of all—you can get started hosting your webapp at no cost.

top ten things we didn't cover

There's Always More to Learn

It was never our intention to try to cover everything. This book's goal was always to show you enough Python to get you up to speed as quickly as possible. There's a lot more we could've covered, but didn't. In this appendix, we discuss the top 10 things that—given another 600 pages or so—we would've eventually gotten around to. Not all of the 10 things will interest you, but quickly flip through them just in case we've hit on your sweet spot, or provided an answer to that nagging question. All the programming technologies in this appendix come baked in to Python and its interpreter.

top ten projects not covered

Even More Tools, Libraries, and Modules

We know what you're thinking as you read this appendix's title.

Why on Earth didn't they make the title of the last appendix: *The Top Twenty Things We Didn't Cover*? Why *another* 10? In the last appendix, we limited our discussion to stuff that comes baked in to Python (part of the language's "batteries included"). In this appendix, we cast the net much further afield, discussing a whole host of technologies that are available to you *because* Python exists. There's lots of good stuff here and—just like with the last appendix—a quick perusal won't hurt you *one single bit*.

getting involved

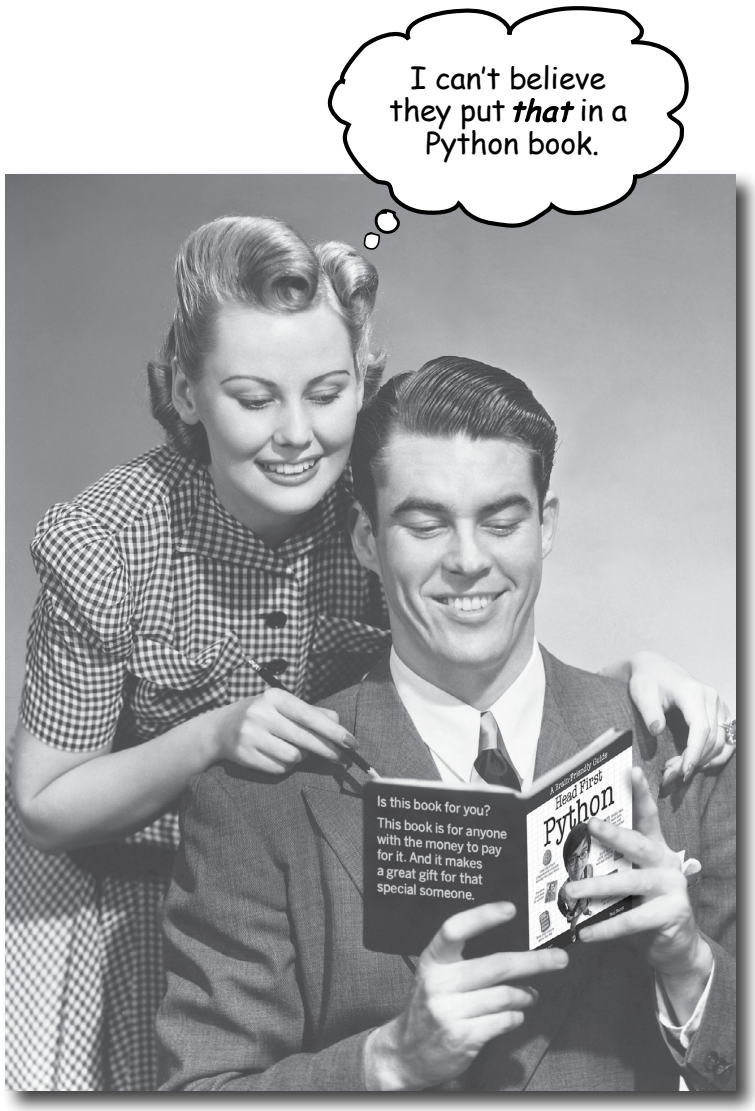
The Python Community

Python is much more than a great programming language.

It's a great community, too. The Python Community is welcoming, diverse, open, friendly, sharing, and giving. We're just amazed that no one, to date, has thought to put that on a greeting card! Seriously, though, there's more to programming in Python than the language. An entire ecosystem has grown up around Python, in the form of excellent books, blogs, websites, conferences, meetups, user groups, and personalities. In this appendix, we take a survey of the Python community and see what it has to offer. Don't just sit around programming on your own: **get involved**!

how to use this book

Intro



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

Who Is This Book For?

If you can answer "yes" to all of these:

1. Do you already know how to program in another programming language?
2. Do you wish you had the know-how to program Python, add it to your list of tools, and make it do new things?
3. Do you prefer actually doing things and applying the stuff you learn over listening to someone in a lecture rattle on for hours on end?

this book is for you.

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Do you already know most of what you need to know to program with Python?
2. Are you looking for a reference book to Python, one that covers all the details in excruciating detail?
3. Would you rather have your toenails pulled out by 15 screaming monkeys than learn something new? Do you believe a Python book should cover *everything* and if it bores the reader to tears in the process, then so much the better?

this book is ***not*** for you.

This is NOT a reference book, and we assume you've programmed before.

[Note from marketing: this book is for anyone with a credit card... we'll accept a check, too.]

We Know What You're Thinking

"How can *this* be a serious Python book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

We know what your *brain* is thinking

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

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

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

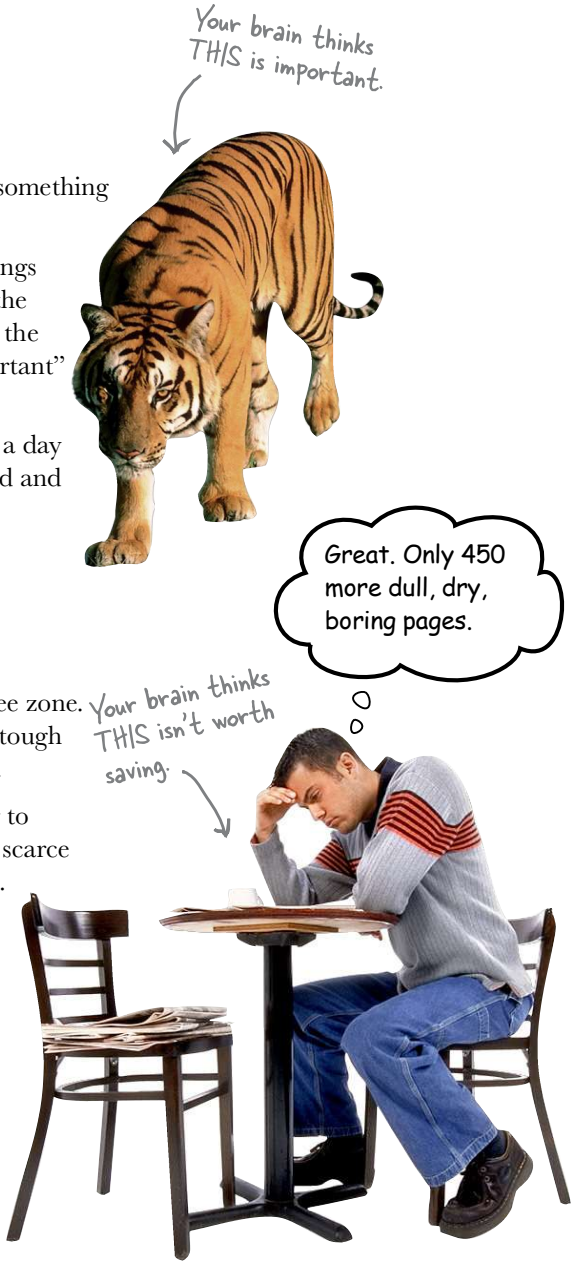
Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows…

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, 10 days at the most.

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



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

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

Some of the Head First learning principles:

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

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion or a lecture?

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

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

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

Metacognition: Thinking About Thinking

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

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

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

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

So just how *DO* you get your brain to treat programming like it was a hungry tiger?

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

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

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

But pictures and conversational style are just the beginning…

Here's What WE Did:

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

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

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

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

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

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

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

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

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

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

Cut this out and stick it on your refrigerator.

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

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

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

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

3. **Read the "There Are No Dumb Questions" sections.**
That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**
Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

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

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

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

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

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

Read Me, 1 of 2

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

This book is designed to get you up to speed as quickly as possible.

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

This book targets Python 3

We use Release 3 of the Python programming language in this book, and we cover how to get and install Python 3 in *Appendix A*. This book does **not** use Python 2.

We put Python to work for you right away.

We get you doing useful stuff in Chapter 1 and build from there. There's no hanging around, because we want you to be *productive* with Python right away.

The activities are NOT optional—you have to do the work.

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

The redundancy is intentional and important.

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

The examples are as lean as possible.

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

Read Me, 2 of 2

Yes, there's more...

This second edition is NOT at all like the first.

This is an update to the first edition of *Head First Python*, which published late in 2010. Although that book and this one share the same author, he's now older and (hopefully) wiser, and thus, decided to completely rewrite the first edition's content for this edition. So...*everything* is new: the order is different, the content has been updated, the examples are better, and the stories are either gone or have been replaced. We kept the cover—with minor amendments—as we figured we didn't want to rock the boat too much. It's been a long six years...we hope you enjoy what we've come up with.

Where's the code?

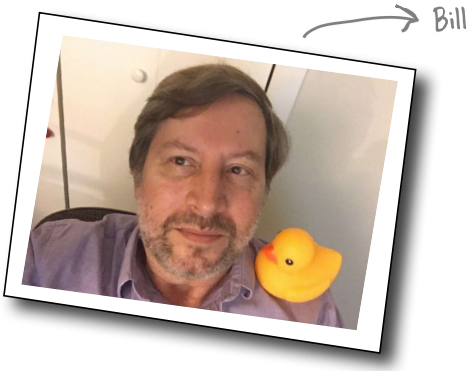
We've placed the code examples on the Web so you can copy and paste them as needed (although we do recommend that you type in the code *as you follow along*). You'll find the code at these locations:

http://bit.ly/head-first-python-2e

http://python.itcarlow.ie

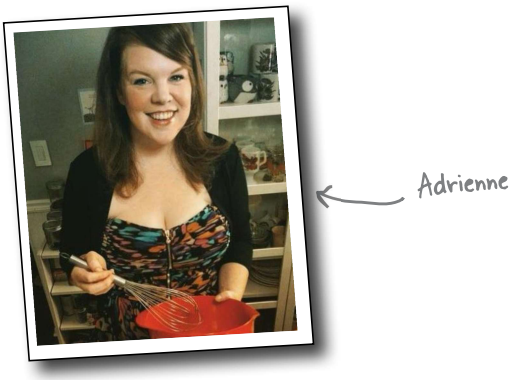
The Technical Review Team

Bill Lubanovic has been a developer and admin for forty years. He's also written for O'Reilly: chapters for two Linux security books, co-authored a Linux admin book, and solo "Introducing Python". He lives by a frozen lake in the Sangre de Sasquatch mountains of Minnesota with one lovely wife, two lovely children, and three fur-laden cats.



Edward Yue Shung Wong has been hooked on coding since he wrote his first line of Haskell in 2006. Currently he works on event driven tradeprocessing in the heart of the City of London. He enjoys sharing his passion for development with the London Java Community and Software Craftsmanship Community. Away from the keyboard, find Edward in his element on a football pitch or gaming on YouTube (@arkangelofkaos).

Adrienne Lowe is a former personal chef from Atlanta turned Python developer who shares stories, conference recaps, and recipes at her cooking and coding blog Coding with Knives (http://codingwithknives.com). She organizes PyLadiesATL and Django Girls Atlanta and runs the weekly Django Girls "Your Django Story" interview series for women in Python. Adrienne works as a Support Engineer at Emma Inc., as Director of Advancement of the Django Software Foundation, and is on the core team of Write the Docs. She prefers a handwritten letter to email and has been building out her stamp collection since childhood.



Monte Milanuk provided valuable feedback.

Acknowledgments and Thanks

My editor: This edition's editor is **Dawn Schanafelt**, and this book is much, much better for Dawn's involvement. Not only is Dawn a great editor, but her eye for detail and the right way to express things has greatly improved what's written here. *O'Reilly Media* make a habit of hiring bright, friendly, capable people, and Dawn is the very personification of these attributes.

The O'Reilly Media team: This edition of *Head First Python* took four years to write (it's a long story). It's only natural, then, that a lot of people from the *O'Reilly Media* team were involved. **Courtney Nash** talked me into doing "a quick rewrite" in 2012, then was on hand as the project's scope ballooned. Courtney was this edition's first editor, and was on hand when disaster struck and it looked like this book was doomed. As things *slowly* got back on track, Courtney headed off to bigger and better things within *O'Reilly Media*, handing over the editing reins in 2014 to the very busy **Meghan Blanchette**, who watched (I'm guessing, with mounting horror) as delay piled upon delay, and this book went on and off the tracks at regular intervals. Things were only just getting back to normal when Meghan went off to pastures new, and Dawn took over as this book's editor. That was one year ago, and the bulk of this book's 12¾ chapters were written under Dawn's ever-watchful eye. As I mentioned above, *O'Reilly Media* hires good people, and Courtney and Meghan's editing contributions and support are gratefully acknowledged. Elsewhere, thanks are due to **Maureen Spencer**, **Heather Scherer**, **Karen Shaner**, and **Chris Pappas** for working away "behind the scenes." Thanks, also, to the invisible unsung heroes known as **Production**, who took my *InDesign* chapters and turned them into this finished product. They did a great job.

A shout-out to **Bert Bates** who, together with **Kathy Sierra**, created this series of books with their wonderful *Head First Java*. Bert spent a lot of time working with me to ensure this edition was firmly pointed in the right direction.

Friends and colleagues: My thanks again to **Nigel Whyte** (Head of the *Department of Computing* at the *Institute of Technology, Carlow*) for supporting my involvement in this rewrite. Many of my students had a lot of this material thrust upon them as part of their studies, and I hope they get a chuckle out of seeing one (or more) of their classroom examples on the printed page.

Thanks once again to **David Griffiths** (my partner-in-crime on *Head First Programming*) for telling me at one particularly low point to stop agonizing over everything and just *write the damned thing!* It was perfect advice, and it's great to know that David, together with Dawn (his wife and Head First coauthor), is only ever an email away. Be sure to check out David and Dawn's great Head First books.

Family: My family (wife **Deirdre**, and children **Joseph**, **Aaron,** and **Aideen**) had to endure four years of ups-and-downs, fits-and-starts, huffs-and-puffs, and a life-changing experience from which we all managed to come through with our wits, thankfully, still intact. This book survived, I survived, and our family survived. I'm very thankful and love them all, and I know I don't need to say this, but will: *I do this for you guys*.

The without-whom list: My technical review team did an excellent job: check out their mini-profiles on the previous page. I considered all of the feedback they gave me, fixed all the errors they found, and was always rather chuffed when any of them took the time to tell me what a great job I was doing. I'm very grateful to them all.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

1 the basics

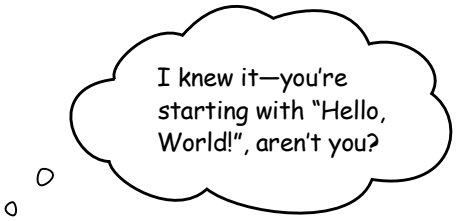
Getting Started Quickly

Get going with Python programming as quickly as possible.

In this chapter, we introduce the basics of programming in Python, and we do this in typical *Head First* style: by jumping right in. After just a few pages, you'll have run your first sample program. By the end of the chapter, you'll not only be able to run the sample program, but you'll understand its code too (and more besides). Along the way, you'll learn about a few of the things that make **Python** the programming language it is. So, let's not waste any more time. Flip the page and let's get going!

Breaking with Tradition

Pick up almost any book on a programming language, and the first thing you'll see is the *Hello World* example.



No, we aren't.

This is a *Head First* book, and we do things differently 'round here. With other books, there is a tradition to start by showing you how to write the *Hello World* program in the language under consideration. However, with Python, what you end up with is a single statement that invokes Python's built-in `print` function, which displays the traditional "Hello, World!" message on screen. It's almost too exciting...and it teaches you next to nothing.

So, no, we aren't going to show you the *Hello World* program in Python, as there's really nothing to learn from it. We're going to take a different path...

Starting with a meatier example

Our plan for this chapter is to start with an example that's somewhat larger and, consequently, more useful than *Hello World*.

We'll be right up front and tell you that the example we have is somewhat *contrived*: it does do something, but may not be entirely useful in the long run. That said, we've chosen it to provide a vehicle with which to cover a lot of Python in as short a timespan as possible. And we promise by the time you've worked through the first example program, you'll know enough to write *Hello World* in Python without our help.

Jump Right In

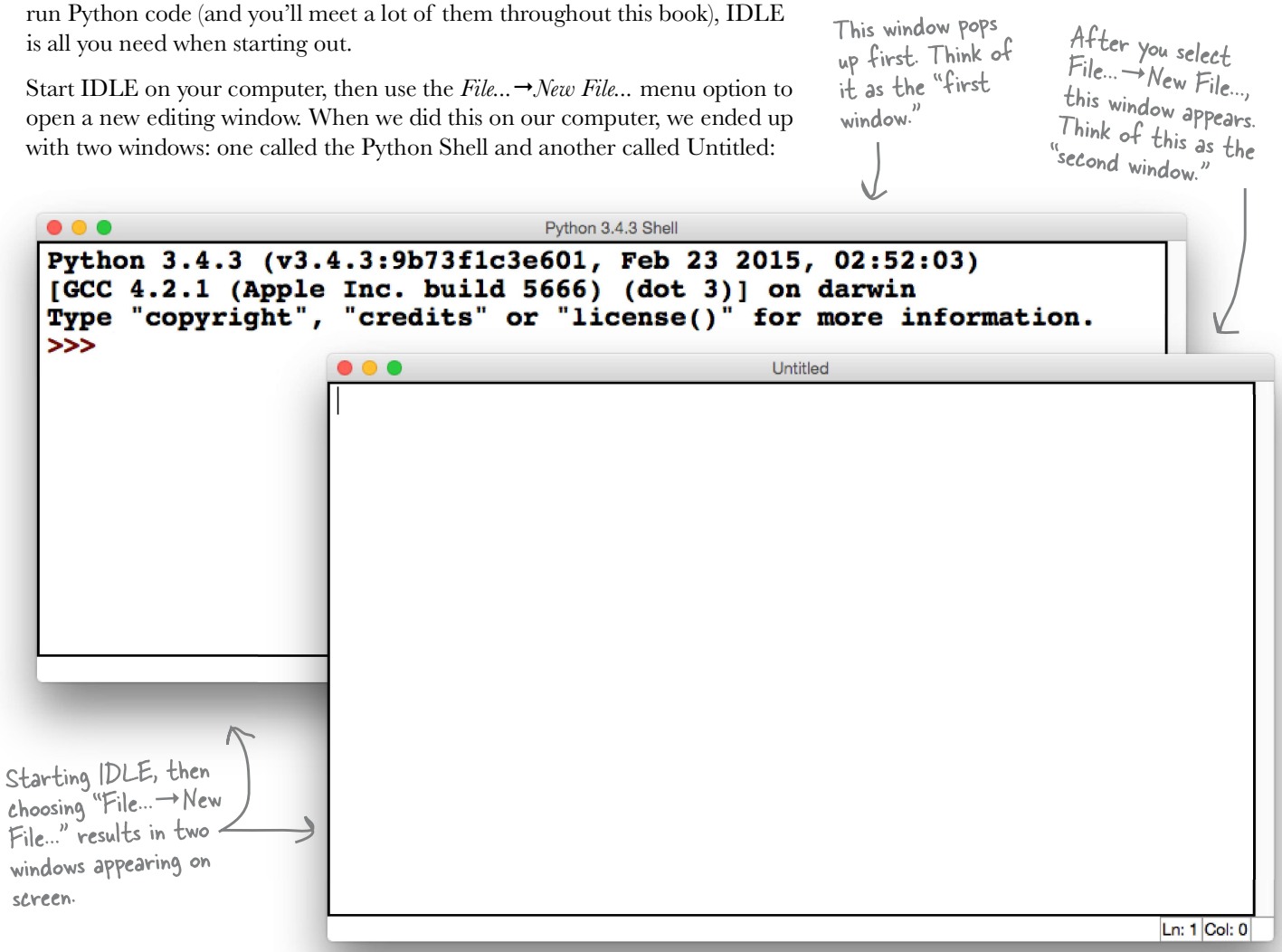
If you haven't already installed a version of Python 3 on your computer, pause now and head on over to Appendix A for some step-by-step installation instructions (it'll only take a couple minutes, promise).

With the latest Python 3 installed, you're ready to start programming Python, and to help with this—for now—we're going to use Python's built-in integrated development environment (IDE).

Python's IDLE is all you need to get going

When you install Python 3 on your computer, you also get a very simple yet usable IDE called IDLE. Although there are many different ways in which to run Python code (and you'll meet a lot of them throughout this book), IDLE is all you need when starting out.

Start IDLE on your computer, then use the *File...*→*New File...* menu option to open a new editing window. When we did this on our computer, we ended up with two windows: one called the Python Shell and another called Untitled:



Understanding IDLE's Windows

Both of these IDLE windows are important.

The first window, the Python Shell, is a REPL environment used to run snippets of Python code, typically a single statement at a time. The more you work with Python, the more you'll come to love the Python Shell, and you'll be using it a lot as you progress through this book. For now, though, we are more interested in the second window.

The second window, Untitled, is a text editing window that can be used to write complete Python programs. It's not the greatest editor in the world (as that honor goes to <*insert your favorite text editor's name here*>), but IDLE's editor is quite usable, and has a bunch of modern features built right in, including color-syntax handling and the like.

As we are jumping right in, let's go ahead and enter a small Python program into this window. When you are done typing in the code below, use the *File...*→*Save...* menu option to save your program under the name `odd.py`.

Geek Bits

What does REPL mean?

It's geek shorthand for "read-eval-print-loop," and describes an interactive programming tool that lets you experiment with snippets of code to your heart's desire. Find out way more than you need to know by visiting *http://en.wikipedia.org/wiki/Read-eval-print_loop*.

Be sure to enter the code *exactly* as shown here:

Don't worry about what this code does for now. Just type it into the editing window. Be sure to save it as "odd.py" before continuing.

odd.py - /Users/Paul/Desktop/_NewBook/ch01/odd.py (3.4.3)

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")

```

Ln: 15 Col: 0

So...*now what?* If you're anything like us, you can't wait to run this code, right? Let's do this now. With your code in the edit window (as shown above), press the F5 key on your keyboard. A number of things can happen...

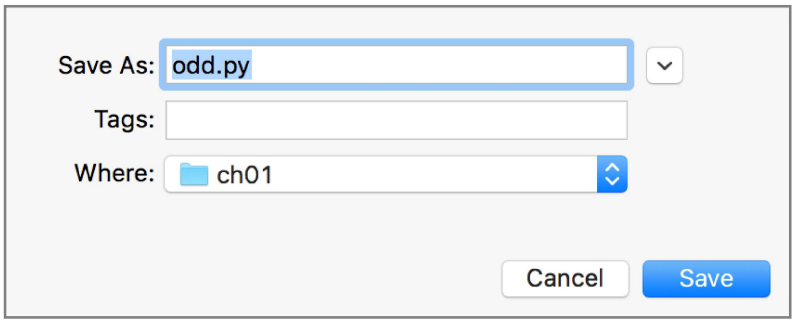
What Happens Next...

If your code ran without error, flip over to the next page, and *keep going*.

If you forgot to save your code *before* you tried to run it, IDLE complains, as you have to save any new code to a file *first*. You'll see a message similar to this one if you didn't save your code:

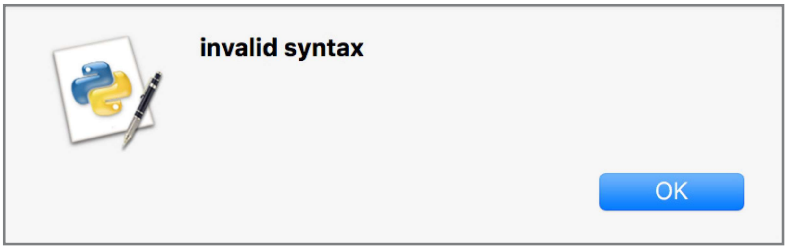
By default, IDLE won't run code that hasn't been saved.

Click the OK button, then provide a name for your file. We've chosen `odd` as the name for our file, and we've added a `.py` extension (which is a Python convention well worth adhering to):



You are free to use whatever name you like for your program, but it's probably best—if you're following along—to stick to the same name as us.

If your code now runs (having been saved), flip over to the next page, and *keep going*. If, however, you have a syntax error somewhere in your code, you'll see this message:



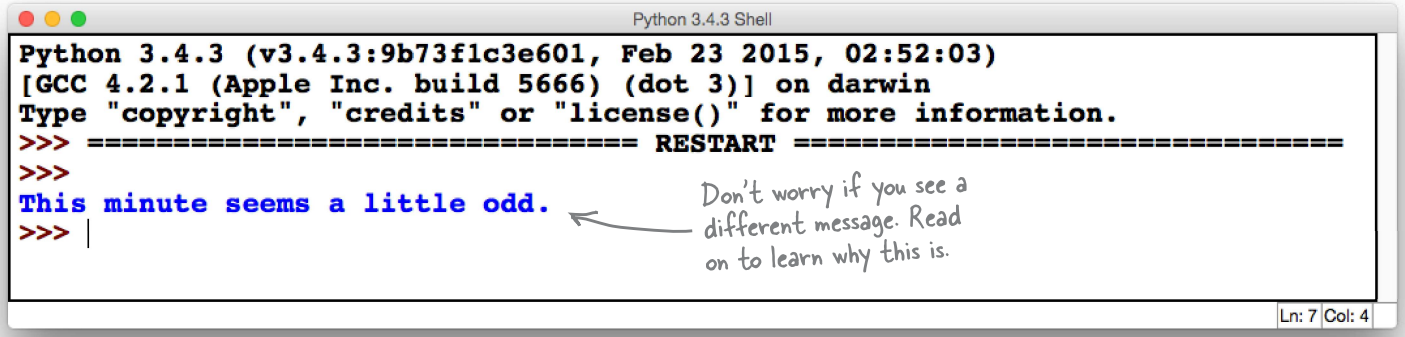
As you can no doubt tell, IDLE isn't great at stating what the syntax error is. But click OK, and a large red block indicates where IDLE thinks the problem is.

Click the OK button, then note where IDLE thinks the syntax error is: look for the large red block in the edit window. Make sure your code matches ours exactly, save your file again, and then press F5 to ask IDLE to execute your code once more.

Press F5 to Run Your Code

Pressing F5 executes the code in the currently selected IDLE text-editing window—assuming, of course, that your code doesn't contain a runtime error. If you have a runtime error, you'll see a **Traceback** error message (in red). Read the message, then return to the edit window to make sure the code you entered is exactly the same as ours. Save your amended code, then press F5 again. When we pressed F5, the Python Shell became the active window, and here's what we saw:

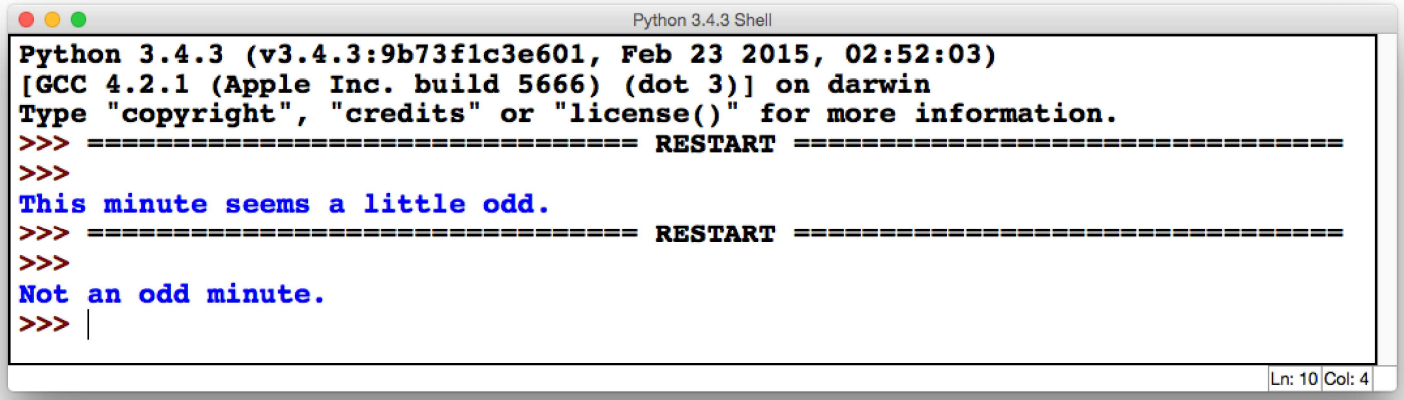
From this point on, we'll refer to "the IDLE text-editing window" simply as "the edit window."



Don't worry if you see a different message. Read on to learn why this is.

Depending on what time of day it is, you may have seen the *Not an odd minute* message instead. Don't worry if you did, as this program displays one or the other message depending on whether your computer's current time contains a minute value that's an odd number (we did say this example was *contrived*, didn't we?). If you wait a minute, then click the edit window to select it, then press F5 again, your code runs again. You'll see the other message this time (assuming you waited the required minute). Feel free to run this code as often as you like. Here is what we saw when we (very patiently) waited the required minute:

Pressing F5 while in the edit window runs your code, then displays the resulting output in the Python Shell.

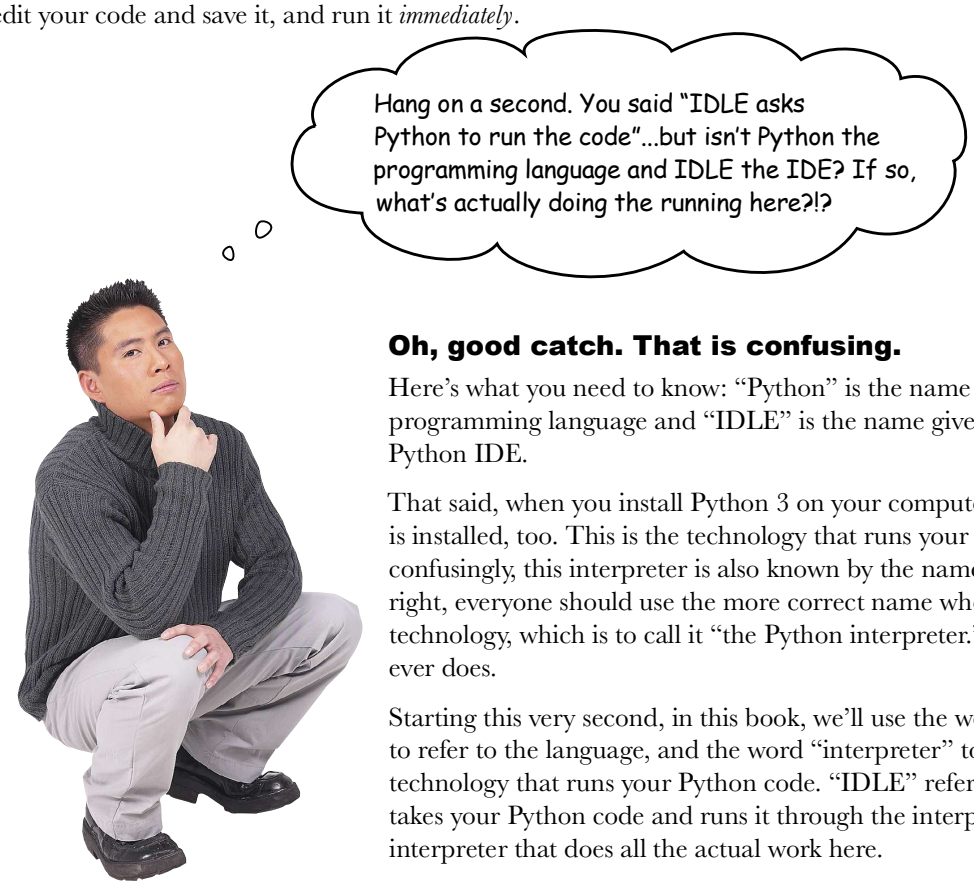


Let's spend some time learning how this code runs.

Code Runs Immediately

When IDLE asks Python to run the code in the edit window, Python starts at the top of the file and begins executing code straightaway.

For those of you coming to Python from one of the C-like languages, note that there is no notion of a `main()` function or method in Python. There's also no notion of the familiar edit-compile-link-run process. With Python, you edit your code and save it, and run it *immediately*.



Oh, good catch. That is confusing.

Here's what you need to know: "Python" is the name given to the programming language and "IDLE" is the name given to the built-in Python IDE.

That said, when you install Python 3 on your computer, an **interpreter** is installed, too. This is the technology that runs your Python code. Rather confusingly, this interpreter is also known by the name "Python." By right, everyone should use the more correct name when referring to this technology, which is to call it "the Python interpreter." But, alas, nobody ever does.

Starting this very second, in this book, we'll use the word "Python" to refer to the language, and the word "interpreter" to refer to the technology that runs your Python code. "IDLE" refers to the IDE, which takes your Python code and runs it through the interpreter. It's the interpreter that does all the actual work here.

there are no Dumb Questions

Q: Is the Python interpreter something like the Java VM?

A: Yes and no. Yes, in that the interpreter runs your code. But no, in how it does it. In Python, there's no real notion of your source code being compiled into an "executable." Unlike the Java VM, the interpreter doesn't run `.class` files, it just runs your code.

Q: But, surely, compilation has to happen at some stage?

A: Yes, it does, but the interpreter does not expose this process to the Python programmer (you). All of the details are taken care of for you. All you see is your code running as IDLE does all the heavy lifting, interacting with the interpreter on your behalf. We'll talk more about this process as this book progresses.

Executing Code, One Statement at a Time

Here is the program code from page 4 again:

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

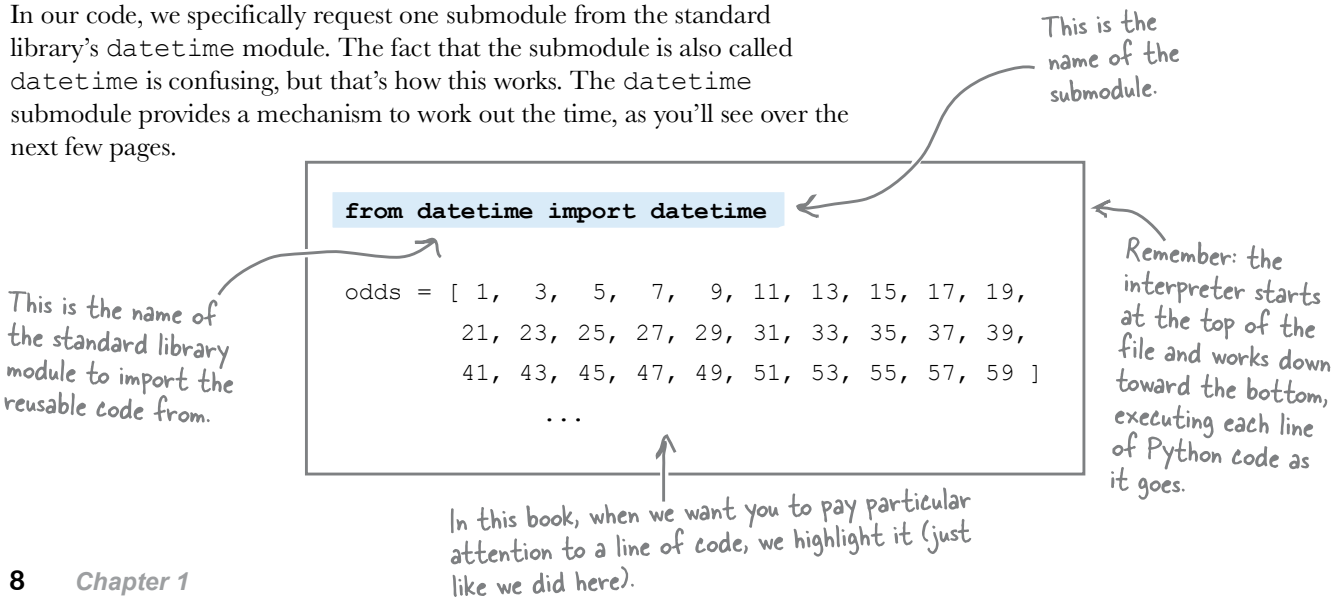
Let's be the Python interpreter

Let's take some time to run through this code in much the same way that the interpreter does, line by line, from the *top* of the file to the *bottom*.

The first line of code **imports** some preexisting functionality from Python's **standard library,** which is a large stock of software modules providing lots of prebuilt (and high-quality) reusable code.

In our code, we specifically request one submodule from the standard library's `datetime` module. The fact that the submodule is also called `datetime` is confusing, but that's how this works. The `datetime` submodule provides a mechanism to work out the time, as you'll see over the next few pages.

Think of modules as a collection of related functions.



Functions + Modules = The Standard Library

Python's **standard library** is very *rich*, and provides a lot of reusable code.

Let's look at another module, called `os`, which provides a platform-independent way to interact with your underlying operating system (we'll return to the `datetime` module in a moment). Let's concentrate on just one provided function, `getcwd`, which—when invoked—returns your *current working directory*.

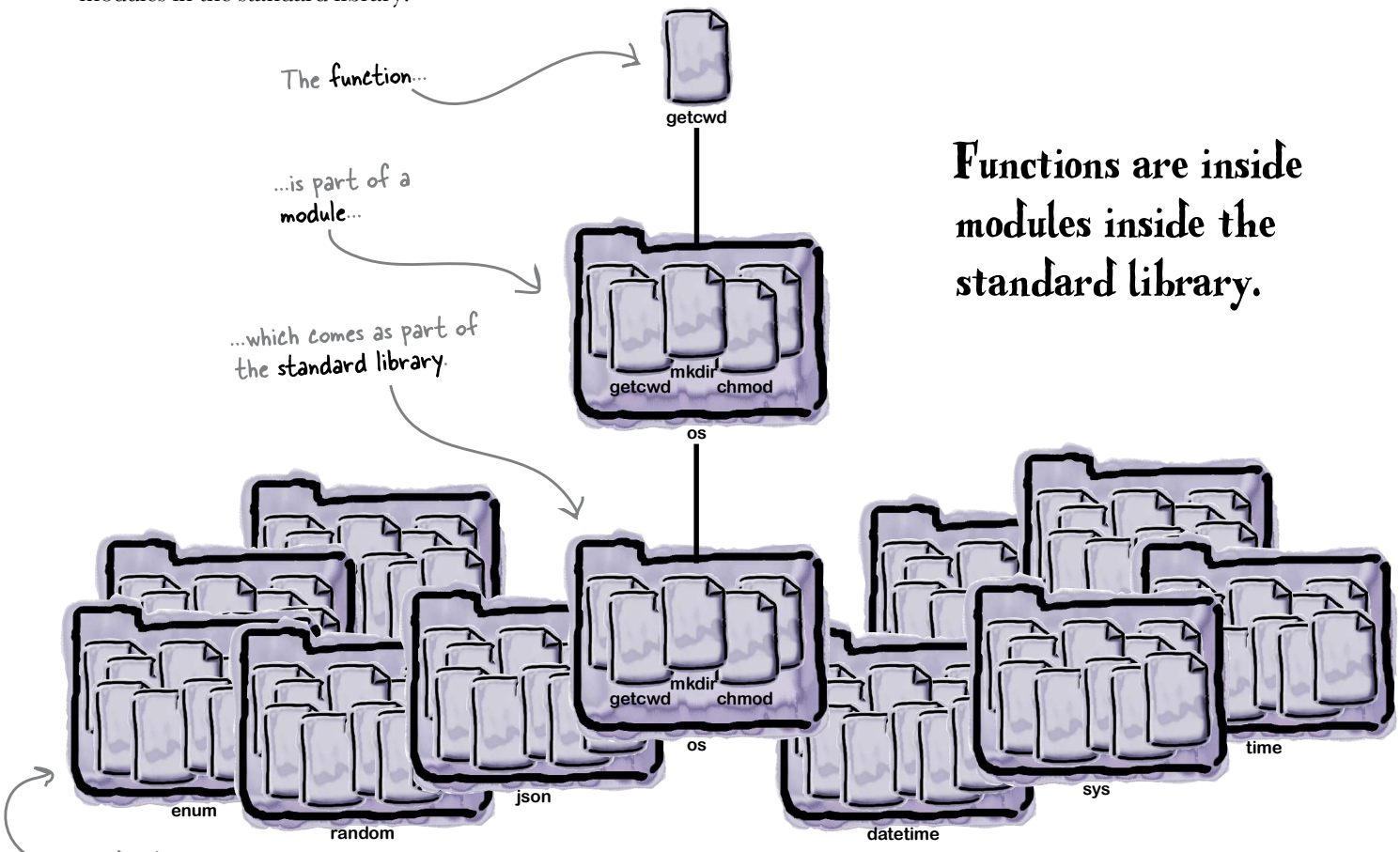
Here's how you'd typically *import*, then *invoke*, this function within a Python program:

Import the function from its module...

```
from os import getcwd

where_am_I = getcwd()
```

...then invoke as required.

A collection of related functions makes up a module, and there are *lots* of modules in the standard library:



Don't worry about what each of these modules does at this stage. We have a quick preview of some of them over the page, and will see more of the rest later in this book.

Up Close with the Standard Library

The **standard library** is the jewel in Python's crown, supplying reusable modules that help you with everything from, for example, working with data, through manipulating ZIP archives, to sending emails, to working with HTML. The standard library even includes a web server, as well as the popular *SQLite* database technology. In this *Up Close*, we'll present an overview of just a few of the most commonly used modules in the standard library. To follow along, you can enter these examples as shown at your >>> prompt (in IDLE). If you are currently looking at IDLE's edit window, choose *Run...→Python Shell* from the menu to access the >>> prompt.

Let's start by learning a little about the system your interpreter is running on. Although Python prides itself on being cross-platform, in that code written on one platform can be executed (generally unaltered) on another, there are times when it's important to know that you are running on, say, *Mac OS X*. The `sys` module exists to help you learn more about your interpreter's system. Here's how to determine the identity of your underlying operating system, by first importing the `sys` module, then accessing the `platform` attribute:

```
>>> import sys
>>> sys.platform
'darwin'
```

Import the module you need, then access the attribute of interest. It looks like we are running "darwin", which is the Mac OS X kernel name.

The `sys` module is a good example of a reusable module that primarily provides access to preset attributes (such as `platform`). As another example, here's how to determine which version of Python is running, which we pass to the `print` function to display on screen:

```
>>> print(sys.version)
3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
```

There's a lot of information about the Python version we're running, including that it's 3.4.3.

The `os` module is a good example of a reusable module that primarily yields functionality, as well as providing a system-independent way for your Python code to interact with the underlying operating system, regardless of exactly which operating system that is.

For example, here's how to work out the name of the folder your code is operating within using the `getcwd` function. As with any module, you begin by importing the module before invoking the function:

```
>>> import os
>>> os.getcwd()
'/Users/HeadFirst/CodeExamples'
```

Import the module, then invoke the functionality you need.

You can access your system's environment variables, as a whole (using the `environ` attribute) or individually (using the `getenv` function):

```
>>> os.environ
'environ({'XPC_FLAGS': '0x0', 'HOME': '/Users/HeadFirst', 'TMPDIR': '/var/
folders/18/t93gmhc546b7b2cngfhz10l00000gn/T/', ... 'PYTHONPATH': '/Applications/
Python 3.4/IDLE.app/Contents/Resources', ... 'SHELL': '/bin/bash', 'USER':
'HeadFirst'})'
>>> os.getenv('HOME')
'/Users/HeadFirst'
```

The "environ" attribute contains lots of data.

You can access a specifically named attribute (from the data contained in "environ") using "getenv".

Up Close with the Standard Library, Continued

Working with dates (and times) comes up a lot, and the standard library provides the `datetime` module to help when you're working with this type of data. The `date.today` function provides today's date:

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2015, 5, 31)
```

Today's date

That's certainly a strange way to display today's date, though, isn't it? You can access the day, month, and year values separately by appending an attribute access onto the call to `date.today`:

```
>>> datetime.date.today().day
31
>>> datetime.date.today().month
5
>>> datetime.date.today().year
2015
```

The component parts of today's date

You can also invoke the `date.isoformat` function and pass in today's date to display a much more user-friendly version of today's date, which is converted to a string by `isoformat`:

```
>>> datetime.date.isoformat(datetime.date.today())
'2015-05-31'
```

Today's date as a string

And then there's time, which none of us seem to have enough of. Can the *standard library* tell us what time it is? Yes. After importing the `time` module, call the `strftime` function and specify how you want the time displayed. In this case, we are interested in the current time's hours (`%H`) and minutes (`%M`) values in 24-hour format:

```
>>> import time
>>> time.strftime("%H:%M")
'23:55'
```

Good heavens! Is that the time?

How about working out the day of the week, and whether or not it's before noon? Using the `%A %p` specification with `strftime` does just that:

```
>>> time.strftime("%A %p")
'Sunday PM'
```

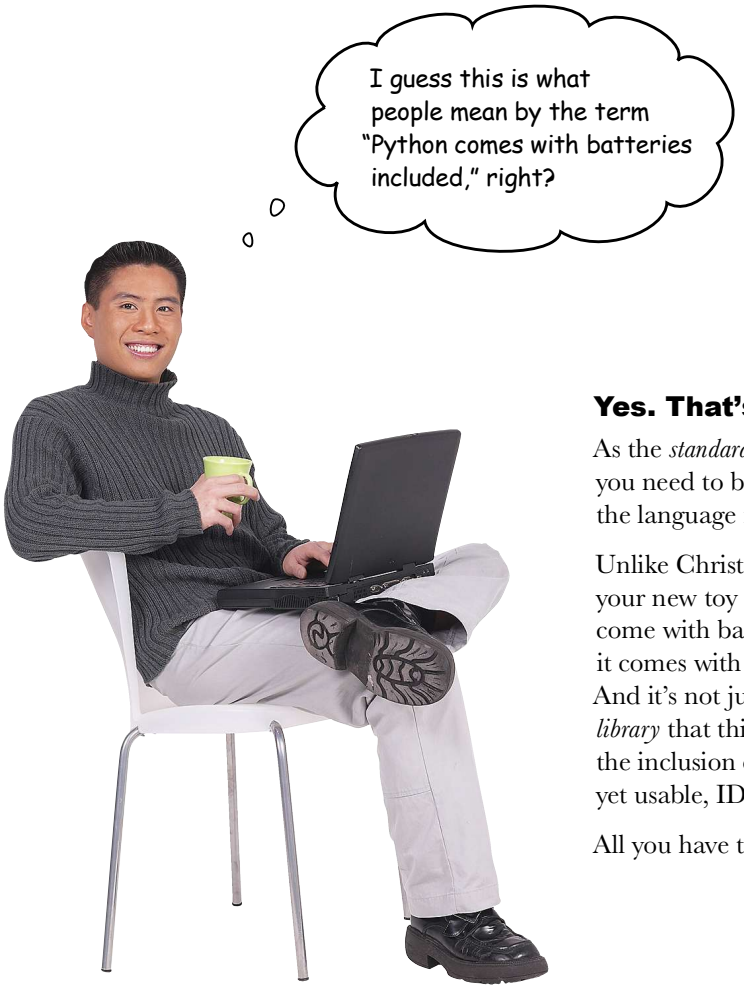
We've now worked out that it's five minutes to midnight on Sunday evening...time for bed, perhaps?

As a final example of the type of reusable functionality the *standard library* provides, imagine you have some HTML that you are worried might contain some potentially dangerous `<script>` tags. Rather than parsing the HTML to detect and remove the tags, why not encode all those troublesome angle brackets using the `escape` function from the `html` module? Or maybe you have some encoded HTML that you'd like to return to its original form? The `unescape` function can do that. Here are examples of both:

```
>>> import html
>>> html.escape("This HTML fragment contains a <script>script</script> tag.")
'This HTML fragment contains a &lt;script&gt;script&lt;/script&gt; tag.'
>>> html.unescape("I &hearts; Python's &lt;standard library&gt;.")
"I ♥ Python's <standard library>."
```

Converting to and from HTML encoded text

Batteries Included



Yes. That's what they mean.

As the *standard library* is *so* rich, the thinking is all you need to be **immediately productive** with the language is to have Python installed.

Unlike Christmas morning, when you open your new toy only to discover that it doesn't come with batteries, Python doesn't disappoint; it comes with everything you need to get going. And it's not just the modules in the *standard library* that this thinking applies to: don't forget the inclusion of IDLE, which provides a small, yet usable, IDE right out of the box.

All you have to do is code.

there are no Dumb Questions

Q: How am I supposed to work out what any particular module from the standard library does?

A: .The Python documentation has all the answers on the standard library. Here's the kicking-off point: *https://docs.python.org/3/library/index.html*.

Geek Bits

The standard library isn't the only place you'll find excellent importable modules to use with your code. The Python community also supports a thriving collection of third-party modules, some of which we'll explore later in this book. If you want a preview, check out the community-run repository: *http://pypi.python.org*.

Data Structures Come Built-in

As well as coming with a top-notch *standard library*, Python also has some powerful built-in **data structures**. One of these is the **list**, which can be thought of as a very powerful *array*. Like arrays in many other languages, lists in Python are enclosed within square brackets (`[]`).

The next three lines of code in our program (shown below) assign a *literal* list of odd numbers to a variable called `odds`. In this code, `odds` is *a list of integers*, but lists in Python can contain *any* data of *any* type, and you can even mix the types of data in a list (if that's what you're into). Note how the `odds` list extends over three lines, despite being a single statement. This is OK, as the interpreter won't decide a single statement has come to an end until it finds the closing bracket (`]`) that matches the opening one (`[`). Typically, **the end of the line marks the end of a statement in Python**, but there can be exceptions to this general rule, and multiline lists are just one of them (we'll meet the others later).

