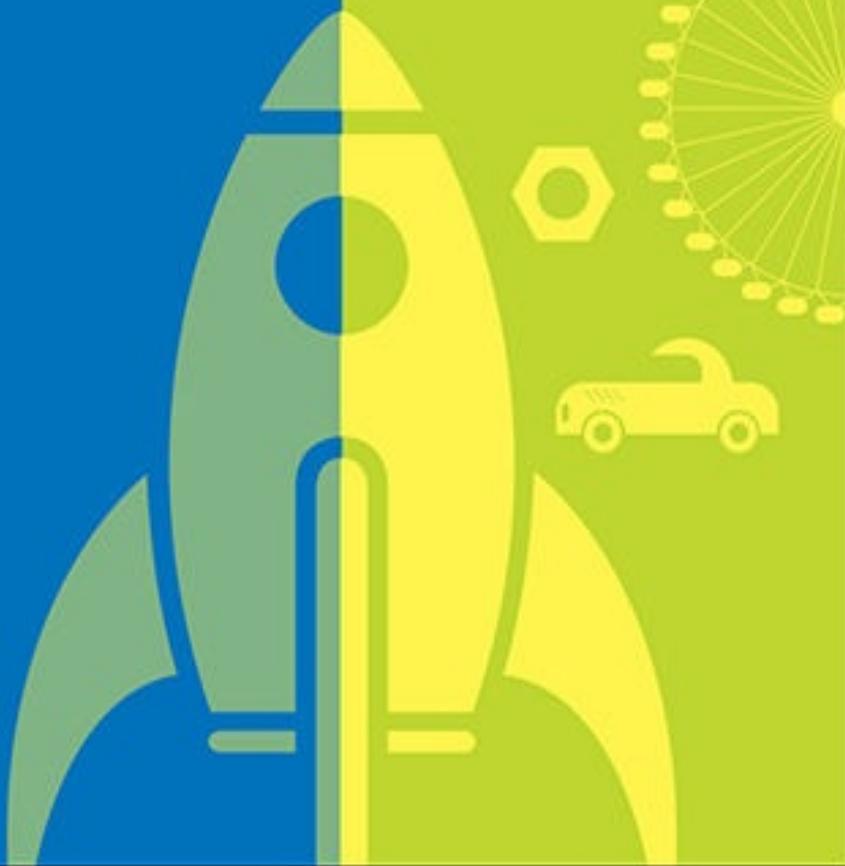




Microsoft

Begin to Code with Python

Rob Miles



Begin to Code with Python

Rob Miles



BEGIN TO CODE WITH PYTHON

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

Copyright © 2018 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-1-5093-0452-3

ISBN-10: 1-5093-0452-5

Library of Congress Control Number: 2017958202

1 17

TRADEMARKS

Microsoft and the trademarks listed at <http://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners. "Python" and the Python logos are trademarks or registered trademarks of the Python Software Foundation, used by Pearson Education with permission from the Foundation.

WARNING AND DISCLAIMER

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

SPECIAL SALES

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor-in-Chief

Greg Wiegand **Senior Acquisitions Editor**
Laura Norman **Development Editor**
Rick Kughen **Managing Editor**
Sandra Schroeder **Senior Project Editor**
Tracey Croom **Copy Editor**
Dan Foster **Indexer**
Valerie Haynes Perry **Proofreader**
Becky Winter **Technical Editor**
John Ray **Editorial Assistant**
Cindy J Teeters **Cover Designer**
Twist Creative, Seattle **Compositor**
Danielle Foster

To Mary

About the author



Rob Miles has spent more than 30 years teaching programming at the University of Hull in the United Kingdom. He's a Microsoft MVP, with a passion for programming, and creating new things. If he had any spare time, he'd spend it writing even more code. He loves making programs and then running them to see what happens. He reckons that programming is the most creative thing you can learn how to do.

He believes that programmers really can claim to be building the future.

He claims to know a lot of really good jokes, but nobody has ever heard him tell one. If you want an insight into the Wacky World™ of Rob Miles, you can read his blog at www.robmiles.com and follow him on Twitter via @RobMiles.

rob@robmiles.com

Contents at a glance

Part 1: Programming fundamentals

- Chapter 1 Starting with Python
- Chapter 2 Python and Programming
- Chapter 3 Python program structure
- Chapter 4 Working with variables
- Chapter 5 Making decisions in programs
- Chapter 6 Repeating actions with loops
- Chapter 7 Using functions to simplify programs
- Chapter 8 Storing collections of data

Part 2: Advanced programming

- Chapter 9 Use classes to store data
- Chapter 10 Use classes to create active objects
- Chapter 11 Object-based solution design
- Chapter 12 Python applications

Part 3: Useful Python (Digital-only)



The chapter PDF files for this Part are available at

<https://aka.ms/BeginCodePython/downloads>.

Chapter 13 Python and Graphical User Interfaces

Chapter 14 Python programs as network clients

Chapter 15 Python programs as network servers

Chapter 16 Create games with Pygame

Contents

Introduction

Part 1: Programming fundamentals

1 Starting with Python

What is Python?

Python origins

Python versions

Build a place to work with Python

Get the tools

Python for Windows PC

Start Python

What you have learned

2 Python and Programming

What makes a programmer

Programming and party planning



Give us feedback

Tell us what you think of this book and help Microsoft improve our products for you.

Thank you!

<http://aka.ms/tellpress>

Programming and problems

Programmers and people

Computers as data processors

Machines and computers and us

Programs as data processors

Python as a data processor

Data and information

Work with Python functions

The `ord` function

The `chr` function

Investigate data storage using `bin`

What you have learned

3 Python program structure

Write your first Python program

Run Python programs using IDLE

Get program output using the `print` function

Use Python libraries

The `random` library

The `time` library

Python comments

Code samples and comments

Run Python from the desktop

Delay the end of the program

Adding some snaps

Adding the Pygame library

Snaps functions

What you have learned

4 Working with variables

Variables in Python

Python names

Working with text

Marking the start and end of strings

Escape characters in text

Read in text using the `input` function

Working with numbers

Convert strings into integer values

Whole numbers and real numbers

Real numbers and floating-point numbers

Convert strings into floating-point values

Perform calculations

Convert between `float` and `int`

Weather snaps

What you have learned

5 Making decisions in programs

Boolean data

Create Boolean variables

Boolean expressions

Comparing values

Boolean operations

The `if` construction

Nesting `if` conditions

Working with logic

Use decisions to make an application

Design the user interface

Implement a user interface

Testing user input
Complete the program

Input snaps
What you have learned

6 Repeating actions with loops

The `while` construction

Repeat a sequence of statements using `while`
Handling invalid user entry
Detect invalid number entry using exceptions
Exceptions and number reading
Handling multiple exceptions
Break out of loops
Return to the top of a loop with `continue`
Count a repeating loop

The `for` loop construction

Make a digital clock using snaps
What you have learned

7 Using functions to simplify programs

What makes a function?

Give information to functions using parameters
Return values from function calls

Build reusable functions

Create a text input function
Add help information to functions
Create a number input function
Convert our functions into a Python module
Use the IDLE debugger

What you have learned

8 Storing collections of data

Lists and tracking sales

Limitations of individual variables

Lists in Python

Read in a list

Display a list using a `for` loop

Refactor programs into functions

Create placeholder functions

Create a user menu

Sort using bubble sort

Initialize a list with test data

Sort a list from high to low

Sort a list from low to high

Find the highest and lowest sales values

Evaluate total and average sales

Complete the program

Store data in a file

Write into a file

Write the sales figures

Read from a file

Read the sales figures

Deal with file errors

Store tables of data

Use loops to work with tables

Use lists as lookup tables

Tuples

What you have learned

Part 2: Advanced programming

9 Use classes to store data

Make a tiny contacts app

 Make a prototype

 Store contact details in separate lists

 Use a class to store contact details

 Use the `Contact` class in the Tiny Contacts program

 Edit contacts

 Save contacts in a file using pickle

 Load contacts from a file using pickle

 Add `save` and `load` to Tiny Contacts

 Set up class instances

Dictionaries

 Create a dictionary

 Dictionary management

 Return a dictionary from a function

 Use a dictionary to store contacts

What you have learned

10 Use classes to create active objects

Create a Time Tracker

 Add a data attribute to a class

 Create a cohesive object

 Create method attributes for a class

 Add validation to methods

 Protect a data attribute against damage

 Protected methods

Create class properties

Evolve class design

 Manage class versions

- The `__str__` method in a class
- Python string formatting
- Session tracking in Time Tracker
- The Python `map` function
- The Python `join` method
- Make music with Snaps
- What you have learned

11 Object-based solution design

- Fashion Shop application
 - Application data design
 - Object-oriented design
 - Creating superclasses and subclasses
 - Data design recap
 - Implement application behaviors
 - Objects as components
- Create a `FashionShop` component
 - Create a user interface component
- Design with classes
- Python sets
 - Sets and tags
 - Sets versus class hierarchies
- What you have learned

12 Python applications

- Advanced functions
 - References to functions
 - Use lambda expressions
 - Iterator functions and the `yield` statement

Functions with an arbitrary number of arguments
Modules and packages
 Python modules
 Add a `readme` function to BTCInput
 Run a module as a program
 Detect whether a module is executed as a program
 Create a Python package
 Import modules from packages
Program testing
 The Python `assert` statement
 The Python `unittest` module
 Create tests
View program documentation
What you have learned

Part 3: Useful Python (Digital-only)



The chapter PDF files for this Part are available at
<https://aka.ms/BeginCodePython/downloads>.

13 Python and Graphical User Interfaces

Visual Studio Code
 Install Visual Studio Code
 Install the Python Extension in Visual Studio Code
 Create a project folder
 Create a program file

- Debug a program
- Other Python editors
- Create a Graphical User Interface with Tkinter
 - Create a graphical application
 - Lay out a grid
 - Create an event handler function
 - Create a mainloop
 - Handle errors in a graphical user interface
 - Display a message box
 - Draw on a Canvas
 - Tkinter events
 - Create a drawing program
 - Enter multi-line text
 - Group display elements in frames
 - Create an editable `StockItem` using a GUI
 - Create a `Listbox` selector
 - An application with a graphical user interface
- What you have learned

14 Python programs as network clients

- Computer networking
- Consume the web from Python
 - Read a webpage
 - Use web-based data
- What you have learned

15 Python programs as network servers

- Create a web server in Python
 - A tiny socket-based server
 - Python web server

Serve webpages from files

Get information from web users

Host Python applications on the web

What you have learned

16 Create games with Pygame

Getting started with pygame

Draw images with pygame

Image file types

Load an image into a game

Make an image move

Get user input from pygame

Create game sprites

Add a player sprite

Control the player sprite

Add a Cracker sprite

Add lots of sprite instances

Catch the crackers

Add a killer tomato

Complete the game

Add a start screen

End the game

Score the game

What you have learned

Index

Introduction

Programming is the most creative thing you can learn how to do. Why? If you learn to paint, you can create pictures. If you learn to play the violin, you can make music. But if you learn to program, you can create entirely new experiences (and you can make pictures and music too, if you wish). Once you've started on the programming path, there's no limit to where you can go. There are always new devices, technologies, and marketplaces where you can use your programming skills.

Think of this book as your first step on a journey to programming enlightenment. The best journeys are undertaken with a destination in mind, and the destination of this journey is "usefulness." By the end of this book, you will have the skills and knowledge to write useful programs.

However, before we begin, a small word of warning. Just as a guide would want to tell you about the lions, tigers, and crocodiles that you might encounter on a safari, I must tell you that our journey might not be all smooth going. Programmers must learn to think slightly differently about problem solving, because a computer just doesn't work the same way humans do. Humans can do complex things rather slowly. Computers can do simple things very quickly. It is the programmer's job to harness the simple abilities of the machine to solve complicated problems. This is what you'll learn to do.

The key to success as a programmer is much the same as for many other endeavors. To become a world-renowned violin player, you will have to practice a lot. The same is true for programming. You must spend a lot of time working on your programs to acquire code-writing skills. However, the good news is that just as a violin player really enjoys making the instrument sing, making a computer do exactly what you want turns out to be a very rewarding experience. It gets even more enjoyable when you see other people using programs that you've written and finding them useful and fun to use.

How this book fits together

I've organized this book in three parts. Each part builds on the previous one with the aim of turning you into a successful programmer. We start off considering the low-level programming instructions that programs use to tell the computer what to do, and we finish by looking at professional software construction.

Part 1: Programming fundamentals

The first part gets you started. It points you to where you'll install and use the programming tools that you'll need to begin coding, and it introduces you to the fundamental elements of the Python programming language. You'll come to grips with writing your first programs and learn all the fundamental code constructions that underpin all programs. You'll also find out where Python fits in the great scheme of programming languages, and what this means for you as a programmer.

Part 2: Advanced programming

[Part 2](#) describes the features of the Python programming language used to create and structure more complex applications. It shows you how to break large programs into smaller elements and how you can create custom data types that reflect the specific problem being solved. You'll also discover how to design, test, and document your Python applications.

Part 3: Useful Python

Once you've learned how to make your own programs, the next step is to learn how to use code written by other people. An important advantage of Python is the wealth of software libraries available for users of the language. In [Part 3](#), you'll explore a number of these libraries and find out how you can use them to create applications with graphical user interfaces, how Python programs can act as clients and servers in network applications, and, finally, create engaging games.

The third part of the book is provided as a downloadable document that you can have open on your desktop as you experiment with the demonstration programs and create applications of your own. The chapter PDF files for this Part are available at <https://aka.ms/BeginCodePython/downloads>

How you will learn

In each chapter, I will tell you a bit more about programming. I'll show you how to do something, and then I'll invite you to make something of your own by using what you've learned. You'll never be more than a page or so away from doing something or making something unique and personal. After that, it's up to you to make something amazing!

You can read the book straight through if you like, but you'll learn much more if you slow down and work with the practical parts along the way. Like learning to ride a bicycle, you'll learn by *doing*. You must put in the time and practice to learn how to do it. But this book will give you the knowledge and confidence to try your hand at programming, and it will also be around to help you if your programming doesn't turn out as you expected. Here are some elements in the book that will help you learn by doing:



MAKE SOMETHING HAPPEN

Yes, the best way to learn things is by doing, so you'll find "Make Something Happen" elements throughout the text. These elements offer ways for you to practice your programming skills. Each starts with an example and then introduces some steps you can try on your own. Everything you create will run on Windows, macOS, or Linux.



CODE ANALYSIS

A great way to learn how to program is by looking at code written by others and working out what it does (and sometimes why it doesn't do what it should). This book contains over 150 sample programs for you to examine. In this book's "Code Analysis" challenges, you'll use your deductive skills to figure out the behavior of a program, fix bugs, and suggest improvements.



WHAT COULD GO WRONG

If you don't already know that programs can fail, you'll learn this hard lesson soon after you begin writing your first program. To help you deal with this in advance, I've included "What Could Go Wrong?" elements, which anticipate problems you might have and provide solutions to those problems. For example, when I introduce something new, I'll sometimes spend some time considering how it can fail and what you need to worry about when you use the new feature.



PROGRAMMER'S POINT

I've spent a lot of time teaching programming. But I've also written many programs and sold a few to paying customers. I've learned some things the hard way that I really wish I'd known at the start. The aim of "Programmer's Points" is to give you this information up front so that you can start taking a professional view of software development as you learn how to do it.

"Programmer's Points" cover a wide range of issues, from programming to people to philosophy. I strongly advise you to read and absorb these points carefully—they can save you a lot of time in the future!

Software and hardware

You'll need a computer and some software to work with the programs in this book. I'm afraid I can't provide you with a computer, but in the first chapter you'll find out where you can get the Python tools and an application called Visual Studio Code that you'll learn to use when we start creating larger applications.

Using a PC or laptop

You can use Windows, macOS, or Linux to create and run the Python programs in the text. Your PC doesn't have to be particularly powerful, but these are the minimum specifications I'd recommend:

- A 1 GHz or faster processor, preferably an Intel i5 or better.
- At least 4 gigabytes (GB) of memory (RAM), but preferably 8 GB or more.
- 256 GB hard drive space. (The full Python and Visual Studio Code installations take about 1 GB of hard drive space.)

There are no specific requirements for the graphics display, although a higher-resolution screen will enable you to see more when writing your programs.

Using a mobile device

You can write and run Python programs on a mobile phone or tablet. If you have an Apple device running iOS, I recommend the Pythonista app. If you’re using an Android device, look at the Pyonic Python 3 interpreter.

Using a Raspberry Pi

If you want to get started in the most inexpensive way possible, you can use a Raspberry Pi running the Raspbian operating system, which has Python 3.6 and the IDLE development environment built in.

Downloads

In every chapter in this book, I’ll demonstrate and explain programs that teach you how to begin to program—and that you can then use to create programs of your own. You can download this book’s sample code from the following page: <https://aka.ms/BeginCodePython/downloads>

Follow the instructions you’ll find in [Chapter 1](#) to install the sample programs and code.

Acknowledgments

I would like to thank Laura Norman for giving me a chance to write this book, Dan Foster and Rick Kughen for putting up with my prose and knocking it into shape, John Ray for astute and constructive technical insights, and Tracey Croom and Becky Winter for making it all look so nice. I’d also like to say

thanks to Rob Nance for all the lovely artwork.

Finally, I'd like to say thanks to my wife, Mary, for her support and endless cups of tea. And biscuits.

Errata, updates, and book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

<https://aka.ms/BegintoCodePython/errata>

If you discover an error not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at

mspinput@microsoft.com

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to <http://support.microsoft.com>

Obtaining MTA qualification

The Microsoft Certified Professional program lets you obtain recognition for your skills. Passing the exam 98-381, “Introduction to Programming Using Python” gives you a Microsoft Technology Associate (MTA) level qualification in Python programming. You can find out more about this examination at <https://www.microsoft.com/en-us/learning/exam-98-381.aspx>

To help you get the best out of this text, if you're thinking of using it as part of a program of study to prepare for this exam, I've put together a mapping of exam topics to book elements that you may find helpful. (Note that these mappings are based on the exam specification as of October 2017.) I've also created an appendix, which puts some of the elements described in the book into the context of the exam, and you can find this in the same place as the sample code

downloads.

<https://aka.ms/BeginCodePython/downloads>

Qualification structure

The qualification is broken down into a number of topic areas, each of which comprises a percentage of the total qualification. We can go through each topic area and identify the elements of this book that apply. The tables below map skill items to sections in chapters of this book.

Perform operations using data types and operators (20-25%)

To prepare for this topic, you should pay special attention to [Chapter 4](#), which describes the essentials of data processing in Python programs as well as how text and numeric data is stored and manipulated. [Chapter 5](#) introduces the Boolean variable type and explains how logical values are manipulated. [Chapter 8](#) describes how to store collections of data, and [Chapter 9](#) explains the creation and storage of data structures. [Chapter 11](#) gives details of sets in Python programs.

| SKILL | BOOK ELEMENT |
|--|---|
| Evaluate an expression to identify the data type Python will assign to each variable. Identify str, int, float, and bool data types. | Chapter 4: Variables in Python |
| | Chapter 4: Working with text |
| | Chapter 4: Working with numbers |
| | Chapter 5: |

| | |
|--|---|
| | <p>Boolean data</p> <p>Chapter 5: Boolean expressions</p> |
| | <p>Chapter 4: Convert strings into floating-point values</p> <p>Chapter 4: Convert between float and int</p> |
| Perform data and data type operations. Convert from one data type to another type; construct data structures; perform indexing and slicing operations. | <p>Chapter 4: Real numbers and floating-point numbers</p> <p>Chapter 8: Lists and tracking sales</p> <p>Chapter 9: Use a class to store contact details</p> |
| Determine the sequence of execution based on operator precedence = Assignment; Comparison; Logical; Arithmetic; Identity (is); Containment (in) | <p>Chapter 4: Performing calculations</p> <p>Chapter 9: Contact objects and references</p> <p>Chapter 11: Sets and tags</p> |

Select the appropriate operator to achieve the intended result: Assignment; Comparison; Logical; Arithmetic; Identity (is); Containment (in).

[Chapter 4:
Performing
calculations](#)

[Chapter 5:
Boolean
expressions](#)

Control flow with decisions and loops (25-30%)

To prepare for this topic, you should pay special attention to [Chapter 5](#), which describes the `if` construction, and [Chapter 6](#), which moves on to describe the `while` and `for` loop constructions that are used to implement looping in Python programs.

| SKILL | BOOK ELEMENT |
|--|---|
| Construct and analyze code segments that use branching statements. | Chapter 5: Boolean data Chapter 5: The if construction |
| <code>if; elif; else; nested and compound conditional expressions</code> | Chapter 5: The if construction |
| Construct and analyze code segments that perform iteration. | Chapter 6: The while construction Chapter 6: The for loop construction |
| | Chapter 6: The while |

| | |
|---|--|
| <p>while; for; break; continue; pass; nested loops and loops that include compound conditional expressions.</p> | <p>construction</p> <p>Chapter 6: The for loop construction</p> <p>Chapter 8: Sort using bubble sort</p> |
|---|--|

Perform input and output operations (20-25%)

The use of console input and output functions is demonstrated throughout the book, starting with the very first programs described in [Chapters 3 and 4](#). [Chapter 8](#) introduces the use of file storage in Python programs, and [Chapter 9](#) expands on this to show how data structures can be saved into files by the use of the Python pickle library. [Chapter 10](#) contains details of the string formatting facilities available to Python programs.

| SKILL | BOOK ELEMENT |
|---|---|
| Construct and analyze code segments that perform file input and output operations. Open; close; read; write; append; check existence; delete; with statement | <p>Chapter 8: Store data in a file</p> <p>Chapter 9: Save contacts in a file using pickle</p> |
| Construct and analyze code segments that perform console input and output operations. Read input from console; print formatted text; use of command line arguments. | <p>Chapter 3: Get program output using the print function</p> <p>Chapter 4: Read in text using the input function</p> |

Chapter 10: Python string formatting

Document and structure code (15-20%)

The importance of well-structured and documented code is highlighted throughout the text. [Chapter 3](#) introduces Python comments, and [Chapter 5](#) contains a discussion highlighting the importance of good code layout. [Chapter 7](#) introduces Python functions in the context of improving program structure and describes how to add documentation to functions to make programs self-documenting.

| SKILL | BOOK ELEMENT |
|---|--|
| Document code segments using comments and documentation strings. Use indentation, white space, comments, and documentation strings; generate documentation using pydoc. | Chapter 3: Python comments Chapter 5: Indented text can cause huge problems Chapter 7: Add help information to functions Chapter 12: View program documentation |
| Construct and analyze code segments that include function definitions: call signatures; default values; return; def; pass. | Chapter 7: What makes a function? |

Perform troubleshooting and error handling (5-10%)

[Chapter 3](#) contains coverage of syntax errors in Python code. In [Chapter 4](#), the description of data processing includes descriptions of runtime errors. In [Chapters 6 and 7](#), the causes and effects of logic errors are discussed in the context of an application development. [Chapters 6 and 10](#) contain descriptions of how Python programs can raise and manage exceptions, and [Chapter 12](#) contains a description of the use of unit tests in Python program testing.

| SKILL | BOOK ELEMENT |
|---|--|
| Analyze, detect, and fix code segments that have errors: Syntax errors; logic errors; runtime errors. | Chapter 3: Broken programs Chapter 4: Typing errors and testing Chapter 5: Equality and floating-point values Chapter 6: When good loops go bad Chapter 7: Investigate programs with the debugger Chapter 12: Program testing |
| Analyze and construct code segments that handle exceptions: try; except; else; finally; raise. | Chapter 6: Detect invalid number entry using exceptions Chapter 10: Raise an exception to indicate an error |

Perform operations using modules and tools (1-5%)

Many Python modules are used throughout the text, starting with the random and time modules. The functions from the random library are used in [Chapter 13](#) to create random artwork, and functions from the time library are used in a time-tracking application used in [Chapter 16](#).

| SKILL | BOOK ELEMENT |
|--|--|
| Perform basic operations using built-in modules: math; datetime; io; sys; os; os.path; random. | Chapter 3: The random library Chapter 3: The time library |
| Solve complex computing problems by using built-in modules: math; datetime; random. | Chapter 10: Session tracking in Time Tracker Chapter 16: Making art |

Free e-books from Microsoft Press

From technical overviews to in-depth information on specific topics, the free e-books from Microsoft Press cover a wide range of topics. These e-books are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at: <https://aka.ms/mspressfree>

Check back often to see what's new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback is our most valuable asset. Please tell us what you think of this book at:

<https://aka.ms/tellpress>

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter:

<http://twitter.com/MicrosoftPress>

Part 1 Programming fundamentals

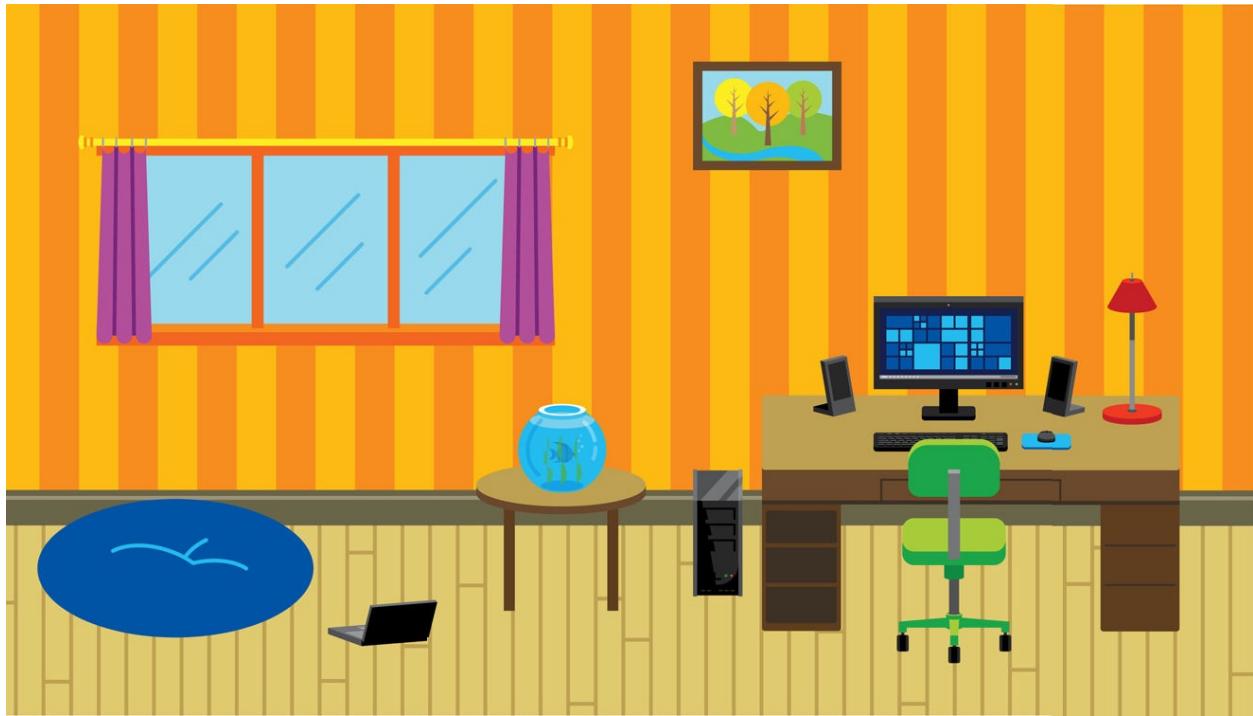


Let's begin our journey toward programming enlightenment. You'll start by considering what a computer actually does and what a programming language is.

Then, you'll move on to installing the programming tools you'll need. Next, you'll take your first steps in using the Python language to tell a computer to do things for you.

The aim of [Part 1](#) is to introduce you to fundamental elements of the Python programming language used by all programs.

1 Starting with Python



What you will learn

We'll start with a discussion about Python, the programming language you're learning. Then you'll discover what kind of computer you need to write programs and how to find and install the tools you'll use to build your code. You'll also have your first of many interactions with the Python Shell and discover that programming-language designers do have a sense of humor.

[What is Python?](#)

[Build a place to work with Python](#)

[Start Python](#)

What you have learned

What is Python?

Before you start learning about Python, it's worth considering just what we are learning about. Python is a *programming language*. In other words, it's a language you use to write programs. A program is a set of instructions that tells a computer how to do something. We can't use a "proper" language like English to do this because "proper English" is just too confusing for a computer to understand. As an example, consider a doctor's instructions:

[Click here to view code image](#)

"Drink your medicine after a hot bath."

Well, we would probably take a hot bath and then drink our medicine. A computer, however, would probably drink the hot bath and then drink its medicine. You can interpret the above instructions either way, because the English language allows you to write ambiguous statements. Programming languages must be designed so that instructions written with them are not open to interpretation; they must tell the computer precisely and unambiguously what to do. This usually means breaking down actions into a sequence of simpler steps:

[Click here to view code image](#)

Step1: Take a hot bath
Step2: Drink your medicine

A programming language forces us to write instructions in this way. Python is one of many programming languages invented to provide humans with a way of telling the computer what to do.

In my programming career, I've learned many different languages over the years, and I confidently expect to need to learn even more in the future. None of them are perfect, and I see each of them as a tool that I would use in a particular

situation, just as I would choose a different tool depending on whether I was making a hole in a brick wall, a pane of glass, or a piece of wood.

Some people get very excited when talking about the “best” programming language. I’m quite happy to discuss what makes the best programming languages, just as I’m happy to tell you all about my favorite type of car, but I don’t see this as something to get worked up about. I love Python for its power and expressiveness. I love the C# programming language for the way it pushes me to produce well-structured solutions. I love the C++ programming language for the way it gives me absolute control of hardware underneath my program. And so on. Python does have things about it that make me want to tear my hair out in frustration, but that’s true of the other languages, too. And all programming languages have aspects about them that I love. And I love using all of them.

Python origins

You might think that programming languages are a bit like space rockets in that they are designed by white-coated scientists with mega-brains who get everything right the first time and always produce perfect solutions. However, this is not the case. Programming languages have been developed over the years for all kinds of reasons, including reasons as simple as “It seemed like a good idea at the time.”

Guido van Rossum, the original developer of Python, had no idea just what he was starting when, in late 1989, he decided to spend a few weeks on a hobby programming project that he could work on while his office at work was closed during Christmas. He named this language “Python,” not because of a liking for snakes, but because he was an enthusiastic fan of the British TV comedy show *Monty Python’s Flying Circus*. However, the language was picked up by programmers the world over who loved its elegance and power. Python is now one of the most popular programming languages in use today.

Python versions

One of the first things you’ll discover about Python is that many versions of the language are available. From a programmer’s point of view, this is not terribly helpful. Programs you write for one version of Python are not guaranteed to work with another version. Python lets you use parts of other people’s programs

in your programs, which is a great way to save time and effort. Of course, this only works properly if all the programs are written using the same Python version. Over the years, the different versions of the language have resolved into two distinct strands of development, which simplifies things somewhat.

Essentially the choice of which version of Python you'll use boils down to a choice of version 2.7 or version 3.*n* (where *n* is a number that keeps increasing).

- Version 2.7 (the old stalwart) is great because a lot of existing software uses version 2.7. If you're looking for a library of Python code to perform a particular task, there's a better chance of it being available in version 2.7.
- Version 3.*n* (the latest one) is great because it eliminates some confusing features of the language that can trip you up.

This book focuses on Python version 3.*n*, but we'll look at differences between the versions when they affect our programs. Also, we'll consider how to use some libraries that are supported only in version 2.7.

The good news is that the fundamentals of program creation are the same for both versions of Python—and indeed for just about all programming languages. Once you've learned how to write Python programs, you'll be able to transfer this skill into many other languages, including C++, C#, Visual Basic, and JavaScript.

Just as you can drive just about any vehicle once you learn the fundamentals of driving, you can transfer your Python programming skills to other programming languages. When driving a new car, you just need to locate the various switches and controls before setting off on your journey. The same holds true with programming.

Build a place to work with Python

If you were a truck driver who spends many hours in the cab hauling goods across the country, you'd want a truck with a comfortable seat, an unobstructed view of the road, and controls that are easy to find and use. It would also help if your truck had enough power to climb hills at a reasonable speed and was easy to handle on twisting mountainside roads.

In the same way, if you expect to spend any amount of time at a keyboard writing programs, you should have a decent place to work. Find somewhere to set up a PC, keyboard, and monitor. Pull up a chair that you don't mind spending quite a few hours sitting in.

You don't need a particularly fancy computer for writing programs, but your PC will need a reasonable amount of memory and processor performance to handle the tools we'll use. I suggest using a device with at least an Intel i5 or equivalent processor, 4 GB of memory, and 256 GB of hard drive space. You can use smaller computers, but they can make the development process somewhat frustrating because they will take a while to update your program after you make changes to it.

The operating system you use is a matter of personal preference. I prefer Windows 10, but if you prefer using macOS or Linux, you can use those operating systems instead. The Python language and the development environment we'll use are available for all three operating systems.

Get the tools

Before you can start sharing or selling your programs, you must download and install the tools that will make this possible. Installation will take a little while, depending on the speed of your network connection. There will be a few occasions when you'll just have to sit and wait while things are fetched from the Internet and installed. Note that it's important to perform the actions in the order I provide. The good news is that you need to perform this installation only once for each computer you want to use.

All the tools we'll use are free to download and install. I find it astonishing and wonderful that such powerful software is available for free for anyone to use. The Python distribution makes it easy for you to get started writing programs, and the Visual Studio Code editor is a great environment for creating large applications.

The tools and how you obtain them are slightly different for the different devices that you might be using. You can download the latest, up-to-date, instructions here:

www.begintocodewithpython.com/tools/

Python for Windows PC

If you have a Windows PC, you can download and install Python from the Python website. You'll download the installer from the Python download site and then run it to install Python on your Windows PC. I used the Microsoft Edge browser. If you use a different browser, you'll see slightly different screens, so my advice would be to perform the following steps using Edge.

Be sure that you are using the official download site (the one given below) and allow the installer to run when asked by Windows.

www.python.org/downloads/

Figure 1-1 shows the download page for Python.



Figure 1-1 Python download page

The web page has determined that I am using a Windows PC and has offered me two versions to download. On the download page, you'll want to download version **3.6.1** of Python, so click this button to start the download.



The browser will ask what you want to do with the file. If you're using

Microsoft Edge, you'll see the dialog box above. Click the **Run** button. When the Python installer has downloaded, it will start to run (**Figure 1-2**).



Figure 1-2 Python installer

The Python installer program will load Python onto your computer and make it available for use. There are several settings that can change the way that Python is installed, but you don't need to change any of these. The only change you should make is to select the Add Python 3.6 to PATH check box at the bottom of the installer dialog. Then click **Install Now**. You might be asked to confirm the changes being made to your system, so click **OK** to accept these changes.

At the end of the installation, you should see the window shown in **Figure 1-3**. Click **Close** to close the window.

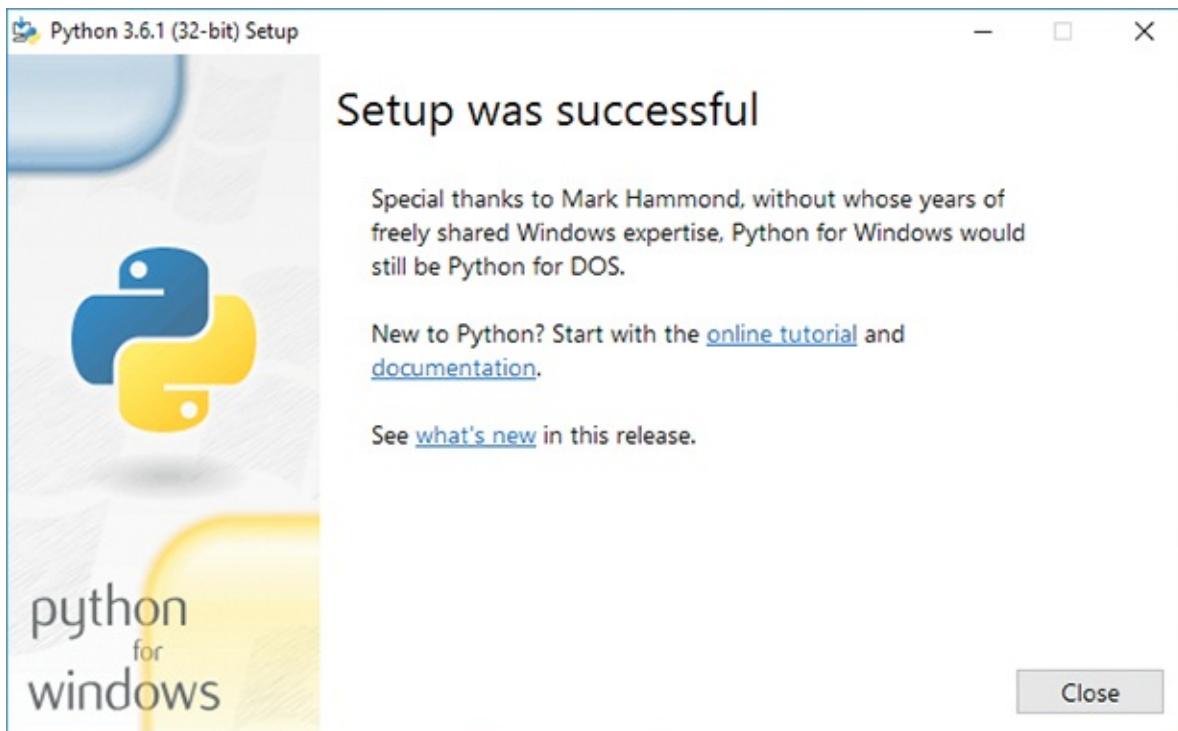


Figure 1-3 Successful setup

You might see the message “Disable path length limit” ([Figure 1-4](#)) at the end of the installation. This refers to the way that Windows manages references to files. If you see this message, click “**Disable path length limit.**” You’ll be asked to confirm the changes to your machine settings.

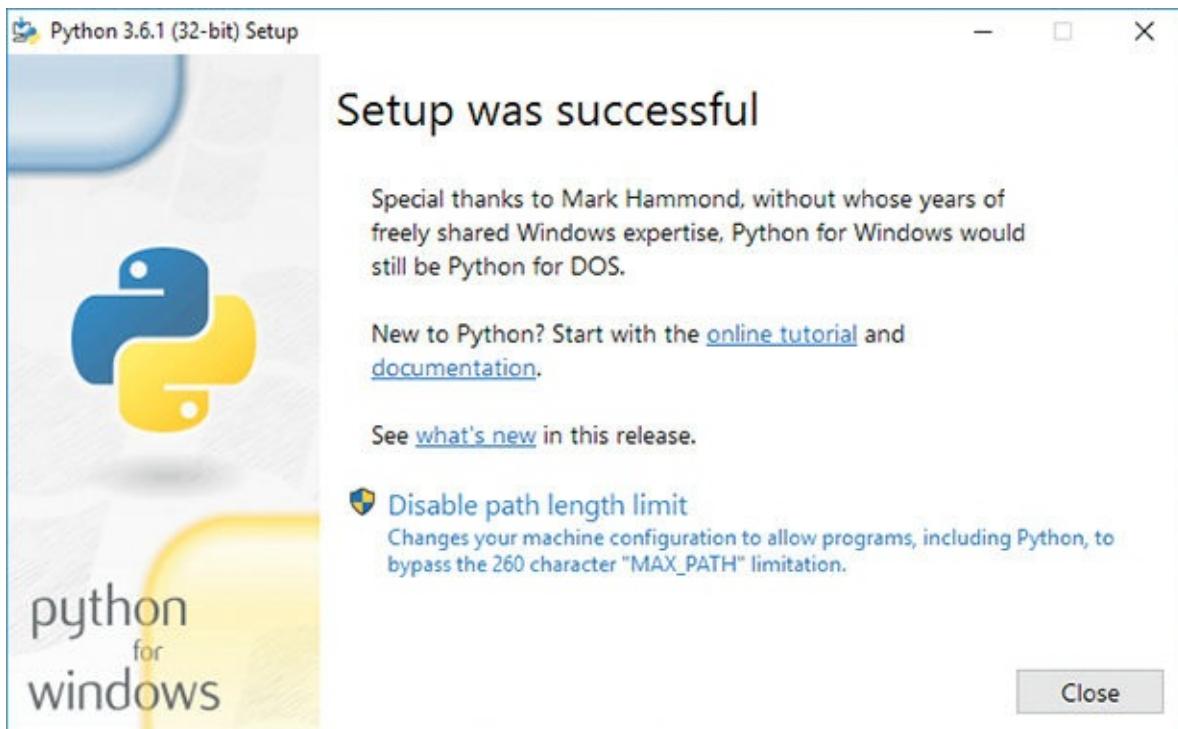


Figure 1-4 Successful installation with path length limit

Once the installation has finished, click **Close** to close the installer program.



WHAT COULD GO WRONG

Python installer problems

The Python download web page can usually work out which operating system you're using and will display buttons that automatically direct you to the correct files for your computer. However; sometimes this might not happen, in which case you won't see any direct download buttons. If this happens to you; just select your operating system from the options displayed on the downloads page and then find the latest release of the language for your computer.

Python is upgraded quite regularly, so you might notice that the version offered is different from the one shown in [Figure 1-1](#) (for example it might have reached version 3.6.3). This is not a problem. As long as the

version number is 3.n you will be able to use the sample programs in this text.

Start Python

You’re about to start up an environment that supports the Python language and interact with it. This is a bit like opening the front door of a new apartment or house or getting into a shiny new car you just bought.

The tool you’ll use is called IDLE, which means “Integrated Development Learning Environment” (and might also be a reference to Eric Idle, one of the members of the Monty Python comedy team). IDLE provides two ways of interacting with Python: the “shell” where you can type Python commands that are executed instantly, and a text editor that allows you to create program code documents. IDLE is available on almost all operating systems these days.



MAKE SOMETHING HAPPEN

Open IDLE

First, open the IDLE environment. On Windows 10, click the **Start** button and then find IDLE in the Python group of programs (**Figure 1-5**).



Figure 1.5 Start IDLE

It might be worth adding IDLE to the Start menu or pinning it to the

taskbar. You can do this by right-clicking IDLE and selecting the appropriate option.

On macOS or Linux, open a terminal, type **idle**, and press **Enter**. It doesn't matter which operating system you use; once IDLE is running, you should see the IDLE command shell appear on the screen ([Figure 1-6](#)).

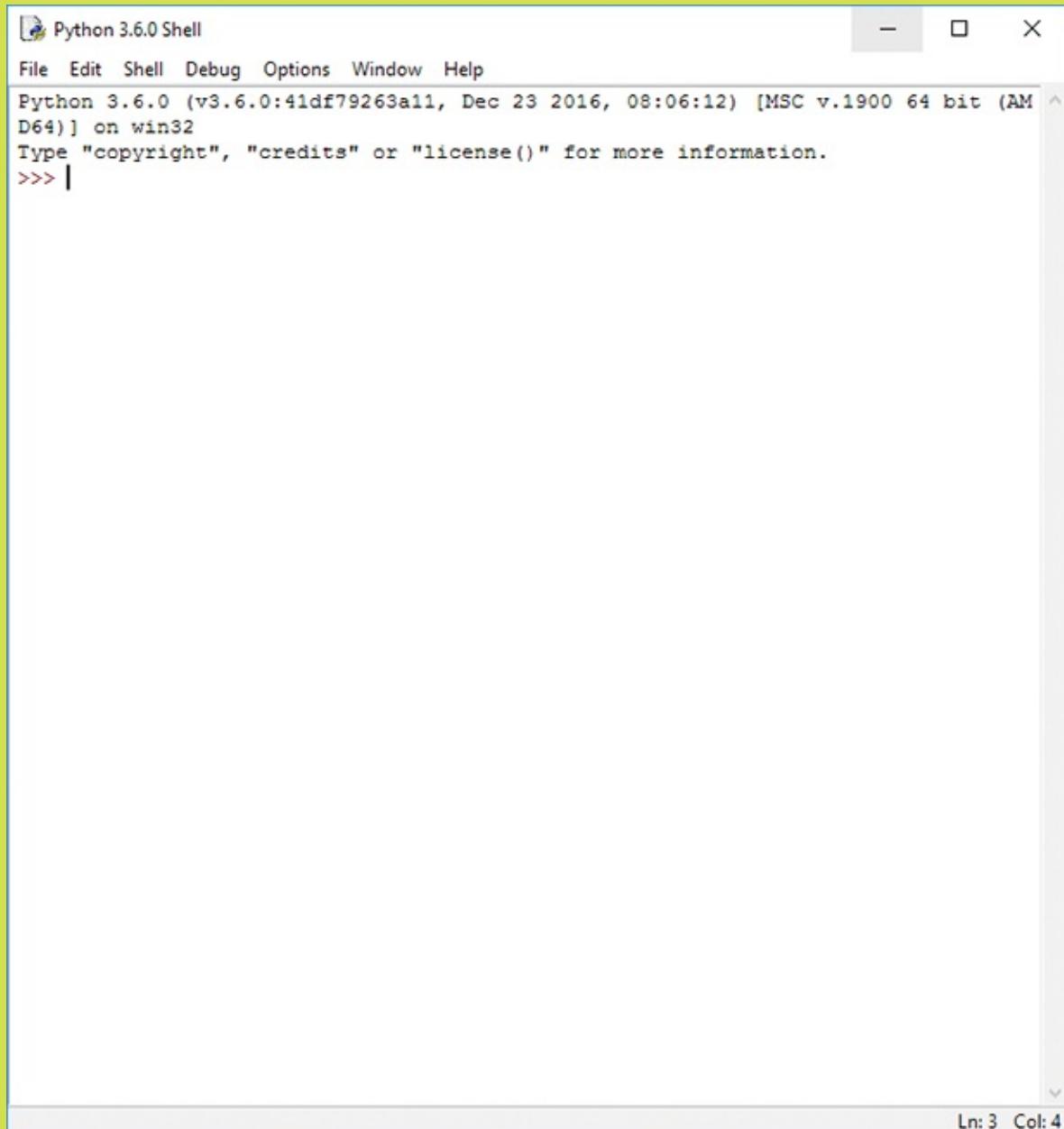
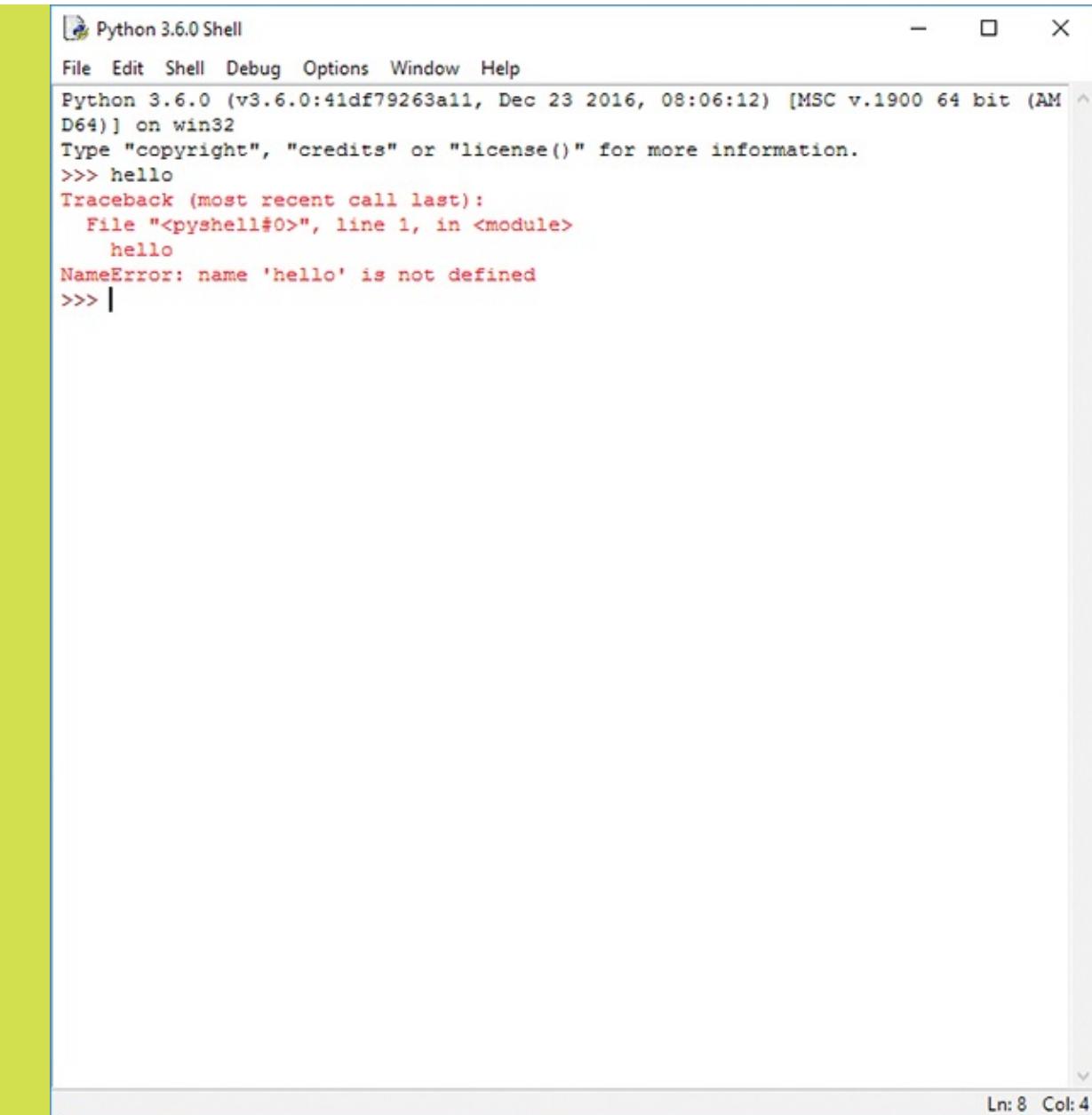


Figure 1.6 IDLE shell

A *shell* encloses or “wraps around” something. The IDLE program is enclosing the Python engine. The Python engine is what runs Python programs. The IDLE Python Shell takes Python commands that you type in, feeds them into the Python engine, and then displays the results.

Think of the IDLE shell as a waiter in a restaurant. You tell a waiter what you want to eat, he goes off to the kitchen and asks the chef to make the food, and then brings the food to your table. The waiter serves as a “shell” around the kitchen.

You can use IDLE to say hello to Python ([Figure 1-7](#)). Type hello and press the Enter key:



The screenshot shows a Windows desktop with a yellow taskbar at the bottom. On the taskbar, there is a green icon for 'Python 3.6.0 Shell'. To the right of the icon, the window title 'Python 3.6.0 Shell' is displayed. Below the title is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main window area contains the following text:

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM  
D64)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> hello  
Traceback (most recent call last):  
  File "<pyshell#0>", line 1, in <module>  
    hello  
NameError: name 'hello' is not defined  
>>> |
```

In the bottom right corner of the window, there is a status bar with 'Ln: 8 Col: 4'.

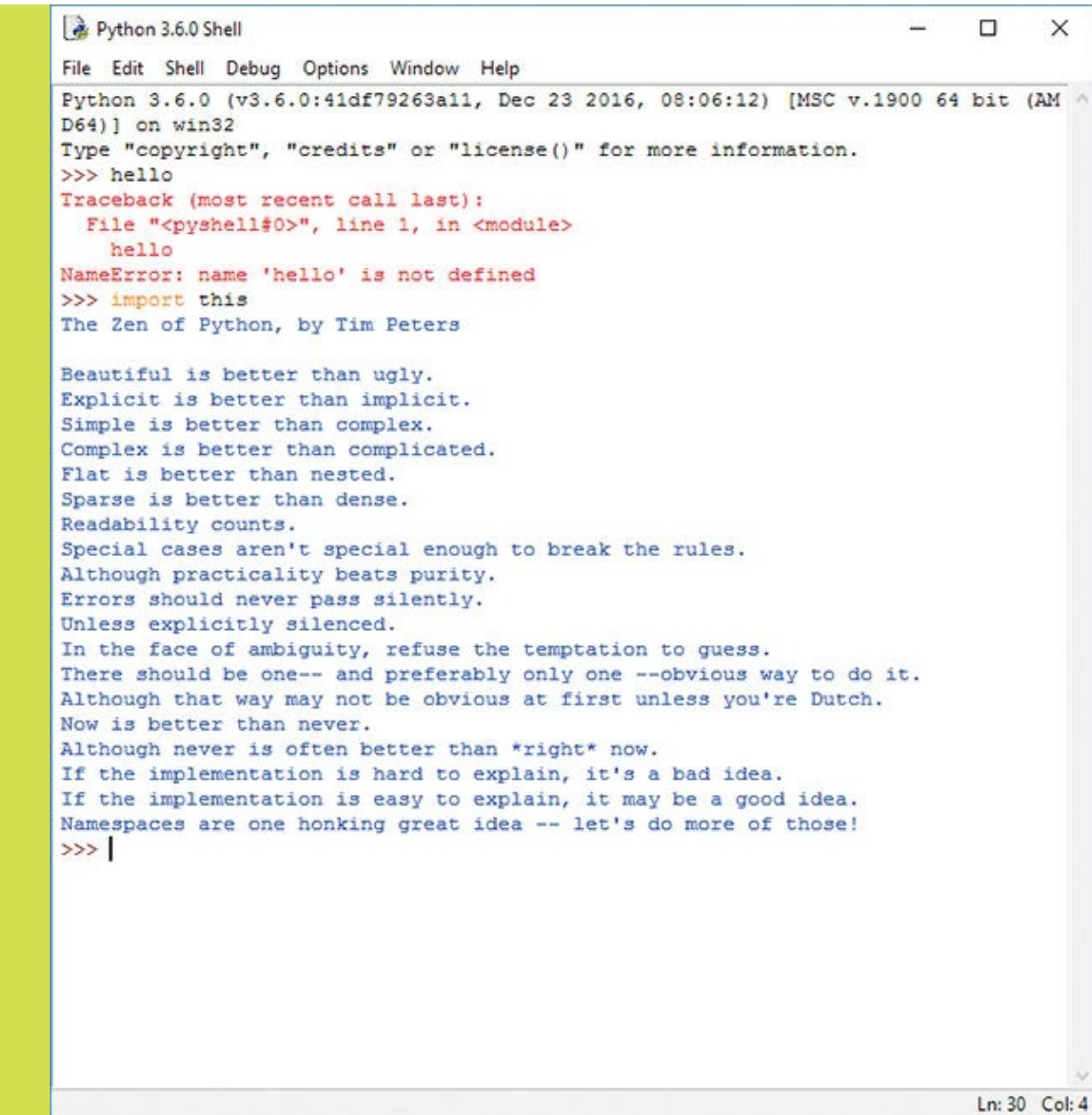
Figure 1.7 A failed hello

Hmmmm...this did not go well. As a rule of thumb, when the computer prints a message in red, this is usually bad news. In this case, the rather long-winded message in red is telling us that Python does not recognize the word “hello.” Whenever you type a command, the program behind the Python Shell will look through the list of words it understands and try to find one that matches. The word “hello” is not defined on this list, so Python issues this error message.

This is a bit like asking a waiter for a dish that the chef doesn't know how to cook.

I suppose that the Python Shell could have printed "I don't know what 'hello' means," but that would make it too easy. As you will find out (or may already know), computer error messages have a way of making simple mistakes sound really complicated.

In the next chapter, we'll explore in more detail how to use the Python Shell to discover what computers actually do when they run programs. However, for now, we can end our session with the shell by typing `import this` ([Figure 1-8](#)), which activates an Easter Egg.

A screenshot of the Python 3.6.0 Shell window. The title bar says "Python 3.6.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the "Zen of Python" code. The code is color-coded: blue for standard text and red for error messages. It starts with a traceback and ends with a final line ">>> |".

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.

>>> hello
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    hello
NameError: name 'hello' is not defined
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>> |
```

Figure 1.8 The Zen of Python

If you’re looking for a “Philosophy of Python,” then this is a very good one. Perhaps that is why this Easter Egg has survived as part of the language for so long. If some of the statements sound a bit profound, don’t worry, we’ll touch on what they mean as we go through the book.