Like arrays, lists can hold data of any type.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
               ...
```

This is a new variable, called "odds", which is assigned a list of odd numbers.

This is the list of odd numbers, enclosed in square brackets. This single statement extends over three lines, which is OK.

There are lots of things that can be done with lists, but we're going to defer any further discussion until a later chapter. All you need to know now is that this list now *exists*, has been *assigned* to the `odds` variable (thanks to the use of the **assignment operator**, =), and *contains* the numbers shown.

Python variables are dynamically assigned

Before getting to the next line of code, perhaps a few words are needed about variables, especially if you are one of those programmers who might be used to predeclaring variables with type information *before* using them (as is the case in statically typed programming languages).

In Python, variables pop into existence the first time you use them, and **their type does not need to be predeclared**. Python variables take their type information from the type of the object they're assigned. In our program, the `odds` variable is assigned a list of numbers, so `odds` is a list in this case.

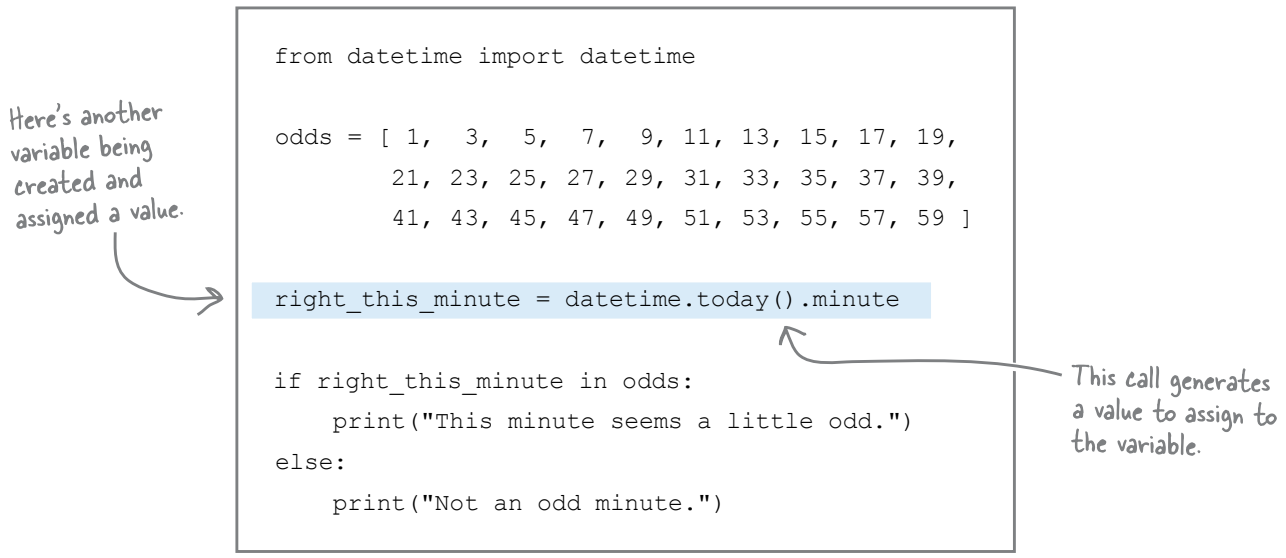
Let's look at another variable assignment statement. As luck would have it, this just so happens to also be the next line of code in our program.

Python comes with all the usual operators, including <, >, <=, >=, ==, !=, as well as the = assignment operator.

Invoking Methods Obtains Results

The third line of code in our program is another **assignment statement**.

Unlike the last one, this one doesn't assign a data structure to a variable, but instead assigns the **result** of a method call to another new variable, called `right_this_minute`. Take another look at the third line of code:

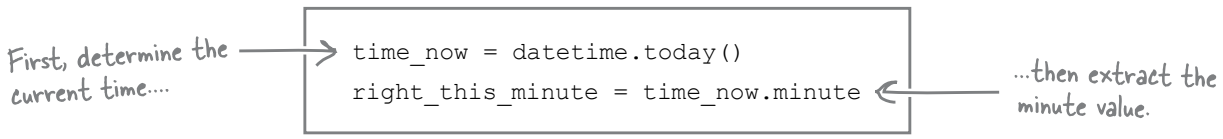


Invoking built-in module functionality

The third line of code invokes a method called `today` that comes with the `datetime` submodule, which is *itself* part of the `datetime` module (we did say this naming strategy *was* a little confusing). You can tell `today` is being invoked due to the standard postfix parentheses: `()`.

When `today` is invoked, it returns a "time object," which contains many pieces of information about the current time. These are the current time's **attributes**, which you can access via the customary **dot-notation** syntax. In this program, we are interested in the minute attribute, which we can access by appending `.minute` to the method invocation, as shown above. The resulting value is then assigned to the `right_this_minute` variable. You can think of this line of code as saying: *create an object that represents today's time, then extract the value of the minute attribute before assigning it to a variable.* It is tempting to *split* this single line of code into two lines to make it "easier to understand," as follows:

You'll see more of the dot-notation syntax later in this book.



You can do this (if you like), but most Python programmers prefer **not** to create the temporary variable (`time_now` in this example) *unless* it's needed at some point later in the program.

Deciding When to Run Blocks of Code

At this stage we have a list of numbers called `odds`. We also have a minute value called `right_this_minute`. In order to work out whether the current minute value stored in `right_this_minute` is an odd number, we need some way of determining if it is in the `odds` list. But how do we do this?

It turns out that Python makes this type of thing very straightforward. As well as including all the usual comparison operators that you'd expect to find in any programming language (such as >, <, >=, <=, and so on), Python comes with a few "super" operators of its own, one of which is `in`.

The `in` operator checks if one thing is *inside* another. Take a look at the next line of code in our program, which uses the `in` operator to check whether `right_this_minute` is *inside* the `odds` list:

```
        ...
right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
        ...
```

This "if" statement will evaluate to either "True" or "False".

The "in" operator is powerful. It can determine whether one thing is inside another.

The `in` operator returns either `True` or `False`. As you'd expect, if the value in `right_this_minute` is in `odds`, the `if` statement evaluates to `True`, and the block of code associated with the `if` statement executes.

Blocks in Python are easy to spot, as they are always indented.

In our program there are two blocks, which each contain a single call to the `print` function. This function can display messages on screen (and we'll see lots of uses of it throughout this book). When you enter this program code into the edit window, you may have noticed that IDLE helps keep you straight by indenting automatically. This is very useful, but do be sure to check that IDLE's indentation is what you want:

```
        ...
right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

Here is one block of code. Note: the code is indented.

The "print" function displays a message on standard output (i.e., your screen).

And here is another block of code. Note: it's indented, too.

Did you notice that there are no curly braces here?

What Happened to My Curly Braces?

If you are used to a programming language that uses curly braces (`{` and `}`) to delimit blocks of code, encountering blocks in Python for the first time can be disorienting, as Python doesn't use curly braces for this purpose. Python uses **indentation** to demarcate a block of code, which Python programmers prefer to call **suite** as opposed to *block* (just to mix things up a little).

It's not that curly braces don't have a use in Python. They do, but—as we'll see in Chapter 3—curly braces have more to do with delimiting data than they have to do with delimiting suites (i.e., *blocks*) of code.

Suites within any Python program are easy to spot, as they are always indented. This helps your brain quickly identify suites when reading code. The other visual clue for you to look out for is the colon character (`:`), which is used to introduce a suite that's associated with any of Python's control statements (such as `if`, `else`, `for`, and the like). You'll see lots of examples of this usage as you progress through this book.

Instead of referring to a code "block," Python programmers use the word "suite." Both names are used in practice, but the Python docs prefer "suite."

A colon introduces an indented suite of code

The colon (`:`) is important, in that it introduces a new suite of code that must be indented to the right. If you forget to indent your code after a colon, the interpreter raises an error.

Not only does the `if` statement in our example have a colon, the `else` has one, too. Here's all the code again:

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

Colons introduce indented suites.

We're nearly done. There's just one final statement to discuss.

What "else" Can You Have with "if"?

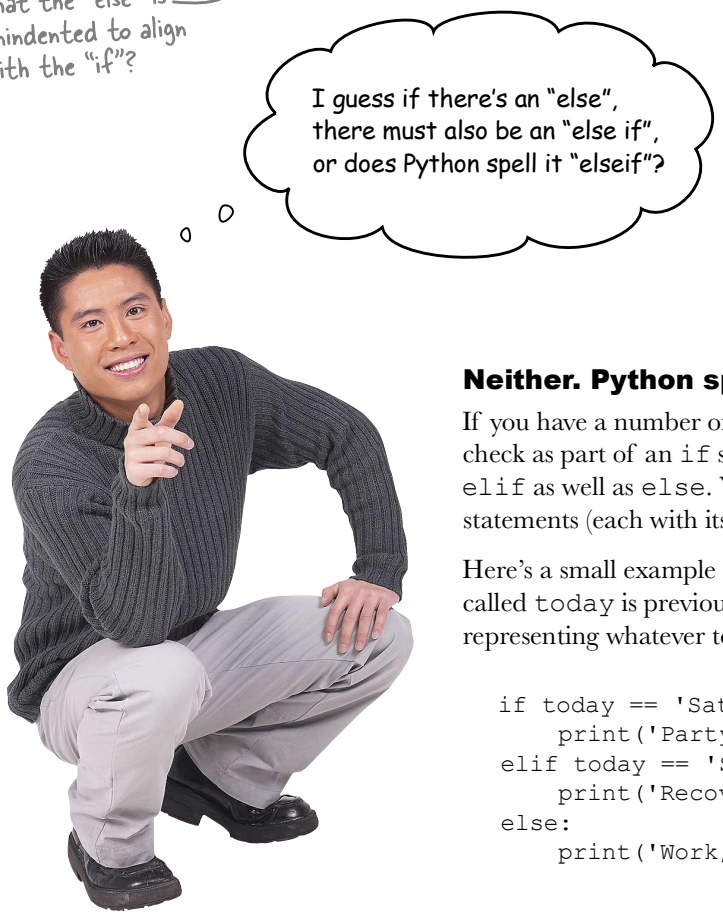
We are nearly done with the code for our example program, in that there is only one line of code left to discuss. It is not a very big line of code, but it's an important one: the `else` statement that identifies the block of code that executes when the matching `if` statement returns a `False` value.

Take a closer look at the `else` statement from our program code, which we need to unindent to align with the `if` part of this statement:

```
if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

See the colon?

Did you spot that the "else" is unindented to align with the "if"?



It is a very common slip-up for Python newbies to forget the colon when first writing code.

Neither. Python spells it `elif`.

If you have a number of conditions that you need to check as part of an `if` statement, Python provides `elif` as well as `else`. You can have as many `elif` statements (each with its own suite) as needed.

Here's a small example that assumes a variable called `today` is previously assigned a string representing whatever today is:

```
if today == 'Saturday':
    print('Party!!')
elif today == 'Sunday':
    print('Recover.')
else:
    print('Work, work, work.')
```

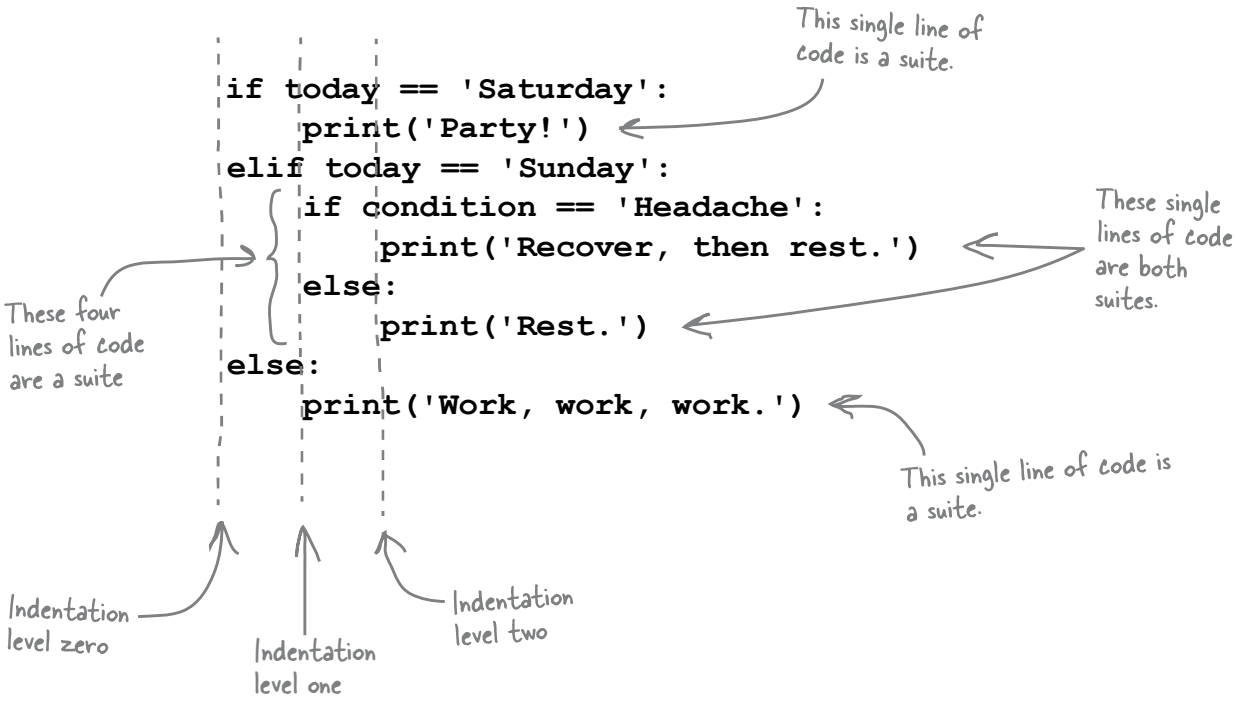
Three individual suites: one for the "if", another for the "elif", and the final catch-all for the "else".

Suites Can Contain Embedded Suites

Any suite can contain any number of embedded suites, which also have to be indented. When Python programmers talk about embedded suites, they tend to talk about **levels of indentation**.

The initial level of indentation for any program is generally referred to as the *first* or (as is so common when it comes to counting with many programming languages) indentation level *zero*. Subsequent levels are referred to as the second, third, fourth, and so on (or level one, level two, level three, and so on).

Here's a variation on the today example code from the last page. Note how an embedded `if/else` has been added to the `if` statement that executes when `today` is set to `'Sunday'`. We're also assuming another variable called `condition` exists and is set to a value that expresses how you're currently feeling. We've indicated where each of the suites is, as well as at which level of indentation it appears:



It is important to note that code at the same level of indentation is only related to other code at the same level of indentation if all the code appears *within the same suite*. Otherwise, they are in separate suites, and it does not matter that they share a level of indentation. The key point is that indentation is used to demarcate suites of code in Python.

What We Already Know

With the final few lines of code discussed, let's pause to review what the `odd.py` program has told us about Python:

BULLET POINTS

- Python comes with a built-in IDE called IDLE, which lets you create, edit, and run your Python code—all you need to do is type in your code, save it, and then press F5.
- IDLE interacts with the Python interpreter, which automates the compile-link-run process for you. This lets you concentrate on writing your code.
- The interpreter runs your code (stored in a file) from top to bottom, one line at a time. There is no notion of a `main()` function/method in Python.
- Python comes with a powerful standard library, which provides access to lots of reusable modules (of which `datetime` is just one example).
- There is a collection of standard data structures available to you when you're writing Python programs. The list is one of them, and is very similar in notion to an array.
- The type of a variable does not need to be declared. When you assign a value to a variable in Python, it dynamically takes on the type of the data it refers to.
- You make decisions with the `if`/`elif`/`else` statement. The `if`, `elif`, and `else` keywords precede blocks of code, which are known in the Python world as "suites."
- It is easy to spot suites of code, as they are always indented. Indentation is the only code grouping mechanism provided by Python.
- In addition to indentation, suites of code are also preceded by a colon (`:`). This is a syntactical requirement of the language.

Let's extend this program to do more.

It's true that we needed more lines to describe what this short program does than we actually needed to write the code. But this is one of the great strengths of Python: *you can get a lot done with a few lines of code.*

Review the list above once more, and then turn the page to make a start on seeing what our program's extensions will be.

Extending Our Program to Do More

Let's extend our program in order to learn a bit more Python.

At the moment, the program runs once, then terminates. Imagine that we want this program to execute more than once; let's say five times. Specifically, let's execute the "minute checking code" and the `if/else` statement five times, pausing for a random number of seconds between each message display (just to keep things interesting). When the program terminates, five messages should be on screen, as opposed to one.

Here's the code again, with the code we want to run multiple times circled:

Let's tweak the program to run this code a number of times.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

What we need to do:

1. **Loop over the encircled code.**
   A loop lets us iterate over any suite, and Python provides a number of ways to do just that. In this case (and without getting into why), we'll use Python's `for` loop to iterate.

2. **Pause execution.**
   Python's standard `time` module provides a function called `sleep` that can pause execution for an indicated number of seconds.

3. **Generate a random number.**
   Happily, another Python module, `random`, provides a function called `randint` that we can use to generate a random number. Let's use `randint` to generate a number between 1 and 60, then use that number to pause the execution of our program on each iteration.

We now know what we want to do. But is there a preferred way of going about making these changes?

What's the Best Approach to Solving This Problem?

Both approaches work with Python

You can follow *both* of these approaches when working with Python, but most Python programmers favor **experimentation** when trying to work out what code they need for a particular situation.

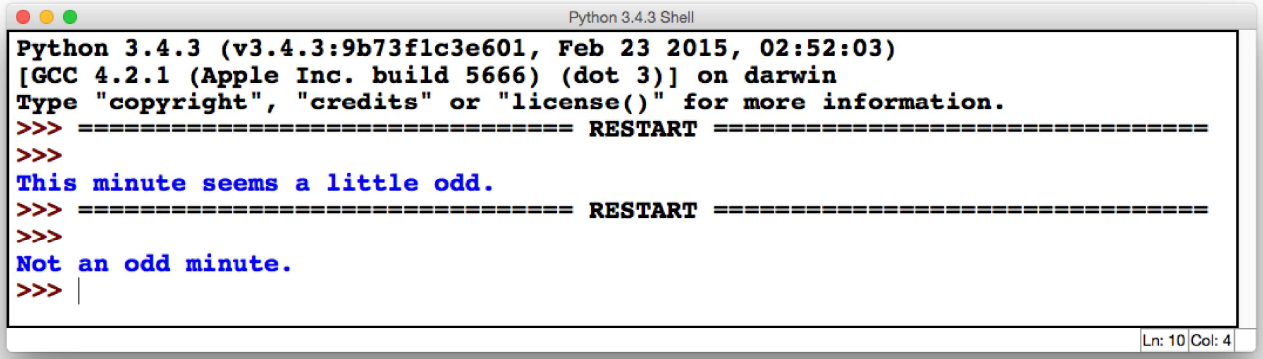
Don't get us wrong: we are not suggesting that Bob's approach is wrong and Laura's is right. It's just that Python programmers have both options available to them, and the Python Shell (which we met briefly at the start of this chapter) makes experimentation a natural choice for Python programmers.

Let's determine the code we need in order to extend our program, by experimenting at the >>> prompt.

Experimenting at the >>> prompt helps you work out the code you need.

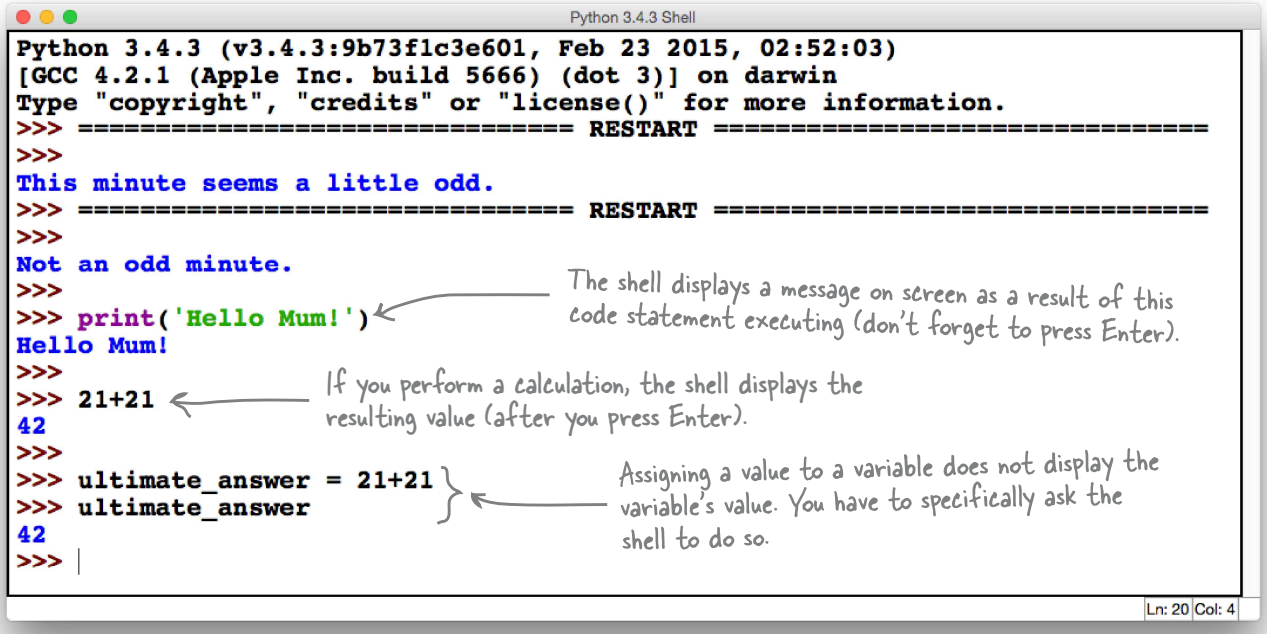
Returning to the Python Shell

Here's how the Python Shell looked the last time we interacted with it (yours might look a little different, as your messages may have appeared in an alternate order):



The Python Shell (or just "shell" for short) has displayed our program's messages, but it can do so much more than this. The >>> prompt allows you to enter any Python code statement and have it execute *immediately*. If the statement produces output, the shell displays it. If the statement results in a value, the shell displays the calculated value. If, however, you create a new variable and assign it a value, you need to enter the variable's name at the >>> prompt to see what value it contains.

Check out the example interactions, shown below. It is even better if you follow along and try out these examples at *your* shell. Just be sure to press the *Enter* key to terminate each program statement, which also tells the shell to execute it *now*:



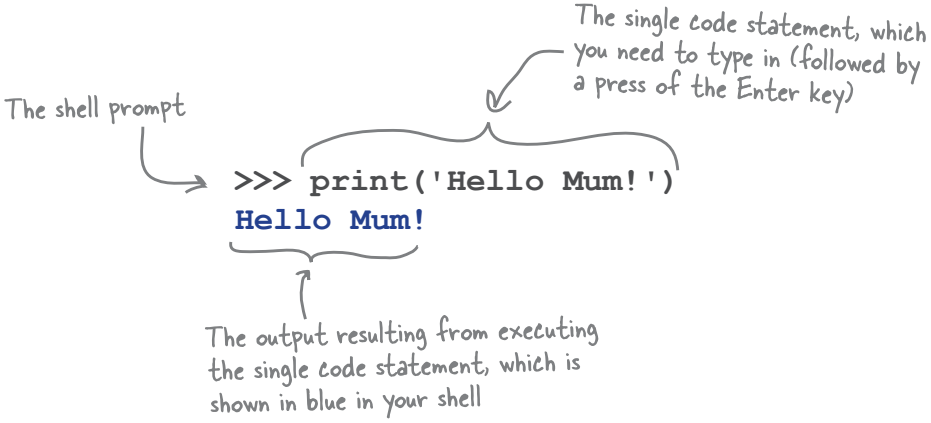
Experimenting at the Shell

Now that you know you can type a single Python statement into the >>> prompt and have it execute immediately, you can start to work out the code you need to extend your program.

Here's what you need your new code to do:

- ☐ **Loop** a specified number of times. We've already decided to use Python's `for` loop here.
- ☐ **Pause** the program for a specified number of seconds. The `sleep` function from the standard library's `time` module can do this.
- ☐ **Generate** a random number between two provided values. The `randint` function from the `random` module will do the trick.

Rather than continuing to show you complete IDLE screenshots, we're only going to show you the >>> prompt and any displayed output. Specifically, from this point onward, you'll see something like the following instead of the earlier screenshots:



Over the next few pages, we're going to experiment to figure out how to add the three features listed above. We'll *play* with code at the >>> prompt until we determine exactly the statements we need to add to our program. Leave the `odd.py` code as is for now, then make sure the shell window is active by selecting it. The cursor should be blinking away to the right of the >>> , waiting for you to type some code.

Flip the page when you're ready. Let the experiments begin.

Iterating Over a Sequence of Objects

We said earlier that we were going to employ Python's `for` loop here. The `for` loop is *perfect* for controlling looping when you know ahead of time how many iterations you need. (When you don't know, we recommend the `while` loop, but we'll save discussing the details of this alternate looping construct until we actually need it). At this stage, all we need is `for`, so let's see it in action at the >>> prompt.

Use "for" when looping a known number of times.

We present three typical uses of `for`. Let's see which one best fits our needs.

Usage example 1. This `for` loop, below, takes a list of numbers and iterates once for each number in the list, displaying the current number on screen. As it does so, the `for` loop assigns each number in turn to a *loop iteration variable*, which is given the name `i` in this code.

As this code is more than a single line, the shell indents automatically for you when you press Enter after the colon. To signal to the shell that you are done entering code, press Enter *twice* at the end of the loop's suite:

```
>>> for i in [1, 2, 3]:
        print(i)

1
2
3
```

We used "i" as the loop iteration variable in this example, but we could've called it just about anything. Having said that, "i", "j", and "k" are incredibly popular among most programmers in this situation.

As this is a suite, you need to press the Enter key TWICE after typing in this code in order to terminate the statement and see it execute.

Note the *indentation* and *colon*. Like `if` statements, the code associated with a `for` statement needs to be **indented**.

Usage example 2. This `for` loop, below, iterates over a string, with each character in the string being processed during each iteration. This works because a string in Python is a **sequence**. A sequence is an ordered collection of objects (and we'll see lots of examples of sequences in this book), and every sequence in Python can be iterated over by the interpreter.

A sequence is an ordered collection of objects.

```
>>> for ch in "Hi!":
        print(ch)

H
i
!
```

Python is smart enough to work out that this string should be iterated over one-character at a time (and that's why we used "ch" as the loop variable name here).

Nowhere did you have to tell the `for` loop *how big the string is*. Python is smart enough to work out when the string *ends*, and arranges to terminate (i.e., end) the `for` loop on your behalf when it exhausts all the objects in the sequence.

Iterating a Specific Number of Times

In addition to using `for` to iterate over a sequence, you can be more exact and specify a number of iterations, thanks to the built-in function called `range`.

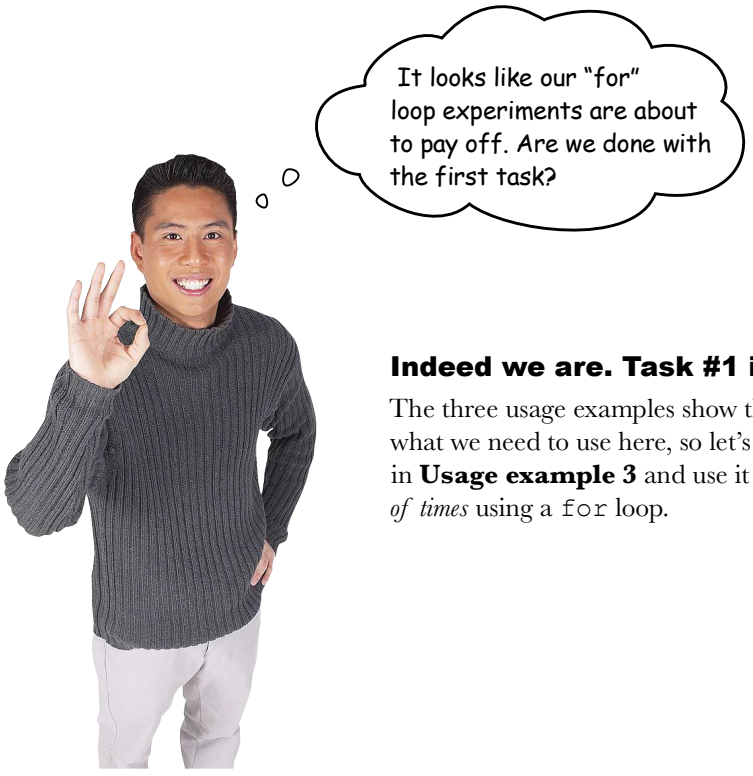
Let's look at another usage example that showcases using `range`.

Usage example 3. In its most basic form, `range` accepts a single integer argument that dictates how many times the `for` loop runs (we'll see other uses of `range` later in this book). In this loop, we use `range` to generate a list of numbers that are assigned one at a time to the `num` variable:

```
>>> for num in range(5):
        print('Head First Rocks!')

Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
```

We asked for a range of five numbers, so we iterated five times, which results in five messages. Remember: press Enter twice to run code that has a suite.

The `for` loop *didn't use* the `num` loop iteration variable *anywhere* in the loop's suite. This did not raise an error, which is OK, as it is up to you (the programmer) to decide whether or not `num` needs to be processed further in the suite. In this case, doing nothing with `num` is fine.

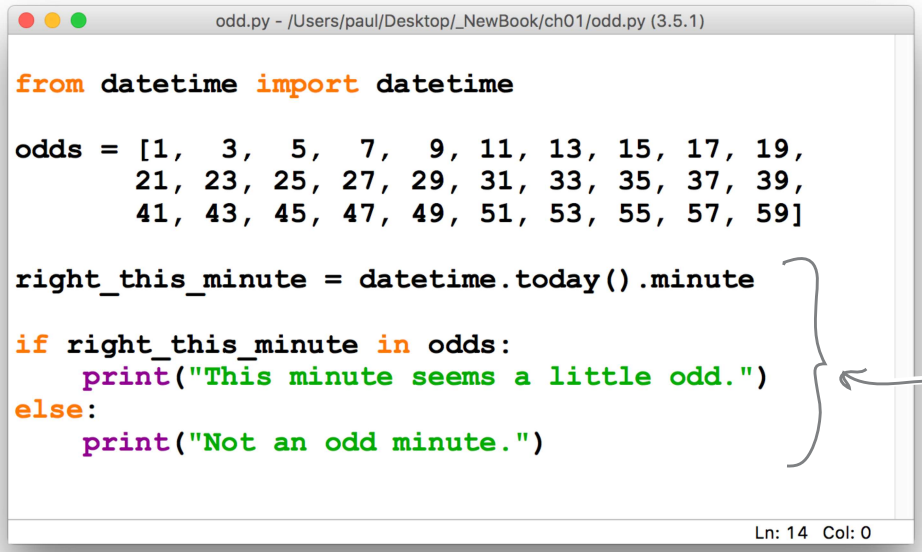


Indeed we are. Task #1 is complete.

The three usage examples show that Python's `for` loop is what we need to use here, so let's take the technique shown in **Usage example 3** and use it to iterate a *specified number of times* using a `for` loop.

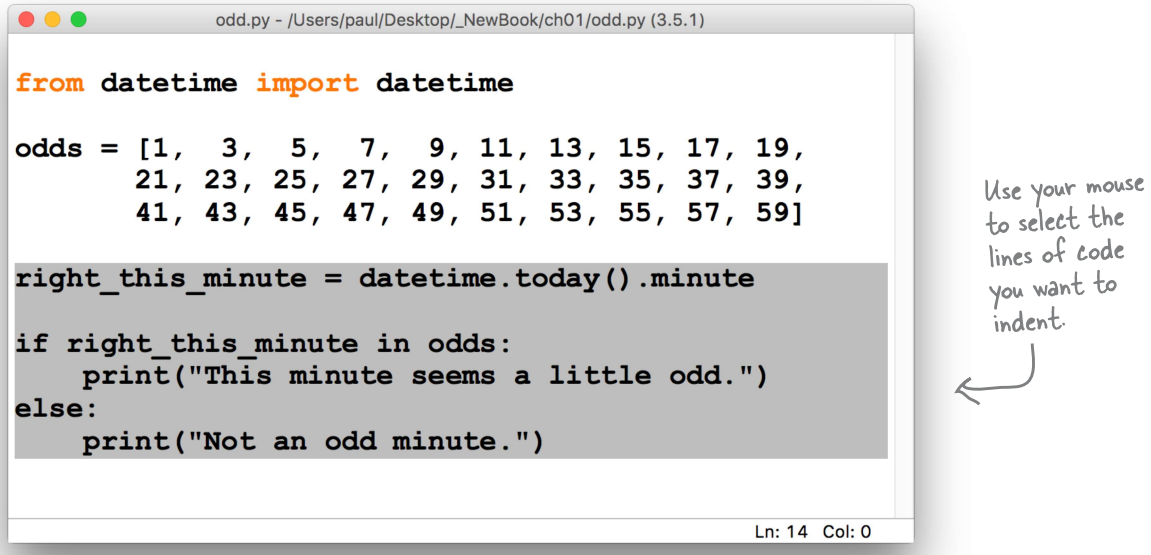
Applying the Outcome of Task #1 to Our Code

Here's how our code looked in IDLE's edit window *before* we worked on Task #1:



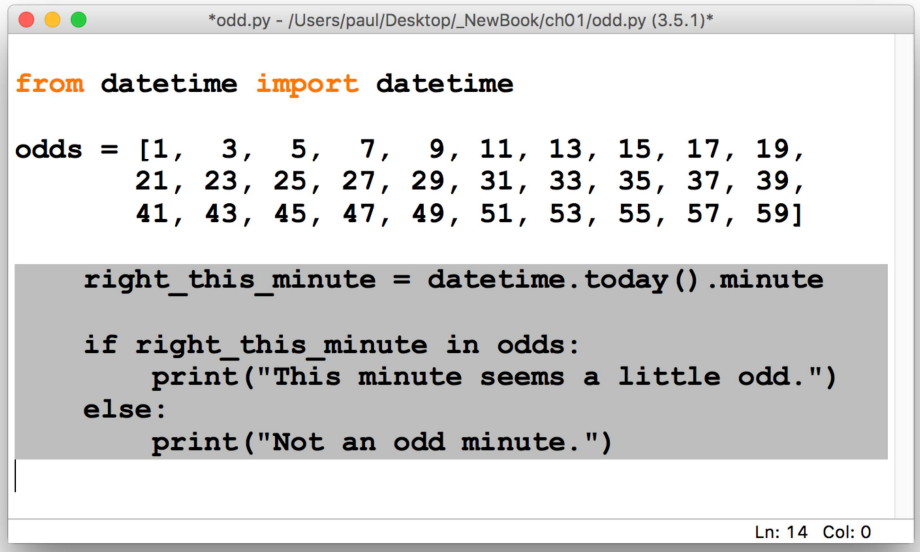
You now know that you can use a `for` loop to repeat the five lines of code at the bottom of this program five times. The five lines will need to be **indented** under the `for` loop, as they are going to form the loop's suite. Specifically, each line of code needs to be indented *once*. However, don't be tempted to perform this action on each individual line. Instead, let IDLE indent the entire suite for you *in one go*.

Begin by using your mouse to select the lines of code you want to indent:



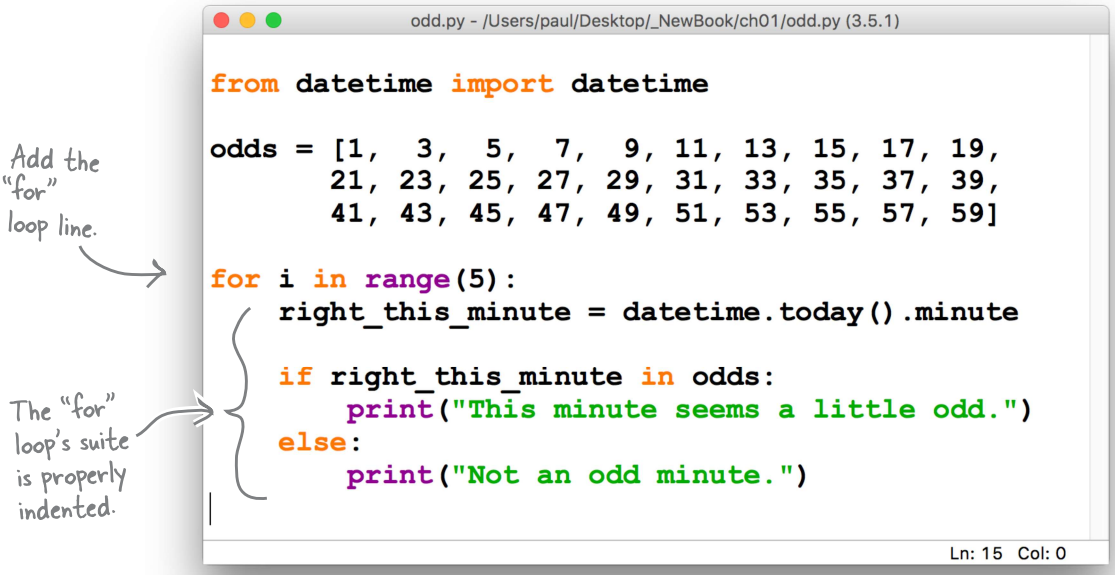
Indent Suites with Format...Indent Region

With the five lines of code selected, choose *Indent Region* from the *Format* menu in IDLE's edit window. The entire suite moves to the right by one indentation level:



The Indent Region option from the Format menu indents all of the selected lines of code in one go.

Note that IDLE also has a *Dedent Region* menu option, which unindents suites, and that both the *Indent* and *Dedent* menu commands have keyboard shortcuts, which differ slightly based on the operating system you are running. Take the time to learn the keyboard shortcuts that your system uses *now* (as you'll use them all the time). With the suite indented, it's time to add the `for` loop:



Add the "for" loop line.

The "for" loop's suite is properly indented.

Arranging to Pause Execution

Let's remind ourselves of what we need this code to do:

- [x] **Loop** a specified number of times.
- [ ] **Pause** the program for a specified number of seconds.
- [ ] **Generate** a random number between two provided values.

We're now ready to return to the shell and try out some more code to help with the second task: *pause the program for a specified number of seconds*.

However, before we do that, recall the opening line of our program, which imported a specifically named function from a specifically named module:

```
from datetime import datetime
```

This usage of "import" brings in the named function to your program. You can then invoke it without using the dot-notation syntax.

This is one way to import a function into your program. Another equally common technique is to import a module *without* being specific about the function you want to use. Let's use this second technique here, as it will appear in many Python programs you'll come across.

As mentioned earlier in this chapter, the `sleep` function can pause execution for a specified number of seconds, and is provided by the standard library's `time` module. Let's **import** the module *first*, without mentioning `sleep` just yet:

```
>>> import time
>>>
```

This tells the shell to import the "time" module.

When the `import` statement is used as it is with the `time` module above, you get access to the facilities provided by the module without anything expressly *named* being imported into your program's code. To access a function provided by a module imported in this way, use the dot-notation syntax to name it, as shown here:

This is the number of seconds to sleep for.

Name the module first (before the period).

```
>>> time.sleep(5)
>>>
```

Specify the function you want to invoke (after the period).

Note that when you invoke `sleep` in this way, the shell pauses for five seconds before the >>> prompt reappears. Go ahead, and *try it now*.

Importation Confusion

Hang on a second...Python supports two importation mechanisms? Doesn't that get kind of confusing?

That's a great question.

Just to be clear, there aren't *two* importation mechanisms in Python, as there is only *one* `import` statement. However, the `import` statement can be used *in two ways*.

The first, which we initially saw in our example program, imports a named function into our program's **namespace**, which then allows us to invoke the function as necessary without having to *link* the function back to the imported module. (The notion of a namespace is important in Python, as it defines the context within which your code runs. That said, we're going to wait until a later chapter to explore namespaces in detail).

In our example program, we use the first importation technique, then invoke the `datetime` function as `datetime()`, *not* as `datetime.datetime()`.

The second way to use `import` is to just import the module, as we did when experimenting with the `time` module. When we import this way, we have to use the dot-notation syntax to access the module's functionality, as we did with `time.sleep()`.

there are no Dumb Questions

Q: Is there a correct way to use `import`?

A: It can often come down to personal preference, as some programmers like to be very specific, while others don't. However, there is a situation that occurs when two modules (we'll call them `A` and `B`) have a function of the same name, which we'll call `F`. If you put `from A import F` and `from B import F` in your code, how is Python to know which `F` to invoke when you call `F()`? The only way you can be sure is to use the nonspecific `import` statement (that is, put `import A` and `import B` in your code), then invoke the specific `F` you want using either `A.F()` or `B.F()` as needed. Doing so negates any confusion.

Generating Random Integers with Python

Although it is tempting to add `import time` to the top of our program, then call `time.sleep(5)` in the `for` loop's suite, we aren't going to do this right now. We aren't done with our experimentations. Pausing for five seconds isn't enough; we need to be able to pause for a *random amount of time*. With that in mind, let's remind ourselves of what we've done, and what remains:

- [x] **Loop** a specified number of times.
- [x] **Pause** the program for a specified number of seconds.
- [ ] **Generate** a random number between two provided values.

Once we have this last task completed, we can get back to confidently changing our program to incorporate all that we've learned from our experimentations. But we're not there yet—let's look at the last task, which is to generate a random number.

As with sleeping, the *standard library* can help here, as it includes a module called `random`. With just this piece of information to guide us, let's experiment at the shell:

```
>>> import random
>>>
```

Use "dir" to query an object.

Now what? We could look at the Python docs or consult a Python reference book...but that involves taking our attention away from the shell, even though it might only take a few moments. As it happens, the shell provides some additional functions that can help here. These functions aren't meant to be used within your program code; they are designed for use at the >>> prompt. The first is called `dir`, and it displays all the **attributes** associated with anything in Python, including modules:

```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF',
'Random',    ...    'randint', 'random', 'randrange',
'sample', 'seed', 'setstate', 'shuffle', 'triangular',
'uniform', 'vonmisesvariate', 'weibullvariate']
```

Buried in the middle of this long list is the name of the function we need.

This is an abridged list. What you'll see on your screen is much longer.

This list has a lot in it. Of interest is the `randint()` function. To learn more about `randint`, let's ask the shell for some **help**.

Asking the Interpreter for Help

Once you know the name of something, you can ask the shell for **help**. When you do, the shell displays the section from the Python docs related to the name you're interested in.

Let's see this mechanism in action at the >>> prompt by asking for **help** with the `randint` function from the `random` module:

Use "help" to read the Python docs.

Ask for help at the >>> prompt...

```
>>> help(random.randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including
    both end points.
```

...and see the associated documentation right in the shell.

A quick read of the displayed docs for the `randint` function confirms what we need to know: if we provide two integers to `randint`, we get back a random integer from the resulting inclusive range.

A few final experiments at the >>> prompt show the `randint` function in action:

```
>>> random.randint(1,60)
27
>>> random.randint(1,60)
34
>>> random.randint(1,60)
46
```

If you're following along, what you'll see on your screen will vary, as the integers returned by "randint" are generated randomly.

Because you imported the "random" module using "import random", you need to remember to prefix the call to "randint" with the module name and a dot. So it's "random.randint()" and not "randint()".

Geek Bits

You can recall the last command(s) typed into the IDLE >>> prompt by typing Alt-P when using *Linux* or *Windows*. On *Mac OS X*, use Ctrl-P. Think of the "P" as meaning "previous."

With this, you are now in a position to place a satisfying check mark against the last of our tasks, as you now know enough to generate a random number between two provided values:

Generate a random number between two provided values.

It's time to return to our program and make our changes.

Reviewing Our Experiments

Before you forge ahead and change your program, let's quickly review the outcome of our shell experiments.

We started by writing a `for` loop, which iterated five times:

```
>>> for num in range(5):
        print('Head First Rocks!')

Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
```

We asked for a range of five numbers, so we iterated five times, which results in five messages.

Then we used the `sleep` function from the `time` module to pause execution of our code for a specified number of seconds:

```
>>> import time
>>> time.sleep(5)
```

The shell imports the "time" module, letting us invoke the "sleep" function.

And then we experimented with the `randint` function (from the `random` module) to generate a random integer from a provided range:

```
>>> import random
>>> random.randint(1,60)
12
>>> random.randint(1,60)
42
>>> random.randint(1,60)
17
```

Note: different integers are generated once more, as "randint" returns a different random integer each time it's invoked.

We can now put all of this together and change our program.

Let's remind ourselves of what we decided to do earlier in this chapter: have our program iterate, executing the "minute checking code" and the `if/else` statement five times, and pausing for a random number of seconds between each iteration. This should result in five messages appearing on screen before the program terminates.

Code Experiments Magnets

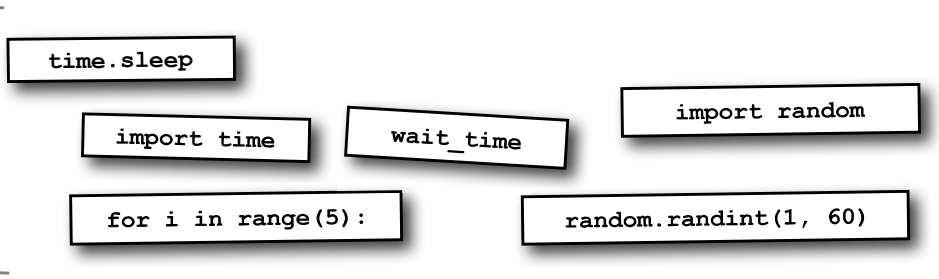
Based on the specification at the bottom of the last page, as well as the results of our experimentations, we went ahead and did some of the required work for you. But, as we were arranging our code magnets on the fridge (don't ask) someone slammed the door, and now some of our code's all over the floor.

Your job is to put everything back together, so that we can run the new version of our program and confirm that it's working as required.

Decide which code magnet goes in each of the dashed-line locations.

```
from datetime import datetime

.....
.....

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]


.....
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = .....
    ..... ( ..... )
```

Where do all these go?



Code Experiments Magnets Solution

Based on the specification from earlier, as well as the results of our experimentations, we went ahead and did some of the required work for you. But, as we were arranging our code magnets on the fridge (don't ask) someone slammed the door, and now some of our code's all over the floor.

Your job was to put everything back together, so that we could run the new version of our program and confirm that it's working as required.

You don't have to put your imports at the top of your code, but it is a well-established convention among Python programmers to do so.

```
from datetime import datetime
import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
```

The "for" loop iterates EXACTLY five times.

```
for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = random.randint(1, 60)
    time.sleep( wait_time )
```

All of this code is indented under the "for" statement, as it is all part of the "for" statement's suite. Remember: Python does not use curly braces to delimit suites; it uses indentation instead.

The "randint" function provides a random integer that is assigned to a new variable called "wait_time", which...

...is then used in the call to "sleep" to pause the program's execution for a random number of seconds.

Test Drive

Let's try running our upgraded program in IDLE to see what happens. Change your version of `odd.py` as needed, then save a copy of your new program as `odd2.py`. When you're ready, press F5 to execute your code.

When you press F5 to run this code...

odd2.py - /Users/Paul/Desktop/_NewBook/ch01/odd2.py (3.4.3)

```
from datetime import datetime

import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = random.randint(1, 60)
    time.sleep(wait_time)
```

Ln: 19 Col: 0

...you should see output similar to this. Just remember that your output will differ, as the random numbers your program generates most likely won't match ours.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
This minute seems a little odd.
This minute seems a little odd.
Not an odd minute.
Not an odd minute.
Not an odd minute.
>>>
```

Ln: 25 Col: 4

Don't worry if you see a different list of messages than those shown here. You should see five messages, as that's how many times the loop code runs.

Updating What We Already Know

With `odd2.py` working, let's pause once more to review the new things we're learned about Python from these last 15 pages:

BULLET POINTS

- When trying to determine the code that they need to solve a particular problem, Python programmers often favor experimenting with code snippets at the shell.
- If you're looking at the >>> prompt, you're at the Python Shell. Go ahead: type in a single Python statement and see what happens when it runs.
- The shell takes your line of code and sends it to the interpreter, which then executes it. Any results are returned to the shell and are then displayed on screen.
- The `for` loop can be used to iterate a fixed number of times. If you know ahead of time how many times you need to loop, use `for`.
- When you don't know ahead of time how often you're going to iterate, use Python's `while` loop (which we have yet to see, but—don't worry—we will see it in action later).
- The `for` loop can iterate over any sequence (like a list or a string), as well as execute a fixed number of times (thanks to the `range` function).
- If you need to pause the execution of your program for a specified number of seconds, use the `sleep` function provided by the standard library's `time` module.
- You can import a specific function from a module. For example, `from time import sleep` imports the `sleep` function, letting you invoke it without qualification.
- If you simply import a module—for example, `import time`—you then need to qualify the usage of any of the module's functions with the module name, like so: `time.sleep()`.
- The `random` module has a very useful function called `randint` that generates a random integer within a specified range.
- The shell provides two interactive functions that work at the >>> prompt. The `dir` function lists an object's attributes, whereas `help` provides access to the Python docs.

there are no Dumb Questions

Q: Do I have to remember all this stuff?

A: No, and don't freak out if your brain is resisting the insertion of everything seen so far. This is only the first chapter, and we've designed it to be a quick introduction to the world of Python programming. If you're getting the gist of what's going on with this code, then you're doing fine.

A Few Lines of Code Do a Lot

It is, but we are on a roll here.

It's true we've only touched on a small amount of the Python language so far. But what we've looked at has been very useful.

What we've seen so far helps to demonstrate one of Python's big selling points: *a few lines of code do a lot*. Another of the language's claims to fame is this: *Python code is easy to read.*

In an attempt to prove just how easy, we present on the next page a completely different program that you already know enough about Python to understand.

Who's in the mood for a nice, cold beer?

Coding a Serious Business Application

With a tip of the hat to *Head First Java*, let's take a look at the Python version of that classic's first serious application: the beer song.

Shown below is a screenshot of the Python version of the beer song code. Other than a slight variation on the usage of the `range` function (which we'll discuss in a bit), most of this code should make sense. The IDLE edit window contains the code, while the tail end of the program's output appears in a shell window:

beersong.py - /Users/Paul/Desktop/_NewBook/ch01/beersong.py (3.4.3)

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on the wall.")
    print(beer_num, word, "of beer.")
    print("Take one down.")
    print("Pass it around.")
    if beer_num == 1:
        print("No more bottles of beer on the wall.")
    else:
        new_num = beer_num - 1
        if new_num == 1:
            word = "bottle"
        print(new_num, word, "of beer on the wall.")
    print()
```

Running this code produces this output in the shell.

Python 3.4.3 Shell

```
3 bottles of beer on the wall.
3 bottles of beer.
Take one down.
Pass it around.
2 bottles of beer on the wall.

2 bottles of beer on the wall.
2 bottles of beer.
Take one down.
Pass it around.
1 bottle of beer on the wall.

1 bottle of beer on the wall.
1 bottle of beer.
Take one down.
Pass it around.
No more bottles of beer on the wall.

>>>
```

Ln: 660 Col: 12

Dealing with all that beer...

With the code shown above typed into an IDLE edit window and saved, pressing F5 produces a lot of output in the shell. We've only shown a little bit of the resulting output in the window on the right, as the beer song starts with 99 bottles of beer on the wall and counts down until there's no more beer. In fact, the only real twist in this code is how it handles this "counting down," so let's take a look at how that works before looking at the program's code in detail.

Python Code Is Easy to Read

There isn't one!

When most programmers new to Python first encounter code like that of the beer song, they assume that something's got to give somewhere else.

There has to be a catch, doesn't there?

No, there doesn't. It's not by accident that Python code is easy to read: the language was designed with that specific goal in mind. *Guido van Rossum*, the language's creator, wanted to create a powerful programming tool that produced code that was easy to maintain, which meant code created in Python has to be easy to read, too.

Is Indentation Driving You Crazy?

Hang on a second. All this indentation is driving me crazy. Surely that's the catch?

Indentation takes time to get used to.

Don't worry. Everyone coming to Python from a "curly-braced language" struggles with indentation *at first*. But it does get better. After a day or two of working with Python, you'll hardly notice you're indenting your suites.

One problem that some programmers do have with indentation occurs when they mix *tabs* with *spaces*. Due to the way the interpreter counts **whitespace**, this can lead to problems, in that the code "looks fine" but refuses to run. This is frustrating when you're starting out with Python.

Our advice: *don't mix tabs with spaces in your Python code.*

In fact, we'd go even further and advise you to configure your editor to replace a tap of the *Tab* key with *four spaces* (and while you're at it, automatically remove any trailing whitespace, too). This is the well-established convention among many Python programmers, and you should follow it, too. We'll have more to say about dealing with indentation at the end of this chapter.

Getting back to the beer song code

If you take a look at the invocation of `range` in the beer song, you'll notice that it takes *three* arguments as opposed to just one (as in our first example program).

Take a closer look, and without looking at the explanation on the next page, see if you can work out what's going on with this call to `range`:

beersong.py - /Users/Paul/Desktop/_NewBook/ch0...

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on
    print(beer_num, word, "of beer.")
    print("Take one down.")
```

This is new: the call to "range" takes three arguments, not one.

Asking the Interpreter for Help on a Function

Recall that you can use the shell to ask for **help** with anything to do with Python, so let's ask for some help with the `range` function.

When you do this in IDLE, the resulting documentation is more than a screen's worth and it quickly scrolls off the screen. All you need to do is scroll back in the window to where you asked the shell for help (as that's where the interesting stuff about `range` is):

```
>>> help(range)
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return a sequence of numbers from start to stop by step.
           ...
```

The "range" function can be invoked in one of two ways.

This looks like it will give us what we need here.

Starting, stopping, and stepping

As `range` is not the only place you'll come across **start**, **stop**, and **step**, let's take a moment to describe what each of these means, before looking at some representative examples (on the next page):

1. **The START value lets you control from WHERE the range begins.**
So far, we've used the single-argument version of `range`, which—from the documentation—expects a value for **stop** to be provided. When no other value is provided, `range` defaults to using `0` as the **start** value, but you can set it to a value of your choosing. When you do, you *must* provide a value for **stop**. In this way, `range` becomes a multi-argument invocation.

2. **The STOP value lets you control WHEN the range ends.**
We've already seen this in use when we invoked `range(5)` in our code. Note that the range that's generated *never* contains the **stop** value, so it's a case of up-to-but-not-including **stop**.

3. **The STEP value lets you control HOW the range is generated.**
When specifying **start** and **stop** values, you can also (optionally) specify a value for **step**. By default, the **step** value is `1`, and this tells `range` to generate each value with a *stride* of `1`; that is, 0, 1, 2, 3, 4, and so on. You can set **step** to any value to adjust the stride taken. You can also set **step** to a negative value to adjust the *direction* of the generated range.

Experimenting with Ranges

Now that you know a little bit about **start**, **stop**, and **step**, let's experiment at the shell to learn how we can use the `range` function to produce many different ranges of integers.

To help see what's going on, we use another function, `list`, to transform `range`'s output into a human-readable list that we can see on screen:

```
>>> range(5)
range(0, 5)

>>> list(range(5))
[0, 1, 2, 3, 4]

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]

>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]

>>> list(range(10, 0, 2))
[]

>>> list(range(99, 0, -1))
[99, 98, 97, 96, 95, 94, 93, 92,  ...  5, 4, 3, 2, 1]
```

This is how we used "range" in our first program.

Feeding the output from "range" to "list" produces a list.

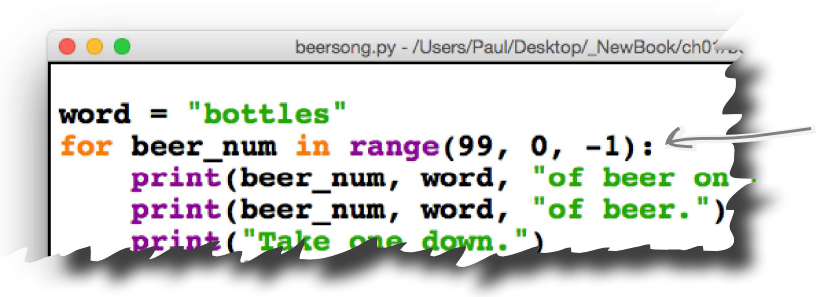
We can adjust the START and STOP values for "range".

It is also possible to adjust the STEP value.

Things get really interesting when you adjust the range's direction by negating the STEP value.

Python won't stop you from being silly. If your START value is bigger than your STOP value, and STEP is positive, you get back nothing (in this case, an empty list).

After all of our experimentations, we arrive at a `range` invocation (shown last, above) that produces a list of values from 99 down to 1, which is exactly what the beer song's `for` loop does:



The call to "range" takes three arguments: start, stop, and step.

Sharpen your pencil

Here again is the beer code, which has been spread out over the entire page so that you can **concentrate** on each line of code that makes up this "serious business application."

Grab your pencil and, in the spaces provided, write in what you thought each line of code does. Be sure to attempt this yourself *before* looking at what we came up with on the next page. We've got you started by doing the first line of code for you.

```
word = "bottles"

for beer_num in range(99, 0, -1):

    print(beer_num, word, "of beer on the wall.")

    print(beer_num, word, "of beer.")

    print("Take one down.")

    print("Pass it around.")

    if beer_num == 1:

        print("No more bottles of beer on the wall.")

    else:

        new_num = beer_num - 1

        if new_num == 1:

            word = "bottle"

        print(new_num, word, "of beer on the wall.")

    print()
```

Assign the value "bottles" (a string) to a new variable called "word".

Sharpen your pencil Solution

Here again is the beer code, which has been spread out over the entire page so that you can **concentrate** on each line of code that makes up this "serious business application."

You were to grab your pencil and then, in the spaces provided, write in what you thought each line of code does. We did the first line of code for you to get you started.

How did you get on? Are your explanations similar to ours?

```
word = "bottles"

for beer_num in range(99, 0, -1):

    print(beer_num, word, "of beer on the wall.")

    print(beer_num, word, "of beer.")

    print("Take one down.")

    print("Pass it around.")

    if beer_num == 1:

        print("No more bottles of beer on the wall.")

    else:

        new_num = beer_num - 1

        if new_num == 1:

            word = "bottle"

        print(new_num, word, "of beer on the wall.")

    print()
```

Assign the value "bottles" (a string) to a new variable called "word".

Loop a specified number of times, from 99 down to none. Use "beer_num" as the loop iteration variable.

The four calls to the print function display the current iteration's song lyrics, "99 bottles of beer on the wall. 99 bottles of beer. Take one down. Pass it around.", and so on with each iteration.

Check to see if we are on the last passed-around beer...

And if we are, end the song lyrics.

Otherwise...

Remember the number of the next beer in another variable called "new_num".

If we're about to drink our last beer...

Change the value of the "word" variable so the last lines of the lyric make sense.

Complete this iteration's song lyrics.

At the end of this iteration, print a blank line. When all the iterations are complete, terminate the program.

Don't Forget to Try the Beer Song Code

If you haven't done so already, type the beer song code into IDLE, save it as `beersong.py`, and then press F5 to take it for a spin. *Do not move on to the next chapter until you have a working beer song.*

there are no Dumb Questions

Q: I keep getting errors when I try to run my beer song code. But my code looks fine to me, so I'm a little frustrated. Any suggestions?

A: The first thing to check is that you have your indentation right. If you do, then check to see if you have mixed tabs with spaces in your code. Remember: the code will look fine (to you), but the interpreter refuses to run it. If you suspect this, a quick fix is to bring your code into an IDLE edit window, then choose *Edit...→Select All* from the menu system, before choosing *Format...→Untabify Region*. If you've mixed tabs with spaces, this will convert all your tabs to spaces in one go (and fix any indentation issues).

You can then save your code and press F5 to try running it again. If it still refuses to run, check that your code is *exactly* the same as we presented in this chapter. Be very careful of any spelling mistakes you may have made with your variable names.

Q: The Python interpreter won't warn me if I misspell `new_num` as `nwe_num`?

A: No, it won't. As long as a variable is assigned a value, Python assumes you know what you're doing, and continues to execute your code. It is something to watch for, though, so be vigilant.

Wrapping up what you already know

Here are some new things you learned as a result of working through (and running) the beer song code:

BULLET POINTS

- Indentation takes a little time to get used to. Every programmer new to Python complains about indentation at some point, but don't worry: soon you'll not even notice you're doing it.
- If there's one thing that you should never, ever do, it's mix tabs with spaces when indenting your Python code. Save yourself some future heartache, and don't do this.
- The `range` function can take more than one argument when invoked. These arguments let you control the start and stop values of the generated range, as well as the step value.
- The `range` function's step value can also be specified with a negative value, which changes the direction of the generated range

With all the beer gone, what's next?

That's it for Chapter 1. In the next chapter, you are going to learn a bit more about how Python handles data. We only just touched on **lists** in this chapter, and it's time to dive in a little deeper.

Chapter 1's Code

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

We started with the "odd.py" program, then...

... extended the code to create "odd2.py", which ran the "minute checking code" five times (thanks to the use of Python's "for" loop).

```
from datetime import datetime

import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = random.randint(1, 60)
    time.sleep(wait_time)
```

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on the wall.")
    print(beer_num, word, "of beer.")
    print("Take one down.")
    print("Pass it around.")
    if beer_num == 1:
        print("No more bottles of beer on the wall.")
    else:
        new_num = beer_num - 1
        if new_num == 1:
            word = "bottle"
        print(new_num, word, "of beer on the wall.")
    print()
```

We concluded this chapter with the Python version of the Head First classic "beer song." And, yes, we know: it's hard not to work on this code without singing along... ☺

2 list data

Working with Ordered Data

All programs process data, and Python programs are no exception.

In fact, take a look around: *data is everywhere*. A lot of, if not most, programming is all about data: *acquiring* data, *processing* data, *understanding* data. To work with data effectively, you need somewhere to *put* your data when processing it. Python shines in this regard, thanks (in no small part) to its inclusion of a handful of *widely applicable* data structures: **lists**, **dictionaries**, **tuples**, and **sets**. In this chapter, we'll preview all four, before spending the majority of this chapter digging deeper into **lists** (and we'll deep-dive into the other three in the next chapter). We're covering these data structures early, as most of what you'll likely do with Python will revolve around working with data.

Numbers, Strings...and Objects

Working with a *single* data value in Python works just like you'd expect it to. Assign a value to a variable, and you're all set. With help from the shell, let's look at some examples to recall what we learned in the last chapter.

Numbers

Let's assume that this example has already imported the `random` module. We then call the `random.randint` function to generate a random number between 1 and 60, which is then assigned to the `wait_time` variable. As the generated number is an **integer**, that's what type `wait_time` is in this instance:

```
>>> wait_time = random.randint(1, 60)
>>> wait_time
26
```

Note how you didn't have to tell the interpreter that `wait_time` is going to contain an integer. We *assigned* an integer to the variable, and the interpreter took care of the details (note: not all programming languages work this way).

A variable takes on the type of the value assigned.

Strings

If you assign a string to a variable, the same thing happens: the interpreter takes care of the details. Again, we do not need to declare ahead of time that the `word` variable in this example is going to contain a **string**:

```
>>> word = "bottles"
>>> word
'bottles'
```

This ability to *dynamically* assign a value to a variable is central to Python's notion of variables and type. In fact, things are more general than this in that you can assign *anything* to a variable in Python.

Everything is an object in Python, and any object can be assigned to a variable.

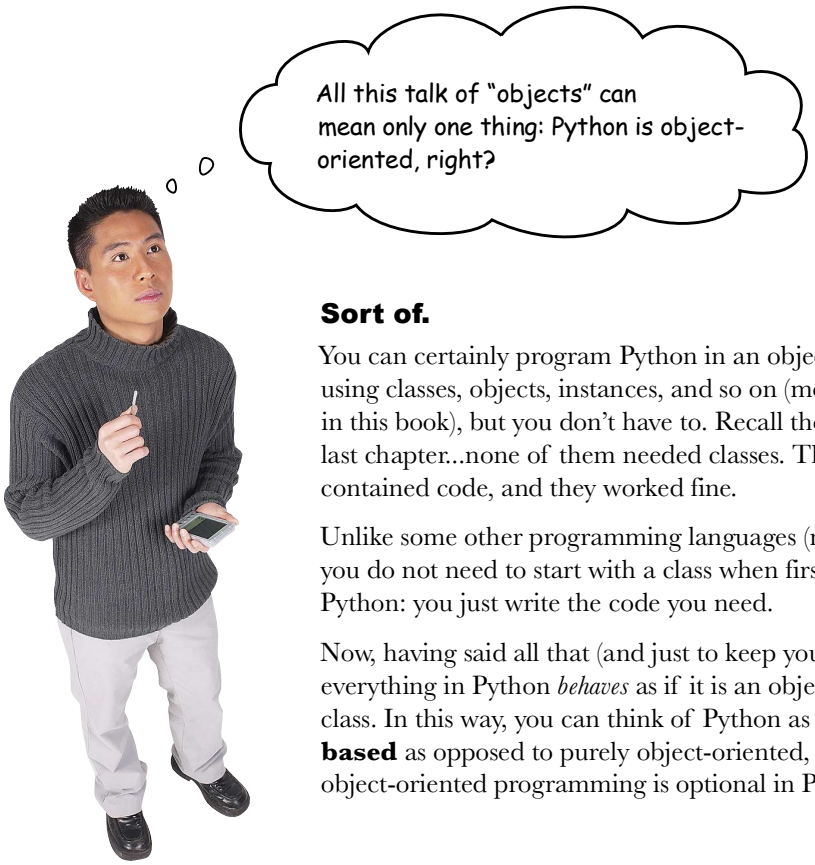
Objects

In Python everything is an object. The means that numbers, strings, functions, modules—*everything*—is an object. A direct consequence of this is that all objects can be assigned to variables. This has some interesting ramifications, which we'll start learning about on the next page.

"Everything Is an Object"

Any object can be dynamically assigned to any variable in Python. Which begs the question: *what's an object in Python?* The answer: **everything is an object**.

All data values in Python are objects, even though—on the face of things—"Don't panic!" is a string and 42 is a number. To Python programmers, "Don't panic!" is a *string object* and 42 is a *number object*. Like in other programming languages, objects can have **state** (attributes or values) and **behavior** (methods).



Sort of.

You can certainly program Python in an object-oriented way using classes, objects, instances, and so on (more on all of this later in this book), but you don't have to. Recall the programs from the last chapter...none of them needed classes. Those programs just contained code, and they worked fine.

Unlike some other programming languages (most notably, *Java*), you do not need to start with a class when first creating code in Python: you just write the code you need.

Now, having said all that (and just to keep you on your toes), everything in Python *behaves* as if it is an object *derived from* some class. In this way, you can think of Python as being more **object-based** as opposed to purely object-oriented, which means that object-oriented programming is optional in Python.

But...what does all this actually mean?

As everything is an object in Python, any "thing" can be assigned to any variable, and variables can be assigned *anything* (regardless of what the thing is: a number, a string, a function, a widget...any object). Tuck this away in the back of your brain for now; we'll return to this theme many times throughout this book.

There's really not a lot more to storing single data values in variables. Let's now take a look at Python's built-in support for storing a **collection** of values.

Meet the Four Built-in Data Structures

Python comes with **four** built-in *data structures* that you can use to hold any *collection* of objects, and they are **list**, **tuple**, **dictionary**, and **set**.

Note that by "built-in" we mean that lists, tuples, dictionaries, and sets are always available to your code and *they do not need to be imported prior to use*: each of these data structures is part of the language.

Over the next few pages, we present an overview of all four of these built-in data structures. You may be tempted to skip over this overview, but please don't.

If you think you have a pretty good idea what a **list** is, think again. Python's list is more similar to what you might think of as an *array*, as opposed to a *linked-list*, which is what often comes to mind when programmers hear the word "list." (If you're lucky enough not to know what a linked-list is, sit back and be thankful).

Python's list is the first of two ordered-collection data structures:

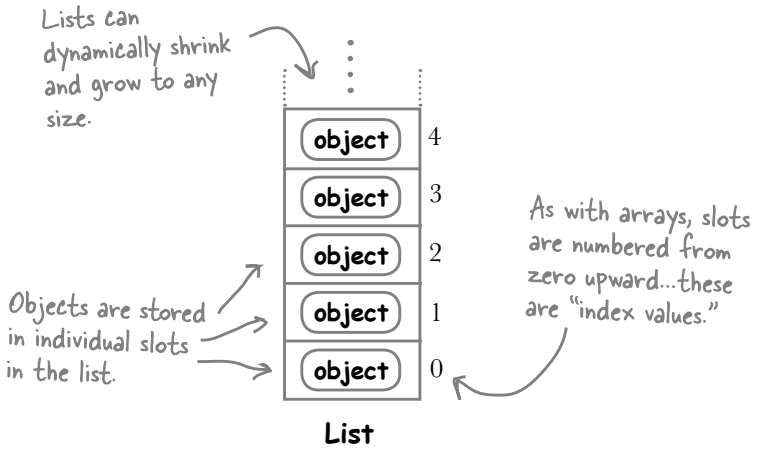
1. **List: an ordered mutable collection of objects**

   A list in Python is very similar to the notion of an **array** in other programming languages, in that you can think of a list as being an indexed collection of related objects, with each slot in the list numbered from zero upward.

   Unlike arrays in a lot of other programming languages, though, lists are **dynamic** in Python, in that they can grow (and shrink) on demand. There is no need to predeclare the size of a list prior to using it to store any objects.

   Lists are also heterogeneous, in that you do not need to predeclare the type of the object you're storing—you can mix'n'match objects of different types in the one list if you like.

   Lists are **mutable**, in that you can change a list at any time by adding, removing, or changing objects.



A list is like an array—the objects it stores are ordered sequentially in slots.

Ordered Collections Are Mutable/Immutable

Python's list is an example of a **mutable** data structure, in that it can change (or mutate) at runtime. You can grow and shrink a list by adding and removing objects as needed. It's also possible to change any object stored in any slot. We'll have lots more to say about lists in a few pages' time as the remainder of this chapter is devoted to providing a comprehensive introduction to using lists.

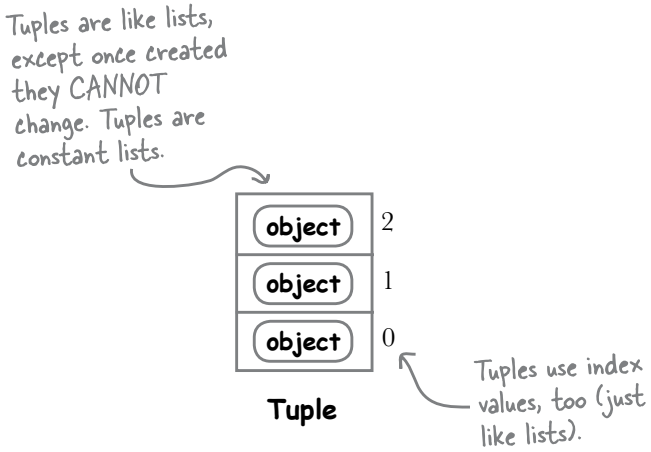
When an ordered list-like collection is **immutable** (that is, it cannot change), it's called a **tuple**:

2 **Tuple: an ordered immutable collection of objects**

A tuple is an immutable list. This means that once you assign objects to a tuple, the tuple cannot be changed under any circumstance.

It is often useful to think of a tuple as a constant list.

Most new Python programmers scratch their head in bemusement when they first encounter tuples, as it can be hard to work out their purpose. After all, what use is a list that cannot change? It turns out that there are plenty of use cases where you'll want to ensure that your objects can't be changed by your (or anyone else's) code. We'll return to tuples in the next chapter (as well as later in this book) when we talk about them in a bit more detail, as well as use them.



A tuple is an immutable list.

Lists and tuples are great when you want to present data in an ordered way (such as a list of destinations on a travel itinerary, where the order of destinations *is* important). But sometimes the order in which you present the data *isn't* important. For instance, you might want to store some user's details (such as their *id* and *password*), but you may not care in what order they're stored (just that they are). With data like this, an alternative to Python's list/tuple is needed.

An Unordered Data Structure: Dictionary

If keeping your data in a specific order isn't important to you, but structure is, Python comes with a choice of two unordered data structures: **dictionary** and **set**. Let's look at each in turn, starting with Python's dictionary.

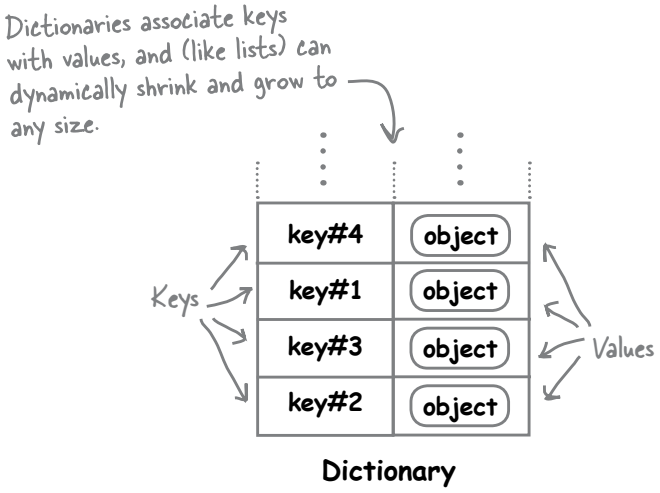
3

Dictionary: an unordered set of key/value pairs

Depending on your programming background, you may already know what a **dictionary** is, but you may know it by another name, such as associative array, map, symbol table, or hash.

Like those other data structures in those other languages, Python's dictionary allows you to store a collection of key/value pairs. Each unique **key** has a **value** associated with it in the dictionary, and dictionaries can have any number of pairs. The values associated with a key can be any object.

Dictionaries are unordered and mutable. It can be useful to think of Python's dictionary as a two-columned, multirow data structure. Like lists, dictionaries can grow (and shrink) on demand.



A dictionary stores key/value pairs.

Something to watch out for when using a dictionary is that you cannot rely upon the internal ordering used by the interpreter. Specifically, the order in which you add key/value pairs to a dictionary is not maintained by the interpreter, and has no meaning (to Python). This can stump programmers when they first encounter it, so we're making you aware of it now so that when we meet it again—and in detail—in the next chapter, you'll get less of a shock. Rest assured: it is possible to display your dictionary data in a specific order if need be, and we'll show you how to do that in the next chapter, too.

A Data Structure That Avoids Duplicates: Set

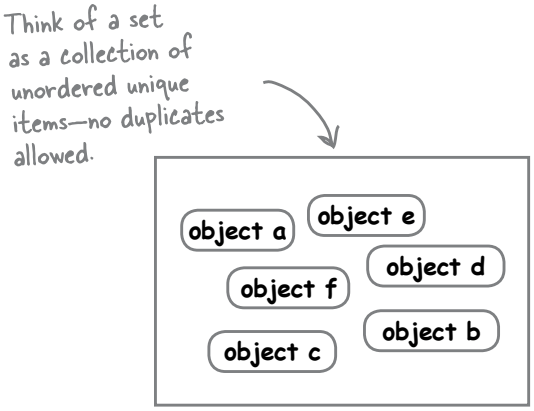
The final built-in data structure is the **set**, which is great to have at hand when you want to remove duplicates quickly from any other collection. And don't worry if the mention of sets has you recalling high school math class and breaking out in a cold sweat. Python's implementation of sets can be used in lots of places.

4 **Set: an unordered set of unique objects**

In Python, a **set** is a handy data structure for remembering a collection of related objects while ensuring none of the objects are duplicated.

The fact that sets let you perform unions, intersections, and differences is an added bonus (especially if you are a math type who loves set theory).

Sets, like lists and dictionaries, can grow (and shrink) as needed. Like dictionaries, sets are unordered, so you cannot make assumptions about the order of the objects in your set. As with tuples and dictionaries, you'll get to see sets in action in the next chapter.



Set

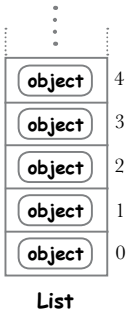
A set does not allow duplicate objects.

The 80/20 data structure rule of thumb

The four built-in data structures are useful, but they don't cover every possible data need. However, they do cover a lot of them. It's the usual story with technologies designed to be generally useful: about 80% of what you need to do is covered, while the other, highly specific, 20% requires you to do more work. Later in this book, you'll learn how to extend Python to support any bespoke data requirements you may have. However, for now, in the remainder of this chapter and the next, we're going to concentrate on the 80% of your data needs.

The rest of this chapter is dedicated to exploring how to work with the first of our four built-in data structures: the **list**. We'll get to know the remaining three data structures, **dictionary**, **set**, and **tuple**, in the next chapter.

A List Is an Ordered Collection of Objects



When you have a bunch of related objects and you need to put them somewhere in your code, think **list**. For instance, imagine you have a month's worth of daily temperature readings; storing these readings in a list makes perfect sense.

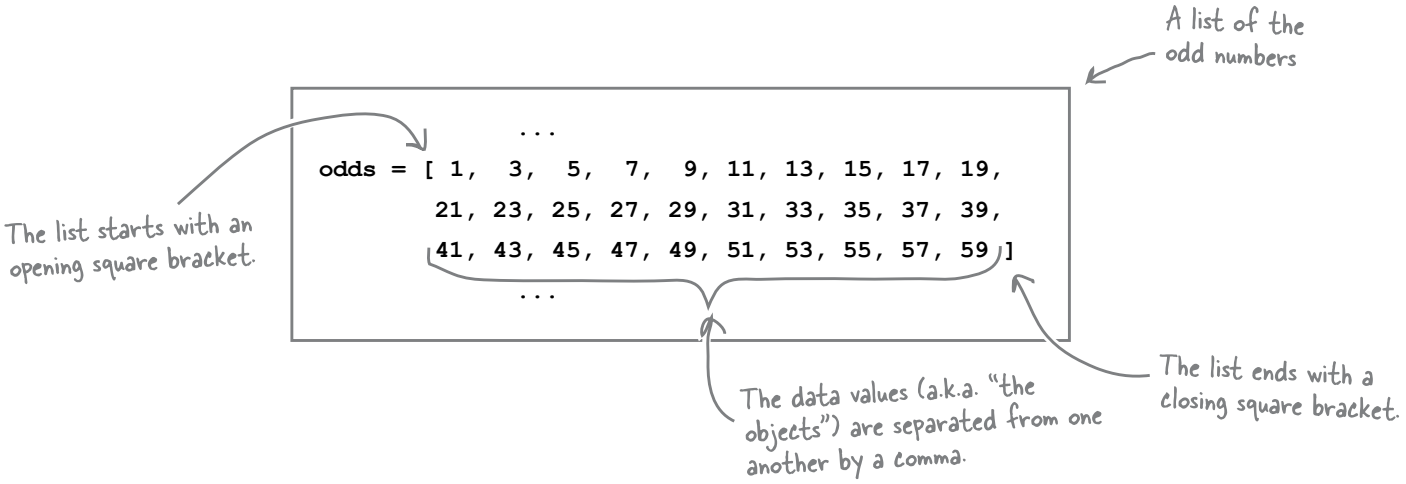
Whereas arrays tend to be homogeneous affairs in other programming languages, in that you can have an array of integers, or an array of strings, or an array of temperature readings, Python's **list** is less restrictive. You can have a list of *objects*, and each object can be of a differing type. In addition to being **heterogeneous**, lists are **dynamic**: they can grow and shrink as needed.

Before learning how to work with lists, let's spend some time learning how to spot lists in Python code.

How to spot a list in code

Lists are always enclosed in **square brackets**, and the objects contained within the list are always separated by a **comma**.

Recall the `odds` list from the last chapter, which contained the odd numbers from 0 through 60, as follows:



When a list is created where the objects are assigned to a new list directly in your code (as shown above), Python programmers refer to this as a **literal list**, in that the list is created *and* populated in one go.

The other way to create and populate a list is to "grow" the list in code, appending objects to the list as the code executes. We'll see an example of this method later in this chapter.

Let's look at some literal list examples.

Lists can be created literally or "grown" in code.

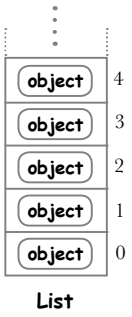
Creating Lists Literally

Our first example creates an **empty** list by assigning `[]` to a variable called `prices`:

```
prices = []
```

The variable name is on the left of the assignment operator...

...and the "literal list" is on the right. In this case, the list is empty.



Here's a list of temperatures in degrees Fahrenheit, which is a list of floats:

```
temps = [ 32.0, 212.0, 0.0, 81.6, 100.0, 45.3 ]
```

Objects (in this case, some floats) are separated by commas and surrounded by square brackets—it's a list.

How about a list of the most famous words in computer programming? Here they are:

```
words = [ 'hello', 'world' ]
```

A list of string objects

Here's a list of car details. Note how it is OK to store data of mixed types in a list. Recall that a list is "a collection of related objects." The two strings, one float, and one integer in this example are *all* Python objects, so they can be stored in a list if needed:

```
car_details = [ 'Toyota', 'RAV4', 2.2, 60807 ]
```

A list of objects of differing type

Our two final examples of literal lists exploit the fact that—as in the last example—everything is an object in Python. Like strings, floats, and integers, *lists are objects, too*. Here's an example of a list of list objects:

```
everything = [ prices, temps, words, car_details ]
```

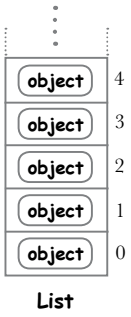
And here's an example of a literal list of literal lists:

Lists inside of a list

```
odds_and_ends = [ [ 1, 2, 3], ['a', 'b', 'c' ],
                  [ 'One', 'Two', 'Three' ] ]
```

Don't worry if these last two examples are freaking you out. We won't be working with anything as complex as this until a later chapter.

Putting Lists to Work



The literal lists on the last page demonstrate how quickly lists can be created and populated in code. Type in the data, and you're off and running.

In a page or two, we'll cover the mechanism that allows you to grow (or shrink) a list while your program executes. After all, there are many situations where you don't know ahead of time what data you need to store, nor how many objects you're going to need. In this case, your code has to grow (or "generate") the list as needed. You'll learn how to do that in a few pages' time.

For now, imagine you have a requirement to determine whether a given word contains any of the vowels (that is, the letters *a*, *e*, *i*, *o*, or *u*). Can we use Python's list to help code up a solution to this problem? Let's see whether we can come up with a solution by experimenting at the shell.

Working with lists

We'll use the shell to first define a list called `vowels`, then check to see if each letter in a word is in the `vowels` list. Let's define a list of vowels:

A list of the five vowels

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
```

With `vowels` defined, we now need a word to check, so let's create a variable called `word` and set it to `"Milliways"`:

Here's a word to check.

```
>>> word = "Milliways"
```

Geek Bits

We're only using the letters *aeiou* as vowels, even though the letter *y* is considered to be both a vowel and a consonant.

Is one object inside another? Check with "in"

If you remember the programs from Chapter 1, you will recall that we used Python's `in` operator to check for membership when we needed to ask whether one object was inside another. We can take advantage of `in` again here:

```
>>> for letter in word:
        if letter in vowels:
            print(letter)

i
i
a
```

Take each letter in the word...

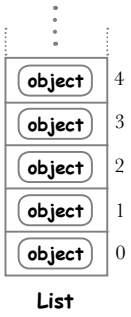
...and if it is in the "vowels" list...

...display the letter on screen.

The output from this code confirms the identity of the vowels in the word "Milliways".

Let's use this code as the basis for our working with lists.

Use Your Editor When Working on More Than a Few Lines of Code



In order to learn a bit more about how lists work, let's take this code and extend it to display each found vowel only once. At the moment, the code displays each vowel more than once on output if the word being searched contains more than one instance of the vowel.

First, let's copy and paste the code you've just typed from the shell into a new IDLE edit window (select *File...→New File...* from IDLE's menu). We're going to be making a series of changes to this code, so moving it into the editor makes perfect sense. As a general rule, when the code we're experimenting with at the >>> prompt starts to run to more than a few lines, we find it more convenient to use the editor. Save your five lines of code as `vowels.py`.

When copying code from the shell into the editor, **be careful** *not* to include the >>> prompt in the copy, as your code won't run if you do (the interpreter will throw a syntax error when it encounters >>>).

When you've copied your code and saved your file, your IDLE edit window should look like this:

Your list example code saved as "vowels.py" inside an IDLE edit window.

vowels.py - /Users/Paul/Desktop/_NewBook/ch02/vowels.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
for letter in word:
    if letter in vowels:
        print(letter)
```

Ln: 7 Col: 0

Don't forget: press F5 to run your program

With the code in the edit window, press F5 and then watch as IDLE jumps to a restarted shell window, then displays the program's output:

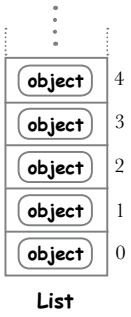
Python 3.4.3 Shell

```
>>>
>>> ========== RESTART ==========
>>>
i
i
a
>>>
```

Ln: 20 Col: 4

As expected, this output matches what we produced at the bottom of the last page, so we're good to go.

"Growing" a List at Runtime



Our current program *displays* each found vowel on screen, including any duplicates found. In order to list each unique vowel found (and avoid displaying duplicates), we need to remember any unique vowels that we find, before displaying them on screen. To do this, we need to use a second data structure.

We can't use the existing `vowels` list because it exists to let us quickly determine whether the letter we're currently processing is a vowel. We need a second list that starts out empty, as we're going to populate it at runtime with any vowels we find.

As we did in the last chapter, let's experiment at the shell *before* making any changes to our program code. To create a new, empty list, decide on a new variable name, then assign an empty list to it. Let's call our second list `found`. Here we assign an empty list (`[]`) to `found`, then use Python's built-in function `len` to check how many objects are in a collection:

```
>>> found = []
>>> len(found)
0
```

An empty list...

...which the interpreter (thanks to "len") confirms has no objects.

The "len" built-in function reports on the size of an object.

Lists come with a collection of built-in **methods** that you can use to manipulate the list's objects. To invoke a method use the *dot-notation syntax*: postfix the list's name with a dot and the method invocation. We'll meet more methods later in this chapter. For now, let's use the `append` method to add an object to the end of the empty list we just created:

```
>>> found.append('a')
>>> len(found)
1
>>> found
['a']
```

Add to an existing list at runtime using the "append" method.

The length of the list has now increased.

Asking the shell to display the contents of the list confirms the object is now part of the list.

Repeated calls to the `append` method add more objects onto the end of the list:

```
>>> found.append('e')
>>> found.append('i')
>>> found.append('o')
>>> len(found)
4
>>> found
['a', 'e', 'i' 'o']
```

More runtime additions

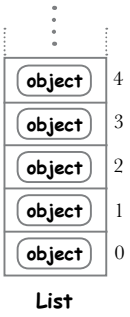
Once again, we use the shell to confirm all is in order.

Lists come with a bunch of built-in methods.

Let's now look at what's involved in checking whether a list contains an object.

Checking for Membership with "in"

We already know how to do this. Recall the "Millyways" example from a few pages ago, as well as the `odds.py` code from the previous chapter, which checked to see whether a calculated minute value was in the `odds` list:



The "in" operator checks for membership.

```
    ...
if right_this_minute in odds:
    print("This minute seems a little odd.")
    ...
```

Is the object "in" or "not in"?

As well as using the `in` operator to check whether an object is contained within a collection, it is also possible to check whether an object *does not exist within a collection* using the `not in` operator combination.

Using `not in` allows you to append to an existing list *only* when you know that the object to be added isn't already part of the list:

```
>>> if 'u' not in found:
        found.append('u')

>>> found
['a', 'e', 'i' 'o', 'u']
>>>
>>> if 'u' not in found:
        found.append('u')

>>> found
['a', 'e', 'i' 'o', 'u']
```

This first invocation of "append" works, as "u" does not currently exist within the "found" list (as you saw on the previous page, the list contained ['a', 'e', 'i', 'o']).

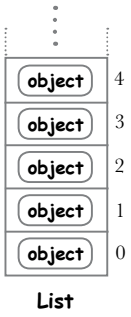
This next invocation of "append" does not execute, as "u" already exists in "found" so does not need to be added again.

Would it not be better to use a set here? Isn't a set a better choice when you're trying to avoid duplicates?

Good catch. A set might be better here.

But, we're going to hold off on using a set until the next chapter. We'll return to this example when we do. For now, concentrate on learning how a list can be generated at runtime with the `append` method.

It's Time to Update Our Code



Now that we know about `not in` and `append`, we can change our code with some confidence. Here's the original code from `vowels.py` again:

The original "vowels.py" code

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
for letter in word:
    if letter in vowels:
        print(letter)
```

This code displays the vowels in "word" as they are found.

Save a copy of this code as `vowels2.py` so that we can make our changes to this new version while leaving the original code intact.

We need to add in the creation of an empty `found` list. Then we need some extra code to populate `found` at runtime. As we no longer display the found vowels as we find them, another `for` loop is required to process the letters in `found`, and this second `for` loop needs to execute *after* the first loop (note how the indentation of both loops is *aligned* below). The new code you need is highlighted:

This is "vowels2.py".

Start with an empty list.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Include the code that decides whether to update the list of found vowels.

When this first "for" loop terminates, this second one gets to run, and it displays the vowels found in "word".

Let's make a final tweak to this code to change the line that sets `word` to "Milliways" to be more *generic* and more *interactive*.

Changing the line of code that reads:

```
word = "Milliways"
```

to:

```
word = input("Provide a word to search for vowels: ")
```

instructs the interpreter to *prompt* your user for a word to search for vowels. The `input` function is another piece of built-in goodness provided by Python.

Make the change as suggested on the left, then save your updated code as `vowels3.py`.

Test Drive

With the change at the bottom of the last page applied, and this latest version of your program saved as `vowels3.py`, let's take this program for a few spins within IDLE. Remember: to run your program multiple times, you need to return to the IDLE edit window *before* pressing the F5 key.

Here's our version of "vowels3.py" with the "input" edit applied.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

And here are our test runs...

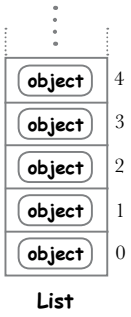
Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Milliways
i
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Hitch-hiker
i
e
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Galaxy
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Sky
>>>
```

Ln: 21 Col: 4

Our output confirms that this small program is working as expected, and it even *does the right thing* when the word contains no vowels. How did you get on when you ran your program in IDLE?

Removing Objects from a List



Lists in Python are just like arrays in other languages, and then some.

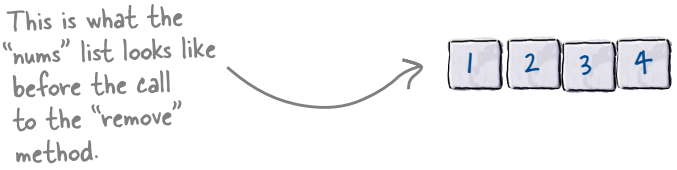
The fact that lists can grow dynamically when more space is needed (thanks to the `append` method) is a huge productivity boon. Like a lot of other things in Python, the interpreter takes care of the details for you. If the list needs more memory, the interpreter dynamically *allocates* as much memory as needed. Likewise, when a list shrinks, the interpreter dynamically *reclaims* memory no longer needed by the list.

Other methods exist to help you manipulate lists. Over the next four pages we introduce four of the most useful methods: `remove`, `pop`, `extend`, and `insert`:

1 **remove: takes an object's value as its sole argument**

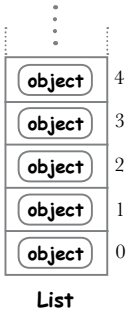
The `remove` method removes the first occurrence of a specified data value from a list. If the data value is found in the list, the object that contains it is removed from the list (and the list shrinks in size by one). If the data value is *not* in the list, the interpreter will *raise an error* (more on this later):

```
>>> nums = [1, 2, 3, 4]
>>> nums
[1, 2, 3, 4]
```




```
>>> nums.remove(3)
>>> nums
[1, 2, 4]
```

Popping Objects Off a List



The `remove` method is great for when you know the value of the object you want to remove. But often it is the case that you want to remove an object from a specific index slot.

For this, Python provides the `pop` method:

2 **pop: takes an optional index value as its argument**

The `pop` method removes *and returns* an object from an existing list based on the object's index value. If you invoke `pop` without specifying an index value, the last object in the list is removed and returned. If you specify an index value, the object in that location is removed and returned. If a list is empty or you invoke `pop` with a nonexistent index value, the interpreter *raises an error* (more on this later).

Objects returned by `pop` can be assigned to a variable if you so wish, in which case they are retained. However, if the popped object is not assigned to a variable, its memory is reclaimed and the object disappears.

Before "pop" is called, we have a list with three objects.

You didn't tell "pop" which item to remove, so it operates on the last item in the list.

```
>>> nums.pop()
4
>>> nums
[1, 2]
```

The "pop" method returns the removed object, which is reclaimed.

After the "pop", the list shrinks.

As before, "pop" returns the removed object. Once again, the object is reclaimed by the interpreter.

This is an index value. Zero corresponds to the first object in the list (the number 1).

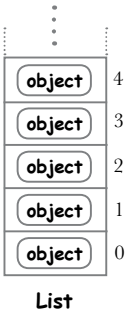
```
>>> nums.pop(0)
1
```

At this point, "nums" has been reduced to a single-item list.

```
>>> nums
[2]
```

The "nums" list has shrunk to a single-item list.

Extending a List with Objects



You already know that `append` can be used to add a single object to an existing list. Other methods can dynamically add data to a list, too:

3 **extend: takes a list of objects as its sole argument**

The `extend` method takes a second list and adds each of its objects to an existing list. This method is very useful for combining two lists into one:

This is what the "nums" list currently looks like: it is a single-item list.


```
>>> nums.extend([3, 4])
[2, 3, 4]
```

Provide a list of objects to append to the existing list.

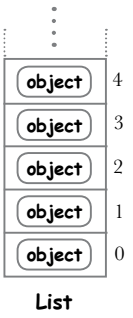
We've extended this "nums" list by taking each of the objects in the provided list and appending its objects.


```
>>> nums.extend([])
[2, 3, 4]
```

Using an empty list here is valid, if a little silly (as you're adding no items to the end of an existing list). If you'd instead called "append([])", an empty list would be added to the end of the existing list, but—in this example—using "extend([])" does nothing.

Because the empty list used to extend the "nums" list contained no objects, nothing changes.

Inserting an Object into a List



The `append` and `extend` methods get a lot of use, but they are restricted to adding objects onto the end (the righthand side) of an existing list. Sometimes, you'll want to add to the beginning (the lefthand side) of a list. When this is the case, you'll want to use the `insert` method.

4 **insert: takes an index value and an object as its arguments**

The `insert` method inserts an object into an existing list *before* a specified index value. This lets you insert the object at the start of an existing list or anywhere within the list. It is not possible to insert at the end of the list, as that's what the `append` method does:

Here's how the "nums" list looked after all that extending from the previous page.


```
>>> nums.insert(0, 1)
>>> nums
[1, 2, 3, 4]
```

The value (aka "object") to insert

The index of the object to insert *before*

Back to where we started

After all that removing, popping, extending, and inserting, we've ended up with the same list we started with a few pages ago: `[1, 2, 3, 4]`.

Note how it's also possible to use `insert` to add an object into any slot in an existing list. In the example above, we decided to add an object (the number 1) to the start of the list, but we could just as easily have used any slot number to insert *into* the list. Let's look at one final example, which—just for fun—adds a string into the middle of the `nums` list, thanks to the use of the value `2` as the first argument to `insert`:

The first argument to "insert" indicates the index value to insert *before*.

```
>>> nums.insert(2, "two-and-a-half")
>>> nums
[1, 2, 'two-and-a-half', 3, 4]
```

And there it is—the final "nums" list, which has five objects: four numbers and one string.

Let's now gain some experience using these list methods.

What About Using Square Brackets?

I'm a little confused. You keep telling me that lists are "just like arrays in other programming languages," but you've yet to say anything about the square bracket notation I use with arrays in my other favorite programming language. What gives?

Don't worry, we're going to get to that in a bit.

The familiar square bracket notation that you know and love from working with arrays in other programming languages does indeed work with Python's lists. However, before we get around to discussing how, let's have a bit of fun with some of the list methods that you now know about.

there are no Dumb Questions

Q: How do I find out more about these and any other list methods?

A: You ask for help. At the >>> prompt, type ***`help(list)`*** to access Python's list documentation (which provides a few pages of material) or type ***`help(list.append)`*** to request just the documentation for the `append` method. Replace `append` with any other list method name to access that method's documentation.

Sharpen your pencil

Time for a challenge.

Before you do anything else, take the seven lines of code shown below and type them into a new IDLE edit window. Save the code as `panic.py`, and execute it (by pressing F5).

Study the messages that appear on screen. Note how the first four lines of code take a string (in `phrase`), and turn it into a list (in `plist`), before displaying both `phrase` and `plist` on screen.

The other three lines of code take `plist` and transform it back into a string (in `new_phrase`) before displaying `plist` and `new_phrase` on screen.

Your challenge is to *transform* the string `"Don't panic!"` into the string `"on tap"` using only the list methods shown thus far in this book. (There's no hidden meaning in the choice of these two strings: it's merely a matter of the letters in "on tap" appearing in `"Don't panic!"`). At the moment, `panic.py` displays `"Don't panic!"` *twice*.

Hint: use a `for` loop when performing any operation multiple times.

We are starting with a string.

We turn the string into a list.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
```

We display the string and the list on screen.

Add your list manipulation code here.

```
new_phrase = ''.join(plist)
```

This line takes the list and turns it back into a string.

We display the transformed list and the new string on screen.

```
print(plist)
print(new_phrase)
```

Sharpen your pencil Solution

It was time for a challenge.

Before you did anything else, you were to take the seven lines of code shown on the previous page and type them into a new IDLE edit window, save the code as `panic.py`, and execute it (by pressing F5).

Your challenge was to *transform* the string `"Don't panic!"` into the string `"on tap"` using only the list methods shown thus far in this book. Before your changes, `panic.py` displayed "Don't panic!" *twice*.

The new string (displaying "`on tap`") is to be stored in the `new_phrase` variable.

You were to add your list manipulation code here. This is what we came up with—don't worry if yours is very different from ours. There's more than one way to perform the necessary transformations using the list methods.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
```

```
for i in range(4):
    plist.pop()
```

This small loop pops the last four objects from "plist". No more "nic!".

Get rid of the 'D' at the start of the list.

```
plist.pop(0)
```

Find, then remove, the apostrophe from the list.

```
plist.remove(" ' ")
```

Swap the two objects at the end of the list by first popping each object from the list, then using the popped objects to extend the list. This is a line of code that you'll need to think about for a little bit. Key point: the pops occur *first* (in the order shown), then the extend happens.

```
plist.extend([plist.pop(), plist.pop()])
```

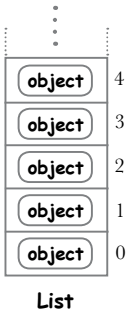
```
plist.insert(2, plist.pop(3))
```

This line of code pops the space from the list, then inserts it back into the list at index location 2. Just like the last line of code, the pop occurs *first*, before the insert happens. And, remember: spaces are characters, too.

```
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

As there's a lot going on in this exercise solution, the next two pages explain this code in detail.

What Happened to "plist"?



Let's pause to consider what actually happened to `plist` as the code in `panic.py` executed.

On the left of this page (and the next) is the code from `panic.py`, which, like every other Python program, is executed from top to bottom. On the right of this page is a visual representation of `plist` together with some notes about what's happening. Note how `plist` dynamically shrinks and grows as the code executes:

The Code

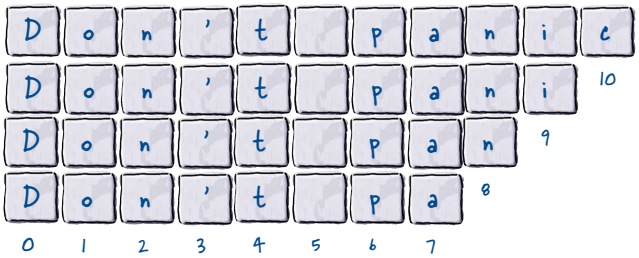
```
phrase = "Don't panic!"

plist = list(phrase)

print(phrase)
print(plist)

for i in range(4):
    plist.pop()

plist.pop(0)

plist.remove("'")
```

These calls to "print" display the current state of the variables (before we start our manipulations).

The State of `plist`

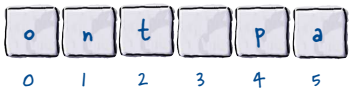
At this point in the code, `plist` does not yet exist. The second line of code *transforms* the `phrase` string into a new list, which is assigned to the `plist` variable:

Each time the `for` loop iterates, `plist` shrinks by one object until the last four objects are gone:



The loop terminates, and `plist` has shrunk until eight objects remain. It's now time to get rid of some other unwanted objects. Another call to `pop` removes the first item on the list (which is at index number 0):

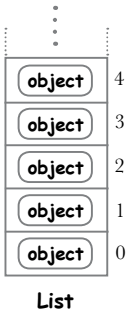
With the letter `D` popped off the front of the list, a call to `remove` dispatches with the apostrophe:



What Happened to "plist", Continued

We've been pausing for a moment to consider what actually happened to `plist` as the code in `panic.py` executed.

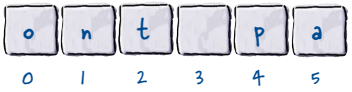
Based on the execution of the code from the last page, we now have a six-item list with the characters `o`, `n`, `t`, space, `p`, and `a` available to us. Let's keep executing our code:



The Code

The State of `plist`

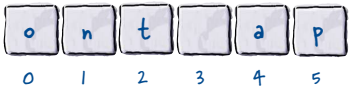
This is what `plist` looks like as a result of the code on the previous page executing:



The next line of code contains **three** method invocations: two calls to `pop` and one to `extend`. The calls to `pop` happen first (from left to right):

```
plist.extend([plist.pop(), plist.pop()])
```

The call to `extend` takes the popped objects and adds them to the end of `plist`. It can be useful to think of `extend` as shorthand for multiple calls to the `append` method:



All that's left to do (to `plist`) is to swap the `t` character at location 2 with the space character at index location 3. The next line of code contains **two** method invocations. The first uses `pop` to extract the space character:

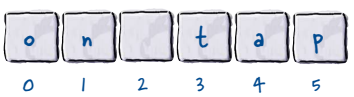
```
plist.insert(2, plist.pop(3))
```

Turn "plist" back into a string.

```
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

These calls to "print" display the state of the variables (after we've performed our manipulations).

Then the call to `insert` slots the space character into the correct place (*before* index location 2):



Ta da!

Lists: What We Know

We're 20 pages in, so let's take a little break and review what we've learned about lists so far:

BULLET POINTS

- Lists are great for storing a collection of related objects. If you have a bunch of similar things that you'd like to treat as one, a list is a great place to put them.
- List are similar to arrays in other languages. However, unlike arrays in other languages (which tend to be fixed in size), Python's lists can grow and shrink dynamically as needed.
- In code, a list of objects is enclosed in square brackets, and the list objects are separated from each other by a comma.
- An empty list is represented like this: `[]`.
- The fastest way to check whether an object is in a list is to use Python's `in` operator, which checks for membership.
- Growing a list at runtime is possible due to the inclusion of a handful of list methods, which include `append`, `extend`, and `insert`.
- Shrinking a list at runtime is possible due to the inclusion of the `remove` and `pop` methods.

Yes. Care is always needed.

As working with and manipulating lists in Python is often very convenient, care needs to be taken to ensure the interpreter is doing exactly what you want it to.

A case in point is copying one list to another list. Are you copying the list, or are you copying the objects in the list? Depending on your answer and on what you are trying to do, the interpreter will behave differently. Flip the page to learn what we mean by this.

What Looks Like a Copy, But Isn't

When to comes to copying an existing list to another one, it's tempting to use the assignment operator:

```
>>> first = [1, 2, 3, 4, 5]
>>> first
[1, 2, 3, 4, 5]
>>> second = first
>>> second
[1, 2, 3, 4, 5]
```

Create a new list (and assign five number objects to it).

The "first" list's five numbers

"Copy" the existing list to a new one, called "second".

The "second" list's five numbers

So far, so good. That looks like it worked, as the five number objects from `first` have been copied to `second`:

Or, have they? Let's see what happens when we `append` a new number to `second`, which seems like a reasonable thing to do, but leads to a problem:

```
>>> second.append(6)
>>> second
[1, 2, 3, 4, 5, 6]
```

This seems OK, but isn't.

Again, so far, so good—but there's a **bug** here. Look what happens when we ask the shell to display the contents of `first`—the new object is appended to `first` too!