What you have learned

In this first chapter, you built yourself a place to work by installing the Python language and the IDLE development environment.

You discovered that Python is a programming language, and that a programming language is a simplified version of English that allows programmers to tell a computer how to do things.

You had a brief conversation with the Python language and discovered that it is not terribly helpful if you just try to be friendly with it. Instead, you must give it exactly the right instructions. It can then be surprisingly chatty.

To reinforce your understanding of this chapter, you might want to consider the following “profound questions” about computers, programs, and programming.

What is the difference between a program and an application?

When people talk about software, you will find the words *program* and *application* used interchangeably. When I talk about a program, I am describing some code that instructs the computer what to do. I regard an application as something larger and more developed. An application brings together program code and assets, such as images and sounds, to make a complete experience for a user. A program can be as simple as a few lines of Python code.

Will “artificial intelligence” (AI) mean that one day we won’t have to write programs?

This is a very deep question. To me, artificial intelligence is a field in which lots of people are working very hard to make a computer very good at guessing. It turns out that by giving computers lots of information—and telling them how the information is related—a program can then use all this stuff to make a good guess as to the context of a statement.

I suppose that humans “guess” at the meaning of things. Maybe one day a doctor really will want me to drink a hot bath before I take my medicine (per the instructions at the beginning of this chapter), in which case I’ll do the wrong thing. However, humans have a much greater capacity to store and link experiences, which puts the computer at a distinct disadvantage when it comes to showing intelligence. Maybe in time, this will change. We are already seeing that in specific fields of expertise—for example, finance, medical diagnosis, and

artificial intelligence—can do very well.

However, when it comes to telling computers exactly what we want them to do, we'll need programmers for quite a long time...certainly long enough for you to pay off your mortgage.

Is IDLE the only tool for writing programs?

No. There are a great many tools that you can use to create programs. Some are tied to one particular programming language, and others are more general purpose. My personal favorite is a tool called *Visual Studio*. It can be used with many programming languages, including Python. However, Visual Studio is probably a bit complex for people getting started in programming; it's like learning to drive in a race car. We'll look at Visual Studio and its cousin, Visual Studio Code, in [Part 3](#) of this book.

What do I do if I break the program?

Some people worry that things they do with a program on the computer might “break” it in some way. I used to worry about this too, but I’ve conquered this fear by making sure that whenever I do something I always have a way back. You are currently in that happy position. You now know how to get Python on your computer, and it’s unlikely that you’ll damage your Python installation simply by running programs in it. Even if everything goes horribly wrong, and you end up breaking a program such that it won’t work again, you can simply install a clean copy of the program and start again.

2 Python and programming



What you will learn

In this chapter, you'll start working with Python. However, before you do, we'll take on a bit of detective work and discover what makes a programmer and what a computer program really does. We'll also look at the Python programming language and discover how it fits in to software development.

[What makes a programmer](#)

[Computers as data processors](#)

[Data and information](#)

[Work with Python functions](#)

What you have learned

What makes a programmer

If you have not programmed before, don't worry. Programming is not rocket science. The hard part about learning to program is that when you start you get hit with a lot of ideas and concepts, which can be confusing. However, if you think learning to program sounds like challenging work and that you might not be able to do it, I strongly suggest that you put those thoughts aside.

Programming is as easy as organizing a birthday party for a bunch of kids.

Programming and party planning

If you were organizing a kid's birthday party, you'd have to decide who to invite. You'd have to remember who wanted the vegetarian pizza and which kids can't sit next to each other without causing trouble. You'd have to work out what presents each kid would take home and what everyone would do at the party. You'd have to organize the timing so that the magician doesn't arrive just as the food is served. To help, you'd organize the party using lists like those in **Figure 2-1**. Programming is just like this; it's all about organization.

| GUEST LIST | MENU | SCHEDULE | GUEST PRESENTS |
|------------|--------------|--------------------|-----------------|
| Rob | Pizza | 3:00 pm Arrival | Hat |
| Mary | Chips | 3:30 pm Xbox Games | Whistle (maybe) |
| David | Soda | 4:30 pm Food | Sweets |
| Jenny | Cola | 5:15 pm Magician | Puzzle |
| Chris | Orange Juice | | Book |
| Imogen | | | |
| Mo | | | |
| Sunil | | | |

Figure 2-1 Party planning is a lot like programming. You must stay organized.

If you can organize a party, you can write a program. What happens in a program is a little different, but the basic principles are the same. And because a program

contains elements that you create and manage (unlike unruly kids), you have complete control over exactly what happens. What's more, once you've done some programming, you might start to approach all tasks in a systematic way, so a bit of programming experience can turn you into a better organizer overall.

Programming is defined by most people as “earning huge sums of money doing something that nobody can understand.” I define programming as “determining a solution to a given problem and expressing it in a form that a computer system can understand and execute.” One or two things are inherent in this definition:

- You need to be able to solve the problem yourself before you can write a program to do it.
- The computer must be made to understand what you're trying to tell it to do.

You can think of a program as a bit like a recipe. If you don't know how to bake a cake, you can't tell someone else how to do it. And if the person you're talking to doesn't understand instructions such as “Fold the flour and sugar into the mix,” you still can't tell him how to bake the cake.

To create a program, you must take a solution you have worked out and then write it down in simple steps that the computer can perform.

Programming and problems

I also like to think of a programmer as a bit like a plumber. A plumber arrives at a job with a big bag of tools and spare parts. Having looked at the plumbing problem for a while, the plumber opens the bag, takes out various tools and parts, fits the parts together, and solves your problem. Programming is like that. You're given a problem to solve, and you have at your disposal a big bag of tools—in this case, a programming language. You look at the problem for a while and work out how to solve it, and then you fit the bits of the language together to solve the problem. The art of programming is knowing which bits you need to take out of your bag of tools to solve each part of the problem.

The art of taking a problem and breaking it down into a set of instructions you can give to a computer is the interesting part of programming. However, learning to program is not simply a matter of learning a programming language. Nor is

programming simply a matter of coming up with a program that solves a problem. You must consider many things when writing a program, and not all of them are directly related to the problem at hand. To start, let's assume that you're writing your programs for a customer. He or she has a problem and would like you to write a program to solve it. We'll also assume that the customer knows even less about computers than we do!

Initially, you're not even going to talk about the programming language, the type of computer, or anything like that; you are simply going to make sure that we know what the customer wants. Because programmers pride themselves on their ability to come up with solutions, as soon as they are given a problem they immediately start thinking of ways to solve it—this is almost a reflex action. Unfortunately, many software projects have failed because the problem they solved was the wrong one. Coming up with a perfect solution to a problem the customer doesn't have is something that happens surprisingly often in the real world. The developers of the software quite simply did not find out what was required or desired. Instead, they created what they thought was required. The customers assumed that because the developers stopped asking questions, the right solution was being built. Only at the final handoff was the awful truth revealed. It's very important that a programmer postpone making something until she knows exactly what is required.

The worst thing you can say to a customer right away is, "I can do that." Instead, you should first ask, "Is that what the customer wants?" "Do I really understand what the problem is?" Asking these questions is a kind of self-discipline. Before you solve a problem, you should be sure that you have a watertight definition of what the problem is, which is agreeable to both you and the customer.

In the real world, such a definition is sometimes called a *functional design specification*, or FDS. An FDS tells you exactly what the customer wants. Both you and the customer sign it, and if you provide a system that behaves according to the design specification, the customer must pay you. Once you have your design specification, you can think about ways of solving the problem.

You might think having a specification isn't necessary if you're writing a program for yourself, but this is not true. Writing some form of specification forces you to think about your problem at a very detailed level. It also forces you to think about what your system is not going to do. You need this clarity when building something for yourself as much as when working with a customer. The

specification sets expectations right at the start.

PROGRAMMER'S POINT

Specifications must always exist

I have written many programs for money. I would never write a program without getting a solid specification first. Defining a specification is essential even (or perhaps especially) when I do a job for a friend.

Modern development techniques put the customer at the heart of development and involve them in the design process in an ongoing way. These approaches are very helpful because it's very hard to get a definitive specification at the start of a project. As a developer, you don't really know much about the customer's business, and the customer doesn't know the limitations and possibilities of the technologies that you can use to solve the problem. It's a good idea to make a series of versions of the solution and discuss each version with the customer before moving on to the next one. We call this *prototyping*.

This approach to problem solving will serve you well irrespective of the programming languages that you're using. The issues of ensuring adequate specifications and avoiding assumptions are equally important when you try to organize anything, including a birthday party.

Programmers and people

Finding out what the customer wants is one of the most important aspects of any programming task. However, communication with other people is important in many other situations, too. Perhaps you want to convince a wealthy backer that you have an idea for the next big thing; perhaps you want to persuade a potential customer that you have the best solution to their problems.

Not all programmers are great communicators in the beginning. However, the important thing to remember is that communication skills can be learned, just like a new programming language. Becoming a better communicator might mean going outside your comfort zone—nobody likes standing in front of an audience for the first time—but with practice, you can master communication skills and vastly increase your chances of going a long way in this business.

Effective communication also extends to writing. The ability to create text that others can read is a very useful skill, and again, the best way to do this is with practice. My advice is to start writing a blog or a diary. It doesn't matter that only your mom reads your blog at first; the important thing is that you write regularly. If you write about something you're interested in (I write about programming—surprise, surprise—at www.robmiles.com), you will quickly become much better at it.



PROGRAMMER'S POINT

Programmers who can communicate well get the most money and the most interesting work

It's possible to make a good living from programming even if you can communicate only in single words and grunts—as long as you can write code quickly that meets the given requirements. But the interesting tasks go to developers who can communicate well. They are the ones who can sell their ideas and are best at talking to customers to find out what the customer wants.

Computers as data processors

Now that we know what programmers do, we can start to consider what a computer is and what makes it so special.

Machines and computers and us

Humans are a race of toolmakers. We invent things to make our lives easier, and we've been doing it for thousands of years. We started with mechanical devices, such as the plow, which made farming more efficient, but in the last century we've moved into electronic devices and, more recently, into computers.

As computers became smaller and cheaper, they found their way into things around us. Many devices (for example, the mobile phone) are possible only because we can put a computer inside to make them work. However, we need to remember what the computer does; it automates operations that formerly required brain power. There's nothing particularly clever about a computer; it simply follows the instructions that it's been given.

A computer works on data in the same way that a sausage machine works on meat: something is put in one end, some processing is performed, and something comes out the other end. You can think of a program as similar to the instructions a coach gives to a football or soccer team before a play. The coach might say something like, “If they attack on the left, I want Jerry and Chris to run back, but if they kick the ball down the field, I want Jerry to chase the ball.” Then, when the game unfolds, the team will respond to events in a way that should let them outplay their opponents.

However, there is one important distinction between a computer program and the way a team might behave in a football game. A football player would know when given some senseless instructions. If the coach said, “If they attack on the left, I want Jerry to sing the first verse of the national anthem and then run as fast as he can toward the exit,” the player would raise an objection.

Unfortunately, a program is unaware of the sensibility of the data it is processing, in the same way that a sausage machine is unaware of what meat is. Put a bicycle into a sausage machine, and the machine will try to make sausages out of it. Put meaningless data into a computer, and it will do meaningless things with it. As far as computers are concerned, data is just a pattern of signals coming in that must be manipulated in some way to produce another pattern of signals. A computer program is the sequence of instructions that tell a computer what to do with the input data and what form the output data should have.

Examples of typical data-processing applications include the following (as shown in **Figure 2-2**):

- Mobile phone—A microcomputer in your phone takes signals from a radio and converts them into sound. At the same time, it takes signals from a microphone and makes them into patterns of bits that will be sent out from the radio.
- Car—A microcomputer in the engine takes information from sensors telling it the current engine speed, road speed, oxygen content of the air, accelerator setting, and so on. The microcomputer produces voltages that control the fuel injection settings, the timing of the spark plugs, and other things to optimize engine performance.
- Game console—A computer takes instructions from the controllers and

uses them to manage the artificial world that it is creating for the gamer.

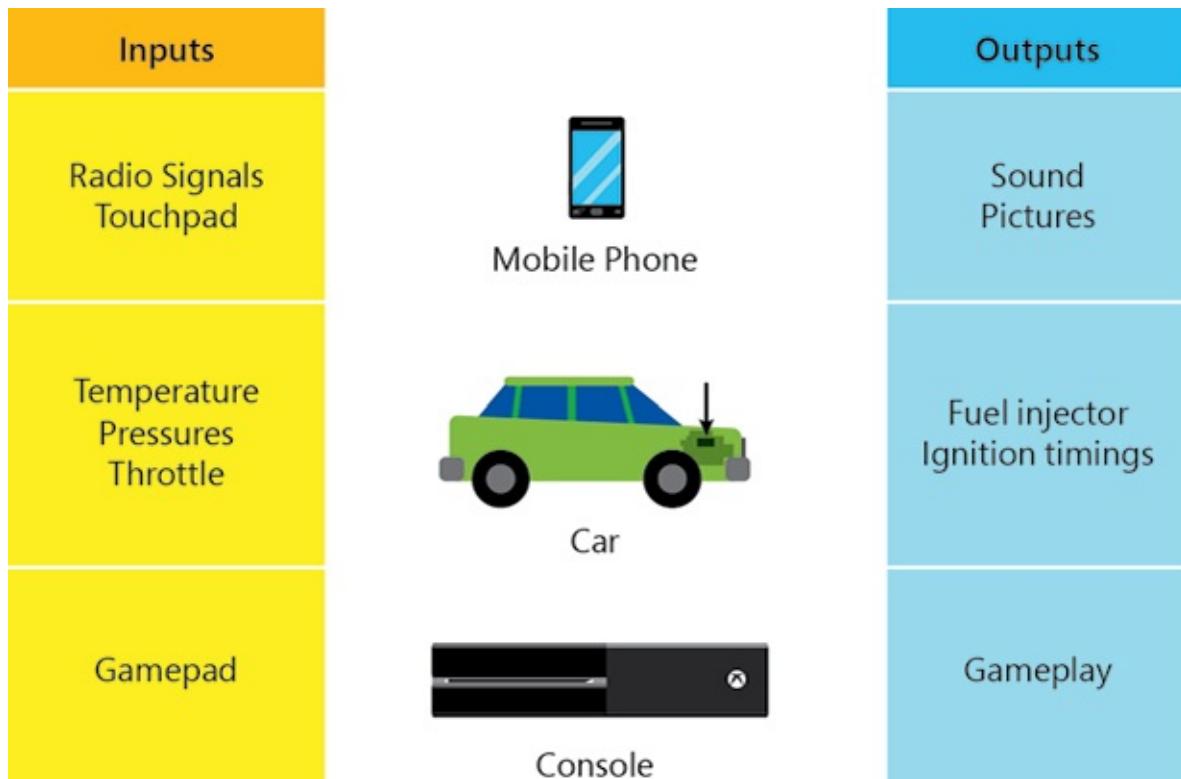


Figure 2-2 Computers in devices

Most reasonably complex devices created today contain data-processing components to optimize their performance, and some exist only because we can build in such capabilities. The growth of “The Internet of Things” is introducing computers into an enormous range of areas. It’s important to think of data processing as much more than working out the company payroll—calculating numbers and printing out results (the traditional uses of computers). As software engineers, we will inevitably spend a great deal of our time fitting data-processing components into other devices to drive them. These embedded systems mean many people will be using computers even if they’re not even aware of it!



PROGRAMMER’S POINT

Software might be a matter of life and death

Remember that seemingly innocuous programs can have life-threatening capabilities. For example,

a doctor may use a spreadsheet you have written to calculate doses of drugs for patients. In this case, a defect in the program could result in physical harm. (I don't think doctors do this—but you never know.) For a deeply scary description of what can go wrong when programmers don't pay attention to the fundamentals, search for **Therac-25** on the web.

Programs as data processors

Figure 2-3 shows what every computer does. Data goes into the computer, which does something with it, and then data comes out of the computer. What form the data takes and what the output means is entirely up to us, as is what the program does.

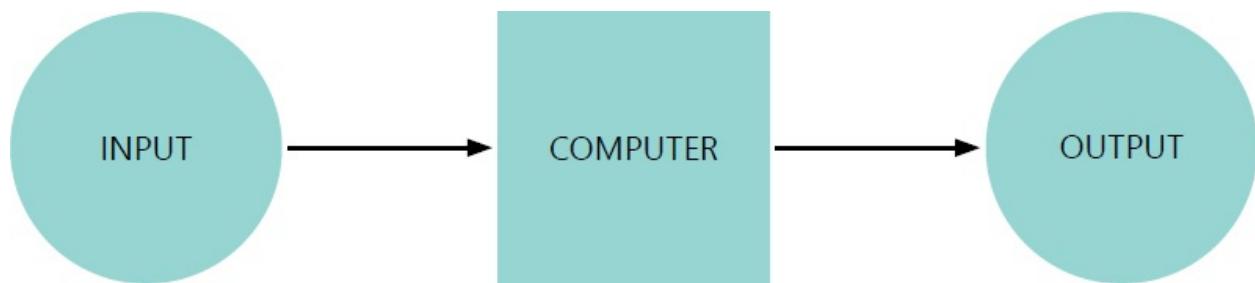


Figure 2-3 A computer as a data processor

As mentioned earlier, another way to think of a program is like a recipe, which is illustrated in **Figure 2-4**.

In this example, the cook plays the role of the computer and the recipe is the program that controls what the cook does with the ingredients. A recipe can work with many different ingredients, and a program can work with many different inputs, too. For example, a program might take your age and the name of a movie you want to see and provide an output that determines whether you can go see that movie based on its suitability rating.

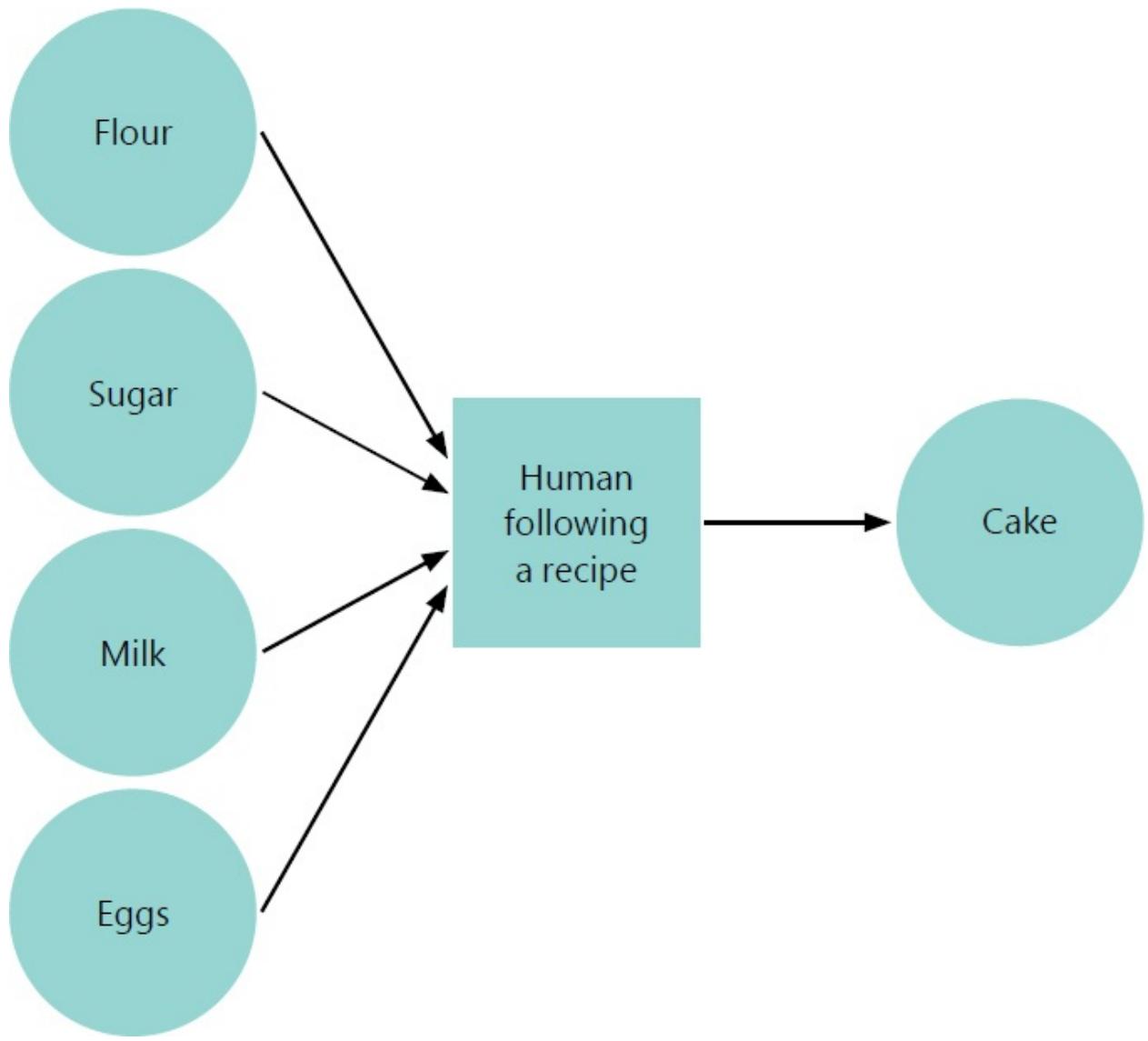


Figure 2-4 Recipes and programs

Python as a data processor

You can regard Python itself as a data processor (see [Figure 2-5](#)). Code written in Python goes into the Python engine, which then produces some output.

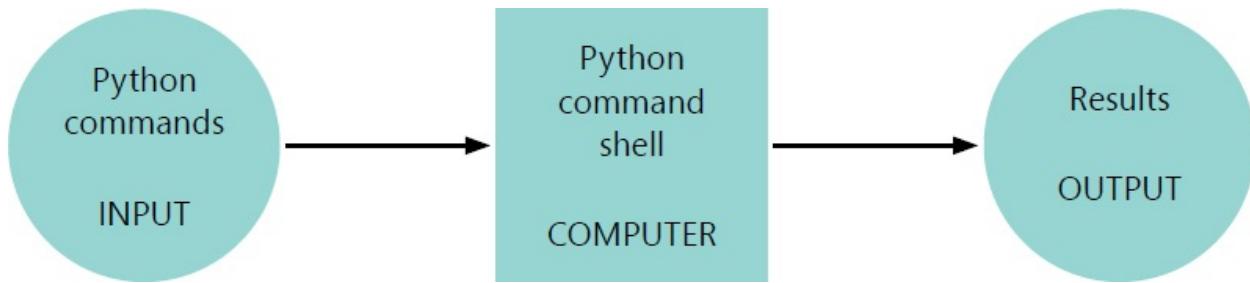


Figure 2-5 Python as a data processor

Sometimes, as we have seen, the output is an error (for example, if we type **hello**). Other times, it can be a philosophical statement on the nature of the Python language (as when we type **import this**). Let's try using the Python command shell to find out more about how the language works.



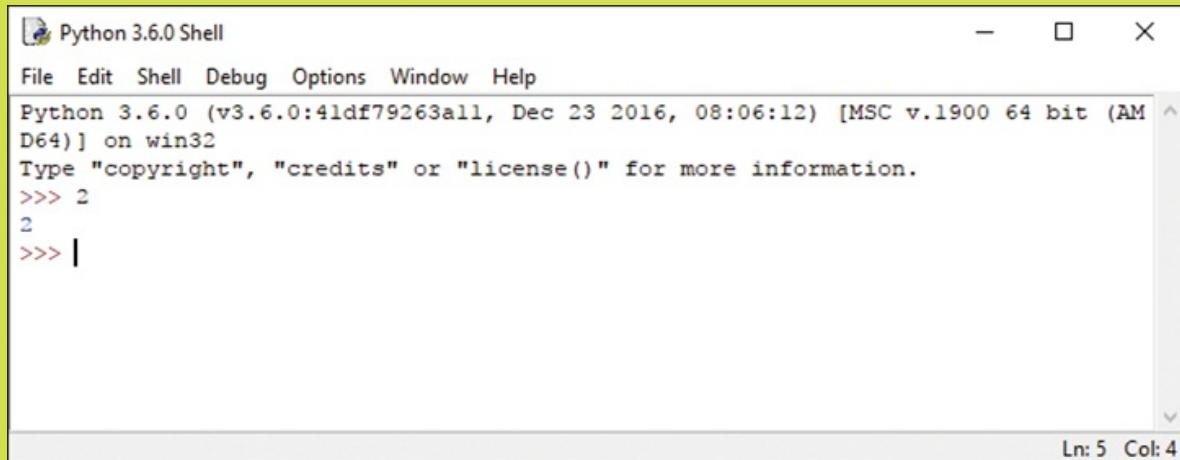
MAKE SOMETHING HAPPEN

Have a conversation with Python

The last time we chatted with Python, we didn't say much. We'll now have a more in-depth conversation and see what we can find out about how the language works. First, we must use the **IDLE** command to start up the Python Shell as we did in the previous chapter:

In [Chapter 1](#), we tried to say **hello** to Python, but it didn't end well. So,

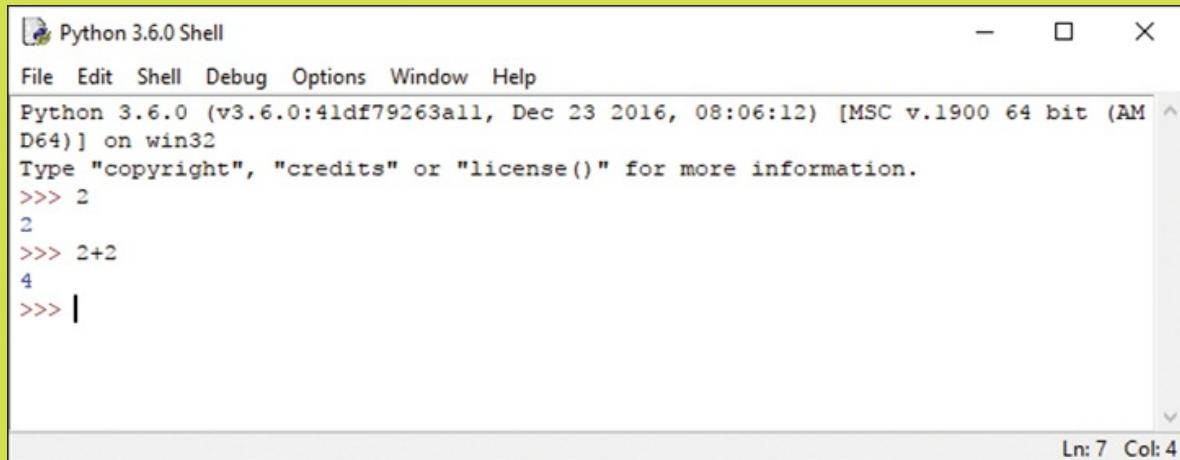
let's give the command shell something that we know computers understand—perhaps a number. Type the value **2** and press **Enter**:



```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263all, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2
2
>>> |
```

Ln: 5 Col: 4

This time we don't get an error; we just get the value **2** coming straight back to us. It looks as if the Python Shell might be working out an answer and sending it back to us. We can prove this by giving it a sum, for example **2+2**:



```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263all, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2
2
>>> 2+2
4
>>> |
```

Ln: 7 Col: 4

This time, rather than echoing **2+2**, the Python Shell seems to have evaluated the result and returned that to us.

It appears that the Python Shell is taking our instructions and acting on them in some way. In fact, this is exactly what's happening. At its heart, Python is an expression evaluator, which is a fancy way of saying it works things out for us. Give the Python Shell an expression, and it will respond with the answer. **Figure**

2-6 shows how a simple expression appears.

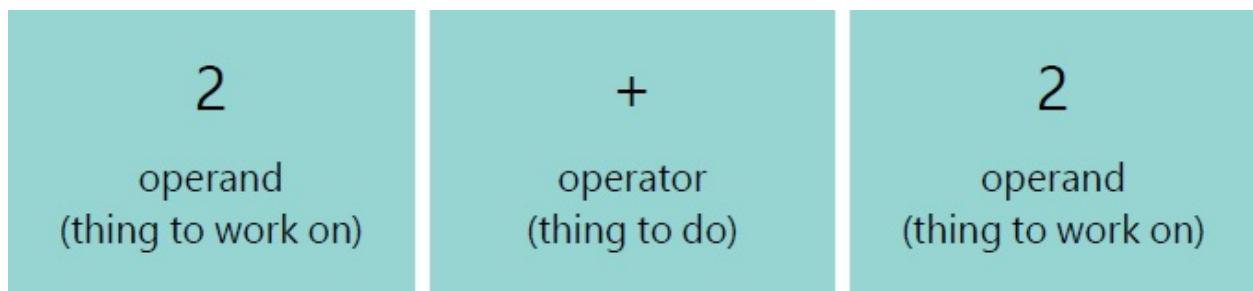


Figure 2-6 The anatomy of a simple expression

Items that the expression works on are called *operands*. Things that do the actual work are called *operators*. In the case of $2+2$, there are two operands (the two values of 2) and one operator (the plus). When you feed an expression into the Python Shell, it identifies the operators and operands and then works out the answer.



WHAT COULD GO WRONG

Bad expressions

We've already seen what can happen if you type something that Python doesn't understand. You get an error message. The same kind of thing will happen if you give Python an invalid expression.

A screenshot of the Python 3.6.0 Shell window. The title bar says "Python 3.6.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area shows the Python interpreter prompt "Python 3.6.0 (v3.6.0:41df79263all, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32". It then shows the command "Type "copyright", "credits" or "license()" for more information." followed by the input "2". The next line "2" is in red, indicating an error. The subsequent line "2+" is also in red, followed by the error message "SyntaxError: invalid syntax". The bottom status bar shows "Ln: 9 Col: 4".

A programmer has entered the expression `2+`. This expression is not valid, so the Python Shell has displayed an unhappy red bar and a red error message.

Python is very good at working out expressions. You can use Python rather than a calculator if you like. Expressions are worked out in the same way that a mathematician would do the calculation, doing things like performing multiplication before addition and obeying parentheses.

We can do some experiments using the Python Shell to investigate expressions. From now on, rather than showing you screenshots of the Python Shell, I'll just show the output that you'll see in IDLE. In other words, the previous three Python commands that we have issued would look like this:

[Click here to view code image](#)

```
>>> 2
2
>>> 2+2
4
>>> 2+
SyntaxError: invalid syntax
```

The typed text is shown in black, the output from Python is shown in blue, and the command prompts are shown in brown. Bad things are shown in red.



CODE ANALYSIS

Python expressions

Now and then, you'll see “Code Analysis” sections that pose questions about the code we have just seen. You might try answering the

questions yourself before reading the answer.

Question: What do you think would happen if you tried to evaluate $2+3^4$?

Answer: The `*` (asterisk) operator means multiply. Python uses the asterisk in place of the `*` (multiplication symbol) used in math. In math, we always perform higher-priority operations like multiply and divide before addition, so I'd expect the expression above to display the value `14`. The calculation 3^4 would be worked out first, giving an answer of `81`, and this would be added to the value `2`. If you try this in IDLE, you should see what you would expect:

```
>>> 2+3^4  
14
```

Question: What do you think would happen if you tried to evaluate $(2+3)^4$?

Answer: The parentheses enclose calculations that should be worked out first, so in the above expression, I'd expect to see the value `5` calculated $(2+3)$ and then this value to be multiplied by `4`, giving a result of `20`.

```
>>> (2+3)^4  
20
```

Question: What do you think would happen if you tried to evaluate $(2+3^4)$?

Answer: This one is quite interesting. You should try it with the Python Shell. What happens is that Python says to itself, “The expression I’m trying to work out is incomplete. I need a closing parenthesis.” So, the Python Shell waits for more input from you. If you type in the closing parenthesis and complete the expression, the value is calculated and the result is displayed. You can even add

more sums on the second line if you want.

```
>>> (2+3*4  
)  
14
```

Question: What do you think would happen if you tried to evaluate `)2+3*4?`

Answer: If the Python Shell sees a closing parenthesis before it sees an opening one, it instantly knows that something is wrong and displays an error.

```
>>> )2+3*4  
SyntaxError: invalid syntax
```

Note that the command shell is trying to help you work out where the error is by highlighting the incorrect character.

Python as a scripting language

We can use the Python Shell for having conversations like this because Python is a “scripting” programming language. You can think of the Python Shell as a kind of “robot actor” who will perform whatever Python commands you give it. In other words, you tell the command shell what you want your program to do using the Python language. If the instructions don’t make sense to the “robot actor,” it tells us it can’t understand them (usually with red text).

The process of taking a program and then acting on the instructions in it is called *interpreting* the program. Actors earn a living interpreting the words of a play; computers solve problems for us by interpreting program instructions.



PROGRAMMER'S POINT

Not all programming languages run like Python

Not all programming languages are “scripting” languages, which are interpreted in the same way as Python. Sometimes program instructions are converted into the very low-level instructions that the hardware of your computer understands. This process is called compilation, and the program that performs this conversion is called a compiler. The compiled instructions can then be loaded into the computer to be executed. This technique produces programs that can run very fast, because when the compiled low-level instructions are performed, the computer doesn’t have to figure out what the instructions mean; they can just be obeyed.

You might think this means that Python is a “slow” computer language, because each time a Python program runs, the “robot actor” must work out the meaning of each command before performing it. However, this is not really a problem because modern computers run very, very fast, and Python uses some clever trickery to compile your program as it runs.

Data and information

Now that we understand computers as machines that process data, and we understand that programs tell computers what to do with the data, let’s delve a little bit deeper into the nature of data and information. People use the words data and information interchangeably, but it’s important to make a distinction between the two, because the way that computers and humans consider data is completely different. Look at [Figure 2-7](#), which shows the difference.

What the computer sees

```
87 104 101 110 32 105 110 32 116 104  
101 32 67 111 117 114 115 101 32 111  
102 32 104 117 109 97 110 32 101 118  
101 110 116 115 44 32 105 116 32 98  
101 99 111 109 101 115 32 110 101 99  
101 115 115 97 114 121 32 102 111  
114 32 111 110 101 32 112 101 111  
112 108 101 32 116 111 32 100 105  
115 115 111 108 118 101 32 116 104  
101 32 112 111 108 105 116 105 99 97  
108 32 98 97 110 100 115 32 119 104  
105 99 104 32 104 97 118 101 32 99  
111 110 110 101 99 116 101 100 32  
116 104 101 109 32 119 105 116 104  
32 97 110 111 116 104 101 114 44 32  
97 110 100 32 116 111 32 97 115 115  
117 109 101 32 97 109 111 110 103 32  
116 104 101 32 112 111 119 101 114  
115 32 111 102 32 116 104 101 32 101  
97 114 116 104 44 32 116 104
```

What we see



Figure 2-7 Data and information

The two items in [Figure 2-7](#) contain the same data, except that the image on the left more closely resembles how the document would be stored in a computer. The computer uses a numeric value to represent each letter and space in the text. If you work through the values, you can figure out each value, beginning with the value 87, which represents an uppercase W (in the “When” that begins the first regular paragraph in the document on the right).

Because of the way computers hold data, yet another layer lies beneath the mapping of numbers to letters. Each number is held by the computer as a unique pattern of on and off signals, or **1**s and **0**s. In the realm of computing, each **1** or **0** is known as a *bit*. (For a wonderful explanation of how computers operate at this level and of how these workings form the basis for all coding, see Charles Petzold’s *Code: The Hidden Language of Computer Hardware and Software*.) The value **87**, which we know means “uppercase W,” is held as the following way:

1010111

This is the *binary* representation of the value. I don't have the space to go into precisely how this works (and Charles Petzold already did this!), but you can think of this bit pattern as meaning "87 is made up of a 1 plus a 2 plus a 4 plus a 16 plus a 64."

Each of the bits in the pattern tells the computer hardware whether a particular power of two is present. Don't worry too much if you don't fully understand this, but do remember that as far as the computer is concerned, data is a collection of 1s and 0s that computers store and manipulate. That's data.

Information, on the other hand, is the interpretation of the data by people to mean something. Strictly speaking, computers process data and humans work on information.

For example, the computer could hold the following bit pattern somewhere in memory:

11111111 11111111 11111111 00000000

You could regard this as meaning "You are \$256 overdrawn at the bank" or "You are 256 feet below the surface of the ground" or "Eight of the thirty-two light switches are off." The transition from data to information is usually made when a human reads the output.

I am being so pedantic because it is vital to remember that a computer does not "know" what the data it is processing means. As far as the computer is concerned, data is just patterns of bits; it is the user who gives meaning to these patterns. Remember this when you get a bank statement that says you have \$8,388,608 in your account when you really have only \$83!

Data processing in Python

We now know that Python is a data processor. A script written in Python is interpreted by the Python system, which then produces some output. We also know that within the computer running a Python program, data values are represented by patterns of bits (ons and offs).



MAKE SOMETHING HAPPEN

Work with text in Python

Let's try to say "hello" to Python in a way that it understands. Return to the Python Shell in IDLE and enter the word, '**hello**'. This time, however, enclose the word in single quote characters:

```
>>> 'hello'  
'hello'
```

This time we don't get any errors, Python just echoes the text that was entered. If you compare this with the behavior we saw when we entered a number, you'll notice that in fact Python did the same thing. Previously, we entered the value **2**, and Python echoed **2**. When we entered a text value, Python echoed the text. The next thing we tried to do with numbers was add them. Let's try that with text strings:

```
>>> 'hello' + ' world'  
'hello world'
```

This is nice; Python is behaving exactly as we would expect. We know that when we give Python a sum to work out, it calculates the result and then returns it. We used this to add **2** and **2**. Now we've discovered that we can also use the same procedure to add '**hello**' to '**world**'. Note how I rather cleverly put a space in front of the word '**world**'; otherwise, the program would have displayed '**helloworld**'.

There is also something clever going on inside Python, in that the way **+** (the addition) behaves is correct for both numbers and strings. If we ask Python to add two numbers, it returns their sum. If we ask Python to add two strings, it returns one string added to the end of the other.



CODE ANALYSIS

Break the rules with Python

Question: What do you think would happen if you missed the closing quote of a string you were typing?

Answer: From our experiments with parentheses, you might expect Python to wait patiently on the next line for you to type in the rest of the string. Unfortunately, this does not happen.

[Click here to view code image](#)

```
>>> 'hello
SyntaxError: EOL while scanning string
literal
```

A “string literal” is a string of text that is “literally” just there in the text. The letters EOL are an abbreviation for “End Of Line.” The Python Shell is saying that it doesn’t like you to put line endings into strings. Python regards individual operands (numbers and strings) as things that are not allowed to span multiple lines. There’s nothing wrong with creating a text expression that spans several lines (you can try this), but Python does not allow the operands of that expression to span multiple lines.

Question: What do you think would happen if you tried to subtract one string from another?

Answer: Python is clever enough to know that, while it is perfectly sensible to use the `-` (subtraction) behavior to mean “subtract one integer from another” (you can try this if you like), it is not sensible

for a program to try to subtract one string from another.

[Click here to view code image](#)

```
>>> 'hello' - ' world'  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    'hello' - ' world'  
TypeError: unsupported operand type(s) for -:  
  'str' and 'str'
```

Python is giving us details of what went wrong, but rather than saying, “Subtracting strings is stupid,” it gives a description that is much harder to understand. To make sense of this, you need to know that the word operand means “something that an operator works on.” In this case, the operator is the `-` (minus) operator, and operands are two strings (`hello` and `world`). Python is saying that you can’t put the minus operator between two strings.

Question: What do you think would happen if you tried to add a number to a string?

Answer: Adding a number to a string is as stupid as subtracting one string from another, and so you would expect Python to give you an error. But you’d also expect the error to be hard to understand:

[Click here to view code image](#)

```
>>> 'hello' + 2  
Traceback (most recent call last):  
  File "<pyshell#14>", line 1, in <module>  
    'hello' + 2  
TypeError: must be str, not int
```

Hopefully, this time the message should make a bit more sense. Python is saying that something “must be a string, not an integer” (although it is not very helpful in that it doesn’t tell you which thing

is wrong).

If we really wanted to put the digit `2` on the end of the word `'hello'`, we could do this by enclosing the digit in quotes:

```
>>> 'hello' + '2'  
'hello2'  
>>>
```

Question: What do you think would happen if you tried to multiply a string by a number?

Answer: It works. The string is repeated the given number of times:

```
>>> 'hello' * 3  
'hellohellohello'
```

Python will try to do something sensible when it can. This expression still works if the order of the operands is the other way around. Python will also do something sensible if you try to multiply a string by zero or a negative number.

Text and numbers as data types

If you look carefully at the results of the statements above, you'll notice that when Python evaluates a numeric expression (one that creates a number as a result), it returns just the digits, but if it evaluates a string, it returns text enclosed in quotes because Python obeys the rules about how various kinds of data are expressed.

Python enforces a strict separation between numeric data (the value `2`) and text data (the string `hello`). The way values are stored, and the effect of operations on them, is different for each data type, even though the operation might have the same operator for each type. We can use the `+` operator to add numbers or

strings of text, and Python ensures that the correct thing happens because Python works out the context of the action. If it sees a `+` operator between two numbers, it will use the numeric version of `+`. If it sees a `+` operator between two strings, it will use the string version of `+`.

Humans do the same kind of thing. We talk about washing our face, washing the dishes, or washing a horse, and although the action will be fundamentally the same (we are washing something), what we actually do will be different in each case. It would be possible for a language to have different words for “wash,” one of which meant “washing a horse,” but English doesn’t work this way (although some languages do). If we just use the word “wash,” the reader must use brain power and experience to work out what’s going on. Of course, brain power and experience is just what a computer doesn’t have very much of, and so when we write a program, we must be consistent about the way we express what we want. The design of Python (and other programming languages) force this to happen.

Work with Python functions

Now that we know something about how Python works with text and numbers, we can start to investigate how text elements are actually represented as numbers, and even patterns of bits. To do this, we’ll use Python itself to investigate how it stores values. We’ll use some of the functions that are built in to the Python language.

A *function* is a behavior with a distinct name. If you were writing a script for an actor to perform, you could include stage directions such as “Move left,” “Look out of a window,” or my personal favorite, “Exit pursued by a bear.” These would trigger actions that the actor would perform during the play. You can regard these actions as “functions” that the actor knows how to perform. Python is like an actor. It knows how to perform a set of built-in functions. We’ll use some of these functions to investigate how text is represented in a computer.



PROGRAMMER’S POINT

Functions are an important part of any programming language

A big part of learning a programming language is learning the functions that it provides. In the next few chapters, we'll learn quite a few functions, and later we will start writing our own functions, too.

Each Python function has a distinct name and might be given data to work on. In this respect, you can think of a function as a tiny data processor, in that something goes into the function and a result is generated.

The `ord` function

One of the actions that Python knows how to perform is called `ord`. The name is short for the “ordinal value.” If you look up “ordinal,” you’ll find a highly confusing description (or at least I did). What it means in this context is, “Give me the value that represents this character in the sequence of possible character numbers.” Or, in shorter form, “Give me the number that represents this character.”

A function is called by giving the name of the function, followed by the things that the function works on, enclosed in parentheses. **Figure 2-8** shows the anatomy of a call of the `ord` function. The programming name for “the thing that a function works on” is an argument.

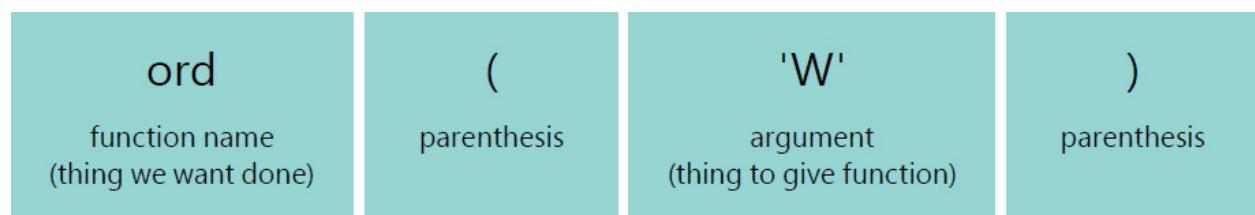


Figure 2-8 The anatomy of a function call



MAKE SOMETHING HAPPEN

Investigate text representation using `ord`

Let's use `ord` to investigate how text is stored inside the computer. We

can start by finding the number used to represent a particular character. We can feed the `ord` function a string that contains just a single `W` character and see what number comes out. Enter the commands into the Python Shell in IDLE and note what comes back.

```
>>> ord('W')
87
```

This is exactly what we saw in [Figure 2-8](#) above. The `W` in the first word of the Declaration of Independence was shown as the value `87`.

However, we need to be careful when we write our Python expression. The `W` that we are interested in must be part of a string of text, so it must be enclosed in the single quotes. If we omit the quotes, the Python system thinks we are asking about something called `W` and tells us it doesn't know anything about such a thing.

[Click here to view code image](#)

```
>>> ord(W)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    ord(W)
NameError: name 'W' is not defined
```

You won't use the `ord` function a lot in your Python programs, but it does provide a very useful window into the way values are manipulated by Python code.

The `chr` function

The `ord` function is complemented by a Python function called `chr`. This function takes a number and delivers the character represented by that number.



MAKE SOMETHING HAPPEN

Convert numbers to text using `chr`

There are absolutely no prizes for working out what you would see if you fed the value `87` into the `chr` function, but you should try it anyway.

```
>>> chr(87)
'W'
```

The numbers that represent the characters are arranged in a sensible way, in that if you display the character corresponding to the value `88`, you get just what you would expect:

```
>>> chr(88)
'X'
```

Within the computer world, international standards map particular values to particular characters. I've been writing computer programs for a very long time, and because of this experience, I happen to know that the capital letter `A` is represented by the value `65`, and the letter space (which is very important because it allows us to put gaps between words) is represented by the value `32`. However, there's normally no need to learn these numbers because Python and the underlying operating system will take care of text display. (I've also discovered that knowing these numbers doesn't really help me impress people at parties.)

Investigate data storage using `bin`

You can regard computer memory as a huge expanse of little boxes, each of

which has a unique numeric address. Each memory location comprises 8 individual bits, which can be either on or off (as we saw earlier). A memory location like this is called a byte. When you brag about your computer having “16 gigabytes of memory,” you are really saying that the computer contains sixteen thousand million individual locations, each of which is the size of a byte.

A single byte can’t store a great range of values, so several bytes can be grouped together to store larger numbers. Later, we’ll look at how this works, and the kinds of values that Python can store, but for now let’s just look at the patterns of bits that are used to hold values.

The `bin` function built into Python takes a number and returns a string of bits that represents the value of that number.



MAKE SOMETHING HAPPEN

Discover binary representation

We can use the `bin` function to investigate how data is stored in the computer.

```
>>> bin(87)
'0b1010111'
```

The `bin` function returns a string that gives the binary representation of that number. Note that this is a string of `0`s and `1`s. The string is prefixed by the characters `0b` which tell the reader that this is a binary representation of a number.



CODE ANALYSIS

Building binary numbers

The `bin` function is not used much in Python programs (unless you are lucky enough to have a customer ask you to write a program that displays binary values), but we can use it to investigate how numbers are stored in a computer. Remember that inside the computer hardware all data is manipulated in terms of either a high or a low voltage present on the given signal. We can think of these as `0` (no voltage) and `1` (some voltage).

Question: What does the binary value of `0` look like?

Answer: We can discover this by using the `bin` function again.

```
>>> bin(0)
'0b0'
```

The value of `0` in binary looks like any other zero.

Question: What does the binary value of `1` look like?

Answer: We can discover this by feeding `1` into the `bin` function.

```
>>> bin(1)
'0b1'
```

The value of `1` in binary looks exactly like any other `1` that we have seen before. So far, there doesn't seem to be anything special about binary values.

Question: What does the binary value of `2` look like?

Answer: We can discover this by feeding `2` into the `bin` function.

```
>>> bin(2)  
'0b10'
```

This is different. To understand how, let's start by considering the decimal number `10`. In this number, the numeral `1` tells us how many tens are in the number. Binary works in the same way, except that the numeral `1` tells us how many twos there are in the number. So, the value `10` in binary means two.

Question: What do you think the binary value `11` means?

Answer: I suspect that binary `11` is the decimal value `3` (a `1` plus a `2`) We can use the `bin` function to test this.

```
>>> bin(3)  
'0b11'
```

The third bit will be the number of fours in a number, the fourth bit will be the number of eights, and so on. You might want to experiment with `bin` to see what the patterns of bits are for numbers and then check your results.

Question: How does the binary value of `86` differ from the binary value of `87`?

Answer: We can discover this by using the `bin` function again.

```
>>> bin(86)  
'0b1010110'  
>>> bin(87)  
'0b1010111'
```

If you compare the two binary patterns, you'll notice that the rightmost bit has changed from a one to a zero because this bit indicates whether the binary value contains a '1'. Any number that contains a 1 as the rightmost bit is an odd number. If you don't believe me, experiment with a few values.

What you have learned

In this chapter, you've learned about how computers actually work and what it means to program. You discovered that a computer views the entire universe as patterns of ons and offs, which represent the data with which the computer is working. The computer performs data processing by transforming one pattern of bits, the input, into another pattern of bits, the output.

When human beings view the output data and act on it, the data becomes *information*. Computers are unaware of the meaning that we place on the patterns of bits that they process, which means that a computer can do "stupid" things with data.

The program tells the computer what to do with the pattern of bits. The computer itself only understands very simple instructions, which must be written in special languages, called *programming languages*. Python is a programming language, and it works as a computer program in that it takes in program instructions and then acts on them, in the same way that a human actor would perform a script.

The job of the programmer is to create a program as a sequence of instructions that describe the tasks to be performed. To create a successful solution, the programmer must not only write a good program but also make sure that the program actually does what the user wants. This means that before a programmer can write any code, she will have to make sure that she has a good understanding of exactly what is required. Talking to people and finding out what they want is a very valuable skill if you want to be a successful programmer.

To reinforce your understanding of the content, consider the following "profound questions" about computers, programs, and programming.

Would a computer “know” that it is stupid for someone to have an age of – 20?

No. As far as the computer is concerned, the age value is just a pattern of bits that represents a number. If we want a computer to reject negative ages, we must actually build that understanding into the program.

If the output from a program is settings for the fuel-injection system on a car, is the output data or information?

As soon as something starts acting on data, it becomes information. A human being is not doing anything with these values, but they will cause the speed of the engine to change, which might affect humans, so I reckon this makes this output information rather than data.

Is the computer stupid because it can’t understand English?

It’s difficult to write something in English that is completely unambiguous. Large parts of the legal profession are built on a precise interpretation of the meaning of texts and how they are applied in particular situations. Since we humans can’t agree on how to understand something, it’s not fair to call a computer stupid because it can’t do this either.

If I don’t know how to solve a problem, can I write a program to do it?

No. You can put some statements together and see what happens when they run, but this is very unlikely to produce what you want. It would be like throwing a bunch of wheels, gears, and an engine against a wall and expecting them to land and form a working car. In fact, the best way to write a program is frequently to get away from the keyboard for a while and just think about what the program is supposed to do.

Is it sensible to assume that the customer measures everything in inches?

It’s never sensible to assume anything about a project. A successful programmer must make sure that everything he is doing is built on a solid understanding. Every assumption you make increases the potential for disaster.

If the program does the wrong thing, is it my fault or the customer's fault?

It depends:

- Specification right, program wrong: programmer's fault
- Specification wrong, program right: customer's fault
- Specification wrong, program wrong: everyone's fault

3 Python program structure



What you will learn

In this chapter, you'll build your first Python program. You'll discover that a Python program is just a sequence of statements obeyed in order by a computer. You'll create and save Python program files for later use. You'll also discover how to add new capabilities to Python programs by importing and using helpful resources, including our first `snap` functions that you can use to create impressive programs.

[Write your first Python program](#)

[Use Python libraries](#)

[Python comments](#)

[Run Python from the desktop](#)

[Adding some snaps](#)

[What you have learned](#)

Write your first Python program

So far, we've used the Python Shell part of IDLE to enter Python program commands. This is a great way to experiment with the Python language, but we discovered that when we want to repeat an action, we must retype the commands again. What we really want to do is create a Python program. A program is a sequence of actions that are performed in order. You can think of a program as like a script you would give an actor to perform. The actor reads each line and then moves on to the next one. Likewise, Python takes each Python instruction, checks to ensure that it's sensible, performs it, and then moves on to the next instruction.

Python programs are stored in files on your computer. There's nothing special about a Python program file; it's just a text file that contains program instructions that Python understands.

Run Python programs using IDLE

From within IDLE, we can open a new window where we can work on Python programs and store them in a file on our computer. We then ask IDLE to run the program so that we can see whether the program works properly. This is exactly what professional developers do when they write programs.



MAKE SOMETHING HAPPEN

Run your first Python program

To create a Python program file, first use IDLE to open a new editing Window on the desktop. Click the **File** menu at the top line of the IDLE window and select **New File** from the list that appears, as shown in **Figure 3-1**. Alternatively, you can press **Ctrl+N**. Note that I'm performing these actions on my PC running Windows 10. You might see slightly different looking screens, but the content will generally be the same.

As shown in **Figure 3-2**, a second window appears on the screen, with the title "Untitled."