```
>>> first
[1, 2, 3, 4, 5, 6]
```

Whoops! The new object is appended to "first" too.

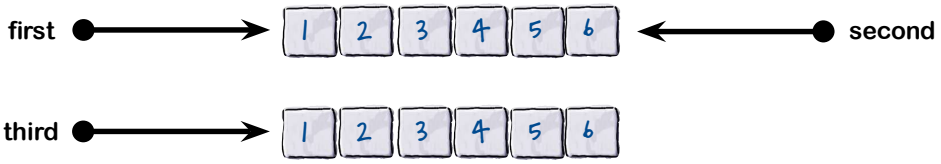
This is a problem, in that both `first` and `second` are pointing to the same data. If you change one list, the other changes, too. This is not good.

How to Copy a Data Structure

If using the assignment operator isn't the way to copy one list to another, what is? What's happening is that a **reference** to the list is *shared* among `first` and `second`.

To solve this problem, lists come with a `copy` method, which does the right thing. Take a look at how `copy` works:

```
>>> third = second.copy()
>>> third
[1, 2, 3, 4, 5, 6]
```



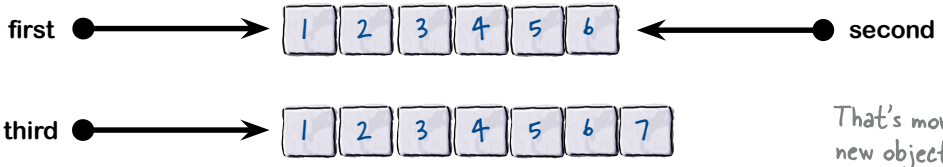
With `third` created (thanks to the `copy` method), let's append an object to it, then see what happens:

```
>>> third.append(7)
>>> third
[1, 2, 3, 4, 5, 6, 7]
>>> second
[1, 2, 3, 4, 5, 6]
```

The "third" list has grown by one object.

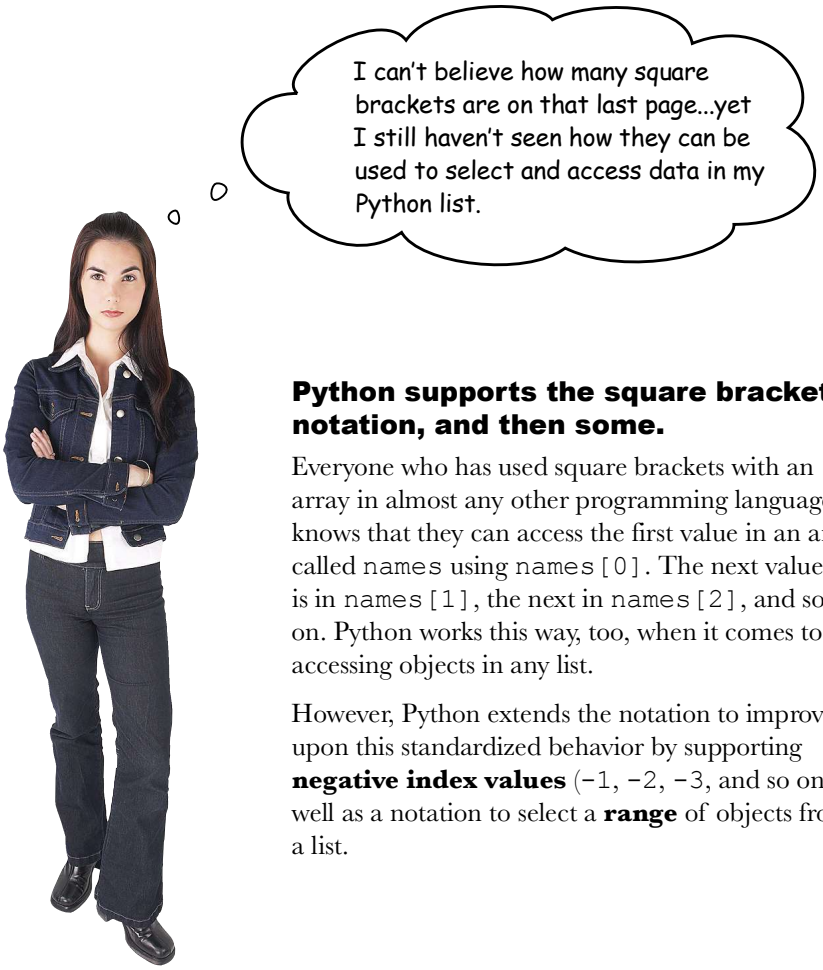
Much better. The existing list is unchanged.

Don't use the assignment operator to copy a list; use the "copy" method instead.



That's more like it—the new object is only added to the "third" list, not to the other two lists ("first" and "second").

Square Brackets Are Everywhere



Python supports the square bracket notation, and then some.

Everyone who has used square brackets with an array in almost any other programming language knows that they can access the first value in an array called `names` using `names[0]`. The next value is in `names[1]`, the next in `names[2]`, and so on. Python works this way, too, when it comes to accessing objects in any list.

However, Python extends the notation to improve upon this standardized behavior by supporting **negative index values** (`-1`, `-2`, `-3`, and so on) as well as a notation to select a **range** of objects from a list.

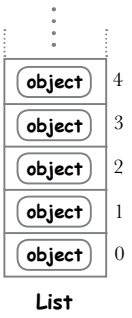
Lists: Updating What We Already Know

Before we dive into a description of how Python extends the square bracket notation, let's add to our list of bullet points:

BULLET POINTS

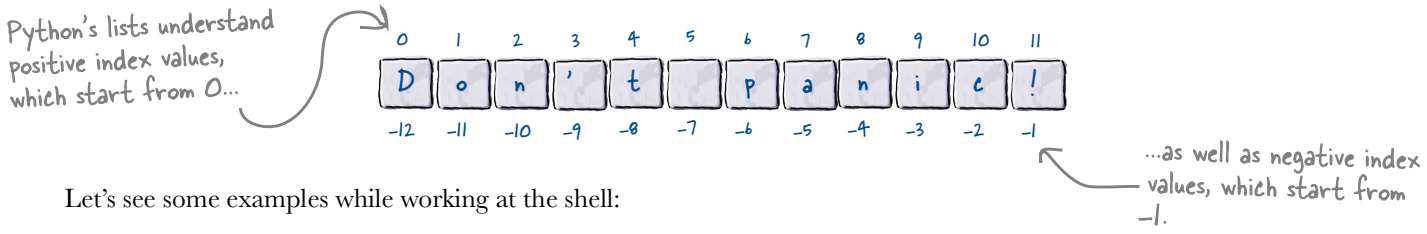
- Take care when copying one list to another. If you want to have another variable reference an existing list, use the assignment operator (=). If you want to make a copy of the objects in an existing list and use them to initialize a new list, be sure to use the `copy` method instead.

Lists Extend the Square Bracket Notation



All our talk of Python's lists being like arrays in other programming languages wasn't just idle talk. Like other languages, Python starts counting from zero when it comes to numbering index locations, and uses the well-known **square bracket notation** to access objects in a list.

Unlike a lot of other programming languages, Python lets you access the list relative to each end: positive index values count from left to right, whereas negative index values count from right to left:



Let's see some examples while working at the shell:

```
>>> saying = "Don't panic!"
>>> letters = list(saying)
>>> letters
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
>>> letters[0]
'D'
>>> letters[3]
"'"
>>> letters[6]
'p'
>>> letters[-1]
'!'
>>> letters[-3]
'i'
>>> letters[-6]
'p'
```

Create a list of letters.

Using positive index values counts from left to right...

...whereas negative index values count right to left.

As lists grow and shrink while your Python code executes, being able to index into the list using a negative index value is often useful. For instance, using `-1` as the index value is always guaranteed to return the last object in the list *no matter how big the list is*, just as using `0` always returns the first object.

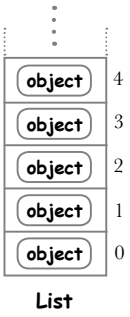
Python's extensions to the square bracket notation don't stop with support for negative index values. Lists understand **start**, **stop**, and **step**, too.

It's easy to get at the first and last objects in any list.

```
>>> first = letters[0]
>>> last = letters[-1]
>>> first
'D'
>>> last
'!'
```

Lists Understand Start, Stop, and Step

We first met **start**, **stop**, and **step** in the previous chapter when discussing the three-argument version of the `range` function:




```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on
    print(beer_num, word, "of beer.")
    print("Take one down.")
```

The call to "range" takes three arguments, one each for start, stop, and step.

Recall what **start**, **stop**, and **step** mean when it comes to specifying ranges (and let's relate them to lists):

The START value lets you control WHERE the range begins.
When used with lists, the **start** value indicates the starting index value.

The STOP value lets you control WHEN the range ends.
When used with lists, the **stop** value indicates the index value to stop at, **but not include**.

The STEP value lets you control HOW the range is generated.
When used with lists, the **step** value refers to the *stride* to take.

You can put start, stop, and step inside square brackets

When used with lists, **start**, **stop**, and **step** are specified *within* the square brackets and are separated from one another by the colon (`:`) character:

```
letters[start:stop:step]
```

The square bracket notation is extended to work with start, stop, and step.

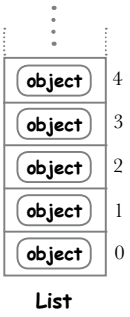
It might seem somewhat counterintuitive, but all three values are *optional* when used together:

When **start** is missing, it has a default value of `0`.

When **stop** is missing, it takes on the maximum value allowable for the list.

When **step** is missing, it has a default value of `1`.

List Slices in Action



Given the existing list `letters` from a few pages back, you can specify values for **start**, **stop**, and **step** in any number of ways.

Let's look at some examples:

All the letters

```
>>> letters
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
```

```
>>> letters[0:10:3]
['D', "'", 'p', 'i']
```

Every third letter up to (but not including) index location 10

```
>>> letters[3:]
["'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
```

Skip the first three letters, then give me everything else.

```
>>> letters[:10]
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i']
```

All letters up to (but not including) index location 10

```
>>> letters[::2]
['D', 'n', 't', 'p', 'n', 'c']
```

Every second letter

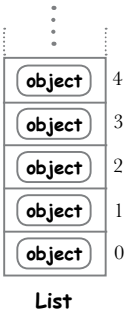
Using the start, stop, step *slice notation* with lists is very powerful (not to mention handy), and you are advised to take some time to understand how these examples work. Be sure to follow along at your >>> prompt, and feel free to experiment with this notation, too.

there are no Dumb Questions

Q: I notice that some of the characters on this page are surrounded by single quotes and others by double quotes. Is there some sort of standard I should follow?

A: No, there's no standard, as Python lets you use either single or double quotes around strings of any length, including strings that contain only a single character (like the ones shown on this page; technically, they are single-character strings, not letters). Most Python programmers use single quotes to delimit their strings (but that's a preference, not a rule). If a string contains a single quote, double quotes can be used to avoid the requirement to escape characters with a backslash (\), as most programmers find it's easier to read `"'"` than `'\''`. You'll see more examples of both quotes being used on the next two pages.

Starting and Stopping with Lists



Follow along with the examples on this page (and the next) at your >>> prompt and make sure you get the same output as we do.

We start by turning a string into a list of letters:

```
>>> book = "The Hitchhiker's Guide to the Galaxy"
>>> booklist = list(book)
>>> booklist
['T', 'h', 'e', ' ', 'H', 'i', 't', 'c', 'h', 'h', 'i', 'k',
'e', 'r', "'", 's', ' ', 'G', 'u', 'i', 'd', 'e', ' ', 't',
'o', ' ', 't', 'h', 'e', ' ', 'G', 'a', 'l', 'a', 'x', 'y']
```

Turn a string into a list, then display the list.

Note that the original string contained a single quote character. Python is smart enough to spot this, and surrounds the single quote character with double quotes.

The newly created list (called `booklist` above) is then used to select a range of letters from within the list:

```
>>> booklist[0:3]
['T', 'h', 'e']
```

Select the first three objects (letters) from the list.

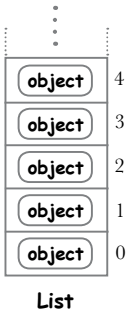
```
>>> ''.join(booklist[0:3])
'The'


>>> ''.join(booklist[-6:])
'Galaxy'
```

Turn the selected range into a string (which you learned how to do near the end of the "panic.py" code). The second example selects the last six objects from the list.

Be sure to take time to study this page (and the next) until you're confident you understand how each example works, and be sure to try out each example within IDLE.

With the last example above, note how the interpreter is happy to use any of the default values for **start**, **stop**, and **step**.

Stepping with Lists



Here are two more examples, which show off the use of **step** with lists.

The first example selects all the letters, starting from the end of the list (that is, it is selecting *in reverse*), whereas the second selects every other letter in the list. Note how the **step** value controls this behavior:

```
>>> backwards = booklist[::-1]
>>> ''.join(backwards)
"yxalaG eht ot ediuG s'rekihhctiH ehT"
```

Looks like gobbledegook, doesn't it? But it is actually the original string reversed.

```
>>> every_other = booklist[::2]
>>> ''.join(every_other)
"TeHthie' ud oteGlx"
```

And this looks like gibberish! But "every_other" is a list made up from every second object (letter) starting from the first and going to the last. Note: "start" and "stop" are defaulted.

Two final examples confirm that it is possible to start and stop anywhere within the list and select objects. When you do this, the returned data is referred to as a **slice**. Think of a slice as a *fragment* of an existing list.

Both of these examples select the letters from `booklist` that spell the word `'Hitchhiker'`. The first selection is joined to show the word `'Hitchhiker'`, whereas the second displays `'Hitchhiker'` in reverse:

A "slice" is a fragment of a list.

```
>>> ''.join(booklist[4:14])
'Hitchhiker'
```

Slice out the word "Hitchhiker".

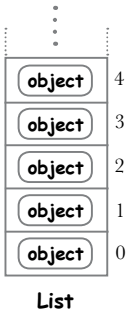
```
>>> ''.join(booklist[13:3:-1])
'rekihhctiH'
```

Slice out the word "Hitchhiker", but do it in reverse order (i.e., backward).

Slices are everywhere

The slice notation doesn't just work with lists. In fact, you'll find that you can slice any sequence in Python, accessing it with **`[start:stop:step]`**.

Putting Slices to Work on Lists



Python's slice notation is a useful extension to the square bracket notation, and it is used in many places throughout the language. You'll see lots of uses of slices as you continue to work your way through this book.

For now, let's see Python's square bracket notation (including the use of slices) in action. We are going to take the `panic.py` program from earlier and refactor it to use the square bracket notation and slices to achieve what was previously accomplished with list methods.

Before doing the actual work, here's a quick reminder of what `panic.py` does.

Converting "Don't panic!" to "on tap"

This code transforms one string into another by manipulating an existing list using the list methods. Starting with the string `"Don't panic!"`, this code produced `"on tap"` after the manipulations:

This is "panic.py".

Display the initial state of the string and list.

Use a collection of list methods to transform and manipulate the list of objects.

Display the resulting state of the string and list.

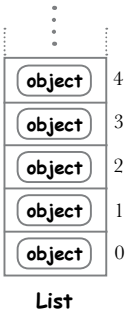
```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Here's the output produced by this program when it runs within IDLE:

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>>
                                                                  Ln: 10 Col: 4
```

The string "Don't panic!" is transformed into "on tap" thanks to the list methods.

Putting Slices to Work on Lists, Continued



It's time for the actual work. Here's the `panic.py` code again, with the code you need to change highlighted:

These are the lines of code you need to change.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Sharpen your pencil

For this exercise, replace the highlighted code above with new code that takes advantage of Python's square bracket notation. Note that you can still use list methods where it makes sense. As before, you're trying to transform `"Don't panic!"` into `"on tap"`. Add your code in the space provided and call your new program `panic2.py`:

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
.....

.....

.....

.....

.....
print(plist)
print(new_phrase)
```

Sharpen your pencil Solution

For this exercise, you were to replace the highlighted code on the previous page with new code that takes advantage of Python's square bracket notation. Note that you can still use list methods where it makes sense. As before, you're trying to transform `"Don't panic!"` into `"on tap"`. You were to call your new program `panic2.py`:

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
```

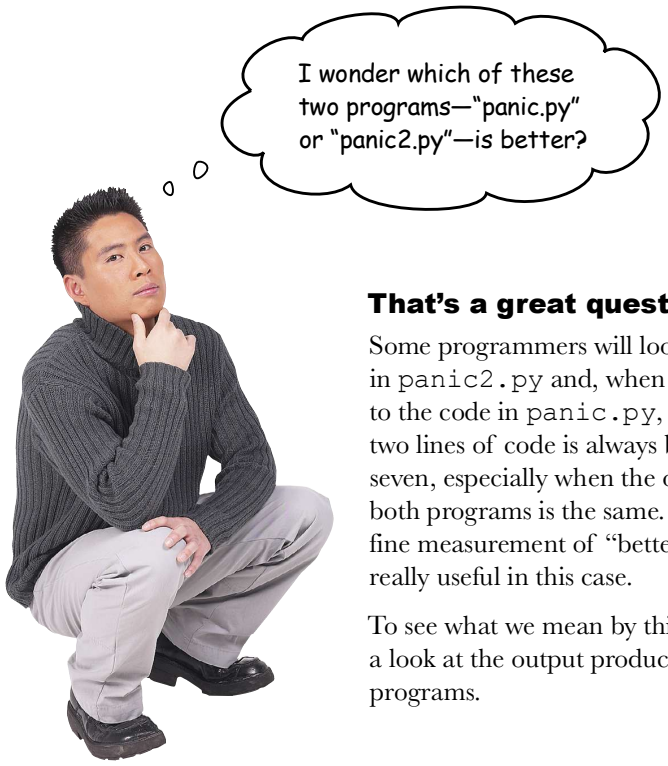
```
new_phrase = ''.join(plist[1:3])
```

We started by slicing out the word "on" from "plist"...

```
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```

...then picked out each additional letter that we needed: space, "t", "a", and "p".

```
print(plist)
print(new_phrase)
```



That's a great question.

Some programmers will look at the code in `panic2.py` and, when comparing it to the code in `panic.py`, conclude that two lines of code is always better than seven, especially when the output from both programs is the same. Which is a fine measurement of "betterness," but not really useful in this case.

To see what we mean by this, let's take a look at the output produced by both programs.

Test Drive

Use IDLE to open `panic.py` and `panic2.py` in separate edit windows. Select the `panic.py` window first, then press F5. Next select the `panic2.py` window, then press F5. Compare the results from both programs in your shell.

"panic.py"

panic.py - /Users/Paul/Desktop/_NewBook/ch02/panic.py (3.4.3)

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Ln: 17 Col: 0

"panic2.py"

panic2.py

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
on tap
>>>
```

The output produced by running the "panic.py" program

The output produced by running the "panic2.py" program

Notice how different these outputs are.

Which Is Better? It Depends...

We executed both `panic.py` and `panic2.py` in IDLE to help us determine which of these two programs is "better."

Take a look at the second-to-last line of output from both programs:

This is the output produced by "panic.py"...

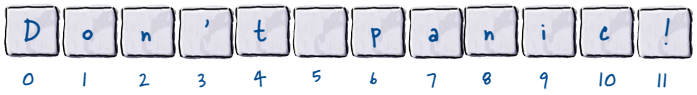
```
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>> ========================== RESTART ==========================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
on tap
>>>
```

...whereas this output is produced by "panic2.py".

Although both programs conclude by displaying the string `"on tap"` (having first started with the string `"Don't panic!"`), `panic2.py` does not change `plist` in any way, whereas `panic.py` does.

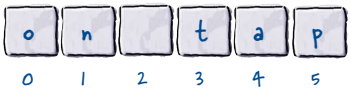
It is worth pausing for a moment to consider this.

Recall our discussion from earlier in this chapter called *"What happened to 'plist'?"*. That discussion detailed the steps that converted this list:



The "panic.py" program started with this list...

into this much shorter list:

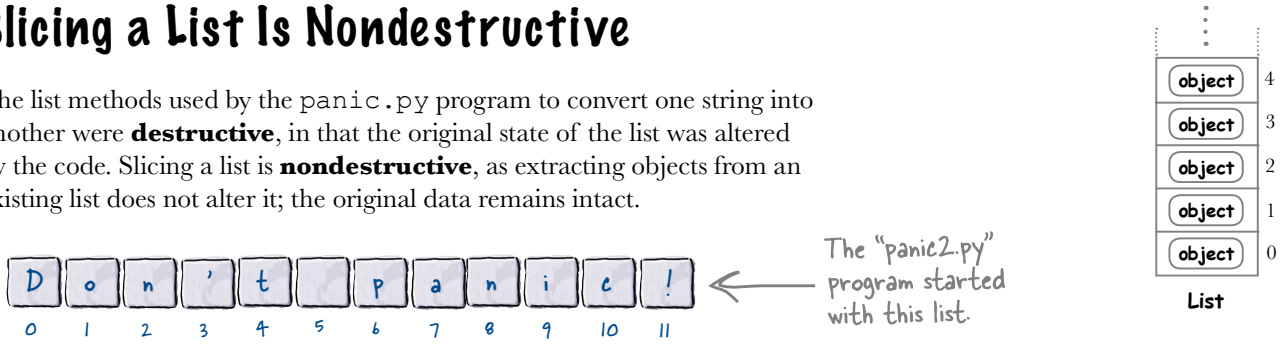


...and turned it into this one.

All those list manipulations using the `pop`, `remove`, `extend`, and `insert` methods changed the list, which is fine, as that's primarily what the list methods are designed to do: change the list. But what about `panic2.py`?

Slicing a List Is Nondestructive

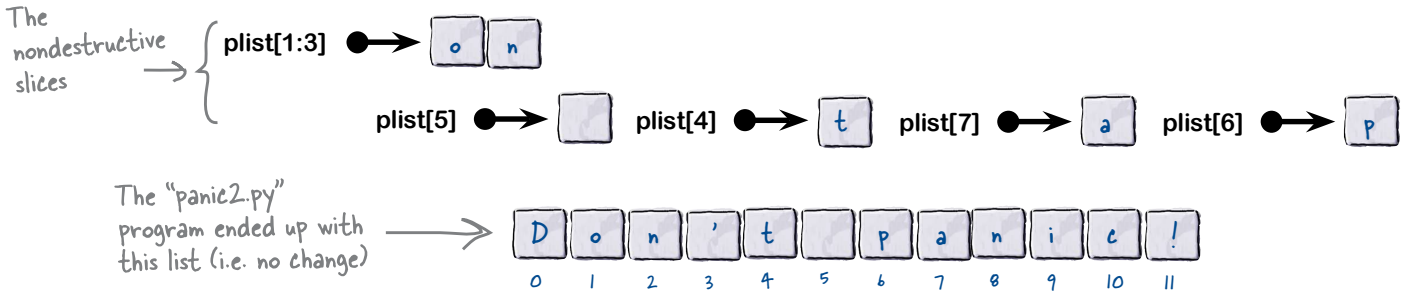
The list methods used by the `panic.py` program to convert one string into another were **destructive**, in that the original state of the list was altered by the code. Slicing a list is **nondestructive**, as extracting objects from an existing list does not alter it; the original data remains intact.



The slices used by `panic2.py` are shown here. Note that each extracts data from the list, but does not change it. Here are the two lines of code that do all the heavy lifting, together with a representation of the data each slice extracts:

The code

```
new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```



So...which is better?

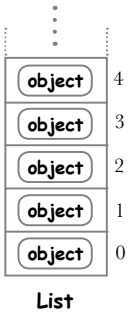
Using list methods to manipulate and transform an existing list does just that: it manipulates *and* transforms the list. The original state of the list is no longer available to your program. Depending on what you're doing, this may (or may not) be an issue. Using Python's square bracket notation generally does *not* alter an existing list, unless you decide to assign a new value to an existing index location. Using slices also results in no changes to the list: the original data remains as it was.

Which of these two approaches you decide is "better" depends on what you are trying to do (and it's perfectly OK not to like either). There is always more than one way to perform a computation, and Python lists are flexible enough to support many ways of interacting with the data you store in them.

We are nearly done with our initial tour of lists. There's just one more topic to introduce you to at this stage: *list iteration*.

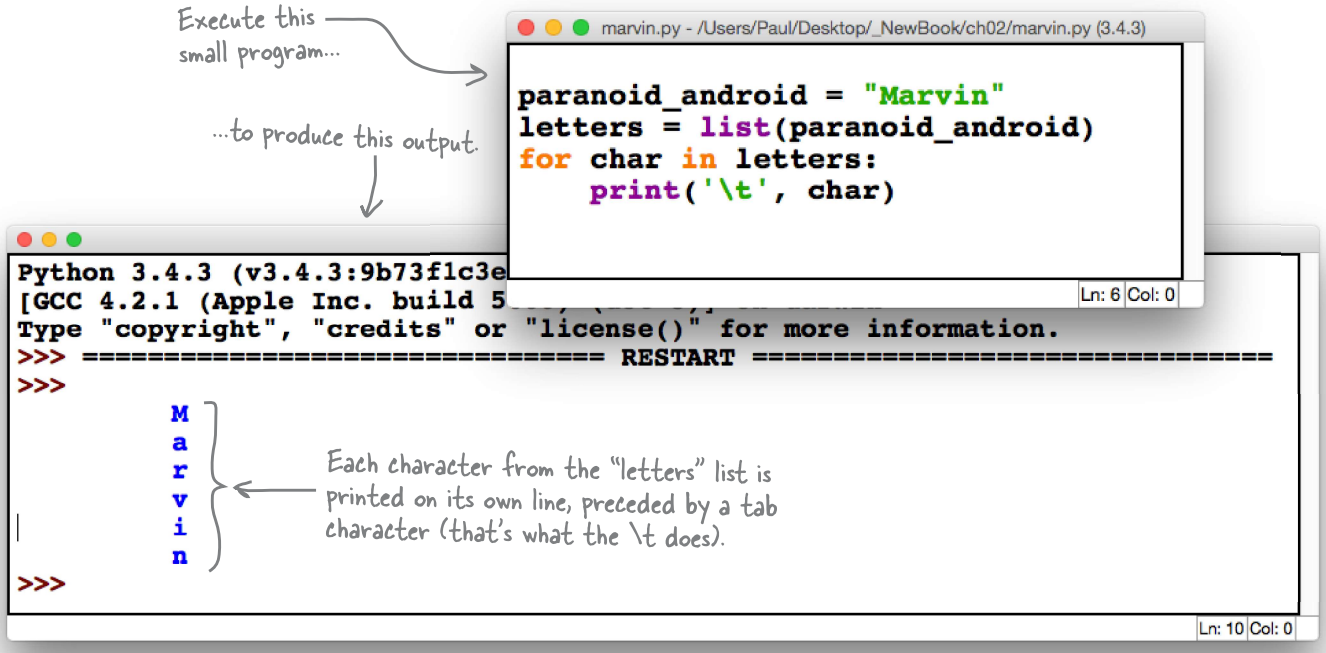
List methods change the state of a list, whereas using square brackets and slices (typically) does not.

Python's "for" Loop Understands Lists



Python's `for` loop knows all about lists and, when provided with *any* list, knows where the start of the list is, how many objects the list contains, and where the end of the list is. You never have to tell the `for` loop any of this, as it works it out for itself.

An example helps to illustrate. Follow along by opening up a new edit window in IDLE and typing in the code shown below. Save this new program as `marvin.py`, then press F5 to take it for a spin:

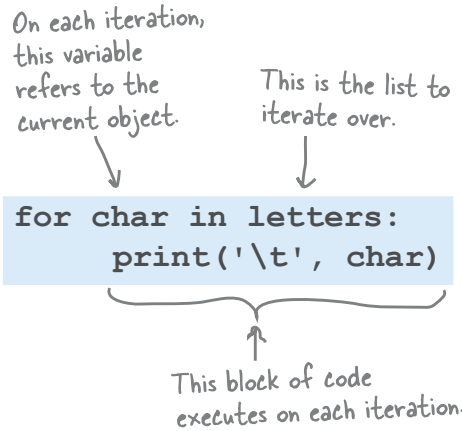


Understanding marvin.py's code

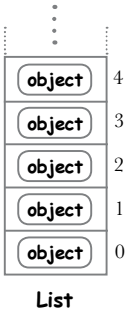
The first two lines of `marvin.py` are familiar: assign a string to a variable (called `paranoid_android`), then turn the string into a list of character objects (assigned to a new variable called `letters`).

It's the next statement—the `for` loop—that we want you to concentrate on.

On each iteration, the `for` loop arranges to take each object in the `letters` list and assign them one at a time to another variable, called `char`. Within the indented loop body `char` takes on the current value of the object being processed by the `for` loop. Note that the `for` loop knows when to *start* iterating, when to *stop* iterating, as well as *how many* objects are in the `letters` list. You don't need to worry about any of this: that's the interpreter's job.



Python's "for" Loop Understands Slices



If you use the square bracket notation to select a slice from a list, the `for` loop "does the right thing" and only iterates over the sliced objects. An update to our most recent program shows this in action. Save a new version of `marvin.py` as `marvin2.py`, then change the code to look like that shown below.

Of interest is our use of Python's **multiplication operator** (`*`) , which is used to control how many tab characters are printed before each object in the second and third `for` loop. We use `*` here to "multiply" how many times we want tab to appear:

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[12:20]:
    print('\t'*3, char)
```

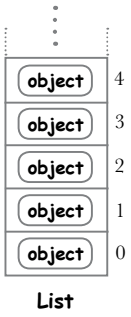
The first loop iterates over a slice of the first six objects in the list.

```
>>>
        M
        a
        r
        v
        i
        n

                A
                n
                d
                r
                o
                i
                d

                        P
                        a
                        r
                        a
                        n
                        o
                        i
                        d
>>>
```

The second loop iterates over a slice of the last seven objects in the list. Note how "*2" inserts two tab characters before each printed object.

The third (and final) loop iterates over a slice from within the list, selecting the characters that spell the word "Paranoid". Note how "*3" inserts three tab characters before each printed object.

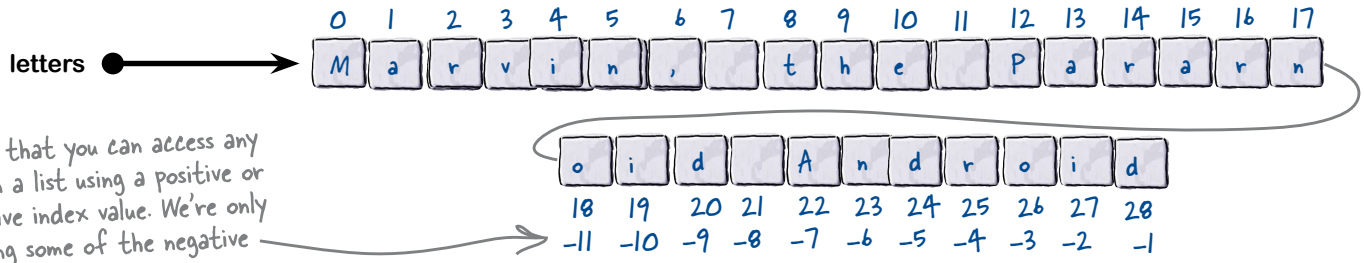
Marvin's Slices in Detail



Let's take a look at each of the slices in the last program in detail, as this technique appears a lot in Python programs. Below, each line of slice code is presented once more, together with a graphical representation of what's going on.

Before looking at the three slices, note that the program begins by assigning a string to a variable (called `paranoid_android`) and converting it to a list (called `letters`):

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
```



Recall that you can access any slot in a list using a positive or negative index value. We're only showing some of the negative index values here.

We'll look at each of the slices from the `marvin2.py` program and see what they produce. When the interpreter sees the slice specification, it extracts the sliced objects from `letters` and returns a copy of the objects to the `for` loop. The original `letters` list is unaffected by these slices.

The first slice extracts from the start of the list and ends (but doesn't include) the object in slot 6:

```
for char in letters[:6]:
    print('\t', char)
```

The second slice extracts from the end of the `letters` list, starting at slot −7 and going to the end of `letters`:

```
for char in letters[-7:]:
    print('\t'*2, char)
```

And finally, the third slice extracts from the middle of the list, starting at slot 12 and including everything up to but not including slot 20:

```
for char in letters[12:20]:
    print('\t'*3, char)
```

Lists: Updating What We Know

Now that you've seen how lists and `for` loops interact, let's quickly review what you've learned over the last few pages:

BULLET POINTS

- Lists understand the square bracket notation, which can be used to select individual objects from any list.
- Like a lot of other programming languages, Python starts counting from zero, so the first object in any list is at index location 0, the second at 1, and so on.
- Unlike a lot of other programming languages, Python lets you index into a list from either end. Using –1 selects the last item in the list, –2 the second last, and so on.
- Lists also provide slices (or fragments) of a list by supporting the specification of start, stop, and step as part of the square bracket notation.

Lists are used a lot, but...

They are *not* a data structure panacea. Lists can be used in lots of places; if you have a collection of similar objects that you need to store in a data structure, lists are the perfect choice.

However—and perhaps somewhat counterintuitively—if the data you're working with exhibits some *structure*, lists can be a **bad choice**. We'll start exploring this problem (and what you can do about it) on the next page.

there are no Dumb Questions

Q: Surely there's a lot more to lists than this?

A: Yes, there is. Think of the material in this chapter as a quick introduction to Python's built-in data structures, together with what they can do for you. We are by no means done with lists, and will be returning to them throughout the remainder of this book.

Q: But what about sorting lists? Isn't that important?

A: Yes, it is, but let's not worry about stuff like that until we actually need to. For now, if you have a good grasp of the basics, that's all you need at this stage. And don't worry: we'll get to sorting soon.

What's Wrong with Lists?

When Python programmers find themselves in a situation where they need to store a collection of similar objects, using a list is often the natural choice. After all, we've used nothing but lists in this chapter so far.

Recall how lists are great at storing a collection of related letters, such as with the `vowels` list:

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

And if the data is a collection of numbers, lists are a great choice, too:

```
nums = [1, 2, 3, 4, 5]
```

In fact, lists are a great choice when you have a collection of related *anythings*.

But imagine you need to store data about a person, and the sample data you've been given looks something like this:

On the face of things, this data does indeed conform to a structure, in that there's *tags* on the left and *associated data values* on the right. So, why not put this data in a list? After all, this data is related to the person, right?

To see why we shouldn't, let's look at two ways to store this data using lists (starting on the next page). We are going to be totally upfront here: *both* of our attempts exhibit problems that make using lists less than ideal for data like this. But, as the journey is often half the fun of getting there, we're going to try lists anyway.

Our first attempt concentrates on the data values on the right of the napkin, whereas our second attempt uses the tags on the left as well as the associated data values. Have a think about how you'd handle this type of structured data using lists, then flip to the next page to see how our two attempts fared.

When Not to Use Lists

We have our sample data (on the back of a napkin) and we've decided to store the data in a list (as that's all we know at this point in our Python travels).

Our first attempt takes the data values and puts them in a list:

```
>>> person1 = ['Ford Prefect', 'Male',
'Researcher', 'Betelgeuse Seven']
>>> person1
['Ford Prefect', 'Male', 'Researcher',
'Betelgeuse Seven']
```

This results in a list of string objects, which works. As shown above, the shell confirms that the data values are now in a list called `person1`.

But we have a problem, in that we have to remember that the first index location (at index value 0) is the person's name, the next is the person's gender (at index value 1), and so on. For a small number of data items, this is not a big deal, but imagine if this data expanded to include many more data values (perhaps to support a profile page on that Facebook-killer you're been meaning to build). With data like this, using index values to refer to the data in the `person1` list is brittle, and best avoided.

Our second attempt adds the tags into the list, so that each data value is preceded by its associated tag. Meet the `person2` list:

```
>>> person2 = ['Name', 'Ford Prefect', 'Gender',
'Male', 'Occupation', 'Researcher', 'Home Planet',
'Betelgeuse Seven']
>>> person2
['Name', 'Ford Prefect', 'Gender', 'Male',
'Occupation', 'Researcher', 'Home Planet',
'Betelgeuse Seven']
```

This clearly works, but now we no longer have one problem; we have two. Not only do we still have to remember what's at each index location, but we now have to remember that index values 0, 2, 4, 6, and so on are tags, while index values 1, 3, 5, 7, and so on are data values.

Surely there has to be a better way to handle data with a structure like this?

There is, and it involves foregoing the use of lists for structured data like this. We need to use something else, and in Python, that something else is called a **dictionary**, which we get to in the next chapter.

If the data you want to store has an identifiable structure, consider using something other than a list.

Chapter 2's Code, 1 of 2

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
for letter in word:
    if letter in vowels:
        print(letter)
```

The first version of the vowels program that displays *all* the vowels found in the word "Milliways" (including any duplicates).

The "vowels2.py" program added code that used a list to avoid duplicates. This program displays the list of unique vowels found in the word "Milliways".

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

The third (and final) version of the vowels program for this chapter, "vowels3.py", displays the unique vowels found in a word entered by our user.

It's the best advice in the universe: "Don't panic!" This program, called "panic.py", takes a string containing this advice and, using a bunch of list methods, transforms the string into another string that describes how the Head First editors prefer their beer: "on tap".

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Chapter 2's Code, 2 of 2

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

When it comes to manipulating lists, using methods isn't the only game in town. The "panic2.py" program achieved the same end using Python's square bracket notation.

```
paranoid_android = "Marvin"
letters = list(paranoid_android)
for char in letters:
    print('\t', char)
```

The shortest program in this chapter, "marvin.py", demonstrated how well lists play with Python's "for" loop. (Just don't tell Marvin...if he hears that his program is the shortest in this chapter, it'll make him even more paranoid than he already is).

The "marvin2.py" program showed off Python's square bracket notation by using three slices to extract and display fragments from a list of letters.

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[12:20]:
    print('\t'*3, char)
```

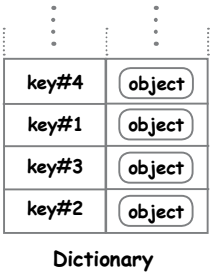
3 structured data

Working with Structured Data

Python's list data structure is great, but it isn't a data panacea.

When you have *truly* structured data (and using a list to store it may not be the best choice), Python comes to your rescue with its built-in **dictionary**. Out of the box, the dictionary lets you store and manipulate any collection of *key/value pairs*. We look long and hard at Python's dictionary in this chapter, and—along the way—meet **set** and **tuple**, too. Together with the **list** (which we met in the previous chapter), the dictionary, set, and tuple data structures provide a set of built-in data tools that help to make Python and data a powerful combination.

A Dictionary Stores Key/Value Pairs

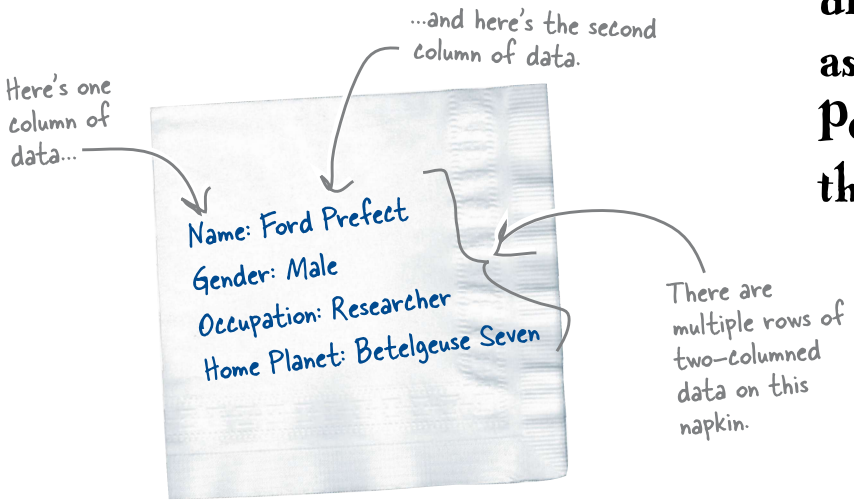


Unlike a list, which is a collection of related objects, the **dictionary** is used to hold a collection of **key/value pairs**, where each unique *key* has a *value* associated with it. The dictionary is often referred to as an *associative array* by computer scientists, and other programming languages often use other names for dictionary (such as map, hash, and table).

The key part of a Python dictionary is typically a string, whereas the associated value part can be any Python object.

Data that conforms to the dictionary model is easy to spot: there are **two columns**, with potentially **multiple rows** of data. With this in mind, take another look at our "data napkin" from the end of the last chapter:

In C++ and Java, a dictionary is known as "map," whereas Perl and Ruby use the name "hash."



It looks like the data on this napkin is a perfect fit for Python's dictionary.

Let's return to the >>> shell to see how to create a dictionary using our napkin data. It's tempting to try to enter the dictionary as a single line of code, but we're not going to do this. As we want our dictionary code to be easy to read, we're purposely entering each row of data (i.e., each key/value pair) on its own line instead. Take a look:

The name of the dictionary. (Recall that we met "person1" and "person2" at the end of the last chapter.)

The key

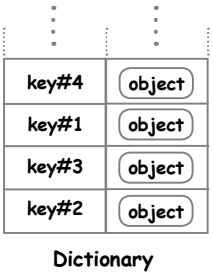
The associated data value

```
>>> person3 = { 'Name': 'Ford Prefect',
                'Gender': 'Male',
                'Occupation': 'Researcher',
                'Home Planet': 'Betelgeuse Seven' }
```

Key

Value

Make Dictionaries Easy to Read



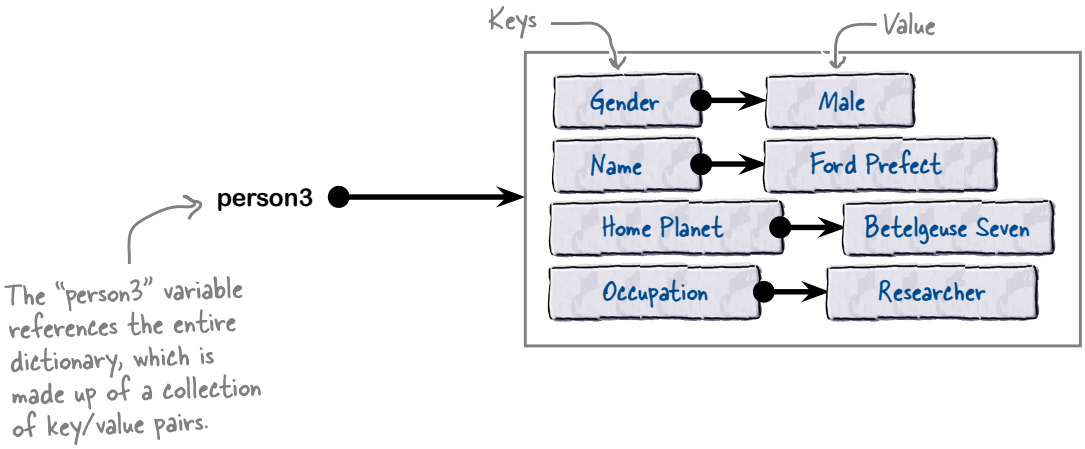
It's tempting to take the four lines of code from the bottom of the last page and type them into the shell like this:

```
>>> person3 = { 'Name': 'Ford Prefect', 'Gender':
'Male', 'Occupation': 'Researcher', 'Home Planet':
'Betelgeuse Seven' }
```

Although the interpreter doesn't care which approach you use, entering a dictionary as one long line of code is hard to read, and should be avoided whenever possible.

If you litter your code with dictionaries that are hard to read, other programmers (which includes *you* in six months' time) will get upset...so take the time to align your dictionary code so that it *is* easy to read.

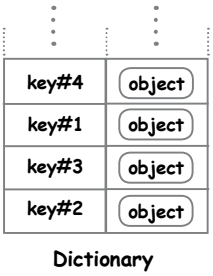
Here's a visual representation of how the dictionary appears in Python's memory after either of these dictionary-assigning statements executes:



This is a more complicated structure than the array-like list. If the idea behind Python's dictionary is new to you, it's often useful to think of it as a **lookup table**. The key on the left is used to *look up* the value on the right (just like you look up a word in a paper dictionary).

Let's spend some time getting to know Python's dictionary in more detail. We'll begin with a detailed explanation of how to spot a Python dictionary in your code, before talking about some of this data structure's unique characteristics and uses.

How to Spot a Dictionary in Code



Take a closer look at how we defined the `person3` dictionary at the >>> shell. For starters, the *entire* dictionary is enclosed in curly braces. Each **key** is enclosed in quotes, as they are strings, as is each **value,** which are also strings in this example. (Keys and values don't have to be strings, however.) Each key is separated from its associated value by a **colon** character (`:`), and each key/value pair (a.k.a. "row") is separated from the next by a **comma**:

An opening curly brace starts each dictionary.

In this dictionary, the values are all string objects, so they are enclosed in quotes.

```
{'Name': 'Ford Prefect',
'Gender': 'Male',
'Occupation': 'Researcher',
'Home Planet': 'Betelgeuse Seven'}
```

Each key/value pair is separated from the next by a comma.

Each key is enclosed in quotes.

A colon associates each key with its value.

A closing curly brace ends each dictionary.

As stated earlier, the data on this napkin maps nicely to a Python dictionary. In fact, any data that exhibits a similar structure—multiple two-columned rows—is as perfect a fit as you're likely to find. Which is great, but it does come at a price. Let's return to the >>> prompt to learn what this price is:

Ask the shell to display the contents of the dictionary...

```
>>> person3
{'Gender': 'Male', 'Name': 'Ford Prefect', 'Home
Planet': 'Betelgeuse Seven', 'Occupation': 'Researcher'}
```

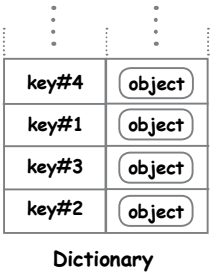
...and there it is. All the key/value pairs are shown.

What happened to the insertion order?

Take a long hard look at the dictionary displayed by the interpreter. Did you notice that the ordering is different from what was used on input? When you created the dictionary, you inserted the rows in name, gender, occupation, and home planet order, but the shell is displaying them in gender, name, home planet, and occupation order. The ordering has changed.

What's going on here? Why did the ordering change?

Insertion Order Is NOT Maintained



Unlike lists, which keep your objects arranged in the order in which you inserted them, Python's dictionary does **not**. This means you cannot assume that the rows in any dictionary are in any particular order; for all intents and purposes, they are **unordered**.

Take another look at the `person3` dictionary and compare the ordering on input to that shown by the interpreter at the >>> prompt:

You insert your data into a dictionary in one order...

...but the interpreter uses another ordering.

```
>>> person3 = { 'Name': 'Ford Prefect',
                'Gender': 'Male',
                'Occupation': 'Researcher',
                'Home Planet': 'Betelgeuse Seven' }
>>> person3
{'Gender': 'Male', 'Name': 'Ford Prefect', 'Home Planet':
'Betelgeuse Seven', 'Occupation': 'Researcher'}
```

If you're scratching your head and wondering why you'd want to trust your precious data to such an unordered data structure, don't worry, as the ordering rarely makes a difference. When you select data stored in a dictionary, it has nothing to do with the dictionary's order, and everything to do with the key you used. Remember: a key is used to look up a value.

Dictionaries understand square brackets

Like lists, dictionaries understand the square bracket notation. However, unlike lists, which use numeric index values to access data, dictionaries use keys to access their associated data values. Let's see this in action at the interpreter's >>> prompt:

Use keys to access data in a dictionary.

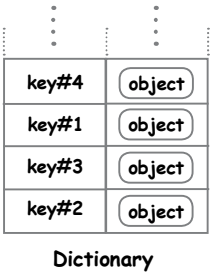
Provide the key between the square brackets.

```
>>> person3['Home Planet']
'Betelgeuse Seven'
>>> person3['Name']
'Ford Prefect'
```

The data value associated with the key is shown.

When you consider you can access your data in this way, it becomes apparent that it does not matter in what order the interpreter stores your data.

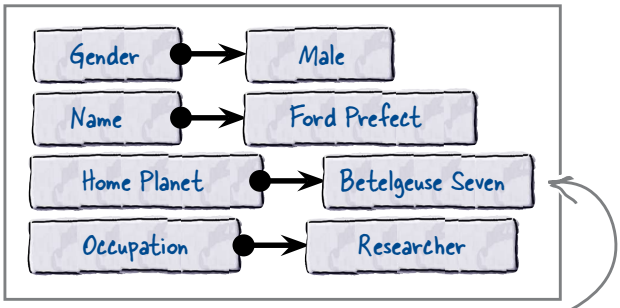
Value Lookup with Square Brackets



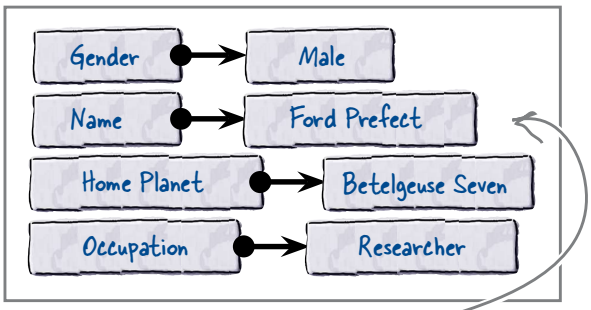
Using square brackets with dictionaries works the same as with lists. However, instead of accessing your data in a specified slot using an index value, with Python's dictionary you access your data via the key associated with it.

As we saw at the bottom of the last page, when you place a key inside a dictionary's square brackets, the interpreter returns the value associated with the key. Let's consider those examples again to help cement this idea in your brain:

```
>>> person3['Home Planet']
'Betelgeuse Seven'
```




```
>>> person3['Name']
'Ford Prefect'
```



Dictionary lookup is fast!

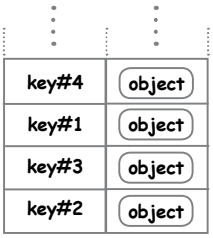
This ability to extract any value from a dictionary using its associated key is what makes Python's dictionary so useful, as there are lots of occasions when doing so is needed—for instance, looking up user details in a profile, which is essentially what we're doing here with the `person3` dictionary.

It does not matter in what order the dictionary is stored. All that matters is that the interpreter can access the value associated with a key *quickly* (no matter how big your dictionary gets). The good news is that the interpreter does just that, thanks to the employment of a highly optimized *hashing algorithm*. As with a lot of Python's internals, you can safely leave the interpreter to handle all the details here, while you get on with taking advantage of what Python's dictionary has to offer.

Geek Bits

Python's dictionary is implemented as a resizeable hash table, which has been heavily optimized for lots of special cases. As a result, dictionaries perform lookups very quickly.

Working with Dictionaries at Runtime



Dictionary

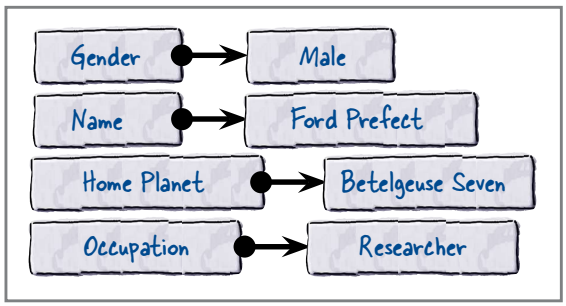
Knowing how the square bracket notation works with dictionaries is central to understanding how dictionaries grow at runtime. If you have an existing dictionary, you can add a new key/value pair to it by assigning an object to a new key, which you provide within square brackets.

For instance, here we display the current state of the `person3` dictionary, then add a new key/value pair that associates `33` with a key called `Age`. We then display the `person3` dictionary again to confirm the new row of data is successfully added:

Before the new row is added

```
>>> person3
{'Name': 'Ford Prefect', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven',
'Occupation': 'Researcher'}
```

Before



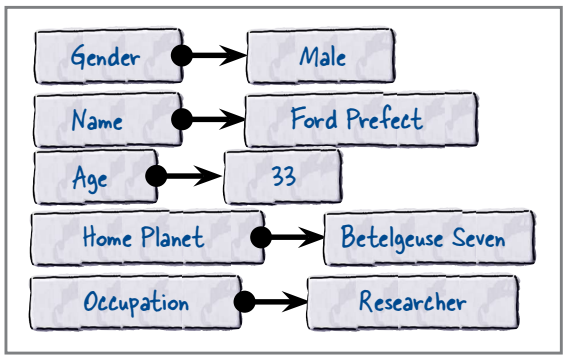

```
>>> person3['Age'] = 33
```

Assign an object (in this case, a number) to a new key to add a row of data to the dictionary.

```
>>> person3
{'Name': 'Ford Prefect', 'Gender': 'Male',
'Age': 33, 'Home Planet': 'Betelgeuse Seven',
'Occupation': 'Researcher'}
```

After the new row is added

Here's the new row of data: "33" is associated with "Age".



After

Recap: Displaying Found Vowels (Lists)

As shown on the last page, growing a dictionary in this way can be used in many different situations. One very common application is to perform a *frequency count*: processing some data and maintaining a count of what you find. Before demonstrating how to perform a frequency count using a dictionary, let's return to our vowel counting example from the last chapter.

Recall that `vowels3.py` determines a unique list of vowels found in a word. Imagine you've now been asked to extend this program to produce output that details how many times each vowel appears in the word.

Here's the code from Chapter 2, which, given a word, displays a unique list of found vowels:

This is "vowels3.py", which reports on the unique vowels found in a word.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

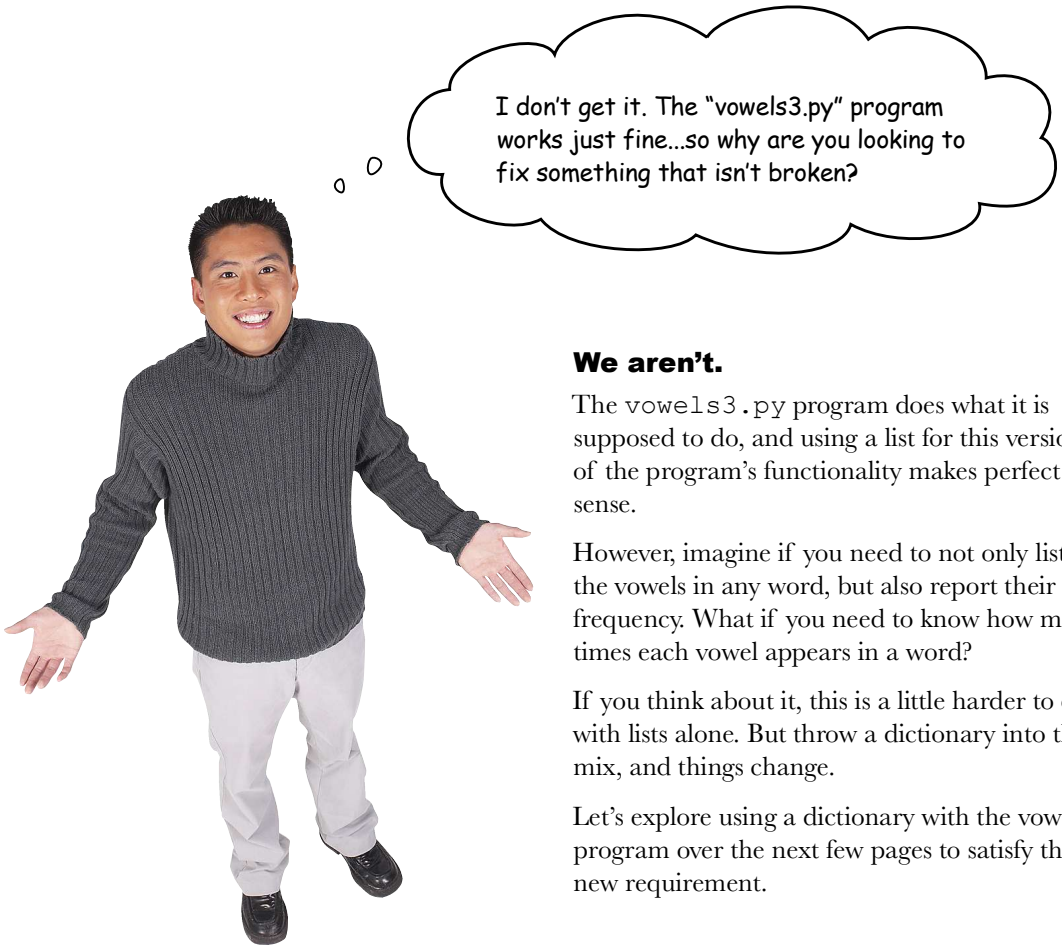
Recall that we ran this code through IDLE a number of times:

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Milliways
i
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Hitch-hiker
i
e
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Galaxy
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Sky
>>>
```

Ln: 21 Col: 4

How Can a Dictionary Help Here?



We aren't.

The `vowels3.py` program does what it is supposed to do, and using a list for this version of the program's functionality makes perfect sense.

However, imagine if you need to not only list the vowels in any word, but also report their frequency. What if you need to know how many times each vowel appears in a word?

If you think about it, this is a little harder to do with lists alone. But throw a dictionary into the mix, and things change.

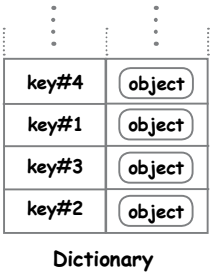
Let's explore using a dictionary with the vowels program over the next few pages to satisfy this new requirement.

there are no Dumb Questions

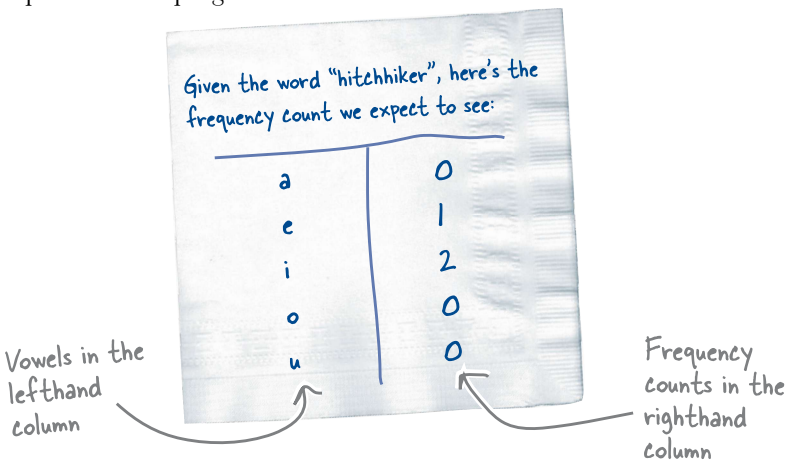
Q: Is it just me, or is the word "dictionary" a strange name for something that's basically a table?

A: No, it's not just you. The word "dictionary" is what the Python documentation uses. In fact, most Python programmers use the shorter "dict" as opposed to the full word. In its most basic form, a dictionary is a table that has exactly two columns and any number of rows.

Selecting a Frequency Count Data Structure



We want to adjust the `vowels3.py` program to maintain a count of how often each vowel is present in a word; that is, what is each vowel's frequency? Let's sketch out what we expect to see as output from this program:



This output is a perfect match with how the interpreter regards a dictionary. Rather than using a list to store the found vowels (as is the case in `vowels3.py`), let's use a dictionary instead. We can continue to call the collection `found`, but we need to initialize it to an empty dictionary as opposed to an empty list.

As always, let's experiment and work out what we need to do at the >>> prompt, before committing any changes to the `vowels3.py` code. To create an empty dictionary, assign `{}` to a variable:

```
>>> found = {}
>>> found
{}
```

Curly braces on their own mean the dictionary starts out empty.

Let's record the fact that we haven't found any vowels yet by creating a row for each vowel and initializing its associated value to `0`. Each vowel is used as a key:

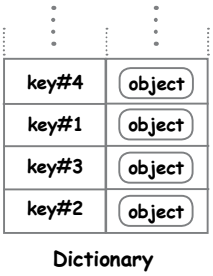
```
>>> found['a'] = 0
>>> found['e'] = 0
>>> found['i'] = 0
>>> found['o'] = 0
>>> found['u'] = 0
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
```

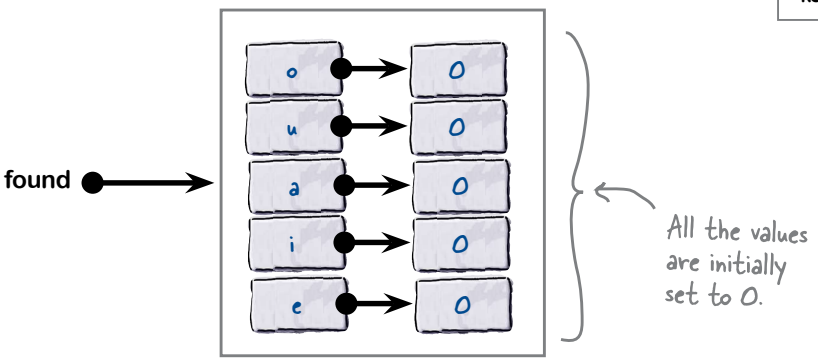
We've initialized all the vowel counts to 0. Note how insertion order is not maintained (but that doesn't matter here).

All we need to do now is find a vowel in a given word, then update these frequency counts as required.

Updating a Frequency Counter

Before getting to the code that updates the frequency counts, consider how the interpreter sees the `found` dictionary in memory after the dictionary initialization code executes:





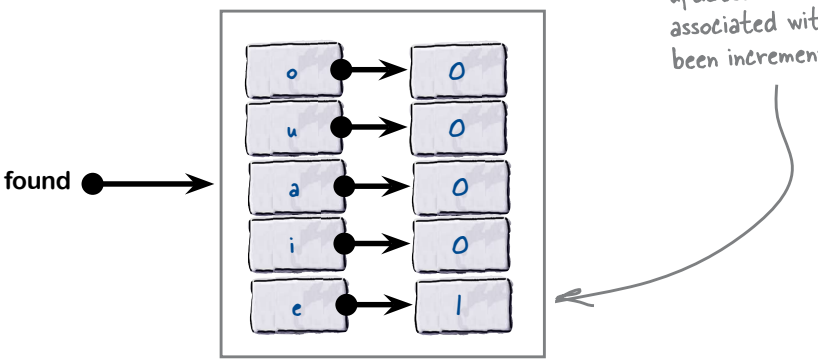
With the frequency counts initialized to 0, it's not difficult to increment any particular value, as needed. For instance, here's how to increment `e`'s frequency count:

```
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
>>> found['e'] = found['e'] + 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 1}
```

Everything is 0.

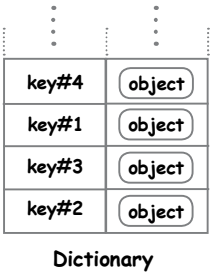
Increment e's count.

The dictionary has been updated. The value associated with "e" has been incremented.



Code like that highlighted above certainly works, but having to repeat `found['e']` on either side of the assignment operator gets very old, very quickly. So, let's look at a shortcut for this operation (on the next page).

Updating a Frequency Counter, v2.0



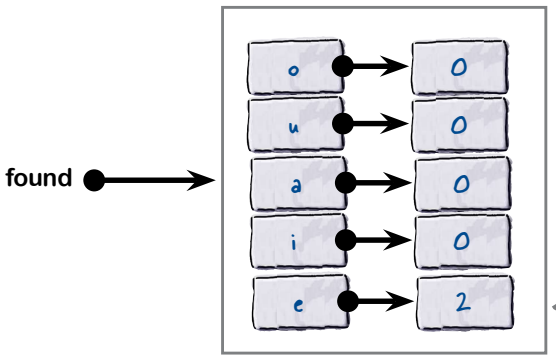
Having to put `found['e']` on either side of the assignment operator (=) quickly becomes tiresome, so Python supports the familiar += operator, which does the same thing, but in a more succinct way:

```
>>> found['e'] += 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 2}
```

Increment e's count (once more).

The dictionary is updated again.

At this point, we've incremented the value associated with the `e` key twice, so here's how the dictionary looks to the interpreter now:



Thanks to the += operator, the value associated with the 'e' key has been incremented once more.

there are no Dumb Questions

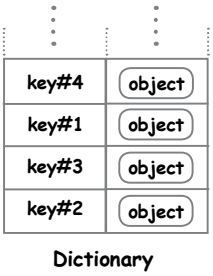
Q: Does Python have ++?

A: No...which is a bummer. If you're a fan of the ++ increment operator in other programming languages, you'll just have to get used to using += instead. Same goes for the −− decrement operator: Python doesn't have it. You need to use −= instead.

Q: Is there a handy list of operators?

A: Yes. Head over to *https://docs.python.org/3/reference/lexical_analysis.html#operators* for a list, and then see *https://docs.python.org/3/library/stdtypes.html* for a detailed explanation of their usage in relation to Python's built-in types.

Iterating Over a Dictionary



Dictionary

At this point, we've shown you how to initialize a dictionary with zeroed data, as well as update a dictionary by incrementing a value associated with a key. We're nearly ready to update the `vowels3.py` program to perform a frequency count based on vowels found in a word. However, before doing so, let's determine what happens when we iterate over a dictionary, as once we have the dictionary populated with data, we'll need a way to display our frequency counts on screen.

You'd be forgiven for thinking that all we need to do here is use the dictionary with a `for` loop, but doing so produces unexpected results:

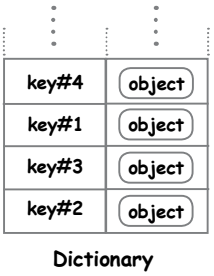
We iterate over the dictionary in the usual way, using a "for" loop. Here, we're using "kv" as shorthand for "key/value pair" (but could've used any variable name).

```
>>> for kv in found:
        print(kv)

o
i
a
u
e
```

The iteration worked, but this isn't what we were expecting. Where have the frequency counts gone? This output is only showing the keys...

Flip the page to learn what happened to the values.

Iterating Over Keys and Values



When you iterated over a dictionary with your `for` loop, the interpreter only processed the dictionary's keys.

To access the associated data values, you need to put each key within square brackets and use it together with the dictionary name to gain access to the values associated with the key.

The version of the loop shown below does just that, providing not just the keys, but also their associated data values. We've changed the suite to access each value based on each key provided to the `for` loop.

As the `for` loop iterates over each key/value pair in the dictionary, the current row's key is assigned to `k`, then `found[k]` is used to access its associated value. We've also produced more human-friendly output by passing two strings to the call to the `print` function:

We're using "k" to represent the key, and "found[k]" to access the value.

```
>>> for k in found:
        print(k, 'was found', found[k], 'time(s).')

o was found 0 time(s).
i was found 0 time(s).
a was found 0 time(s).
u was found 0 time(s).
e was found 2 time(s).
```

This is more like it. The keys and the values are being processed by the loop and displayed on screen.

If you are following along at your >>> prompt and your output is ordered differently from ours, don't worry: the interpreter uses a random internal ordering as you're using a dictionary here, and there are no guarantees regarding ordering when one is used. Your ordering will likely differ from ours, but don't be alarmed. Our primary concern is that the data is safely stored in the dictionary, which it is.

The above loop *obviously* works. However, there are two points that we'd like to make.

Firstly: it would be nice if the output was ordered `a`, `e`, `i`, `o`, `u`, as opposed to randomly, wouldn't it?

Secondly: even though this loop clearly works, coding a dictionary iteration in this way is not the preferred approach—most Python programmers code this differently.

Let's explore these two points in a bit more detail (after a quick review).

Dictionaries: What We Already Know

Here's what we know about Python's dictionary data structure so far:

BULLET POINTS

- Think of a dictionary as a collection of rows, with each row containing exactly two columns. The first column stores a **key**, while the second contains a **value**.
- Each row is known as a **key/value pair**, and a dictionary can grow to contain any number of key/value pairs. Like lists, dictionaries grow and shrink on demand.
- A dictionary is easy to spot: it's enclosed in curly braces, with each key/value pair separated from the next by a comma, and each key separated from its value by a colon.
- Insertion order is *not* maintained by a dictionary. The order in which rows are inserted has nothing to do with how they are stored.
- Accessing data in a dictionary uses the **square bracket notation**. Put a key inside square brackets to access its associated value.
- Python's `for` loop can be used to iterate over a dictionary. On each iteration, the key is assigned to the loop variable, which is used to access the data value.

Specifying the ordering of a dictionary on output

We want to be able to produce output from the `for` loop in `a`, `e`, `i`, `o`, `u` order as opposed to randomly. Python makes this trivial thanks to the inclusion of the `sorted` built-in function. Simply pass the `found` dictionary to the `sorted` function as part of the `for` loop to arrange the output alphabetically:

```
>>> for k in sorted(found):
        print(k, 'was found', found[k], 'time(s).')

a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

It's a small change to the loop's code, but... it packs quite the punch. Look: the output is sorted in "a, e, i, o, u" order.

That's point one of two dealt with. Next up is learning about the approach that most Python programmers *prefer* over the above code (although the approach shown on this page is often used, so you still need to know about it).

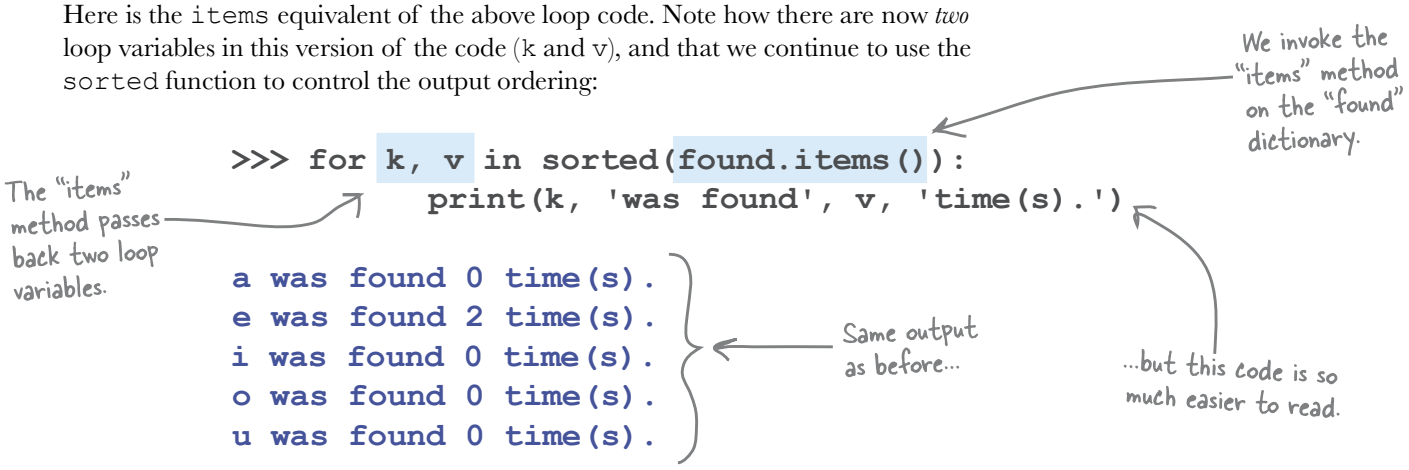
Iterating Over a Dictionary with "items"

We've seen that it's possible to iterate over the rows of data in a dictionary using this code:

```
>>> for k in sorted(found):
        print(k, 'was found', found[k], 'time(s).')

a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

Like lists, dictionaries have a bunch of built-in methods, and one of these is the `items` method, which returns a list of the key/value pairs. Using `items` with `for` is often the preferred technique for iterating over a dictionary, as it gives you access to the key *and* the value as loop variables, which you can then use in your suite. The resulting suite is easier on the eye, which makes it easier to read.

Here is the `items` equivalent of the above loop code. Note how there are now *two* loop variables in this version of the code (`k` and `v`), and that we continue to use the `sorted` function to control the output ordering:



there are no Dumb Questions

Q: Why are we calling `sorted` again in the second loop? The first loop arranged the dictionary in the ordering we want, so this must mean we don't have to sort it a second time, right?

A: No, not quite. The `sorted` built-in function doesn't change the ordering of the data you provide to it, but instead returns an **ordered copy** of the data. In the case of the `found` dictionary, this is an ordered copy of each key/value pair, with the key being used to determine the ordering (alphabetical, from A through Z). The original ordering of the dictionary remains intact, which means every time we need to iterate over the key/value pairs in some specific order, we need to call `sorted`, as the random ordering still exists in the dictionary.

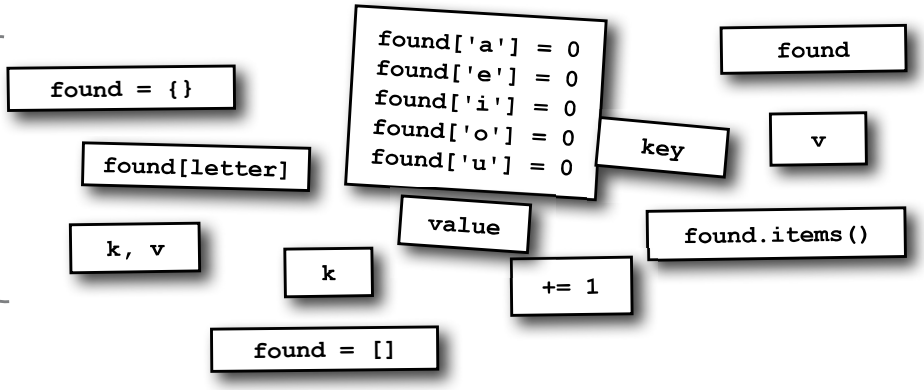
Frequency Count Magnets

Having concluded our experimentation at the >>> prompt, it's now time to make changes to the `vowels3.py` program. Below are all of the code snippets we think you might need. Your job is to rearrange the magnets to produce a working program that, when given a word, produces a frequency count for each vowel found.

Decide which code magnet goes in each of the dashed-line locations to create "vowels4.py".

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
.....
.....
.....
.....
.....
.....
.....
for letter in word:
    if letter in vowels:
        ..... .....

for ..... in sorted( ..... ):
    print( ..... , 'was found', ..... , 'time(s).')
```

Where do all these go? Be careful: not all these magnets are needed.



Once you've placed the magnets where you think they should go, bring `vowels3.py` into IDLE's edit window, rename it `vowels4.py`, and then apply your code changes to the new version of this program.

Frequency Count Magnets Solution

Having concluded our experimentation at the >>> prompt, it was time to make changes to the `vowels3.py` program. Your job was to rearrange the magnets to produce a working program that, when given a word, produces a frequency count for each vowel found.

Once you'd placed the magnets where you thought they should go, you were to bring `vowels3.py` into an IDLE's edit window, rename it `vowels4.py`, and then apply your code changes to the new version of this program.

This is the "vowels4.py" program.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = {}
found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0
for letter in word:
    if letter in vowels:
        found[letter] += 1
for k, v in sorted( found.items() ):
    print( k , 'was found', v , 'time(s).')
```

Create an empty dictionary.

Initialize the value associated with each of the keys (each vowel) to 0.

Increment the value referred to by "found[letter]" by one.

As the "for" loop is using the "items" method, we need to provide two loop variables, "k" for the key and "v" for the value.

Invoke the "items" method on the "found" dictionary to access each row of data with each iteration.

The key and the value are used to create each output message.

These magnets weren't needed.

found

key

value

found = []

Test Drive

Let's take `vowels4.py` for a spin. With your code in an IDLE edit window, press F5 to see how it performs:

The "vowels4.py" code →

vowels4.py - /Users/Paul/Desktop/_NewBook/ch02/vowels4.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

We ran the code three times to see how well it performs.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
a was found 0 time(s).
e was found 1 time(s).
i was found 2 time(s).
o was found 0 time(s).
u was found 0 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
a was found 1 time(s).
e was found 6 time(s).
i was found 3 time(s).
o was found 0 time(s).
u was found 1 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: sky
a was found 0 time(s).
e was found 0 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
>>>
```

These three "runs" produce the output we expect them to.

Just How Dynamic Are Dictionaries?

The `vowels4.py` program reports on all the found vowels, even when they aren't found. This may not bother you, but let's imagine that it does and you want this code to only display results when results are *actually* found. That is, you don't want to see any of those "found 0 time(s)" messages.

How might you go about solving this problem?

That sounds like it might work.

We currently have five lines of code near the start of the `vowels4.py` program that we've included in order to *initially* set each vowel's frequency count to `0`. This creates a key/value pair for each vowel, even though some may never be used. If we take those five lines away, we should end up only recording frequency counts for found vowels, and ignore the rest.

Let's give this idea a try.

Do this!

Take the code in `vowels4.py` and save it as `vowels5.py`. Then remove the five lines of initialization code. Your IDLE edit window should look like that on the right of this page.

This is the "vowels5.py" code with the initialization code removed.

vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Ln: 13 Col: 0

Test Drive

You know the drill. Make sure `vowels5.py` is in an IDLE edit window, then press F5 to run your program. You'll be confronted by a runtime error message:

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitchhiker
Traceback (most recent call last):
  File "/Users/Paul/Desktop/_NewBook/ch03/vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
                                                                  Ln: 11 Col: 0
```

This can't be good.

It's clear that removing the five lines of initialization code wasn't the way to go here. But why has this happened? The fact that Python's dictionary grows dynamically at runtime should mean that this code *cannot* crash, but it does. Why are we getting this error?

Dictionary keys must be initialized

Removing the initialization code has resulted in a runtime error, specifically a `KeyError`, which is raised when you try to access a value associated with a nonexistent key. Because the key can't be found, the value associated with it can't be found either, and you get an error.

Does this mean that we have to put the initialization code back in? After all, it is only five short lines of code, so what's the harm? We can certainly do this, but let's think about doing so for a moment.

Imagine that, instead of five frequency counts, you have a requirement to track a thousand (or more). Suddenly, we have *lots* of initialization code. We could "automate" the initialization with a loop, but we'd still be creating a large dictionary with lots of rows, many of which may end up never being used.

If only there were a way to create a key/value pair on the fly, just as soon as we realize we need it.

Geek Bits

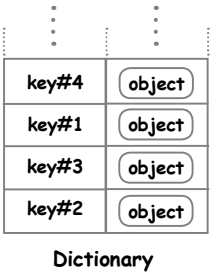
An alternative approach to handling this issue is to deal with the run-time exception raised here (which is a "KeyError" in this example). We're holding off talking about how Python handles run-time exceptions until a later chapter, so bear with us for now.

That's a great question.

We first met `in` when checking lists for a value. Maybe `in` works with dictionaries, too?

Let's experiment at the >>> prompt to find out.

Avoiding KeyErrors at Runtime



As with lists, it is possible to use the `in` operator to check whether a key exists in a dictionary; the interpreter returns `True` or `False` depending on what's found.

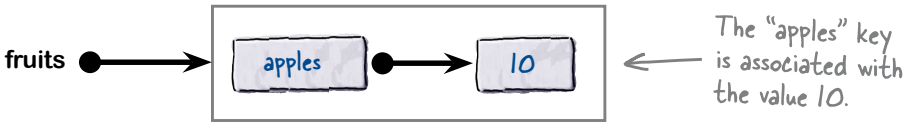
Let's use this fact to avoid that `KeyError` exception, because it can be annoying when your code stops as a result of this error being raised during an attempt to populate a dictionary at runtime.

To demonstrate this technique, we're going to create a dictionary called `fruits`, then use the `in` operator to avoid raising a `KeyError` when accessing a nonexistent key. We start by creating an empty dictionary; then we assign a key/value pair that associates the value `10` with the key `apples`. With the row of data in the dictionary, we can use the `in` operator to confirm that the key `apples` now exists:

```
>>> fruits
{}
>>> fruits['apples'] = 10
>>> fruits
{'apples': 10}
>>> 'apples' in fruits
True
```

This is all as expected. The value is associated with the key, and there's no runtime error when we use the "in" operator to check for the key's existence.

Before we do anything else, let's consider how the interpreter views the `fruits` dictionary in memory after executing the above code:

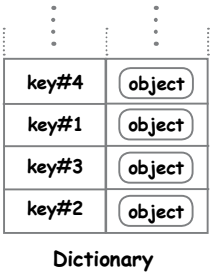


there are no Dumb Questions

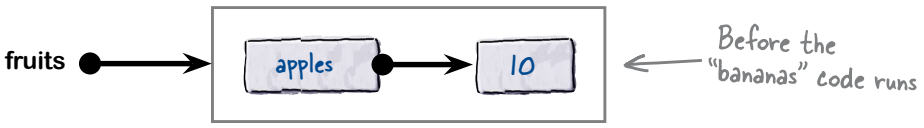
Q: I take it from the example on this page that Python uses the constant value `True` for *true*? Is there a `False`, too, and does case matter when using either of these values?

A: Yes, to all those questions. When you need to specify a boolean in Python, you can use either `True` or `False`. These are constant values provided by the interpreter, and must be specified with a leading uppercase letter, as the interpreter treats `true` and `false` as variable names, *not* boolean values, so care is needed here.

Checking for Membership with "in"



Let's add in another row of data to the `fruits` dictionary for `bananas` and see what happens. However, instead of a straight assignment to `bananas`, (as was the case with `apples`), let's increment the value associated with `bananas` by `1` if it already exists in the `fruits` dictionary or, if it doesn't exist, let's initialize `bananas` to `1`. This is a very common activity, especially when you're performing frequency counts using a dictionary, and the logic we employ should hopefully help us avoid a `KeyError`.



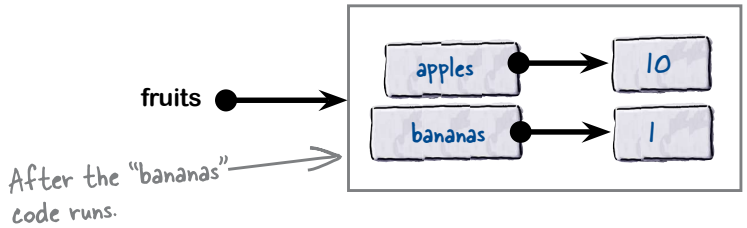
In the code that follows, the `in` operator in conjunction with an `if` statement avoids any slip-ups with `bananas`, which—as wordplays go—is pretty bad (even for us):

```
>>> if 'bananas' in fruits:
      fruits['bananas'] += 1
else:
      fruits['bananas'] = 1

>>> fruits
{'bananas': 1, 'apples': 10}
```

We check to see if the "bananas" key is in the dictionary, and as it isn't, we initialize its value to 1. Critically, we avoid any possibility of a "KeyError".

We've set the "bananas" value to 1.

The above code changes the state of the `fruits` dictionary within the interpreter's memory, as shown here:



As expected, the `fruits` dictionary has grown by one key/value pair, and the `bananas` value has been initialized to `1`. This happened because the condition associated with the `if` statement evaluated to `False` (as the key wasn't found), so the second suite (that is, the one associated with `else`) executed instead. Let's see what happens when this code runs again.

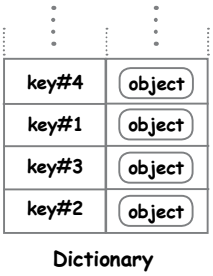
Geek Bits

If you are familiar with the **?:** **ternary operator** from other languages, note that Python supports a similar construct. You can say this:

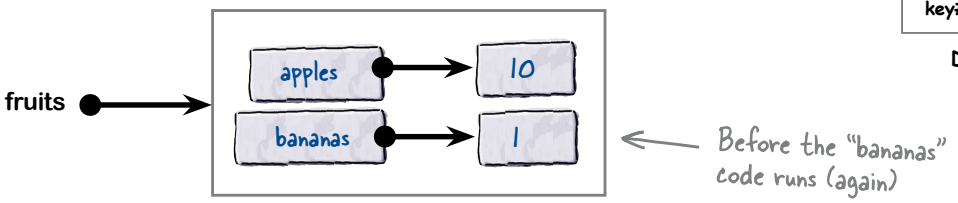
```
x = 10 if y > 3 else 20
```

to set **x** to either **10** or **20** depending on whether or not the value of **y** is greater than **3**. That said, most Python programmers frown on its use, as the equivalent **if... else...** statements are considered easier to read.

Ensuring Initialization Before Use



If we execute the code again, the value associated with `bananas` should now be increased by 1, as the `if` suite executes this time due to the fact that the `bananas` key already exists in the `fruits` dictionary:



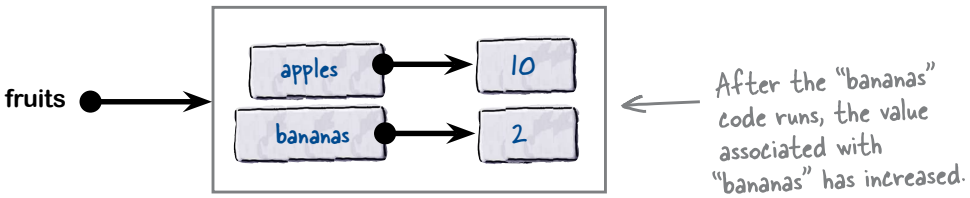
To run this code again, press *Ctrl-P* (on a Mac) or *Alt-P* (on Linux/Windows) to cycle back through your previously entered code statements while at IDLE's >>> prompt (as using the up arrow to recall input doesn't work at IDLE's >>> prompt). Remember to press Enter *twice* to execute the code once more:

```
>>> if 'bananas' in fruits:
      fruits['bananas'] += 1
else:
      fruits['bananas'] = 1

>>> fruits
{'bananas': 2, 'apples': 10}
```

This time around, the "bananas" key does exist in the dictionary, so we increment its value by 1. As before, our use of "if" and "in" together stop a "KeyError" exception from crashing this code.

We've increased the "bananas" value by 1.

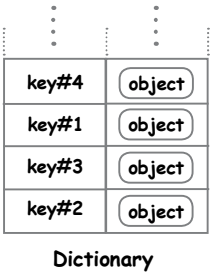
As the code associated with the `if` statement now executes, the value associated with `bananas` is incremented within the interpreter's memory:



This mechanism is so common that many Python programmers shorten these four lines of code by inverting the condition. Instead of checking with `in`, they use `not in`. This allows you to initialize the key to a starter value (usually `0`) if it isn't found, then perform the increment right after.

Let's take a look at how this mechanism works.

Substituting "not in" for "in"



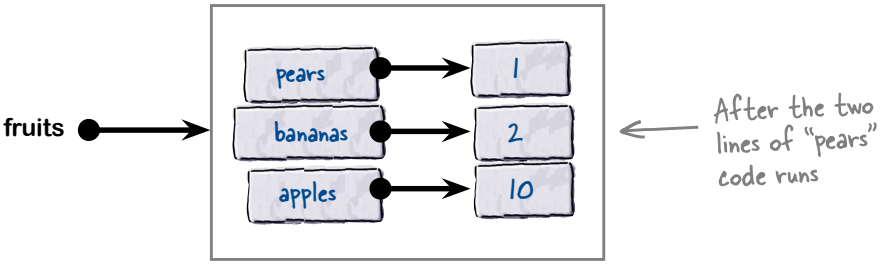
At the bottom of the last page, we stated that most Python programmers refactor the original four lines of code to use `not in` instead of `in`. Let's see this in action by using this mechanism to ensure the `pears` key is set to `0` before we try to increment its value:

```
>>> if 'pears' not in fruits:
        fruits['pears'] = 0

>>> fruits['pears'] += 1
>>> fruits
{'bananas': 2, 'pears': 1, 'apples': 10}
```

Initialize (if needed).

Increment.

These three lines of code have grown the dictionary once more. There are now three key/value pairs in the `fruits` dictionary:



The above three lines of code are so common in Python that the language provides a dictionary method that makes this `if/not in` combination more convenient and less error prone. The `setdefault` method does what the two-line `if/not in` statements do, but uses only a *single* line of code.

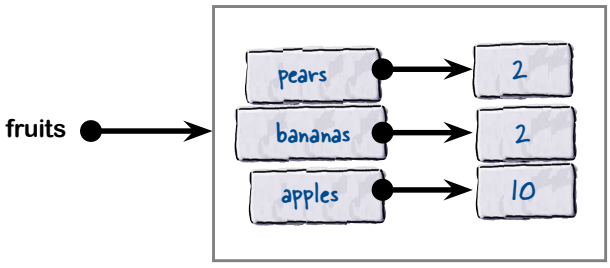
Here's the equivalent of the `pears` code from the top of the page rewritten to use `setdefault`:

```
>>> fruits.setdefault('pears', 0)
>>> fruits['pears'] += 1
>>> fruits
{'bananas': 2, 'pears': 2, 'apples': 10}
```

Initialize (if needed).

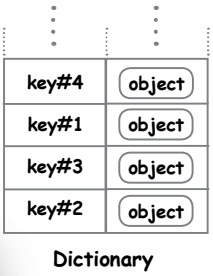
Increment.

The single call to `setdetfault` has replaced the two-line `if/not in` statement, and its usage guarantees that a key is always initialized to a starter value before it's used. Any possibility of a `KeyError` exception is negated. The current state of the `fruits` dictionary is shown here (on the right) to confirm that invoking `setdefault` after a key already exists has no effect (as is the case with `pears`), which is exactly what we want in this case.



Putting the "setdefault" Method to Work

Recall that our current version of `vowels5.py` results in a runtime error, specifically a `KeyError`, which is raised due to our code trying to access the value of a nonexistent key:



This code produces this error.

vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitchhiker
Traceback (most recent call last):
  File "/Users/Paul/Desktop/_NewBook/ch03/vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
```

Ln: 11 Col: 0

From our experiments with `fruits`, we know we can call `setdefault` as often as we like without having to worry about any nasty errors. We know `setdefault`'s behavior is guaranteed to initialize a nonexistent key to a supplied default value, or to do nothing (that is, to leave any existing value associated with any existing key alone). If we invoke `setdefault` immediately before we try to use a key in our `vowels5.py` code, we are guaranteed to avoid a `KeyError`, as the key will either exist or it won't. Either way, our program keeps running and no longer crashes (thanks to our use of `setdefault`).

Use "setdefault" to help avoid the "KeyError" exception.

Within your IDLE edit window, change the first of the `vowels5.py` program's `for` loops to look like this (by adding the call to `setdefault`), then save your new version as `vowels6.py`:

```
for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1
```

A single line of code can often make all the difference.

Test Drive

With the most recent `vowels6.py` program in your IDLE edit window, press F5. Run this version a few times to confirm the nasty `KeyError` exception no longer appears.

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
e was found 1 time(s).
i was found 2 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
a was found 1 time(s).
e was found 6 time(s).
i was found 3 time(s).
u was found 1 time(s).
>>>
                                                              Ln: 23 Col: 4
```

This is looking good. The "KeyError" is gone.

The use of the `setdefault` method has solved the `KeyError` problem we had with our code. Using this technique allows you to dynamically grow a dictionary at runtime, safe in the knowledge that you'll only ever create a new key/value pair when you actually need one.

When you use `setdefault` in this way, you **never** need to spend time initializing all your rows of dictionary data ahead of time.

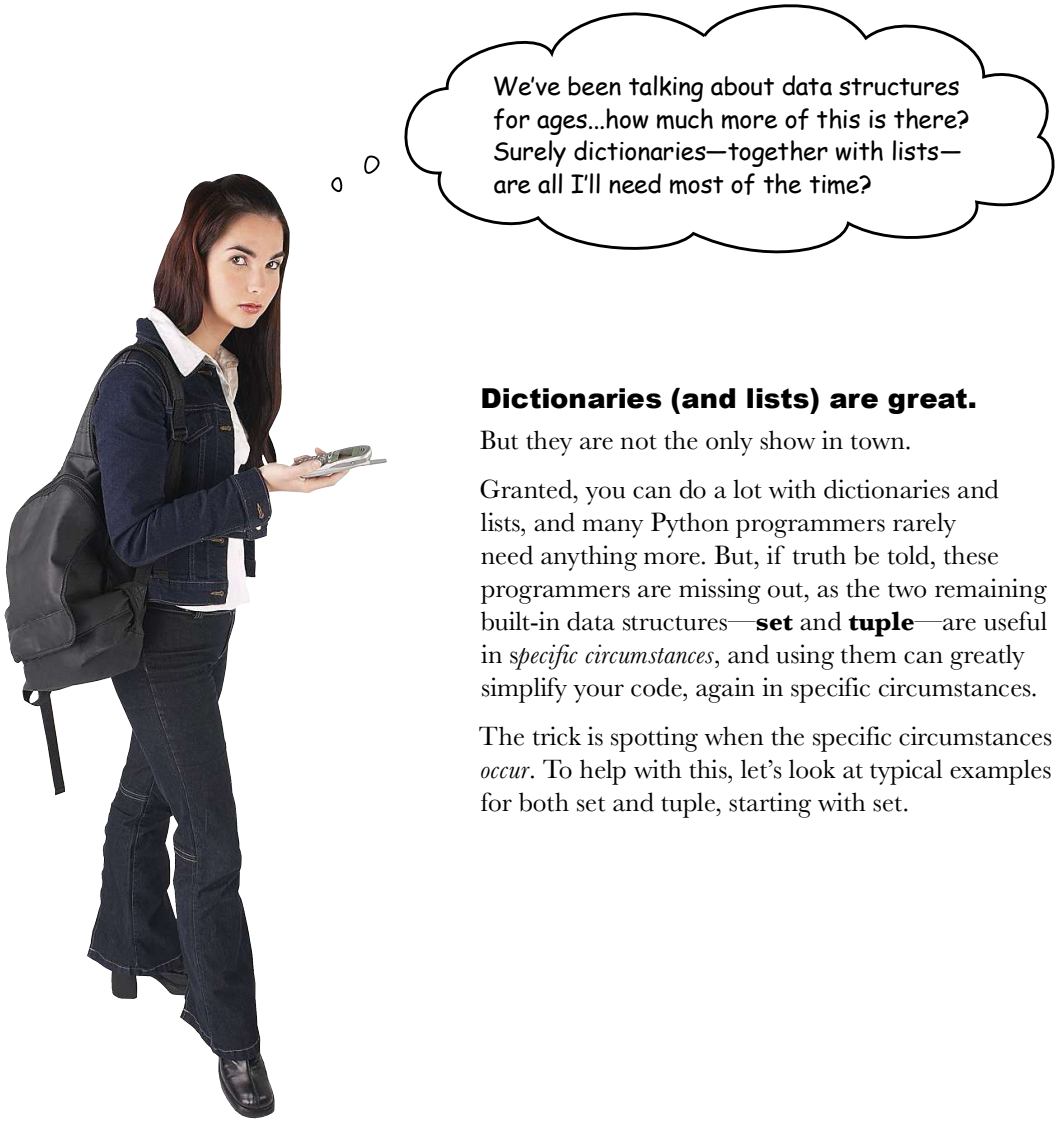
Dictionaries: updating what we already know

Let's add to the list of things you now know about Python's dictionary:

BULLET POINTS

- By default, every dictionary is unordered, as insertion order is not maintained. If you need to sort a dictionary on output, use the `sorted` built-in function.
- The `items` method allows you to iterate over a dictionary by row—that is, by key/value pair. On each iteration, the `items` method returns the next key and its associated value to your `for` loop.
- Trying to access a nonexistent key in an existing dictionary results in a `KeyError`. When a `KeyError` occurs, your program crashes with a runtime error.
- You can avoid a `KeyError` by ensuring every key in your dictionary has a value associated with it before you try to access it. Although the `in` and `not in` operators can help here, the established technique is to use the `setdefault` method instead.

Aren't Dictionaries (and Lists) Enough?



Dictionaries (and lists) are great.

But they are not the only show in town.

Granted, you can do a lot with dictionaries and lists, and many Python programmers rarely need anything more. But, if truth be told, these programmers are missing out, as the two remaining built-in data structures—**set** and **tuple**—are useful in s*pecific circumstances*, and using them can greatly simplify your code, again in specific circumstances.

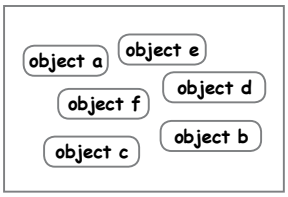
The trick is spotting when the specific circumstances *occur*. To help with this, let's look at typical examples for both set and tuple, starting with set.

there are no Dumb Questions

Q: Is that it for dictionaries? Surely it's common for the value part of a dictionary to be, for instance, a list or another dictionary?

A: Yes, that is a common usage. But we're going to hang on until the end of this chapter to show you how to do this. In the meantime, let what you already know about dictionaries sink in...

Sets Don't Allow Duplicates



Set

Python's **set** data structure is just like the sets you learned about in school: it has certain mathematical properties that always hold, the key characteristic being that *duplicate values are forbidden*.

Imagine you are provided with a long list of all the first names for everyone in a large organization, but you are only interested in the (much smaller) list of unique first names. You need a quick and foolproof way to remove any duplicates from your long list of names. Sets are great at solving this type of problem: simply convert the long list of names to a set (which removes the duplicates), then convert the set back to a list and—ta da!—you have a list of unique first names.

Python's set data structure is optimized for very speedy lookup, which makes using a set much faster than its equivalent list when lookup is the primary requirement. As lists always perform slow sequential searches, sets should always be preferred for lookup.

Spotting sets in your code

Sets are easy to spot in code: a collection of objects are separated from one another by commas and surrounded by curly braces.

For example, here's a set of vowels:

Sets start and end with a curly brace.

```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'e', 'u', 'a', 'i', 'o'}
```

Check out the ordering. It's changed from what was originally inserted, and the duplicates are gone too.

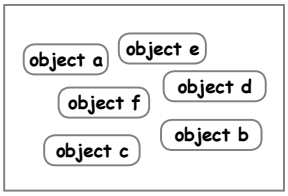
Objects are separated from one another by a comma.

The fact that a set is enclosed in curly braces can often result in your brain mistaking a set for a dictionary, which is *also* enclosed in curly braces. The key difference is the use of the colon character (`:`) in dictionaries to separate keys from values. The colon never appears in a set, only commas.

In addition to forbidding duplicates, note that—as in a dictionary—insertion order is *not* maintained by the interpreter when a set is used. However. like all other data structures, sets can be ordered on output with the `sorted` function. And, like lists and dictionaries, sets can also grow and shrink as needed.

Being a set, this data structure can perform set-like operations, such as *difference*, *intersection*, and *union*. To demonstrate sets in action, we are going to revisit our vowel counting program from earlier in this chapter once more. We made a promise when we were first developing `vowels3.py` (in the last chapter) that we'd consider a set over a list as the primary data structure for that program. Let's make good on that promise now.

Creating Sets Efficiently



Set

Let's take yet another look at `vowels3.py`, which uses a list to work out which vowels appear in any word.

Here's the code once more. Note how we have logic in this program to ensure we only remember each found vowel once. That is, we are very deliberately ensuring that no duplicate vowels are *ever* added to the `found` list:

This is "vowels3.py", which reports on the unique vowels found in a word. This code uses a list as its primary data structure.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

We never allow duplicates in the "found" list.

Before continuing, use IDLE to save this code as `vowels7.py` so that we can make changes without having to worry about breaking our list-based solution (which we know works). As is becoming our standard practice, let's experiment at the >>> prompt first before adjusting the `vowels7.py` code. We'll edit the code in the IDLE edit window once we've worked out the code we need.

Creating sets from sequences

We start by creating a set of vowels using the code from the middle of the last page (you can skip this step if you've already typed that code into your >>> prompt):

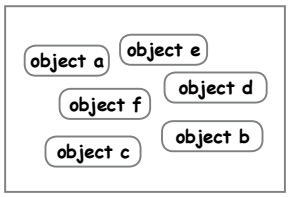
```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'e', 'u', 'a', 'i', 'o'}
```

These two lines of code do the same thing: both assign a new set object to a variable.

Below is a useful shorthand that allows you to pass any sequence (such as a string) to the `set` function to quickly generate a set. Here's how to create the set of vowels using the `set` function:

```
>>> vowels2 = set('aeeiouu')
>>> vowels2
{'e', 'u', 'a', 'i', 'o'}
```

Taking Advantage of Set Methods



Set

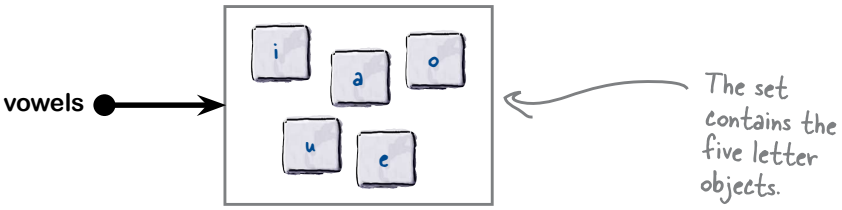
Now that we have our vowels in a set, our next step is to take a word and determine whether any of the letters in the word are vowels. We could do this by checking whether each letter in the word is in the set, as the `in` operator works with sets in much the same way as it does with dictionaries and lists. That is, we could use `in` to determine whether a set contains any letter, and then cycle through the letters in the word using a `for` loop.

However, let's not follow that strategy here, as the set methods can do a lot of this looping work for us.

There's a much better way to perform this type of operation when using sets. It involves taking advantage of the methods that come with every set, and that allow you to perform operations such as union, difference, and intersection. Prior to changing the code in `vowels7.py`, let's learn how these methods work by experimenting at the >>> prompt and considering how the interpreter sees the set data. Be sure to follow along on your computer. Let's start by creating a set of vowels, then assigning a value to the `word` variable:

```
>>> vowels = set('aeiou')
>>> word = 'hello'
```

The interpreter creates two objects: one set and one string. Here's what the `vowels` set looks like in the interpreter's memory:



Let's see what happens when we perform a union of the `vowels` set and the set of letters created from the value in the `word` variable. We'll create a second set on-the-fly by passing the `word` variable to the `set` function, which is then passed to the `union` method provided by `vowels`. The result of this call is another set, which we assign to another variable (called `u` here). This new variable is a *combination* of the objects in both sets (a union):

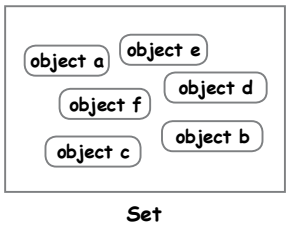
```
>>> u = vowels.union(set(word))
```

Python converts the value in "word" into a set of letter objects (removing any duplicates as it does so).

The "union" method combines one set with another, which is then assigned to a new variable called "u" (which is another set).

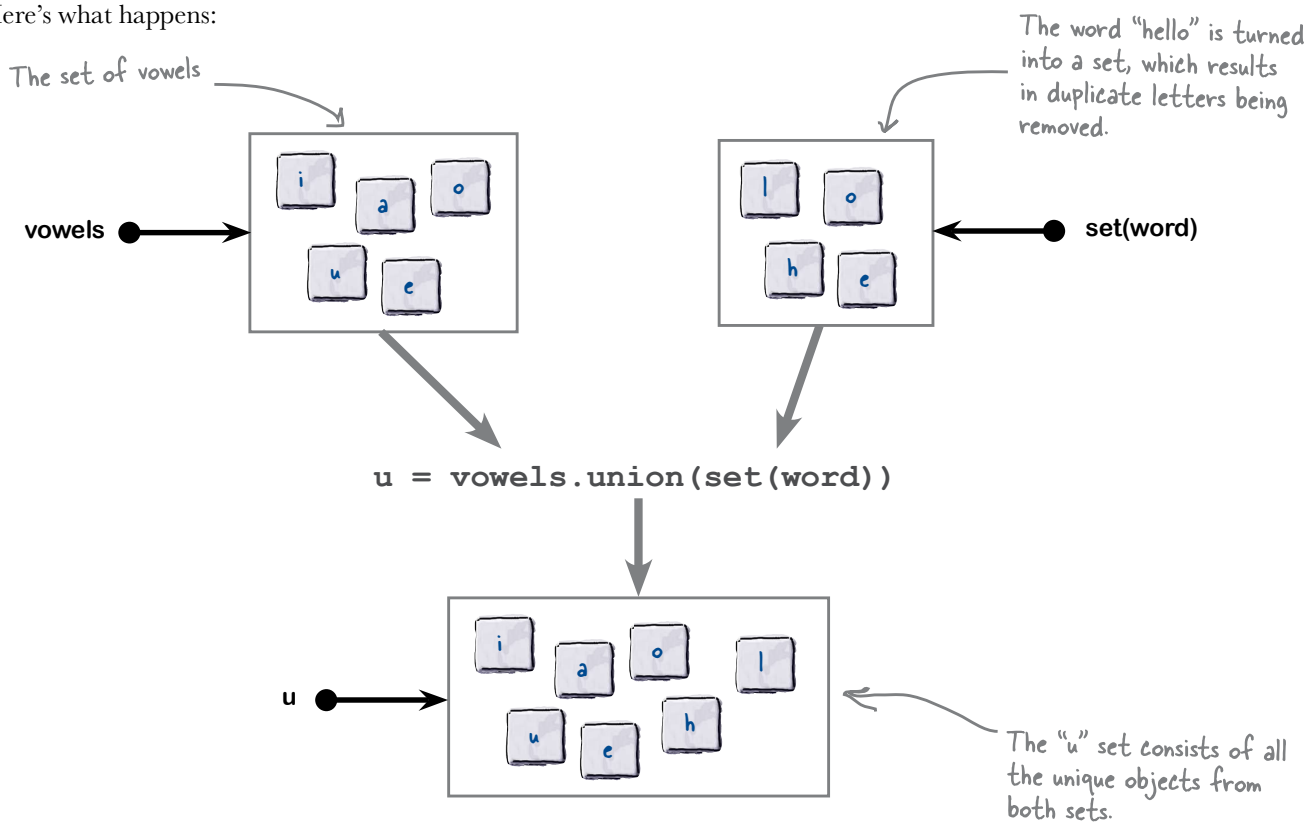
After this call to the `union` method, what do the *`vowels`* and *`u`* sets look like?

union Works by Combining Sets



At the bottom of the previous page we used the `union` method to create a new set called `u`, which was a combination of the letters in the `vowels` set together with the set of unique letters in `word`. The act of creating this new set has no impact on `vowels`, which remains as it was before the union. However, the `u` set is new, as it is created as a result of the union.

Here's what happens:



What happened to the loop code?

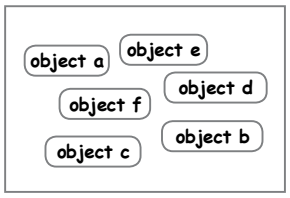
That single line of code packs a lot of punch. Note that you haven't specifically instructed the interpreter to perform a loop. Instead, you told the interpreter *what* you wanted done—not *how* you wanted it done—and the interpreter has obliged by creating a new set containing the objects you're after.

A common requirement (now that we've created the union) is to turn the resulting set into a sorted list. Doing so is trivial, thanks to the `sorted` and `list` functions:

A sorted list of unique letters