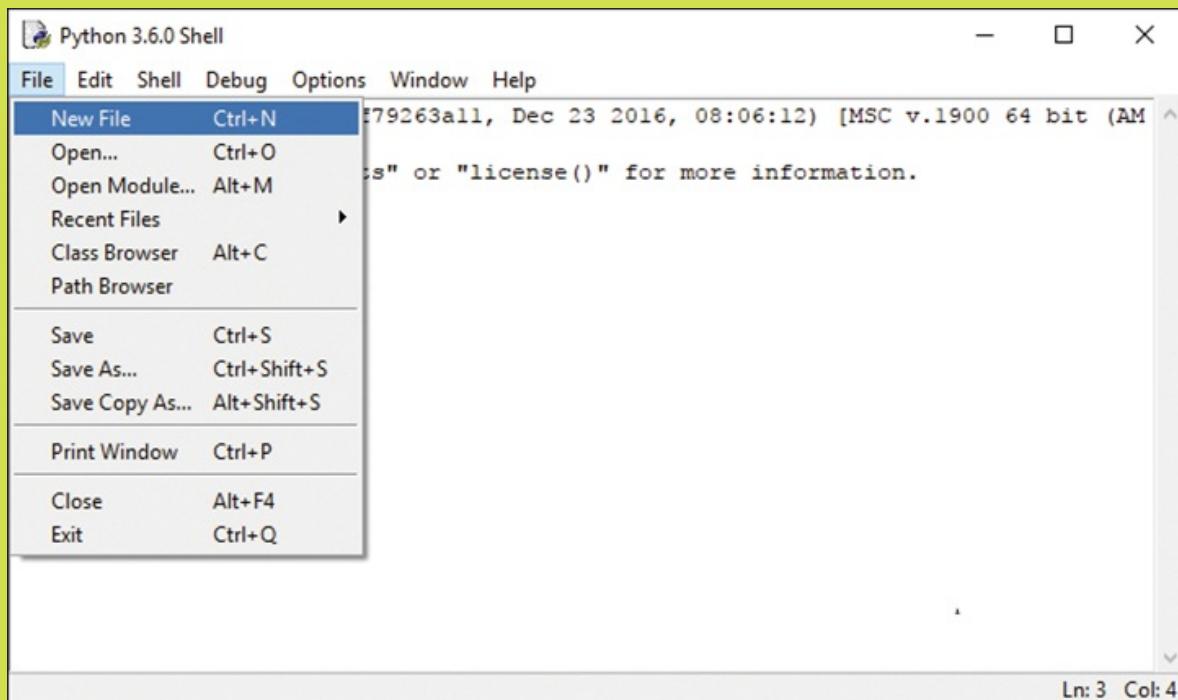


Figure 3-1 IDLE New File

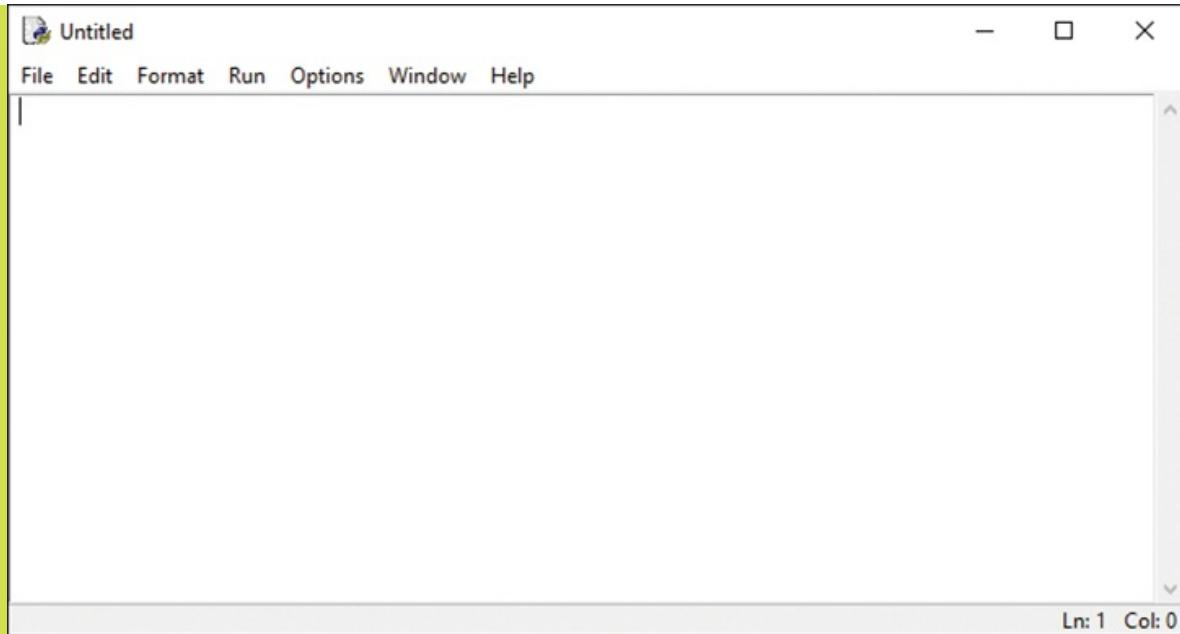
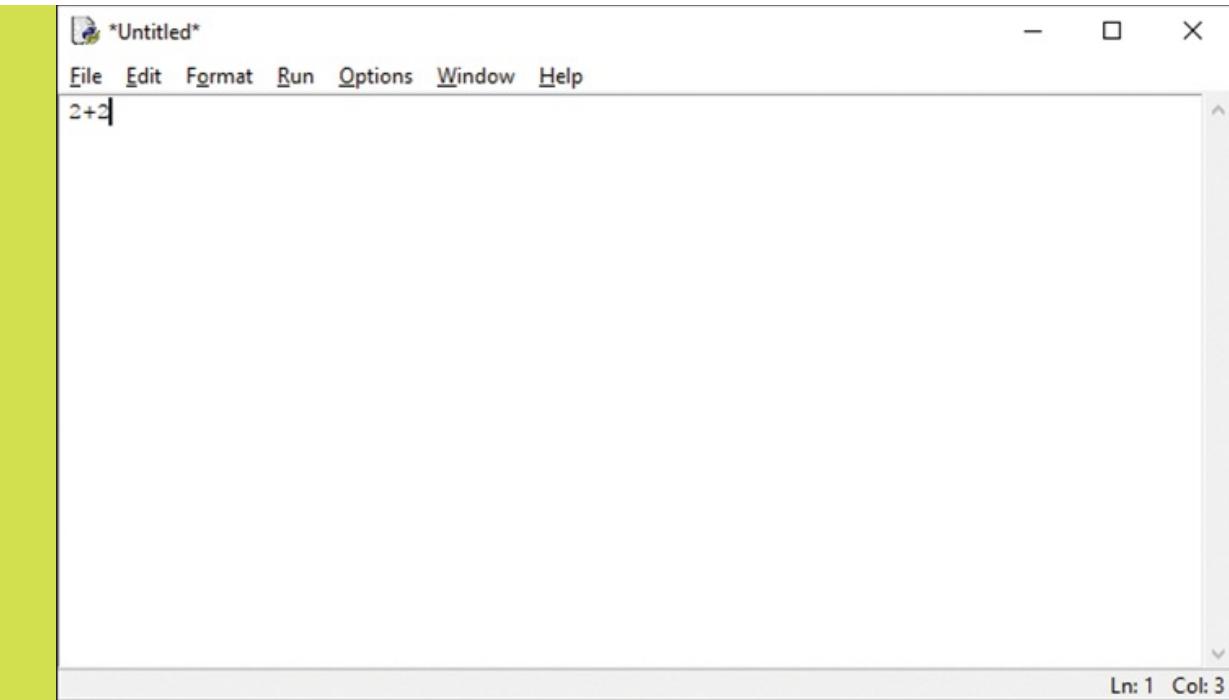


Figure 3-2 IDLE Untitled edit window

This window is not the same as the Python Command Shell. It lacks the >>> prompt where you type Python commands. Python statements typed into this new window are not executed when you press Enter. The statements are held as part of a program. You can treat this window like any other text editor you've used. Think of it as a word processor for programs. The editor will color the various elements in the program in the same way as the Python Shell does. It will also provide some pop-up help messages as you type in your programs.

Now we can type in the same Python expression we used at the beginning of our programming journey ([Figure 3-3](#)).



The screenshot shows a window titled '*Untitled*' with a menu bar containing File, Edit, Format, Run, Options, Window, and Help. The main text area contains the expression '2+2'. In the bottom right corner of the window, there is a status bar displaying 'Ln: 1 Col: 3'.

Figure 3-3 An expression in a program

This program doesn't do much, but it should print out the result of the calculation. Now we need to run it. Click the **Run** menu option and select **Run Module** (**Figure 3-4**).

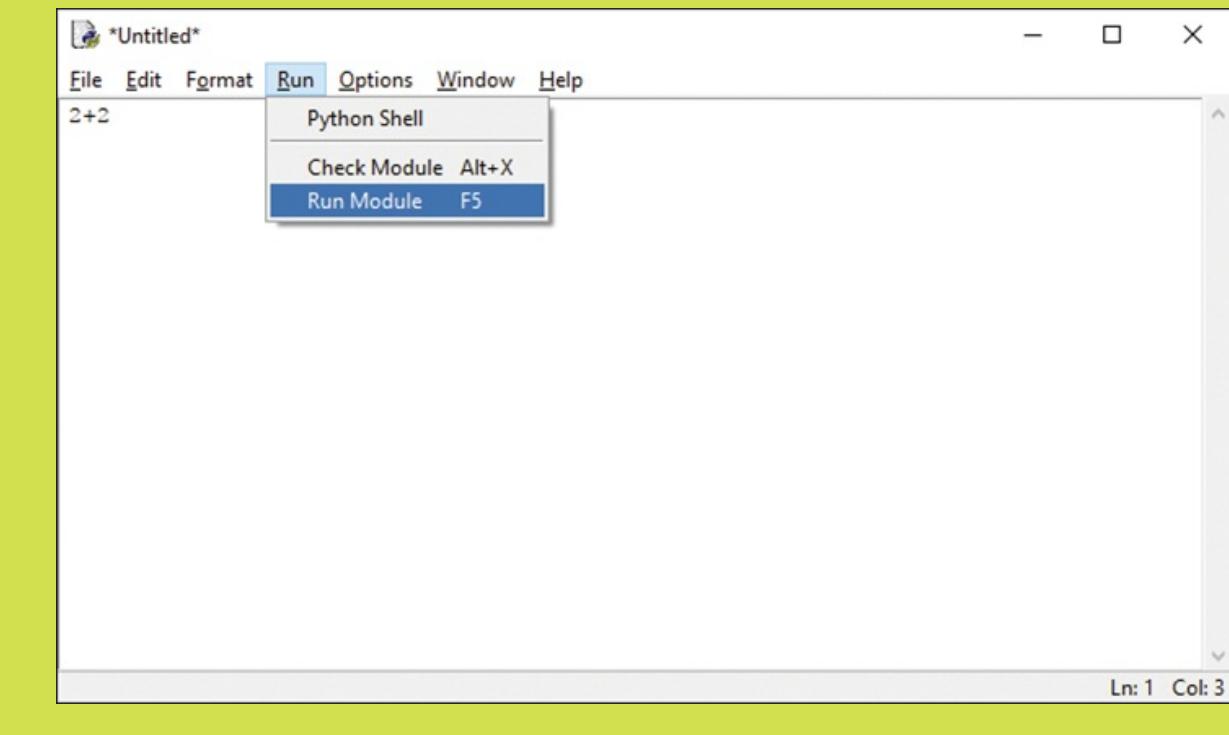


Figure 3-4 Run a program

The first time we run an Untitled program, Python asks if we want to save it in a file (**Figure 3-5**).

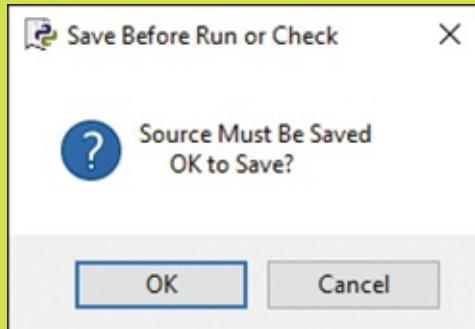


Figure 3-5 OK to save

Click **OK** to open a file save menu (**Figure 3-6**):

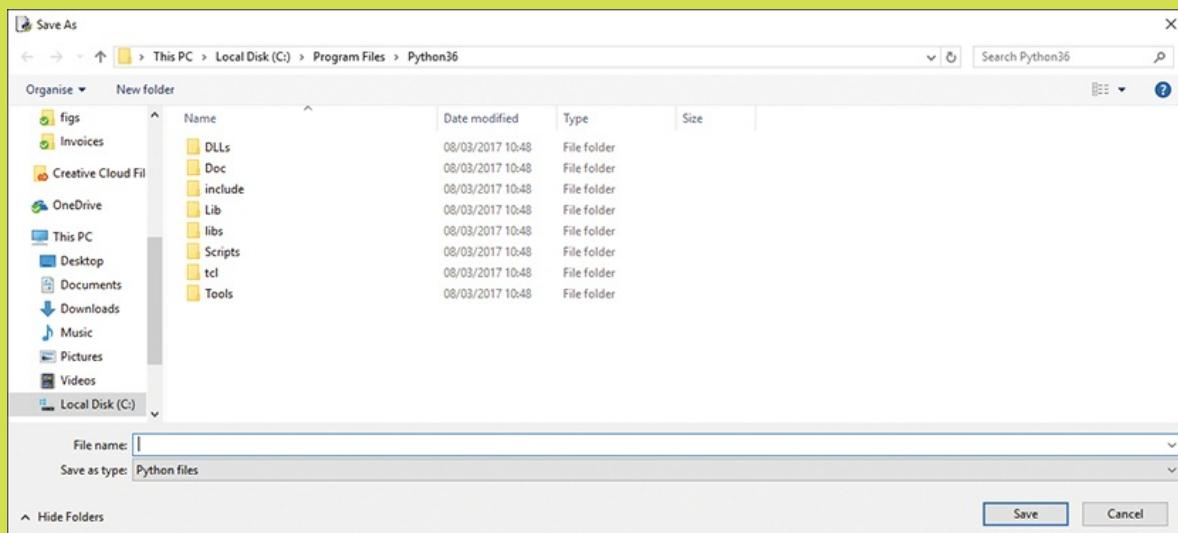


Figure 3-6 Default file save location

IDLE is not offering to save your program files in a practical location. You shouldn't form the habit of saving Python programs in the Program Files area of your computer (even assuming that your computer will let you). Instead, I suggest that you navigate to your Documents folder and create a Python folder there (**Figure 3-7**).

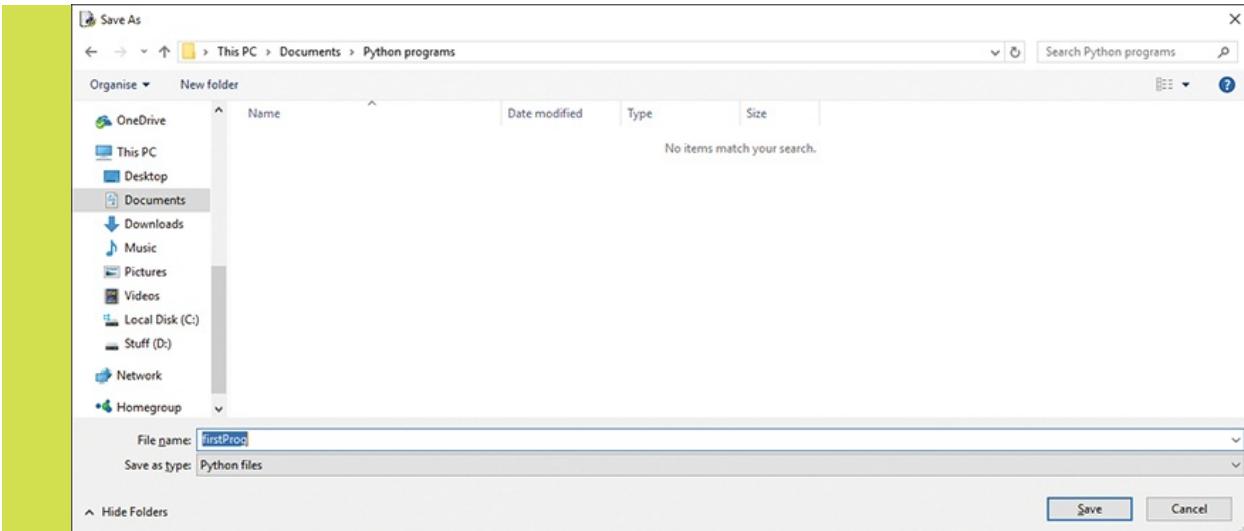


Figure 3-7 Saving your first program

Give the program a sensible name (I called mine **firstProg**) and click **Save**. IDLE will add the Python file extension (.py) to the file name when it saves it. This is an exciting moment. As soon as you click **Save**, your first program will start running, and the results will be displayed in the the Python Shell part of IDLE. **Figure 3-8** shows the output from our program.

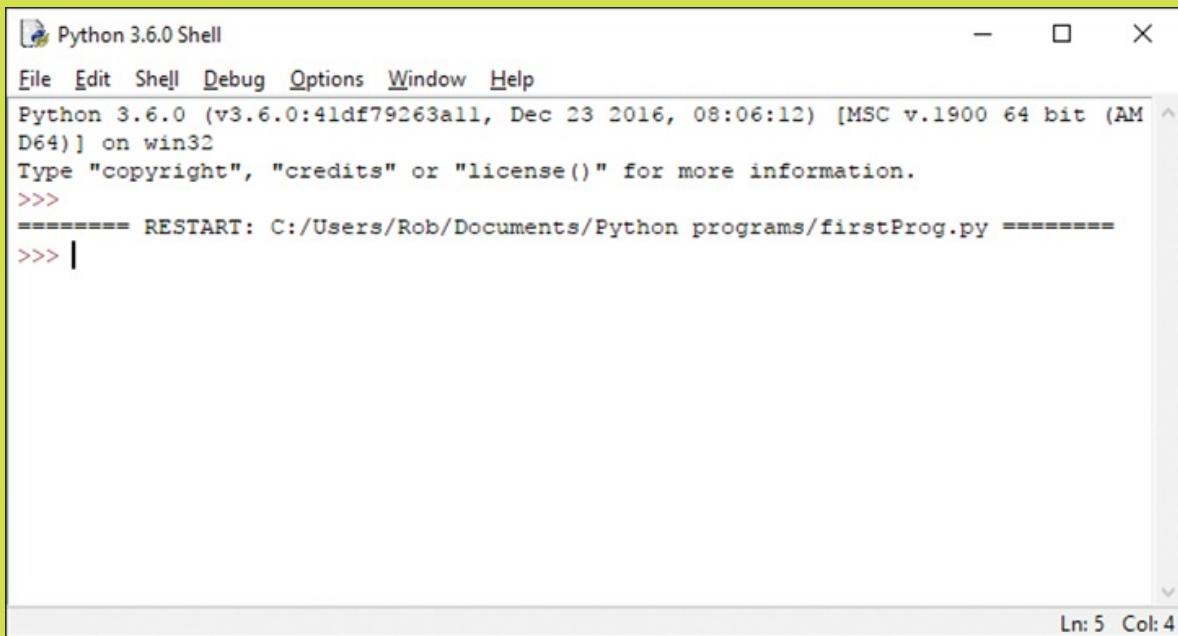


Figure 3-8 Running our first program

The program has definitely run (you see the full name of the file being used), but it doesn't seem to have printed anything, which is unfortunate. We expected to see the value **4** printed (the result of calculating **2+2**), but we see nothing. This does not bode well for future programs. What's happening here?

Get program output using the `print` function

It turns out that our tiny program is actually running perfectly. It's just that we don't yet understand how a Python program communicates with the user. Recall that the Python system is a pure data processor. A Python program goes in one end and the results of running the program come out of the other. A Python program can be as simple as the single statement **2+2**, or it may contain many thousands of Python statements.

The output from a program comprising **2+2** is the value **4** (as we have seen). The output from a larger program usually indicates whether or not the program worked properly.

The Python Shell envelops the Python system (that's why it's called a shell) and lets us type in Python and view the results of the program. But a program that runs and generates a single result is not usually what we want. We want a program that will have a conversation with us as it runs. **Figure 3-9** shows the arrangement we want.

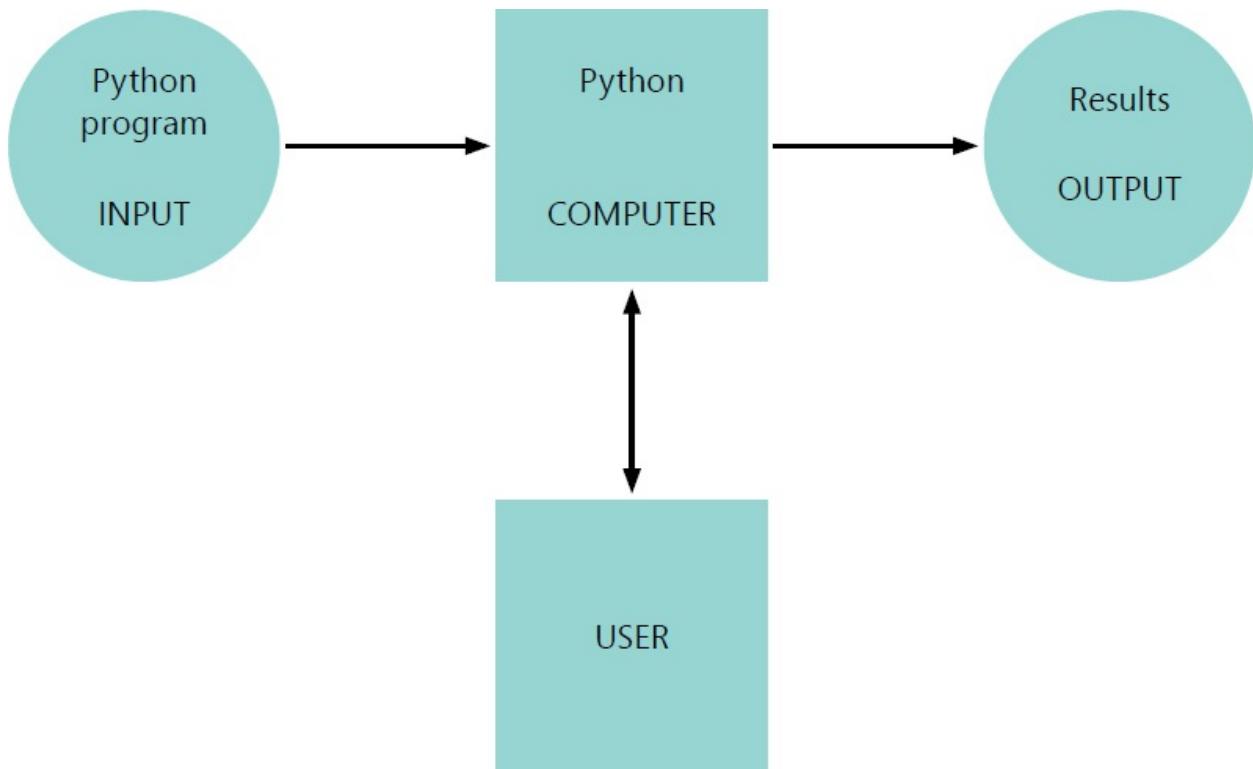


Figure 3-9 Python as a data processor with input/output

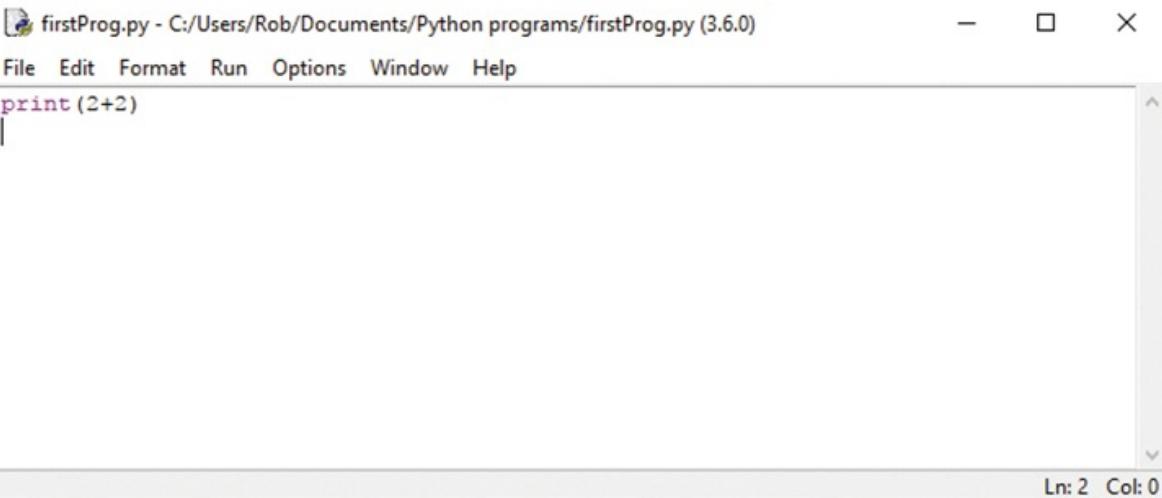
When the program runs, it should send messages to the user and receive information from the user. Python provides functions that can interact with the user, called `print` and `input`. We'll take a look at the `input` function in the next chapter. Let's start with a look at how to use the `print` function.



MAKE SOMETHING HAPPEN

Work with `print` in a program

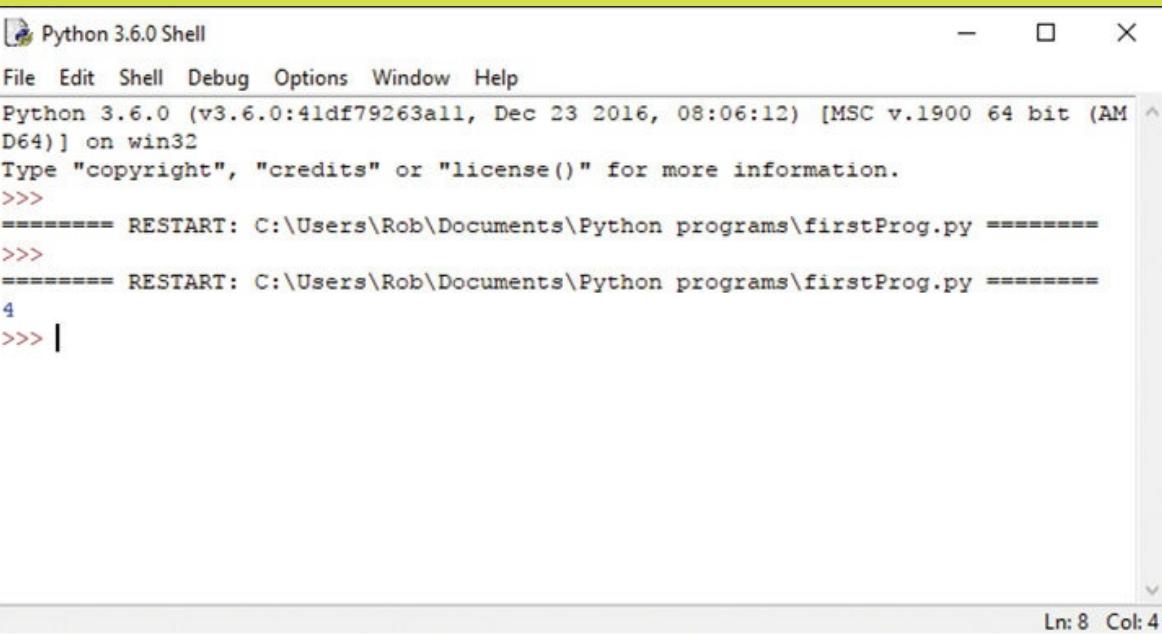
You first saw functions in the previous chapter, where you used `ord` and `chr` to work with the character codes for text. You give the `print` function an expression to print out for the user to read. We can use it to allow our program to print messages to the user (**Figure 3-10**). The messages are displayed by the Python Shell part of IDLE.



A screenshot of a Windows-style application window titled "firstProg.py - C:/Users/Rob/Documents/Python programs/firstProg.py (3.6.0)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The main text area contains the Python code: "print(2+2)". In the bottom right corner, there are status indicators "Ln: 2 Col: 0".

Figure 3-10 Using print in a program

Add the `print` function as shown and run the program again. You'll be asked again if you want to save the program before you run it. Select **Yes** and look at the output (**Figure 3-11**).



A screenshot of the Python 3.6.0 Shell window titled "Python 3.6.0 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell displays the Python version information: "Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AM D64)] on win32". It then shows the command prompt "Type >copyright", "credits" or "license()" for more information. The user enters "4" followed by a carriage return. The response is "===== RESTART: C:\Users\Rob\Documents\Python programs\firstProg.py =====". The status indicators in the bottom right corner are "Ln: 8 Col: 4".

Figure 3-11 Printing a message from the program

Finally, we get the value `4` printed out. If we want to print out more messages, we can just add more calls to the `print` function. (Note that I'm just showing the program code and output now, rather than whole

screenshots from the IDLE editor).

```
print('The answer is: ')
print(2+2)
```

This would print out two lines of text:

[Click here to view code image](#)

```
===== RESTART:
C:\Users\Rob\Documents\Python
programs\firstProg.py =====
The answer is:
4
```

If we want to print out several things on one line, we can give the `print` function a list of arguments.

Each item in the list is separated by a comma character:

[Click here to view code image](#)

```
print('The answer is:', 2+2)
```

This would print out just one line of text:

[Click here to view code image](#)

```
===== RESTART:
C:\Users\Rob\Documents\Python
programs\firstProg.py ===== The answer is:
4
```

The values of the arguments are printed out one after the other. Note

that the `print` function automatically inserts a space between each of the items it prints.



MAKE SOMETHING HAPPEN

Eliminate save requests

By now, the incessant requests to save after each edit should be driving you slightly mad. You can stop IDLE from repeatedly asking you the question by changing a setting.

On a Windows PC, click the **Options** menu and select **Configure IDLE**.

On a Mac, you can produce the same dialog by selecting **Preferences** from the IDLE menu.

Then move to the **General** tab in the dialog that appears and select the **No Prompt** option in the **Autosave Preferences**, as you can see in [Figure 3-12](#).

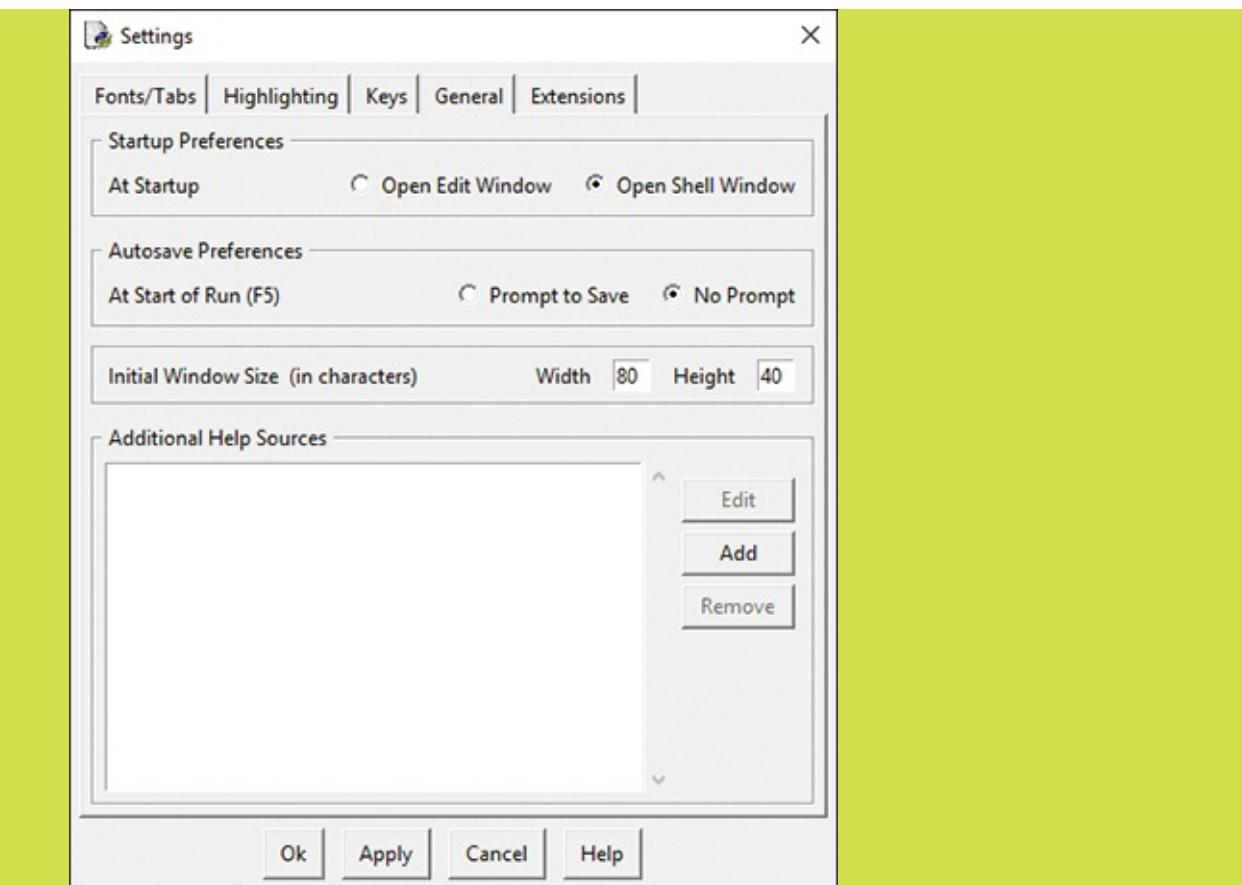


Figure 3-12 Save options

You can use this menu to set many other preferences, including the size of the text in the IDLE display. This can be very useful if you are creating a program and want your program output to be visible from a distance—for example, if you were creating a party game, as we'll do later in this chapter.



WHAT COULD GO WRONG

Broken programs

Python performs the same error checking on programs as it did when you entered statements using the the Python Shell part of IDLE. Let's

look at a couple ways that our simple program could go wrong. Consider the following couple lines of code:

```
print('The answer is: ')
print(2+2)
```

This code looks correct at first glance, but it contains a serious error. The closing parenthesis that should be at the end of the top line has been replaced with a closing brace instead. When I try to run the program, I get an error, as shown in **Figure 3-13**.

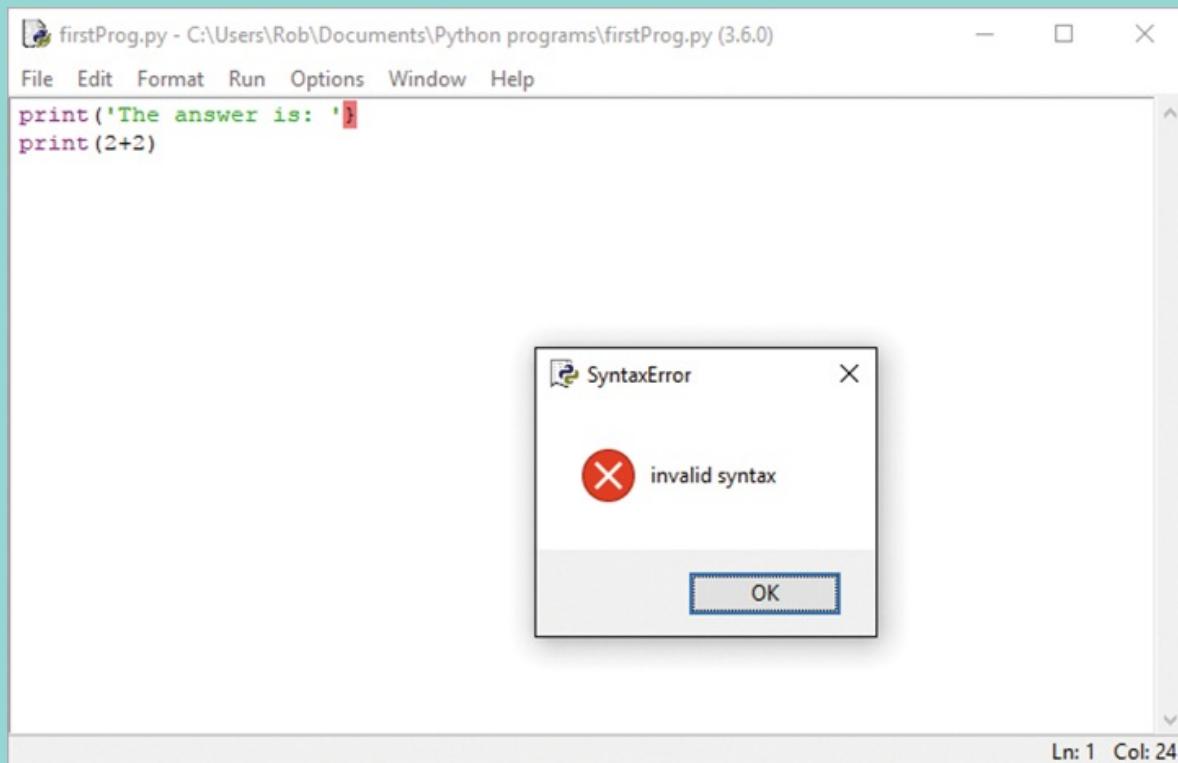


Figure 3-13 Invalid syntax error

The editor has helpfully highlighted the incorrect closing brace. The IDLE editor actually checks the syntax of your program as you type it in. When I typed the incorrect brace above, my PC produced a warning sound to indicate that I'd done something silly. As you type in your programs, you'll notice that the IDLE editor tries to help you get them right. When you type a closing parenthesis, you'll notice that the entire

sequence of characters surrounded by parentheses is highlighted. This is very useful if you nest one set of parentheses inside another. You can use this highlighting feature to see precisely which elements are enclosed in a particular pair of parentheses.

If you want to check the syntax of your program without running it, you can use the **Check Module** option from the **Run** menu in the IDLE editor. This option checks that the code is correct but doesn't actually try to make it run.

Here's another example of problematic code:

```
print('The answer is: ')
Print(2+2)
```

This code looks correct at first glance, but it also contains an error. In this case, I've misspelled the second `print` function call, using the name `Print` by mistake. Python is case sensitive. It regards the words `Print` and `print` as different. This time when I run the program, the error is not displayed in the editor but in the Python Shell part of IDLE.

[Click here to view code image](#)

```
===== RESTART:
C:\Users\Rob\Documents\Python
programs\firstProg.py =====
The answer is:
Traceback (most recent call last):
  File "C:\Users\Rob\Documents\Python
programs\firstProg.py", line 2, in <module>
    Print(2+2)
NameError: name 'Print' is not defined
```

The error is only detected when the program runs. As you can see, the first call to `print` worked correctly, with the message '`The answer is:` ' displayed. But on the next line, the attempt to use a function

called `Print` failed with an error message because that function does not exist.

This is called a run time error. The Python syntax checker detects mismatched parentheses, but it doesn't verify that all function calls have matching functions.

This means that you have to get used to the fact that even if your program doesn't contain syntax errors, it might still not work.

The `print` function and Python versions

There are currently two versions of Python in popular use. We discussed this at the very beginning of the book. Version 2.7 is the “old school” version, which is interesting because there are many libraries of code written using this version. Version 3.n (where n is a value larger than 2) is the “new kid on the block,” with new features and with some features “tidied up.” The older version of Python supports a variety of the print behavior that works without parentheses around the items to print:

```
print 'hello from Python'
```

I’m telling you about this, not because I want you to write your programs this way, but because you might encounter older Python programs that look like this. You might also have problems if you use parentheses in print statements in a program written for the older version of Python:

```
print('The answer is:', 2+2)
```

This statement, which we know should print, “The answer is: 4”, will print the following in Python 2.7:

```
('The answer is:', 4)
```

So, if your Python program doesn't seem to be printing correctly, check which version of Python you're using.

This alternate behavior is unfortunate and is one of the things about Python that I find most confusing. But, in the same way that we must live with keyboards designed to slow down typing, we must also live with these inconsistencies between Python versions.

Use Python libraries

We've already seen some built-in functions provided by Python. We have used the `ord` and `print` functions, among others. Python also provides a vast number of function libraries that we can add to our programs. Some libraries are provided as part of a Python installation on a computer; others can be loaded from the Internet. A Python program can make use of multiple libraries simultaneously.

The `random` library

We can start by exploring the `random` library, which provides a source of random numbers that we can use in our programs. Adding randomness to programs is a fun thing to do, and random numbers are the basis of many games. We need to tell Python that we want to use the functions in a library in our program by using the `import` command.

```
import random
```

The `import` command is followed by the name of the library we wish to import. This command will import the random library. A program can now use functions contained in the library. The random library contains lots of functions, including one called `randint`, which generates a random integer in a particular range. A program can tell `randint` the range by supplying two arguments (remember that an argument is the name for some data that we're passing into a function call) that give the lowest and highest numbers to be produced. For example, we could use a lowest value of 1 and a highest value of 6 to simulate the throw of a die.

You can see what a library function call looks like in **Figure 3-14**. The name of the function is preceded by the name of the library to look in. The library name and the function name are separated by a period (a full stop).

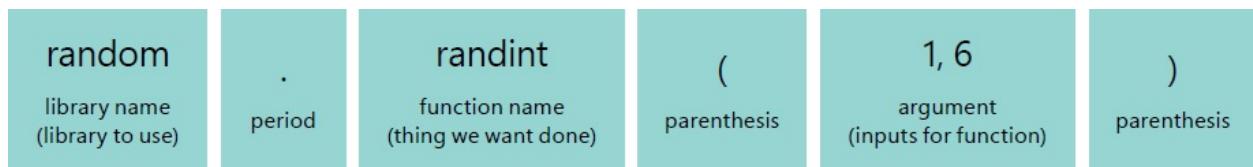


Figure 3-14 Anatomy of a library function call



MAKE SOMETHING HAPPEN

Investigate the `random` library

The `import` commands can be used in the Python Shell as well as inside a program. Open up the Python Shell part of IDLE to get the `>>>` prompt.

Enter the call of `randint`.

```
>>> random.randint(1, 6)
```

Will this work?

[Click here to view code image](#)

```
>>> random.randint(1, 6)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    random.randint(1, 6)
NameError: name 'random' is not defined
```

It would seem not. The Python Shell part of IDLE is complaining that the word `random` is not defined. So, let's import the random library:

```
>>> import random
```

The statement doesn't produce any output; we just get the command prompt back. Nevertheless, we've successfully imported the `random` library into our program. Note, however, that if we had misspelled the name of the library (remember that Python is case-sensitive), we would have seen an error:

[Click here to view code image](#)

```
>>> import Random
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import Random
ModuleNotFoundError: No module named 'Random'
```

Now that we have the `random` library installed successfully, we can begin using it. Type the library name, followed by a period and then the `randint` function call. Once you have typed the opening parenthesis (which marks the beginning of the function's arguments), you should notice something interesting, as shown in **Figure 3-15**.

The screenshot shows a Python 3.6.0 Shell window. The user has typed `>>> random.randint(1,6)`. A pop-up message box displays the function's documentation: `(a, b)` and `Return random integer in range [a, b], including both end points.`. The shell also shows a traceback for a previous attempt to import the Random module.

```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> random.randint(1,6)
Traceback (most recent call last):
  File "<pyshell1#0>", line 1, in <module>
    random.randint(1,6)
NameError: name 'random' is not defined
>>> import random
>>> import Random
Traceback (most recent call last):
  File "<pyshell1#2>", line 1, in <module>
    import Random
ModuleNotFoundError: No module named 'Random'
>>> random.randint(
        (a, b)
        Return random integer in range [a, b], including both end points.

Ln: 14 Col: 19
```

Figure 3-15 A pop-up help message

The Python Shell can access information about each of the library functions and will pop up a helpful message about the function's arguments. In this case, it's telling us that the function returns a random integer in the range a to b, including both end points. There are two arguments, a and b. You don't have to do anything to remove this pop-up message; just type in the rest of the statement and press Enter. Remember that the Python Shell just works out results and returns them, so you should see the result of the call to `randint`.

```
>>> random.randint(1,6)
4
```

You might see the value `4` when you run the command. Obviously, there is a one in six chance for each of the six options to appear. You can change the bounds of the lower and upper values of the random number to any that you like, but you must make sure that the lower bound is lower than the upper bound; otherwise, you'll get an error.

Now we can make a program that will simulate the throw of a single die. Click **File**, **New File** to create a new Python file. Enter the following lines into the file.

[Click here to view code image](#)

```
import random
print('You have rolled: ', random.randint(1,
6))
```

Run the program and save it in a file called **throwDie**. This Python program will print a number in the range 1 to 6, inclusively. The program works because the **print** function can print numbers, and the **randint** function returns a random number within the range of its arguments.

The first time I ran this program, it printed “You have rolled: 4”. The second time I ran the program, it printed “You have rolled: 2.”

The **time** library

Another useful Python library is the time library. In [Chapter 5](#), we’ll use the time library to get the date and time from the PC’s clock, but for now, I want to focus on just one function in the library: the **sleep** function. This can be used to make a Python program sleep for a particular amount of time.

I use the **sleep** function in my programs to give the user the impression that the computer is “thinking” about my problem. The **sleep** function can also be used to give the user time to read what the program displays.

It’s important to make it clear that calling the **sleep** function doesn’t actually cause your entire computer to stop. It simply tells the operating system (whether that is Windows, macOS, or Linux) to pause this particular Python program for a duration of time. The operating system is in charge of deciding which programs are active at any given time, and making a program sleep just means that the program will not be running. All the other programs in the computer will be active as normal.

The **sleep** function is given a single argument, which is the number of seconds that the program should pause.

[Click here to view code image](#)

```
import time
print('I will need to think about that..')
time.sleep(5)
print('The answer is: 42')
```

This program will print out the first message, pause majestically for 5 seconds, and then print out the second message. The `sleep` function uses the system clock in the computer, which is very accurate. You can make the program sleep for a very long time if you wish.



MAKE SOMETHING HAPPEN

Make an egg timer

You can use the `sleep` function to make an egg timer program. The program should allow the user to time a 5-minute boiled egg. You can do this by modifying the previous program which paused for 5 seconds.

For extra style points, you could make the program print, “Nearly cooked, get your spoon ready,” 30 seconds before the 5-minute deadline.

You could even expand this into an interactive recipe program that describes the steps to be performed at each point in the recipe and then pauses until the next step is performed.

Python comments

It's very important that you write programs in a way that makes it easy for people reading your code to understand what's going on. You don't write

comments for the computer; you write comments for someone reading your program. You can also use comments to indicate the particular version of the program, when it was last modified and why, and the name of the programmer who wrote it—even if it was you.

A comment begins at the character # (hash or number sign) and extends to the end of that line, like this:

```
# Egg timer program 1.0 by Rob Miles
import time
print('Put the egg in boiling water now')
time.sleep(300)
print('Take the egg out now')
```

A comment on a single line

As soon as the Python system encounters the beginning of a comment (the # character), it ignores everything until the end of that line. In IDLE, comments are displayed in red to make them stand out. You can add comments to the end of a statement.

[Click here to view code image](#)

```
time.sleep(300) # sleep while the egg cooks (300
seconds, or 5 minutes)
```

The above comment is a good one because it explains why the program is performing the sleep.

[Click here to view code image](#)

```
print('Put the egg in boiling water now') # print
put the egg in boiling water now
```

The preceding comment is less useful. Anyone who understands Python (and indeed most people) will be quite able to understand what the above statement does without reading the comment.

We'll discuss comments in more detail later in the book. For now, form the habit

of giving every program a title and adding comments to statements to explain their use.

Code samples and comments

Now that we've started creating programs, we can start to use the code samples provided with this book. Each chapter has a number of associated code samples. I'll identify the code samples by adding a comment at the top of the text:

[Click here to view code image](#)

```
# EG3-01 Throw a single die
import random
print('You have rolled: ' + random.randint(1, 6))
```

You can find this program in the code sample folder for [Chapter 3](#). It has the file name **EG3-01 Throw a single die**.

Run Python from the desktop

Until now, we've been running our programs from inside the IDLE environment. However, it's also possible to run Python programs directly from your computer's desktop. Your operating system can use the file extension mechanism to identify Python programs and send them to the Python system to be executed.

A file extension is the means by which an operating system determines the application associated with a given file. It is expressed as a number of letters appended to the end of a file name. Microsoft Word documents have the file extension .docx, text documents have the extension .txt. Python programs are stored with the file extension .py.

When you save a Python program from IDLE, the file extension .py is automatically added to the file name. When you select the Python program file from the desktop, the operating system runs the Python system on that file.

Figure 3.16 shows three of our sample programs in a folder on the Windows 10 operating system.

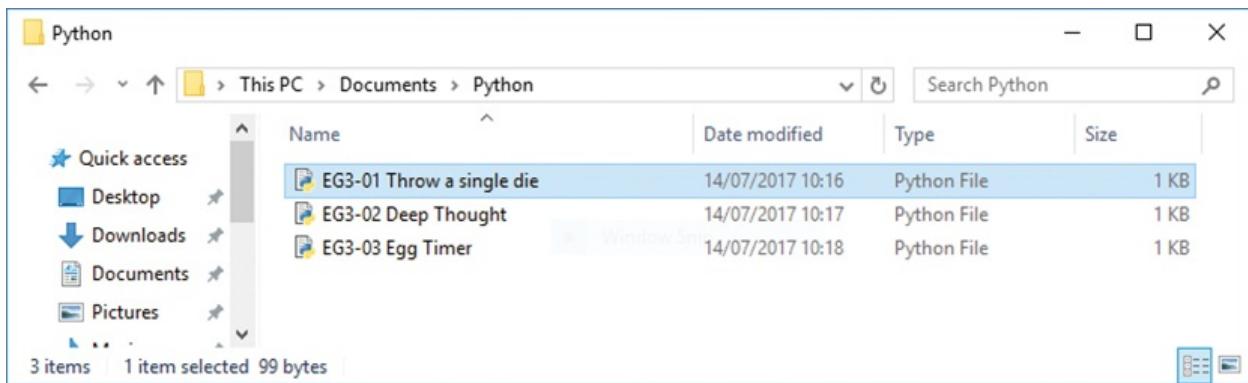


Figure 3-16 Python programs in a folder

If we double-click any of the files, the Python system starts to run the program. You can see that the folder indicates that the files are of the type “Python File,” and the icon for each file is the icon for a Python source file.

If you have problems with Python starting correctly, you can change the file associations in Windows 10 to select the Python system. Right-click the file name and select **Open with** from the context menu that appears. As you can see in **Figure 3-17**, you can select the application that you want to associate with the Python source files.

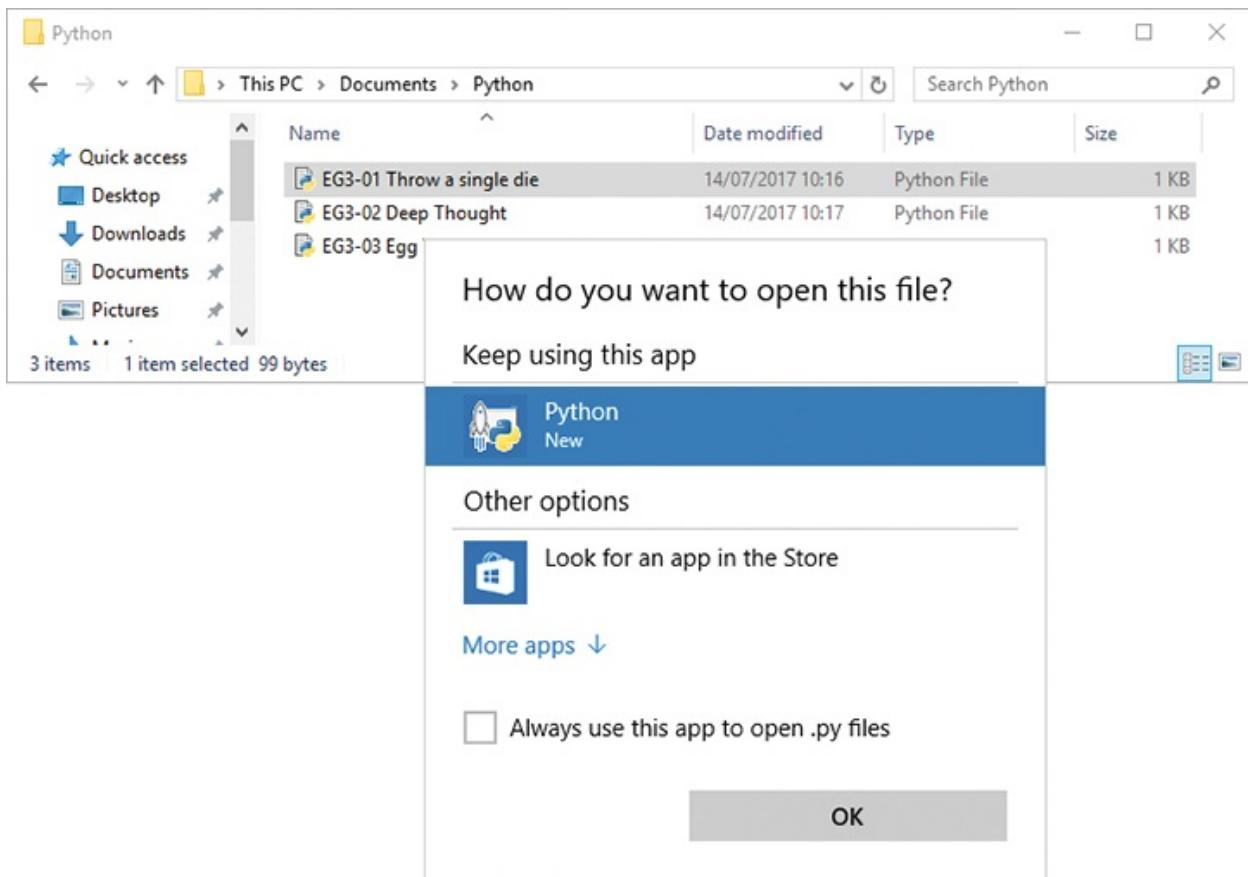


Figure 3-17 Managing file associations in Windows 10

I can select a suggested application or select **More apps** to find the Python system on my machine. However, if you've followed the installation instructions at the beginning of the book, it's unlikely that you will have to do this.

Delay the end of the program

One problem that you might have is that the program runs and then finishes before you can read the results. For now, you can solve this problem by adding a sleep statement as the last statement in the program. In the next chapter, you'll find out how to make a program wait for input from the user before continuing.

Adding some snaps

Printing messages in the Python Shell part of IDLE is all very well, but it would be great if we could display pictures and play sounds from our Python programs. In [Chapter 16](#), we'll learn to do this and more when we discover how to make

games using the pygame framework.

But before we do that, we'll use the pygame framework to support a library of functions that I've written to make our programs more interesting. I've called the library "snaps" because it lets you get things done "in a snap." Every now and then we'll learn about some new snaps functions you can use. The snaps we'll use in this chapter make it really easy to add text, images, and sounds to your programs.

Adding the Pygame library

To start, we need to add the Pygame library to our Python installation. Adding libraries to Python is very easy; you use the **pip** program to do this. The pip program is supplied with your Python installation. It fetches library files and adds them to your Python installation.

If you're using Windows 10, you run the pip program from the Windows 10 PowerShell command prompt. This is like the Python Shell part of IDLE, except that the commands you type can be used to start other windows programs. You can open the PowerShell command prompt by right-clicking the Windows **Start** button at the bottom left of your screen and selecting **Windows PowerShell** from the menu that appears.

This should cause a new window to open on your desktop. To install Pygame, issue the following command:

[Click here to view code image](#)

```
py -m pip install pygame --user
```

The pip program displays a progress bar as it works, followed by a success message. **Figure 3-18** shows a successful installation of pygame. You only have to install pygame once. It is made part of your Python installation.

```
Windows PowerShell
PS C:\Users\Rob> py -m pip install pygame --user
Collecting pygame
  Downloading pygame-1.9.3-cp36-cp36m-win_amd64.whl (4.2MB)
    100% |██████████| 4.2MB 376kB/s
Installing collected packages: pygame
Successfully installed pygame-1.9.3
PS C:\Users\Rob>
```

Figure 3-18 Installing pygame

The pip command to install pygame on Apple Mac or Linux machines is slightly different. You run this command from a Terminal window:

[Click here to view code image](#)

```
python3 -m pip install pygame --user
```

Further installation help can be found at
<http://www.pygame.org/wiki/GettingStarted>

Snaps functions

The snaps library is a Python program file held in the same folder as the example programs for each chapter. The file contains all the functions we'll use. If you want to use snaps in another folder, simply copy the file snaps.py into the folder.