```
>>> u_list = sorted(list(u))
>>> u_list
['a', 'e', 'h', 'i', 'l', 'o', 'u']
```

difference Tells You What's Not Shared

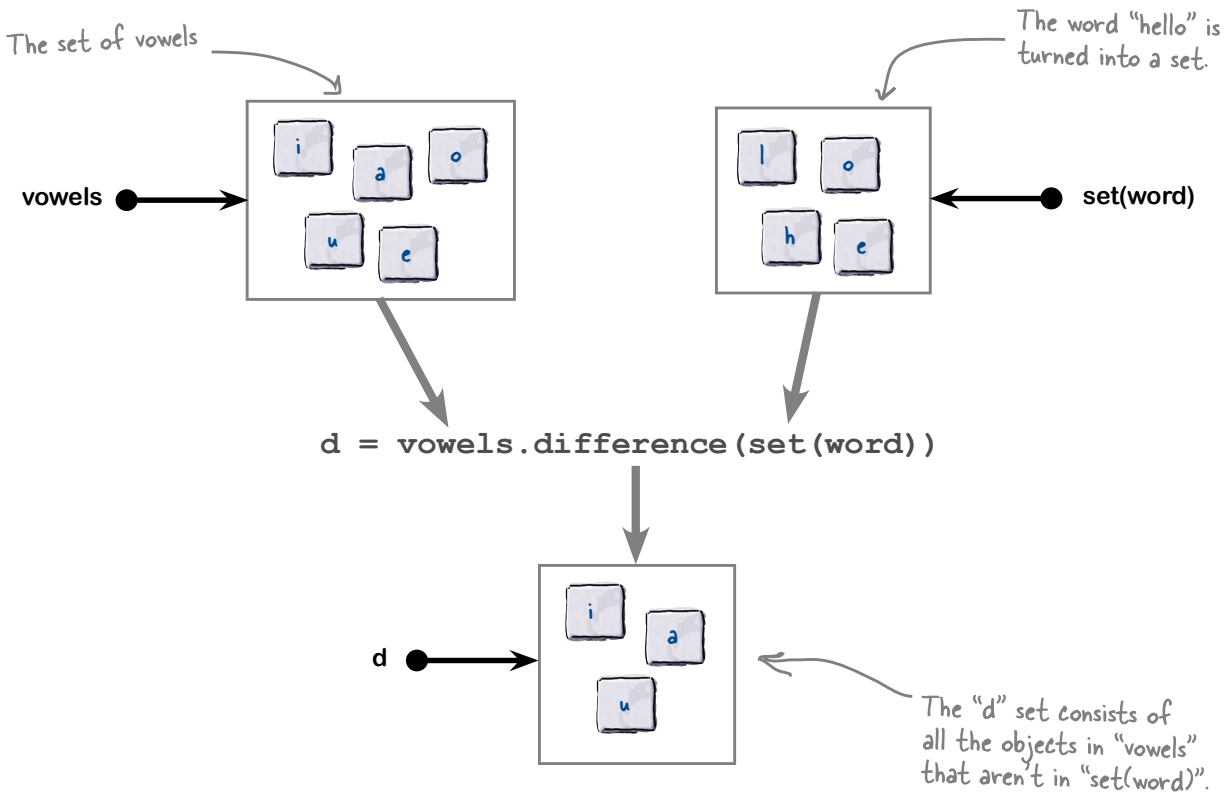


Another set method is `difference`, which, given two sets, can tell you what's in one set but not the other. Let's use `difference` in much the same way as we did with `union` and see what we end up with:

```
>>> d = vowels.difference(set(word))
>>> d
{'u', 'i', 'a'}
```

The `difference` function compares the objects in `vowels` against the objects in `set(word)`, then returns a new set of objects (called `d` here) which are in the `vowels` set but *not* in `set(word)`.

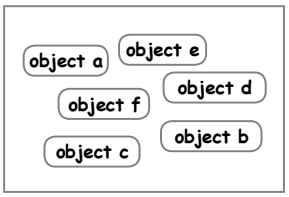
Here's what happens:



We once again draw your attention to the fact that this outcome has been accomplished *without* using a `for` loop. The `difference` function does all the grunt work here; all we did was state what was required.

Flip over to the next page to look at one final set method: `intersection`.

intersection Reports on Commonality



Set

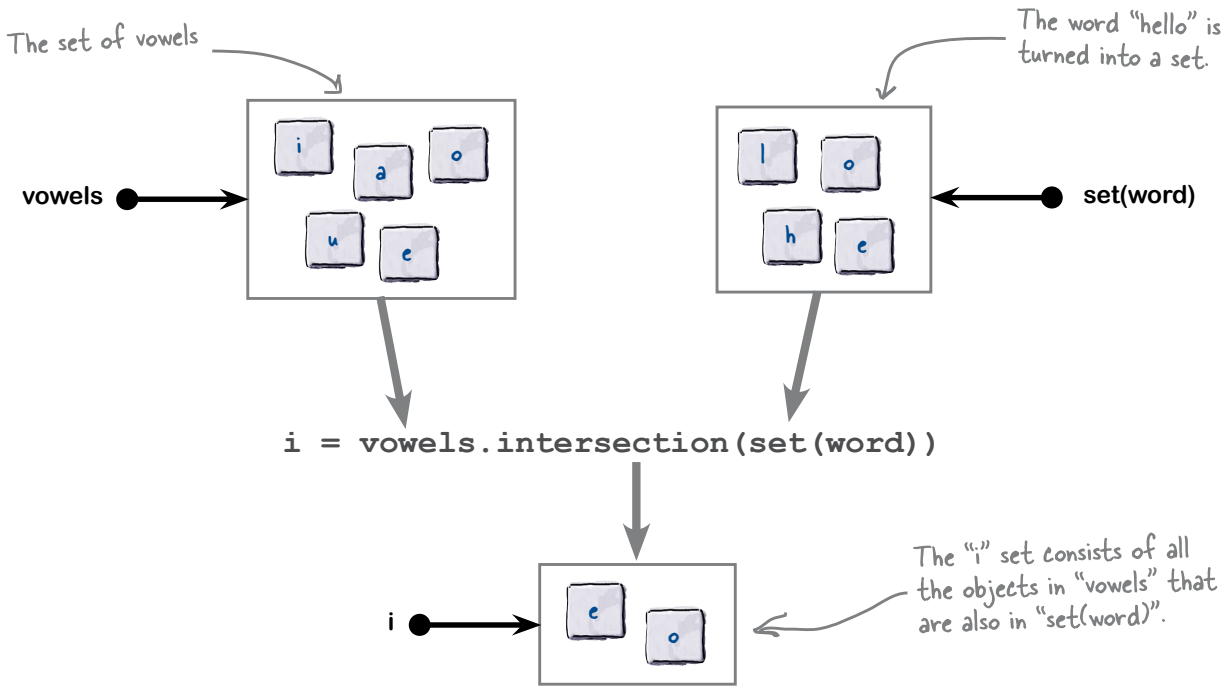
The third set method that we'll look at is `intersection`, which takes the objects in one set and compares them to those in another, then reports on any common objects found.

In relation to the requirements that we have with `vowels7.py`, what the `intersection` method does sounds very promising, as we want to know which of the letters in the user's word are vowels.

Recall that we have the string `"hello"` in the `word` variable, and our vowels in the `vowels` set. Here's the `intersection` method in action:

```
>>> i = vowels.intersection(set(word))
>>> i
{'e', 'o'}
```

The `intersection` method confirms the vowels `e` and `o` are in the `word` variable. Here's what happens:



There are more set methods than the three we've looked at over these last few pages, but of the three, `intersection` is of most interest to us here. In a single line of code, we've solved the problem we posed near the start of the last chapter: *identify the vowels in any string*. And all without having to use any loop code. Let's return to the `vowels7.py` program and apply what we know now.

Sets: What You Already Know

Here's a quick rundown of what you already know about Python's set data structure:

BULLET POINTS

- Sets in Python do not allow duplicates.
- Like dictionaries, sets are enclosed in curly braces, but sets do not identify key/value pairs. Instead, each unique object in the set is separated from the next by a comma.
- Also like dictionaries, sets do not maintain insertion order (but can be ordered with the `sorted` function).
- You can pass any sequence to the `set` function to create a set of elements from the objects in the sequence (minus any duplicates).
- Sets come pre-packaged with lots of built-in functionality, including methods to perform union, difference, and intersection.

Sharpen your pencil

Here is the code to the `vowels3.py` program once more.

Based on what you now know about sets, grab your pencil and strike out the code you no longer need. In the space provided on the right, provide the code you'd add to convert this list-using program to take advantage of a set.

Hint: you'll end up with a lot less code.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

When you're done, be sure to rename your file `vowels7.py`.

Sharpen your pencil Solution

Here is the code to the `vowels3.py` program once more.

Based on what you now know about sets, you were to grab your pencil and strike out the code you no longer needed. In the space provided on the right, you were to provide the code you'd add to convert this list-using program to take advantage of a set.

Hint: you'll end up with a lot less code.

There's lots of code to get rid of.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Create a set of vowels.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

These five lines of list-processing code are replaced by a single line of set code.

When you were done, you were to rename your file `vowels7.py`.

I feel cheated...all that time wasted learning about lists and dictionaries, and the best solution to this vowels problem all along was to use a set? Seriously?

It wasn't a waste of time.

Being able to spot when to use one built-in data structure over another is important (as you'll want to be sure you're picking the right one). The only way you can do this is to get experience using *all* of them. None of the built-in data structures qualify as a "one size fits all" technology, as they all have their strengths and weaknesses. Once you understand what these are, you'll be better equipped to select the correct data structure based on your application's specific data requirements.

Test Drive

Let's take `vowels7.py` for a spin to confirm that the set-based version of our program runs as expected:

Our latest code

vowels7.py - /Users/Paul/Desktop/_NewBook/ch03/vowels7.py (3.4.3)

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Python 3.4.3 Shell

```
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: hitch-hiker
e
i
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: Galaxy
a
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: life, the universe, and everything
i
a
u
e
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: sky
>>>
```

Ln: 23 Col: 4

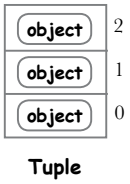
Everything is working as expected.

Using a set was the perfect choice here...

But that's not to say that the two other data structures don't have their uses. For instance, if you need to perform, say, a frequency count, Python's dictionary works best. However, if you are more concerned with maintaining insertion order, then only a list will do...which is almost true. There's one other built-in data structure that maintains insertion order, and which we've yet to discuss: the **tuple**.

Let's spend the remainder of this chapter in the company of Python's tuple.

Making the Case for Tuples



Tuple

When most programmers new to Python first come across the **tuple**, they question why such a data structure even exists. After all, a tuple is like a list that cannot be changed once it's created (and populated with data). Tuples are immutable: *they cannot change*. So, why do we need them?

It turns out that having an immutable data structure can often be useful. Imagine that you need to guard against side effects by ensuring some data in your program never changes. Or perhaps you have a large constant list (which you know won't change) and you're worried about performance. Why incur the cost of all that extra (mutable) list processing code if you're never going to need it? Using a tuple in these cases avoids unnecessary overhead and guards against nasty data side effects (were they to occur).

there are no Dumb Questions

Q: Where does the name "tuple" come from?

A: It depends whom you ask, but the name has its origin in mathematics. Find out more than you'd ever want to know by visiting *https://en.wikipedia.org/wiki/Tuple*.

How to spot a tuple in code

As tuples are closely related to lists, it's no surprise that they look similar (and behave in a similar way. Tuples are surrounded by parentheses, whereas lists use square brackets. A quick visit to the >>> prompt lets us compare tuples with lists. Note how we're using the `type` built-in function to confirm the type of each object created:

There's nothing new here. A list of vowels is created.

```
>>> vowels = [ 'a', 'e', 'i', 'o', 'u' ]
>>> type(vowels)
<class 'list'>
>>> vowels2 = ( 'a', 'e', 'i', 'o', 'u' )
>>> type(vowels2)
<class 'tuple'>
```

This tuple looks like a list, but isn't. Tuples are surrounded by parentheses (not square brackets).

The "type" built-in function reports the type of any object.

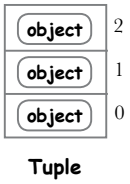
Now that `vowels` and `vowels2` exist (and are populated with data), we can ask the shell to display what they contain. Doing so confirms that the tuple is not quite the same as the list:

```
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels2
('a', 'e', 'i', 'o', 'u')
```

The parentheses indicate that this is a tuple.

But what happens if we try to change a tuple?

Tuples Are Immutable



As tuples are sort of like lists, they support the same square bracket notation commonly associated with lists. We already know that we can use this notation to change the contents of a list. Here's what we'd do to change the lowercase letter `i` in the `vowels` list to be an uppercase `I`:

```
>>> vowels[2] = 'I'
>>> vowels
[' a', 'e', 'I', 'o', 'u']
```

Assign an uppercase "I" to the third element of the "vowels" list.

As expected, the third element in the list (at index location 2) has changed, which is fine and expected, as lists are mutable. However, look what happens if we try to do the same thing with the `vowels2` tuple:

The interpreter complains loudly if you try to change a tuple.

```
>>> vowels2[2] = 'I'
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    vowels2[2] = 'I'
TypeError: 'tuple' object does not support item assignment
>>> vowels2
(' a', 'e', 'i', 'o', 'u')
```

No change here, as tuples are immutable

Tuples are immutable, so we can't complain when the interpreter protests at our trying to change the objects stored in the tuple. After all, that's the whole point of a tuple: once created and populated with data, a tuple cannot change.

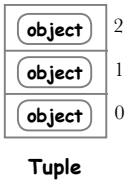
Make no mistake: this behavior is useful, especially when you need to ensure that some data can't change. The only way to ensure this is to put the data in a tuple, which then instructs the interpreter to stop any code from trying to change the tuple's data.

As we work our way through the rest of this book, we'll always use tuples when it makes sense to do so. With reference to the vowel-processing code, it should now be clear that the `vowels` data structure should always be stored in a tuple as opposed to a list, as it makes no sense to use a mutable data structure in this instance (as the five vowels *never* need to change).

There's not much else to tuples—think of them as immutable lists, nothing more. However, there is one usage that trips up many a programmer, so let's learn what this is so that you can avoid it.

If the data in your structure never changes, put it in a tuple.

Watch Out for Single-Object Tuples



Tuple

Let's imagine you want to store a single string in a tuple. It's tempting to put the string inside parentheses, and then assign it to a variable name...but doing so does not produce the expected outcome.

Take a look at this interaction with the >>> prompt, which demonstrates what happens when you do this:

```
>>> t = ('Python')
>>> type(t)
<class 'str'>
>>> t
'Python'
```

This is not what we expected. We've ended up with a string. What happened to our tuple?

What looks like a single-object tuple isn't; it's a string. This has happened due to a syntactical quirk in the Python language. The rule is that, in order for a tuple to be a tuple, every tuple needs to include at least one comma between the parentheses, even when the tuple contains a single object. This rule means that in order to assign a single object to a tuple (we're assigning a string object in this instance), we need to include the trailing comma, like so:

```
>>> t2 = ('Python',)
```

That trailing comma makes all the difference, as it tells the interpreter that this is a tuple.

This looks a little weird, but don't let that worry you. Just remember this rule and you'll be fine: *every tuple needs to include at least one comma between the parentheses.* When you now ask the interpreter to tell you what type `t2` is (as well as display its value), you learn that `t2` is a tuple, which is what is expected:

```
>>> type(t2)
<class 'tuple'>
>>> t2
('Python',)
```

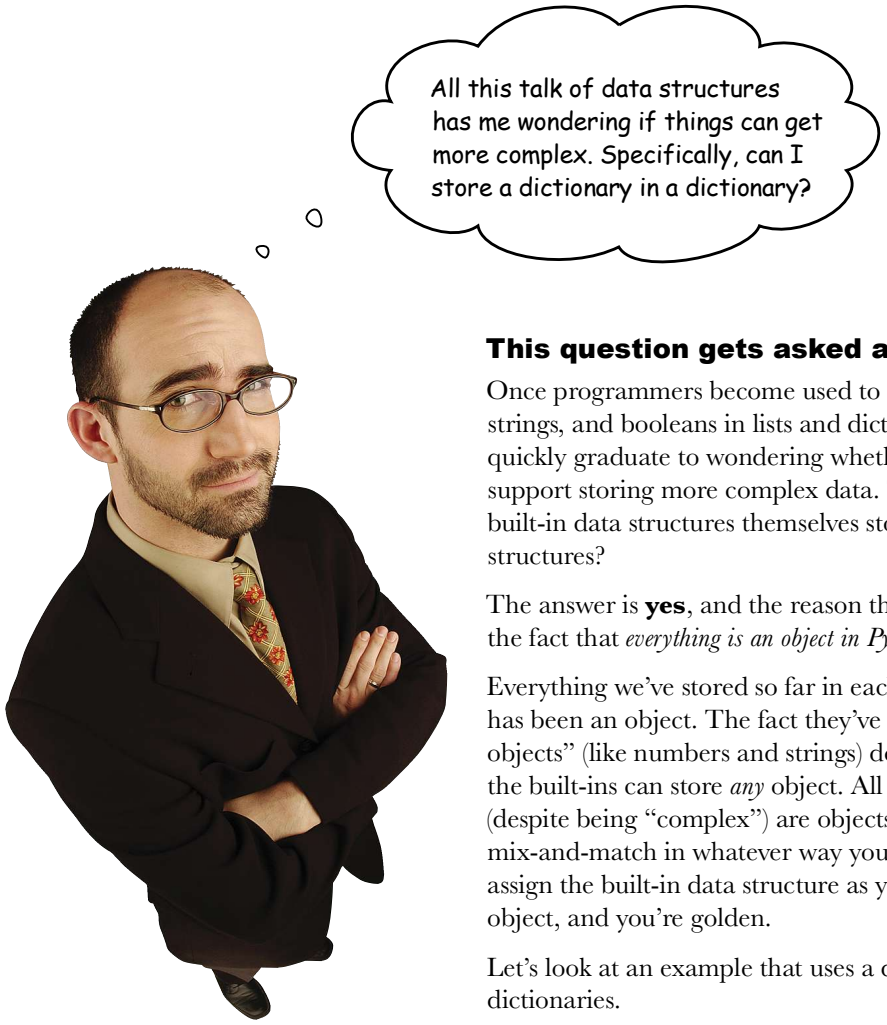
That's better: we now have a tuple.

The interpreter displays the single-object tuple with the trailing comma.

It is quite common for functions to both accept and return their arguments as a tuple, even when they accept or return a single object. Consequently, you'll come across this syntax often when working with functions. We'll have more to say about the relationship between functions and tuples in a little bit; in fact, we'll devote the next chapter to functions (so you won't have long to wait).

Now that you know about the four data structure built-ins, and before we get to the chapter on functions, let's take a little detour and squeeze in a short—and fun!—example of a more complex data structure.

Combining the Built-in Data Structures



This question gets asked a lot.

Once programmers become used to storing numbers, strings, and booleans in lists and dictionaries, they very quickly graduate to wondering whether the built-ins support storing more complex data. That is, can the built-in data structures themselves store built-in data structures?

The answer is **yes**, and the reason this is so is due to the fact that *everything is an object in Python*.

Everything we've stored so far in each of the built-ins has been an object. The fact they've been "simple objects" (like numbers and strings) does not matter, as the built-ins can store *any* object. All of the built-ins (despite being "complex") are objects, too, so you can mix-and-match in whatever way you choose. Simply assign the built-in data structure as you would a simple object, and you're golden.

Let's look at an example that uses a dictionary of dictionaries.

there are no Dumb Questions

Q: Does what you're about to do only work with dictionaries? Can I have a list of lists, or a set of lists, or a tuple of dictionaries?

A: Yes, you can. We'll demonstrate how a dictionary of dictionaries works, but you can combine the built-ins in whichever way you choose.

Storing a Table of Data

As everything is an object, any of the built-in data structures can be stored in any other built-in data structure, enabling the construction of arbitrarily complex data structures...subject to your brain's ability to actually visualize what's going on. For instance, although *a dictionary of lists containing tuples that contain sets of dictionaries* might sound like a good idea, it may not be, as its complexity is off the scale.

A complex structure that comes up a lot is a dictionary of dictionaries. This structure can be used to create a *mutable table*. To illustrate, imagine we have this table describing a motley collection of characters:

Name	Gender	Occupation	Home Planet
Ford Prefect	Male	Researcher	Betelgeuse Seven
Arthur Dent	Male	Sandwich-Maker	Earth
Tricia McMillan	Female	Mathematician	Earth
Marvin	Unknown	Paranoid Android	Unknown

Recall how, at the start of this chapter, we created a dictionary called `person3` to store Ford Prefect's data:

```
person3 = { 'Name': 'Ford Prefect',
            'Gender': 'Male',
            'Occupation': 'Researcher',
            'Home Planet': 'Betelgeuse Seven' }
```

Rather than create (and then grapple with) four individual dictionary variables for each line of data in our table, let's create a single dictionary variable, called `people`. We'll then use `people` to store any number of other dictionaries.

To get going, we first create an empty `people` dictionary, then assign Ford Prefect's data to a key:

Start with a new, empty dictionary.

```
>>> people = {}
>>> people['Ford'] = { 'Name': 'Ford Prefect',
                       'Gender': 'Male',
                       'Occupation': 'Researcher',
                       'Home Planet': 'Betelgeuse Seven' }
```

The key is "Ford", and the value is another dictionary.

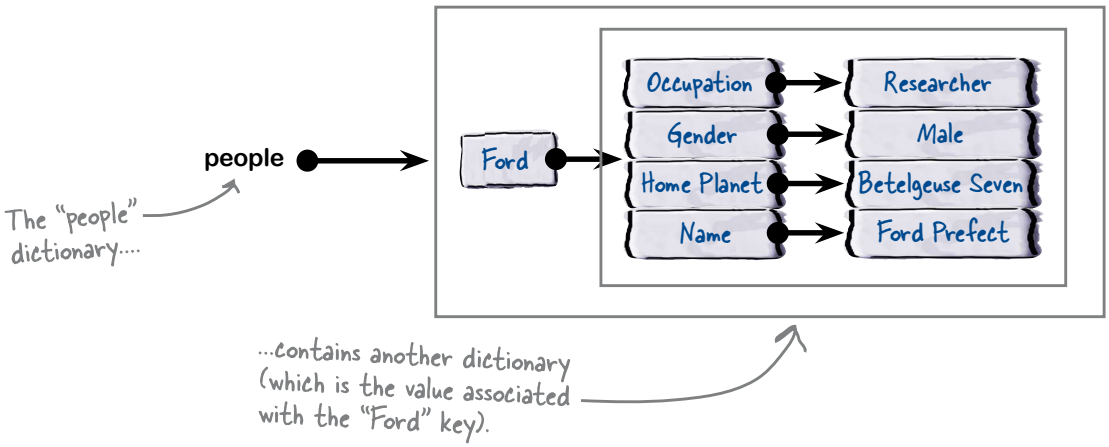
A Dictionary Containing a Dictionary

With the `people` dictionary created and one row of data added (Ford's), we can ask the interpreter to display the `people` dictionary at the >>> prompt. The resulting output looks a little confusing, but all of our data is there:

A dictionary embedded in a dictionary—note the extra curly braces.

```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'}}
```

There is only one embedded dictionary in `people` (at the moment), so calling this a "dictionary of dictionaries" is a bit of a stretch, as `people` contains just the one right now. Here's what `people` looks like to the interpreter:



We can now proceed to add in the data from the other three rows in our table:

Arthur's data

```
>>> people['Arthur'] = { 'Name': 'Arthur Dent',
                         'Gender': 'Male',
                         'Occupation': 'Sandwich-Maker',
                         'Home Planet': 'Earth' }
>>> people['Trillian'] = { 'Name': 'Tricia McMillan',
                           'Gender': 'Female',
                           'Occupation': 'Mathematician',
                           'Home Planet': 'Earth' }
>>> people['Robot'] = { 'Name': 'Marvin',
                        'Gender': 'Unknown',
                        'Occupation': 'Paranoid Android',
                        'Home Planet': 'Unknown' }
```

Tricia's data is associated with the "Trillian" key.

Marvin's data is associated with the "Robot" key.

A Dictionary of Dictionaries (a.k.a. a Table)

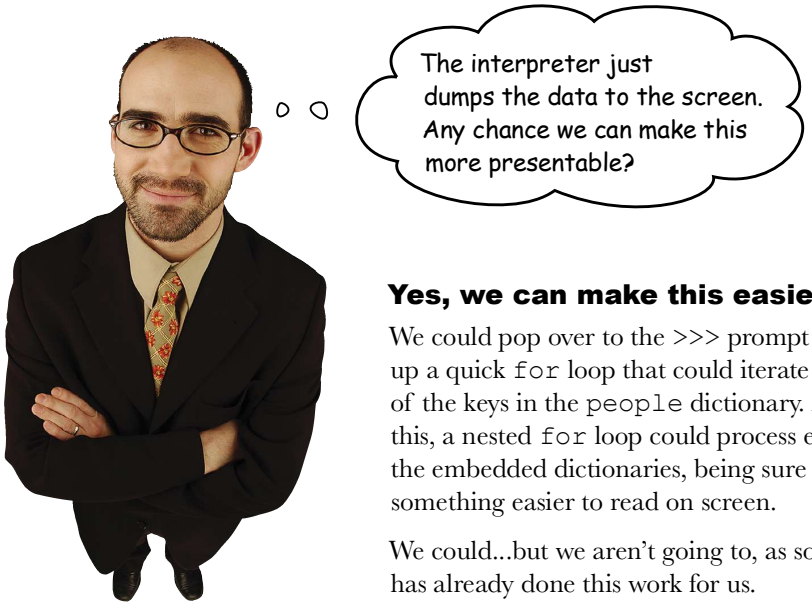
With the `people` dictionary populated with four embedded dictionaries, we can ask the interpreter to display the `people` dictionary at the >>> prompt.

Doing so results in an unholy mess of data on screen (see below).

Despite the mess, all of our data is there. Note that each opening curly brace starts a new dictionary, while a closing curly brace terminates a dictionary. Go ahead and count them (there are five of each):

It's a little hard to read, but all the data is there.

```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'},
'Trillian': {'Occupation': 'Mathematician', 'Gender':
'Female', 'Home Planet': 'Earth', 'Name': 'Tricia
McMillan'}, 'Robot': {'Occupation': 'Paranoid Android',
'Gender': 'Unknown', 'Home Planet': 'Unknown', 'Name':
'Marvin'}, 'Arthur': {'Occupation': 'Sandwich-Maker',
'Gender': 'Male', 'Home Planet': 'Earth', 'Name': 'Arthur
Dent'}}
```



Yes, we can make this easier to read.

We could pop over to the >>> prompt and code up a quick `for` loop that could iterate over each of the keys in the `people` dictionary. As we did this, a nested `for` loop could process each of the embedded dictionaries, being sure to output something easier to read on screen.

We could...but we aren't going to, as someone else has already done this work for us.

Pretty-Printing Complex Data Structures

The standard library includes a module called `pprint` that can take any data structure and display it in a easier-to-read format. The name `pprint` is a shorthand for "pretty print."

Let's use the `pprint` module with our `people` dictionary (of dictionaries). Below, we once more display the data "in the raw" at the >>> prompt, and then we import the `pprint` module before invoking its `pprint` function to produce the output we need:

Our dictionary of dictionaries is hard to read.

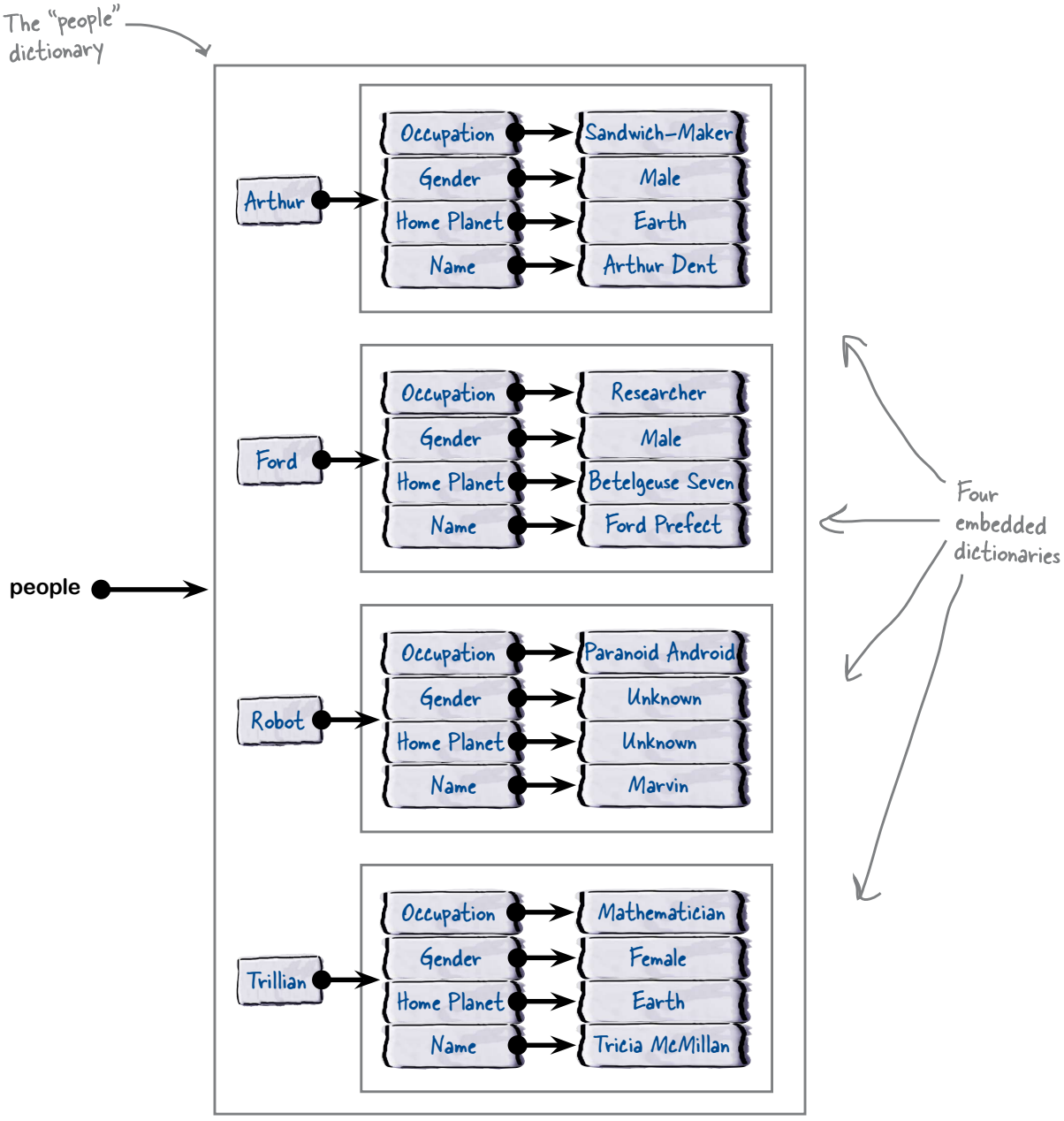
```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'},
'Trillian': {'Occupation': 'Mathematician', 'Gender':
'Female', 'Home Planet': 'Earth', 'Name': 'Tricia
McMillan'}, 'Robot': {'Occupation': 'Paranoid Android',
'Gender': 'Unknown', 'Home Planet': 'Unknown', 'Name':
'Marvin'}, 'Arthur': {'Occupation': 'Sandwich-Maker',
'Gender': 'Male', 'Home Planet': 'Earth', 'Name': 'Arthur
Dent'}}
>>>
>>> import pprint
>>>
>>> pprint.pprint(people)
{'Arthur': {'Gender': 'Male',
            'Home Planet': 'Earth',
            'Name': 'Arthur Dent',
            'Occupation': 'Sandwich-Maker'},
 'Ford': {'Gender': 'Male',
          'Home Planet': 'Betelgeuse Seven',
          'Name': 'Ford Prefect',
          'Occupation': 'Researcher'},
 'Robot': {'Gender': 'Unknown',
           'Home Planet': 'Unknown',
           'Name': 'Marvin',
           'Occupation': 'Paranoid Android'},
 'Trillian': {'Gender': 'Female',
              'Home Planet': 'Earth',
              'Name': 'Tricia McMillan',
              'Occupation': 'Mathematician'}}
```

Import the "pprint" module, then invoke the "pprint" function to do the work.

This output is much easier on the eye. Note that we still have five opening and five closing curly braces. It's just that—thanks to "pprint"—they are now so much easier to see (and count).

Visualizing Complex Data Structures

Let's update our diagram depicting what the interpreter now "sees" when the `people` dictionary of dictionaries is populated with data:



At this point, a reasonable question to ask is: *Now that we have all this data stored in a dictionary of dictionaries, how do we get at it?* Let's answer this question on the next page.

Accessing a Complex Data Structure's Data

We now have our table of data stored in the `people` dictionary. Let's remind ourselves of what the original table of data looked like:

Name	Gender	Occupation	Home Planet
Ford Prefect	Male	Researcher	Betelgeuse Seven
Arthur Dent	Male	Sandwich-Maker	Earth
Tricia McMillan	Female	Mathematician	Earth
Marvin	Unknown	Paranoid Android	Unknown

If we were asked to work out what Arthur does, we'd start by looking down the **Name** column for Arthur's name, and then we'd look across the row of data until we arrived at the **Occupation** column, where we'd be able to read "Sandwich-Maker."

When it comes to accessing data in a complex data structure (such as our `people` dictionary of dictionaries), we can follow a similar process, which we're now going to demonstrate at the >>> prompt.

We start by finding Arthur's data in the `people` dictionary, which we can do by putting Arthur's key between square brackets:

Ask for Arthur's row of data.

```
>>> people['Arthur']
{'Occupation': 'Sandwich-Maker', 'Home Planet': 'Earth',
'Gender': 'Male', 'Name': 'Arthur Dent'}
```

The row of dictionary data associated with the "Arthur" key

Having found Arthur's row of data, we can now ask for the value associated with the `Occupation` key. To do this, we employ a **second** pair of square brackets to index into Arthur's dictionary and access the data we're looking for:

Identify the row.

Identify the column.

```
>>> people['Arthur']['Occupation']
'Sandwich-Maker'
```

Using double square brackets lets you access any data value from a table by identifying the row and column you are interested in. The row corresponds to a key used by the enclosing dictionary (`people`, in our example), while the column corresponds to any of the keys used by an embedded dictionary.

Data Is As Complex As You Make It

Whether you have a small amount of data (a simple list) or something more complex (a dictionary of dictionaries), it's nice to know that Python's four built-in data structures can accommodate your data needs. What's especially nice is the dynamic nature of the data structures you build; other than tuples, each of the data structures can grow and shrink as needed, with Python's interpreter taking care of any memory allocation/deallocation details for you.

We are not done with data yet, and we'll come back to this topic again later in this book. For now, though, you know enough to be getting on with things.

In the next chapter, we start to talk about techniques to effectively reuse code with Python, by learning about the most basic of the code reuse technologies: functions.

Chapter 3's Code, 1 of 2

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

This is the code for "vowels4.py", which performed a frequency count. This code was (loosely) based on "vowels3.py", which we first saw in Chapter 2.

In an attempt to remove the dictionary initialization code, we created "vowels5.py", which crashed with a runtime error (due to us failing to initialize the frequency counts).

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

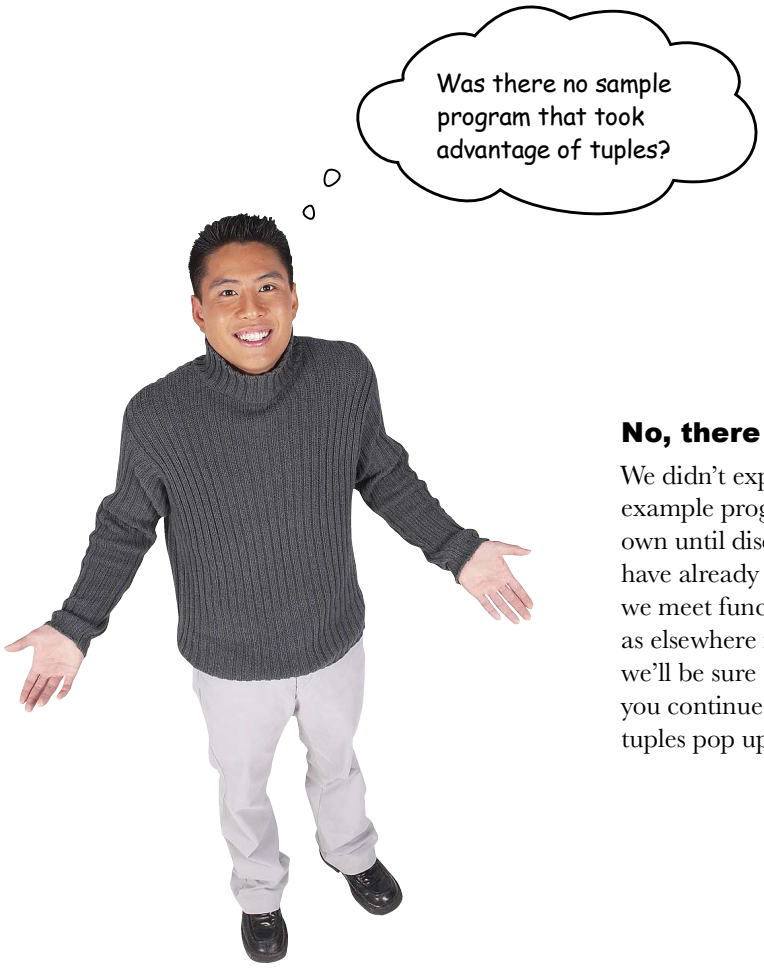
```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

"vowels6.py" fixed the runtime error thanks to the use of the "setdefault" method, which comes with every dictionary (and assigns a default value to a key if a value isn't already set).

Chapter 3's Code, 2 of 2

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

The final version of the vowels program, "vowels7.py", took advantage of Python's set data structure to considerably shrink the list-based "vowels3.py" code, while still providing the same functionality.



No, there wasn't. But that's OK.

We didn't exploit tuples in this chapter with an example program, as tuples don't come into their own until discussed in relation to functions. As we have already stated, we'll see tuples again when we meet functions (in the next chapter), as well as elsewhere in this book. Each time we see them, we'll be sure to point out each tuple usage. As you continue with your Python travels, you'll see tuples pop up all over the place.

4 code reuse

Functions and Modules

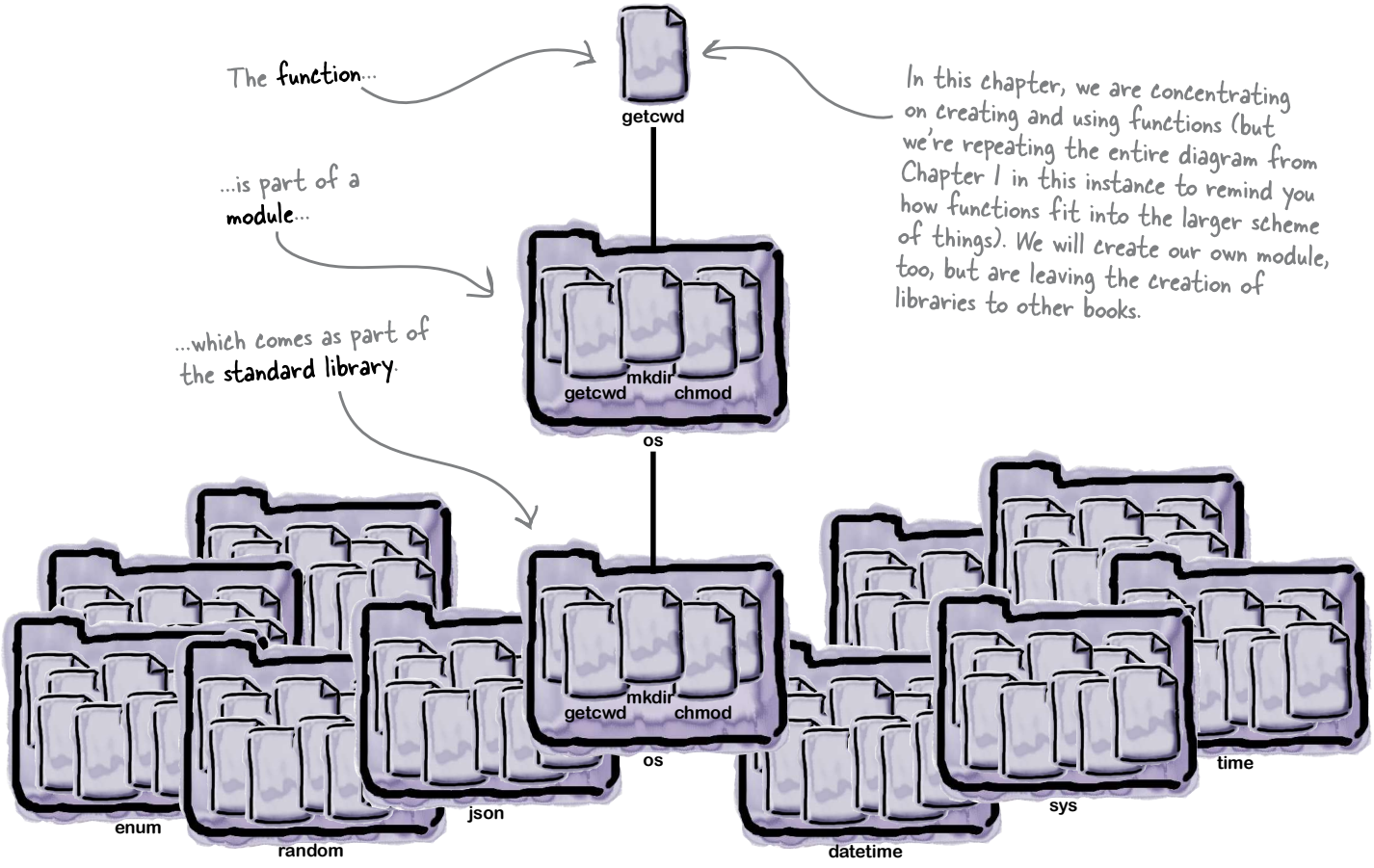
Reusing code is key to building a maintainable system.

And when it comes to reusing code in Python, it all starts and ends with the humble **function**. Take some lines of code, give them a name, and you've got a function (which can be reused). Take a collection of functions and package them as a file, and you've got a **module** (which can also be reused). It's true what they say: *it's good to share*, and by the end of this chapter, you'll be well on your way to **sharing** and **reusing** your code, thanks to an understanding of how Python's functions and modules work.

Reusing Code with Functions

Although a few lines of code can accomplish a lot in Python, sooner or later you're going to find your program's codebase is growing...and, when it does, things quickly become harder to manage. What started out as 20 lines of Python code has somehow ballooned to 500 lines or more! When this happens, it's time to start thinking about what strategies you can use to reduce the complexity of your codebase.

Like many other programming languages, Python supports **modularity**, in that you can break large chunks of code into smaller, more manageable pieces. You do this by creating **functions**, which you can think of as named chunks of code. Recall this diagram from Chapter 1, which shows the relationship between functions, modules, and the standard library:



In this chapter, we're going to concentrate on what's involved in creating your own functions, shown at the very top of the diagram. Once you're happily creating functions, we'll also show you how to create a module.

Introducing Functions

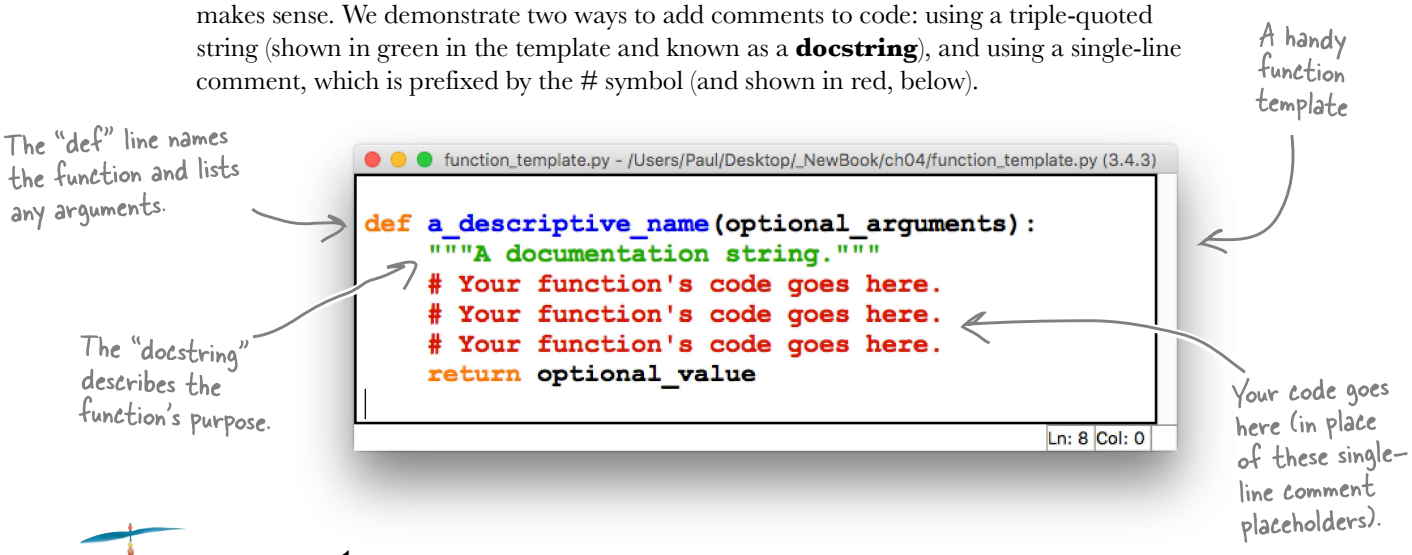
Before we get to turning some of our existing code into a function, let's spend a moment looking at the anatomy of *any* function in Python. Once this introduction is complete, we'll look at some of our existing code and go through the steps required to turn it into a function that you can reuse.

Don't sweat the details just yet. All you need to do here is get a feel for what functions look like in Python, as described on this and the next page. We'll delve into the details of all you need to know as this chapter progresses. The IDLE window on this page presents a template you can use when creating any function. As you are looking at it, consider the following:

1. **Functions introduce two new keywords: `def` and `return`**
   Both of these keywords are colored orange in IDLE. The `def` keyword names the function (shown in blue), and details any arguments the function may have. The use of the `return` keyword is optional, and is used to pass back a value to the code that invoked the function.

2. **Functions can accept argument data**
   A function can accept argument data (i.e., input to the function). You can specify a list of arguments between the parentheses on the `def` line, following the function's name.

3. **Functions contain code and (usually) documentation**
   Code is indented one level beneath the `def` line, and should include comments where it makes sense. We demonstrate two ways to add comments to code: using a triple-quoted string (shown in green in the template and known as a **docstring**), and using a single-line comment, which is prefixed by the # symbol (and shown in red, below).



Geek Bits

Python uses the name "function" to describe a reusable chunk of code. Other programming languages use names such as "procedure," "subroutine," and "method." When a function is part of a Python class, it's known as a "method.". You'll learn all about Python's classes and methods in a later chapter.

What About Type Information?

Take another look at our function template. Other than some code to execute, do you think there's anything missing? Is there anything you'd expect to be specified, but isn't? Take another look:

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value
```

Ln: 8 Col: 0

Is there anything missing from this function template?

I'm a little freaked out by that function template. How does the interpreter know what types the arguments are, as well as what type the return value is?

It doesn't know, but don't let that worry you.

The Python interpreter does not force you to specify the type of your function's arguments or the return value. Depending on the programming languages you've used before, this may well freak you out. Don't let it.

Python lets you send any *object* as a argument, and pass back any *object* as a return value. The interpreter doesn't care or check what type these objects are (only that they are provided).

With Python 3, it is possible to *indicate* the expected types for arguments/return values, and we'll do just that later in this chapter. However, indicating the types expected does not "magically" switch on type checking, as Python *never* checks the types of the arguments or any return values.

Naming a Chunk of Code with "def"

Once you've identified a chunk of your Python code you want to reuse, it's time to create a function. You create a function using the `def` keyword (which is short for *define*). The `def` keyword is followed by the function's name, an optionally empty list of arguments (enclosed in parentheses), a colon, and then one or more lines of indented code.

Recall the `vowels7.py` program from the end of the last chapter, which, given a word, prints the vowels contained in that word:

This is "vowels7.py" from the end of Chapter 3.

Take a set of vowels...

...and a word...

...then perform an intersection.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Display any results.

Let's imagine you plan to use these five lines of code many times in a much larger program. The last thing you'll want to do is copy and paste this code everywhere it's needed...so, to keep things manageable and to ensure you only need to maintain **one copy** of this code, let's create a function.

Take the time to choose a good descriptive name for your function.

We'll demonstrate how at the Python Shell (for now). To turn the above five lines of code into a function, use the `def` keyword to indicate that a function is starting; give the function a descriptive name (*always* a good idea); provide an optionally empty list of arguments in parentheses, followed by a colon; and then indent the lines of code relative to the `def` keyword, as follows:

Give your function a nice, descriptive name.

Provide an optional list of arguments—in this case, this function has no arguments, so the list is empty.

Start with the "def" keyword.

Don't forget the colon.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

The fives lines of code from the "vowels7.py" program, suitably indented

As this is the shell, remember to press the Enter key TWICE to confirm that the indented code has concluded.

Now that the function exists, let's invoke it to see if it is working the way we expect it to.

Invoking Your Function

To invoke functions in Python, provide the function name together with values for any arguments the function expects. As the `search4vowels` function (currently) takes no arguments, we can invoke it with an empty argument list, like so:

```
>>> search4vowels()
Provide a word to search for vowels: hitch-hiker
e
i
```

Invoking the function again runs it again:

```
>>> search4vowels()
Provide a word to search for vowels: galaxy
a
```

There are no surprises here: invoking the function executes its code.

Edit your function in an editor, not at the prompt

At the moment, the code for the `search4vowels` function has been entered into the >>> prompt, and it looks like this:

Our function as entered at the shell prompt.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

In order to work further with this code, you can recall it at the >>> prompt and edit it, but this becomes very unwieldy, very quickly. Recall that once the code you're working with at the >>> prompt is more than a few lines long, you're better off copying the code into an IDLE edit window. You can edit it much more easily there. So, let's do that before continuing.

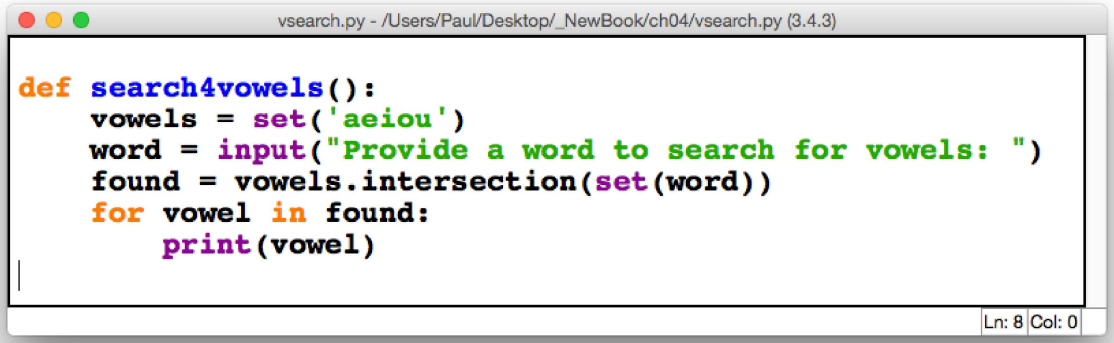
Create a new, empty IDLE edit window, then copy the function's code from the >>> prompt (being sure *not* to copy the >>> characters), and paste it into the edit window. Once you're satisfied that the formatting and indentation are correct, save your file as `vsearch.py` before continuing.

Be sure you've saved your code as "vsearch.py" after copying the function's code from the shell.

Use IDLE's Editor to Make Changes

Here's what the `vsearch.py` file looks like in IDLE:

The function's code is now in an IDLE edit window, and has been saved as "vsearch.py".



If you press F5 while in the edit window, two things happen: the IDLE shell is brought to the foreground, and the shell restarts. However, nothing appears on screen. Try this now to see what we mean: press F5.

If IDLE displays an error when you press F5, don't panic! Return to your edit window and check that your code is the exact same as ours, then try again.

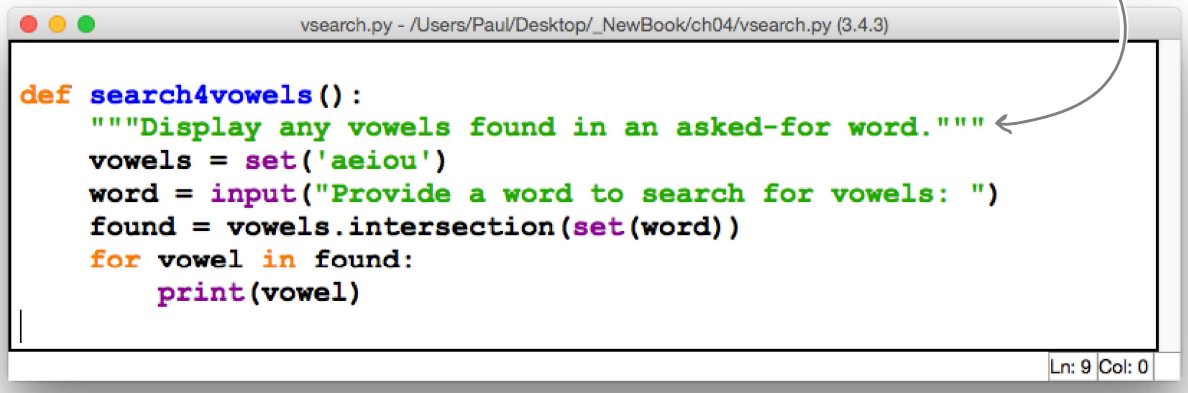
The reason for nothing displaying is that you have yet to invoke the function. We'll invoke it in a little bit, but for now let's make one change to our function before moving on. It's a small change, but an important one nonetheless.

Let's add some documentation to the top of our function.

To add a multiline comment (a **docstring**) to any code, enclose your comment text in triple quotes.

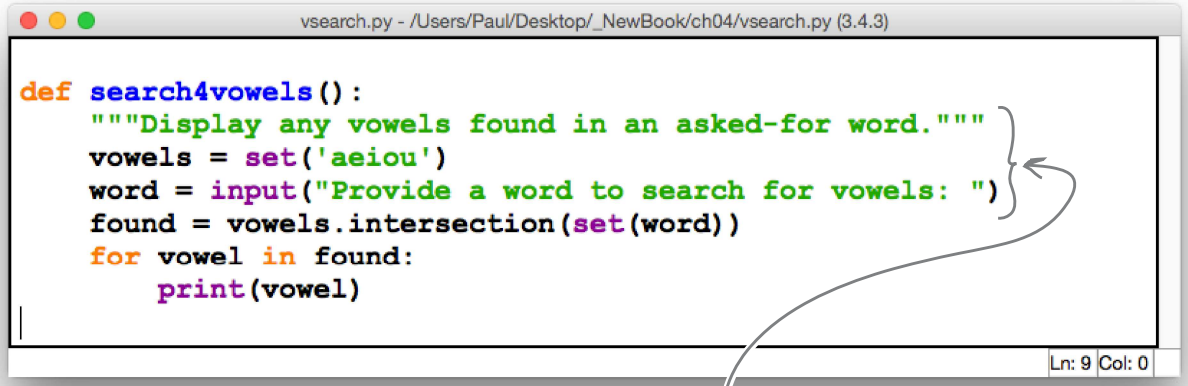
Here's the `vsearch.py` file once more, with a docstring added to the top of the function. Go ahead and make this change to your code, too:

A docstring has been added to the function's code, which (briefly) describes the purpose of this function.



What's the Deal with All Those Strings?

Take another look at the function as it currently stands. Pay particular attention to the three strings in this code, which are all colored green by IDLE:



IDLE's syntax-highlighting shows that we have a consistency problem with our use of string quotes. When do we use which style?

Understanding the string quote characters

In Python, strings can be enclosed in a single quote character (`'`), a double quote character (`"`), or what's known as triple quotes (`"""` or `'''`).

As mentioned earlier, triple quotes around strings are known as **docstrings**, because they are mainly used to document a function's purpose (as shown above). Even though you can use `"""` or `'''` to surround your docstrings, most Python programmers prefer to use `"""`. Docstrings have an interesting characteristic in that they can span multiple lines (other programming languages use the name "heredoc" for the same concept).

Strings enclosed by a single quote character (`'`) or a double quote character (`"`) **cannot** span multiple lines: you must terminate the string with a matching quote character on the same line (as Python uses the end of the line as a statement terminator).

Which character you use to enclose your strings is up to you, although using the single quote character is very popular with the majority of Python programmers. That said, and above all else, your usage should be consistent.

The code shown at the top of this page (despite being only a handful of lines of code) is *not* consistent in its use of string quote characters. Note that the code runs fine (as the interpreter doesn't care which style you use), but mixing and matching styles can make the code harder to read than it needs to be (which is a shame).

Be consistent in your use of string quote characters. If possible, use single quotes.

Follow Best Practice As Per the PEPs

When it comes to formatting your code (not just strings), the Python programming community has spent a long time establishing and documenting best practice. This best practice is known as **PEP 8**. PEP is shorthand for "Python Enhancement Protocol."

There are a large number of PEP documents in existence, and they primarily detail proposed and implemented enhancements to the Python programming language, but can also document advice (on what to do and what not to do), as well as describe various Python processes. The details of the PEP documents can be very technical and (often) esoteric. Thus, the vast majority of Python programmers are aware of their existence but rarely interact with PEPs in detail. This is true of most PEPs *except* for PEP 8.

Find the list of PEPs here: https://www.python.org/dev/peps/.

PEP 8 is *the* style guide for Python code. It is recommended reading for all Python programmers, and it is the document that suggests the "be consistent" advice for string quotes described on the last page. Take the time to read PEP 8 at least once. Another document, PEP 257, offers conventions on how to format docstrings, and it's worth reading, too.

Here is the `search4vowels` function once more in its PEP 8– and PEP 257–compliant form. The changes aren't extensive, but standardizing on single quote characters around our strings (but not around our docstrings) does look a bit better:

This is a PEP 257–compliant docstring.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels():
    """Display any vowels found in an asked-for word."""
    vowels = set('aeiou')
    word = input('Provide a word to search for vowels: ')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

We've heeded PEP 8's advice on being consistent with the single quote character we use to surround our strings.

Of course, you don't have to write code that conforms *exactly* to PEP 8. For example, our function name, `search4vowels`, does not conform to the guidelines, which suggests that words in a function's name should be separated by an underscore: a more compliant name is `search_for_vowels`. Note that PEP 8 is a set of guidelines, not rules. You don't have to comply, only consider, and we like the name `search4vowels`.

That said, the vast majority of Python programmers will thank you for writing code that conforms to PEP 8, as it is often easier to read than code that doesn't.

Let's now return to enhancing the `search4vowels` function to accept arguments.

Functions Can Accept Arguments

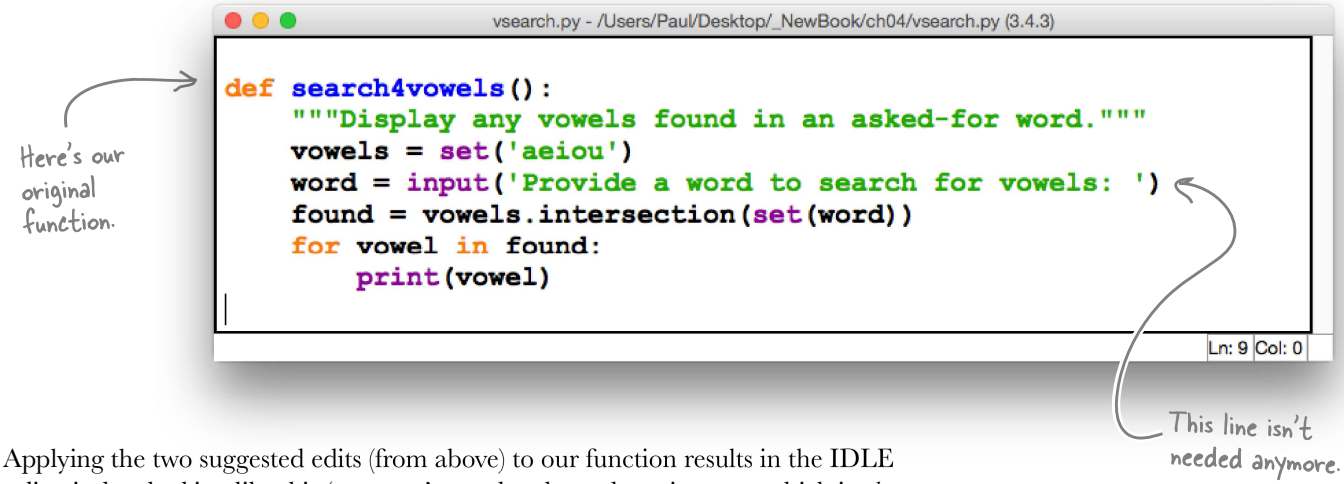
Rather than having the function prompt the user for a word to search, let's change the `search4vowels` function so we can pass it the word as input to an argument.

Adding an argument is straightforward: you simply insert the argument's name between the parentheses on the `def` line. This argument name then becomes a variable in the function's suite. This is an easy edit.

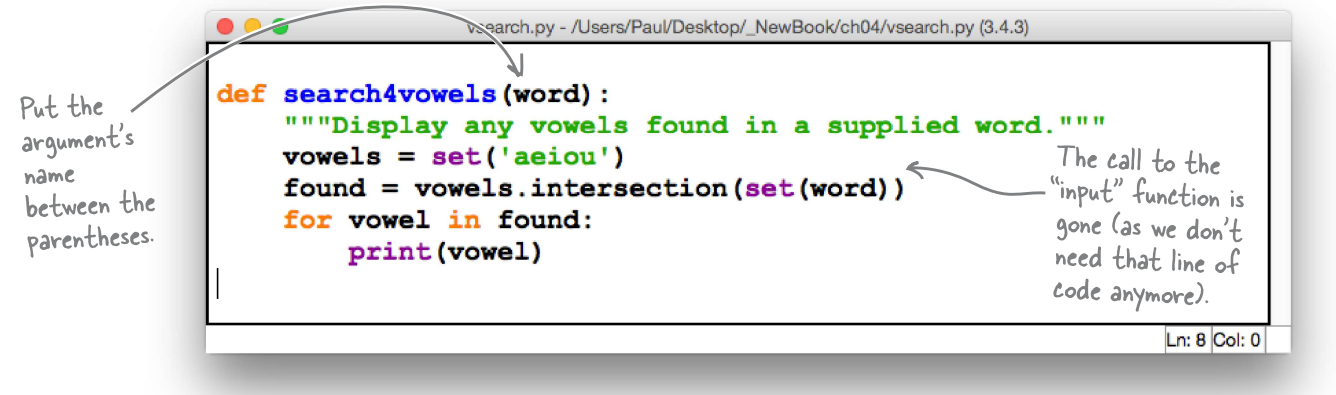
Let's also remove the line of code that prompts the user to supply a word to search, which is another easy edit.

Let's remind ourselves of the current state of our code:

Remember: "suite" is Python-speak for "block."



Applying the two suggested edits (from above) to our function results in the IDLE edit window looking like this (note: we've updated our docstring, too, which is *always* a good idea):



Be sure to save your file after each code change, before pressing F5 to take the new version of your function for a spin.

Test Drive

With your code loaded into IDLE's edit window (and saved), press F5, then invoke the function a few times and see what happens:

The current "search4vowels" code

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
>>> search4vowels()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    search4vowels()
TypeError: search4vowels() missing 1 required positional argument: 'word'
>>> search4vowels('hitch-hiker')
e
i
>>> search4vowels('hitch-hiker', 'galaxy')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    search4vowels('hitch-hiker', 'galaxy')
TypeError: search4vowels() takes 1 positional argument but 2 were given
>>>
```

Ln: 12 Col: 4

Although we've invoked the "search4vowels" function three times in this Test Drive, the only invocation that ran successfully was the one that passed in a single, stringed argument. The other two failed. Take a moment to read the error messages produced by the interpreter to learn why each of the incorrect calls failed.

there are no Dumb Questions

Q: Am I restricted to only a single argument when creating functions in Python?

A: No, you can have as many arguments as you want, depending on the service your function is providing. We are deliberately starting off with a straightforward example, and we'll get to more involved examples as this chapter progresses. You can do a lot with arguments to functions in Python, and we plan to discuss most of what's possible over the next dozen pages or so.

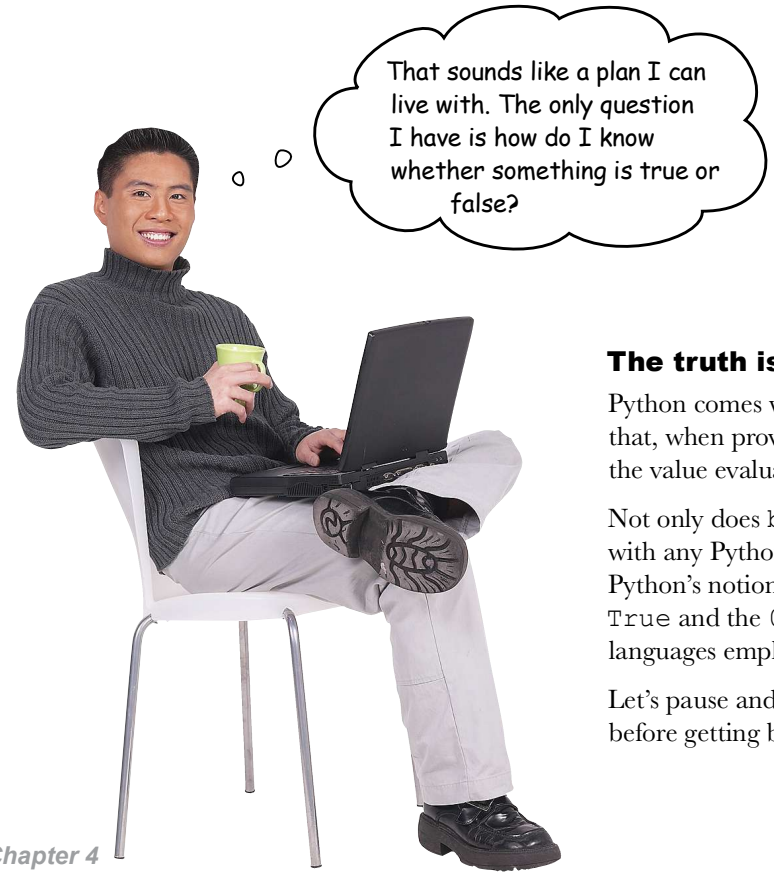
Functions Return a Result

As well as using a function to abstract some code and give it a name, programmers typically want functions to return some calculated value, which the code that called the function can then work with. To support returning a value (or values) from a function, Python provides the `return` statement.

When the interpreter encounters a `return` statement in your function's suite, two things happen: the function terminates at the `return` statement, and any value provided to the `return` statement is passed back to your calling code. This behavior mimics how `return` works in the majority of other programming languages.

Let's start with a straightforward example of returning a single value from our `search4vowels` function. Specifically, let's return either `True` or `False` depending on whether the `word` supplied as an argument contains any vowels.

This *is* a bit of a departure from our function's existing functionality, but bear with us, as we are going to build up to something more complex (and useful) in a bit. Starting with a simple example ensures we have the basics in place first, before moving on.



The truth is...

Python comes with a built-in function called `bool` that, when provided with any value, tells you whether the value evaluates to `True` or `False`.

Not only does `bool` work with any value, it works with any Python object. The effect of this is that Python's notion of truth extends far beyond the `1` for `True` and the `0` for `False` that other programming languages employ.

Let's pause and take a brief look at `True` and `False` before getting back to our discussion of `return`.

Truth Up Close

Every object in Python has a truth value associated with it, in that the object evaluates to either `True` or `False`.

Something is `False` if it evaluates to `0`, the value `None`, an empty string, or an empty built-in data structure. This means all of these examples are `False`:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(None)
False
```

If an object evaluates to 0, it is always False.

An empty string, an empty list, and an empty dictionary all evaluate to False.

Python's "None" value is always False.

Every other object in Python evaluates to `True`. Here are some examples of objects that are `True`:

```
>>> bool(1)
True
>>> bool(-1)
True
>>> bool(42)
True
>>> bool(0.00000000000000000000000000000001)
True
>>> bool('Panic')
True
>>> bool([42, 43, 44])
True
>>> bool({'a': 42, 'b':42})
True
```

A number that isn't 0 is always True, even when it's negative.

It may be really small, but it is still not 0, so it's True.

A nonempty string is always True.

A nonempty built-in data structure is True.

We can pass any object to the `bool` function and determine whether it is `True` or `False`.

Critically, any nonempty data structure evaluates to `True`.

Returning One Value

Take another look at our function's code, which currently accepts any value as an argument, searches the supplied value for vowels, and then displays the found vowels on screen:

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

We'll change these two lines.

Changing this function to return either `True` or `False`, based on whether any vowels were found, is straightforward. Simply replace the last two lines of code (the `for` loop) with this line of code:

```
return bool(found)
```

Call the "bool" function, and...

...pass in the name of the data structure that contains the results of the vowels search.

If nothing is found, the function returns `False`; otherwise, it returns `True`. With this change made, you can now test this new version of your function at the Python Shell and see what happens:

```
>>> search4vowels('hitch-hiker')
True
>>> search4vowels('galaxy')
True
>>> search4vowels('sky')
False
```

The "return" statement (thanks to "bool") gives us either "True" or "False".

As in earlier chapters, we are not classing 'y' as a vowel.

If you continue to see the previous version's behavior, ensure you've saved the new version of your function, as well as pressed F5 from the edit window.

Geek Bits

Don't be tempted to put parentheses around the object that `return` passes back to the calling code. You don't need to. The `return` statement is not a function call, so the use of parentheses isn't a syntactical requirement. You can use them (if you *really* want to), but most Python programmers don't.

Returning More Than One Value

Functions are designed to return a single value, but it is sometimes necessary to return more than one value. The only way to do this is to package the multiple values in a single data structure, then return that. Thus, you're still returning one thing, even though it potentially contains many individual pieces of data.

Here's our current function, which returns a boolean value (i.e., one thing):

Note: we've updated the comment.

```
def search4vowels(word):
    """Return a boolean based on any vowels found."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return bool(found)
```

It's a trivial edit to have the function return multiple values (in one set) as opposed to a boolean. All we need to do is drop the call to `bool`:

```
def search4vowels(word):
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return found
```

We've updated the comment again.

Return the results as a data structure (a set).

We can further reduce the last two lines of code in the above version of our function to one line by removing the unnecessary use of the `found` variable. Rather than assigning the results of the `intersection` to the `found` variable and returning that, just return the `intersection`:

```
def search4vowels(word):
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Return the data without the use of the unnecessary "found" variable.

Our function now returns a set of vowels found in a word, which is exactly what we set out to do.

However, when we tested it, one of our results has us scratching our head...

Test Drive

Let's take this latest version of the `search4vowels` function for a spin and see how it behaves. With the latest code loaded into an IDLE edit window, press F5 to import the function into the Python Shell, and then invoke the function a few times:

```
Python 3.4.3 Shell
>>> ================================== RESTART ==================================
>>>
>>> search4vowels('hitch-hiker')
{'e', 'i'}
>>> search4vowels('galaxy')
{'a'}
>>> search4vowels('life, the universe and everything')
{'e', 'u', 'a', 'i'}
>>> search4vowels('sky')
set()
>>>
                                                                        Ln: 38 Col: 4
```

Each of these function invocations works as expected, even though the result from the last one looks a little weird.

What's the deal with "set()"?

Each example in the above *Test Drive* works fine, in that the function takes a single string value as an argument, then returns the set of vowels found. The one result, the set, contains many values. However, the last response looks a little weird, doesn't it? Let's have a closer look:

We don't need a function to tell us that the word "sky" doesn't contain any vowels...

```
>>> search4vowels('sky')
set()
```

...but look what our function returns. What gives?

You may have expected the function to return `{}` to represent an empty set, but that's a common misunderstanding, as `{}` represents an empty dictionary, *not* an empty set.

An empty set is represented as `set()` by the interpreter.

This may well look a little weird, but it's just the way things work in Python. Let's take a moment to recall the four built-in data structures, with a eye to seeing how each empty data structure is represented by the interpreter.

Recalling the Built-in Data Structures

Let's remind ourselves of the four built-in data structures available to us. We'll take each data structure in turn, working through list, dictionary, set, and finally tuple.

Working at the shell, let's create an empty data structure using the data structure built-in functions (BIFs for short), then assign a small amount of data to each. We'll then display the contents of each data structure after each assignment:

BIF is short-hand for "built-in function."

```
>>> l = list()
>>> l
[]
>>> l = [ 1, 2, 3 ]
>>> l
[1, 2, 3]
```

An empty list

Use the "list" BIF to define an empty list, then assign some data.

```
>>> d = dict()
>>> d
{}
>>> d = { 'first': 1, 'second': 2, 'third': 3 }
>>> d
{'second': 2, 'third': 3, 'first': 1}
```

An empty dictionary

Use the "dict" BIF to define an empty dictionary, then assign some data.

```
>>> s = set()
>>> s
set()
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
```

An empty set

Use the "set" BIF to define an empty set, then assign some data.

Even though sets are enclosed in curly braces, so too are dictionaries. An empty dictionary is already using the double curly braces, so an empty set has to be represented as "set()".

```
>>> t = tuple()
>>> t
()
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
```

An empty tuple

Use the "tuple" BIF to define an empty tuple, then assign some data.

Before moving on, take a moment to review how the interpreter represents each of the empty data structures as shown on this page.

Use Annotations to Improve Your Docs

Our review of the four data structures confirms that the `search4vowels` function returns a set. But, other than calling the function and checking the return type, how can users of our function know this ahead of time? How do they know what to expect?

A solution is to add this information to the docstring. This assumes that you very clearly indicate in your docstring what the arguments and return value are going to be and that this information is easy to find. Getting programmers to agree on a standard for documenting functions is problematic (PEP 257 only suggests the *format* of docstrings), so Python 3 now supports a notation called **annotations** (also known as *type hints*). When used, annotations document—in a standard way—the return type, as well as the types of any arguments. Keep these points in mind:

1. **Function annotations are optional**
   It's OK not to use them. In fact, a lot of existing Python code doesn't (as they were only made available to programmers in the most recent versions of Python 3).

2. **Function annotations are informational**
   They provide details about your function, but they do not imply any other behavior (such as type checking).

Let's annotate the `search4vowels` function's arguments. The first annotation states that the function expects a string as the type of the `word` argument (`:str`), while the second annotation states that the function returns a set to its caller (`-> set`):

We are stating that the "word" argument is expected to be a string.

We are stating that the function returns a set to its caller.

```
def search4vowels(word:str) -> set:
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Annotation syntax is straightforward. Each function argument has a colon appended to it, together with the type that is expected. In our example, `:str` specifies that the function expects a string. The return type is provided after the argument list, and is indicated by an arrow symbol, which is itself followed by the return type, then the colon. Here `-> set:` indicates that the function is going to return a set.

So far, so good.

We've now annotated our function in a standard way. Because of this, programmers using our function now know what's expected of them, as well as what to expect from the function. However, the interpreter **won't** check that the function is always called with a string, nor will it check that the function always returns a set. Which begs a rather obvious question...

For more details on annotations, see PEP 3107 at https://www.python.org/dev/peps/pep-3107/.

Why Use Function Annotations?

If the Python interpreter isn't going to use your annotations to check the types of your function's arguments and its return type, why bother with annotations at all?

The goal of annotations is *not* to make life easier for the interpreter; it's to make life easier for the user of your function. Annotations are a **documentation standard**, *not* a type enforcement mechanism.

In fact, the interpreter does not care what type your arguments are, nor does it care what type of data your function returns. The interpreter calls your function with whatever arguments are provided to it (no matter their type), executes your function's code, and then returns to the caller whatever value it is given by the `return` statement. The type of the data being passed back and forth is not considered by the interpreter.

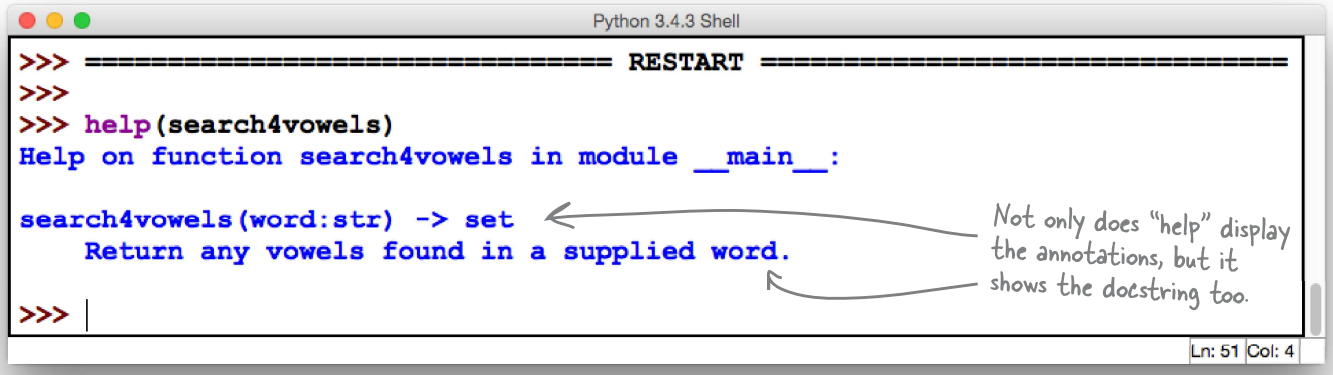
What annotations do for programmers using your function is rid them of the need to read your function's code to learn what types are expected by, and returned from, your function. This is what they'll have to do if annotations aren't used. Even the most beautifully written docstring will still have to be read if it doesn't include annotations.

Which leads to another question: how do we view the annotations without reading the function's code? From IDLE's editor, press F5, then use the `help` BIF at the >>> prompt.

Use annotations to help document your functions, and use the "help" BIF to view them.

Test Drive

If you haven't done so already, use IDLE's editor to annotate your copy of `search4vowels`, save your code, and then press the F5 key. The Python Shell will restart and the >>> prompt will be waiting for you to do something. Ask the `help` BIF to display `search4vowels` documentation, like so:



Functions: What We Know Already

Let's pause for a moment and review what we know (so far) about Python functions.

BULLET POINTS

- Functions are named chunks of code.
- The `def` keyword is used to name a function, with the function's code indented under (and relative to) the `def` keyword.
- Python's triple-quoted strings can be used to add multiline comments to a function. When they are used in this way, they are known as *docstrings*.
- Functions can accept any number of named arguments, including none.
- The `return` statement lets your functions return any number of values (including none).
- Function annotations can be used to document the type of your function's arguments, as well as its return type.

Let's take a moment to once more review the code for the `search4vowels` function. Now that it accepts an argument and returns a set, it is more useful than the very first version of the function from the start of this chapter, as we can now use it in many more places:

The most recent version of our function

```
def search4vowels(word:str) -> set:
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

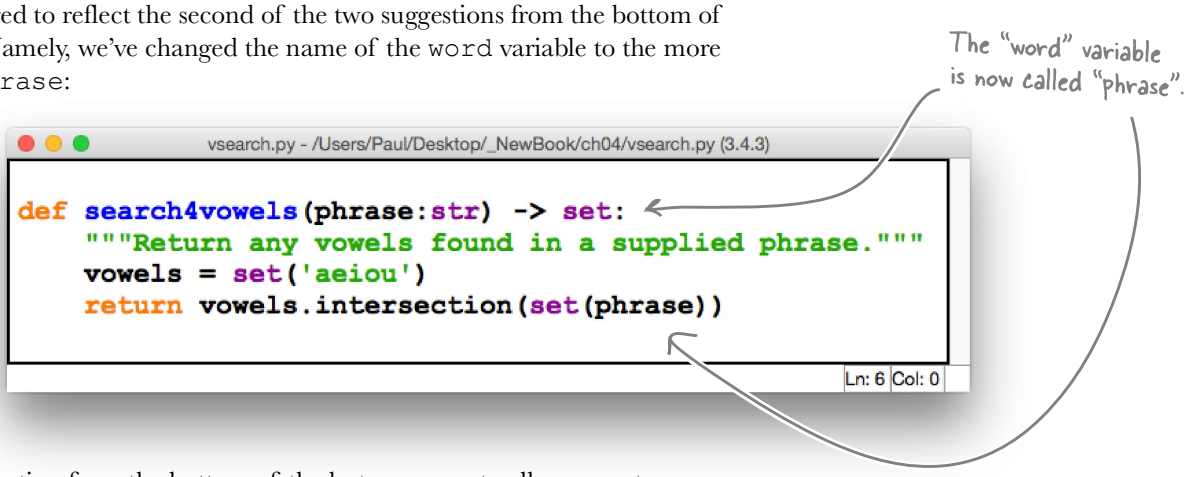
This function would be even more useful if, in addition to accepting an argument for the word to search, it also accepted a second argument detailing what to search for. This would allow us to look for any set of letters, not just the five vowels.

Additionally, the use of the name `word` as an argument name is OK, but not great, as this function clearly accepts *any* string as an argument, as opposed to a single word. A better variable name might be `phrase`, as it more closely matches what it is we expect to receive from the users of our function.

Let's change our function now to reflect this last suggestion.

Making a Generically Useful Function

Here's a version of the `search4vowels` function (as it appears in IDLE) after it has been changed to reflect the second of the two suggestions from the bottom of the last page. Namely, we've changed the name of the `word` variable to the more appropriate `phrase`:



The other suggestion from the bottom of the last page was to allow users to specify the set of letters to search for, as opposed to always using the five vowels. To do this we can add a second argument to the function that specifies the letters to search `phrase` for. This is an easy change to make. However, once we make it, the function (as it stands) will be incorrectly named, as we'll no longer be searching for vowels, we'll be searching for any set of letters. Rather than change the current function, let's create a second one that is based on the first. Here's what we propose to do:

1. **Give the new function a more generic name**
   Rather than continuing to adjust `search4vowels`, let's create a new function called `search4letters`, which is a name that better reflects the new function's purpose.

2. **Add a second argument**
   Adding a second argument allows us to specify the set of letters to search the string for. Let's call the second argument `letters`. And let's not forget to annotate `letters`, too.

3. **Remove the `vowels` variable**
   The use of the name `vowels` in the function's suite no longer makes any sense, as we are now looking for a user-specified set of letters.

4. **Update the docstring**
   There's no point copying, then changing, the code if we don't also adjust the docstring. Our documentation needs be updated to reflect what the new function does.

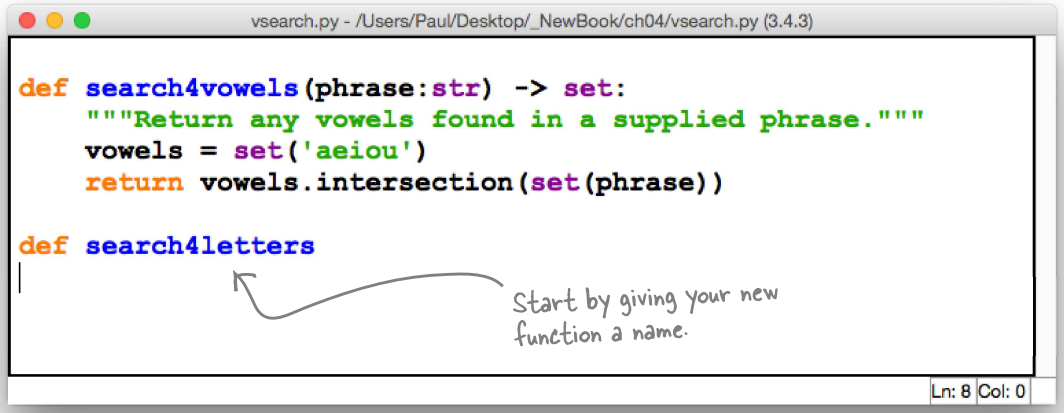
We are going to work through these four tasks together. As each task is discussed, be sure to edit your `vsearch.py` file to reflect the presented changes.

Creating Another Function, 1 of 3

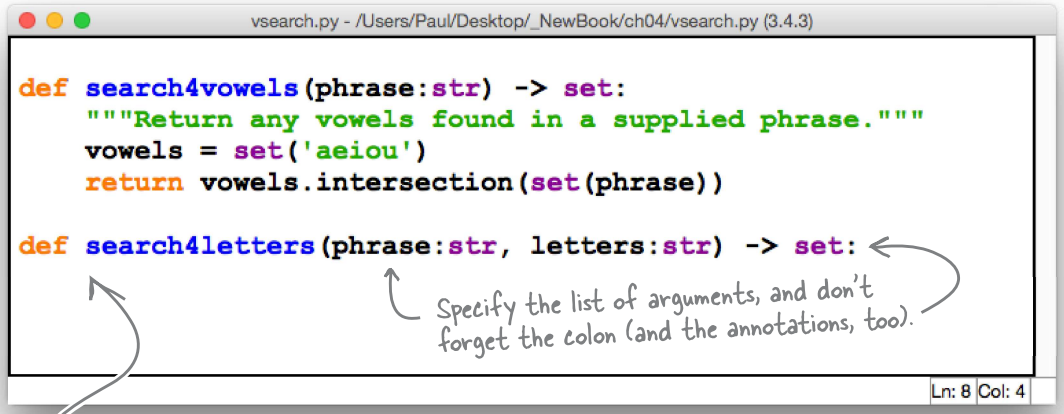
If you haven't done so already, open the `vsearch.py` file in an IDLE edit window.

Step 1 involves creating a new function, which we'll call `search4letters`. Be aware that PEP 8 suggests that all top-level functions are surrounded by two blank lines. All of this book's downloads conform to this guideline, but the code we show on the printed page doesn't (as space is at a premium here).

At the bottom of the file, type ***def*** followed by the name of your new function:



For **Step 2** we're completing the function's `def` line by adding in the names of the two required arguments, `phrase` and `letters`. Remember to enclose the list of arguments within parentheses, and don't forget to include the trailing colon (and the annotations):



Did you notice how IDLE's editor has anticipated that the next line of code needs to be indented (and automatically positioned the cursor)?

With Steps 1 and 2 complete, we're now ready to write the function's code. This code is going to be similar to that in the `search4vowels` function, except that we plan to remove our reliance on the `vowels` variable.

Creating Another Function, 2 of 3

On to **Step 3**, which is to write the code for the function in such a way as to remove the need for the `vowels` variable. We could continue to use the variable, but give it a new name (as `vowels` no longer represents what the variable does), but a temporary variable is not needed here, for much the same reason as why we no longer needed the `found` variable earlier. Take a look at the new line of code in `search4letters`, which does the same job as the two lines in `search4vowels`:

Two lines of code become one.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    return set(letters).intersection(set(phrase))
```

Ln: 9 Col: 0

If that single line of code in `search4letters` has you scratching your head, don't despair. It looks more complex than it is. Let's go through this line of code in detail to work out exactly what it does. It starts when the value of the `letters` argument is turned into a set:

```
set(letters)
```

Create a set object from "letters".

This call to the `set` BIF creates a set object from the characters in the `letters` variable. We don't need to assign this set object to a variable, as we are more interested in using the set of letters right away than in storing the set in a variable for later use. To use the just-created set object, append a dot, then specify the method you want to invoke, as even objects that aren't assigned to variables have methods. As we know from using sets in the last chapter, the `intersection` method takes the set of characters contained in its argument (`phrase`) and intersects them with an existing set object (`letters`):

Perform a set intersection on the set object made from "letters" with the set object made from "phrase".

```
set(letters).intersection(set(phrase))
```

And, finally, the result of the intersection is returned to the calling code, thanks to the `return` statement:

```
return set(letters).intersection(set(phrase))
```

Send the results back to the calling code.

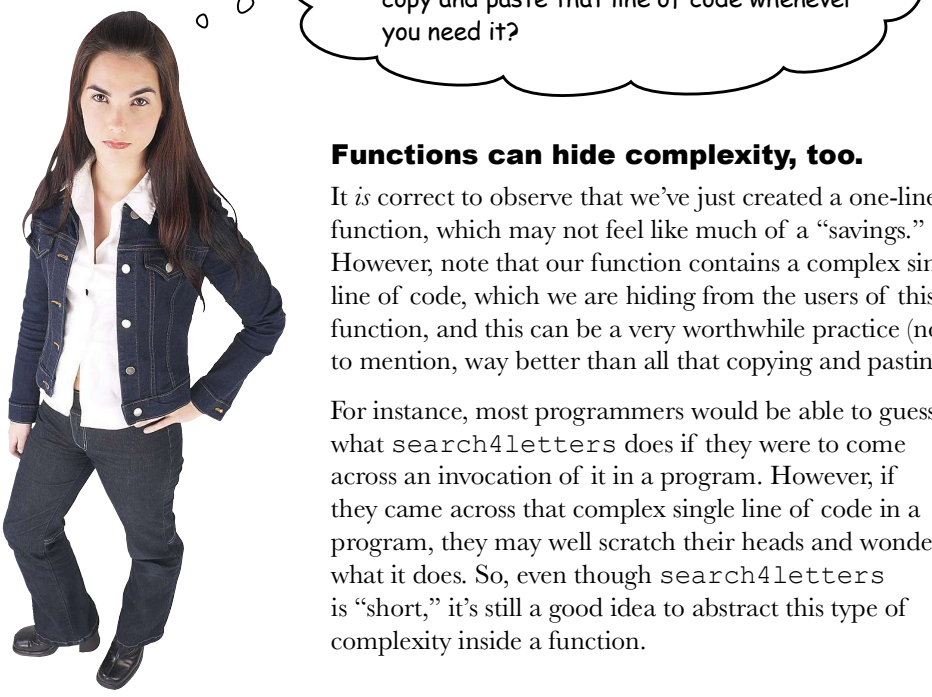
Creating Another Function, 3 of 3

All that remains is **Step 4**, where we add a docstring to our newly created function. To do this, add a triple-quoted string right after your new function's `def` line. Here's what we used (as comments go it's terse, but effective):

A docstring

```
vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))

Ln: 10  Col: 0
```

And with that, our four steps are complete and `search4letters` is ready to be tested.



Functions can hide complexity, too.

It *is* correct to observe that we've just created a one-line function, which may not feel like much of a "savings." However, note that our function contains a complex single line of code, which we are hiding from the users of this function, and this can be a very worthwhile practice (not to mention, way better than all that copying and pasting).

For instance, most programmers would be able to guess what `search4letters` does if they were to come across an invocation of it in a program. However, if they came across that complex single line of code in a program, they may well scratch their heads and wonder what it does. So, even though `search4letters` is "short," it's still a good idea to abstract this type of complexity inside a function.

Test Drive

Save the `vsearch.py` file once more, and then press F5 to try out the `search4letters` function:

```
Python 3.4.3 Shell
>>> ========== RESTART ==========
>>>
>>> help(search4letters)
Help on function search4letters in module __main__:

search4letters(phrase:str, letters:str) -> set
    Return a set of the 'letters' found in 'phrase'.

>>> search4letters('hitch-hiker', 'aeiou')
{'e', 'i'}
>>> search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> search4letters('life, the universe, and everything', 'o')
set()
>>>
Ln: 78 Col: 4
```

Use the "help" BIF to learn how to use "search4letters".

All of these examples produce what we expect them to.

The `search4letters` function is now more generic than `search4vowels`, in that it takes *any* set of letters and searches a given phrase for them, rather than just searching for the letters `a`, `e`, `i`, `o`, and `u`. This makes our new function much more useful than `search4vowels`. Let's now imagine that we have a large, existing codebase that has used `search4vowels` extensively. A decision has been made to retire `search4vowels` and replace it with `search4letters`, as the "powers that be" don't see the need for both functions, now that `search4letters` can do what `search4vowels` does. A global search-and-replace of your codebase for the name "search4vowels" with "search4letters" won't work here, as you'll need to add in that second argument value, which is always going to be `aeiou` when simulating the behavior of `search4vowels` with `search4letters`. So, for instance, this single-argument call:

```
search4vowels("Don't panic!")
```

now needs to be replaced with this dual-argument one (which is a much harder edit to automate):

```
search4letters("Don't panic!", 'aeiou')
```

It would be nice if we could somehow specify a *default value* for `search4letters`'s second argument, then have the function use it if no alternative value is provided. If we could arrange to set the default to `aeiou`, we'd then be able to apply a global search-and-replace (which is an easy edit).

Specifying Default Values for Arguments

Any argument to a Python function can be assigned a default value, which can then be automatically used if the code calling the function fails to supply an alternate value. The mechanism for assigning a default value to an argument is straightforward: include the default value as an assignment in the function's `def` line.

Here's `search4letters`'s current `def` line:

```
def search4letters(phrase:str, letters:str) -> set:
```

This version of our function's `def` line (above) expects *exactly* two arguments, one for `phrase` and another for `letters`. However, if we assign a default value to `letters`, the function's `def` line changes to look like this:

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

A default value has been assigned to the "letters" argument and will be used whenever the calling code doesn't provide an alternate value.

We can continue to use the `search4letters` function in the same way as before: providing both arguments with values as needed. However, if we forget to supply the second argument (`letters`), the interpreter will substitute in the value `aeiou` on our behalf.

If we were to make this change to our code in the `vsearch.py` file (and save it), we could then invoke our functions as follows:

These three function calls all produce the same results.

```
>>> search4letters('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
>>> search4letters('life, the universe, and everything', 'aeiou')
{'a', 'e', 'i', 'u'}
>>> search4vowels('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
```

In this invocation, we are calling "search4vowels", not "search4letters".

Not only do these function calls produce the same output, they also demonstrate that the `search4vowels` function is no longer needed now that the `letters` argument to `search4letters` supports a default value (compare the first and last invocations above).

Now, if we are asked to retire the `search4vowels` function and replace all invocations of it within our codebase with `search4letters`, our exploitation of the default value mechanism for function arguments lets us do so with a simple global search-and-replace. And we don't have to use `search4letters` to only search for vowels. That second argument allows us to specify *any* set of characters to look for. As a consequence, `search4letters` is now more generic, *and* more useful.

Positional Versus Keyword Assignment

As we've just seen, the `search4letters` function can be invoked with either one or two arguments, the second argument being optional. If you provide only one argument, the `letters` argument defaults to a string of vowels. Take another look at the function's `def` line:

Our function's "def" line

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

As well as supporting default arguments, the Python interpreter also lets you invoke a function using **keyword arguments**. To understand what a keyword argument is, consider how we've invoked `search4letters` up until now, for example:

```
search4letters('galaxy', 'xyz')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

In the above invocation, the two strings are assigned to the `phrase` and `letters` arguments based on their position. That is, the first string is assigned to `phrase`, while the second is assigned to `letters`. This is known as **positional assignment**, as it's based on the order of the arguments.

In Python, it is also possible to refer to arguments by their argument name, and when you do, positional ordering no longer applies. This is known as **keyword assignment**. To use keywords, assign each string *in any order* to its correct argument name when invoking the function, as shown here:

The ordering of the arguments isn't important when keyword arguments are used during invocation.

```
search4letters(letters='xyz', phrase='galaxy')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

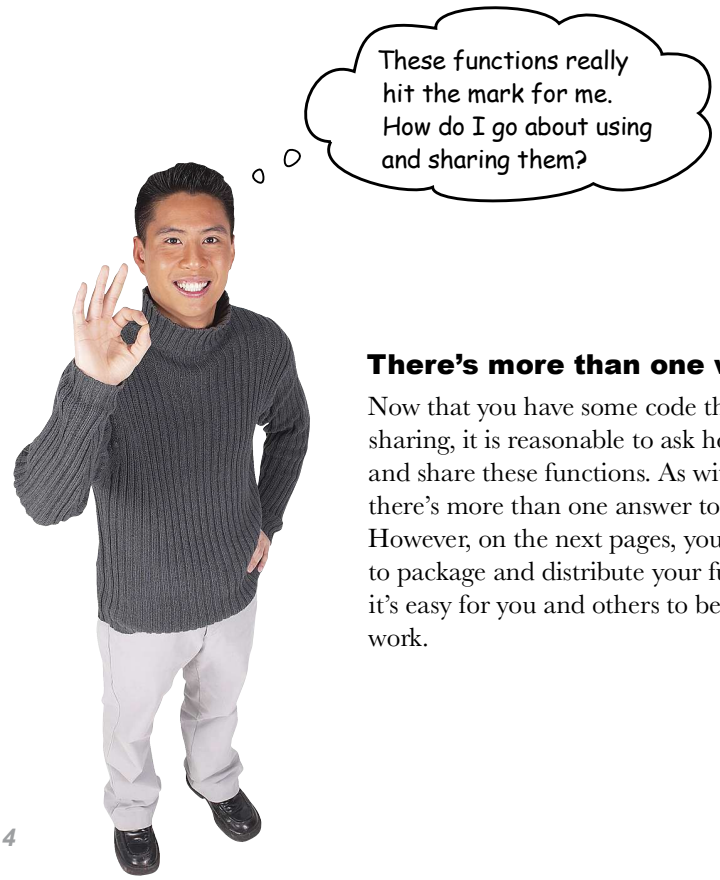
Both invocations of the `search4letters` function on this page produce the same result: a set containing the letters `y` and `z`. Although it may be hard to appreciate the benefit of using keyword arguments with our small `search4letters` function, the flexibility this feature gives you becomes clear when you invoke a function that accepts many arguments. We'll see an example of one such function (provided by the standard library) before the end of this chapter.

Updating What We Know About Functions

Let's update what you know about functions now that you've spent some time exploring how function arguments work:

BULLET POINTS

- As well as supporting code reuse, functions can hide complexity. If you have a complex line of code you intend to use a lot, abstract it behind a simple function call.
- Any function argument can be assigned a default value in the function's `def` line. When this happens, the specification of a value for that argument during a function's invocation is optional.
- As well as assigning arguments by position, you can use keywords, too. When you do, any ordering is acceptable (as any possibility of ambiguity is removed by the use of keywords and position doesn't matter anymore).



There's more than one way to do it.

Now that you have some code that's worth sharing, it is reasonable to ask how best to use and share these functions. As with most things, there's more than one answer to that question. However, on the next pages, you'll learn how best to package and distribute your functions to ensure it's easy for you and others to benefit from your work.

Functions Beget Modules

module

Having gone to all the trouble of creating a reusable function (or two, as is the case with the functions currently in our `vsearch.py` file), it is reasonable to ask: *what's the best way to share functions?*

It is possible to share any function by copying and pasting it throughout your codebase where needed, but as that's such a wasteful and bad idea, we aren't going to consider it for very much longer. Having multiple copies of the same function littering your codebase is a sure-fire recipe for disaster (should you ever decide to change how your function works). It's much better to create a **module** that contains a single, canonical copy of any functions you want to share. Which raises another question: *how are modules created in Python?*

Share your functions in modules.

The answer couldn't be simpler: a module is any file that contains functions. Happily, this means that `vsearch.py` is *already* a module. Here it is again, in all its module glory:

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Ln: 10 Col: 0

"vsearch.py" contains functions in a file, making it a fully formed module.

Creating modules couldn't be easier, however...

Creating modules is a piece of cake: simply create a file of the functions you want to share.

Once your module exists, making its contents available to your programs is also straightforward: all you have to do is import the module using Python's `import` statement.

This in itself is not complex. However, the interpreter makes the assumption that the module in question is in the **search path**, and ensuring this is the case can be tricky. Let's explore the ins and outs of module importation over the next few pages.

How Are Modules Found?

Recall from this book's first chapter how we imported and then used the `randint` function from the `random` module, which comes included as part of Python's standard library. Here's what we did at the shell:

Identify the module to import, then...

```
>>> import random
>>> random.randint(0, 255)
42
```

...invoke one of the module's functions.

What happens during module importation is described in great detail in the Python documentation, which you are free to go and explore if the nitty-gritty details float your boat. However, all you really need to know are the three main locations the interpreter searches when looking for a module. These are:

1. **Your current working directory**
   This is the folder that the interpreter thinks you are currently working in.

2. **Your interpreter's site-packages locations**
   These are the directories that contain any third-party Python modules you may have installed (including any written by you).

3. **The standard library locations**
   These are the directories that contains all the modules that make up the standard library.

Geek Bits

Depending on the operating system you're running, the name given to a location that holds files may be either **directory** or **folder**. We'll use "folder" in this book, except when we discuss the *current working directory* (which is a well-established term).

The order in which locations 2 and 3 are searched by the interpreter can vary depending on many factors. But don't worry: it is not important that you know how this searching mechanism works. What *is* important to understand is that the interpreter always searches your current working directory *first*, which is what can cause trouble when you're working with your own custom modules.

To demonstrate what can go wrong, let's run though a small exercise that is designed to highlight the issue. Here's what you need to do before we begin:

- ☐ Create a folder called `mymodules`, which we'll use to store your modules. It doesn't matter where in your filesystem you create this folder; just make sure it is somewhere where you have read/write access.

- ☐ Move your `vsearch.py` file into your newly created `mymodules` folder. This file should be the only copy of the `vsearch.py` file on your computer.

Running Python from the Command Line

We're going to run the Python interpreter from your operating system's command line (or terminal) to demonstrate what can go wrong here (even though the problem we are about to discuss also manifests in IDLE).

If you are running any version of *Windows*, open up a command prompt and follow along with this session. If you are not on *Windows*, we discuss your platform halfway down the next page (but read on for now anyway). You can invoke the Python interpreter (outside of IDLE) by typing ***py -3*** at the *Windows* `C:\>` prompt. Note below how prior to invoking the interpreter, we use the `cd` command to make the `mymodules` folder our current working directory. Also, observe that we can exit the interpreter at any time by typing `quit()` at the >>> prompt:

Change into the "mymodules" folder.

Start Python 3.

Import the module.

Use the module's functions.

Exit the Python interpreter and return to your operating system's command prompt.

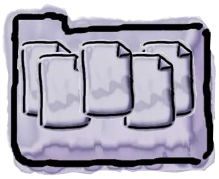
```
File Edit Window Help Redmond #1
C:\Users\Head First> cd mymodules

C:\Users\Head First\mymodules> py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'y', 'x'}
>>> quit()

C:\Users\Head First\mymodules>
```

This works as expected: we successfully import the `vsearch` module, then use each of its functions by prefixing the function name with the name of its module and a dot. Note how the behavior of the >>> prompt at the command line is identical to the behavior within IDLE (the only difference is the lack of syntax highlighting). It's the same Python interpreter, after all.

Although this interaction with the interpreter was successful, it only worked because we started off in a folder that contained the `vsearch.py` file. Doing this makes this folder the current working directory. Based on how the interpreter searches for modules, we know that the current working directory is searched first, so it shouldn't surprise us that this interaction worked and that the interpreter found our module.

But what happens if our module isn't in the current working directory?

Not Found Modules Produce ImportErrors



Repeat the exercise from the last page, after moving out of the folder that contains our module. Let's see what happens when we try to import our module now. Here is another interaction with the *Windows* command prompt:

Change to another folder (in this case, we are moving to the top-level folder).

Start Python 3 again.

Try to import the module...

...but this time we get an error!

```
File Edit Window Help Redmond #2
C:\Users\Head First> cd \

C:\>py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

C:\>
```

The `vsearch.py` file is no longer in the interpreter's current working directory, as we are now working in a folder other than `mymodules`. This means our module file can't be found, which in turn means we can't import it—hence the `ImportError` from the interpreter.

If we try the same exercise on a platform other than *Windows*, we get the same results (whether we're on *Linux*, *Unix*, or *Mac OS X*). Here's the above interaction with the interpreter from within the `mymodules` folder on *OS X*:

Change into the folder and then type "python3" to start the interpreter.

Import the module.

It works: we can use the module's functions.

```
File Edit Window Help Cupertino #1
$ cd mymodules

mymodules$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> quit()

mymodules$
```

Exit the Python interpreter and return to your operating system's command prompt.

ImportErrors Occur No Matter the Platform

If you think running on a non-*Windows* platform will somehow fix this import issue we saw on that platform, think again: the same `ImportError` occurs on UNIX-like systems, once we change to another folder:

Change to another folder (in this case, we are moving to our top-level folder).

Start Python 3 again.

Try to import the module...

...but this time we get an error!