Displaying text

The snaps `display_message` function takes a string of text and displays the string in a new window on your display, as shown in **Figure 3-19**.

```
# EG3-04 hello world

import snaps import *
snaps.display_message('hello world')
```

Import all the functions in the snaps library

Display a message on the screen

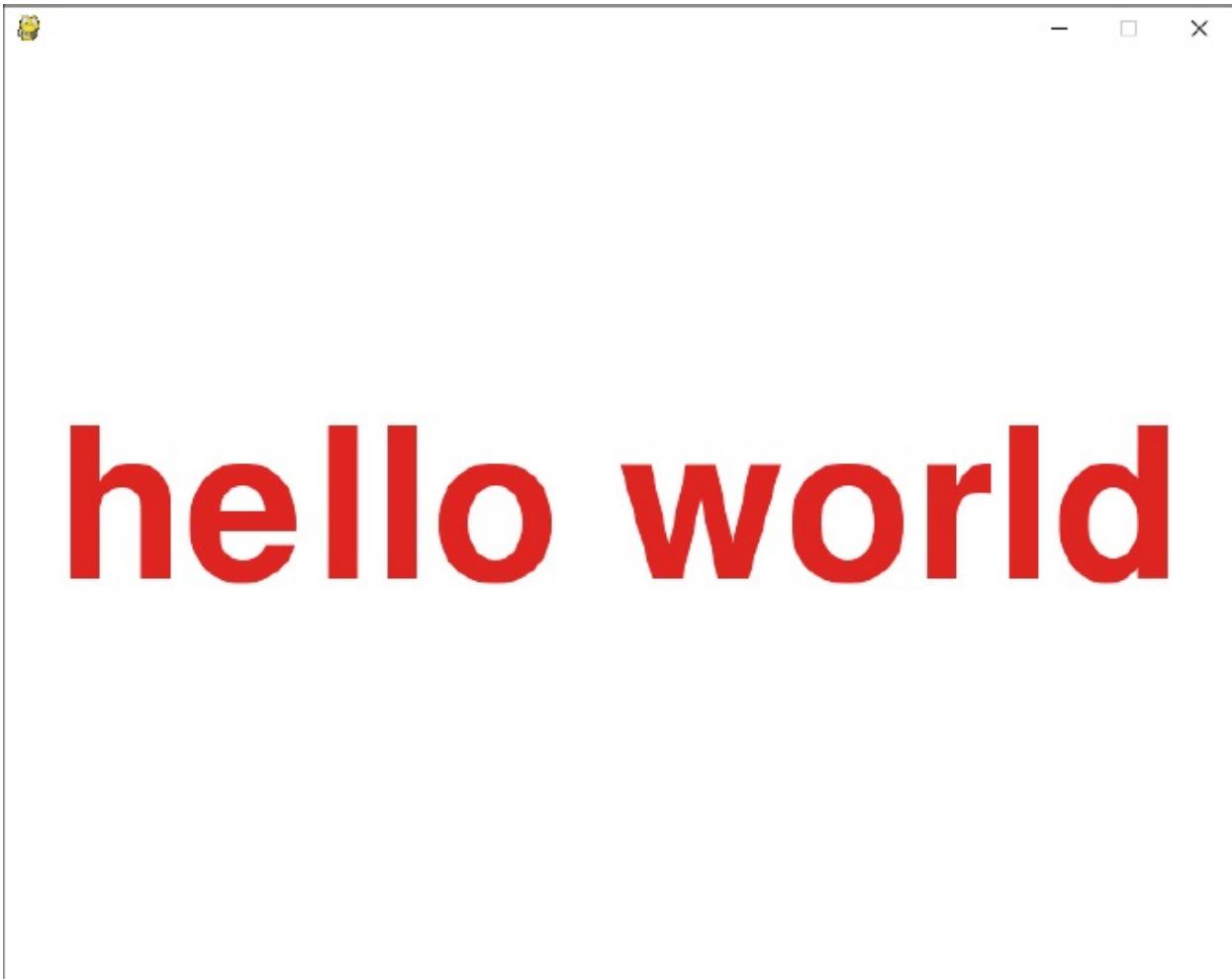


Figure 3-19 Displaying “hello world”

Python functions support optional arguments, which are values you can use to provide additional information about the way you want a function to work. We’ll explore these in detail in [Chapter 7](#). The `display_message` function provides optional arguments you can use to select the size, color, and alignment of the text you want to display:

[Click here to view code image](#)

```
snapshots.display_message('This is smaller text in  
green on the top left',  
                           color=  
(0, 255, 0), size=50, horiz='left', vert='top')
```

The text color is expressed as three values: one for red, one for green, and a third value for blue. The largest possible value for any color is 255. Increase the size value to make your text larger. The `display_message` function will generate an error if the text won't fit on the display. You can use the attribute `horiz` to align text to the left, right, or center of the screen. You can use the attribute `vert` to align text to the top, bottom, or center of the screen. The best way to find out how to use these arguments is to experiment with the function and see what happens.

Displaying images

The `display_image` function takes the name of an image file and displays that image on the screen. The image is scaled to fit the screen. The image file must be in the same folder as the program.

[Click here to view code image](#)

```
# EG3-05 housemartins

import snaps

snaps.display_image('Housemartins.jpg')

snaps.display_message('Hull Rocks',color=
(255,255,255), vert='top')
```

The image file can be either a png or a jpeg image. If the file cannot be found, the function will generate an error. You can display text on top of an image (as seen in [Figure 3-20](#)) if you call the `display_message` function after the call of `display_image`.



Figure 3-20 Displaying images

Making sounds

The **play_sound** function takes a file in the wav audio format and plays it through the speaker on your computer.

[Click here to view code image](#)

```
# EG3-06 Ding  
  
import snaps  
  
snaps.play_sound('ding.wav')
```

The audio file must be in the same folder as the program. The above program

plays a ding sample that I created using a wooden spoon and a pan half full of water. This sounds good to me, and might be useful as an alarm sound for an egg timer.

If you want to work with audio files to prepare them for use in your programs, I strongly recommend the Audacity program, which you can download for free from www.audacityteam.org/.

Using snaps in your programs

You can use the snaps functions to make an egg timer and some big-screen versions of the programs that we've already written. I've supplied some background graphics that you might find useful in the code sample folder for this chapter, although it's much more fun to draw your own.



MAKE SOMETHING HAPPEN

Make the “High–Low” and “Nerves of Steel” party games

At the beginning of this book, I said that programming is as easy (or difficult) as organizing a party. With that in mind, here are a couple ideas for Python programs that you can use for entertainment at your next party.

You can use the random number generator and the sleep function to make a high–low party game. The game works like this:

1. The program displays a number between 1 and 10, inclusively.
2. The program then sleeps for 20 seconds. While the program is asleep, the players are invited to decide whether the next number will be higher or lower than the number just printed. Players who choose “high” stand on the right. Players who choose “low” stand on the left.

3. The program then displays a second number between 1 and 10, and anyone who was wrong is eliminated from the game. The program is then re-run with the players that are left until you have a winner.

This game can get very tactical, with players taking a chance on an unlikely number just so that they will be one of the people to go forward to the next round.

Another use for the random number generator and the sleep function is to make a “Nerves of Steel” game. This game works like this:

1. The program displays “Players stand.”
2. The program sleeps for a random time between 5 and 20 seconds. While the program is sleeping, players can sit down. Keep track of the last person to sit down.
3. The program displays “Last to sit down wins.” Players still standing are eliminated, and the winner is the last person to sit down.

What you have learned

In this chapter, you’ve learned the difference between executing Python statements using the Python Shell part of IDLE and creating complete Python programs. You’ve discovered that a program is a sequence of Python statements stored in a file on your computer. You can use the IDLE program to create files containing Python scripts that are stored on the computer and can be loaded, edited, and run from within the IDLE environment.

When a program wants to communicate with the user, it can use the `print` function to display strings of text and numeric values. A program can also use other functions that are made available by use of the `import` command, which imports the library that contains the functions. The `random` library holds a function called `randint`, which can be used to generate random numbers, and the time library contains a function called `sleep`, which can be used to pause

program execution (but not stop the computer) for a specified duration.

Programs can (and should) contain comments that are ignored by the Python system but can provide valuable information to someone reading the program text. A comment starts with the # (hash or number sign) character and continues on to the end of that line.

The snaps library provides a set of functions you can use to display graphics and text, and play sounds in your Python programs. It's not a formal part of the language, but the functions can be useful when you want to impress someone with your programming skills.

To reinforce your understanding of this chapter, you might want to consider the following “profound questions” about computers, programs, and programming.

Would a user ever use the Python Command Shell?

The Python Command Shell is very useful for programmers. It lets us “play with” Python statements and see their results immediately, rather than having to run an entire program and view the results. However, it is very unlikely that a user would ever need to do this. You can think of the Python Shell as a special tool for programmers; a user would just use the Python program.

What would happen if two libraries contained a function with the same name?

It sounds like this might be a problem, but in fact, there is nothing to worry about here. Remember that when we use a function from a library, we put the library name in front, so, for example, we would use “user.menu” and “system.menu” to refer to menu functions in two different modules.

Can I make comments that are more than one line long?

Some programming languages (for example, Java and C#), let you write “multi-line” comments in programs. As the name implies, a multi-line comment spans several lines of the program. Such comments are used to provide a more exhaustive explanation of a program than is possible with a single-line comment. Python does not allow this. The only way you can create a comment that is

several lines long would be to start each successive line with a # character.

This sounds like it might be a problem, and might get in the way of writing well-documented programs. However, many Python editors, including IDLE, have a command that lets a programmer “comment out” a large block of text that has been selected in the editor. When you select the command, the editor places a # at the beginning of each statement for you. Later in the book, we’ll look at clever ways you can add strings of text to code that you write that can be picked up and used to help a programmer.

Can a Python program run on any computer?

The answer is, “Yes, but you might have to install the Python language.” The standard operating system distributions for some computers (for example, the Raspberry Pi and many other Linux systems), already have Python installed. However, other computers may not have the language installed. The good news is that the Python language is a free download, and there are versions of Python for just about every operating system.

Is the Python language the same on every machine?

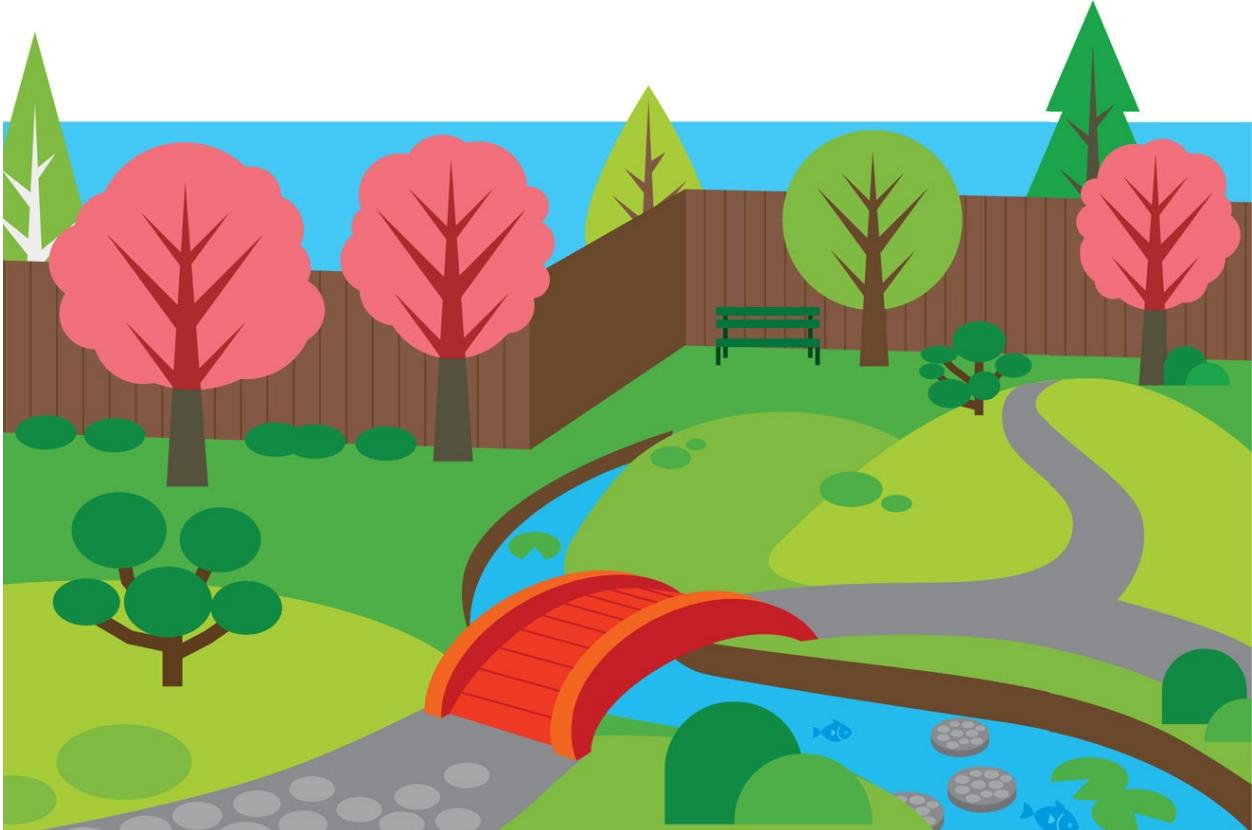
The answer is, “Yes, but not all libraries are available for all machines.” The actual specification of the Python language (that is, how you write programs and what they do) is common to all computers. But remember that there are two versions of Python in common use: Version 2.7 and Version 3.6. Libraries written for one version of Python may not be available for the other.

However, not all Python functions libraries are available on all machines. For example, a Python library called “winsound” can be used to play sounds on Microsoft Windows PCs, but, unfortunately, this library is not available for any other operating systems.

Can I use snaps in my programs?

Absolutely. Remember that snaps are not actually part of Python but are provided to give you a good starting place for writing impressive programs. There are many other libraries that you might like to use, and we’ll cover these as we explore all the things you can do with Python.

4 Working with variables



What you will learn

In this chapter, you'll build more Python programs. You'll discover how programs store data using variables, and you'll learn how Python manages diverse kinds of data that can be stored by a program. You'll learn that computers don't store every value accurately, and you'll learn how a program can request a value from the user and then do something with it. By the end of this chapter, you'll be able to create useful programs.

[Variables in Python](#)

[Working with text](#)

[Working with numbers](#)

[Weather snaps](#)

[What you have learned](#)

Variables in Python

We've seen how Python lets us work with numbers and text. We can use Python as a calculator; we can type calculations as expressions, and Python will evaluate them and display the result. A calculator usually has a memory function that you can use to avoid typing in the same value repeatedly. You can store a frequently used value or a running total in memory and recall it with the touch of a button.

A *variable* is how we add memory to our Python programs. You can think of a variable as a storage location you can refer to by name. You create a variable in a Python program by thinking of a name for the variable and then putting a value in the variable.

```
total = 0
```

This Python statement creates a variable with the name `total`. This variable could be used to hold the total of a sequence of numbers. When Python sees the word `total` in the program, it fetches the contents of the variable called `total`.

The statement above is called an *assignment*, which is not a piece of homework (thankfully); instead, it's the act of assigning one thing to another. [Figure 4-1](#) shows the anatomy of an assignment statement like this.

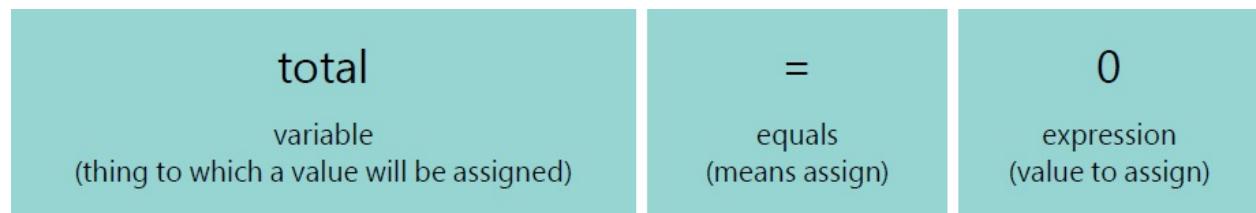


Figure 4-1 Anatomy of an assignment statement

The box on the left of the assignment shows the variable to be assigned. The symbol in the middle is the equals operator, which means “assignment.” The box on the right is an expression, which gives the value to be assigned. The expression can be as complicated as you like.

[Click here to view code image](#)

```
total = us_sales + world_wide_sales
```

This statement would set the value of the variable `total` to the result of adding the contents of the variable `us_sales` to the contents of the variable `world_wide_sales`.



MAKE SOMETHING HAPPEN

Working with variables

Let’s start up the IDLE program and do some work with the Python Shell part of IDLE. We can begin by creating a `total` variable.

```
>>> total = 0
```

If you type the above variable into the Python Shell part of IDLE, you’ll notice something different about the way it behaves. When we give Python something to work out, it calculates the answer and displays it. However, if we set a value in a variable (as we did above), Python doesn’t display an answer. Instead, Python performs the assignment and then awaits another command. We could ask Python to tell us the contents of `total` simply by entering the name of the variable into the Python Shell part of IDLE.

```
>>> total  
0
```

Remember that in [Chapter 2](#), when we entered the value `2`, Python returned that value. In the above statement, we've given Python the value of `total`, which Python has worked out and returned a value of `0`. Now, let's try something a little more complex.

```
>>> total = total+10
```

This expression might appear confusing. If you've worked with mathematical equations, you'll remember that the equals character means that one value is equal to another. From a mathematical point of view, the statement is obviously wrong, because the `total` cannot equal the `total` plus `10`. However, it's important to remember that in Python, the equals operator means "assign." So, the expression `total+10` is evaluated on the right side and then is assigned to the variable on the left side.

In other words, you've added 10 to the contents of `total`. You can check the value of the `total` variable to prove this.

```
>>> total  
10
```

This sequence of actions has performed some simple data processing. The data going into the process was the value in the variable `total`, which emerged from the process with its value increased by 10.

Python names

We used the name `total` for the first variable that we created. When you write

a program, you must come up with names for the variables in that program. Python has rules about the way you can form names:

A name must start with a letter or the underscore character (`_`) and can contain letters, numerals (digits), or the underscore character.

The name `total` is a perfectly legal name for a variable, as is the name `xyz`. However, the variable `2_be_or_not_2_be` would be rejected with an error, because it starts with a numeral instead of a letter or the underscore character. Also, Python views uppercase and lowercase letters differently; for example, `FRED` is regarded by Python as a different name from `fred`.

PROGRAMMER'S POINT

Create meaningful names

I find it terribly surprising that some programmers use variable names such as `X21` or `silly` or `hello_mom`. I don't. I work very hard to make my programs as easy to read as possible. So, I'll use names such as `length` or, perhaps even better, `window_length_in_inches`. The designers of Python have written a style guide that sets out how you should format your variable names. The names we are creating should be expressed as lowercase words separated by the underscore character. You can find more details here: <https://www.python.org/dev/peps/pep-0008/#naming-conventions>

Some other programming languages advise the use of *camel case* to separate the words in their variable names, so they create variables that look like this: `windowLengthInInches`. This is called camel case because the capital letters in the words cause humps, like the back of a camel. Either standard works well, but if you're writing Python I'd advise you to stick with the Python way of doing things.

I don't care which method you choose to make up your variable names; I only care that you strive to create names with meaning. If the name applies to a particular thing, then identify that thing. And if that thing has particular units of measurement, then add those, too. For example, if I was storing the age of a customer, I'd create a variable called `customer_age_in_years`.

Python allows names of any length, and the length of a variable name does not affect the speed of the program, meaning longer variable names don't slow down the program. However, very long names can be a bit hard to read, so you should try to keep them down to the lengths shown in the examples.



CODE ANALYSIS

Typing errors and testing

We've seen what can happen if you type something that Python doesn't understand. You get an error message, which is usually displayed in red. However, you can introduce other errors into your Python code that are difficult to find.

```
Total = total + 10
```

Question: Can you identify an error in the statement above, which is supposed to add **10** to the variable **total**?

Answer: Earlier in this chapter, we used a statement that looks like this to add 10 to the value in a variable called **total**. It looks like we are doing the same thing here, but that's not the case. There is a crucial difference between this statement and the one we saw earlier. This statement assigns the result of the calculation to a newly created variable called **Total**. The statement would not generate any complaints from Python when it runs, but it would also not update the value of **total** correctly.

This is a *logic* error. The statement is completely legal as far as Python is concerned, but it will do the wrong thing when it runs. These errors are very dangerous because Python doesn't alert you to their presence; instead, the program just doesn't work correctly. Python insists on the use of lowercase letters in variable names precisely to avoid this type of error.

Question: How do we prevent logic errors?

Answer: The only way to attack logic errors is to use testing. We

need to run a program with some known values (values for which we know the total) and then verify that the answers agree with the test total. If the answers make sense, we can start to build confidence in our code. However, even if a program passes all the tests, it could still be faulty because there might be a fault that is not picked up by those tests.

Tests don't prove that a program is good; tests simply prove that a program is not as bad as it would be if it had failed the tests.

Tests work best if they are added at the time the program is created. We'll talk about testing strategies every time we make a new program.

```
total = Total + 10
```

Question: The statement above also contains a misspelling of a variable named `total`. However, this time the name on the right-hand side of the equals is misspelled.

What will happen when this program runs?

Answer: Python will refuse to run this statement. It will tell you that you are using a variable with the name `Total`, which it hasn't seen yet. Sometimes typing mistakes will be detected before your program runs, but other times they might not.

You might be thinking that you've been set up to fail because I've suggested that you use long, meaningful names, and typing those long, meaningful names creates more opportunities for mistakes. For now, one way around this problem is to use the text copy feature of your editor to copy names from one part of the program to another.

Later, we'll see other ways that a programmer's editor can prevent you from making typing errors by automatically completing names as they are typed.



MAKE SOMETHING HAPPEN

Make a “Self-Timer” party game

One of the problems with making programs is that when people use them, they come up with ideas to make them better. And that means as the programmer, you must do more work. In this “Make Something Happen,” we’ll look at improving a program we created in the previous chapter.

Refer to the “Nerves of Steel” party game that we created at the end of [Chapter 3](#). In the game, players must remain standing right up to the moment before they think a random timer will expire.

However, one suggestion I’ve received is that the game might require more skill if the program told the players how long they had to stand. The game could be renamed “Self-Timer,” and the winner would be the person who was best at keeping track of time. The sequence of actions the program must follow are:

1. Set the time to remain standing to a random number.
2. Display the time to remain standing.
3. Sleep for the time to remain standing.
4. Display the “Time Up” message

The program will need to use a variable to store the random number of seconds for which the players must remain standing. The name `stand_time` would work well for such a variable.

```
# EG4-01 Self Timer

import time
import random

print('Welcome to Self Timer')
print()
print('Everybody stand up')
print('Stay standing until you think the time has ended')
print('Then sit down.')
print('Anyone still standing when the time ends loses.')
print('The last person to sit down before the time ended will win.')

stand_time = random.randint(5, 20) —————— Get a random stand time and store it

print('Stay standing for', stand_time, 'seconds.') —————— Display the stand time

time.sleep(stand_time) —————— Sleep for the stand time

print('*****TIME UP*****') —————— Display the finished message
```

Working with text

In the previous chapter, we saw that we can write expressions that work with text. It turns out that we can also create variables that can hold text.

```
customer_name = 'Fred'
```

This statement looks exactly like the statement we used to create the `total` variable except that the value being assigned is a string of text. The variable `customer_name` is different from the `total` variable in that it holds text rather than a number. We can use this variable anywhere we would use a string.

[Click here to view code image](#)

```
message = 'the name is '+customer_name
```

In the expression being assigned, the text in the variable `customer_name` is added onto the end of the string "`the name is` ". As `customer_name` currently holds the string "`Fred`" (we set this in the previous statement), the above assignment would create another string variable called `message` which contains "`the name is Fred`".



MAKE SOMETHING HAPPEN

Text and numeric variables

Python keeps track of the contents of each variable and will not allow them to be combined incorrectly. We can use the Python Shell part of IDLE to experiment with string and number variables

[Click here to view code image](#)

```
>>> customer_age_in_years = 25  
>>> customer_name = 'Fred'
```

Start by entering the above two lines into the Python Shell part of IDLE. Python creates two variables. The first is called `customer_age_in_years` and holds the integer value `25`. The second is called `customer_name` and holds the string `'Fred'`. The following code adds these two variables.

[Click here to view code image](#)

```
>>> customer_age_in_years+customer_name  
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    customer_age_in_years + customer_name  
TypeError: unsupported operand type(s) for +:  
'int' and 'str'
```

We saw the same kind of error when we tried to add a string to a number in [Chapter 2](#). Python checks to make sure the operands (the elements on each side of the + operator) make sense. Python doesn't know how to add an integer and a string, so this error is Python's response. If we try to do something silly with our variables, there is a good chance that Python will notice. Now try this:

[Click here to view code image](#)

```
>>> customer_age_in_years='Fred'
```

You might think that this statement should generate an error. The statement is trying to put a string into a variable that Python has created to hold a number. Unfortunately, no error is produced. Instead, Python discards the old, numeric version of the variable and makes a new variable that holds a string. Personally, I don't like this behavior, but it is just the way that Python works. The same thing happens if you put a number into the `customer_name` variable.

Marking the start and end of strings

When I first saw how strings worked in Python, I wondered how to enter text containing a single quote. For example, let's say you want to print the message, "It's a trap." We know Python uses the single quote character to define the limits (or *delimit*) of a string of text. However, the single quote in the word "it's" would confuse Python, making it think the string had ended early.

One way to solve this problem is to enclose the string with double quotation marks rather than single quotation marks.

```
print("It's a trap")
```

Python lets you use either kind of quotation mark (single or double) to delimit a string of text in a program. This works, but of course the next thing I want to ask

is, “How do you enter text that contains both single and double quotes?” The designer of Python thought of that, too, and allows us to use “triple quotes” to delimit a string. A triple quote is three single- or double-quote characters in a row:

[Click here to view code image](#)

```
print('''). . . and then Luke said "It's a trap"'''')
```

This statement prints this message:

[Click here to view code image](#)

```
. . . and then Luke said "It's a trap"
```

Triple quoted strings look a bit cumbersome, but they have another advantage over “ordinary” strings. Any new lines in a triple-quoted string are made part of the string. To see how this might be useful, consider the instructions for the “Nerves of Steel” party game that we looked at in [Chapter 3](#).

[Click here to view code image](#)

```
print('Welcome to Nerves of Steel')
print()
print('Everybody stand up')
print('Stay standing as long as you dare.')
print('Sit down just before you think the time will
end.')
```

To produce these instructions, I had to write several print statements. By using a triple-quoted string, I can make this a lot easier.

[Click here to view code image](#)

```
print('''Welcome to Nerves of Steel
```

```
Everybody stand up  
Stay standing as long as you dare.  
Sit down just before you think the time will end.  
'''')
```

The `print` statement now spans several lines. When the program runs, the new lines are printed so that the text looks as it did before, although it now spans several lines. Note that the blank line below the heading is also preserved when the print is performed.

One thing to remember is that you must use matching delimiters to start and end a string. If you start the string with triple quotes, you must end it that way, too.

Escape characters in text

Another way to include quote characters in a string of text is to use an *escape sequence*. Normally, each character in a string represents that character. In other words, an A in a string means ‘A’. However, when Python sees the escape character—the backslash (\) character—it looks at the text following the escape character to decide what character is being described. This is called an escape sequence. There are many different escape sequences you can use in a Python string. The most useful escape sequences are shown in the following table.

| ESCAPE SEQUENCE | WHAT IT MEANS | WHAT IT DOES |
|-----------------|-------------------------|--------------------------------------|
| \\ | Backslash character (\) | Enter a backslash into the string |
| \' | Single quote (‘) | Enter a single quote into the string |
| \” | Double quote (“) | Enter a double quote into the string |
| \n | ASCII Line | End this line and take a new one |

| Feed/New Line | | |
|---------------|-----------------------|---|
| \t | ASCII Tab | Move to the right to the next tab stop |
| \r | ASCII Carriage return | Return the printing position to the start of the line |
| \a | ASCII Bell | Sound the bell on the terminal |

Python includes other escape sequences, but these will suffice for now.

If you're wondering what ASCII (American Standard Code for Information Interchange) means, it is a mapping of numbers to printed characters. It was developed in the early 1960s for use by computers and persists to this day. (In [Chapter 2](#), we learned that ASCII is the standard that maps the letter W to the decimal value 84).

ASCII is a perfectly fine standard if you don't want to print more than 100 or so different characters. However, because many modern languages use more than 100 characters, UNICODE has become the new standard. UNICODE allows for many more characters, emojis, and emoticons. Some Python escape sequences produce UNICODE characters from our programs. UNICODE characters are frequently used in Graphical User Interfaces (GUIs)..

Not all the escape sequences above work on all computers. Also, their functions are not always consistent from one computer to another. For example, the \a escape sequence, which means ASCII Bell, was intended to sound the bell on a mechanical computer terminal. However, if you print this sequence, there is no guarantee that the computer you're using will make a sound. The paragraph return character, \r, which is supposed to send the print head of a computer terminal back to the start of the line, is not very useful and may not do anything on some computers.

The most useful escape sequences are those that you can use to print quotes and the backspace character. You can also use the new line character (\n) to make new lines in strings you print.

Within Python programs, the end of a line of text is always marked by a single

new line character. The underlying operating system might work differently, for example the Windows operating system uses the sequence “\r\n” (a return followed by a new line feed) to mark the end of each line of text in a file. The Python system performs automatic translation of line endings to match the computer system being used, so our programs can always use a single new line character to mark the end of a line.



CODE ANALYSIS

Investigating escape sequences

The best way to learn about escape sequences is to use them from the Python Shell.

```
print('hello\nworld')
```

Question: What do you think the above statement would print?

Answer: It would print out “hello” on one line and “world” on the next line. The new line escape sequence will cause the program to print on successive lines.

[Click here to view code image](#)

```
print('Item\tSales\nCar\t50\nBoat\t10')
```

Question: What do you think the above statement would print?

Answer: The string to be printed contains tabs and new line characters. It would, in fact, print a tiny table telling us how many cars and boats we had sold:

| Item | Sales |
|------|-------|
| car | 50 |
| boat | 10 |

This form of layout looks quite attractive, but the precise arrangement of the items that are printed depends on how your computer responds to the tab character. Later in the book, we'll investigate better ways of formatting the output from Python programs.

Question: How could I use Python escape sequences to print out this message?

...and then Luke said “It’s a trap.”

Answer: We can use the escape sequence \' to print out the apostrophe (single quote) in the word It’s and then enclose the whole string in single quote characters.

[Click here to view code image](#)

```
print('...and then Luke said "It\'s a trap"')
```

We could also use escape sequences to print the double-quote characters, but we don't have to because the statement above uses single quotes to delimit the string.

Question: Must I use an escape sequence to print a backslash character?

Answer: Yes, I'm afraid you do.

Read in text using the `input` function

Up until now, all our programs have worked with values stored in the Python code itself. These are called *literal* values because they are literally “just there”

in the code. We use the `input` function to make a “complete” program that takes in data, does something with it, and then produces a result.

```
name = input()
```

This statement would pause the program and wait for the user to type in a string and press the Enter key. The string of text is stored in the variable called `name`. We can make the program display a prompt for the user by adding a string to the call for the `input` function.

[Click here to view code image](#)

```
name=input('Enter your name please: ')
```

The `name` variable will contain whatever string the user types in. If the user just presses the Enter key without typing anything, the `name` variable will contain an empty string.

You can also use the `input` statement to pause your program so that the user can read the results.

[Click here to view code image](#)

```
input('Press Enter to continue')
```

In the statement above, the result of the call to the `input` function is ignored.



MAKE SOMETHING HAPPEN

Use `input` to make a “greeter” program

We can use the `input` function in the Python Shell, but it's most useful in a program. We can start by making a simple program that asks for our name and then gives us a personalized greeting. Use IDLE to create a new program file and then enter the following program code.

[Click here to view code image](#)

```
name = input('Enter your name please: ')
print('Hello', name)
```

Run the program, and save it in a file named “greeter.”

When the program runs, it will print out the prompt, request the name, and then print “hello” to that name. The name text is entered as a string. Enter the program and run it. Type in your name and have your computer talk to you.



WHAT COULD GO WRONG

Python versions and the `input` function

One thing about Python that gives me nightmares is how `input` works in different versions of the language. The input examples we looked at will work very well, but *only if you are using version 3.6 of Python*. If you’re using version 2.7, you’ll get errors.

[Click here to view code image](#)

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    name=input('Enter your name please: ')
  File "<string>", line 1, in <module>
NameError: name 'Rob' is not defined
```

The reason for this error is that the `input` statement works differently in earlier versions of Python. Rather than just taking what you type and storing it in a variable, the `input` function in Python 2.7 tries to evaluate the expression.

In the case of the above error, I entered my name “Rob.” Python tried to look for a variable called “Rob” to use in an expression, didn’t find one, and produced the error.

In Python 2.7, we would use the function `raw_input` to read in a string of text from the user:

[Click here to view code image](#)

```
name = raw_input('Enter your name please: ')
```

This statement in version 2.7 produces the same result as the `input` statement in version 3.6.

One final thing to keep in mind is that the `raw_input` function does not exist in Python 3.6, which makes things even more confusing. The bottom line is that if your program suddenly starts to report problems with the input process, make sure you’re using correct functions for your version of Python. This is particularly important when other people start using programs you’ve written.

If you think the ability to evaluate what the user types in as a Python statement sounds fun, it turns out that it is; you can use the `eval` function to do just that. The `eval` function can be used to evaluate any string of text, not just those entered by the user.

Working with numbers

Convert strings into integer values

The `input` function returns a string of text, which is fine if we just want to read names. However, it is not so useful if we want to read in numbers. For instance, we might want to modify the egg timer we created in [Chapter 2](#) so that the user can enter the number of seconds that the timer must run. This would allow the user to customize their egg from raw (0 seconds) to hard boiled (600 seconds). Python provides a function called, not surprisingly, `int`. The `int` function accepts a string and returns the number in that string.

```
time_text = input('Enter the cooking time in seconds: ') Enter the time as a text string  
time_int = int(time_text) Convert the text string into a number
```

The first statement uses the `input` function to read in a string from the user. The second statement uses the `int` function to convert this string into a number that can be used to set the length of time the program sleeps while the egg is cooked. The final egg timer program looks like this:

[Click here to view code image](#)

```
# EG4-02 User Configurable Egg Timer  
  
import time  
  
time_text = input('Enter the cooking time in  
seconds: ')  
  
time_int = int(time_text)  
  
print('Put the egg in boiling water now')  
  
time.sleep(time_int)  
  
print('Take the egg out now')
```

You could use this program anywhere you want to have a configurable timer or alarm. You just change the messages displayed to the user.



CODE ANALYSIS

Reading numbers

Let's look at this program and consider a few questions.

Question: How many variables are used in the program above?

Answer: There are two variables. One is called `time_text` and contains a string; the other is called `time_int` and contains an integer. I didn't have to use these particular names; I could have called the variables `x` and `y`, but I think these names make the program clearer.

Question: Could you write the program without using the `time_text` variable?

Answer: Yes, I could. The `input` function returns a string that I could feed directly into the `int` function.

[Click here to view code image](#)

```
time_int = int(input('Enter the cooking time  
in seconds: '))
```

I'm not a huge fan of compressing programs like this though. They don't make the program easier to understand, and they have a negligible effect on the speed of the program or the amount of memory used when it runs. This statement would also make it impossible for the program to display the string entered by the user because that string is not stored anywhere.

Question: What do you think will happen if the user enters something other than a number?

Answer: Try it using the Python Shell part of IDLE.

```
>>> x = int('kaboom')
```

This Python statement will attempt to convert the string ‘kaboom’ into a number and store it in a variable called `x`. As you might expect, this will not end well.

[Click here to view code image](#)

```
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    x = int('kaboom')
ValueError: invalid literal for int() with
base 10: 'kaboom'
```

This looks worrying because it means if the user types in text rather than a number, the program will fail. For now, we’ll just have to live with this problem, but later in the book we’ll look at how our programs can deal with this error and give the user another chance to enter a number.

Whole numbers and real numbers

We know that Python is aware of two fundamental kinds of data—text data and numeric data. Now we need to delve a little deeper into how numeric data works. There are two kinds of numeric data—whole numbers and real numbers. Whole numbers have no fractional part. Up until now, every program that we have written has made use of whole numbers. A computer stores the value of a whole number exactly as entered. Real numbers, on the other hand, have a fractional element that can’t always be held accurately in a computer.

As a programmer, you need to choose which kind of number you want to use to store each value.



CODE ANALYSIS

Whole numbers versus real numbers

You can learn about the difference between whole numbers and real numbers by looking at a few situations in which they might be used.

Question: I'm building a device that can count the number of hairs on your head. Should I store this value as a whole number or as a real number?

Answer: This should be a whole number because there is no such thing as half a hair.

Question: I want to use my hair-counting machine on 100 people and determine the average number of hairs on all their heads. Should I store this value as a whole number or as a real number?

Answer: When we work out the result, we'll find that the average includes a fractional part, which means that we should use a real number to store it.

Question: I want to keep track of the price of a product in my program. Should I use whole numbers or real numbers?

Answer: This is very tricky. You might think that the price should be stored as a real number—for example, \$1.50 (one and a half dollars). However, you could also store the price as the whole number, 150 cents. The type of number you use in a situation like this depends on how that number is used. If you're just keeping track of the total amount of money you take in while selling your product, you can use a whole number to hold the price and the total. However, if you are also lending money to people to buy your product and you want to calculate the interest to charge them, you would need a fractional component to hold the number more

precisely.



PROGRAMMER'S POINT

The way you store a variable depends on what you want to do with it

It seems obvious that you would use a whole number to count the number of hairs on your head. However, one could argue that we could also use a whole number to represent the average number of hairs on 100 people's heads. This is because the calculated average would be in the thousands. Fractions of a hair would not add much useful information, so we could drop any fractional parts and round to the nearest whole number. When you consider how you are going to represent data in a program, you must take into account how it will be used.

Real numbers and floating-point numbers

Real number types have a fractional part, which is the part of the number after the decimal point. Real numbers are not always stored exactly as they are entered into Python programs. Numbers are mapped to computer memory in a way that stores a value that is as close as possible to the original. The stored data is often called a *floating-point* representation. You can increase the accuracy of the storage process by using larger amounts of computer memory, but you are never able to hold all real values precisely.

This is not a problem, however. Values such as pi can never be held exactly because they “go on forever.” (I’ve got a book that contains the value of pi to 1 million decimal places, but I still can’t say that this is the exact value of pi. All I can say is that the value in the book is many times more accurate than anyone will ever need.)

When considering how numbers are stored, we need to think about *range* and *precision*. Precision sets out how precisely a number is stored. A particular floating-point variable could store the value 123456789.0 or 0.123456789, but it can’t store 123456789.123456789 because it does not have enough precision to hold 18 digits. The range of floating-point storage determines how far you can “slide” the decimal point around to store very large or very small numbers. For example, we could store the value 123,456,700, or we can store 0.0001234567.

For a floating-point number in Python, we have 15 to 16 digits of precision, and we can slide the decimal point 308 places to the right (to store huge numbers) or 324 places to the left (to store tiny numbers).

The mapping of real numbers to a floating-point representation does bring some challenges when using computers. It turns out that a simple fraction such as 0.1 (a tenth) cannot be held accurately by a computer because of the way values are held. The value stored to represent 0.1 will be very close to that value, but not the same. This has implications for the way we write programs.



CODE ANALYSIS

Floating-point variables and errors

We can find out more about how floating-point values work by doing some experiments using the Python Shell. If we just type numeric expressions, we can view the results that Python calculates.

Question: What happens if we try to store a value that can't be held accurately as a floating-point value?

Answer: We know that the value 0.1 can't be held accurately in a computer, so let's enter that value into the Python Shell and see what comes back.

```
>>> 0.1  
0.1
```

At this point, you might think that I've been lying to you because I said that the value 0.1 can't be held accurately, and now this example shows Python returning the value 0.1. However, I'm not lying to you—Python is. The Python print routine “rounds” values when it prints them. In other words, it says that if the number to be

printed is `0.1000000000000000551115` or thereabouts (which it is), then it will just print 0.1.

Question: Does this “rounding” really happen?

Answer: At the moment, you’ve just got my word for it that values are rounded when printed and that errors are being hidden from us as a result. However, if we perform a simple calculation, we can introduce an error that is large enough to escape being rounded. Enter the following calculation into the Python Shell and note what comes back.

```
>>> 0.1+0.2  
0.3000000000000004
```

The result of adding 0.1 to 0.2 should be 0.3. But, because the values are held as binary floating-point values, the result of the calculation contains an error large enough to escape being rounded. It turns out that our highly expensive computer really can’t get its sums right!

You might think that your all-powerful computer should be able to hold all values precisely. It comes as a bit of a shock to discover that this is not true and that a simple pocket calculator can outperform your powerful PC.

However, this lack of accuracy is not a problem in programming because we don’t usually have incoming data that is particularly precise anyway. For example, if I refine my hair counting device to measure hair length, it would be very difficult for me to measure hair length with more than a tenth of an inch (2.4 millimeters) of accuracy. For hair data analysis, we need only around three or four digits of accuracy. It is very unlikely that you will ever process any data that requires the 15 digits that Python can give you.

It is also worth noting at this point that these issues have nothing to do with Python. Most, if not all, modern computers store and manipulate floating-point values using a standard established by the Institute of Electronic and Electrical Engineers (IEEE) in 1985. All programs that run on a computer will manipulate

values in the same way, so floating-point numbers in Python are no different from those in any other language.

The only difference between Python floating-point values and those in other languages is that a floating-point variable in Python occupies 8 bytes of memory, which is twice the size of the float type in the languages C, C++, Java, and C#. A Python floating-point variable equates with a *double precision* value in those languages.

If you really, really want to store things with even more accuracy (and I think this is terribly unlikely), you should look at the [decimal](#) and [fraction](#) libraries supplied with Python.

PROGRAMMER'S POINT

Don't confuse precision with accuracy

It is very important to remember that numbers don't become more accurate just because they are stored with more precision. Scientists in a laboratory measuring the length of ant legs will not be able to do this to more than a few digits of accuracy (unless they have some amazing technology), so there is no point in them using much higher precision to store and process their results. Using higher precision slows down the program and means that the variables take up more space in memory.



CODE ANALYSIS

Working with floating-point variables

We know Python automatically creates variables for use in our programs. The type of a variable is determined by what a program stores in it.

```
name = 'Rob'  
age = 25
```

The above statements create two variables. The variable `name` is of string type, because it has been assigned a string of text. However, the variable `age` is of integer type, because it has been assigned an integer.

We can use the Python Shell part of IDLE to learn how floating-point variables work.

Question: How do I create a floating-point variable?

Answer: We can create a floating-point variable by assigning a floating-point expression to the variable.

```
>>> x = 1.5
```

This statement creates a floating point variable called `x` which contains the value `1.5`. We can view the contents of the variable by just entering its name.

```
>>> x  
1.5
```

Question: What happens if I assign an integer to a floating-point variable?

Answer: We can investigate this behavior by creating a new variable.

```
>>> y = 1.0
```

This statement creates a variable called `y` that contains the integer value `1`. But is `y` an integer or floating-point variable? We can find out by viewing the contents of `y`.

```
>>> y  
1.0
```

Python has printed out the value with a fractional part (which is zero). This indicates that the value is a floating-point value. In other words, the presence of a decimal point in a printed value tells us that the value is floating point. The value is always floating point when a decimal is included, even if there are no decimal places.

Question: What about floating-point and integer calculations?

Answer: We can investigate the results of calculations by entering a few and then looking at the results.

```
>>> 2+2  
4
```

Recall from earlier chapters that when we add two integers together, we get an integer result.

```
>>> 2.0+2.0  
4.0
```

This is the same calculation, but this time Python produces a floating-point result because the operands (the things upon which the + operator is working) are both floating-point values.

```
>>> 2.0+2  
4.0
```

It turns out that when an expression contains one floating-point value, the result of the calculation is automatically converted to a floating-point value. Generally, if the expression contains integers, it will generate an integer result. If the expression contains one

floating-point value, the expression will generate a floating-point result.

One exception to the “integers make integers” rule is when we divide one integer by another.

```
>>> 1/2  
0.5
```

The above statement divides one integer by another. The / (slash) character is how we denote division in Python programs. In this case, the result of dividing 1.0/2.0 is to produce a floating-point result of 0.5.



WHAT COULD GO WRONG

Python versions and integer division

Integer division in different versions of Python is another situation in which the strangeness of Python makes me tear my hair out. Above, I just told you that dividing an integer by an integer produces a floating-point result. This is true in Python version 3.6, the version we’re using for the examples in this book. However, in Python 2.7, this is not the case. In Python 2.7, the result of dividing an integer by an integer is another integer.

```
>>> 1/2  
0
```

The value of 1/2 is a half, which can’t be held in an integer, so the result of this calculation is zero in Python 2.7. In fact, any number less than 1

is truncated to zero in this calculation. That means the result of 9/10, which should be 0.9, also turns out to be zero. This can result in sums being quite different from the values we might expect.

This raises the horrible prospect of Python programs that work correctly in Python 3.6 producing completely wrong answers when they run in older versions of Python. This has happened to me on numerous occasions. The best advice I can give is to always put a decimal point on a value if you want it to generate floating-point results when used in calculations.

I feel bad telling you this, but I'd feel worse keeping it a secret. You can get a Python 2 program to behave the same way as Python 3 by telling it to use the updated division routines:

[Click here to view code image](#)

```
from __future__ import division
```

You could give this command at the start of a program to make Python 2 division behave the same as Python 3 division.

Convert strings into floating-point values

The `float` function is used to convert values into floating-point values. A program can use the `float` function to convert a string of text into a floating-point value. It works in the same way as the `int` function:

```
time_text=input('Enter the cooking time in seconds: ')
```

Read in the time as a string

```
time_float=float(time_text)
```

Convert the string into a
floating point number

The statements above could form the basis of an ultra-precise version of the egg timer program that lets the user time their egg to a fraction of a second. You can find this program in **EG4-03 Ultra-precise Egg Timer**.

The program works because the `sleep` function accepts a floating-point value for the duration of the delay.

```
time.sleep(time_float)
```

This means we can use the `sleep` function to create very short delays in programs.

The `float` function can also be used to convert an integer value into a floating-point value. This can be useful in older Python programs as a way of forcing the correct behavior of the divide operator (see the “What Could Go Wrong: Python Programs and Integer Division” discussion above).

```
z = float(1)
```

In the statement above, the variable `z` will be of type `float`, even though the value being assigned is an integer.

The behavior of `float` might seem rather familiar. This is because it behaves similarly to the `int` function, except that it delivers floating-point results.

Perform calculations

In Chapter 2, we saw that Python expressions are made up of operators and operands. The operators identify actions to be performed, and the operands are worked on by the operators. Now we can add a bit more detail to this explanation. Expressions can be as simple as a single value or as complex as a large calculation. Here are a few examples of numeric expressions:

```
2 + 3 * 4  
-1 + 3  
(2 + 3) * 4
```

These expressions are evaluated by Python working from left to right, just as you

would read them yourself. Again, just as in traditional math, multiplication and division are performed first, followed by addition and subtraction.

Python achieves this order by giving each operator a priority. When it works out an expression, it finds all the operators with the highest priority and applies them first. It then looks for the operators next in priority, and so on, until the result is obtained. The order of evaluation means that the expression $2 + 3 * 4$ will calculate to 14, not 20.

If you want to force the order in which an expression evaluates, you can put parentheses around the elements of the expression you want to evaluate first, as in the final example above. You can also put parentheses inside parentheses if you want—provided you make sure that you have as many opening parentheses as closing ones. Being a simple soul, I tend to make things very clear by putting parentheses around everything.

It is probably not worth getting too worked up about expression evaluation. Generally speaking, things tend to be evaluated how you would expect.

Here is a list of some other operators, with descriptions of what they do, and their precedence (priority). The operators are listed with the highest priority first.

| OPERATOR | HOW IT'S USED |
|----------|--|
| - | Unary minus, the minus that Python finds in negative numbers, |
| * | Multiplication. Note the use of the * rather than the more mathematically correct but confusing x. |
| / | Division, because of the difficulty of drawing one number above another during editing, we use this character instead. |
| + | Addition. |
| - | Subtraction. Note that we use the same character as for unary minus. |

This is not a complete list of the operators available, but it will do for now. Because these operators work on numbers, they are often called *numeric operators*. However, one of them, the + operator, can be applied between strings, as you've already seen.



CODE ANALYSIS

Work out the results

Question: See if you can work out the values of a, b, and c when the following statements have been evaluated:

```
a = 1  
b = 2  
c = a + b  
  
c = c * (a + b)  
b = a + b + c
```

Answer: a=1, b=12, c=9. The best way to work this out is to behave like a computer would and work through each statement in turn.

When I do this, I write down the variable values on a piece of paper and then update each as I go along. Doing this means that you can predict what a program will do without having to run it.



WHAT COULD GO WRONG

Dumb calculations

Because you can use the division operator in an expression, you can write silly code such as this:

```
>>> 1/0
```

This code tries to divide one by zero, giving a non-sensible result. You might think that this would cause the computer itself to crash. In the old days, this might have happened. I have fond memories of a calculator I used to own. If I tried to divide one by zero, it would just keep counting up, trying to reach a result of infinity. In a Python program, the Python engine will simply stop your program from going any further.

[Click here to view code image](#)

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

Convert between `float` and `int`

Sometimes a program might need to convert between floating-point and integer values. A program can do this by using the `int` function. We've already used the `int` function to convert strings of text into integer values. To do this, we used the version of the `int` function that accepts a string as an input:

```
i = int('25')
```

If the `int` function is supplied with a string containing a number, it will read that number out of the string. In other words, the statement above will result in the variable `i` containing the integer value 25.

Programs can also use the `int` function to convert a value from floating point to integer.

```
i = int(2.9)
```

This would result in the creation of an integer variable called `i` that contains the value 2. You might ask why the value if `i` is not set to 3, because it seems reasonable to “round” a value to the nearest integer. However, in this respect Python is not reasonable. Floating-point values are always truncated. In other words, the fractional part is always discarded.



MAKE SOMETHING HAPPEN

Calculating a pizza order

You might wonder why you would need to convert floating-point values into integers. Here’s an example. Rigorous scientific research conducted by me at many hackathons I’ve attended has arrived at a figure of exactly 1.5 people per pizza. In other words, if I’m catering for 30 students I’ll need 20 pizzas, and so on.

I decided to write a Python program that works out how many pizzas I need to order, given a particular number of students. The program reads in the number of students, does the calculation, and then prints the result. This is my first version:

```
# EG4-04 Pizza Order Calculator

students_text = input('How many students: ') ————— Get the number of students
                                                    as a string

students_int = int(students_text) ————— Convert the text into a float

pizza_count = students_int/1.5 ————— Calculate the number of
                                                    pizzas required

print('You will need', pizza_count, 'pizzas') ————— Print the result
```

This version is fine with the test data above. If I say there are 30 students, the program will tell me I need 20 pizzas. However, there are problems with some numbers of pizzas:

[Click here to view code image](#)

```
How many students: 40
You will need 26.666666666666668 pizzas
```

I can't ask the pizza place for a fraction of the pizza, so I need a way of converting the number of pizzas to an integer. At this point, I also must decide what the conversion will do. If I just use the int conversion on the result above, this will result in an order for 26 pizzas because the int function truncates the floating-point value. This effectively means that I'll have pizza for only 39 people rather than 40, leaving one hungry student.

There are several ways to address this problem. I might think the best way to attack the problem is to add one extra pizza to the order to take care of any "spares."

[Click here to view code image](#)

```
pizza_count = (students_int/1.5)+1
```

Load the example program and modify it using the above statement so that when you tell it there are 40 students, the program suggests that you buy 27 pizzas.

Then change the program to make it less generous and always rounds down to the nearest integer number of pizzas to order.



PROGRAMMER'S POINT

Never assume that you know what a program is supposed to do

If you wrote the pizza calculator for a customer, you should *not* decide for yourself what the program should do if it must order a fraction of a pizza. Your customer might be happy to “round down” the number of pizzas to keep their costs down. If this is the case, they will complain when your program adds an extra pizza.

As a programmer, never assume that you know what the program should do. You must always ask the customer. Otherwise, you might find yourself paying for over-ordered pizzas.



MAKE SOMETHING HAPPEN

Converting between Fahrenheit and centigrade

To convert a temperature from Fahrenheit to centigrade, you subtract 32 from the Fahrenheit value and then divide the result by 1.8. You could write a Python program to perform this calculation. If you look at the pizza order program above, you’ll discover that you already have most of the program written. You just need to change the messages that the program displays and the statements that perform the calculation.

You can now write any kind of conversion program you like, converting feet to meters, grams to ounces, or liters to gallons.

Weather snaps

Converting from Fahrenheit to centigrade is even more useful if you actually have some weather data to work from. The snaps function named `get_weather_temp` returns the temperature of a location in the United States. The information is provided by the US National Weather Service, www.weather.gov. You must supply the method with the latitude and longitude of the location for which you want the temperature. Here’s an example that gets

the temperature for Seattle, Washington.

```
# EG4-05 Seattle Temperature
```

import snaps Import the snaps library

temp = snaps.get_weather_temp(latitude=47.61, longitude=-122.33) Get the temperature

print('The temperature in Seattle is:', temp) Print the temperature

The latitude and longitude items in the call to the function are given as named arguments. You first saw examples of these in [Chapter 3](#) when we gave named arguments to control the behavior of the snaps `display_message` function. Here we are giving two arguments that describe a location.

You can find the latitude of a town or city in the United States by using the Bing search engine. Just search for “Placename Latitude”, where Placename is where you live, and you’ll get the values you need to use.

If you want a brief description of the weather conditions at a location, you can use the `get_weather_description` method, which returns a short string describing the conditions at a given location.

[Click here to view code image](#)

```
# EG4-06 Seattle Weather
```

import snaps

desc=snaps.get_weather_descipton(latitude=47.61,
longitude=-122.33)

print('The conditions are:', desc)

Keep in mind that these methods provide weather information only for locations in the United States. If you try to use latitude and longitude values for other countries or regions, the methods will cause your program to fail.



MAKE SOMETHING HAPPEN

Weather display program

You can now create a program that displays the current weather conditions. If you use the `display_text` function from Snaps, your program can display the temperature and the current weather description.

What you have learned

In this chapter, I've shown you how to create variables in Python programs. You've learned that a variable is a named location in memory and that there are rules concerning what a valid name can be. Essentially, a Python name must start with a letter or underscore (_) character and contain letters, numbers, or an underscore after the first character. You've seen how Python determines the type of variable to create from the type of value being assigned. Assigning an integer value to a variable creates an integer variable, and so on.

You've also discovered that there are two fundamental kinds of data in programs: text and numbers. Python provides the string type to hold text, and a program can use the `input` function to read a line of text from a user at the Python Shell and store that text in a string variable (although you also noticed that this behavior differs between the two major versions of Python). You've seen how Python provides the `int` function to convert a string that holds a number into the value that string represents, and that programs can use a combination of `input` and `int` to read numbers from program uses.

Considering numeric values, you've seen the fundamental difference between an integer value with no fractional part (for example, 1) and a real-number value that contains a fractional element (for example, 1.5). We also explored how a floating-point value stored in the computer can approximate the actual value, which might lead to problems when performing calculations on real numbers.

Finally, you looked at how calculations are performed in programs, how to convert between floating-point values and integer values, and why a program might need to do this.

To reinforce your understanding of the content, consider the following “profound questions” about variables and values.

What happens if I “overwrite” a variable of one type with a value of another type?

You might think that this would cause an error. For example, you might expect Python to give you an error along the lines of “Last time you used this variable you put a number in it, and now you’re putting a string in it.” However, this is not what happens. Every time you make an assignment, Python creates a brand-new variable with the appropriate type and discards any existing variable with the same name. So, a program never “overwrites” a variable; it simply makes a new one with the same name and different contents.

Does using long variable names slow down a program?

You might think using a variable called “sales_total_for_July” would be more difficult than using one called “sj.” And to an extent, you might be right. However, the effect on a Python program’s speed is so small that it’s insignificant. I’d much rather use longer variable names that make a program easy to understand.

Can we write all our programs using only floating-point variables?

You might think that using floating-point variables would make our programs simpler. However, you can still make a compelling case for using integers where appropriate. We’ve seen that the results of some calculations might not be what we would like them to be. It is perfectly possible that a calculation that should produce the value 1 can produce values such as 1.0000000004. If a program compares the values 1 and 1.0000000004, it would decide that they are different and perhaps cause a program to behave incorrectly.

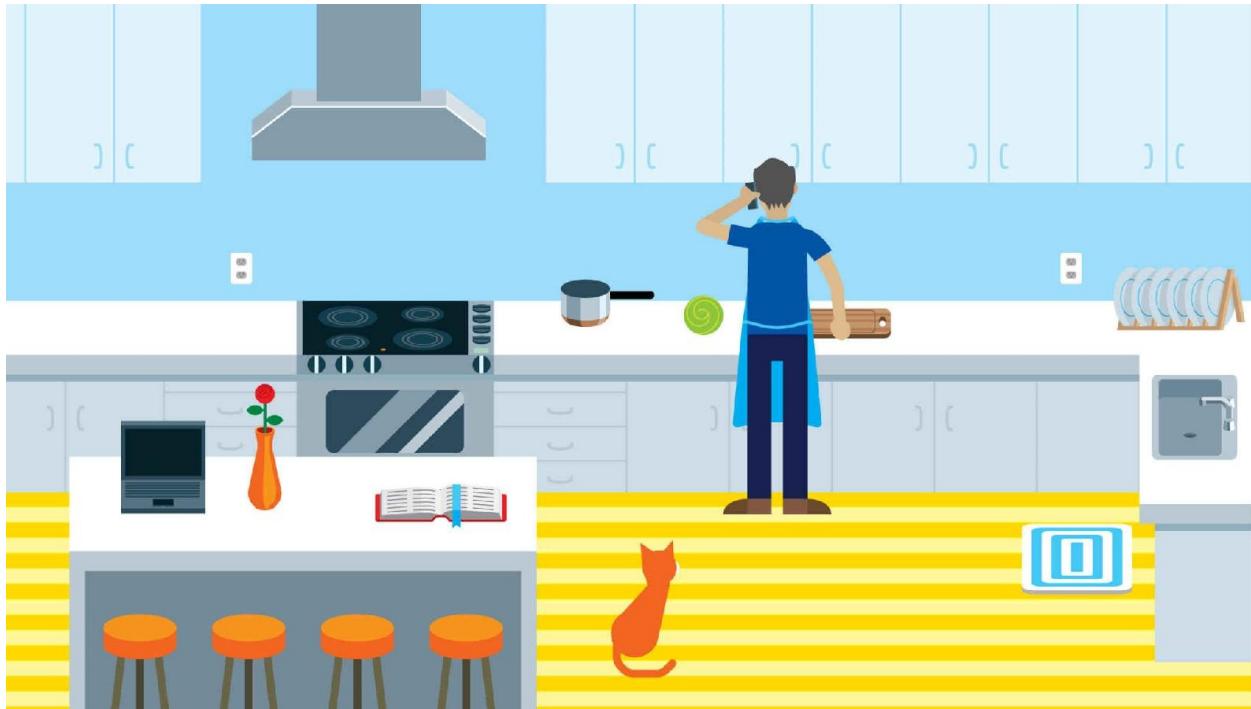
For this reason, I try to make sure that when I create variables, I use values that will create an appropriate type. When counting things, I’ll use integers; if I need

to perform calculations, I'll use floating-point values.

Can I stop my program from crashing if someone types in an invalid number?

Yes. You haven't yet learned how to do it, but there is a mechanism in Python that lets a program take control when Python encounters an error. You can make a program that displays an error message when a user makes a mistake and gives the user another chance to enter the value. We'll explore this in the section "["Detect invalid number entry using exceptions"](#)" in [Chapter 6](#).

5 Making decisions in programs



What you will learn

I've described a computer as like a sausage machine that accepts an input, does something with it, and then produces an output. This is a great way to start thinking about computers, but a computer does a lot more than that. Unlike a real-life sausage machine, which simply tries to create sausage from anything you put in it, a computer can respond to different inputs in different ways. In this chapter, you'll learn how to make your programs respond to different inputs. You'll also learn about the responsibility that comes with making the computer work in this way because you must be sure that the decisions your programs make are sensible.

[Boolean data](#)

[The `if` construction](#)

[Use decisions to make an application](#)

[Input snaps](#)

[What you have learned](#)

Boolean data

In [Chapter 4](#), you learned that programs use variables to represent different types of data. I like to think that you will forever associate the number of hairs on your head with whole numbers (integers) and the average length of your hair with real numbers (floating-point values). Now it's time to meet another data type: the Boolean values. Unlike the numeric data types, which provide a range of values, the Boolean type has only two possible values: True or False. Perhaps you could use a Boolean value to represent whether a given person has any hair.

Create Boolean variables

A program can create variables that can hold Boolean values. As with other Python data types, the Python engine will deduce the type of a variable from the context in which it is used.

[Click here to view code image](#)

```
it_is_time_to_get_up = True
```

The above statement creates a Boolean variable called `it_is_time_to_get_up` and sets its value to `True`. In my world, it seems that it is always time to get up. In the highly unlikely event of me ever being allowed to stay in bed, we can change the assignment to set the value to `False`:

[Click here to view code image](#)

```
it_is_time_to_get_up = False
```

The words `True` and `False` are “built in” to Python, so that when Python sees

these values, it thinks in terms of Boolean values. Note that you must capitalize the first letters in **True** and **False**; the words “true” and “false” don’t work.



CODE ANALYSIS

Boolean values

Question: What do you think would happen if you printed the contents of a Boolean value?

[Click here to view code image](#)

```
print(it_is_time_to_get_up )
```

Answer: When you print any value, Python will try to convert that value into something sensible for us to look at. In the case of Boolean values, it will print “True” or “False”.

```
True
```

Question: What do you think will happen if I give the word **True** to the **input** function?

[Click here to view code image](#)

```
>>> x=input("True or False: ")
True or False: True
```

This statement asks me to type in True or False. I’ve obligingly typed in the word True and the result has been assigned to the variable x. Given that we know the word “True” is regarded as special when written

inside a program, what will be the type of the variable x?

Answer: The type of the variable x will be a string. Words Python regards as special only have meaning in the program text, not when they are entered into a running program.

The value of x will be displayed as a string enclosed in quotes, just like any other string we might store in a program.

```
>>> x  
'True'
```

In Python 3.6 the `input` function takes text from the user and stores it as a string. However, in Python 2 anything given to an `input` function is evaluated as a Python expression. The value of the Python expression “True” is, not surprisingly, the Boolean value “True.” So, if we ran the above statement in Python 2, the type of x really would be a Boolean variable if you typed in `True` or `False`. See the discussion, “What could go wrong? - Python versions and the `input` function” in [Chapter 4](#) for more details of this scary state of affairs.

Question: Is there a Python function called `bool` that will convert things into Boolean, just like there are `int` and `float` functions?

Answer: Indeed there is. We can investigate the way it works using the Python Command Shell part of IDLE.

[Click here to view code image](#)

```
>>> bool(1)  
True  
>>> bool(0)  
False  
>>> bool(0.0)  
False  
>>> bool(0.1)
```

```
True
>>> bool('')
False
>>> bool('hello')
True
```

I used the Python Shell to evaluate some expressions involving the `bool` function. You can see that the value zero is regarded as `False`, as is an empty string. Anything else is `True`.

Question: What happens if a program combines Boolean values with other values?

Answer: We know if a Python program tries to combine things in an improper way (for example, adding a number to a string), an error is produced.

[Click here to view code image](#)

```
>>> 'Hello'+True
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    'Hello'+True
TypeError: must be str, not bool
```

If we try to add a Boolean value to a string, we get an error. Python returns an error message. However, things get more complicated when we combine Boolean values with other types of values:

```
>>> 1+True
2
```

The above statement adds the value `True` to the value 1. And it works, producing the result 2. Python is quite happy to use Boolean values in arithmetic. The Boolean value `True` is mapped to the

value 1; the Boolean value `False` is mapped to 0.

```
>>> 1+False  
1
```

Boolean expressions

A Boolean variable can be assigned to any expression that returns a value that is either `True` or `False`. As an example, let's see if we can create a Python program that will serve as an alarm clock.

The first thing we'll need is a way for the program to acquire time information. It turns out that the sensibly named `time` library provides a function that will fetch the time.

```
import time  
current_time = time.localtime()  
hour = current_time.tm_hour
```

The diagram illustrates the execution flow of the provided Python code. It consists of three horizontal lines. The first line starts with 'import time' and ends with a light blue box labeled 'Import the time library'. The second line starts with 'current_time = time.localtime()' and ends with a light blue box labeled 'Get the current time'. The third line starts with 'hour = current_time.tm_hour' and ends with a light blue box labeled 'Split off the hour value'.

The code above will fetch the hour component of the time and store it in a variable called `hour`. Note that the first statement imports the `time` library that contains the `localtime` function.

The `localtime` function returns a Python *object* that contains attributes that represent the current time, including the hour. The variable `current_time` is set to the time returned by `localtime`. One of the attributes of the `current_time` object is the hour value, which is what we want to use.

An *attribute* is what Python calls a piece of data that is part of an object. In the same way that several great programming books can be attributed to the author "Rob Miles," the `tm_hour` value can be attributed to an object produced by the `localtime` function.

| ATTRIBUTE | VALUE |
|-----------|--|
| tm_year | Year (for example, 2017) |
| tm_mon | Month in the range 1 to 12 with 1 being January |
| tm_mday | Day in the month in the range 1 to month length |
| tm_hour | Hour in the day in the range 0 to 23 |
| tm_min | Minute in the hour in the range 0 to 59 |
| tm_sec | Second in the minute in the range 0 to 59 |
| tm_wday | Day of the week in the range 0 to 6 with Monday as 0 |
| tm_yday | Day in the year in the range 0-364 or 0-365 in a leap year |

This table shows some of the most useful attributes of the object returned by the [localtime](#) function. We can use these to create Python programs that are aware of the current date and time.

If all this is hurting your head, you can think of the time value as a piece of paper that has a table containing time data printed on it. [Figure 5-1](#) shows what the table might look like.

| ATTRIBUTE | VALUE |
|-----------|-------|
| tm_year | 2017 |
| tm_mon | 7 |
| tm_mday | 19 |
| tm_hour | 11 |
| tm_min | 40 |
| tm_sec | 30 |
| tm_wday | 2 |
| tm_yday | 200 |

Figure 5-1 A time value table

The Attribute column on the left contains the name of each attribute. The Value column on the right contains the value of each attribute. The `localtime()` function returns this piece of paper, and then the program can look at values against each attribute in the table to get the current time information. We'll learn a lot more about objects later in the book.



MAKE SOMETHING HAPPEN

Make a one-handed clock

Let's start up the IDLE program and use the `localtime()` function to make a clock that displays only the hour value. These are called “one-handed clocks” and are supposed to promote a more relaxed

attitude to life. Open IDLE and click **New File** on the **File** menu to create a new program file. Enter the following program:

```
# EG5-01 One handed clock Version 1.0

import time ————— Import the time library

current_time = time.localtime() ————— Get the current time

hour = current_time.tm_hour ————— Split off the hour value

print('The hour is', hour) ————— Print the hour value
```

Run the program and save it in a file called “OneHandedClock.” It should print out the hour the program was run.

You could modify this to produce a more fully featured clock that gives the time (and perhaps even the date) when it runs. Use **Save As** on the **File** menu to save the improved clock in a different file.

Comparing values

We’ve said that Python expressions are made up of operators (which identify the operation) and operands (which identify the items being processed). **Figure 5-2** shows our first expression, which worked out the calculation, $2+2$.

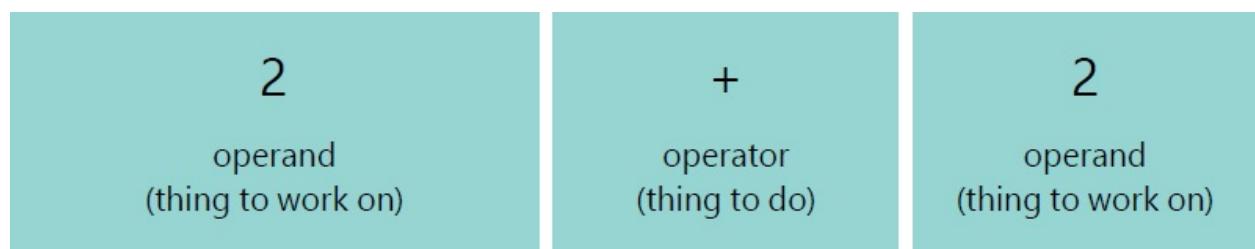


Figure 5-2 An arithmetic operator

An expression can contain a comparison operator (**Figure 5-3**):

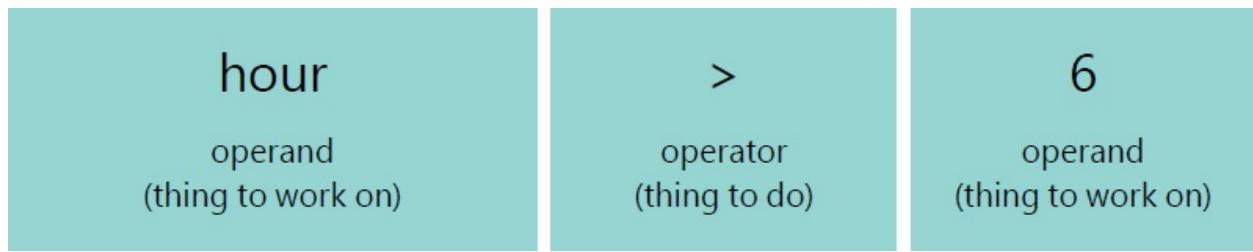


Figure 5-3 A comparison operator

An expression containing a comparison operator returns a Boolean value, which is a value that is either **True** or **False**. The **>** operator in this expression means “greater than.” If you read the expression aloud, you say “hour greater than six.” In other words, this expression is **True** if the hour value is greater than 6. I need to get up after 7 o’clock, so this is what I need for my alarm clock.

Comparison operators

These are the comparison operators that you can use in Python programs:

| OPERATOR | NAME | EFFECT |
|--------------|------------------------|--|
| > | Greater than | True if the value on the left is greater than the value on the right |
| < | Less than | True if the value on the left is less than the value on the right |
| >= | Greater than or equals | True if the value on the left is greater than or equal to the value on the right |
| <= | Less than or equals | True if the value on the left is less than or equal to the value on the right |
| == | Equals | True if the value on the left is equal to the value on the right |
| != | Not equals | True if the value on the left is not equal to the value on the right |

A program can use comparison operators in an expression to set a Boolean value.

[Click here to view code image](#)

```
it_is_time_to_get_up = hour>6
```

This Python statement will set the variable `it_is_time_to_get_up` to the value `True` if the value in hours is greater than 6 and `False` if the value in hours is not greater than 6. If this seems hard to understand, try reading the statement and listen to how it sounds. “`it_is_time_to_get_up` equals hour greater than six” is a good explanation of the action of this statement.



CODE ANALYSIS

Examining comparison operators

Question: How does the equality operator work?

Answer: The equality operator evaluates to `True` if the two operands hold the same value.

```
>>> 1==1  
True
```

The equality operator can be used to compare strings and Boolean values too.

```
>>> 'Rob'=='Rob'  
True  
>>> True==True
```

True

Question: How do I remember which relational operator is which?

Answer: When I was learning to program, I associated the < in the <= operator with the letter L, which reminded me that <= means “less than or equal to.”

Question: Can we apply relational operators between other types of expressions?

Answer: Yes, we can. If a relational operator is applied between two string operands, it uses an alphabetic comparison to determine the order. You can test this by using the Python Shell:

```
>>> 'Alice'<'Brian'  
True
```

The Python Shell returned **True** because the name Alice appears alphabetically before Brian.



WHAT COULD GO WRONG

Equality and floating-point values

In [Chapter 4](#), we saw that a floating-point number is sometimes only an approximation of the real number value our program is using. In other words, some numbers are not stored precisely.

This approximation of real number values can lead to serious problems when we write programs that test to see whether two variables hold the same floating-point values. Consider the following Python statements, which I’ve typed into the Python Shell:

```
>>> x = 0.3  
>>> y = 0.1+0.2
```

These statements create two variables, `x` and `y`, which are both set to the value `0.3`.

The variable `x` has the value `0.3` directly assigned, whereas the second variable, `y`, gets the value `0.3` as the result of a calculation that works out the result of `0.1 + 0.2`.

What do you think we will see if we test the two variables for equality?

```
>>> x == y  
False
```

This expression uses the equality operator (`==`) which will produce a result of `True` if its two operands hold the same value. However, Python decides that `x` and `y` are different because the variable `x` holds the value `0.3` and the variable `y` holds the value `0.3000000000000004`. This illustrates a problem with program code that compares floating-point values to determine whether they are equal. The tiny floating-point errors mean values we think are the same do not always evaluate that way.

If a program needs to compare two floating-point values for equality, the best approach is to decide they are equal if they differ by only a very small amount. If you don't do this, you might find that your programs don't behave as you might expect.

The date and time values returned from the Python `time` functions are supplied as integers so you can test these for equality without problems. However, when you use functions from other libraries, you might want to check what type of result is returned. You can use the Python built-in function `type` to ask an object its data type.

```
>>> mystery = 1
>>> type(mystery)
<class 'int'>
```

The `type` function returns an object that represents the type of the item supplied as a parameter. In the above Python Shell example, you can see that the `mystery` variable is `int` (which is not very surprising because it was created from an `int`). You might want to experiment with other variables to see their data types.

Boolean operations

At the moment, my test to determine whether it is time to get up is controlled only by the hour value of the time.

[Click here to view code image](#)

```
it_is_time_to_get_up = hour>6
```

The above statement sets the value of `it_is_time_to_get_up` to `True` if the hour is greater than 6 (that is, from seven o'clock onward), but we might want to get up at seven thirty. To be able to do this, we need a way of testing for a time when the hour is greater than 6 and the minute is greater than 29. Python provides three logical operators we can use to work with logical values. Perhaps they can help solve this problem.

| OPERATOR | EFFECT |
|----------|--|
| not | Evaluates to True if the operand it is working on is False Evaluates to False if the operand it is working on is True |
| and | Evaluates to True if the left-hand value and the right-hand value are True |

or

Evaluates to True if the left-hand value or the right-hand value is True

The **and** operator is applied between two Booleans value and returns **True** if both values are **True**. An **or** operator applied between two Boolean values returns **True** if at least one of the values is **True**. The third operator, **not**, can be used to invert a Boolean value.



CODE ANALYSIS

Boolean operators

We can investigate the behavior of Boolean operators by using the Python Shell part of IDLE. We can just type in expressions and see how they evaluate.

Question: What does the following expression evaluate to?

```
>>> not True
```

Answer: The effect of **not** is to invert a Boolean value, turning the **True** into a **False**.

```
>>> not True  
False
```

Question: How about this expression?

```
>>> True and True
```

Answer: The operands each side of the `and` are `True`, so the result evaluates to `True`.

```
>>> True and True  
True
```

Question: What about the following expression?

```
>>> True and False
```

Answer: Because both sides (operands) of the `and` operator need to be `True` for the result to be `True`, you shouldn't be surprised to see a result of `False`.

```
>>> True and False  
False
```

Question: What about the following expression?

```
>>> True or False
```

Answer: Because only one side of an `or` operator needs to be `True` for the result to be `True`, the expression evaluates to `True`.

```
>>> True or False  
True
```

Question: So far, the examples have used only Boolean values. What happens when we start to combine Boolean and numeric values?

```
>>> True and 1
```

Answer: Python is quite happy to use and combine logical and numeric values. The above combination would not return `True`, however. Instead, it will return `1`.

```
>>> True and 1  
1
```

This looks a bit confusing, but it gives us an insight into how Python evaluates our logical expression. Python will start at the beginning of a logical expression and then work along the expression until it finds the first value it can use to determine the result of the expression. It will then return that value.

In the above expression, when Python sees that the left-hand operand is `True`, it says to itself “Aha. The value of the `and` expression is now determined by the right-hand value. If the right-hand value is `True`, the result is `True`. If the right-hand value is `False`, the result is `False`.” So, the expression simply returns the right-hand operand. We can test this behavior by reversing the order of the operands:

```
>>> 1 and True  
True
```

We know that any value other than `0` is `True`, so Python will return the right-hand operand, which in this case is `True`. We can see this behavior with the `or` operation, too. Python only looks at the operands of a logical operator until it can determine whether the result is `True` or `False`.

```
>>> 1 or False
```

```
1  
>>> 0 or True  
True
```

You might wish to experiment with other values to confirm that you understand what is happening.

We could try to make an alarm that triggers after 7:30 by writing the following statement:

[Click here to view code image](#)

```
it_is_time_to_get_up = hour>6 and minute>29
```

The `and` operator is applied between two Boolean values and returns `True` if both expressions evaluate to true. The above statement would set the variable `it_is_time_to_get_up` to `True` if the value in `hour` is greater than 6 and the value in `minute` is greater than 29, which you might think is what we want. However, this statement is incorrect. We can discover the bug by designing some tests:

| HOUR | MINUTE | REQUIRED RESULT | OBSERVED RESULT |
|------|--------|-----------------|-----------------|
| 6 | 0 | False | False |
| 7 | 29 | False | False |
| 7 | 30 | True | True |
| 8 | 0 | True | False |

The table shows four times, along with the required result (what should happen) and the observed result (what does happen). One of the times has been observed to work incorrectly. When the time is 8:00, the value of

`it_is_time_to_get_up` is set to `False`, which is wrong.

The condition we're using evaluates to `True` if the hour value is greater than 6 and the minute value is greater than 29. This means that the condition evaluates to `False` for any minute value less than 29, meaning it is `False` at 8:00.

To fix the problem, we need to develop a slightly more complex test:

[Click here to view code image](#)

```
it_is_time_to_get_up = (hour>7) or (hour==7 and  
minute>29)
```

I've added parentheses to show how the two tests are combined by the `or` operator. If the value of `hour` is greater than 7 we don't care what the value of `minute` is. If the `hour` is equal to 7, we need to test that the `minute` is greater than 29. If you try the values in the table with the above statement, you'll find that it works correctly.

This illustrates an important point when designing code intended to perform logic like this. You need to design tests that you can use to ensure that the program will do what you want.

The `if` construction

Suppose I want to make a program that will display a message telling me if it's time to get out of bed. We can use the Boolean value we just created to control the execution of programs by using Python's `if` construction.

```

# EG5-02 Simple Alarm Clock

import time -> Import the time library

current_time = time.localtime() -> Get the time value

hour = current_time.tm_hour -> Get the hour value
minute = current_time.tm_min

it_is_time_to_get_up = (hour>7) or (hour==7 and minute>29) -> Set the flag value

if it_is_time_to_get_up:
    print('IT IS TIME TO GET UP') -> Is the flag value true?
                                            If so, print the message

```

The `if` construction starts with a statement that begins with the word `if`. This is followed by a Boolean value called the *condition*. The condition is followed by a colon (:). The colon is very important. It marks the start of the statements controlled by the `if` condition.

The indented statements (as is the `print` statement above) are performed only if the Boolean value is `True`. In other words, looking at the code above, the statement that prints ‘IT IS TIME TO GET UP’ is obeyed only if `it_is_time_to_get_up` is `True`.

Conditions in Python

The condition determines whether the statements controlled by the condition are obeyed. If the condition is `True`, the statements controlled by the `if` construction are obeyed. If the condition is `False`, the Python program will skip the statements controlled by the `if` construction. The condition in the test above is easy to understand because the value is `True` or `False`.

[Click here to view code image](#)

```

if it_is_time_to_get_up:
    print('IT IS TIME TO GET UP')

```

We can simplify the program using the result of a logical expression as the

condition:

[Click here to view code image](#)

```
if (hour>7) or (hour==7 and minute>29):  
    print('IT IS TIME TO GET UP')
```

The behavior of the statement is the same, but there's no need for the intermediate Boolean value.

Combine Python statements into a “suite”

Printing a message on the screen is all very well, but printed output alone won't get me out of bed. I seem to need a large message and a loud noise to get me going in the morning. Fortunately, we can use the snaps functions to display messages and play sounds. We first saw the snaps functions at the end of [Chapter 3](#). We can use these prewritten functions to create graphical displays and play sounds. I've created a suitably noisy siren sound sample we can use to make an alarm clock that would wake even the deepest sleeper.

When the alarm goes off, I want the program to display a large message, play the siren sound, and then wait while the sound playback is completed. In other words, I want the program to perform three statements.

```
# EG5-03 Siren Alarm Clock
```

```
import time Import the time library  
import snaps Import the snaps library  
current_time = time.localtime() Get the time  
hour = current_time.tm_hour Get the hour  
minute = current_time.tm_min  
  
if (hour>7) or (hour==7 and minute>29):  
    snaps.display_message('TIME TO GET UP')  
    snaps.play_sound('siren.wav')  
    # pause the program to give the sound time to play  
    time.sleep(10)
```

All these statements are controlled by the condition

An `if` condition can control any number of statements indented underneath it. The program above displays a message, plays the siren sound, and then sleeps for 10 seconds while the sound is played.

Once I've completed writing the statements to be performed when it's time to get up, I can continue writing statements that will always be performed. I indicate this by writing the code without an indent in the left-hand margin.

```
# EG5-04 Alarm clock with time display
```

```
import time  
  
current_time = time.localtime()  
  
hour = current_time.tm_hour  
minute = current_time.tm_min  
  
if (hour>7) or (hour==7 and minute>29):  
    print('IT IS TIME TO GET UP')  
    print('RISE AND SHINE')  
    print('THE EARLY BIRD GETS THE WORM')  
    print('The time is', hour, ':', minute)
```

This statement is always performed

The preceding program prints three inspiring messages when it is time to get up. The last statement, which prints out the time, is always obeyed, whether or not

it's time to get up. Many programming languages (C++, Java, JavaScript, and C#) use specific characters to mark the start and end of a block of statements controlled by an `if` condition. Python doesn't work that way. A statement in the left margin will be obeyed if the program ever reaches it. Indented statements are controlled by the conditions above them. The good news about this approach is that it forces you to arrange your program so that it is quite easy to understand which statements are controlled by which conditions.



WHAT COULD GO WRONG

Indented text can cause huge problems

Using indenting to show how various parts of a program are related is a good idea but can result in problems when you try to run your program. The first problem is that if you get the indenting even slightly wrong, your program will not run:

The screenshot shows a Python code editor window titled "EG5-04 Alarm clock with time display.py". The code contains a syntax error. A modal dialog box titled "SyntaxError" is displayed, showing the message "unexpected indent" with a red "X" icon. The code in the editor is:

```
# EG5-04 Alarm clock with time display

import time

current_time = time.localtime()

hour = current_time.tm_hour
minute = current_time.tm_min

if (hour>7) or (hour==7 and minute>29):
    print('IT IS TIME TO GET UP')
    print('RISE AND SHINE')
    print('THE EARLY BIRD GETS THE WORM')
print('The time is ', hour,':',minute)
```

The line "print('THE EARLY BIRD GETS THE WORM')" is highlighted in red, indicating it is causing the syntax error.

Above, you can see what happens when I try to run a program when one of the statements is misaligned. It is easy to see the mistake. The IDLE editor will highlight the faulty indent character. Now look at the following program and figure out what's wrong with it.

[Click here to view code image](#)

```
if (hour>7) or (hour==7 and minute>29):
    print('IT IS TIME TO GET UP')
    print('RISE AND SHINE')
```

```
print('THE EARLY BIRD GETS THE WORM')
print('The time is',hour,':',minute)
```

The answer is that the code looks fine. However, it turns out that the third line of this program was indented using the tab key (which moves the cursor along the line), and the others were indented by adding spaces. When printed on paper or viewed on a screen, these statements look identical, but Python can detect the differences and will refuse to run the program.

We've already seen that the IDLE editor does a good job of warning about indenting errors, but if you're given programs that have been edited by other programs you might find tab characters in some of the lines that cause errors. If Python is producing errors on a line that looks completely correct, check to see whether it was indented using spaces or the tab character.

The formal definition of the Python `if` construction looks a bit like this:

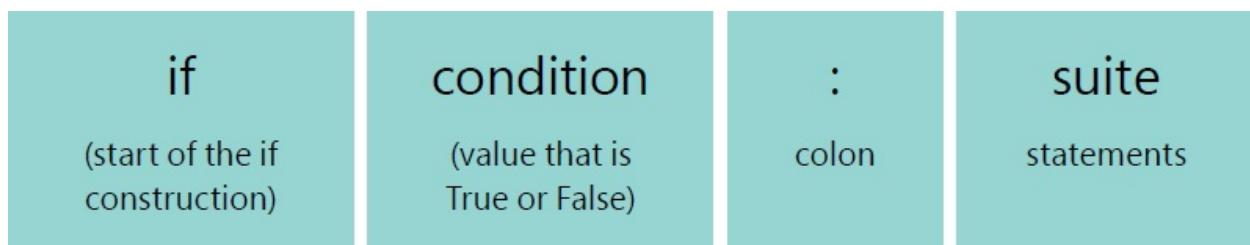


Figure 5-4 `if` condition

The item(s) following the colon in an `if` condition is called a *suite*. In my living room, I have a “three-piece suite” consisting of two chairs and a sofa. In Python, a suite can be one of two things:

- A set of indented statements as you have seen above, with nothing after the colon on the line with the `if` statement
- A set of statements on the same line as the `if` statement after the colon, with each statement separated from the next by a semicolon

If the second meaning of the word suite is confusing, consider this statement:

[Click here to view code image](#)

```
if hour > 6:print('IT IS TIME TO GET  
UP');print('THE EARLY BIRD GETS THE WORM')
```

In this example, both `print` statements would be obeyed if the value of `hour` is greater than 6. You can use this statement format when you only need to perform a few statements when a condition is `True`. Note that you can't combine these two types of suites:

[Click here to view code image](#)

```
if hour > 6:print('IT IS TIME TO GET UP');  
print('RISE AND SHINE')  
    print('THE EARLY BIRD GETS THE WORM')  
SyntaxError: unexpected indent
```

This arrangement of statements would not work. The Python suite is either on the line following the colon or on lines under the `if` statement.

Personally, I never use the format in which Python code is placed after the colon on the same line as the `if` condition. I find that the indented layout does a very good job of showing me when particular statements are executed.



CODE ANALYSIS

Conditional statement layout

Question: Can we work with conditional statements using the Python Shell?

Answer: Yes, you can. Open the Python Shell and type the following:

```
>>> if True:
```

When you press Enter at the end of this line, you will notice something interesting about the cursor. It doesn't go back to the left-hand margin. Instead, the cursor is indented. Add a `print` statement at this position and press Enter:

```
>>> if True:  
    print('True')
```

When you add the `print` statement, you'll notice two things. First, the `print` statement doesn't print anything. Secondly, you'll see that the cursor remains indented. So, add a second `print` statement:

[Click here to view code image](#)

```
>>> if True:  
    print('True')  
    print('Still true')
```

The second statement has no effect, and the cursor is indented. Nothing happens because Python is waiting for the end of the suite of statements controlled by the `if` statement. You can end the suite by entering an empty line:

[Click here to view code image](#)

```
>>> if True:  
    print('True')  
    print('Still true')
```

`True`
`Still true`

When you type in an empty line (that is, press the Enter key without entering any text), the suite is completed, and Python performs the condition. Note that it is not very sensible to type conditions into the Python Shell, but you will notice the same automatic indenting when you enter this program text into the IDLE text editor.

Question: How many spaces must you indent lines of a suite of Python statements controlled by an `if` condition?

Answer: The IDLE editor is configured to use four spaces for indenting, but you can change this in the options for the program. There is no “approved” amount of indentation that Python programs should have, but you *must* be consistent in the number of spaces you use. In other words, if the first line of the suite is indented four spaces, all the other lines must be indented four spaces. You can get into real trouble if you start to use the TAB character to indent your Python statements, because this might lead to a situation in which lines of your program appear to have the same indenting, but some of them contain tabs instead of spaces.

Add an `else` part to an `if` construction

Many programs want to perform one action if a condition is `True` and another action if the condition is `False`. You can add an `else` to the `if` construction that identifies a statement to be performed if the condition is `False`.

```
# EG5-05 Simple Alarm Clock with else
```

```
if (hour>7) or (hour==7 and minute>29): Start of the if construction
    print('IT IS TIME TO GET UP')
else: Start of the else statements
    print('Go back to bed')
```

This program displays a different message depending on the time of day that the user runs the program. Before 7:30 a.m., it displays, “Go back to bed”. After 7:30 a.m., the program displays, “IT IS TIME TO GET UP”. The `else` is followed by another “suite” of statements.



CODE ANALYSIS

If constructions

Question: Must an `if` construction have an `else` part?

Answer: No. They are very useful sometimes, but their usefulness depends on the problem the program is trying to solve.

Question: What happens if a condition is never `True`?

Answer: If a condition is never `True`, the statement controlled by the condition never gets to run.

Compare strings in programs

A program can use the equality operators to compare two strings. We can use this to make a program that will recognize us by our name:

```
# EG5-06 Broken Greeter  
name = input('Enter your name: ')  
  
if name == 'Rob':  
    print('Hello, Oh great one')
```

Read in the name

Compare the text entered with 'Rob'

If you tried to show off by using this program, you might have a problem, depending on how you type your name. The equality test regards uppercase and lowercase characters as different. In other words, if you enter the string “ROB,” you will not see the flattering greeting.

As a way around this, you can ask any string to provide the uppercase version of itself. A string value provides an `upper()` method that returns a version of a string in which the lowercase letters are replaced by uppercase characters. You can think of a method as a function that an object can be made to perform for you.

```
# EG5-07 Uppercase Greeter  
name = input('Enter your name: ')  
  
if name.upper() == 'ROB':  
    print('Hello, Oh great one')
```

Convert the name to uppercase before testing

This version of the program will work whether the user enters “rob,” “Rob,” or “ROB.” Whenever you write a program that accepts string input, you must decide how the program should behave if the user enters case-sensitive text.

There is also a string method called `lower()` that we can use to convert uppercase letters in a string to lowercase.



CODE ANALYSIS

Methods and functions

Question: How do the `lower()` and `upper()` methods work?

Answer: Everything in Python is an object that can provide a set of *methods*. Methods are called in the same way as functions such as `input` and `print`. Later in the book, we'll discover how we can design our objects and give them methods.

Question: Why do we have to put the parentheses on the end of `lower()`?

Answer: An effective way to explain this is to leave the parentheses off, and see what happens:

[Click here to view code image](#)

```
>>> name = 'Rob'  
>>> name.upper  
<built-in method upper of str object at  
0x0000021CDA0FE880>
```

Leaving out the parentheses doesn't cause a program to fail. But instead of the method running, we get a description of the `upper` attribute of the `name` object.

We've seen attributes before when we extracted the hour value out of current date and time by using `t.hour`.

Some attributes can be called as a method. Python knows that a method is being called because method calls have *arguments*. Arguments are values enclosed in parentheses after the method name.

When we call the `print` function, we give it arguments that tell the function what to print. We can call the `print` function with an empty set of arguments to print a blank line. In the case of the `upper()` method, there's no need to pass any arguments to the call, but we do need to add the parentheses so that Python knows to call a method.

```
>>> name = 'Rob'  
>>> name.upper()  
ROB
```

Question: What's the difference between a function and a method?

Answer: Programs use methods and functions in the same way. The only difference is where they are created. Functions are behaviors not associated with any particular object. You can think of them as “always there.” They don’t need an object to exist. Functions you’ve used already include `print` and `input`.

Methods are behaviors that are attributes of objects. The `upper()` method allows a string to do something for us, so it is packaged as part of the string object.

Nesting `if` conditions

Python lets a program put one condition inside another. Perhaps we might want to add a password test to make sure an important person really is who they say they are.

```
# EG5-08 Greeter with password  
  
name = input('Enter your name: ')  
  
if name.upper() == 'ROB':  
    password = input('Enter the password: ')  
    if password == 'secret':  
        print('Hello, Oh great one')  
    else:  
        print('Begone. Imposter')
```



Obeyed if the name is Rob

Obeyed if the name is Rob and the password is correct

This version of the greeter program asks for a password if the name Rob is entered. The second `if` condition (the one that tests the password) is *nested* inside the first one. You can tell that the `else` element applies to the second `if`

condition by the way it is indented.

The following code has the same arrangement of `if` and `else` elements, but this time `else` is paired with the *outer* condition, which means that the suite it controls runs if the name entered is not ‘Rob’.

```
# EG5-11 Greeter with outer else

name = input('Enter your name: ')

if name.upper() == 'ROB':
    password = input('Enter the password: ')
    if password == 'secret':
        print('Hello, Oh great one')
else:
    print('You are not Rob. Shame.'')
```

Obeyed if the name is Rob

Obeyed if the name is Rob
and the password is correct

Working with logic

Writing code that makes logical decisions like this is one of the hardest parts of learning to program. If you think it’s like solving a logic puzzle, you’re right, because that is just what you are doing. The best advice I can give is that you write down what you want to do in English and then work on converting the text into logical expressions. For example, “I want to pay overtime when the hours worked are more than 40 or the day is a Saturday.” Even after many years of programming, I still must sometimes resort to writing things out. Once I’ve written some code that I think will work, I then test it by trying some values and observing the outcomes. Creating a test plan, as we did earlier for the alarm clock, is also a good idea.



MAKE SOMETHING HAPPEN

Make an advanced alarm clock

There are lots of ways we can improve the alarm clock. The

information returned by the `localtime` function includes the date and the day of the week. You could make an alarm clock that tests the day of the week value and allows you to sleep in on weekends. You could use the sound playback feature of snaps to make an alarm clock that plays a suitable fanfare on the morning of your birthday.

Use decisions to make an application

Now that you know how to make decisions in your programs, you can start to make more useful software. Let's say your next-door neighbor is the owner of a theme park and has a job for you. Some rides at the theme park are restricted to people by age, and he wants to install some computers around his theme park so that people can find out which rides they may go on. He needs some software for the computers, and he's offering a season pass to the park if you can come up with the goods, which is a very tempting proposition. He provides you with the following information about the rides at his park:

| RIDE NAME | MINIMUM AGE REQUIREMENT |
|---|--|
| Scenic River Cruise | None |
| Carnival Carousel | At least 3 years old |
| Jungle Adventure Water Splash | At least 6 years old |
| Downhill Mountain Run | At least 12 years old |
| The Regurgitator (a super scary roller coaster) | Must be at least 12 years old and less than 70 |

You discuss with him the design of the program. Users will select the ride they want to go on. The program will ask for their ages and then display a message indicating whether they can go on this ride. For now, your customer is happy with text input, but later he would like to move to a graphical user interface with touch buttons. (We'll learn how to create graphical user interfaces in [Part 3](#) of

the book.)

Design the user interface

You and your customer discuss how the program should be used and come up with the following text-based user interface:

[Click here to view code image](#)

```
Welcome to our Theme Park
```

```
These are the available rides:
```

- ```
1. Scenic River Cruise
2. Carnival Carousel
3. Jungle Adventure Water
 Splash
4. Downhill Mountain Run
5. The Regurgitator
```

```
Please enter the ride number you want: 1
You have selected the Scenic River Cruise
There are no age limits for this ride
```

Here, the user has selected the Scenic River Cruise and has been told there are no age limits for this ride.



### PROGRAMMER'S POINT

## Design the user interface with the customer

You might think that an interface like this would be simple to design and that the customer will have no strong opinions on how the user interface looks and functions. I've found this to be wrong. I've had the awful experience of proudly showing my finished solution to a customer only to be told that it was "Not what they wanted" and "Hard to use." I now understand that this was my fault. Rather than showing only my finished design, I should have created the design with the customer. That would have saved me a lot of work.

## Implement a user interface

Now that we have our design, we can create the Python code to implement it. This is the code that I came up with:

[Click here to view code image](#)

```
EG5-10 Ride Selector Start

print('''Welcome to our Theme Park

These are the available rides:

1. Scenic River Cruise
2. Carnival Carousel
3. Jungle Adventure Water Splash
4. Downhill Mountain Run 5. The Regurgitator
''')

ride_number_text = input('Please enter the ride
number you want: ')
ride_number = int(ride_number_text)

if ride_number == 1:
 print('You have selected the Scenic River
Cruise')
 print('There are no age limits for this ride')
```

This code handles the case when a user selects the Scenic River Cruise ride. With the information you received from the theme park's owner, you know that if the user selects any ride other than the Scenic River Cruise, the program must obtain the age of the user. You can add an `else` statement to the code to meet this need. Remember that the `if` construction will perform the `else` part if the ride selection is anything other than Scenic River Cruise, which is what we want. Here, I've put a comment in the code at the point where the program needs to read the age value.

[Click here to view code image](#)

```
if ride_number == 1:
 print('You have selected the Scenic River
Cruise')
 print('There are no age limits for this ride')
else:
 # We need to get the age of the user
```

The selection of the Scenic River Cruise is easy to handle because anyone can go on this ride. For the other rides, the program must obtain the age of the person who wants to ride. The program can just use the same sequence as was used to read the ride number.

[Click here to view code image](#)

```
if ride_number == 1:
 print('You have selected the Scenic River
Cruise')
 print('There are no age limits for this ride')
else:
 # We need to get the age of the user
 age_text = input('Please enter your age: ')
 age = int(age_text)
```

## Testing user input

Once our funfair program knows the age of the user, it can decide whether the user can go on the ride. The program has two items of data with which to work:

- The selected ride, held in a variable named `ride_number`.
- The age of the user, held in an integer variable named `age`.

The program can use a sequence of `if...else` constructions to make its decision:

[Click here to view code image](#)

```
if ride_number == 2:
 print('You have selected the Carnival
Carousel')
 if age >= 3 :
 print('You can go on the ride.')
 else:
 print('Sorry. You are too young.')
```

These conditions work for the Carnival Carousel. The first `if` statement is used to determine the ride selected. The inner `if` statement makes the appropriate decision based on the age of the user. Notice that I've added a comment to make it clear for which ride this code is used.

Now that you have code that works for the Carnival Carousel, you can use it as the basis for the code that handles some of the other rides. To make the program work correctly for the Jungle Adventure Water Splash, you need to check for a different ride name and confirm or reject the user based on a different age value. Remember that for this ride, a visitor must be at least six years old. You could check whether the visitor is older than five (`age > 5`), or use the greater-than-or-equal-to operator when you test for the value of `age`.

[Click here to view code image](#)

```
if ride_number == 3:
 print('You have selected the Jungle Adventure
Water Splash')
 if age >= 6:
 print('You can go on the ride.')
 else:
 print('Sorry. You are too young.')
```

You can implement the Downhill Mountain Run very easily by using the same pattern as for the previous two rides. But the final ride, The Regurgitator, is the most difficult. The ride is so extreme that the owner of the theme park is concerned for the health of older people who use it and has added a maximum age restriction as well as a minimum age. The program must test for users who

are older than 70 as well as those who are younger than 12. We must design a sequence of conditions to deal with this situation.

## Complete the program

The code that deals with The Regurgitator is the most complex piece of the program that we've had to write so far. To make sense of how it needs to work, you need to know more about the way that `if` constructions are used in programs. Consider the following code:

[Click here to view code image](#)

```
if ride_number == 5:
 print('You have selected The Regurgitator')
```

The print statement tells the user what is going on, and also makes it clear that all the statements we add inside this suite will run only if the selected ride is The Regurgitator. In other words, there is no need for any statement in that suite to ask the question, “Is the selected ride The Regurgitator?” because the statements are run only if this is the case. The decisions leading up to a statement in a program determine the context in which that statement will run. I like to add comments to clarify the context:

[Click here to view code image](#)

```
if ride_number == 5:
 print('You have selected The Regurgitator ')
 if age >= 12:
 # Age is not too low
 if age > 70:
 # Age is too high
 print('Sorry. You are too old.')
 else:
 # Age is in the correct range
 print('You can go on the ride.')
 else:
 # Age is too low
```

```
print('Sorry. You are too young.')
```

These comments make the program slightly longer, but they also make it a lot clearer. This code is the complete construction that deals with The Regurgitator. The best way to work out what it does is to work through each statement in turn with a value for the user's age. You can download and run the entire program from the sample “**EG5-13 Complete Ride Selector.**”

## Input snaps

The snaps framework is a prebuilt set of functions that we can use in our programs. We've already seen snaps functions used to display images, display text, play sounds, and even get the current weather conditions. Now we'll discover a new snaps function that we can use to make a really good-looking ride selection program.

The `get_string` function works the same way as the Python `input` function. It takes a string of text as a prompt, displays the prompt, and then allows the user to type in text.

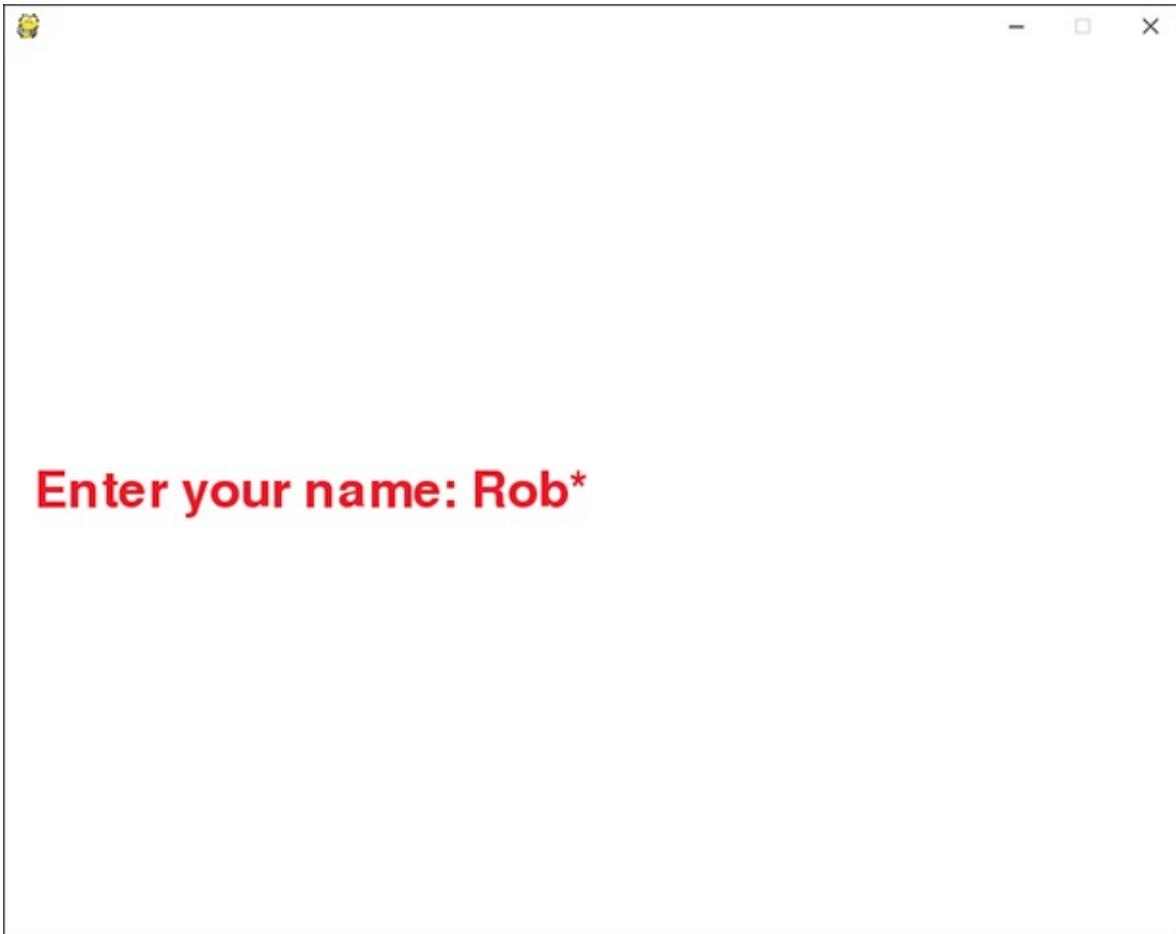
[Click here to view code image](#)

```
EG5-12 Snaps get_string function

import snaps

name = snaps.get_string('Enter your name: ')
snaps.display_message('Hello ' + name)
```

The program above is a snaps version of the greeter program that we've already written. [Figure 5-5](#) shows the display produced.



**Figure 5-5** Reading a string

We can use optional arguments to control where the input prompt appears on the screen.

[Click here to view code image](#)

```
EG5-13 Theme Park Snaps Display

import snaps

snaps.display_image('themepark.png')

prompt = '''These are the rides that are available

1: Scenic River Cruise
2: Carnival Carousel
```

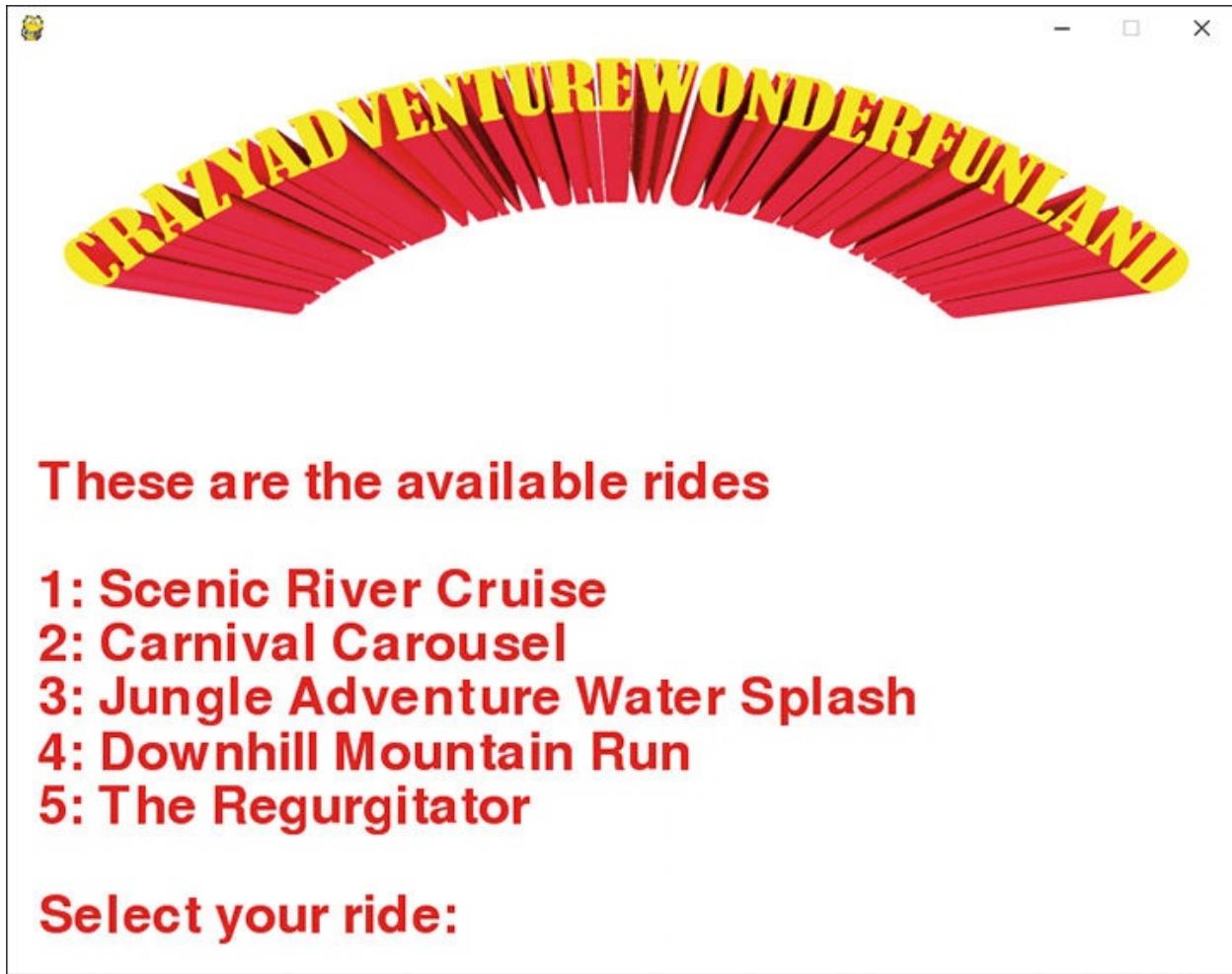
```
3: Jungle Adventure Water Splash
4: Downhill Mountain Run
5: The Regurgitator

Select your ride: '''

ride_number_text =
 snaps.get_string(prompt, vert='bottom',
 max_line_length=3)

confirm='Ride ' + ride_number_text
snaps.display_message(confirm)
```

The program displays a background image and then uses the `get_string` function to request the ride number from the user. **Figure 5-6** shows the effect of the `vert` argument when it is used to display the input at the bottom of the screen. The `max_line_length` argument is used to set the maximum length for the string to be read. The above program restricts the string length to three characters.



**Figure 5-6** Snaps theme park ride selector

A screenshot of a Snaps ride selector interface. It features a red rocket ship icon on the left and the text "MAKE SOMETHING HAPPEN" in bold black capital letters to its right. Below this, the text "Snaps ride selector" is displayed. A descriptive paragraph follows, explaining how to use the EG5-13 Theme Park Snaps Display sample program as a starting point for a ride selector program, mentioning the potential to add custom graphics, sound effects, and even weather helpers.

You can use the **EG5-13 Theme Park Snaps Display** sample program as the starting point for a very good ride selector program. You could even design custom graphics for each ride and display them when the ride is selected. You could even add suitable sound effects for each ride.

Weather helper

At the end of the last chapter we discovered some snaps functions that let us write programs that can determine the current weather conditions. You could use these in `if` constructions to create a program that would remind us to wrap up warm or look out for ice.

[Click here to view code image](#)

```
EG5-14 Weather helper

import snaps

temp = snaps.get_weather_temp(latitude=47.61,
longitude=-122.33)

print('The temperature is:', temp)

if temp < 40:
 print('Wear a coat - it is cold out
there')
elif temp > 70:
 print('Remember to wear sunscreen')
```

This is a very simple weather helper program that reminds me to wear a coat when it's cold and wear sunscreen when it's hot. You can improve this by adding images and sounds for different weather conditions.

## Fortune teller

The `randint` function from the random library can be used in `if` constructions to make programs that perform in a way that appears random.

[Click here to view code image](#)

```
import random
if random.randint(1,6)<4:
 print('You will meet a tall, dark
```

```
stranger')
else:
 print('You will not meet anyone at all')
```

The `if` construction tests the value produced by a call to the `randint` method that will produce a value in the range 1 to 6. If the returned value is less than 4, the program tells the user that he or she will meet a tall, dark stranger. Otherwise, it tells the user he or she will not meet anyone interesting. You could use a sequence of such conditions to make a fun fortune teller program. You could also create some graphical images to go along with the program predictions.

## What you have learned

In this chapter, you've learned that Python can work with Boolean values as well as numbers and text. Boolean values can be either `True` or `False`, and we can use comparison (for example, less than) operators to test strings and numbers and generate Boolean results. You've also discovered that the Python `if` construction lets you change a program's behavior depending on the data it is given. This works by executing one or more Python statements (called a "suite") only if a given Boolean value is `True`. This allows a programmer to make software that can respond to input in a useful way.

We also learned that there are three additional logical operators, `and`, `or` and `not`. The `and` operator evaluates to `True` if both of its operands are `True`, whereas the `or` operator evaluates to `True` only if both its operands are `True`.

We discovered how to create useful programs, which can work with logical conditions to create code that makes decisions. The best way to do this is to transcribe an English description of the decision into Python conditional statements. For example, "If it is Saturday or Sunday and it is after 9:00 a.m., I must get out of bed" could be converted into a single logical expression that makes that decision.

Here are some questions that you might like to ponder about making decisions in programs:

## **Does the use of Boolean values mean that a program will always do the same thing given the same data inputs?**

It is very important that, given the same inputs, the computer does the same thing each time. If the computer starts to behave inconsistently, this makes it much less useful. When we want random behavior from a computer (for example, when writing a fortune teller program), we have to obtain values that are explicitly random and make decisions based on those. Nobody wants a “moody” computer that changes its mind (although, of course, it might be fun to try to program one using random numbers).

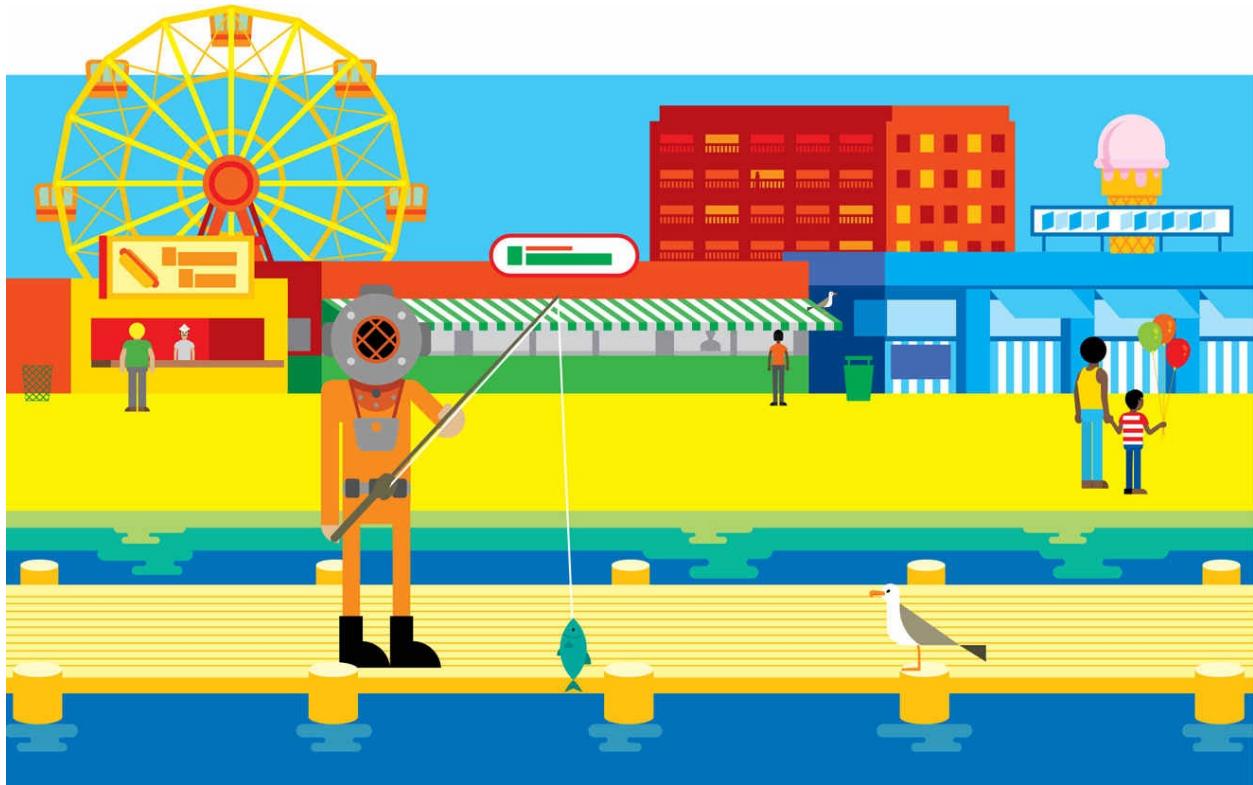
## **Will the computer always do the right thing when we write programs that make decisions?**

It would be great if we could guarantee that the computer will always do the right thing. However, the computer is only ever as good as the program it is running. If something happens that the program was not expecting, it might respond incorrectly. For example, if a program was working out cooking time for a bowl of soup, and the user entered ten servings rather than one, the program would set the cooking time to be far too long (and probably burn down the kitchen in the process). In that situation, you can blame the user (because they input the wrong data), but there should probably also be a test in the program that checks to see if the value entered is sensible. If the cooker can't hold more than three servings, it would seem sensible to perform a test that limits the input to three. When you write a program, you need to “second guess” what the user might do and create decisions that make your program behave sensibly in each situation.

## **Is there a limit to how many `if` conditions you can nest inside each other?**

No. The Python compiler will be quite happy to let you put 100 `if` statements in a row (although you would have a problem editing them in IDLE). If you find yourself doing this, you might want to step back from the problem a bit and see if there is a better way of attacking the problem.

## 6 Repeating actions with loops



### What you will learn

You know that programs work by receiving input data, doing something with it, and then producing an output. You also know how to use conditional statements to make programs respond to input in a sensible way. A program can test for values and other conditions and change what it does depending on what it finds. Up to this point, all the programs we've written run through their statements and then stop. However, you often need to make a program repeat a sequence of statements. For example, if a user enters an invalid value, you want the program to reject that value and repeat the sequence to request another value. Programs use a *loop* construction to repeat a series of actions.

All programming languages use loop constructions. In fact, you might be encouraged to know that once you've learned how to create loops, you can use

them to repeat statements you already know, which is enough to create just about every program that's ever been written.

There are essentially two loop constructions available to Python programmers. One is the `while` construction; the other is the `for` construction. We'll discuss the `for` construction later in the chapter.

The `while` construction

The `for` loop construction

Make a digital clock using snaps

What you have learned

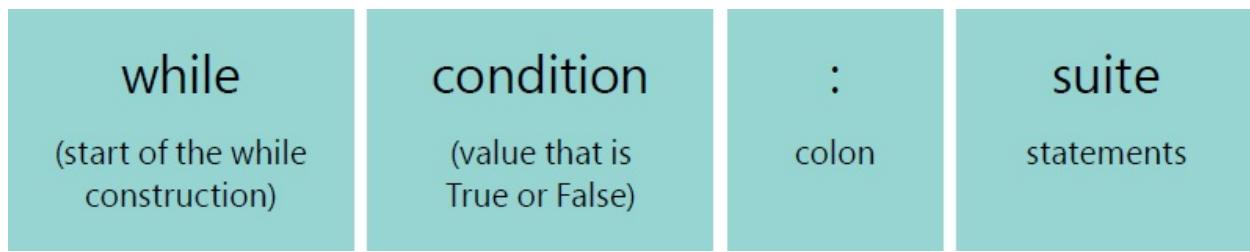
## The `while` construction

The `while` construction allows a program to repeat one or more statements while a given condition is True. There's no point in learning about a program construction without considering where you would use it, so let's see how we can use the `while` construction to improve the theme park ride selector that we worked on in the previous chapter.

### Repeat a sequence of statements using `while`

In Chapter 5, we wrote a ride selector program to help theme park visitors discover whether they can go on a particular ride. The user selects a ride, and the program asks for his or her age (if the ride has an age restriction) and then tells the would-be rider whether he or she can go on that ride.

The program we created works very well but currently works only once. The program ends after it has told the user whether he or she can go on the selected ride. We need a way to make the program repeat so that it can deal with multiple users. We do this using a `while` construction. **Figure 6-1** shows the anatomy of the `while` construction in a Python program.



**Figure 6-1** The `while` construction

The `while` construction looks remarkably like the `if` construction we saw in [Chapter 5](#). However, the constructions differ in their behavior. An `if` construction performs the statements it controls if a condition is True. A `while` construction performs the statements while a condition is True. The sequence of operations of a `while` construction is as follows:

1. The Python engine sees the word “while” and starts performing a `while` construction.
2. The Python engine tests the condition after the word `while`. If the condition is found to be `False`, the statements controlled by the `while` are skipped and the Python engine moves to the first statement after the `while` construction.
3. If the condition is found to be `True`, the statements controlled by the `while` construction are performed.
4. The Python engine loops back to step 2.

The Python engine only moves past a `while` construction when the `while` condition becomes `False`. If you find this confusing, consider that we humans do this kind of thing frequently:

- `while there_are_dishes_to_wash: wash a dish`
- `while there_are_exams_to_grade: grade an exam`
- `while the_kettle_has_not_boiled: wait a minute`

The `while` construction is just the way that we make a Python program repeat a behavior as many times as we need.



## CODE ANALYSIS

Investigating the `while` construction We can use the Python Shell in IDLE to investigate how the `while` construction works: **Question:** Can we use Boolean values to control a `while` construction?

**Answer:** Yes, we can. Open the Python Shell and type the following:

```
>>> while False:
```

When you press Enter at the end of this line, you'll notice that the cursor doesn't go back to the left-hand margin. You saw this behavior in [Chapter 5](#) when you investigated the `if` construction. The indented statements form the suite of statements controlled by the `while` construction. Add a `print` statement and press Enter:

```
>>> while False:
 print('Loop')
```

After you've entered the `print` statement above, enter an empty line statement. Of course, nothing will be printed because the statements controlled by the `while` are performed only if the condition is True. In this respect, the `while` construction is exactly like the `if` construction.

**Question:** Can I make a loop that goes on forever?

**Answer:** It turns out that this is very easy. Enter the following statements followed by an empty line:

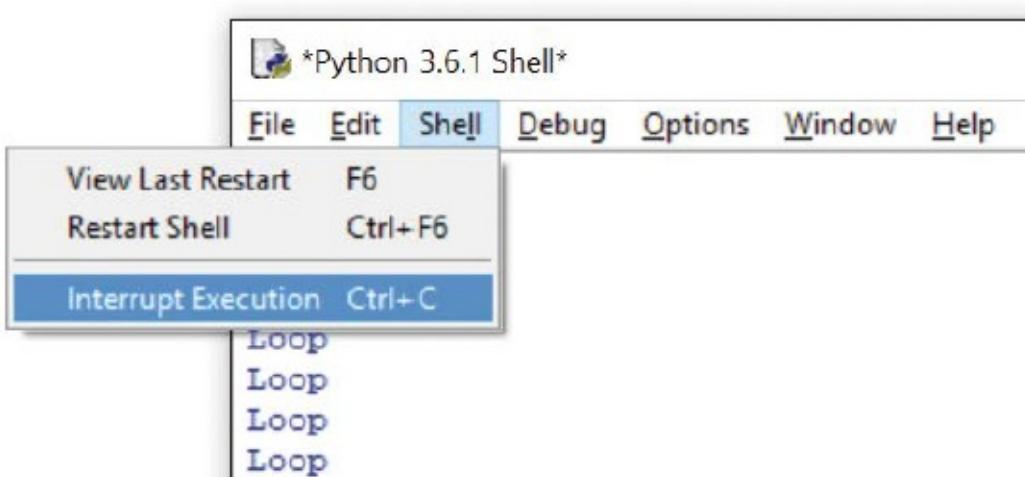
```
>>> while True:
 print('Loop')
```

The word “Loop” will be printed repeatedly. This program will never stop. You might think that Python would refuse to run a program that would never end, but this turns out not to be the case. Fortunately, Python provides a way of stopping a running program in its tracks. Hold down the Ctrl key and press C. This sends a “break” command to the Python Shell telling it to interrupt the running program and stop.

[Click here to view code image](#)

```
>>> while True:
 print('Loop')
Loop
Loop
Loop
LoopTraceback (most recent call last):
 File "<pyshell#8>", line 2, in <module>
 print('Loop')
KeyboardInterrupt
>>>
```

Depending on what else your computer is doing, you might need to press Ctrl+C several times. If the program refuses to stop, make sure your cursor is in the IDLE Python Shell when you enter the command. If you can't get the command to work, you can use the Interrupt Execution option from the Shell menu, as shown below.



You can use the Interrupt Execution option to interrupt any running program that is stuck in a loop. You might wonder why an infinite loop program didn't completely halt your computer. After all, this usually happens in science fiction shows when one of the crew saves the day by giving some rogue hardware a tough puzzle to solve. Fortunately, modern operating systems, such as Windows, macOS, and Linux are very good at sharing computer time with multiple active programs.

**Question:** Will the following program ever print out the message, **Outside loop**?

[Click here to view code image](#)

```
while True:
 print('Inside loop')
print('Outside loop')
```

**Answer:** No. The **while** construction will never end, so the final **print** statement will never be reached.

**Question:** Will the following program ever print out the message **Inside loop**? Will it print **Outside loop**?

[Click here to view code image](#)

```
while False:
 print('Inside loop')
print('Outside loop')
```

**Answer:** The **while** construction will never execute the statements controlled by it, which means it will never print **Inside loop**. However, the program will print **Outside loop** once it has moved past the statements controlled by the **while** construction.

**Question:** What will the following program print?

[Click here to view code image](#)

```
EG6-01 Loop with boolean flag
flag = True
while flag:
 print('Inside loop')
 flag = False
print('Outside loop')
```

**Answer:** The only way to work out what this program will do is for us to run it just like the Python engine would. The variable `flag` is of type Boolean, and it is set to `True` by the `flag = True` statement at the start of the program. The first time that `flag` is tested by the `while` construction, it is set to `True`, and the statements in the construction are obeyed. The first statement controlled by the `while` prints the message `Inside loop`. The second statement sets the value of the variable `flag` to `False`. When the `while` condition tests the value of `flag` the second time, it finds that it is `False`, so the loop ends.

You might think there's something dangerous about changing the value of the variable that controls the execution of a `while` construction, but it is a very common programming technique.

**Question:** What will the following program print?

[Click here to view code image](#)

```
flag = True
while flag:
 print('Inside loop')
 Flag = False
print('Outside loop')
```

**Answer:** You might think this is a stupid question. The code looks the same as in the previous question. However, there is a crucial difference between the two code samples that would cause this program to print `Inside loop` indefinitely. If you look very carefully, you'll see that we have set the value of a variable called `Flag` to `False` inside the loop. The intention is to stop

the loop, but this will actually create a new Boolean variable called `Flag`, which is set to `False`. The variable controlling the loop is called `flag`, and the two variables are different.

This illustrates a really important point with Python programs: Tiny errors can produce significant effects. We've seen that the Python engine sometimes tells us that we have typed something incorrectly, but in this case, no error was detected. I don't know of a magic solution for this, but if one of your loops suddenly seems to run without stopping, at least you now know one possible reason.

**Question:** What will the following program print?

[Click here to view code image](#)

```
EG6-02 Loop with counter
count = 0
while count < 5:
 print('Inside loop')
 count = count+1
print('Outside loop')
```

**Answer:** When you understand the answer to this question, you can call yourself a “while construction ninja.” The `count` variable is initially set at 0, and the condition controlling the `while` construction will become False as soon as the `count` variable is no longer less than 5. Each time round the loop, the `count` variable is made larger by one. So, the program should print out the message, `Inside loop`, five times before the `while` construction completes.

If you're having trouble understanding what's going on here, think of the `while` construction as acting like a doorman at a nightclub. The doorman is given a piece of paper to keep track of the number of people coming into the nightclub. Initially, the piece of paper has the number 0 written on it. Each time the doorman is about to let someone in, he checks this number against the room limit (which is 5). If there are fewer than five people, he lets the person in and increases the count. Otherwise, he displays a “Club Full” sign and goes off to do other things.

You might like to consider answers to the following additional questions:  
How do I change the program to print out **Inside loop** 100 times?

Can you think of ways that the loop could fail, bearing in mind the problems that we had with a **Flag** variable?



## MAKE SOMETHING HAPPEN

Create a looping selection program You can use a **while** to make a theme park ride selector that runs continuously. All you need to do is put all of the statements that implement the theme park behavior into a **while True** construction.

Create a looping countdown program Above, you saw how you could use a counter to count up to 5. Now, we'll create a countdown program that counts down from ten to zero over ten seconds. You could use it to manage a rocket launch, should you ever decide to open a launch pad. You can use the example program above as a starting point. You can use the **sleep** function from the time library to pause the program for a second between each count.

Once you've made a counter like this, you can use it for just about anything, including cooking or exercising.

## Handling invalid user entry

Currently, the theme park ride selector doesn't detect whether the user enters an invalid ride number. If the user types in a ride number outside the range 1–5, the program doesn't fail, but it doesn't report an error either. The owner of the theme park is concerned that users might think that they can go on a ride because the program hasn't told them they can't. So, we need to write some extra code to detect invalid ride numbers.



### PROGRAMMER'S POINT

Great programmers think defensively I'm a big fan of what is sometimes called “defensive programming.” I like to view my programs as little castles that I'm trying to defend from people trying to do them harm. Before I let anyone into my castle, I must try hard to make sure that I can trust them not to wreck the place. In the case of the castle, this means having a strong drawbridge and some guards always on duty to ask, “Friend or Foe?” of people trying to gain entry. For the theme park ride selector, this means checking to ensure that incoming data is sensible. The ride number must be in the range 1 to 5, and the user age must be in a sensible range, too. We call this part of program development data validation.

From experience, I've discovered that if anyone manages to make your program do stupid things by typing in silly values, they'll look clever, and you, the programmer, will look stupid. I'm very keen to avoid looking stupid. When I write code, I'm very careful to make sure that invalid inputs will not cause problems for the program.

We've been talking about a computer as something that takes in data, does something with it, and

then produces more data output. You can think of data validation as a kind of filter between raw data and the correct values with which your program needs to work.

Keep in mind that the process of data validation (which you must do to avoid looking stupid) makes your programs much larger. We'll find that adding the code needed to validate input values will probably double the size of the code. Whenever you're thinking about the amount of work you'll need to put in to create a program, make sure you allow for this extra effort.

If I were giving data validation instructions to a human about acceptable ride values, I'd probably say something like "If the ride number is greater than five or less than one, then the number is invalid." We could write this into our program as the following conditional statement:

```
if ride_number < 1 or ride_number > 5:
 print('Invalid ride number')
```

If either condition is true, the ride number is wrong

Print out an error message if the ride number is wrong

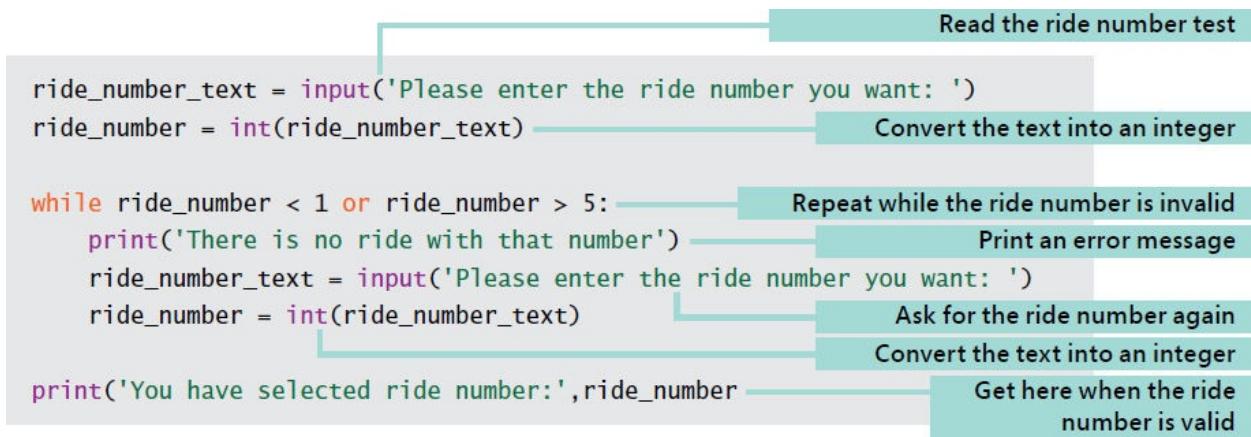
The variable `ride_number` holds the number of the ride that the user entered. The `if` construction contains two tests combined using a logical `or`. In other words, if one or the other condition is `True`, the result evaluated by the logical operator is also True. The `print` statement displays a message for the user of the program.

If this code looks strange to you, go back to [Chapter 5](#) and refresh your understanding of `if` constructions and Boolean expressions. To understand what the `if` condition in the above code does, try reading the first statement out loud. Remember that the `<` operator means "less than" and the `>` operator means "greater than."

The statement allows a program to detect when a ride number is invalid, but it doesn't provide any repeating behavior. We want the program to repeatedly request ride numbers until a number in the correct range is entered.

## Make a loop to validate input

You can use the above `if` construction to create a test that validates a ride number, but you really want the program to evaluate the number, determine if the number is or is not in the acceptable range, and if unacceptable, ask the user for another number. In other words, you want to repeat a `read` operation while the number given is wrong.



The statements that print the error message and read in a new value are controlled by the condition in the `while` construction. If an invalid ride number is entered, the user will be asked to input another. This means that the only way the program can reach the `print` statement following the `while` construction is by the user entering a valid ride number.



## MAKE SOMETHING HAPPEN

Add ride number validation to the theme park ride selector You can use the above code to add ride number validation to the theme park ride selector. Remember that the `while` construction above must be added after the `ride_number` value has been read by the program.



## CODE ANALYSIS

When good loops go bad You can learn a bit more about how loops work by looking at another example. At first glance, the following code might seem identical to the previous code. And if you run this program, it seems to work fine. If you give a valid age, the program prints **Thank you for entering your age.**

[Click here to view code image](#)

```
1. # We need to get the age of the user
2. age_text = input('Please enter your age:
')
3. age = int(age_text)
4. while age < 1 and age > 95:
5. # repeat this code while the age is
invalid
6. print('This age is not valid')
7. age_text = input('Please enter your
age: ')
8. age = int(age_text)
9. # when we get here, we have a valid age
value
10. print('Thank you for entering your age')
```

**Question:** What is the fault in the program?

**Answer:** The fault is in line 4. The logical expression used here is slightly different from the one used earlier. This expression says, “while age is less than one and age is greater than 95.” When you read it out loud, it sounds silly. How can a number be less than one and greater than 95? No such value exists. But it turns out that the compiler is quite happy to compile a program that contains a mistake like this.

**Question:** What will the fault cause the program to do?

**Answer:** Because there is no number that is both less than 1 and greater than 95, the expression controlling the `while` construction can never be `True`, which means that the condition will *never* repeat. In other words, it will regard every age value as correct. This is very dangerous, because if you don't test the program with invalid values, you will never notice this problem.

**Question:** How do I fix this?

**Answer:** It looks like the programmer's intention was to create a `while` construction to reject ages less than 1 or greater than 95. We can get this behavior by replacing the `and` in line 4 with an `or`.

[Click here to view code image](#)

```
4. while age < 1 or age > 95:
```



## WHAT COULD GO WRONG

Always test failure behaviors along with successful ones. This is a very important point to consider when you write software. You must test the code you write that is supposed to deal with errors. Software engineers talk about the “happy path” through a program in which the user enters the right values, the network connection works, there’s enough space on the disk drive, and the printer doesn’t jam. When programmers write software,

they tend to focus on this happy path without giving much thought to the depressingly large number of ways a program can go wrong. However, this is a dangerous way to write code. A great programmer will proactively look for things that can go wrong, build in the code to deal with the error conditions, and then—crucially—take the trouble to test to ensure that this code works. This is another aspect of the “defensive programming” approach.

In the case of this age program, I’d insist on testing it with the ages 0, 1, 50, 94, 95, and 96. These values should let me ensure that the invalid ages (0 and 101) are rejected and that all the other ages (including the values on the boundaries) are accepted. In fact, I would find a way that I could test the code automatically (you’ll learn about this later in the book) so that I can perform the tests at regular intervals. The best test values are the ones around the boundaries. So, if I’m writing a program that’s supposed to reject any numbers larger than 40, I’d test it with the numbers 39, 40, and 41



## MAKE SOMETHING HAPPEN

Add validation to the theme park age input Now you can add age validation to the theme park ride selector. The theme park owner has told you that the minimum age for anyone going on a ride at the theme park is 1 year, and the maximum age is 95.

Use these values in your program.

## Detect invalid number entry using exceptions

The theme park ride selector is almost ready for release, but there is still one problem that must be addressed. We have made the program reject values outside the correct range, but the program will still fail if the user doesn't type in a valid number value.

```
Please enter the ride number you want: three
Traceback (most recent call last):
 File "C:/Users/Rob/RideSelector.py", line 16, in <module>
 ride_number = int(ride_number_text)
ValueError: invalid literal for int() with base 10: three
```

The user has typed the text `three`, and the program has crashed with a red error message. The `int` function is not clever enough to work out that the word `three` means a number. The method just sees this as a string that doesn't contain any numeric digits.

This is a big problem for the `int` function, which doesn't want to return a number if it can't make sense of the string it has been given. The `int` function would much rather the program be made to stop because there is no point in continuing if the incoming data is not valid. In Python terms, the `int` function raises an exception. Raising an exception is the computer equivalent of kicking over the table when you're losing a game of chess. The current program is abandoned.

You might think this is a bit extreme. All the user did was enter text when a number was expected. Why such a fuss? The answer is very important. When a program goes wrong, it is crucial that the user knows as soon as possible. There is only one thing worse than a broken program, and that is a broken program that the user doesn't know is broken. It is one thing for a word processor to give you an error when you try to save a file; it is quite another (and much worse) thing for a word processor to leave you thinking the file was saved when it wasn't.

If `int` just kept going—perhaps returning a value of `-1000000`, which means,

“I didn’t understand the text that was entered,”—there would be potential for huge problems. If a programmer just assumed that `int` always returns a value, it would cause programs to be given invalid data, which would result in incorrect outputs. The only sensible thing that `int` can do in this situation is to raise an exception.

In other words, exceptions are how Python programs deal with errors in situations where it would be dangerous to continue running. Exceptions provide a way for a program to be stopped from doing the wrong thing.

You can think of an exception as a description of why something didn’t work. We will see quite a few different exceptions as we gain more experience writing Python programs. When the Python engine detects an exception, it prints a brief description of the position the program had reached, followed by details of the exception that was raised. If the `int` function is unable to convert a text string into an integer it raises a `ValueError` exception.

If we want our program to retain control when an exception is raised, we can add error handlers using a Python construction called `try`. Statements that might raise an exception are written after the keyword `try`. If any of the statements raise an exception, the execution of the program instantly moves to a block of code (the error handler) that deals with that exception. You can see how this works in the following program.

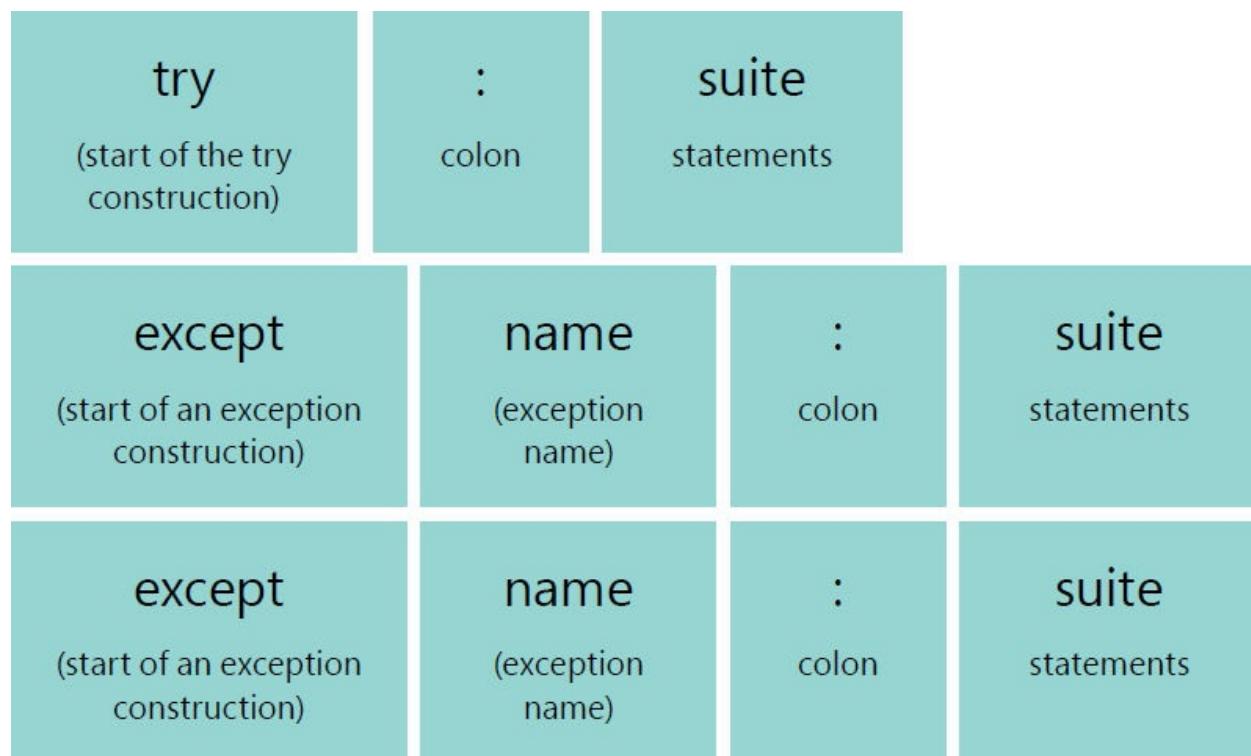
```
EG6-03 Catching exceptions
try: Start of the try construction
 ride_number_text = input('Please enter the ride number you want: ')
 ride_number = int(ride_number_text) Statements that might raise an exception
 print('You have entered', ride_number)
except ValueError: Start of an exception handler
 print('Invalid number') Statements that are performed if an exception is raised
```

In the above code, the start of the exception handler is marked by the `except` keyword, which is followed by the exception type that the handler will deal with. In the above example, the exception handler is dealing with the `ValueError` exception that would be raised by the `int` function if the user entered text that didn’t contain a valid number. The `ValueError` exception handler displays an appropriate message.

If the user enters a valid ride number value, the program will continue to the `print` statement after the `int` exception. If the `int` function raises an exception, the statements beyond the point at which the exception was raised are not obeyed.

If you are unclear about what's happening here, consider what we are trying to do. We know a user might enter text rather than a number, causing the `int` function to raise an exception because it can't convert text into a number. Our program needs a way of responding to this eventuality; the block of code after the `except` statement does just that.

**Figure 6-2** shows the layout of an exception construction, showing two exception handlers. However, a program could have lots of handlers or only one. You'll see how to handle multiple exceptions later in this chapter when we create a program that rejects invalid number text and prevents a user from being able to interrupt the program using Ctrl+C.



**Figure 6-2** An exception construction

## Exceptions and number reading

When an exception is raised, the flow of a program is interrupted and all statements following the exception will be ignored.

```
ride_number = int(ride_number_text)
print('You have entered', ride_number)
```

This statement may raise an exception  
This statement might never be reached

In the above code sample, there is no guarantee that the second statement will be obeyed. If the contents of `ride_number_text` cannot be converted into an integer, the `int` function will raise an exception that diverts the program before the `print` statement is reached. However, we really want the program to give users another chance to enter a value if they happen to enter an invalid number. We've already done this in the section “[Make a loop to validate input](#)” above. There we created a `while` construction that repeatedly asked for another value when the user enters values that are out of range (for example, a ride number value of 10). Now, we need to improve our program to deal with invalid number text.



## CODE ANALYSIS

Handling exceptions in loops We want to make a program that will perform a `while` construction as long as the user keeps typing in text that cannot be converted into a number. Look at the following code.

[Click here to view code image](#)

1. `#EG6-04 Handling invalid text`
2. `ride_number_valid = False #`  
create a flag value and set it to False
3. `while ride_number_valid == False: #`  
repeat while the flag is False

```
4. try: #
start of code that might throw exceptions
5. ride_number_text = input('Please
enter the ride number you want: ')
6. ride_number =
int(ride_number_text) # convert the text
into a number
7. ride_number_valid = True # if we
get here, we know the number is OK.
8. except ValueError: # the
handler for an invalid number
9. print('Invalid number text.
Please enter digits.') # display an error
10. # When we get here, we have a valid ride
number
11. print('You have selected ride',
ride_number)
```

**Question:** What is the purpose of the variable, `ride_number_valid`?

**Answer:** This variable is a flag or state variable. It is not concerned with managing data held by the program (that is for variables such as `ride_number`). Instead, this variable allows the program to track whether the user has entered a valid ride number. At line 2, the value of `ride_number_valid` is set to `False`. It is only set to `True` following the successful completion of the `int` function call on line 6. The statement on line 7 is obeyed only if no exception is raised by the call to `int`.

**Question:** How many times would you expect the `while` construction to loop when the program is used?

**Answer:** I'd expect the user to enter a valid number. If they do this, the `while` construction will be performed once. The second time around the loop, the value of `ride_number_valid` would be `True`, which would cause the loop to stop.

**Question:** Why don't we have to test `ride_number_valid` at line 10, to make sure that the ride number is valid?

**Answer:** We know that a `while` construction will continue while the condition controlling it is `True`. Line 10 is outside the `while` construction (we know this because it is not indented). We can be sure that the ride number must be valid at this line because the program would not have reached it otherwise.

## Handling multiple exceptions

Earlier in this chapter, you created a program that ran without stopping, and you had to use a control sequence (Ctrl+C) to stop the program running. A user can enter this key combination at any time to stop your program, which means that people using the ride selection program could cause it to fail.

[Click here to view code image](#)

```
Please enter the ride number you want:
Traceback (most recent call last):
 File "C:/Users/Rob/OneDrive/Begin to code
 Python/Part 1 Final/Ch 06 Loops/code
 /samples/#EG6-04 Handling invalid text.py", line
 5, in <module>
 ride_number_text = input('Please enter the ride
 number you want: ') # read in
 some text
KeyboardInterrupt
```

If the user presses Ctrl+C while entering a number, the program is interrupted, as you see above. One way to fix this would be not to use keyboards that contain the Ctrl key. However, we can also address this issue in the program.

The `try` construction can be followed by a number of `except` handlers, one for each exception that the program must handle. The exception caused by the user pressing Ctrl+C is called a `KeyboardInterrupt`. We can add a handler

for that as follows: [Click here to view code image](#)

```
1. #EG6-04 Handling invalid text
2. ride_number_valid = False # create a
flag value and set it to False
3. while ride_number_valid == False: # repeat
while the flag is False
4. try: # start of
code that might throw exceptions
5. ride_number_text = input('Please enter
the ride number you want: ')
6. ride_number = int(ride_number_text) #
(might raise exception)
7. ride_number_valid = True # if we get
here, we know the number is OK.
8. except ValueError: # the handler
for an invalid number
9. print('Invalid number text. Please
enter digits.') #
10. except KeyboardInterrupt: # the handler
for an invalid number
11. print('Please do not try to stop the
program.') #
12. # When we get here, we have a valid ride number
13. print('You have selected ride', ride_number)
```

There are now two `except` parts to the `try`—at lines 8 and 10. If either of these exceptions are raised, the program moves to the matching handler and prints the appropriate message.



## WHAT COULD GO WRONG

Plan for Failure It might seem depressing, but

when you write a program, you should always be thinking about how it could fail and what the program should do about it. Whenever you expect the user to type in some data, you should regard this as a potential failure point and make appropriate arrangements. Another important rule is that you should never catch exceptions in a way that hides errors. You could stop a program from generating exceptions by enclosing all the statements in a `try...except` construction, but this might mean that other programmers (and perhaps users) will think your program is working perfectly when it has actually gone wrong internally, which would be very bad.

In the case of the program above, we know exactly what will cause exceptions (the `int` function) and exactly why errors would occur (because the user has typed in something that is not a number or tried to stop the program). Armed with this knowledge, we can make the program behave sensibly in these situations.

## Break out of loops

The program to reject invalid entries works well, but we can simplify the construction slightly by using another feature of Python loops. The `break` statement tells a program to break out of a loop. As soon as Python finds a `break` statement, it stops running code in the loop and instead moves to the statement immediately following the loop program.

[Click here to view code image](#)

```
1. # EG6-06 Using break to exit loops
2. while True: # repeat until we break out of
the loop
3. try: # start of
code that might throw exceptions
4. ride_number_text = input('Please enter
the ride number you want: ')
5. ride_number = int(ride_number_text) # #
(might raise exception)
6. break # number OK -
break out of loop
7. except ValueError: # the handler
for an invalid number
8. print('Invalid number text. Please
enter digits.') # display error
9. # When we get here, we have a valid ride number
10. print('You have selected ride', ride_number)
```

The code above uses a `break` statement to stop reading numbers when the user has entered a correct number. We know that line 6 is reached only if the `int` function call on line 5 succeeds. This means that the program can break out of the loop. The statement that will be obeyed after line 6 is on line 9, because this is the first statement after the `while` construction.

You can control the execution of the `break` statement by using an `if` construction, so that a program can cause a loop to end “early.”

[Click here to view code image](#)

```
1. # EG6-07 Loop with condition ending early
2. count=0
3. while count<5:
4. print('Inside loop')
5. count = count+1
6. if count == 3:
7. break
8. print('Outside loop')
```

---

Line 6 tests the value of `count`, and the `break` statement is performed when `count` reaches the value `3`. This means that the `while` construction will not end when `count` reaches `5`. Instead it will end earlier when `count` reaches `3`.

### PROGRAMMER'S POINT

Don't use too many `break` statements A loop can contain many `break` statements, but I'm not keen on adding lots of breaks. Each time you add a `break` statement, it provides another way in which a loop can end. In the above loop, with only one `break` statement, I can be sure that the only way to reach statement 10 is for the `int` function to complete successfully. If there were lots of `break` statements scattered through the loop, this would not be the case, and I'd find the program much harder to understand.

## Return to the top of a loop with `continue`

Every now and then you'll write a program that needs to go back to the top of a loop and run the loop again. You'll do this when you have gone through the statements as much as needed for a particular pass around the loop. To return to the loop's beginning, Python provides the `continue` keyword, which says something along the lines of, "Please do not go any further this time around the loop. Go back to the top of the loop and then go around again if you are supposed to."

As an example, imagine that the theme park ride number 3, Jungle Adventure Water Splash, has sprung a leak and is now no longer available. If the user

selects ride number 3, you want the program to display a message and then ask the user to select another ride.

```
EG6-08 Ignore Ride 3
while True:
 ride_number_text = input('Please enter the ride number you want: ')
 ride_number = int(ride_number_text)
 if ride_number == 3:
 print('Sorry, this ride is not available')
 continue
 print('You have selected ride number:', ride_number)
```

If this statement is reached, the loop goes around again

If the user selects ride number 3, the `if` condition is triggered, which controls two statements. The first statement prints a message for the user, and the second performs the `continue`. This means that the final statement is reached only if the user has selected a ride number other than 3. Note that this is a greatly simplified version of the ride number entry program, but it does show how the `continue` statement is used.



## PROGRAMMER'S POINT

You won't use `continue` as often as you use `break`

There are quite a few situations in programs where the `break` keyword is useful. However, the `continue` keyword is used much less frequently. Don't feel like you aren't a true programmer if you don't find yourself using `continue` very often.

## Count a repeating loop

The loops in the theme park ride selector are quite simple. However, you can also make loops that repeat a number of times. We saw this earlier in the chapter when we examined the `while` loop. This is achieved by using a variable to count the number of times that the loop has been performed. The program can set the counter variable to a starting value, and each time around the loop, the variable can be updated until it reaches the limit that causes the loop to stop.

You might use this kind of loop to create a times-table tutor to help you (or someone else) with multiplication. You could use the loop to make this program print, “1 times 2 is 2, 2 times 2 is 4,” and so on. Here is the entire program. It uses a `while` loop, which produces each successive output as it runs.

[Click here to view code image](#)

```
EG6-09 Times Table Tutor
count = 1
times_value = 2
while count < 13:
 result = count * times_value
 print(count, 'times',
 times_value, 'equals', result)
 count = count + 1
```

There are two parts of this program that you really must understand. The first is the loop and the expression that controls it:

`while count < 13:`

The `while` loop is controlled by a logical expression that becomes `False` when the value of the `count` variable reaches the value 13 (this is because the value 13 is not less than 13; it is equal to 13).

The second important part of the program is the assignment statement that updates the counter:

`count = count + 1`

Each time this statement runs, it calculates the value of `count` plus one and then stores this in the variable `count`.



## CODE ANALYSIS

Counterintelligence Here is the times-table code with line numbers. Let's take a closer look: [Click here to view code image](#)

```
1. # EG6-09 Times Table Tutor
2. count = 1
3. times_value = 2
4. while count < 13:
5. result = count * times_value
6. print(count,'times',
times_value,'equals',result)
7. count = count + 1
```

**Question:** Which statement would you have to change if you wanted to generate the times table for 3 instead of 2?

**Answer:** You would change the assignment statement at line 3. If you set the variable `times_value` to 3, this will cause the times table to display multiples of 3.

**Question:** Which statement would you have to change if you wanted to generate up to the 24 times table, rather than stopping at 12?

**Answer:** You would change the end-point of the loop in line 4 so that the loop continues while the value of count is less than 25.

**Question:** What would happen to the program if I changed the statement at line 7 to the following statement?

```
count = count - 1
```

**Answer:** This statement makes the variable `count` smaller each time the statement is obeyed. The code in the times-table loop would calculate and display negative multiples, and the loop would never stop because the `count` variable would always be less than

13. At this point, the user would have to use Ctrl+C to stop the program.



## MAKE SOMETHING HAPPEN

Allow the user to select the times value You can improve the times-table program to make one that asks the user for a value to work with. You could allow the user to calculate multiples of 25 if you like, or you could use validation so that the only times tables that can be produced are in the range 2 to 12.

## The `for` loop construction

You have seen that you can manage perfectly well with `while` loop constructions. The times-table program works fine. However, the designers of Python invented a second kind of loop that was created to make it easy for programmers to work through lots of data. This is called the `for` loop (see [Figure 6-3](#)).

|                                        |                                                               |    |                                  |   |                     |
|----------------------------------------|---------------------------------------------------------------|----|----------------------------------|---|---------------------|
| for<br>(start of the for construction) | variable<br>function name<br>(variable controlled in the for) | in | items<br>(items to work through) | : | suite<br>statements |
|----------------------------------------|---------------------------------------------------------------|----|----------------------------------|---|---------------------|

**Figure 6-3** The `for` loop construction

In Python, a loop works on a collection of items, taking each item in turn. Each time the loop is run, the variable is set to the next item in the collection. In Python, it is very easy to create a collection of items. One type of Python collection is called a tuple. We'll discuss tuples in more detail in [Chapter 8](#). For now, perhaps the most important thing you need to know about a tuple is that it doesn't really matter how you pronounce it. You can say the word to rhyme with "supple" or with "scruple."

Tuples are very useful for making a quick collection of things that you want to treat as a single lump of data. To do this, simply write a sequence of values separated with commas and enclosed in brackets: [Click here to view code image](#)

```
names=
('Rob', 'Mary', 'David', 'Jenny', 'Chris', 'Imogen')
```

The variable `names` now contains six name strings. A Python program can use a `for` loop to work through these names and print each one:



The loop will go around once for each item in the tuple. The control variable (which in this loop is called `name`) will be set to the next name in the tuple each time the loop is run. In other words, the first time the loop is run, the value of `name` will be `Rob`. Next time around, the value will be `Mary`, and so on, to the end of the list.

[Click here to view code image](#)

```
EG6-10 Name printer
names=
('Rob', 'Mary', 'David', 'Jenny', 'Chris', 'Imogen')
for name in names:
 print(name)
```

This means that the above Python program will print the following:

```
Rob
Mary
David
Jenny
Chris
Imogen
```

You might think that I've been a bit silly using variables called `name` and `names` because it would be easy to get the two confused. However, I think this makes sense. The variable `names` denotes a plural, indicating that it contains multiple items. However, the variable `name` is singular, which indicates that it is one name in the list.

Python provides a function called `range` that will generate a sequence of numbers you can use if you want to make a program count through a succession of values.

```
EG6-11 Times Table Loops
times_value = 2
for count in range(1, 13):
 result = count * times_value
 print(count, 'times', times_value, 'equals', result)
```

Create a range of values from 1 to 12

This is the `for` loop-powered version of the times-table program we saw earlier. The `range` function above is given two arguments. The first is the lower limit of the range to produce; we want to start our times table at 1. The second is the exclusive upper limit of the range of values, meaning that this value is the first one that will be excluded from the list. In other words, the range will stop at 13, but will not contain 13.

The `for` loop construction can contain `break` and `continue` statements, which work in exactly the same way as they do in the `while` loop constructions. When a `continue` statement is performed in a `for` loop, it causes the loop to move the control variable onto the next item in the collection.



## CODE ANALYSIS

Loops, break, and continue You can improve your understanding of the way **break** and **continue** are used by looking at a few simple programs.

**Question:** What would the following code print?

[Click here to view code image](#)

```
1. # EG6-12 Code Analysis 1
2. for count in range(1, 13):
3. if count == 5:
4. break
5. print(count)
6. print('Finished')
```

**Answer:** It would print “1,2,3,4” and then “Finished.” When the value of **count** reaches 5, the logical expression in the **if** condition on line 3 would become **True** (because **count** is now equal to 5). The **break** statement would cause the program to exit the loop immediately and continue running the program at line 6. The program would not print the value 5 because it breaks before it reaches the statement that prints the value of **count**.

**Question:** What would the following code print?

[Click here to view code image](#)

```
1. # EG6-13 Code Analysis 2
2. for count in range(1, 13):
3. if count == 5:
```

```
4. continue
5. print(count)
6. print('Finished')
```

**Answer:** It would print “1,2,3,4,6,7,8,9,10,11,12”. Note that it would not print “5” because when the value of `count` is 5, the conditional statement at line 3 will cause the program to restart the loop, which means that the `print` method is not called for the value 5.

**Question:** What would the following code print?

[Click here to view code image](#)

```
1. # EG6-14 Code Analysis 3
2. for count in range(1, 13):
3. count = 13
4. print(count)
5. print('Finished')
```

**Answer:** You need to be careful with this example. If you’ve used other programming languages, you might expect the loop to end earlier because the value of `count` (which controls the loop) is being set to a value that should cause it to end. This is not what happens. In fact, the program will print out “13” twelve times. This is because each time around the loop, the value that has been extracted from the range is replaced with the value 13 before it is printed.

**Question:** Would the following program run forever?

[Click here to view code image](#)

```
1. # EG6-15 Code Analysis 4
2. while True:
3. break
4. print('Finished')
```

**Answer:** No. It is true that the logical expression controlling the `while` construction is set to `True`, which means always repeat the loop, but the content of the loop body contains a `break` statement that would cause the loop to exit.

**Question:** Would the following program print the message “Looping”?

[Click here to view code image](#)

```
1. # EG6-16 Code Analysis 5
2. while True:
3. continue
4. print('Looping')
```

**Answer:** No. The `continue` will send program execution back to the top of the `while` loop before the `print` statement is reached. The program will run forever, but it will never print the message.

**Question:** What would the following program do? Is it legal?

[Click here to view code image](#)

```
1. # EG6-17 Code Analysis 6
2. for letter in 'hello world':
3. print(letter)
```

**Answer:** This program would work. Python regards a string as a collection of letters. So, it is perfectly possible to use a string as the basis of a `for` loop like this. The program would print out each letter on a separate line.

h  
e  
l



### MAKE SOMETHING HAPPEN

Make a times-table quiz Reverse the behavior of the times-table program so that rather than printing out the times-table your program instead asks questions like “What is 6 times 4?” The user could enter their answer, and the program could compare it with the correct answer and keep score of how many correct answers are given. You could use a loop to make the program produce 12 “times-table” questions, and you could use random numbers so that the quiz is different every time.

## Make a digital clock using snaps

We can use a loop to repeatedly display the time using the function `draw_text` from the snaps library. We used this method in [Chapter 5](#) to create a program that displayed alarm messages. Now we can use it to display a digital clock that

updates every second.

```
EG6-18 Digital Clock

import time

import snaps

while True: Loop that never ends

 current_time = time.localtime() Get the time

 hour_string = str(current_time.tm_hour)
 minute_string = str(current_time.tm_min)
 second_string = str(current_time.tm_sec) Get strings containing the time information

 time_string = hour_string+':'+minute_string+':'+second_string Build the time string
 snaps.display_message(time_string) Display the time string
 time.sleep(1) Pause the program for a second
```

This program contains a loop that continuously reads the time from the clock and displays it. It also contains a call to the `sleep` function that will stop the program from updating the screen more than once a second.



## MAKE SOMETHING HAPPEN

Make a digital alarm clock You can use the code that we worked on in [Chapter 6](#) to create a digital clock that also sounds alarms and displays messages at particular times of the day. You could even display background images behind the time digits by using the `display_image` function from `snaps`.

# What you have learned

In this chapter, you learned how to create programs that contain statements that are repeated when the program runs. To learn this, you worked with the different looping constructions provided by Python.

The first of these, the `while` construction, repeats statements as long as the logical expression in the condition is `True`. If you simply put the Boolean value `True` as the condition, the loop will never end. In some cases, this is a reasonable thing to do because many programs (games, for example) contain behaviors that must be repeated while they run.

The second loop construction is completely different from the `while` construction. It has as much to do with collections of data as repeating code. The `for` loop is designed for situations in which the programmer wants to work through a collection of values and perform some action on each one. The collection of values can be held in a structure (we know about a data structure called a “tuple”), or we can use another Python function called `range` that can produce a defined sequence of values.

The Python language also provides a way for a program to break out of a loop by using the `break` keyword. This is useful if the program has reached a state where it is not meaningful for the loop to repeat. The `continue` keyword causes a loop to continue from the start of the loop statements, once the end condition has been tested.

Here are some points to ponder about loops.

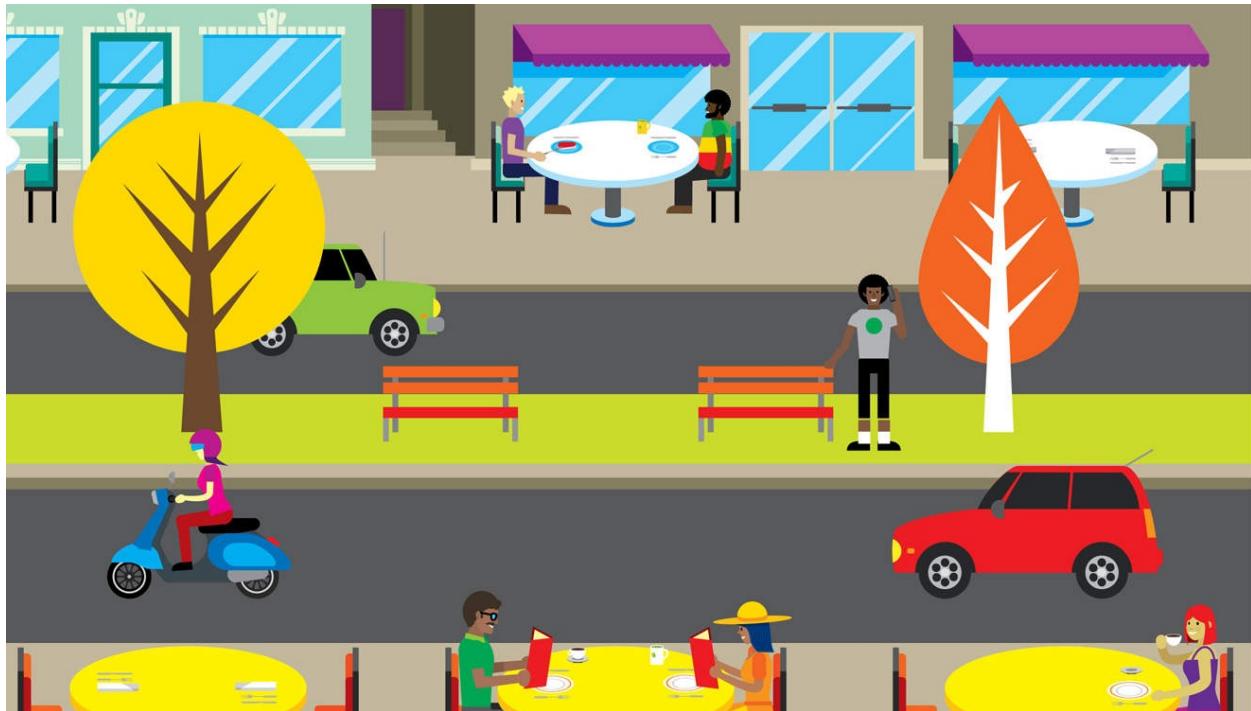
## **Do we really need loops?**

No. In theory, we could write every program using a sequence of statements and conditions. Loops could be “unrolled” into sections of repeated code. A loop that performs an action 10 times could be replaced by 10 copies of the code in the loop. Doing without loops would make programs much larger, but it would work.

## **Are loops dangerous?**

In a way. An “unrolled” loop is guaranteed to run through to completion. There is no way it can get stuck or execute the wrong number of times. However, we have seen several times that if we get the end conditions wrong, we can have loops that get stuck looping forever or loop the wrong number of times. In other words, using loops in a program introduces the potential for new kinds of errors. In some absolutely critical programs, such as those controlling aircraft or nuclear reactors, programmers sometimes avoid loops for just this reason.

# 7 Using functions to simplify programs



## What you will learn

Functions are an essential part of program design. You can use functions to break up a large solution into individual components and to create libraries of behaviors that can be used by your programs. Up to this point, our programs have worked with functions provided with Python (the `print` function, for example). In this chapter, you'll learn how to create and use functions of your own. You'll see how to give functions data to work on and how a program can receive results that a function returns. Functions make programs more concise and easier to manage.

[What makes a function?](#)

[Build reusable functions](#)

[What you have learned](#)

# What makes a function?

A function is a chunk of Python code that you name. When Python encounters a function, it takes the statements that describe what the function should do and stores them, ready for use later in the program. Let's look at a simple function.

```
def greeter():
 print('Hello')
```

This very simple function simply prints a message. Once the function has been defined, a program can use it. When a function is called, it performs the statements given when it was defined.

```
>>> greeter()
Hello
>>>
```

The **greeter** function doesn't do much, but you can create functions that contain many statements. Remember that your program must define the function before it can be called.



**MAKE SOMETHING HAPPEN**

## Investigating functions

We can use the Python Shell to investigate how functions are created and used. Open the IDLE command shell and enter the Python statements below at the **>>>** prompt. Press Enter at the end of the second statement.

```
>>> def greeter():
 print('Hello')
```

**Question:** Why did the program not perform the print action after you entered the `print` statement?

**Answer:** Currently, the statements you're entering are being stored as part of the `greeter` function. The function has not yet been called.

**Question:** How do I tell Python that I've finished entering the `greeter` function?

**Answer:** You do this in the same way that you tell the Python Shell you've finished entering the statements in a loop, or those controlled by an `if` construction: Enter an empty line.

[Click here to view code image](#)

```
>>> def greeter():
 print('Hello')
```

```
>>>
```

**Question:** How do I make a call to the `greeter` function?

**Answer:** You can call `greeter` in the same way you would call any other Python function. Remember to add an empty list of parameters so that Python knows a function is being called.

```
>>> greeter()
Hello
```

When a function runs, it performs all the statements it contains. In this

case, a single message is printed.

Now look at the following statements (and maybe even run them).

```
>>> x=greeter
>>> x()
Hello
```

This is probably the scariest piece of Python you've seen so far in this book. It shows you that functions are just like other variables. In the first statement, a variable called `x` is set to the value of `greeter`. Then, in the second statement, we call `x` as if it is a function. Python prints `Hello`, which is just what the `greeter` function does.

A program can store the “value” of a function in the same way as it can store a string or a floating-point value, simply by assigning it to a variable. This is a powerful feature that we’ll investigate in more detail in [Chapter 12](#).



## CODE ANALYSIS

### Program pathfinder

In Python, it’s common for one function to call another function. Let’s build our understanding of how functions work by looking at some code.

[Click here to view code image](#)

```
EG7-01 Pathfinder
def m2():
 print('the')
```

```
def m3():
 print('sat on')
 m2()

def m1():
 m2()
 print('cat')
 m3()
 print('mat')

m1()
```

**Question:** What will this program display when it runs?

**Answer:** The best way to figure this out is to work through the program one statement at a time, just like the computer does when it runs the program. Remember that when a function is complete, the program's execution continues at the statement following the function call. It turns out that the output from the program is exactly what you might expect:

```
the
cat
sat on
the
mat
```

**Question:** What happens if a function calls itself? For example, what if the `m1` function called `m1`?

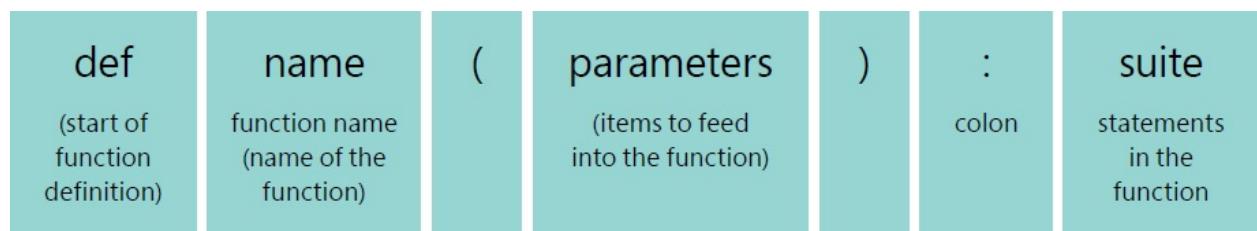
**Answer:** The effect is like what you see if you arrange two mirrors so that they face each other. In the mirrors, you see reflections going off into infinity. When the `m1` function calls itself, your computer will go very quiet for a few seconds and then produce an error message something like “`RecursionError: maximum recursion depth exceeded.`” Each time a function is called, the Python stores

the return address (the place it must go back to) in a special piece of memory called the “stack.” The idea is that when a running program reaches the end of a function, it grabs the most recently stored address of the top of the stack and returns to where that address points. This means that as functions are being called and returned, the stack grows and shrinks.

However, when a function calls itself, the Python engine repeatedly adds return addresses on the stack. Each time the function calls itself, another return address is added to the top of the stack. At some point, the Python system decides that this has gone on long enough and the program is halted.

Programmers have a name for a function that works by calling itself. They call it recursion. Recursions are occasionally useful in programs, particularly when the program is searching for values in large data structures. However, I’ve been programming for many years and have used recursion only a handful of times. I advise you to regard recursion as strong magic that you don’t need to use now (or hardly ever). Loops are usually your best bet for repeating blocks of code.

**Figure 7-1** shows the form of a Python function definition. We can work through each of these items in turn.



**Figure 7-1** Python function definition

The word **def** (short for “define”) tells Python that a function is being defined. Python will allocate space for the function and get ready to start storing function statements. The word **def** is followed by a single space and then the name of the function. We decide the name for the function in just the same way as we have chosen names for the variables we have created. Because a function is associated with an action, it’s a very good idea to make the name reflect this. I

give functions names in the form verb\_noun. The verb specifies the action the function will perform and the noun specifies the item it will work on. An example would be `display_menu`. Python has functions called `print` and `input`, and the names for both match their use.

After the name of the function, we have the parameters that are fed into the function. The parameters are separated by commas and enclosed within parentheses. Parameters provide a function with something to work on. So far, the functions we've created haven't had any parameters, so there has been nothing between the two parentheses. Finally, the definition contains a colon, followed by a suite of Python statements forming the body of the function.

## Give information to functions using parameters

The `greeter` function shows how functions can be used, but it isn't really that useful because it does the same thing each time it's called. To make a function truly useful, we need to give the function some data with which to work. You've already seen many functions that are used in this way. The `print` function accepts items to print. The `sleep` function accepts the length of time that the program should sleep. We can make a times-table function that accepts the times-table to produce.

```
def print_times_table(times_value):
 count = 1
 while count < 13:
 result = count * times_value
 print(count, 'times', times_value, 'equals', result)
 count = count + 1
```

A diagram illustrates the flow of data from a parameter to its use in a function. A blue box labeled "times\_value parameter" has a line pointing to the variable `times_value` in the first line of the function definition. Another line points from this variable to its use in the expression `result = count * times_value`. A callout box labeled "Use the value of `times_value` in the function" is positioned over this expression.

A program can use the `print_times_table` function any time it wants to print a times table. The function accepts a single argument, which is the times table to be produced.

```
print_times_table(5)
```

The statement above would call the `print_times_table` function and ask it

to print out the times table for 5. If we want to see the times table for 99, we just need to change the number that we pass to the function.

```
print_times_table(99)
```

## Arguments and parameters

From the title of this section, you might expect that we will have a difference of opinion, but in Python, the word *argument* has a particular meaning. In Python, the word argument means “that thing you give to the call of a function.”

```
print_times_table(7)
```

In the above statement, the argument is the value 7. So, when you hear the word argument you should think of the code that is making a call of the function.

In Python, the word *parameter* means “the name within the function that represents the argument.” The parameters in a function are specified in the function definition.

[Click here to view code image](#)

```
def print_times_table(times_value):
```

This is the definition of the `print_times_table` function. It specifies that the function has a single parameter, which is the name `times_value`. When the function is called, the value of the `times_value` parameter is set to whatever has been given as an argument to the function call. Statements within the function can use the parameter in the same way as they could use a variable with that name.



## CODE ANALYSIS

### Arguments and parameters

We can find out more about arguments and parameters by looking at the code that uses them.

**Question:** What would the following program do?

[Click here to view code image](#)

```
EG7-02 Times Table
def print_times_table(times_value):
 count = 1
 while count < 13:
 result = count * times_value
 print(count, 'times', times_value,
't>equals', result)
 count = count + 1

print_times_table(6)
```

**Answer:** The program prints out the times table for 6.

**Question:** What would happen if we changed the call of the `print_times_table` function to the one below that has a string as the argument? Would the program fail?

```
print_times_table('six')
```

**Answer:** The program doesn't fail, but it does something you might not expect.

[Click here to view code image](#)

```
1 times six equals six
2 times six equals sixsix
3 times six equals sixsixsix
4 times six equals sixsixsixsix
5 times six equals sixsixsixsixsix
6 times six equals sixsixsixsixsixsix
7 times six equals sixsixsixsixsixsix
8 times six equals sixsixsixsixsixsixsix
9 times six equals
sixsixsixsixsixsixsixsix
10 times six equals
sixsixsixsixsixsixsixsixsix
11 times six equals
sixsixsixsixsixsixsixsixsix
12 times six equals
sixsixsixsixsixsixsixsixsixsix
```

It turns out that Python is able to perform the multiplication operation between strings and numbers.

The statement below is the one in `print_times_table` that works out the result. It takes the `count` (which goes from 1 to 12) and multiplies it by the `times_value` (which is a parameter in the function).

[Click here to view code image](#)

```
result = count * times_value
```

Multiplying two numbers will produce a numeric result. Multiplying a string by a value will repeat the string the number of times equal to the product. This illustrates an important principle of the Python language. It will decide what to do based on the type of things with which it is working. This can lead to programs that

don't do what you might expect.

**Question:** How do we make the `print_times_table` function work with integer parameters only?

**Answer:** Before we decide to fix this problem, we must decide whether we need to fix it at all. If we're using this function in a program that's already checking the input values, then perhaps we don't have to worry about this issue.

If we do try to fix the problem, we must know what should happen. Should the function print a warning message? Should it stop the program? Deciding on an error strategy is an important part of program design, and you should do this in consultation with the customer (if you have one).

In this case, we might decide to be very strict and make the `print_times_table` function cause an error if it is not given an integer to work with. It turns out that Python has a built-in function called `isinstance` that a program can use to check whether a given item holds a particular type of data. The `isinstance` function accepts two arguments, the item to be tested and the type we are checking. It returns `True` if the item is of the given type, and `False` if not.

```
EG7-03 Safe Times Table
if isinstance(times_value,int)==False: _____ Test the type of the times_value
 raise Exception('print_times_table only works with integers') _____ Raise an
 exception if the
 type is not integer
```

The statements above show how we could use `isinstance` to cause an exception to be raised if the parameter to the function is invalid. The first statement performs the test to see if the function has been given an integer. The second statement is one we haven't seen before. The second statement raises an exception, which causes the program to stop with an error.

[Click here to view code image](#)

```
Traceback (most recent call last):
 File "C:/EG7-03 Safer Times Table.py", line
11, in <module>
 print_times_table('six')
 File "C:/ EG7-03 Safer Times Table.py",
line 4, in print_times_table
 raise Exception('print_times_table only
works with integers')
Exception: print_times_table only works with
integers
```

You can think of an **Exception** as a chunk of data that describes why something went wrong. When an exception is created, it is given a string that describes the error. The exception can be picked up in a **try** construction to allow a program to deal with errors, as we saw in the section “Exceptions and number reading” in Chapter 6. We’ll cover exceptions in detail later in the text.

## Multiple parameters in a function

A function can have multiple parameters. Currently, the **print\_times\_table** function always prints out 12 results, starting with 1 times the **times\_value** and ending with 12 times the **times\_value**. If we are printing out tables for mathematical geniuses, we might want to produce a times table that goes up to 20 times the input value. Alternatively, some people might prefer smaller tables that only go up as far as five times. We could write a different function for each of these table sizes, or we could make the function more flexible by making it accept the size of the times table as well as the number to multiply.

[Click here to view code image](#)

```
EG7-04 Two Parameter Times Table
def print_times_table(times_value, limit):
 count = 1
 while count < limit+1:
```

```
 result = times_value * count
 print(count, 'times', times_value,
'equals', result)
 count = count + 1
```

This version of the function has two parameters. The first parameter, `times_value`, is the number for which times table is desired; the second parameter is the `limit` for the table to be produced.

Now let's call the function.

[Click here to view code image](#)

```
print_times_table(6, 5)
```

The statement above would call the `print_times_table` function and ask for the times table for 6 up to 5 times 6.

[Click here to view code image](#)

```
1 times 6 equals 6
2 times 6 equals 12
3 times 6 equals 18
4 times 6 equals 24
5 times 6 equals 30
```

## Positional and keyword arguments

Consider the following function call.

[Click here to view code image](#)

```
print_times_table(12, 7)
```

The statement above makes a call of the `print_times_table` function, but you might be forgiven for wondering whether it prints out the times table for 12 or the times table for 7. You might need to go back to the original code to check the sequence in which the arguments (12 and 7) are mapped to the parameters (`times_value` and `limit`). Arguments mapped in this way are called *positional* arguments because the positions of the arguments given to the function and the parameters defined in the function determine which argument value maps to which parameter. In other words, the sample above would print the times table for 12, because the `times_value` parameter was given first in the original definition.

To make things easier for programmers, Python allows you to use keywords to identify the arguments to a function when you call it.

[Click here to view code image](#)

```
EG7-05 Keyword Arguments
print_times_table(times_value=12, limit=7)
```

If you use keyword arguments, you don't have to worry about getting the order of the arguments correct when you call functions.

[Click here to view code image](#)

```
print_times_table(limit=7, times_value=12)
```

This call of the `print_times_table` function will produce the same result as the previous one. I find keyword arguments very helpful. When I write a Python function that accepts more than one argument, I try hard to use keyword arguments for every call of that function.



## WHAT COULD GO WRONG

## Don't mix positional and keyword arguments

Python will let you mix positional arguments and keyword arguments in a call to a function. However, it can be hard to work out what is going on when you do this. I strongly suggest using either all positional arguments (if it is obvious what the arguments mean) or all keyword arguments.

## Default parameter values

When we created the first `print_times_table` function, we assumed that the limit of the times table to be produced was 12. In other words, the output would go from “1 times” up to “12 times.” Then we allowed the user to specify the limit. However, most users of our function will want to go up to a limit of 12 times. We can reflect this by providing a default value for the limit parameter.

```
EG7-06 Default parameters
def print_times_table(times_value, limit=12):
 count = 1
 while count < limit+1:
 result = times_value * count
 print(count, 'times', times_value, 'equals', result)
 count = count + 1
```

Default value for the limit parameter

Default in this context means, “If I leave this argument out, use this value.” So, users of the `print_times_table` function can still specify a different limit value and that will be used. However, if they omit the limit argument, the default value (12) will be used instead.

[Click here to view code image](#)

```
print_times_table(times_value=7)
```

The statement above would print out a times table for the value 7, and it would print up to 12 times 7.

The IDLE editor is able to find function definitions and help you fill in the argument values when you’re writing calls to the functions. In **Figure 7-2**, you can see what happens when I start to write a call of the `print_times_table` function.

## Interactive help and functions

```
print_times_table()
 (times_value, limit=12)
```

**Figure 7-2** IDLE function help

The text beneath the cursor is generated by the editor. When I typed in the name of the `print_times_table` function, IDLE found that function definition and read the parameter list. It will then display that information as you fill in the arguments. You see this happen for Python’s built-in functions, and it also works for functions that you create.



### PROGRAMMER'S POINT

## Why I use named arguments and default parameters

I love the named arguments and default parameters features of Python. They make programs clearer, and you don’t have to wonder what on earth a function actually does. Named arguments and default parameters also reduce the possibility of programmers getting the arguments confused, which means a programmer can provide a “standard” behavior for a function that is easy to modify.



### CODE ANALYSIS

## Parameters as values

When a function is called, the value of the argument is passed into the

function parameter. What exactly does this mean?

The following program contains a function (with the interesting name `what_would_I_do`) that accepts a single parameter. The function doesn't do much; it just sets the value of the parameter to 99. The function is then called using the value of a variable named `test` as an argument.

[Click here to view code image](#)

```
EG7-07 Parameters as values
def what_would_I_do(input_value):
 input_value = 99

test = 0
what_would_I_do(test)
print('The value of test is', test)
```

**Question:** What would this program print when it runs? 0 or 99?

**Answer:** When the code runs, Python follows this sequence:

1. Set the value of `test` to 0. (Remember, the program starts running at the first statement after the definition of the function.)
2. Call the `what_would_I_do` function, passing the *value* of `test` as an argument.
3. When the `what_would_I_do` function starts, the parameter called `input_value` is assigned the value `0`.
4. The `what_would_I_do` function sets the value of the parameter called `input` to `99`.
5. The `what_would_I_do` function now ends, and execution returns to the calling statements.
6. The value of `test` is printed.

Remember that an argument is the item (a variable) fed into the function. However, Python uses the value of that variable, not the variable itself. So, the value displayed by the program is `0`. In other words, the program prints:

```
The value of test is 0
```



## MAKE SOMETHING HAPPEN

### Creating a teletype printer

For some reason, output from a computer looks more impressive if it is printed slowly. We've seen how to use a `for` loop to work through the characters in a string, and we know about the `sleep` function in the time library, so it might be fun to create a function that will print out a string one character at a time with a delay between each character.

We could call the function `teletype_print`. I think it should have two parameters. The first parameter should be the string to be printed. The second parameter should be the time interval between the printing of each character. We could give a default value for the delay of 0.1 (a tenth of a second). This would make the definition of our function look like this:

[Click here to view code image](#)

```
def teletype_print(text, delay=0.1):
```

The function can use a `for` loop to go through each character in the input string, print the character, and then delay:

[Click here to view code image](#)

```
for ch in text:
 print(ch)
 time.sleep(delay)
```

This code looks like it might work, but in fact there is a problem. If we try to print out a word, we find that each character is printed on a separate line. If the program tries to print out `hello`, it will produce the following:

```
h
e
l
l
o
```

The word, `hello`, is printed with one letter on each line because the default behavior for the `print` function is to make a new line at the end of the print. However, we can use IDLE's Help feature to discover how to fix this problem.

```
print()
 print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

If we start writing a call to the `print` function and then pause, IDLE will show us the Help for the `print` function. The four items at the end of the Help are the ones that we're interested in. These items define four parameters and their default values.

The `sep` argument specifies the separator to be used between successive items printed by this `print` call. By default, a separator space is inserted between printed items. We can change the separator to a different string or, if we prefer, we can change the separator to an empty string, which will allow us to print multiple items with no separator.

The `end` argument tells the `print` function what to print at the end of the print action. As you can see from the Help information, the default setting for this argument is `\n`, which is the escape sequence for a new line. We can change this to an empty string to prevent the `print` function from moving to a new line after each printed item.

The `file` and `flush` parameters give a program a low-level control over how the `print` function behaves and where it sends its output. We don't need to change these arguments.

```
print(ch, end='')
```

Above, you can see the `print` statement with the additional argument that changes the end-of-line string to an empty string. This will print the character but won't create a new line.

Now, you can create your own teletype printer. Remember that you might need to print a blank line after the loop that prints out all the characters in the input string.

You can use this `print` function in some of the programs you've already written. It is especially impressive in the fortune teller program we discussed in [Chapter 5](#). You can also make the computer output seem even more human by making the delay between the characters slightly random (using the `randomInt` function from the `random` library we saw in [Chapter 3](#)) and by having longer delays when a space character is displayed.

## Return values from function calls

A function can return a value. You've seen this in many of the programs we've written. Here's an example:

[Click here to view code image](#)

```
name = input('Enter your name please : ')
```

This statement uses the `input` function. The function accepts an argument (the text prompt to be shown to the user) and returns a value (the string that the user enters). Now look at this function header:

[Click here to view code image](#)

```
def get_value(prompt, value_min, value_max):
```

The function is called `get_value`, and it has three parameters. The first parameter is the prompt to be displayed for the user—`prompt`. The second and third variables are the minimum (`value_min`) and maximum (`value_max`) values that the `get_value` parameter can return. A program would use this function as follows:

[Click here to view code image](#)

```
ride_number=get_value(prompt='Please enter the ride
number you want:',
value_min=1, value_max=5)
```

The above call to the `get_value` function could be used to allow the user to select a ride in the theme park ride selector program we created in [Chapter 4](#). The program could also use this function to read the age of a user. We would just need to change the prompt and the limits of the input value.

The idea here is that we'll take some software that we've already created (our number input and validation code) and package it up as a function so that we can use it many times in our programs.

```
def get_value(prompt, value_min, value_max):
{
 return 1;}
```

Function header for `get_value`

This version of the function always returns 1

This version of `get_value` is not very useful because it always returns the value `1`. However, it does show how `return` is used. At the start of a project, programmers frequently make “empty” functions that they fill in later.



## CODE ANALYSIS

### Functions and `return`

Let's take a look at how `return` is used in a function.

[Click here to view code image](#)

```
EG7-08 get_value investigation 1
def get_value(prompt, value_min, value_max):
 return 1
 return 2

ride_number=get_value(prompt='Please enter
the ride number you want:',
value_min=1,value_max=5)
print('You have selected ride:',ride_number)
```

**Question:** What would this program print?

**Answer:** It would print that the user had selected ride number 1.

[Click here to view code image](#)

```
You have selected ride: 1
```

The second `return` would not be reached because execution of a function ends when a `return` statement is reached.

[Click here to view code image](#)

```
EG7-09 get_value investigation 2
def get_value(prompt, value_min, value_max):
 return

ride_number=get_value(prompt='Please enter
the ride number you want:',
value_min=1,value_max=5)
print('You have selected ride:', ride_number)
```

**Question:** What would this program print? Would it run correctly?

**Answer:** If a function is intended to perform a particular activity instead of delivering a result, the function can contain a `return` that is not followed by a value (see the code above). In this case, the function returns a special value called `None`, which is used in Python to represent the lack of a useable value. The program above would print out the value of `None`, which is the string `None`.

[Click here to view code image](#)

```
You have selected ride: None
```

Python will also return the `None` value if a statement tries to use the value returned by a function that does not contain any `return` statements.

**Question:** Can a function contain multiple `return` statements?

**Answer:** Yes. The program will return from the function when it reaches the first `return` statement.

Below, you can see the complete `get_value` function.

```

EG7-10 complete get_value
def get_value(prompt, value_min, value_max):
 while True:
 number_text = input(prompt)
 try:
 number = int(number_text)
 except ValueError:
 print('Invalid number text. Please enter digits.')
 continue # return to the top of the loop
 if number<value_min:
 print('Value too small')
 print('The minimum value is',value_min)
 continue # return to the top of the loop
 if number>value_max:
 print('Value too large')
 print('The maximum value is',value_max)
 continue # return to the top of the loop
 # If we get here the number is valid
 # return it
 return number

```

Function header

This loop will repeat forever

This function repeatedly reads integers until supplied with one in the required range. In other words, a value of **number** that is less than **value\_min** or larger than **value\_max** will cause the loop to repeat. The **get\_value** function ends when a valid value is entered, at which point the **return** statement is reached and the function returns the number that has been read in. We can use this function to read in values from the user.

[Click here to view code image](#)

```

ride_number=get_value(prompt='Please enter the ride
number you want:',
value_min=1, value_max=5)
print('You have selected ride:', ride_number)

```



## PROGRAMMER'S POINT

### Designing with functions

Functions are a very useful part of the programmer's toolkit and form an important part of the

development process. Once you've worked out what a customer wants the application to do, you can start thinking about how you'll break down the program into functions. Once you've specified the behavior of each function in the application, you can write the function headers (in other words, pick the function name, the parameters, and any return value) and then you could even get someone else to write that function for you.

Functions are also useful for saving you from writing too much code. Often, you find that as you write a program, you write code that repeats a particular action. If you do this, you should consider taking that action and turning it into a function. There are two reasons why this is a good idea:

- First, you only write the code once. If you find a fault function you only have to fix it once.
- Secondly, functions make a program easy to test. You can regard each function as a “data processor.” Data goes into the function via the arguments, and output is produced via the return value. We can write what is called a “test harness” to call a function with test data and then check to ensure the output is sensible. In other words, we can make a program that tests itself. Professional developers will create the test code alongside the program code. Frameworks can be used to automate this testing process even more. We'll look at these in [Chapter 12](#).

## Local variables in Python functions

Imagine several cooks working together in a kitchen. Each cook is working on a different recipe. The kitchen contains a limited number of pots and pans for the cooks to share. The cooks would need to coordinate so that two of them didn't try to use the same pot. Otherwise, we might get sugar added to our soup and custard instead of gravy on our roast beef.

The designer of Python faced a similar problem when creating functions. He didn't want functions to fight over variables in the same way that two cooks might fight over a particular frying pan. You might think that it would be unlikely that two functions would try to use variables with the same name, but this is actually very likely.

Many programmers (including me) have an affection for the variable name `i`, which they use for counting. If two functions use a variable called `i` and one function calls the other function, this could lead to programs that don't work properly because the second function might change `i` to a value that the first function didn't expect.

Python solves this problem by giving each function its own *local* variable space. This is the programming equivalent of giving each cook their own personal set of pots and pans. Any function can declare a local variable called `i` that is

specific to that function call. When a function returns, all local variables are destroyed. Variables declared outside functions are called global variables because they are not tied to any particular function.

[Click here to view code image](#)

```
EG7-11 Local Variables

def func_2():
 i = 99

def func_1():
 i = 0
 func_2()
 print('The value of i is: ', i)

func_1()
```

The code above shows how this works. Both `func_1` and `func_2` use a variable called `i`. When we run the program, it follows this sequence:

1. The function `func_1` is called.
2. The first statement of `func_1` creates a variable called `i` and sets it to `0`.
3. The second statement of `func_1` makes a call to `func_2`.
4. The first, and only, statement of `func_2` creates a variable called `i` and sets it to `99`.
5. The function `func_2` finishes and control returns to the third statement of `func_1`.
6. The third statement of `func_1` prints out the value of `i`.

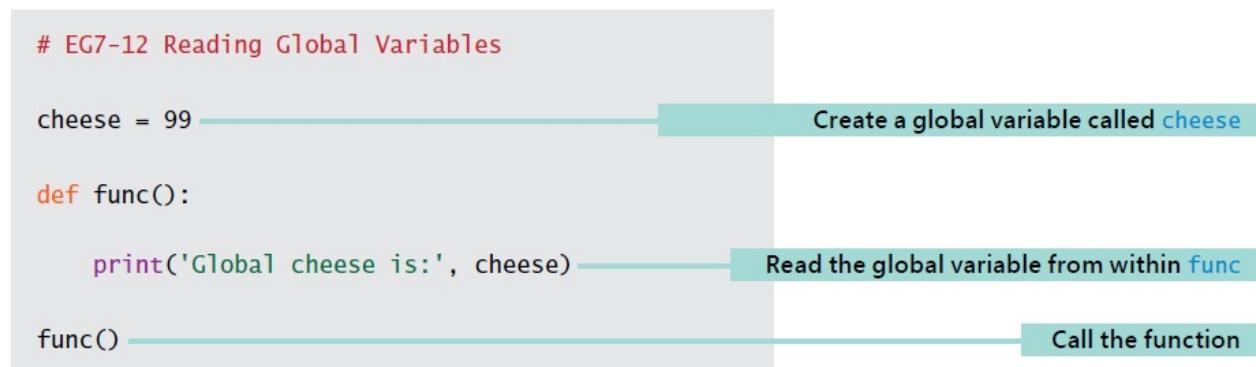
The question we must consider is, “What value is printed?” Is it the value `0` (which is set inside `func_1`) or is it `99` (which is set inside `func_2`)?

If you’ve read the first part of this section, you know the value that will be

printed is `0`. The variables both have the same name (they are both called `i`), but they each “live” in different functions. This form of isolation is called encapsulation. Encapsulation means that the operation of one function is isolated from the operation of other functions. Different programmers can work on different functions with no danger of problems being caused by variable names clashing with each other.

## Global variables in Python programs

Local variables are very useful, but sometimes a program contains data that needs to be shared among all functions. For example, you might want to share a player name among several functions that implement a game. Python allows functions to have access to variables held at the *global* level. A global variable is declared outside any function.



The example program above shows how a function can read the content of a variable declared at a global level. This program runs perfectly and will print:

```
Global cheese is: 99
```

The message is printed from code running inside the `func` function. So, we can see that it's easy to read global data from within a function. We just need to use the variable. Unfortunately, storing values is a bit more complicated.

```
EG7-13 Shadowing Global Variables
```

```
cheese = 99
```

Create a global variable called `cheese`

```
def func():
```

```
 cheese = 100
```

```
 print('Local cheese is:', cheese)
```

Create a local variable called `cheese`

Print the `cheese` local variable

```
func()
```

Call the function

```
print('Global cheese is:', cheese)
```

Print the `cheese` global variable

You might think you understand the above code from looking at it. The program contains a global variable called `cheese`. This variable is initially set to `99`. The program then calls the function `func`. A statement within the function sets the value of `cheese` to `100`. Then the function returns. You might expect the program to print out the following, because when the function runs it sets the value of `cheese` to `100`.

```
Local cheese is: 100
Global cheese is: 100
```

However, this is not what happens. Instead, the program prints this:

```
Local cheese is: 100
Global cheese is: 99
```

Python creates a new local variable with the same name as a global variable. This is called *shadowing*. The local shadow `cheese` variable is used in the function instead of the global `cheese` variable. In effect, this program contains two variables called `cheese`. One is global, and the other is local to the `func` function.

The shadowing behavior can lead to much confusion. Unless you know how it works, you can lose many hours trying to work out why your variables are not updating. This behavior is unfortunate because reading from a global variable in a function works perfectly, but storing values in the global variable results in the

creation of a shadow.

If you want a function to be able to access global variables, you can identify global variables to be used inside the function.

```
EG7-14 Storing Global Variables

cheese=99 Declare a global variable called cheese

def func():
 global cheese
 cheese=100
 print('Global cheese is:',cheese)

func()
print('Global cheese is:',cheese)
```

The **global** statement is followed by the name of the global variable in which you want to store a value. In the above program, there is only one variable called **cheese**, and it is shared among all functions.

You might wonder why global variables work in this confusing manner. A function can read a global variable but must use a special **global** statement if the function wants to store values in the global variable. This is because the designer of the Python language was anxious to avoid problems if a local variable in a function is accidentally given the name of global variable. If the function was able to change a global variable, other parts of the program would be affected by an unexpected change to the global value.

Python forces the programmer to use a **global** statement to link a function to a global variable so that global variables are only written to when the programmer explicitly chooses to do so.



## PROGRAMMER'S POINT

### Use global data with care

Global data can be very useful. However, it can also be the source of hard-to-find problems with your programs. If variables can be changed by many functions, a mistake in one function could affect the proper operation of many others. If you do decide to use global variables, I suggest that

you use plenty of comments to clarify how the variables are being used.

## Build reusable functions

Asking users for text input is a dangerous business. Users can break our programs by typing the wrong thing, and they can stop our program completely by using the keyboard interrupt command Ctrl+C. Because many of our programs request user input, it makes sense to create some Python functions that can manage the input process for us. We can then use these functions in all our future programs.

### Create a text input function

The first function we'll create will read in a string of text from the user. We could use the Python `input` function for this, but a user could enter the Ctrl+C key combination, which will raise an exception and stop the program. You've learned how to deal with exceptions, so now we'll put this behavior into a function. I'll call the function `read_text`. When we design a function, the first thing we decide on is the parameters that the function accepts and the value it returns.

```
def read_text(prompt):
```

This definition indicates that the function has a single parameter called `prompt`. A program could use the function as follows:

[Click here to view code image](#)

```
name = read_text(prompt='Please enter your name: ')
```

This would set the `name` variable to the result of the function call. The user of the program would see the following:

[Click here to view code image](#)

```
Please enter your name: Rob
```

In this version of the function, the `read_text` function must be supplied with a prompt string as the argument to the function. We could modify this code to allow the function to be used without a prompt:

[Click here to view code image](#)

```
def read_text(prompt='Please enter some text: '):
```

Now a program can use `read_text` without supplying an argument:

```
name=read_text()
```

When `read_text` runs, the prompt parameter is now set to the default value:

[Click here to view code image](#)

```
Please enter some text: Rob
```

You can have an interesting discussion about whether the function should provide a default prompt. It's sensible for the function to always require a prompt because the prompt forces the programmer using the function to provide a sensible message for the user. Now that we've defined how the function should be used, we can go ahead and add the code that will make it work:

[Click here to view code image](#)

```
1. def read_text(prompt):
2. while True: # repeat forever
3. try:
4. result=input(prompt) # read the
input
5. # if we get here, no exception was
```

```
raised
6. # break out of the loop
7. break
8. except KeyboardInterrupt:
9. # if we get here, the user pressed
Ctrl+C
10. print('Please enter text')
11. return result
```

This function will read a line of text from the user and ignore any keyboard interrupts.



## CODE ANALYSIS

### Investigating the `read_text` function

Let's look at the `read_text` function and how it works.

**Question:** What is the `result` variable used to accomplish?

**Answer:** The `result` variable holds the text that the function will return to the caller. It is a local variable. It exists only inside the `read_text` function.

**Question:** What stops the function from repeating continuously?

**Answer:** Line 7 contains a `break` that will end the loop and cause the program to continue running at the statement after the loop, which returns the text in the `result` variable.

**Question:** Why does the text reading loop repeat after the exception has been dealt with?

**Answer:** A `while` construction will repeat all statements in the

suite of code that it controls. In the `read_text` function, the indented text underneath the start of the `while` is repeated. The `return` statement is the first non-indented statement under the `while`. So, Python will go back to the top of the loop when it finds the first statement that is not part of that loop. When the loop ends, the `return` is performed and the function ends.

## Add help information to functions

You've learned how to add comments to Python programs that explain how the code works. Python also has a commenting convention for functions we create. The first statement in a Python function can be a Python string describing what the function does. These strings can be picked up by programs that read Python source code and produce documentation.

[Click here to view code image](#)

```
def read_text(prompt):
 'Displays a prompt and reads in a string of
 text'
```

This is a single-line string that provides descriptive information about the function. If you want to provide more detail, the program can contain a multi-line string doing just that:

[Click here to view code image](#)

```
def read_text(prompt):
 """
 Displays a prompt and reads in a string of
 text.
 Keyboard interrupts (Ctrl+C) are ignored
 returns a string containing the string input by the
 user
 """
```

In a Python program, we can create a string of text that spans several statements by using triple quotes to mark the start and the end of the string (see [Chapter 3](#) to learn more). The description string above is the kind of thing I'd write for one of my functions. It describes broadly what the function does, mentions “interesting” behaviors, and tells the reader what the function returns.

## Use pydoc

We can use the `pydoc` library to search for the descriptive strings for a specific function in a program:

[Click here to view code image](#)

```
>>> import pydoc
>>> pydoc.help(read_text)
Help on function read_text in module __main__:

read_text(prompt)
 Displays a prompt and reads in a string of
text.
 Keyboard interrupts (Ctrl+C) are ignored
 returns a string containing the string input by
the user
```

Above, you can see how we can use the `pydoc` library. After importing the library, we can use the `pydoc.help` function to display the help information for a particular function (in this case, the `read_text` function). We can also use `pydoc` to get help on built-in functions. We will use `pydoc` in [Chapter 12](#) to produce documents that describe our programs.

[Click here to view code image](#)

```
>>> pydoc.help(print)
Help on built-in function print in module builtins:

print(*args, **kwargs)
 print(value, ..., sep=' ', end='\n',
```

```
file=sys.stdout, flush=False)

 Prints the values to a stream, or to sys.stdout
 by default.
 Optional keyword arguments:
 file: a file-like object (stream); defaults to
 the current sys.stdout.
 sep: string inserted between values, default
 a space.
 end: string appended after the last value,
 default a newline.
 flush: whether to forcibly flush the stream.
```



## PROGRAMMER'S POINT

### Form a habit of documenting your code

In the old days, a programmer would have to write a large document that describes how their program works and all its internal behaviors. These days, the documentation is actually made part of the program text. Make sure when you write a function that you add the documentation text. You'll be surprised just how quickly you can forget how a program works.

## Create a number input function

Now that we have a function that can read a string of text, we can use this to make a function that will read in a number from the user. The number input function will reject input that does not contain digits. The number input function will deal with any exceptions that might be raised when the user input is converted into a number.

[Click here to view code image](#)

```
def read_float(prompt):
 """
 Displays a prompt and reads in a number.
 Keyboard interrupts (Ctrl+C) are ignored
 Invalid numbers are rejected
```

```
 returns a float containing the value input by
the user
 """
 while True: # repeat forever
 try:
 number_text = read_text(prompt)
 result = float(number_text) # read the
input
 # if we get here, no exception was
raised
 # break out of the loop
 break
 except ValueError:
 # if we get here, the user entered an
invalid number
 print('Please enter a number')

 # return the result
 return result
```

The function `read_float` reads a string of text and tries to convert it into a floating-point value. It would be used as follows.

[Click here to view code image](#)

```
age=read_float('Please enter your age: ')
```

If the conversion process throws an exception, the read loop is repeated. Notice that this function looks remarkably like the `read_text` function. This shouldn't be too much of a surprise, because the problem the functions are solving (they keep trying to do something until it works) is the same for both. Programmers call these kinds of things “patterns.” We’ll see this pattern in action when we create the next function, which will read a number in and reject values that are out of the given range.

```
def read_float_ranged(prompt, min_value, max_value):
 """
 Displays a prompt and reads in a number.
 min_value gives the inclusive minimum value
 max_value gives the inclusive maximum value
 Keyboard interrupts (Ctrl+C) are ignored
 Invalid numbers are rejected
 returns a float containing the value input by the user
 """

 while True: # repeat forever
 result = read_float(prompt) Use the read_float method we have already written
 if result < min_value:
 # Value entered is too low
 print('That number is too low')
 print('Minimum value is:', min_value)
 # Repeat the number reading loop
 continue
 if result > max_value:
 # Value entered is too high
 print('That number is too high')
 print('Maximum value is:', max_value)
 # Repeat the number reading loop
 continue
 # If we get here, the number is valid
 # break out of the loop
 break
 # return the result
 return result
```

The `read_float_ranged` function is used as follows:

[Click here to view code image](#)

```
age=read_float_ranged('Please enter your age: ',
 min_value=5, max_value=90)
```

Note that I've used keyword arguments to make the meaning of the parameters clear.



## CODE ANALYSIS

### Investigating the `read_float_ranged` function

The `read_float_ranged` function uses the same pattern as the earlier functions, but it is worth taking a closer look at some parts of it.

**Question:** Why doesn't this function have any code in it to catch exceptions?

**Answer:** There is no need for this function to catch exceptions. If the user tries to break the program by using Ctrl+C, the exception raised will be caught by the `read_text` function, which is called by the `read_float` function. And the `read_float` function will catch any exceptions raised if the user types in text that is not part of a number.

**Question:** Will chaining these functions together slow down the program?

**Answer:** We've built up our library from a low-level function (fetch some text) all the way up to a high-level function (fetch a numeric value in a particular range). You could write a "free standing" `read_float_ranged` function that didn't make use of any other functions. This would probably run slightly faster because it wouldn't spend as much time assembling function calls. However, I prefer my version because I think it's much easier to understand and maintain.

**Question:** What would happen if a programmer reversed the maximum and minimum values?

[Click here to view code image](#)

```
age=read_float_ranged('Enter your age:',
```

```
min_value=90, max_value=5)
```

**Answer:** This is a serious error. We're asking `read_float_ranged` to deliver a number greater than 90 and less than 5. No such number exists. The program would never complete the `read_float_ranged` function because every value that was entered would be rejected. This would be very upsetting for the user of the program.

Because we're using keyword arguments, it's much harder to make this mistake, but it's still possible. You could take the view that a programmer making this mistake deserves the bad things that will happen to their program, but you should at least let programmers know that your function doesn't detect this error. You can do this by adding a note to the description string for the method.

[Click here to view code image](#)

```
>>> pydoc.help(read_float_ranged)
Help on function read_float_ranged in module
__main__:

read_float_ranged(prompt, min_value,
max_value)
 Displays a prompt and reads in a number.
 min_value gives the inclusive minimum
 value
 max_value gives the inclusive maximum
 value ** Does not detect if max and min are
 reversed **
 Keyboard interrupts (Ctrl+C) are ignored
 Invalid numbers are rejected
 returns a float containing the value
 input by the user
```

With a bit of luck, other programmers will notice that the function doesn't detect if max and min are reversed, and they'll use

`read_float_ranged` correctly. Otherwise, they'll find that their number validation will repeatedly reject values. If we wanted to go one better, we could add a test to the function that detects when the max and the min values are reversed:

[Click here to view code image](#)

```
if min_value > max_value:
 # If we get here, the min and the max
 # are reversed
```

If a programmer inadvertently swaps the `min` and `max` values, simply returning them to the correct order would be a *very bad thing to do*. This is a bad idea because we should not assume that we know the kind of mistake the programmer has made. We are assuming that they reversed the values. However, it is equally likely that the mistake could arise as a simple typing error.

[Click here to view code image](#)

```
age = read_float_ranged('Enter your
age:', min_value=5, max_value=.90)
```

In the above code, the programmer has accidentally pressed the period (decimal point) key when typing the maximum age. If the program simply swapped the `min` and `max` values, the number validation would now take place in the range between 0.9 and 5, which is wrong. The best thing the function should do in this situation is raise an exception.

[Click here to view code image](#)

```
if min_value > max_value:
 # If we get here, the min and the max
 # are the wrong way around
 raise Exception('Min value is greater
```

```
than max value')
```

Raising an exception ensures that the program will fail and that the error will be brought to the attention of the programmer. This is much better than the function guessing what the mistake might be and then trying to fix it. If a function throws exceptions in this way, I consider it good manners to add details in the documentation for the function.

## Convert our functions into a Python module

Currently, the functions we've used have been defined at the start of the program file where we want to use them. However, for the number reading functions that we just created, it would be wonderful if we could use them in every program that we write from now on. It turns out that in Python this is a very easy thing to do. We can create a *module*. A module is a program file that contains some Python code that we want to use in many different programs. Some program languages call such a thing a *library*, but in Python the proper term is module.

To make a module, we just must put the function code into a Python source file, and then we can import the functions in that file into any program. The only thing we must remember is that the module file and the program file that is using the module must be located in the same folder. In [Chapter 12](#) we will learn more about how to create folders that contain Python modules.

I put the functions in a source file called [\*\*BTCInput.py\*\*](#). At the start of any program that wants to use these functions, I just need to import the functions from this file:

```
import BTCInput
```

We can call functions from this module in the same way as we have called functions from other modules.

[Click here to view code image](#)

```
age = BTCInput.read_float_ranged('Enter your age:',
min_value=5, max_value=90)
```

Python has an alternative import mechanism that might make our programs slightly simpler. We can import functions so that we can use them directly.

[Click here to view code image](#)

```
from BTCInput import read_float_ranged
age = read_float_ranged('Enter your age:',
min_value=5, max_value=90)
```

Once a function has been explicitly imported using the `from...import` construction, it can be used without the module name in front of it. If you want to explicitly import all the functions from a module, you can use the `*` character as a wildcard that will match all the function names in the module.

[Click here to view code image](#)

```
from BTCInput import *
age = read_float_ranged('Enter your age:',
min_value=5, max_value=90)
```

This form of importing makes the functions from a module easier to use, but it does raise the prospect of names clashing. If two modules contain a function with the same name, you'll find that one of the functions will be overwritten by the other if you import both functions using the `*` wildcard. To understand why you might encounter problems, it's worth considering what happens when you import a module.

We know that Python defines functions as a program executes. When the Python engine encounters a statement starting with the word `def`, the engine stops executing statements and instead starts to build a function that is stored for use later in the program.

When a Python program contains an `import` statement, the Python engine reads the contents of the imported file. The Python engine obeys any Python statements in the imported file and builds any functions defined in this file. If you create a second version of something in Python, the original version is replaced. If two files define the same function, the definition in the second file that is read will replace the first definition. Of course, this might lead to strange behavior in your programs.



## MAKE SOMETHING HAPPEN

### Add number input to all your programs

You can find the number input functions in the module file `BTCInput.py` in the sample programs for this chapter. The `BTCInput.py` file also contains functions that can be used to read integer values. The sample program **EG7-15 Using the input module** shows how these functions are used. You can add these number-reading routines in all the programs we've written so far.

## Use the IDLE debugger

We can check our programs by working through them by hand, but we can also use IDLE to view the actions of a program as it runs. We use the IDLE *debugger* to do this. As the name implies, a debugger is a tool that helps you remove bugs from your programs. You can use a debugger to determine the path your program is following, rather than the path you *think* it is following. You can also use the IDLE debugger to discover how Python constructions work.

We'll start by adding a *breakpoint* to our program. A breakpoint doesn't cause the program to break; rather, it causes the program to "take a break." When a program reaches a statement designated as a breakpoint, the program is paused, the Python engine hands control back to the programmer, and then the programmer can verify that each variable has the necessary contents. A program

can contain many breakpoints; the first breakpoint that the program reaches will pause the program.



## MAKE SOMETHING HAPPEN

### Investigate programs with the debugger

I've written a little program we can investigate using the debugger.

Open the file **EG7-16 Investigating the debugger**. You'll find the code in the downloadable samples for this chapter. Open this sample program using the IDLE editor.

The screenshot shows the IDLE Python editor window with the title "EG7-16 Investigating the debugger.py - C:/Users/Rob/Desktop/EG7-16 Investigating the debu...". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code in the editor is:

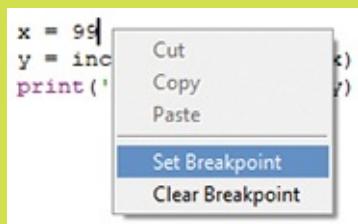
```
EG7-16 Investigating the debugger

def increment_function(input_value):
 result = input_value + 1
 return result

x = 99
y = increment_function(x)
print('The answer is:', y)
```

The status bar at the bottom right indicates "Ln: 12 Col: 0".

We'll put a breakpoint on the statement that sets the value of x to 99. Right-click on a character in this statement to open the context menu.



Selecting **Set Breakpoint** highlights the line containing the breakpoint.

The screenshot shows a Python script titled "EG7-16 Investigating the debugger.py" in the IDLE editor. The code defines a function "increment\_function" and uses it to calculate the value of "y". A yellow highlight bar is positioned over the line "x = 99", indicating it is the current statement being debugged. The status bar at the bottom right shows "Ln: 9 Col: 6".

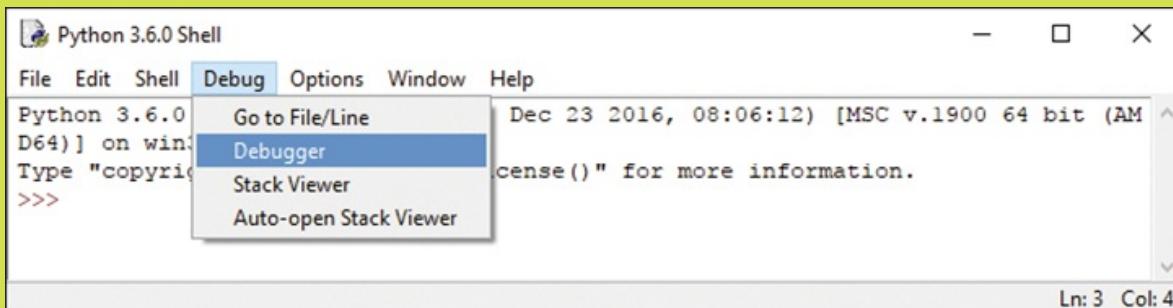
```
EG7-16 Investigating the debugger

def increment_function(input_value):
 result = input_value + 1
 return result

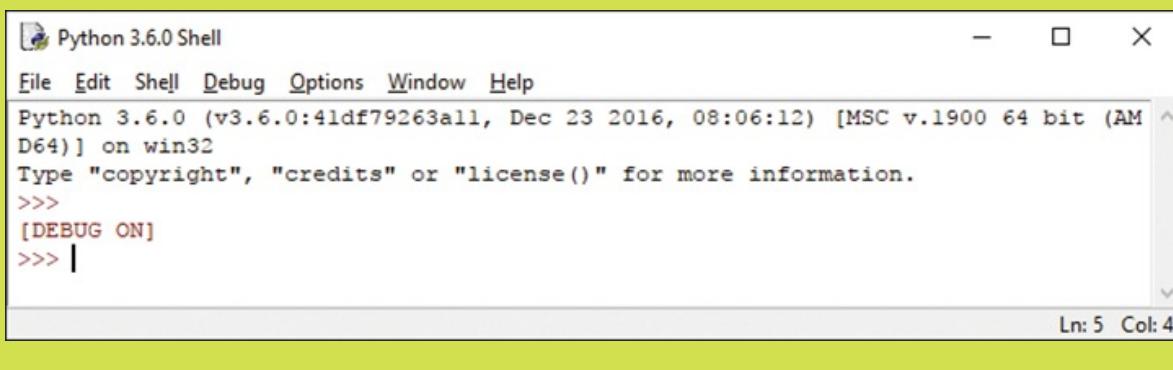
x = 99
y = increment_function(x)
print('The answer is:', y)
```

If you inadvertently highlight the wrong statement, don't worry. You can set a breakpoint on the correct statement and use the **Clear Breakpoint** option to remove the breakpoint from the wrong statement.

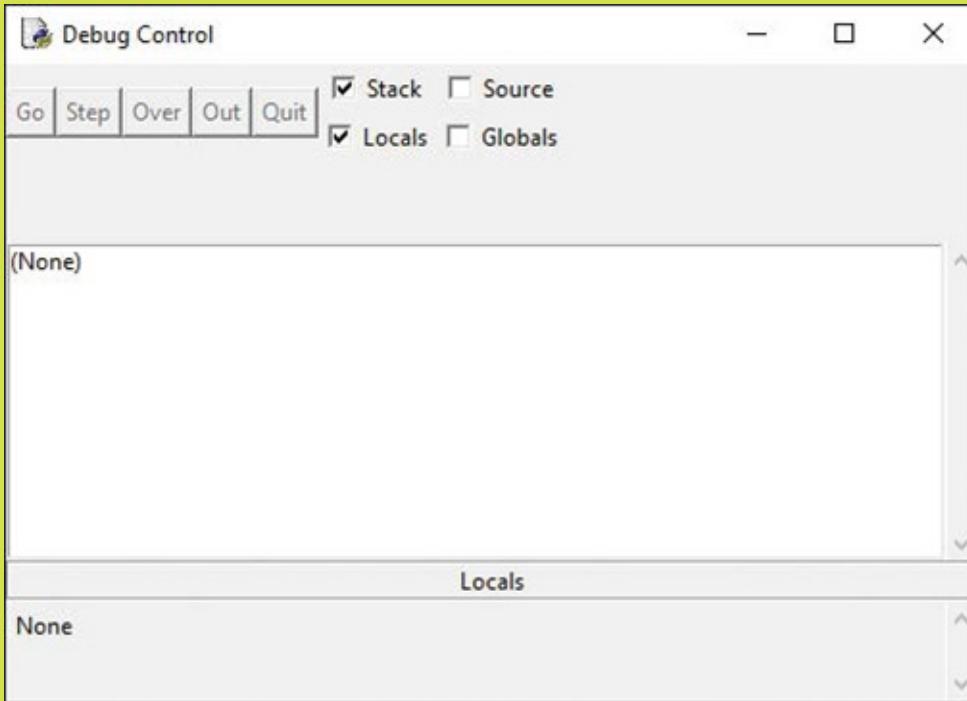
The Python Shell in IDLE only responds to breakpoints when it's in Debug mode. Next, enable Debug mode in the shell by selecting **Debug>Debugger**.



When you turn on the debugger, you'll notice two things happen. First, the shell will display a message to indicate that debugging is enabled.



Second, the Debug Control window opens.



The Debug Control window contains buttons you can use to start and stop your program, along with a display area where you can view the contents of the variables in your program.

This window is active only when a program is being debugged. To start a debugging session, go back to your program file and start the program using the **Run>Run Module** menu option or by pressing **F5**. The Debug Control window now comes to life.

The screenshot shows the 'Debug Control' window with the title bar 'Debug Control'. At the top, there is a menu bar with buttons for 'Go', 'Step', 'Over', 'Out', and 'Quit'. To the right of these buttons are checkboxes for 'Stack' (checked), 'Source' (unchecked), 'Locals' (checked), and 'Globals' (unchecked). Below the menu is a status message: 'EG7-16 Investigating the debugger.py:4: <module>()'.

The main area displays the current stack trace:

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__.<module>()', line 4: def increment_function(input_value):
```

At the bottom, there is a 'Locals' section showing the current variable values:

| Variable        | Value                                                      |
|-----------------|------------------------------------------------------------|
| __annotations__ | {}                                                         |
| __builtins__    | <module 'builtins' (built-in)>                             |
| __doc__         | None                                                       |
| __file__        | 'C:/Users/Rob/Desktop/EG7-1...vestigating the debugger.py' |
| __loader__      | <class '_frozen_importlib.BuiltinImporter'>                |
| __name__        | '__main__'                                                 |
| __package__     | None                                                       |
| __spec__        | None                                                       |

You can think of the Debug Control window as a “dashboard” for your running program. At the top are controls, in the middle is a display that shows the current position that’s been reached in the program, and at the bottom is a view of the program variables. The highlighted line shows that the program is at the very beginning, which is the **def** statement that defines **increment\_function**, as shown below.

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__.<module>()', line 4: def increment_function(input_value):
```

Note that the program control buttons in the top left corner of the window—Go, Step, Over, Out, Quit—are now enabled, so we can use

them to control how the program runs.



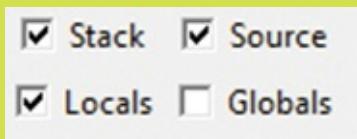
Start the program by pressing **Go**, which causes the program to run until it hits a breakpoint. We put the breakpoint on the assignment `x = 99`. If you look in the display in the center of the Debug Control window, you can see that this statement has been reached.

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 9: x = 99
```

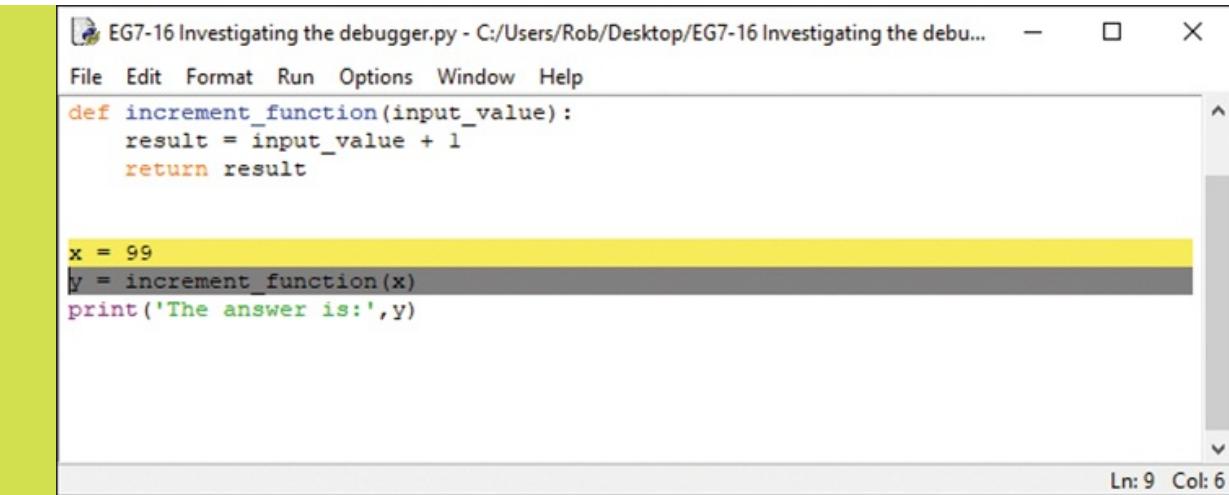
We can press the **Step** button to move the program to the next statement that will be executed. The display in the center of the Debug Control window will update to show you this statement:

```
'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 10: y = increment_function(x)
```

It would be useful to see the statement in the source code. If you turn on the **Source** check box in the Debug Control window, you can see the running Python code.



If you select the **Source** check box as shown above, the Python debugger will find and highlight the file containing the currently active Python statement.

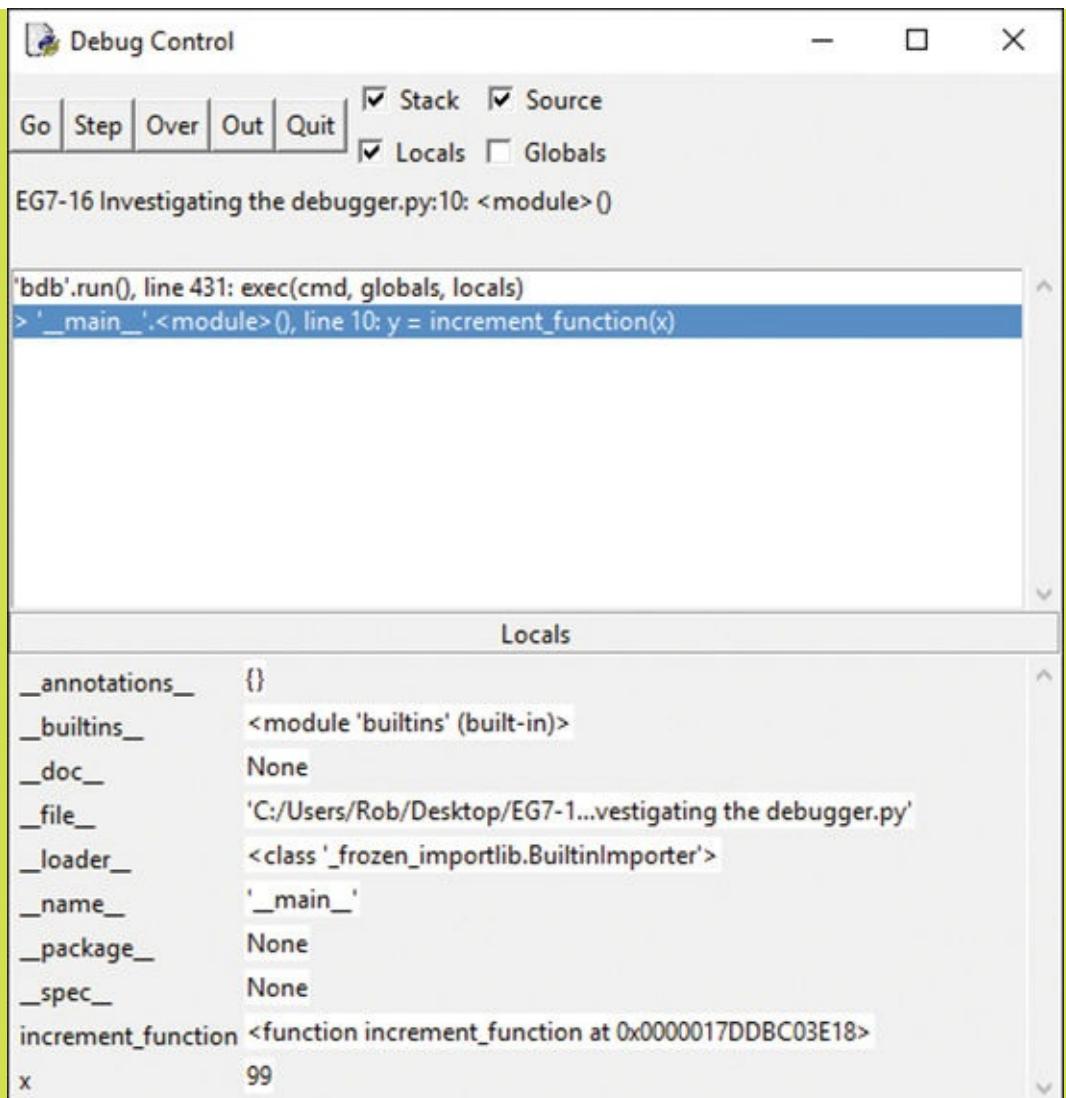


```
File Edit Format Run Options Window Help
def increment_function(input_value):
 result = input_value + 1
 return result

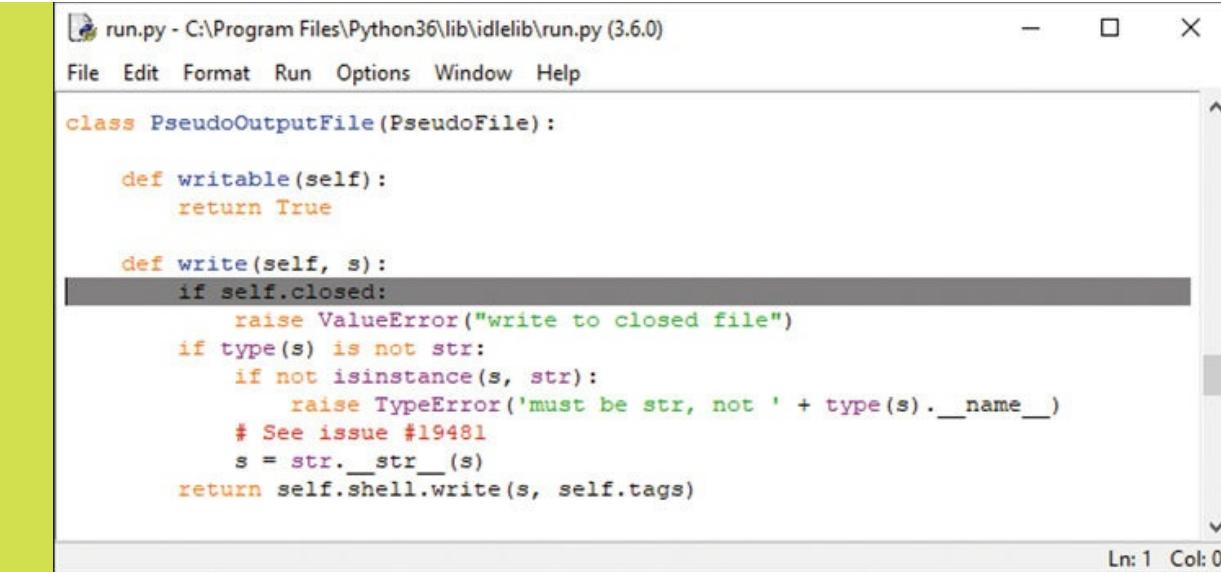
x = 99
y = increment_function(x)
print('The answer is:', y)

Ln: 9 Col: 6
```

The Debug Control window also displays the values in variables. If you look at the bottom of the window, you'll see that the value of `x` is now displayed.



You can continue pressing the **Step** button and watch the program move into `increment_function`. If you press **Step** four more times, you'll see the program return from `increment_function` and call the `print` function, which is the last statement in the program. The `print` function is part of Python. The debugger will open the file that contains this code and show it to you.



```
run.py - C:\Program Files\Python36\lib\idlelib\run.py (3.6.0)
File Edit Format Run Options Window Help

class PseudoOutputFile(PseudoFile):

 def writable(self):
 return True

 def write(self, s):
 if self.closed:
 raise ValueError("write to closed file")
 if type(s) is not str:
 if not isinstance(s, str):
 raise TypeError('must be str, not ' + type(s).__name__)
 # See issue #19481
 s = str.__str__(s)
 return self.shell.write(s, self.tags)

Ln: 1 Col: 0
```

This part of the system is about to print the output from our program. It's pleasing to find that we can recognize the statements, but we don't really want to read this code just now. The **Out** button in the Debug Control window is used to tell the debugger to complete the currently active function and return from it. Press the **Out** button to complete the print action and end the program.

If you want to just step over a function call without going inside it (which is usually the case with functions such as `print` and `input`), you can use the **Over** button to step over these calls.

You might find it fun to add more breakpoints in the program and run it. You can also use the debugger to run any of your earlier programs and look at the path your programs follow.

## What you have learned

In this chapter, you learned how to take a block of code and turn it into a function that can be used from other parts of the program. You've seen that a function contains a header, which describes the function, and a block of code that is the body of the function. The function header supplies the name of the function and any parameters that are accepted by the function. When a function is called, the programmer supplies an argument that matches each parameter.

Parameters are items that the function can work on. They are passed by value, in that a copy is made of the argument given in the function call. If the function body contains statements that change the value of the parameter, this change is local to the function body. Parameters can be given “default” values that are used in the function call if a matching argument is not supplied. In the call of the function, programmers can add keywords that directly map values to parameters in the function.

A function can return a single value. This is achieved by using the `return` statement, which can be followed by a value to be returned. If no value is returned, or the function does not obey a `return` statement, the function will return a special Python value called `None`, which is used to denote a missing value.

Variables created inside the body of a function are local to the function and cannot be used by statements outside that function. Variables declared outside any function are called global variables. Functions can read values from global variables but must explicitly use the `global` statement to identify global variables to which they want to write. If a function writes to a local variable with the same name as a global variable, the global variable is said to be “shadowed” and cannot be used by the function. The reason for this complication is to make it less likely that matching global and local variables will not cause a program to fail.

A function can contain a string as the first statement in the function. This string should provide documentation for the function, explaining what it does, what inputs it has, and what value, if any, it returns.

Functions can be made available to Python programs by placing them in a Python source file, which is then imported into the program.

Here are some questions that you might like to ponder about the use of functions in programs:

### **Does using functions in programs slow down the program?**

Not normally. There is a certain amount of work required to create the call of a function and then return from it, but this is not normally an issue. The benefits of functions far outweigh the performance issues.

## **Can I use functions to spread work around a group of programmers?**

Indeed, you can. This is a very good reason to use functions. There are several ways that you can use functions to spread work around. One popular way is to write placeholder functions and build the application from them. A function will have the correct parameters and return value, but the body will do very little. As the program develops, programmers fill in and test each function in turn.

## **How do I come up with names for my functions?**

The best functions have names given in a verb-noun form. `read_string` is a good name for a function. The first part indicates what it does, and the second part indicates what it delivers. I find that thinking of function names (and variable names, for that matter) can be quite hard at times.

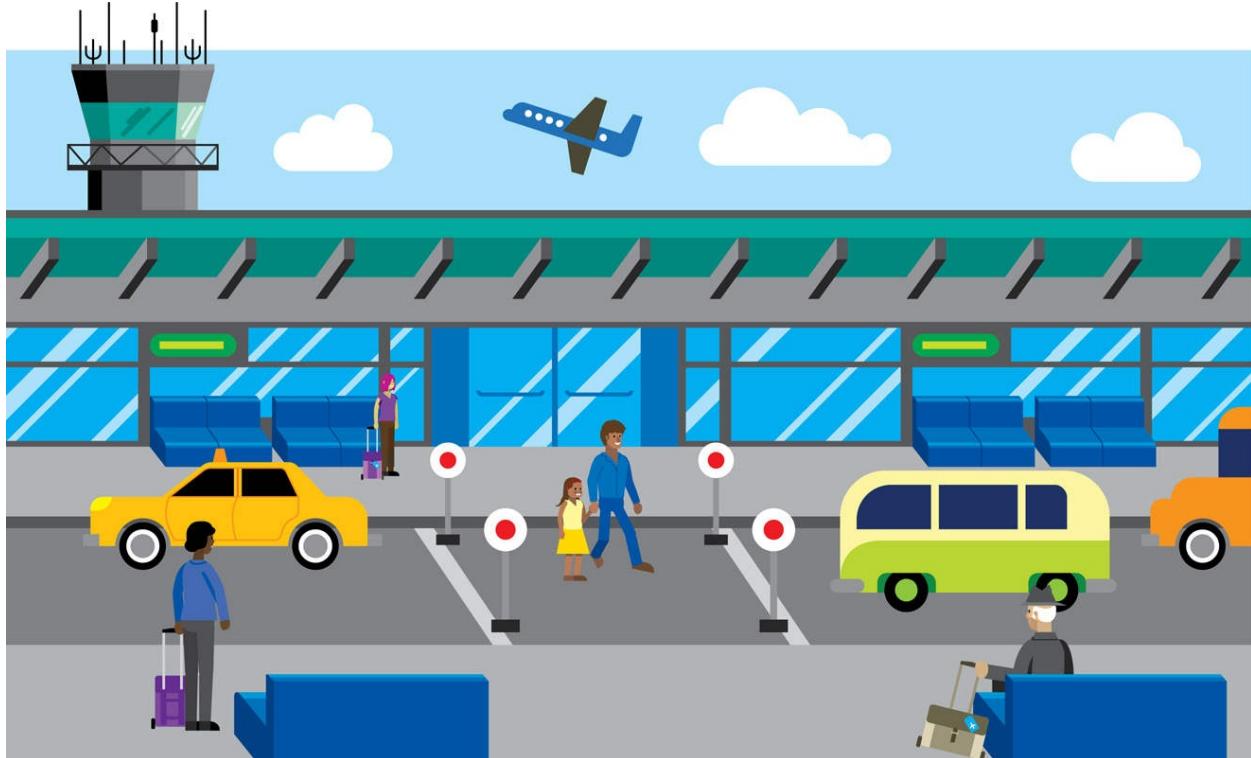
## **Can functions in libraries use global variables?**

We've seen that a function in a Python source file can access "global" variables declared in that file. A global variable is one that is not declared inside a function. A Python library file can contain global variables that can be used inside that library, but global variables declared in one Python source file cannot be used in another. In other words, if a program contains a global variable called "status," this variable would not be useable in any libraries that are imported by this program.

## **Should I put all my functions in modules/libraries?**

Libraries are very useful, but you probably shouldn't put all your functions in module files. It's fine for utility functions such as `read_text` to be placed in a library. However, when you start creating functions for use in a particular application, you might find it works better if you define such functions in the files where they are used, particularly if you want to use global variables to share values between functions.

## 8 Storing collections of data



### What you will learn

You might find this surprising, but you've already learned most of what you need to know to tell a computer what to do. You can write a program that stores items of data, makes decisions based on data values, and repeats behaviors as long as particular conditions are true. You can also create functions that accept data, perform actions on that data, and return results. These are the fundamentals of programming, and all programs are built on these core capabilities.

However, there's one more thing you need to know before you can write most any kind of program. You must understand how to manage substantial amounts of data in your programs. In this chapter, you'll learn how to work with collections of data by using lists, and you'll learn how to use loops to work through those lists. You'll also learn about the strangely named "tuples" and how

to use them. Additionally, you'll discover how to store your data in files on your computer.

[Lists and tracking sales](#)

[Refactor programs into functions](#)

[Sort using bubble sort](#)

[Store data in a file](#)

[Store tables of data](#)

[Use lists as lookup tables](#)

[Tuples](#)

[What you have learned](#)

## Lists and tracking sales

Let's say the owner of a group of ice-cream stands asks that you write a program to help her track sales results. She has ten ice-cream stands around the city, each selling a variety of ice-cream treats. She wants to enter the sales value from each stand and then view the data in different ways:

- Sorted from lowest to the highest
- Sorted from highest to the lowest
- Showing just the highest and the lowest numbers
- Showing the total number of sales
- Showing the average sales value

She can use this information to help plan the location of her stands and reward the best sellers. If you get this right, you might be getting some free ice cream, so you agree to help.



## PROGRAMMER'S POINT

### Getting the specification right: Storyboarding

It's important to agree on a specification with your customers. There are many ways that you can develop a specification. I find that the best way is to sit down with your user and a large pad of paper—as far away from the computer as you can get—and draw up a “storyboard.” Storyboards are used in moviemaking to show everyone how the film will tell the story. Programs can have storyboards, too.

Whereas a movie storyboard describes one sequence—the narrative of the film—the storyboard for a computer program has branches that show how the user follows different paths through the application. The ice-cream sales tracker will contain a menu from which the user will select how they want to view the data (lowest to highest sales, highest to lowest sales, and so on). You would need to create a separate storyboard for each of the actions that the user can select.

In a storyboard, you can also draw up how the program will work and how the user will move from window to window within the program. You could even decide what color scheme to use. If the user says he's not worried about how the program will work and that he will leave that to you, you need to know that the user most certainly will be worried about how the program will work once you've created it. Work directly with the customer to design the program to ensure that you deliver exactly what's required. Storyboards show you exactly what needs to happen so that you write the program accordingly. If there's anything that the customer hasn't thought of, it will most likely be spotted as you build the storyboard. Building an understanding of how programs fit together can be a tremendous help when you get to the point of creating them. This activity is frequently called “paper prototyping” or “wireframing” a program design.

With the information you've gathered, all you need to do now is write the actual program itself. This example program will do the following:

- It will use variables to hold the sales values entered by the user.
- The program can use logical expressions to compare two sales values and choose the larger of the two (so that it can sort values and find the largest sales).
- The program will use `print` statements to display results to the user and the newly created `BTCInput` functions from [Chapter 7](#) to read in the data.
- Each feature of the program will be performed by a function that will work on global data stored in the program and shared among the functions.

You have decided that the program will have the following user interface:



The program will start by storing the sales figures. Next, the user can select the viewing option by entering the number for the command she wants to perform. If this looks somewhat familiar, it should, because it's very similar to how we created the ride selector for the theme park (see [Chapter 5](#)).

## Limitations of individual variables

Now that we've decided how the program will be used, we need to build the code that provides the behaviors that the user wants. The first thing the program should do is read in sales figures from the user, so let's start there.

This program needs to store 10 sales figures, so we could use 10 variables one for each of the values you want to store in the program. We can use the number reading functions that we created at the end of the previous chapter. I've created a library of input functions based on these functions. The library is in the file `BTCInput`, which is imported at the start of the program. The `read_int` method reads an integer from the user.

```
from BTCInput import *

sales1=read_int('Enter the sales for stand 1: ')
sales2=read_int('Enter the sales for stand 2: ')
sales3=read_int('Enter the sales for stand 3: ')
sales4=read_int('Enter the sales for stand 4: ')
sales5=read_int('Enter the sales for stand 5: ')
sales6=read_int('Enter the sales for stand 6: ')
sales7=read_int('Enter the sales for stand 7: ')
sales8=read_int('Enter the sales for stand 8: ')
sales9=read_int('Enter the sales for stand 9: ')
sales10=read_int('Enter the sales for stand 10: ')
```

Import the number reading functions  
Read in the first value  
Read in the second value

Now that we have the data in our program, we can start to work with it. First, we could create an `if` condition to decide whether the sales from stand 1 are the largest. You saw in [Chapter 5](#) how to combine conditions to make complicated logical expressions. The output from the following condition is true only if `sales1` is larger than all the other sales values. Note that a Python statement can be continued onto another line by use of the backslash (\) character at the end of the line of the statement you want to continue.

[Click here to view code image](#)

```
EG8-01 Finding the largest sales
if sales1>sales2 and sales1>sales3 and
sales1>sales4 \
 and sales1>sales5 and sales1>sales6 and
sales1>sales7 \
 and sales1>sales8 and sales1>sales9 and
sales1>sales10:
 print('Stand 1 had the best sales')
```

This statement works fine, the problem is (as you might have already spotted) that this program would have to repeat this condition 10 times to display the correct message for every possible stand with the best sales. The problem would become worse if your customer added another 20 sales stands, because the program would become even more complex, requiring 20 more variables, 20 more read statements, and 20 more complex conditions. That's not the path we want to follow to manage this volume of data.

## Lists in Python

Storing and working with large amounts of data is actually quite easy, but you need something better than single variables. You need to create a *collection*, and the simplest form of a collection is the Python *list*, so let's look at that.

A list is exactly what you might expect: a list of items. When I go shopping, I try to make a list before I head out to the store. When I have a bunch of things that I need to do, I like to create a to-do list. (Then I usually throw it away or lose it, but at least I tried.)

In the case of the sales program, I want to create an empty list and then add the sales values to the list as the values are read in.



### MAKE SOMETHING HAPPEN

## Creating a list

We can use the Python Shell to investigate how a list is created. Open the IDLE command shell and enter the statement below. This statement creates an empty list called sales. The brackets in this statement are very important. Don't use braces {} or parentheses (). A list contains a collection of *items* held in order.

```
>>> sales=[]
```

Once you've created a list, you can append items to the end of the list.

```
>>> sales.append(99)
```

The above statement would create an item containing the integer value

`99` and append it to the list called `sales`. List variables contain a method called `append` that can be called to add an item to the end of the list. If the list is empty (as ours was), then the new value of `99` becomes the first item in the list. We can append another item to the list by using `append` again.

```
>>> sales.append(100)
```

We now have a list that contains two items. We can view the contents of the list just by giving the name of the list to the Python Shell.

```
>>> sales
[99, 100]
```

We could add as many items as we like to the list by making further calls to the `append` method. The next thing we need to do is determine how to access the individual items in the list.

To do this, we use a process called *indexing*. We can use an index value to identify a specific item in the list. The item at the beginning of the list has the index value of 0, which can be confusing because humans don't number things starting with zero. You wouldn't say, "I'll have the zeroth item on the menu" or "I live at house number zero at the top of the street." Humans naturally link the first item in a list with the number 1. You might find it best to think of the index as the distance down the list that you must travel to get to the item you want.

To access the item in the list, you provide the index value enclosed in square brackets.

```
>>> sales[0]
99
```

This statement displays the item at the start of the list. We can use

indexes to allow us to change the contents of an item in a list.

```
>>> sales[1]=101
```

This statement will change the contents of the item at the end of the list (remember that there are only two items in this list). If we view the list, we'll see the effect of the change.

```
>>> sales
[99, 101]
```

If a program tries to index an item not present in the list, Python will produce an exception. For example, try finding the item with an index of 2.

```
>>> sales[2]
```

There are two items in the list we know have the index values 0 and 1. There is no item with the index value of 2, so Python complains.

[Click here to view code image](#)

```
Traceback (most recent call last):
 File "<pyshell#57>", line 1, in <module>
 sales[2]
IndexError: list index out of range
```

A list contains a collection of items. These items might all be the same type (our sales list really should contain integer values), but Python does not insist on this. Try adding a string to the end of the sales list.

```
>>> sales.append('Rob')
```

The statement above would work perfectly well. The first two items of sales are numbers, and the last one is a string. We can also replace items in the list with items of a completely different type at any time.

```
>>> sales[0]='Python'
```

This replaces the integer value of 99 at the start of the list with a string containing the word, “Python.”

```
>>> sales
['Python', 101, 'Rob']
```

Note that just because you *can* create lists that contain lots of different types of data, I suggest that you don’t do this. The sales figure application we’re making will rely on all the items in the sales list being numeric values. The application will crash if any items are strings of text. Just because Python lets you do something doesn’t mean that you should do it.

## Read in a list

Now that we know how to use a list, we could write some code to read in the list values. The best way to do this would be to create a loop that repeatedly reads the values.

[Click here to view code image](#)

This code will read in 10 sales values and store them in a list called `sales`. It uses a `for` loop that counts from 1 to 10. Each time around the loop, a `prompt` string is assembled and used in a call to `read_int`, which returns a value to be appended to the sales list.



## CODE ANALYSIS

### A list-reading loop

There are a few questions we might like to consider about this code.

**Question:** What is the purpose of the `count` variable?

**Answer:** The `for` loop sets the `count` variable to each value in the range. The `count` is used to produce the number prompt for the user so that the prompt for number input starts with “`Enter the sales for stand 1`” and then counts upward from there.

**Question:** Why does the `range` of the `count` value go from 1 to 11 when we only want to read in 10 values?

**Answer:** In Python, the upper limit of a range is *exclusive*. The loop will stop when the value of the counter reaches or exceeds the limit.

**Question:** Which item in the list would hold the sales for stand number 1?

**Answer:** The sales for stand number 1 would be held in the first item, which would be the one with the index of zero.

**Question:** What would I have to change in the program if I wanted to read in sales values from 100 stands?

**Answer:** The answer to this question illustrates just how useful loops and lists are. You would only need to change the upper limit of the range from 11 (for 10 numbers) to 101 (for 100 numbers).

[Click here to view code image](#)

```
for count in range(1, 101):
```

```
prompt = 'Enter the sales for stand ' +
str(count) + ': '
sales.append(read_int(prompt))
```

This version of the read code would read in and store the sales figures for 100 stands. You can make it work for any number of stands simply by changing 100 to a different value. You could even ask the user how many ice-cream stands she owns:

[Click here to view code image](#)

```
no_of_stands = read_int('Enter the number of
stands: ')
for count in range(1, no_of_stands+1):
 prompt='Enter the sales for stand ' +
str(count) + ': '
 sales.append(read_int(prompt))
```

Note that we must add 1 to the number of stands entered by the customer because the values produced by a range do not include the upper limit value.

It's great to provide this kind of flexibility, but you need to be careful for two reasons. First, the user might not want that flexibility. Suppose she has always had 10 stands and doesn't like having to type in a number she knows will never change. Second, every new feature you add brings the potential for new errors. You need to consider what your program should do if the customer tries to store the sales details of 1,000,000 stands.

**Question:** If I got one sales value wrong, would it be possible to edit the list to put in a corrected version?

**Answer:** We'd have to write the Python code to do this, but in principle a program can replace any item in the list with a new value without needing to change any other items.

## Display a list using a **for** loop

In Python, the action of a **for** loop is to work through a number of items. We've seen that a loop can work through the items in a range; a loop can also work through the characters in a string. We can also use a **for** loop to work through the items in a list.

[Click here to view code image](#)

```
EG8-03 Read and Display loop

#fetch the input functions
from BTCInput import *

#create an empty sales list
sales = []

read in 10 sales figures
for count in range(1, 11):
 # assemble a prompt string
 prompt='Enter the sales for stand ' +
str(count) + ': '
 # read a value and append it to sales list
 sales.append(read_int(prompt))

print a heading
print('Sales figures')
initialize the stand counter
count = 1
work through the sales figures and print them
for sales_value in sales:
 # print an item
 print('Sales for stand',
count,'are',sales_value)
 # advance the stand counter
 count = count + 1
```

This complete program reads in 10 sales values and then prints them out in the order they were entered by working through the `sales` list. You can use this pattern every time you want to read some data in and display it.

[Click here to view code image](#)

```
Enter the sales for stand 1: 50
Enter the sales for stand 2: 54
Enter the sales for stand 3: 29
Enter the sales for stand 4: 33
Enter the sales for stand 5: 22
Enter the sales for stand 6: 100
Enter the sales for stand 7: 45
Enter the sales for stand 8: 54
Enter the sales for stand 9: 89
Enter the sales for stand 10: 75
Sales figures
Sales for stand 1 are 50 Sales for stand 2 are 54
Sales for stand 3 are 29
Sales for stand 4 are 33
Sales for stand 5 are 22
Sales for stand 6 are 100
Sales for stand 7 are 45
Sales for stand 8 are 54
Sales for stand 9 are 89
Sales for stand 10 are 75
```



**MAKE SOMETHING HAPPEN**

Read the names of guests for a party

Lists can hold any type of data that you need to store, including strings. You could change the ice-cream sales program to read and store the names of guests for a party or an event you're planning.

Make a modified version of the sales program that reads in some guest names and then displays them. Make your program handle between 5 and 15 guests.

## Refactor programs into functions

Currently, our program is just a long sequence of statements. The first set of statements reads in the data into a list, and the second set of statements prints out the data. However, this might not be the best way to arrange the code. There might be situations in which we want to read in a second set of data, and we will probably want to print out the sales list more than once. With this in mind, we can take the program above and *refactor* it so that these two activities are performed by functions.

Refactoring a program is the process of taking the code and changing how the components fit together. We must refactor programs because it's often quite difficult to decide on the best way to do something until you start doing it. Usually, I get about half way through writing a program before I discover how it really should be structured and then must make some adjustments. I've noticed that I end up doing this *no matter how much time I spend planning before I write the program*. Now, this might just be me, but many other programmers have told me that they have similar experiences. Note that refactoring doesn't mean that I must tear up all my code and start again; instead, it means that I must rearrange the components to better reflect the problem I'm solving.

It turns out that changing the ice-cream program so that it uses functions is not very difficult. The IDLE editor can even indent the function code for me if I use the **Format, Indent Region** command.

```

EC8-04 Functions

#fetch the input functions
from BTCInput import *

#sales list used by the program
sales=[]

def read_sales(no_of_sales):
 """
 Reads in the sales values and stores them in
 the sales list.
 no_of_sales gives the number of sales values to store
 """

 # remove all the previous sales values
 sales.clear()
 # read in sales figures
 for count in range(1, no_of_sales+1):
 # assemble a prompt string
 prompt = 'Enter the sales for stand ' + str(count) + ':'
 # read a value and append it to sales list
 sales.append(read_int(prompt))

def print_sales():
 """
 Prints the sales figures on the screen with
 a heading. Each figure is numbered in sequence
 """

 # print a heading
 print('Sales figures')
 # initialize the stand counter
 count = 1
 sales[0] = 99

 # work through the sales figures
 for sales_value in sales:
 # print an item
 print('Sales for stand', count, 'are', sales_value)
 # advance the stand counter
 count = count + 1

#Program runs here
read_sales(10)
print_sales()

```

Function to be used to read the sales

Empty the sales list to remove old values

Function to print the sales values

First statement of program. Read 10 sales

Second statement of program. Display the results



## CODE ANALYSIS

# Functions in the sales analysis program

The program now consists of two functions, one called `read_sales` and one called `print_sales`.

**Question:** What does the parameter for the `read_sales` function do?

**Answer:** In the future, we might need to change the number of ice-cream stands that the program supports. To make the change as easy as possible, the `read_sales` function accepts a parameter that sets the number of sales values that it will read.

**Question:** What does `clear` do?

**Answer:** When we read a new set of values, we must make sure that any old values are discarded. A list provides a clear behavior that can be used to clear out all existing values.

**Question:** Why don't we need to tell the `print_sales` function how many sales figures to print?

**Answer:** The `for` loop in the `print_sales` function will work through all the items of the list and doesn't need to be told how many items are included.

**Question:** Why wasn't the `sales` list made global in the `read_sales` function? I thought functions must specifically identify global variables to be modified. The `read_sales` function appends items to the sales list, which looks to me like a change to the value of `sales`. Why does this work?

**Answer:** This is a very good question. To understand the answer, you must consider what Python variables do. Items stored in a

Python programs are *objects* that are referred to by *references*. When we create a named variable, we actually create an object and a named reference that refers to it.

```
age = 6
happy = True
```

The Python statement above would create two objects. One is an object that can hold an integer value; the other is an object that can hold a Boolean value. The `age` reference is attached to the integer, and the `happy` reference is attached to the Boolean value.

```
age = 7
```

When Python performs the above statement, the `age` reference is now attached to an integer object that holds the value `7`. When talking about some programming languages, you could say, “The box called `age` now has the value `7` in it.” However, in Python, it’s best not to think of it this way. In Python, you should say, “The `age` reference is now attached to a box that holds the value `7`.”

Assignments just change a reference to refer to a different object.

```
sales=[]
```

The Python statement above makes a reference called `sales` and attaches it to an empty list. If a program makes a change to the sales list—which it can do by using things like `append`—the object to which the `sales` reference is attached doesn’t change; instead the contents of that object are changed.

```
sales.append(99)
```

This statement would add the value `99` onto the end of the sales list. However, this statement would not cause the `sales` reference to become attached to a different object.

If this discussion has your head spinning a bit, don't worry. We'll return to this theme in later chapters when we start designing objects we've created.

## Create placeholder functions

During the development process, we can create “placeholder” functions for the behaviors we want in our programs. These are sometimes called stub functions because they need to be filled out into completed functions at a later date. When I wrote this book, I started with a set of headings for the things I wanted to write about, and then filled in each heading later. Stub functions are used in a similar way.

```
def sort_high_to_low():
 ...
 Print out a list of the sales figures sorted high to low
 ...
 pass
```

Placeholder for sort high to low

```
def sort_low_to_high():
 ...
 Print out a list of the sales figures sorted low to high
 ...
 pass
```

Placeholder for sort low to high

`pass` is a Python placeholder statement

`pass` is a Python placeholder statement

These are placeholders for two of the functions we will implement. Each of them just contains the function description (a string that is provided as the first statement in the function to explain what it does) and a new Python keyword that we've not seen before. The keyword is `pass`.

You can think of the `pass` keyword as a placeholder statement. We can use it anywhere that Python is expecting a statement. The `pass` statement doesn't actually do anything when the program runs. In this case, we will go back and

fill in the function later.

## Create a user menu

At the beginning of development, we agreed with the customer on the design of the user menu of the program. This method will print this menu and then allow the selection of the desired function.

```
EG8-05 Functions and Menu
```

```
menu='''Ice-cream Sales
```

```
1: Print the Sales
2: Sort High to Low
3: Sort Low to High
4: Highest and Lowest
5: Total Sales
6: Average Sales
7: Enter Figures
```

```
Enter your command: ''' Create the menu string
```

```
command=read_int_ranged(menu,1,7) Read in the command number
```

```
if command==1: Test for command 1
```

```
 print_sales()
```

```
else:
```

```
 if command==2:
 sort_high_to_low()
```

```
 else:
```

```
 if command==3:
 sort_low_to_high()
```

```
 else:
```

```
 if command==4:
 highest_and_lowest()
```

```
 else:
```

```
 if command==5:
 total_sales()
```

```
 else:
```

```
 if command==6:
 average_sales()
```

```
 else:
```

```
 if command==7:
 read_sales()
```

This code creates a menu string and then reads in a command number from the user. The command value is an integer in the range 1 to 7. The value in command number is used in conditional statements to select the function to perform that particular command.

This code uses the **if...else** construction to match command values with

functions. As you can see, we get a program that appears to be headed toward the right margin of the page. Each time we add another condition to the `else` part to test whether a command matches a particular value, we must indent the statements that follow. This is how we tell Python that the statements in the condition are controlled by that condition.

## Use the `elif` keyword to simplify conditions

Fortunately, Python provides a way that conditions of this form can be simplified. The `else if` statements can be combined into the single keyword `elif`.

```
EG8-06 Functions and Menu elif
command=read_int_ranged(menu,1,7) Read in the command
if command==1: Test to see if the command is number 1
 print_sales() Perform the print_sales function for command 1
elif command==2: Test to see if the command is number 2
 sort_high_to_low()
elif command==3:
 sort_low_to_high()
elif command==4:
 highest_and_lowest()
elif command==5:
 total_sales()
elif command==6:
 average_sales()
elif command==7:
 read_sales(10)
```

## Sort using bubble sort

The next thing we need to do is to write code that does some sorting. Sorting is something that computer programs spend a lot of time doing. However, as with other operations, you must tell a computer exactly how to do that sorting. A computer can't sort an entire list at once; it can work on only one item at a time. Looking at sorting programs is a good idea because doing so helps you understand how a complex problem can be broken down into a series of smaller steps.

Computer scientists talk a lot about *algorithms*. An algorithm expresses a series of actions that can be performed to solve a particular problem. Programming is really about taking an algorithm and converting it into a sequence of instructions that tells the computer what to do. This brings into focus one of the most important points of programming: If you don't have the algorithm, you can't write the program. In other words, if you don't know the sequence of steps that solves the problem, you can't make a program to solve the problem.

When it comes to sorting collections of data, there are several different algorithms, including the *bubble sort*. Bubble sorting progressively sorts lists one step at a time by comparing adjacent items and swapping items that are in the wrong order.

Next, we'll look at how bubble sorting works in detail and then convert the algorithm into Python code. (Bubble sorting works well for small data sets, but it is not always the best way to sort large amounts of data. If you're interested in how computers perform sorting, you can find many online resources.)

## Initialize a list with test data

While we're creating the sort program, it would be useful to have some test sales values with which to work. We could enter the sales values by hand each time, but that would be rather tiresome. Python lets us create a list of values very easily:

[Click here to view code image](#)

```
sales=[50, 54, 29, 33, 22, 100, 45, 54, 89, 75]
```

This statement creates a sales list that contains the values that were typed in above. A program can still append new values to the end of this list.

## Sort a list from high to low

**Figure 8-1** shows the list items—the test data—that we're using. Suppose we want to implement the behavior of the `sort_high_to_low` function, which will leave the list with the highest value at the item with the index 0 and the

lowest value at the index 9.

|    |    |    |    |    |     |    |    |    |    |
|----|----|----|----|----|-----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5   | 6  | 7  | 8  | 9  |
| 50 | 54 | 29 | 33 | 22 | 100 | 45 | 54 | 89 | 75 |

**Figure 8-1** List items

A Python program can perform only one comparison at a time. To sort the values, the program will keep making the list “less unsorted” until finally the items in the list are in the correct order. We could start by comparing the items at the beginning of the list:

[Click here to view code image](#)

```
def sort_high_to_low():
 """
 Print out a list of the sales figures sorted
 low to high
 """
 if sales[0]<sales[1]:
 # these two items are in the wrong order
 # the program must swap them
```

The `if` construction is controlled by a logical expression that compares `sales[0]` with `sales[1]`. If `sales[0]` is less than `sales[1]`, it’s in the wrong order (we want the largest values at the start of the list), and the two items need to be swapped because we’re sorting from highest to lowest.



## WHAT COULD GO WRONG

Swap two values in variables

Swapping two values in variables turns out to be a bit more complex than you might first think.

[Click here to view code image](#)

```
if sales[0]<sales[1]:
 # these two items are in the wrong order
 # the program must swap them
 sales[0]=sales[1]
 sales[1]=sales[0]
```

**Question:** This code looks like it might work, but in fact it is broken. Any idea why?

**Answer:** What the code actually does is put a copy of `sales[1]` into `sales[0]`. Here's why:

- The first statement puts the value of `sales[1]` into `sales[0]`. Both list items now contain `sales[1]` (in our case, 54).
- The second statement puts the value of `sales[0]` (which is 54, remember) back into `sales[1]`.
- So, both items end up with the same value in them, which is bad.

The way to fix this is to store the value of `sales[0]` temporarily so that we don't lose the value when we put `sales[1]` into it:

[Click here to view code image](#)

```
if sales[0]<sales[1]:
 # these two items are in the wrong order
 # the program must swap them
 temp=sales[0]
 sales[0]=sales[1]
 sales[1]=temp
```

The variable `temp` is used to hold this temporary value.

By swapping two items that are in the wrong order, we make the list a bit less out of order. Our program could now move on to the next pair of numbers and repeat the process to improve the sort still more.

[Click here to view code image](#)

```
if sales[1]<sales[2]:
 # these two items are in the wrong order
 # the program must swap them
 temp=sales[1]
 sales[1]=sales[2]
 sales[2]=temp
```

We could repeat this construction all the way to the end of the list, but it would be rather time-consuming to write the program. And when your customer with the ice-cream stands comes to you and says that she now has 50 sales outlets, you would be forgiven for bursting into tears.

However, if you take a careful look at the code used for swapping items, you'll notice something interesting. The action the code performs is the same for each pair of numbers; it is just that we move one position down the list to perform the second test. This means we can use a loop to count through the list and work through it with just a single `if` construction:

[Click here to view code image](#)

```
1. for count in range(0, len(sales)-1):
2. if sales[count]<sales[count+1]:
3. temp=sales[count]
4. sales[count]=sales[count+1]
5. sales[count+1]=temp
```



## CODE ANALYSIS

### Work through a list using a loop

This code uses some new features of Python and is worthy of careful study.

**Question:** Why have you used a `for` loop, rather than a `while` loop?

**Answer:** Either kind of loop will work fine. However, it turns out that the `for` loop version is slightly smaller. The range statement doesn't waste memory producing a list of values that the `for` loop then works through. You can think of a range as a "number generator" that will give you another value in the sequence each time you ask it for one.

**Question:** What does the `len` function on line 1 do?

**Answer:** The `len` function measures the length of a collection, such as a list, and returns the number of items in the list. I'm using it in this program because I don't want to have to change anything if the number of items in the list changes. This version of sorting will be able to sort any size of list because the loop is controlled by the length of the list being sorted.

**Question:** Why is the limit of `count` the same as the length of the list minus 1? (You can see this on line 1 of the program.)

**Answer:** This is because the bubble sort in the program compares an item in the list with the one after it. If we allowed the range to go all the way to the last item of the list, the program would try to compare the last item with the value beyond it, which doesn't exist.

The first time through the loop, the `count` variable will contain the value `0`, so

the test will compare `sales[0]` and `sales[1]`. Next time around the loop, `count` will contain the value `1`, so the test will compare `sales[1]` and `sales[2]`. The loop will continue down the list until it reaches the end.

[Click here to view code image](#)

```
EG8-07 Bubble sort first pass
def sort_high_to_low():
 """
 Print out a list of the sales figures sorted
 high to low
 """
 for count in range(0, len(sales)-1):
 if sales[count] < sales[count+1]:
 temp=sales[count]
 sales[count]=sales[count+1]
 sales[count+1]=temp
```

Above, you can see a version of the sort function that performs a single pass through the data. This function doesn't completely sort the list, but it does produce a result that is slightly less out of order than the original. **Figure 8-2** shows the contents of the list after the sort function has made a single pass through it.

| 0  | 1  | 2  | 3  | 4   | 5  | 6  | 7  | 8  | 9  |
|----|----|----|----|-----|----|----|----|----|----|
| 54 | 50 | 33 | 29 | 100 | 45 | 54 | 89 | 75 | 22 |

**Figure 8-2** Partially sorted list

You can see that some values have not moved much, while others have moved quite a bit. In general, all the high numbers have “bubbled” toward the left (the top of the list), while all the low numbers have moved toward the right. The value 22, which is the lowest number in the list, has been carried all the way to the right (the bottom) of the list. The value 100, which is the largest number in

the list, has been moved one step toward the top of the list. Numbers are “bubbling” toward their correct positions in the same way that bubbles go up and down in a fizzy drink. This is how the bubble sort technique gets its name.

We can complete the sort by making multiple passes through the data, swapping values that are out of order. With each pass through the list, the larger values bubble to the top as they are swapped with the smaller values that move toward the bottom. After just one pass through the list, we can be sure that the smallest value is now at the bottom of the list, and we can now make another pass to push the next smallest value into position. In a worst-case scenario, where the largest value was at the bottom of the list, it would take nine (or  $\text{length} - 1$ ) passes to bubble this value to the top.

Here is the code that performs multiple passes by using a loop to repeat the sort:

[Click here to view code image](#)

```
EG8-08 Bubble sort multiple passes
def sort_high_to_low():
 """
 Print out a list of the sales figures sorted
 high to low
 """
 for sort_pass in range(0, len(sales)):
 for count in range(0, len(sales)-1):
 if sales[count] < sales[count+1]:
 temp=sales[count]
 sales[count]=sales[count+1]
 sales[count+1]=temp
 print_sales()
```

The outer loop causes the program to make multiple passes through the code. The variable `sort_pass` is used to count the passes through the list. When the loops finish, the numbers will all be sorted from high to low.

[Click here to view code image](#)

```
Sales figures
Sales for stand 1 are 100
Sales for stand 2 are 89
Sales for stand 3 are 75
Sales for stand 4 are 54
Sales for stand 5 are 54
Sales for stand 6 are 50 Sales for stand 7 are 45
Sales for stand 8 are 33
Sales for stand 9 are 29
Sales for stand 10 are 22
```



## CODE ANALYSIS

### Improving performance

The sorting process works correctly, but it might be possible to improve the efficiency of the program.

**Question:** Is the program making more comparisons than necessary?

**Answer:** Yes. If you think about it, once the program has made one pass through the list, the smallest number is guaranteed to be at the bottom of the list. It's now a waste of time to check to see whether this value needs to be swapped with another value because it never will be. We can use the pass counter to make the program travel a shorter distance down the list with each pass:

[Click here to view code image](#)

```
for sort_pass in range(0,len(sales)):
 for count in range(0,len(sales)-1-
sort_pass):
 if sales[count]<sales[count+1]:
```

```
temp=sales[count]
sales[count]=sales[count+1]
sales[count+1]=temp
```

Take a careful look at this code. The crucial statement is the one controlling the inner loop:

[Click here to view code image](#)

```
for count in range(0, len(sales)-1-sort_pass):
```

This statement uses the value of `sort_pass` to reduce the distance down the list that each pass travels. This simple change roughly halves the number of comparisons that the program does.

**Question:** Is the program performing more passes through the list than necessary?

**Answer:** The answer is probably. The outer loop has been written to handle the worst-case scenario, in which the largest number is at the bottom of the list and needs to be bubbled all the way to the top. If the largest value is somewhere else in the list, the program will be making passes through the list when it is already sorted, which is a waste of computer time. It would be best if the sorting stopped as soon as the list was in the correct order. But how can the program detect that?

If the program makes a pass through the data and doesn't make any swaps, then the list must be in the correct order. We can add a flag to the program that is set when two items are swapped. If this flag is still clear after a pass, it means that the list is in order:

[Click here to view code image](#)

```
EG-09 Efficient Bubble Sort
for sort_pass in range(0, len(sales)):
 done_swap=False
```

```
for count in range(0,len(sales)-1-sort_pass):
 if sales[count]<sales[count+1]:
 temp=sales[count]
 sales[count]=sales[count+1]
 sales[count+1]=temp
 done_swap=True
 if done_swap==False:
 break
```

The program uses a Boolean variable called `doneSwap`. This variable is set to `False` before we make a pass through the data. It is checked after the pass, and if it is still false, the program breaks out of the loop that controls the passes through the list.



## MAKE SOMETHING HAPPEN

### Sort alphabetically

The bubble sort algorithm works for strings as well as for integers, and we saw in [Chapter 5](#) that the Python relational operators work between strings. Now see if you can make your party guest program display the guest names for your party in alphabetical order. You could use this program any time you want to sort some words into order.

### Sort a list from low to high

Our program also needs a low-to-high display of the sales data. Implementing this request turns out to be quite easy. We just need to change the less-than operator to a greater-than operator in the statement in the middle of the loop that compares values as the loop works through each of the items.

[Click here to view code image](#)

```
EG8-10 Sort low to high
if sales[count]>sales[count+1]:
 temp=sales[count]
 sales[count]=sales[count+1]
 sales[count+1]=temp
```

## Find the highest and lowest sales values

You might also want to find the highest and lowest sales in the set of results. Before you write the code to do this, it's worth thinking about the best algorithm to use. In this case, the program can implement an approach very much like one that a human would use. If you gave me some numbers and asked me to find the highest value, I would compare each number with the highest value I had seen so far and replace the current highest value each time I found a larger one. In programming terms, this algorithm would look a bit like the following. (This is not Python as such; a description like this is sometimes called *pseudocode*. It looks something like a program, but it just expresses an algorithm; it does not run as part of the program.)

[Click here to view code image](#)

```
if(new value > highest I've seen)
 highest I've seen = new value
```

When the program starts, we can set the “highest I’ve seen” value to the value of the item at the start of the list (because this is the highest value we’ve seen at the start of the process). We could then use a **for** loop to work through the items, checking each one against the current highest value.

[Click here to view code image](#)

```
highest=sales[0]
for sales_value in sales:
 if sales_value>highest:
```

```
highest=sales_value
```

We can use the same approach to find the smallest value. This time, we're looking for values that are smaller than the smallest one we have seen so far.

[Click here to view code image](#)

```
lowest=sales[0]
for sales_value in sales:
 if sales_value<lowest:
 lowest=sales_value
```

However, because we're already making a pass through the list to find the largest value, we can make the program slightly more efficient by using the same loop to find the highest and lowest in a single pass through the data. (Note that at the start of the loop, the initial item in the list is both the highest and lowest value.)

[Click here to view code image](#)

```
def highest_and_lowest():
 """
 Print out the highest and the lowest sales
 values
 """
 highest=sales[0]
 lowest=sales[0]
 for sales_value in sales:
 if sales_value>highest:
 highest=sales_value
 if sales_value<lowest:
 lowest=sales_value
 print('The highest is:', highest)
 print('The lowest is:', lowest)
```

## Evaluate total and average sales

To work out the total of sales, the program must add all the items in the list. You can do this by using another `for` loop or by adding code to the loop that we also use to find the highest and lowest sales values.

[Click here to view code image](#)

```
EG8-12 Total Sales
def total_sales():
 """
 Print out the total sales value
 """
 total=0
 for sales_value in sales:
 total = total+sales_value
 print('Total sales are:', total)
```

Once we have the total sales, we can calculate the average sales value. Of course, the average of a set of numbers is the sum of the numbers divided by the number of items in the list. For example, if this list contains the four numbers 4, 6, 10, and 12, the average is determined by adding the numbers  $4 + 6 + 10 + 12 = 32$  and then dividing the sum by four (the total number of items in the list:  $32/4 = 8$ ). With the total number of sales calculated, working out the average is very easy.

[Click here to view code image](#)

```
EG8-13 Average Sales
def average_sales():
 """
 Print out the average sales value
 """
 total=0
 for sales_value in sales:
 total = total+sales_value
 average_sales=total/len(sales)
 print('Average sales are:', average_sales)
```

The code to work out the total is the same as the code you've already seen. The value `average_sales` is set to the total sales divided by the number of sales values, which we can get from the length of the sales list.

## Complete the program

We now have all the features we need to create the finished application, but we still need to complete the logic. At this point, we can go back to the storyboards that we created with the customer. The storyboards give us the sequence we want. Essentially, the program breaks down into two loops, an outer loop and an inner loop. The outer loop runs forever. When it starts running, it first allows the user to enter some data. Once the program has some data with which to work, it performs the inner loop. This loop repeatedly reads in a command and acts on it. The following code shows the structure of the nested loops.

[Click here to view code image](#)

```
EG8-14 Complete Program

Start by reading in the sales
read_sales(10)

Now get the command from the user

menu='''Ice-cream Sales

1: Print the sales
2: Sort High to Low
3: Sort Low to High
4: Highest and Lowest
5: Total Sales
6: Average sales
7: Enter Figures
Enter your command: '''

Now repeatedly read commands and act on them
while True:
 command=read_int_ranged(menu,1,7)
```

```
if command==1:
 print_sales()
elif command==2:
 sort_high_to_low()
elif command==3:
 sort_low_to_high()
elif command==4:
 highest_and_lowest()
elif command==5:
 total_sales()
elif command==6:
 average_sales()
elif command==7:
 read_sales(10)
```

## Store data in a file

What if your customer wants the program to be able to store and retrieve sales values so that she doesn't need to enter them more than once? In this example, you'll add two new menu items: Save Sales and Load Sales.

[Click here to view code image](#)

### **Ice-cream Sales**

- 1: Print the Sales
- 2: Sort High to Low
- 3: Sort Low to High
- 4: Highest and Lowest
- 5: Total Sales
- 6: Average Sales
- 7: Enter Figures
- 8: Save Sales
- 9: Load Sales

This is the new menu display. The two new commands are numbers 8 and 9. The

commands themselves are implemented by two new functions.

[Click here to view code image](#)

```
EG8-15 Load and Save
def save_sales(file_path):
 """
 Saves the contents of the sales list in a file
 file_path gives the path to the file to save
 Raises file exceptions if the save fails
 """
 print('Save the sales in:', file_path)

def load_sales(file_path):
 """
 Loads the sales list from a file
 file_path gives the path to the file to load
 Raises file exceptions if the load fails
 """
 print('Load the sales from:', file_path)
```

These are the “stub” versions of the functions. Each function has one parameter, which is the path to the file that will be used to store the sales values. I’ve added `print` commands so that we can test the program to make sure that the functions can be selected by a user of the program. If you run the example program, you will see that these functions can be selected and run from the menu. Now we must fill in the contents of each function.

## Write into a file

When a program interacts with a file, Python creates an object that represents a connection to that file. The function `open` creates one of these objects. If you were lucky enough to have a personal assistant, you could ask him to “Write a letter to the boss at Microsoft,” and he would take down everything you say, put it in a letter, and send it off. Your personal assistant would understand commands such as “write a letter” and “read me a report,” and the `open` function is very similar.



The `open` function accepts two arguments. The first argument is a string containing the path to the file to be opened. In its simplest form, a path can be just the name of the file. The above statement opens a file called `test.txt`. The second argument is a string containing the *mode* of the connection to the file. This controls how the file will be used. The mode string `'w'` means “write.” The statement above will prepare a file for writing. If the file already exists, the `open` function will erase the existing file before writing new content into it.



## WHAT COULD GO WRONG

### It's very easy to overwrite an existing file

Most programs are very careful to prevent a user from overwriting important files. An “Are you sure?” message will appear if someone tries to save a new file over an existing one. However, the Python `open` function doesn’t do any of this. If you really want to stop files from being overwritten by your programs, you’ll need to add this behavior yourself. The `os` library supplied with Python contains a `path` library (libraries can contain libraries) that can help with this as you can see in the Python code below:

[Click here to view code image](#)

```

import os.path
if os.path.isfile('text.txt'):
 print('The file exists')

```

The `isfile` method accepts a file path as an argument and returns `True` if that file is found.

Once the file has been opened, our program can begin storing lines of text in the file. The program can do this by calling the `write` method provided by the file object.

[Click here to view code image](#)

```
output_file.write('First line\n')
output_file.write('Second line\n')
output_file.close()
```

When a program has finished writing to a file, it must call the `close` method on the file. This method completes any unfinished writes and releases the file. It's very important that your program closes a file when it has finished writing. There are two reasons why closing is important. First, closing a file ensures that all the data is stored. Second, closing a file makes the file available for use by other programs. A file opened for writing is “locked” and cannot be accessed until the write operation is complete. Trying to write to a file that has been closed will result in an error.

[Click here to view code image](#)

```
EG8-16 File Output

output_file=open('test.txt','w')
output_file.write('line 1\n')
output_file.write('line 2\n')
output_file.close()
```

The above program creates a file called `test.txt` and writes two lines of text into the file. The file is created in the folder containing the program when it runs.



## CODE ANALYSIS

### File writing

**Question:** Why have you called the `write` behavior a method? Isn't it a function?

**Answer:** In Python, a method is much like a function, except that it's created as part of an object. A function is code that exists outside any object. The `print` and `input` functions are not part of an object. A program can just use them directly. However, the `write` method is part of the file writing object. If we want to use `write`, we must have a file object. You can think of functions as "things a program can just do" and methods as "things an object can do for us." If the `write` behavior was a function, a program would need some way of knowing which file was being written. Making the `write` method part of a file object makes it very easy to work with more than one file at the same time. We can just use the `write` method on the different file objects. We'll investigate objects and methods in detail later in the book.

**Question:** What does the `\n` mean at the end of the strings?

**Answer:** We've seen `\n` before. It's the escape sequence that means "new line." The `write` function doesn't automatically take a new line at the end of a write. If we want to write a new line in a file, we must do that explicitly. This behavior is different from how we've seen the `print` function work. The `print` function automatically takes a new line at the end of every print, but with the `write` method you must explicitly ask for one.

**Question:** Where is the file `test.txt` actually created?

**Answer:** The file is created in the same folder that holds the running Python program. In other words, if I had a folder called My

Programs, which contained a Python program called MakeFiles, when I run the MakeFiles program, any files it creates will be stored in the My Programs folder.

Folders (or directories) are used to organize information we store on the computer. Each file you create is placed in a particular folder. In Windows, several folders are automatically created—Documents, Music, Pictures, and Videos. You can create your own folders inside these folders.

A path, or file path, refers to the location in which a file is stored, such as C:/Documents/Finances/MyFinances.xls. The file MyFinances.xls is stored in the Finances folder, which is stored inside the Documents folder, which is found on the C drive. The path to a file can be broken into two parts: the location of the folder and the name of the file itself. If you don't give a folder location when you open a file (as we have been doing with the file test.txt) then Python assumes that the file being used is stored in the same folder as the running program.

If you want to use a file in a different folder (which is a good idea, because data files are hardly ever stored in the same folder as the program that opens that file), you can add path information to a file name:

[Click here to view code image](#)

```
path = 'c:/data/2017/June/sales.txt'
```

The above statement creates a string variable that contains the path to a file called sales.txt. This file resides in the folder June, which is stored in the folder 2017, which is stored in the folder data on drive C.

The forward slash (/) characters in the string serve to separate the folders along the path to the file. Note that if you're using a Windows PC, you might be used to using the backslash (\) character to separate items of a path. In Python, you must use the forward

slash character, as above.

**Question:** Can any program use a file written from a Python program?

**Answer:** Yes. You can open the file test.txt with any application on your machine. The Python file handling is always based on the file functions of the underlying operating system.

**Question:** Can I add lines on the end of a Python file?

**Answer:** Yes, you can. If you open the file in **append** mode by using the mode string '**a**' then any writing you do will be appended to the end of an existing file. If the file you're appending to doesn't exist, it is created automatically, just as it would be for the '**w**' file mode.

## Write the sales figures

We can now fill in the **save\_sales** function. You can find this function in the example file **EG8-17 Save sales**.

[Click here to view code image](#)

```
1. def save_sales(file_path):
2. """
3. Saves the contents of the sales list in a
file
4. file_path gives the path to the file to
save
5. Raises file exceptions if the save fails
6. """
7. print('Save the sales in:', file_path)
8. # Open the output file
9. output_file=open(file_path, 'w')
10. # Work through the sales values in the
list
11. for sale in sales:
```

```
12. # write out the sale as a string
13. output_file.write(str(sale) + '\n')
14. # Close the output file
15. output_file.close()
```



## CODE ANALYSIS

### The `save_sales` function

The `save_sales` function is the most complicated function we've seen so far and is worth close study. However, before we start considering specific questions, it's important to consider the purpose of `save_sales`. The program contains a list of sales figures. We want to store that list in a text file. We have a method called `write` that we can use to write a string of text into a file. So, the `save_sales` function must take each sales figure and write it into a file.

**Question:** What does the `str` function do? Why are we using it?

**Answer:** You can find the `str` function used in the statement on line 13. The function is used to take a number (a sales value) and convert it into a string of text. We don't have to convert things into strings with the `print` function because `print` behaves differently than the `write` method. The `print` function can accept any kind of value and will print it as a string. The `write` method must be given a string to write to the file. This means that our program must explicitly convert numbers into text before passing them into `write`. The `str` function performs this conversion.

**Question:** Why can't we just write out the sales list as one object?

**Answer:** A list is a container, which provides methods, such as `append`, that can be used to add things to a list. However, the list

doesn't contain any code that could be used to write its contents to a file. Our program must take each item from the list and write it out. When reading back the list, the program must build up a list from the items in the file.

## Read from a file

Reading a file is very much like writing a file. The program creates an object that provides the connection to the file and then calls methods on the object to perform the required actions. A program can open a file for reading by using the mode string '`r`'.

[Click here to view code image](#)

```
input_file=open('test.txt','r')
```

This statement creates an object that can be used to read items from a file. Our program can treat the file object as a collection of lines that can be used to control a `for` loop construction:

```
for line in input_file:
 print(line)
```

Work through each line in the file  
Print the line

The `for` construction above will work through the lines in the file and print each one. The loop ends when the last line has been read from the file.

```
input_file.close()
```

Once the file has been read, it must be closed.

[Click here to view code image](#)

```
EG8-18 File Input
```

```
input_file=open('test.txt', 'r')
for line in input_file:
 print(line)
input_file.close()
```

This is the complete file printer program. It opens the file for reading, prints out each line, and then closes the file. If you run this program on the text file we created earlier, you'll see that the contents of the file are printed.



## CODE ANALYSIS

### Reading from files

**Question:** If you look at the following output, you'll notice that there are empty lines after each line of the text. Why is this?

```
line 1
```

```
line 2
```

**Answer:** This is because each line read from the file has a new line character ('`\n`') on the end. We added the new line when we wrote the file. The new line on the end of the line is read back in when Python reads the file. The `print` function adds a new line at the end of each line when it prints the line, so the text as printed has two new lines at the end of each line.

There are two ways we can fix this problem. One way would be to tell the `print` function not to add a new line when it prints the text.

```
print(line, end='')
```

The `end` parameter to the `print` function specifies the character to be printed at the end of each line. The parameter has a default value of new line character (`'\n'`), but when we call the function we can give an argument that sets a new value for the line end. In the above statement, I've set the line end to an empty string, so that only the line ending from the input string is printed.

A better way to solve this problem is to remove the line feeds from the line that was read from the file. The `strip` method asks a string to return a version of itself minus any “whitespace” characters. Whitespace characters are all the spaces that are not visible when printed, including spaces and tabs, and can appear at the beginning or end of a string.

```
line = line.strip()
```

The above statement creates a version of `line` with no whitespace characters. Whitespace characters inside the string, such as the spaces between words, are preserved. Only the start and end of the string is affected.

Software developers talk about “conditioning” input to make sure there are no unexpected items in the text. The `strip` method is useful for making sure that there are no nonprintable characters at the start or end of text that is read in. If you want to strip whitespace only from the left or right ends of