```
File Edit Window Help Cupertino #2
mymodules$ cd

$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

$
```

As was the case when we were working on *Windows*, the `vsearch.py` file is no longer in the interpreter's current working directory, as we are now working in a folder other than `mymodules`. This means our module file can't be found, which in turn means we can't import it—hence the `ImportError` from the interpreter. This problem presents no matter which platform you're running Python on.

there are no Dumb Questions

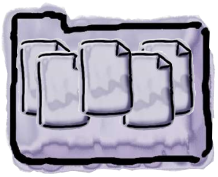
Q: Can't we be location specific and say something like `import C:\mymodules\vsearch` on Windows platforms, or perhaps `import /mymodules/vsearch` on UNIX-like systems?

A: No, you can't. Granted, doing something like that does sound tempting, but ultimately won't work, as you can't use paths in this way with Python's `import` statement. And, anyway, the last thing you'll want to do is put hardcoded paths into any of your programs, as paths can often change (for a whole host of reasons). It is best to avoid hardcoding paths in your code, if at all possible.

Q: If I can't use paths, how can I arrange for the interpreter to find my modules?

A: If the interpreter can't find your module in the current working directory, it looks in the **site-packages** locations as well as in the standard library (and there's more about site-packages on the next page). If you can arrange to add your module to one of the **site-packages** locations, the interpreter can then find it there (no matter its path).

Getting a Module into Site-packages


module

Recall what we had to say about **site-packages** a few pages back when we introduced them as the second of three locations searched by the interpreter's import mechanism:

2. **Your interpreter's site-packages locations**
   These are the directories that contain any third-party Python modules which you may have installed (including any written by you).

As the provision and support of third-party modules is central to Python's code reuse strategy, it should come as no surprise that the interpreter comes with the built-in ability to add modules to your Python setup.

Note that the set of modules included with the standard library is managed by the Python core developers, and this large collection of modules has been designed to be widely used, but not tampered with. Specifically, don't add or remove your own modules to/from the standard library. However, adding or removing modules to your site-packages locations is positively encouraged, so much so that Python comes with some tools to make it straightforward.

Using "setuptools" to install into site-packages

As of release 3.4 of Python, the standard library includes a module called `setuptools`, which can be used to add any module into site-packages. Although the details of module distribution can—initially—appear complex, all we want to do here is install `vsearch` into site-packages, which is something `setuptools` is more than capable of doing in three steps:

Python 3.4 (or newer) makes using setuptools a breeze. If you aren't running 3.4 (or newer), consider upgrading.

1. **Create a distribution description**
   This identifies the module we want `setuptools` to install.
2. **Generate a distribution file**
   Using Python at the command line, we'll create a shareable distribution file to contain our module's code.
3. **Install the distribution file**
   Again, using Python at the command line, install the distribution file (which includes our module) into site-packages.

Step 1 requires us to create (at a minimum) two descriptive files for our module: `setup.py` and `README.txt`. Let's see what's involved.

Creating the Required Setup Files

- [ ] Create a distribution description.
- [ ] Generate a distribution file.
- [ ] Install the distribution file.

We'll check off each completed step as we work through this material.

If we follow the three steps shown at the bottom of the last page, we'll end up creating a **distribution package** for our module. This package is a single compressed file that contains everything required to install our module into site-packages.

For Step 1, *Create a distribution description*, we need to create two files that we'll place in the same folder as our `vsearch.py` file. We'll do this no matter what platform we're running on. The first file, which must be called `setup.py`, describes our module in some detail.

Find below the `setup.py` file we created to describe the module in the `vsearch.py` file. It contains two lines of Python code: the first line imports the `setup` function from the `setuptools` module, while the second invokes the `setup` function.

The `setup` function accepts a large number of arguments, many of which are optional. Note how, for readability purposes, our call to `setup` is spread over nine lines. We're taking advantage of Python's support for keyword arguments to clearly indicate which value is being assigned to which argument in this call. The most important arguments are highlighted; the first names the distribution, while the second lists the `.py` files to include when creating the distribution package:

Import the "setup" function from the "setuptools" module.

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='The Head First Python Search Tools',
    author='HF Python 2e',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

The "name" argument identifies the distribution. It's common practice to name the distribution after the module.

This is an invocation of the "setup" function. We're spreading its arguments over many lines.

This is a list of ".py" files to include in the package. For this example, we only have one: "vsearch".

In addition to `setup.py`, the `setuptools` mechanism requires the existence of one other file—a "readme" file—into which you can put a textual description of your package. Although having this file is required, its contents are optional, so (for now) you can create an empty file called `README.txt` in the same folder as the `setup.py` file. This is enough to satisfy the requirement for a second file in Step 1.

Creating the Distribution File

- [x] Create a distribution description.
- [ ] Generate a distribution file.
- [ ] Install the distribution file.

At this stage, you should have three files, which we have put in our `mymodules` folder: `vsearch.py`, `setup.py`, and `README.txt`.

We're now ready to create a distribution package from these files. This is Step 2 from our earlier list: *Generate a distribution file*. We'll do this at the command line. Although doing so is straightforward, this step requires that different commands be entered based on whether you are on *Windows* or on one of the UNIX-like operating systems (*Linux*, *Unix*, or *Mac OS X*).

Creating a distribution file on Windows

If you are running on *Windows*, open a command prompt in the folder that contains your three files, then enter this command:

Run Python 3 on Windows.

```
C:\Users\Head First\mymodules> py -3 setup.py sdist
```

Execute the code in "setup.py"...

... and pass "sdist" as an argument.

The Python interpreter goes to work immediately after you issue this command. A large number of messages appear on screen (which we show here in an abridged form):

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
creating dist
creating 'dist\vsearch-1.0.zip' and adding 'vsearch-1.0' to it
adding 'vsearch-1.0\PKG-INFO'
adding 'vsearch-1.0\README.txt'
      ...
adding 'vsearch-1.0\vsearch.egg-info\top_level.txt'
removing 'vsearch-1.0' (and everything under it)
```

If you see this message, all is well. If you get errors, check that you're running at least Python 3.4, and also make sure your "setup.py" file is identical to ours.

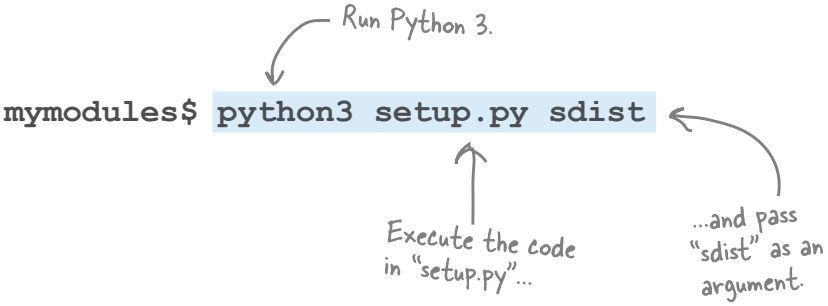
When the *Windows* command prompt reappears, your three files have been combined into a single **distribution file**. This is an installable file that contains the source code for your module and, in this case, is called `vsearch-1.0.zip`.

You'll find your newly created ZIP file in a folder called `dist`, which has also been created by `setuptools` under the folder you are working in (which is `mymodules` in our case).

Distribution Files on UNIX-like OSes

- [x] Create a distribution description.
- [ ] Generate a distribution file.
- [ ] Install the distribution file.

If you are not working on *Windows*, you can create a distribution file in much the same way as on the previous page. With the three files (`setup.py`, `README.txt`, and `vsearch.py`) in a folder, issue this command at your operating system's command line:



Like on *Windows*, this command produces a slew of messages on screen:

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
running check
creating vsearch-1.0
creating vsearch-1.0/vsearch.egg-info
      ...
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

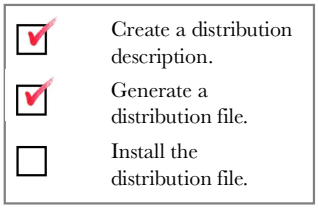
The messages differ slightly from those produced on Windows. If you see this message, all is well. If not (as with Windows) double-check everything.

When your operating system's command line reappears, your three files have been combined into a **source distribution** file (hence the `sdist` argument above). This is an installable file that contains the source code for your module and, in this case, is called `vsearch-1.0.tar.gz`.

You'll find your newly created archive file in a folder called `dist`, which has also been created by `setuptools` under the folder you are working in (which is `mymodules` in our case).

With your source distribution file created (as a ZIP or as a compressed tar archive), you're now ready to install your module into site-packages.

Installing Packages with "pip"



Now that your distribution file exists as a ZIP or a tarred archive (depending on your platform), it's time for Step 3: *Install the distribution file*. As with many such things, Python comes with the tools to make this straightforward. In particular, Python 3.4 (and newer) includes a tool called `pip`, which is *the* **P**ackage **I**nstaller for **P**ython.

Step 3 on Windows

Locate your newly created ZIP file under the `dist` folder (recall that the file is called `vsearch-1.0.zip`). While in the *Windows Explorer*, hold down the Shift key, then right-click your mouse to bring up a context-sensitive menu. Select *Open command window here* from this menu. A new *Windows* command prompt opens. At this command prompt, type this line to complete Step 3:

Run Python 3 with the module pip, and then ask pip to install the identified ZIP file.

```
C:\Users\...\dist> py -3 -m pip install vsearch-1.0.zip
```

If this command fails with a permissions error, you may need to restart the command prompt as the *Windows* administrator, then try again.

When the above command succeeds, the following messages appear on screen:

```
Processing c:\users\...\dist\vsearch-1.0.zip
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Success!

Step 3 on UNIX-like OSes

On *Linux*, *Unix*, or *Mac OS X*, open a terminal within the newly created `dict` folder, and then issue this command at the prompt:

Run Python 3 with the module pip, and then ask pip to install the identified compressed tar file.

```
.../dist$ sudo python3 -m pip install vsearch-1.0.tar.gz
```

We are using the "sudo" command here to ensure we install with the correct permissions.

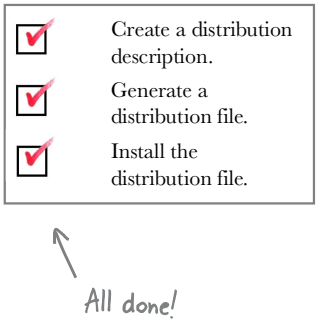
When the above command succeeds, the following messages appear on screen:

```
Processing ./vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Success!

The vsearch module is now installed as part of site-packages.

Modules: What We Know Already



Now that our `vsearch` module has been installed, we can use `import vsearch` in any of our programs, safe in the knowledge that the interpreter can now find the module's functions when needed.

If we later decide to update any of the module's code, we can repeat these three steps to install any update into site-packages. If you do produce a new version of your module, be sure to assign a new version number within the `setup.py` file.

Let's take a moment to summarize what we now know about modules:

BULLET POINTS

- A module is one or more functions saved in a file.
- You can share a module by ensuring it is always available with the interpreter's *current working directory* (which is possible, but brittle) or within the interpreter's *site-packages locations* (by far the better choice).
- Following the `setuptools` three-step process ensures that your module is installed into *site-packages*, which allows you to `import` the module and use its functions no matter what your *current working directory* happens to be.

Giving your code away (a.k.a. sharing)

Now that you have a distribution file created, you can share this file with other Python programmers, allowing them to install your module using `pip`, too. You can share your file in one of two ways: informally, or formally.

To share your module informally, simply distribute it in whatever way you wish and to whomever you wish (perhaps using email, a USB stick, or via a download from your personal website). It's up to you, really.

To share your module formally, you can upload your distribution file to Python's centrally managed web-based software repository, called PyPI (pronounced "pie-pee-eye," and short for the *Python Package Index*). This site exists to allow all manner of Python programmers to share all manner of third-party Python modules. To learn more about what's on offer, visit the PyPI site at: ***https://pypi.python.org/pypi***. To learn more about the process of uploading and sharing your distribution files through PyPI, read the online guide maintained by the *Python Packaging Authority*, which you'll find here: ***https://www.pypa.io***. (There's not much to it, but the details are beyond the scope of this book.)

We are nearly done with our introduction to functions and modules. There's just a small mystery that needs our attention (for not more than five minutes). Flip the page when you're ready.

Any Python programmer can also use pip to install your module.

The case of the misbehaving function arguments

Tom and Sarah have just worked through this chapter, and are now arguing over the behavior of function arguments.

Tom is convinced that when arguments are passed into a function, the data is passed **by value**, and he's written a small function called `double` to help make his case. Tom's `double` function works with any type of data provided to it.

Here's Tom's code:


```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)
```

Sarah, on the other hand, is convinced that when arguments are passed into a function, the data is passed **by reference**. Sarah has also written a small function, called `change`, which works with lists and helps to prove her point.

Here's a copy of Sarah's code:

```
def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

We'd rather nobody was arguing about this type of thing, as—until now—Tom and Sarah have been the best of programming buddies. To help resolve this, let's experiment at the >>> prompt in an attempt to see who is right: "by value" Tom, or "by reference" Sarah. They can't both be right, can they? It's certainly a bit of a mystery that needs solving, which leads to this often-asked question:

Do function arguments support by-value or by-reference call semantics in Python?

Geek Bits

In case you need a quick refresher, note that **by-value argument passing** refers to the practice of using the value of a variable in place of a function's argument. If the value changes in the function's suite, it has no effect on the value of the variable in the code that called the function. Think of the argument as a *copy* of the original variable's value. **By-reference argument passing** (sometimes referred to as **by-address argument passing**) maintains a link to the variable in the code that called the function. If the variable in the function's suite is changed, the value in the code that called the function changes, too. Think of the argument as an *alias* to the original variable.

Demonstrating Call-by-Value Semantics

To work out what Tom and Sarah are arguing about, let's put their functions into their very own module, which we'll call `mystery.py`. Here's the module in an IDLE edit window:

These two functions are similar. Each takes a single argument, displays it on screen, manipulates its value, and then displays it on screen again.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

This function doubles the value passed in.

This function appends a string to any passed in list.

Ln: 11 Col: 0

As soon as Tom sees this module on screen, he sits down, takes control of the keyboard, presses F5, and then types the following into IDLE's >>> prompt. Once done, Tom leans back in his chair, crosses his arms, and says: "See? I told you it's call-by-value." Take a look at Tom's shell interactions with his function:

Tom invokes the "double" function three times: once with an integer value, then with a string, and finally with a list.

```
>>> num = 10
>>> double(num)
Before:  10
After:   20
>>> num
10
>>> saying = 'Hello '
>>> double(saying)
Before:  Hello
After:   Hello Hello
>>> saying
'Hello '
>>> numbers = [ 42, 256, 16 ]
>>> double(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 42, 256, 16]
>>> numbers
[42, 256, 16]
```

Each invocation confirms that the value passed in as an argument is changed within the function's suite, but that the value at the shell remains unchanged. That is,: the function arguments appear to conform to call-by-value semantics.

Demonstrating Call-by-Reference Semantics

Undeterred by Tom's apparent slam-dunk, Sarah sits down and takes control of the keyboard in preparation for interacting with the shell. Here's the code in the IDLE edit window once more, with Sarah's `change` function ready for action:

The is the "mystery.py" module.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Tom's function

Sarah's function

Ln: 11 Col: 0

Sarah types a few lines of code into the >>> prompt, then leans back in her chair, crosses her arms, and says to Tom: "Well, if Python only supports call-by-value, how do you explain this behavior?" Tom is speechless.

Take a look at Sarah's interaction with the shell:

Using the same list data as Tom, Sarah invokes her "change" function.

```
>>> numbers = [ 42, 256, 16 ]
>>> change(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 'More data']
>>> numbers
[42, 256, 16, 'More data']
```

Look what's happened! This time the argument's value has been changed in the function as well as at the shell. This would seem to suggest that Python functions *also* support call-by-reference semantics.

This *is* strange behavior.

Tom's function clearly shows call-by-value argument semantics, whereas Sarah's function demonstrates call-by-reference.

How can this be? What's going on here? Does Python support *both*?

Solved: the case of the misbehaving function arguments

Do Python function arguments support by-value or by-reference call semantics?

Here's the kicker: both Tom *and* Sarah are right. Depending on the situation, Python's function argument semantics support **both** call-by-value *and* call-by-reference.

Recall once again that variables in Python aren't variables as we are used to thinking about them in other programming languages; variables are **object references**. It is useful to think of the value stored in the variable as being the memory address of the value, not its actual value. It's this memory address that's passed into a function, not the actual value. This means that Python's functions support what's more correctly called *by-object-reference call semantics*.

Based on the type of the object referred to, the actual call semantics that apply at any point in time can differ. So, how come in Tom's and Sarah's functions the arguments appeared to conform to by-value and by-reference call semantics? First off, they didn't—they only appeared to. What actually happens is that the interpreter looks at the type of the value referred to by the object reference (the memory address) and, if the variable refers to a **mutable** value, call-by-reference semantics apply. If the type of the data referred to is **immutable**, call-by-value semantics kick in. Consider now what this means for our data.

Lists, dictionaries, and sets (being mutable) are always passed into a function by reference—any changes made to the variable's data structure within the function's suite are reflected in the calling code. The data is mutable, after all.

Strings, integers, and tuples (being immutable) are always passed into a function by value—any changes to the variable within the function are private to the function and are not reflected in the calling code. As the data is immutable, it cannot change.

Which all makes sense until you consider this line of code:

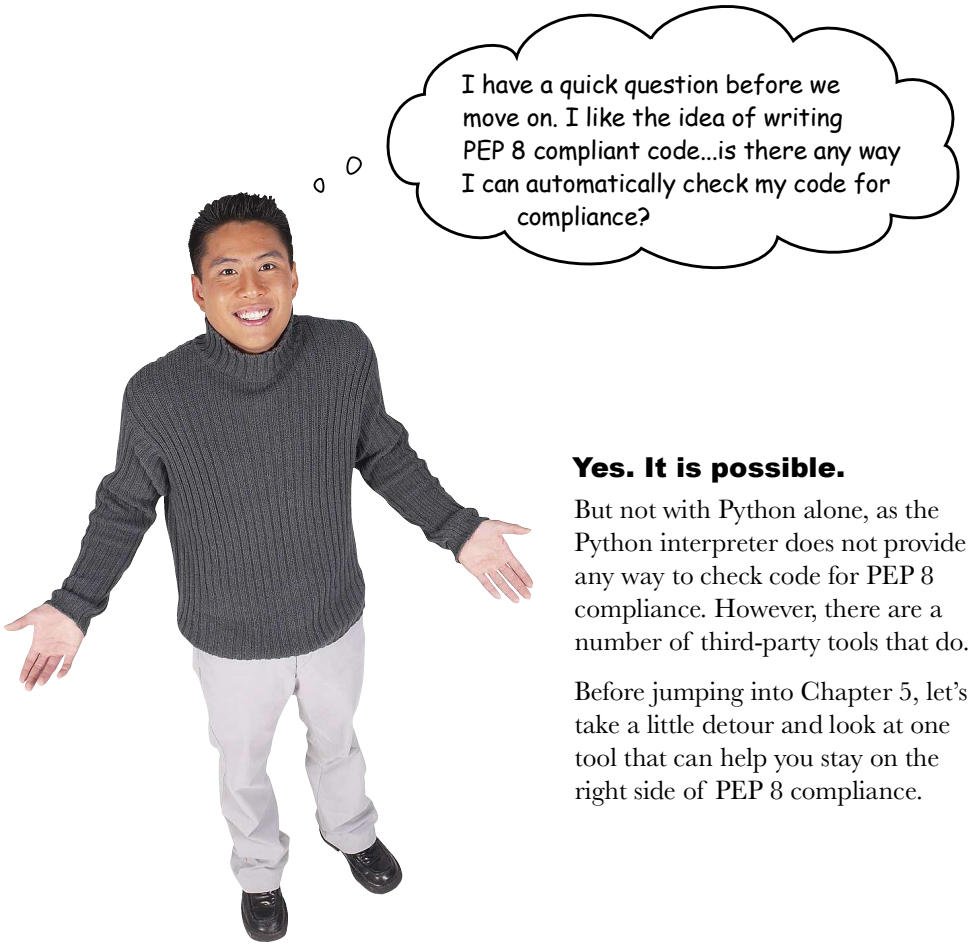
```
arg = arg * 2
```

How come this line of code appeared to change a passed-in list within the function's suite, but when the list was displayed in the shell after invocation, the list hadn't changed (leading Tom to believe—incorrectly—that all argument passing conformed to call-by-value)? On the face of things, this looks like a bug in the interpreter, as we've just stated that changes to a mutable value are reflected back in the calling code, but they aren't here. That is, Tom's function *didn't* change the `numbers` list in the calling code, even though lists are mutable. So, what gives?

To understand what has happened here, consider that the above line of code is an **assignment statement**. Here's what happens during assignment: the code to the right of the = symbol is executed *first*, and then whatever value is created has its object reference assigned to the variable on the left of the = symbol. Executing the code `arg * 2` creates a *new* value, which is assigned a *new* object reference, which is then assigned to the `arg` variable, overwriting the previous object reference stored in `arg` in the function's suite. However, the "old" object reference still exists in the calling code and its value hasn't changed, so the shell still sees the original list, not the new doubled list created in Tom's code. Contrast this behavior to Sarah's code, which calls the `append` method on an existing list. As there's no assignment here, there's no overwriting of object references, so Sarah's code changes the list in the shell, too, as both the list referred to in the functions' suite and the list referred to in the calling code have the *same* object reference.

With our mystery solved, we're nearly ready for Chapter 5. There's just one outstanding issue.

Can I Test for PEP 8 Compliance?



Yes. It is possible.

But not with Python alone, as the Python interpreter does not provide any way to check code for PEP 8 compliance. However, there are a number of third-party tools that do.

Before jumping into Chapter 5, let's take a little detour and look at one tool that can help you stay on the right side of PEP 8 compliance.

Getting Ready to Check PEP 8 Compliance

Let's detour for just a moment to check our code for PEP 8 compliance.

The Python programming community at large has spent a great deal of time creating developer tools to make the lives of Python programmers a little bit better. One such tool is **pytest**, which is a *testing framework* that is primarily designed to make the testing of Python programs easier. No matter what type of tests you're writing, **pytest** can help. And you can add plug-ins to **pytest** to extend its capabilities.

One such plug-in is **pep8**, which uses the **pytest** testing framework to check your code for violations of the PEP 8 guidelines.

Learn more about pytest from http://doc.pytest.org/en/latest/.

Recalling our code

Let's remind ourselves of our `vsearch.py` code once more, before feeding it to the **pytest**/**pep8** combination to find out how PEP 8–compliant it is. Note that we'll need to install both of these developer tools, as they do not come installed with Python (we'll do that over the page).

One more, here is the code to the `vsearch.py` module, which is going to be checked for compliance to the PEP 8 guidelines:

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

This code is in "vsearch.py".

Installing pytest and the pep8 plug-in

Earlier in this chapter, you used the `pip` tool to install your `vsearch.py` module into the Python interpreter on your computer. The `pip` tool can also be used to install third-party code into your interpreter.

To do so, you need to operate at your operating system's command prompt (and be connected to the Internet). You'll use `pip` in the next chapter to install a third-party library. For now, though, let's use `pip` to install the **pytest** testing framework and the **pep8** plug-in.

Install the Testing Developer Tools

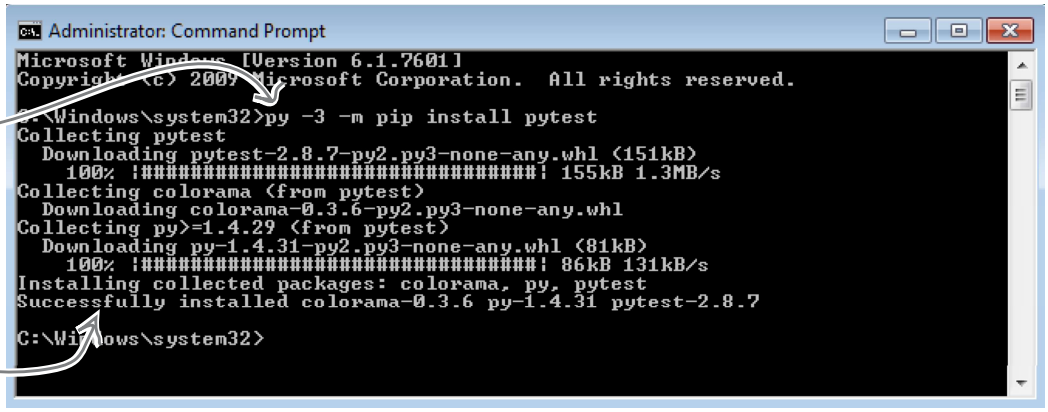
In the example screens that follow, we are showing the messages that appear when you are running on the *Windows* platform. On *Windows*, you invoke Python 3 using the `py -3` command. If you are on *Linux* or *Mac OS X*, replace the *Windows* command with `sudo python3`. To install **pytest** using `pip` on *Windows*, issue this command from the command prompt while running as administrator (search for `cmd.exe`, then right-click on it, and choose *Run as Administrator* from the pop-up menu):

```
py -3 -m pip install pytest
```

Start in Administrator mode...

...then issue the "pip" command to install "pytest"...

...then check whether it installed successfully.



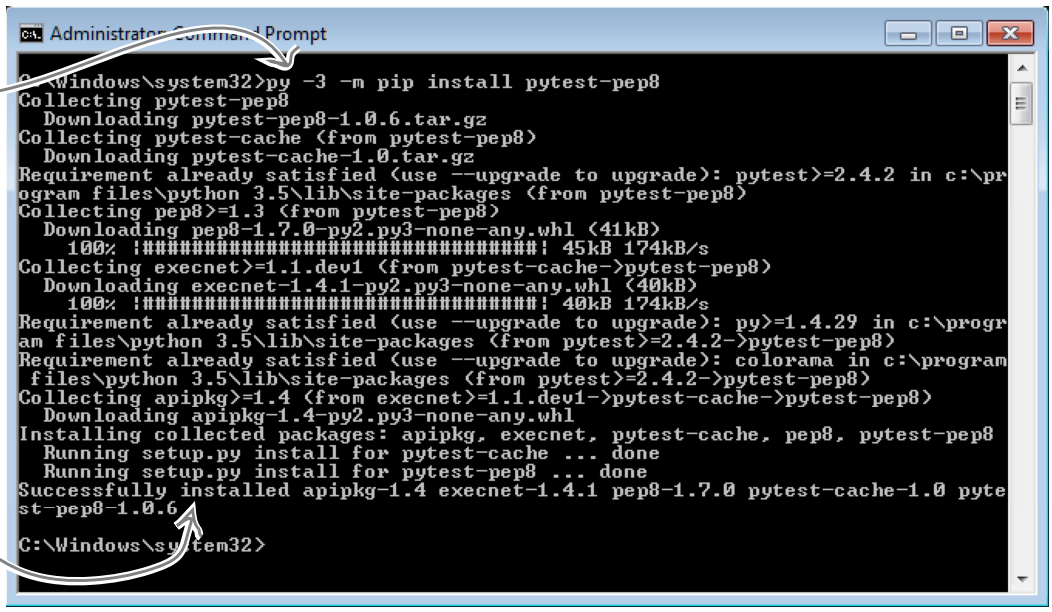
If you examine the messages produced by `pip`, you'll notice that two of **pytest**'s dependencies were also installed (**colorama** and **py**). The same thing happens when you use `pip` to install the **pep8** plug-in: it also installs a host of dependencies. Here's the command to install the plug-in:

```
py -3 -m pip install pytest-pep8
```

Remember: if you aren't running Windows, replace "py -3" with "sudo python3".

While still in Administrator mode, issue this command, which installs the "pep8" plug-in.

This command succeeded too, and also installed the required dependencies.



How PEP 8-Compliant Is Our Code?

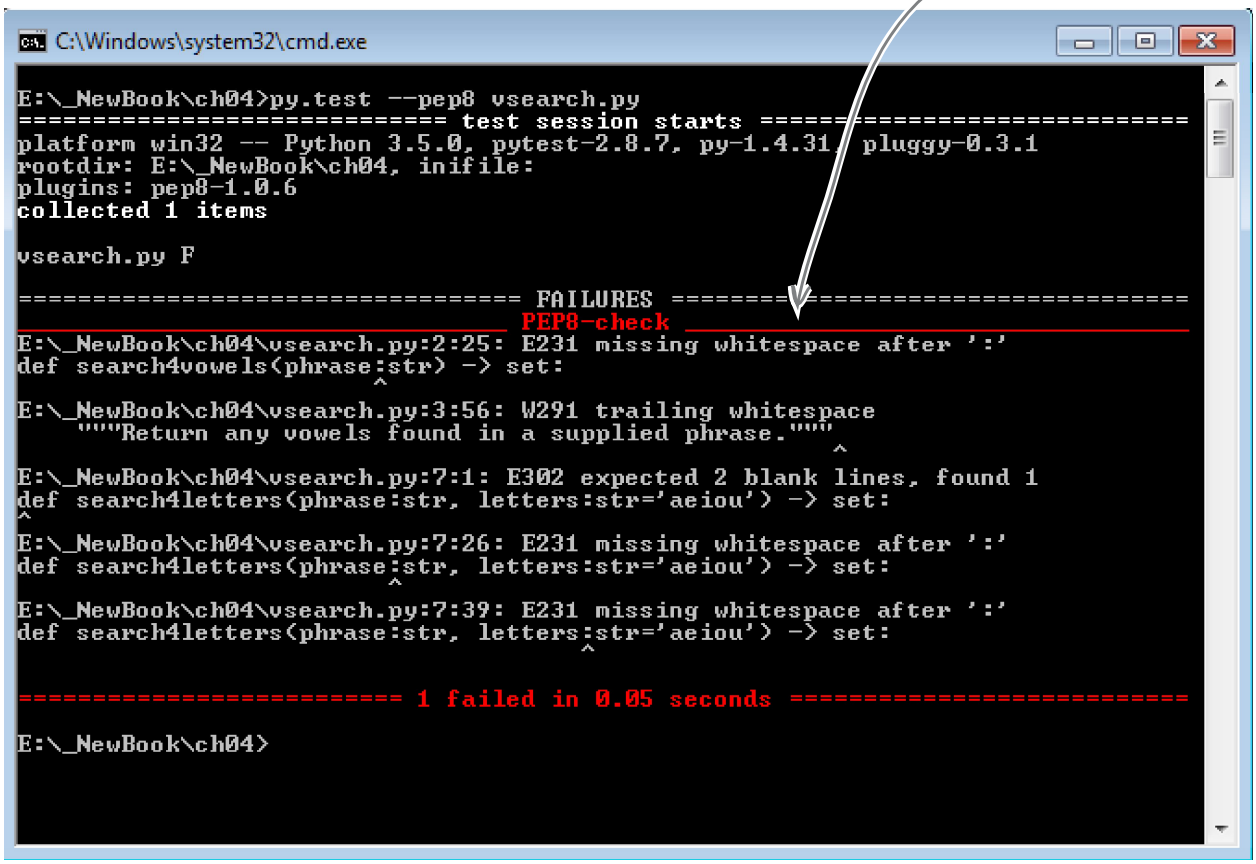
With **pytest** and **pep8** installed, you're now ready to test your code for PEP 8 compliance. Regardless of the operating system you're using, you'll issue the same command (as only the installation instructions differ on each platform).

The **pytest** installation process has installed a new program on your computer called `py.test`. Let's run this program now to check our `vsearch.py` code for PEP 8 compliance. Make sure you are in the same folder as the one that contains the `vsearch.py` file, then issue this command:

```
py.test  --pep8  vsearch.py
```

Here's the output produced when we did this on our *Windows* computer:

Uh, oh. The red output can't be good, can it?



Whoops! It looks like we have **failures**, which means this code is not as compliant with the PEP 8 guidelines as it could be.

Take a moment to read the messages shown here (or on your screen, if you are following along). All of the "failures" appear to refer—in some way—to *whitespace* (for instance, spaces, tabs, newlines, and the like). Let's take a look at each of them in a little more detail.

Understanding the Failure Messages

Together, **pytest** and the **pep8** plug-in have highlighted *five* issues with our `vsearch.py` code.

The first issue has to do with the fact that we haven't inserted a space after the `:` character when annotating our function's arguments, and we've done this in three places. Look at the first message, noting **pytest**'s use of the *caret* character (^) to indicate exactly where the problem is:

```
...:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                         ^
```

Here's what's wrong.

Here's where it's wrong.

If you look at the two issues at the bottom of **pytest**'s output, you'll see that we've repeated this mistake in three locations: once on line 2, and twice on line 7. There's an easy fix: *add a single space character after the colon*.

The next issue may not seem like a big deal, but is raised as a failure because the line of code in question (line 3) does break a PEP 8 guideline that says not to include extra spaces at the end of lines:

What's wrong

```
...:3:56: W291 trailing whitespace
"""Return any vowels found in a supplied phrase."""
                                                  ^
```

Where it's wrong

Dealing with this issue on line 3 is another easy fix: *remove all trailing whitespace*.

The last issue (at the start of line 7) is this:

```
...7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
```

This issue presents at the start of line 7.

Here's what's wrong.

There is a PEP 8 guideline that offers this advice for creating functions in a module: *Surround top-level function and class definitions with two blank lines*. In our code, the `search4vowels` and `search4letters` functions are both at the "top level" of the `vsearch.py` file, and are separated from each other by a single blank line. To be PEP 8–compliant, there should be *two* blank lines here.

Again, it's an easy fix: *insert an extra blank line between the two functions*. Let's apply these fixes now, then retest our amended code.

BTW: Check out http://pep8.org/ for a beautifully rendered version of Python's style guidelines.

Confirming PEP 8 Compliance

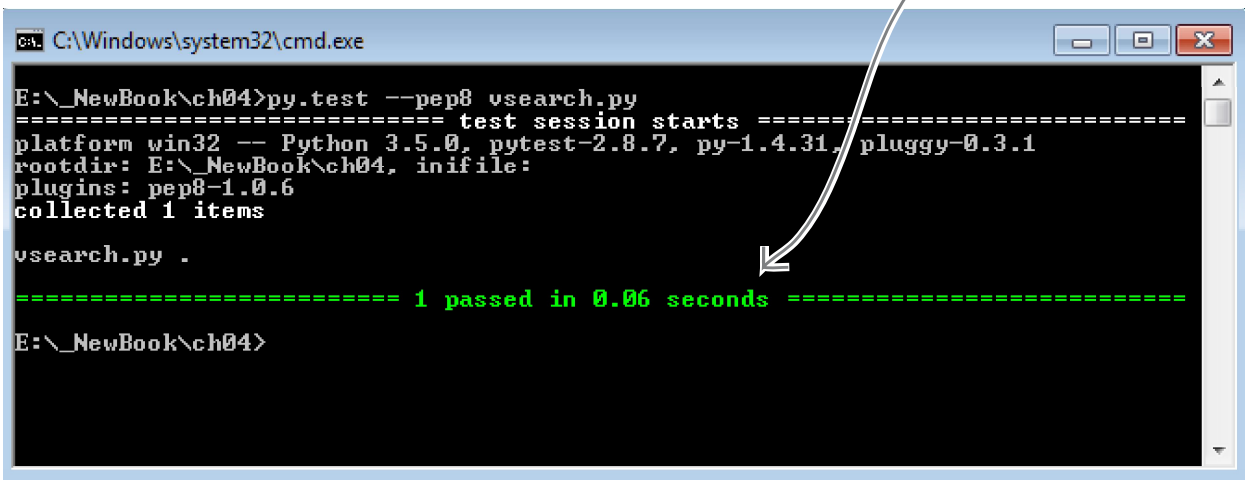
With the amendments made to the Python code in `vsearch.py`, the file's contents now look like this:

```
def search4vowels(phrase: str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

The PEP 8–compliant version of "vsearch.py".

When this version of the code is run through **pytest**'s **pep8** plug-in, the output confirms we no longer have any issues with PEP 8 compliance. Here's what we saw on our computer (again, running on *Windows*):

Green is good—this code has no PEP 8 issues. ☺



Conformance to PEP 8 is a good thing

If you're looking at all of this wondering what all the fuss is about (especially over a little bit of whitespace), think carefully about why you'd want to comply to PEP 8. The PEP 8 documentation states that *readability counts*, and that code is *read much more often than it is written*. If your code conforms to a standard coding style, it follows that reading it is easier, as it "looks like" everything else the programmer has seen. Consistency is a very good thing.

From this point forward (and as much as is practical), all of the code in this book will conform to the PEP 8 guidelines. You should try to ensure your code does too.

This is the end of the pytest detour. See you in Chapter 5.

Chapter 4's Code

This is the code from the "vsearch.py" module, which contains our two functions: "search4vowels" and "search4letters".

```
def search4vowels(phrase: str) -> set:
    """Returns the set of vowels found in 'phrase'."""
    return set('aeiou').intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Returns the set of 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

This is the "setup.py" file, which allowed us to turn our module into an installable distribution.

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='The Head First Python Search Tools',
    author='HF Python 2e',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

And this is the "mystery.py" module, which had Tom and Sarah upset at each other. Thankfully, now that the mystery is solved, they are back to being programming buddies once more. ☺

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)


def change(arg: list):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

5 building a webapp

Getting Real

At this stage, you know enough Python to be dangerous.

With this book's first four chapters behind you, you're now in a position to productively use Python within any number of application areas (even though there's still lots of Python to learn). Rather than explore the long list of what these application areas are, in this and subsequent chapters, we're going to structure our learning around the development of a web-hosted application, which is an area where Python is especially strong. Along the way, you'll learn a bit more about Python. Before we get going, however, let's have a quick recap of the Python you already know.

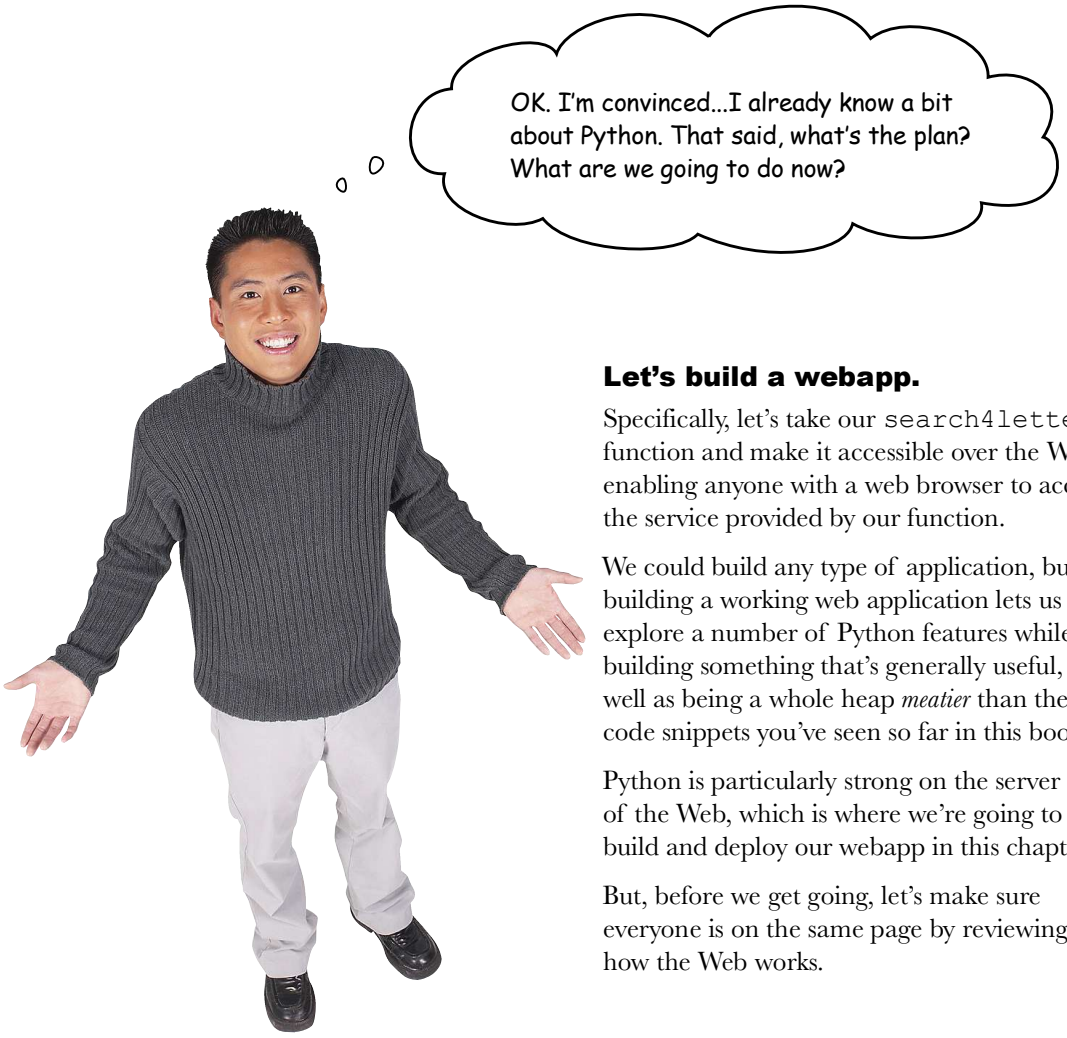
Python: What You Already Know

Now that you've got four chapters under your belt, let's pause for a moment and review the Python material presented so far.

BULLET POINTS

- IDLE, Python's built-in IDE, is used to experiment with and execute Python code, either as single-statement snippets or as larger multistatement programs written within IDLE's text editor. As well as using IDLE, you ran a file of Python code directly from your operating system's command line, using the `py -3` command (on Windows) or `python3` (on everything else).
- You've learned how Python supports single-value data items, such as integers and strings, as well as the booleans `True` and `False`.
- You've explored use cases for the four built-in data structures: lists, dictionaries, sets, and tuples. You know that you can create complex data structures by combining these four built-ins in any number of ways.
- You've used a collection of Python statements, including `if`, `elif`, `else`, `return`, `for`, `from`, and `import`.
- You know that Python provides a rich standard library, and you've seen the following modules in action: `datetime`, `random`, `sys`, `os`, `time`, `html`, `pprint`, `setuptools`, and `pip`.
- As well as the standard library, Python comes with a handy collection of built-in functions, known as the BIFs. Here are some of the BIFs you've worked with: `print`, `dir`, `help`, `range`, `list`, `len`, `input`, `sorted`, `dict`, `set`, `tuple`, and `type`.
- Python supports all the usual operators, and then some. Those you've already seen include: `in`, `not in`, `+`, `-`, = (assignment), == (equality), `+=`, and `*`.
- As well as supporting the square bracket notation for working with items in a sequence (i.e., `[]`) , Python extends the notation to support **slices**, which allow you to specify **start**, **stop**, and **step** values.
- You've learned how to create your own custom functions in Python, using the `def` statement. Python functions can optionally accept any number of arguments as well as return a value.
- Although it's possible to enclose strings in either single or double quotes, the Python conventions (documented in **PEP 8**) suggest picking one style and sticking to it. For this book, we've decided to enclose all of our strings within single quotes, unless the string we're quoting itself contains a single quote character, in which case we'll use double quotes (as a one-off, special case).
- Triple-quoted strings are also supported, and you've seen how they are used to add docstrings to your custom functions.
- You learned that you can group related functions into modules. Modules form the basis of the code reuse mechanism in Python, and you've seen how the `pip` module (included in the standard library) lets you consistently manage your module installations.
- Speaking of things working in a consistent manner, you learned that in Python **everything is an object**, which ensures—as much as possible—that everything works just as you expect it to. This concept really pays off when you start to define your own custom objects using classes, which we'll show you how to do in a later chapter.

Let's Build Something



Let's build a webapp.

Specifically, let's take our `search4letters` function and make it accessible over the Web, enabling anyone with a web browser to access the service provided by our function.

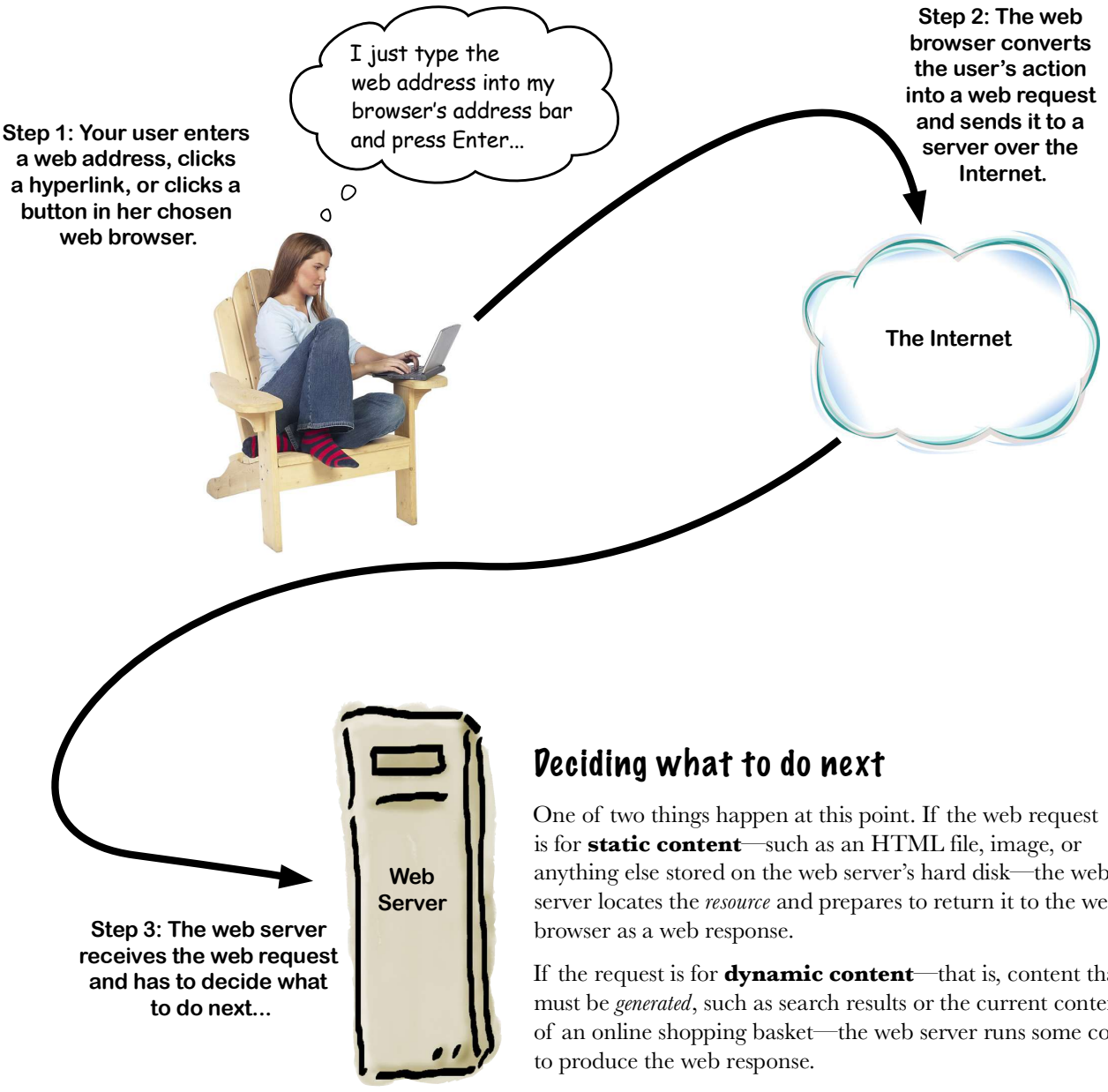
We could build any type of application, but building a working web application lets us explore a number of Python features while building something that's generally useful, as well as being a whole heap *meatier* than the code snippets you've seen so far in this book.

Python is particularly strong on the server side of the Web, which is where we're going to build and deploy our webapp in this chapter.

But, before we get going, let's make sure everyone is on the same page by reviewing how the Web works.

Webapps Up Close

No matter what you do on the Web, it's all about *requests* and *responses*. A **web request** is sent from a web browser to a web server as the result of some user interaction. On the web server, a **web response** (or *reply*) is formulated and returned to the web browser. The entire process can be summarized in five steps, as follows:



Deciding what to do next

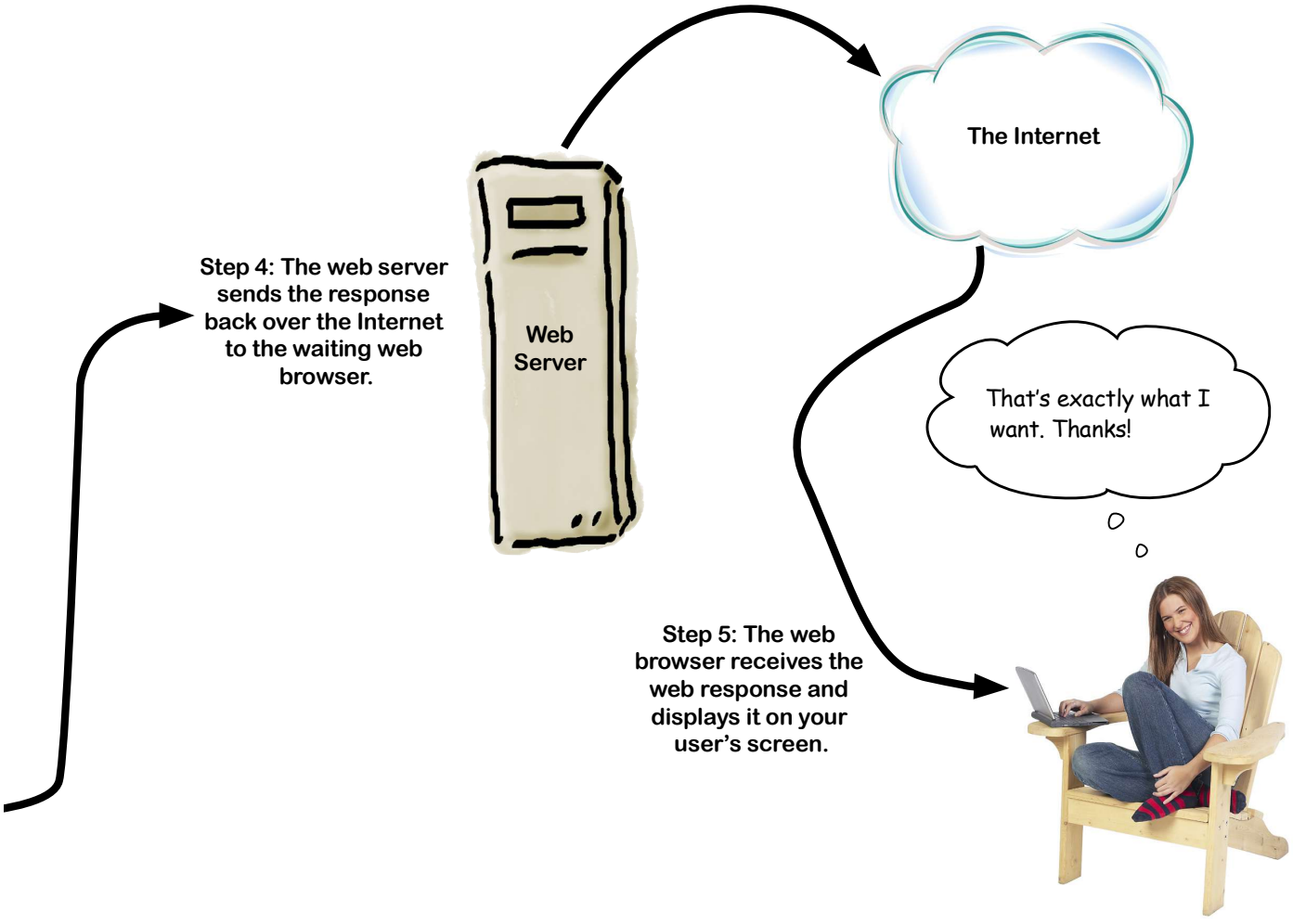
One of two things happen at this point. If the web request is for **static content**—such as an HTML file, image, or anything else stored on the web server's hard disk—the web server locates the *resource* and prepares to return it to the web browser as a web response.

If the request is for **dynamic content**—that is, content that must be *generated*, such as search results or the current contents of an online shopping basket—the web server runs some code to produce the web response.

The (potentially) many substeps of Step 3

In practice, Step 3 can involve multiple substeps, depending on what the web server has to do to produce the response. Obviously, if all the server has to do is locate static content and return it to the browser, the substeps aren't too taxing, as it's just a matter of reading from the web server's disk drive.

However, when dynamic content must be generated, the substeps involve the web server running code and then capturing the output from the program as a web response, before sending the response back to the waiting web browser.



What Do We Want Our Webapp to Do?

As tempting as it always is to *just start coding*, let's first think about how our webapp is going to work.

Users interact with our webapp using their favorite web browser. All they have to do is enter the URL for the webapp into their browser's address bar to access its services. A web page then appears in the browser asking the user to provide arguments to the `search4letters` function. Once these are entered, the user clicks on a button to see their results.

Recall the `def` line for our most recent version of `search4letters`, which shows the function expecting at least one—but no more than two—arguments: a `phrase` to search, together with the `letters` to search for. Remember, the `letters` argument is optional (defaulting to `aeiou`):

The "def" line for the "search4letters" function, which takes one, but no more than two, arguments

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Let's grab a paper napkin and sketch out how we want our web page to appear. Here's what we came up with: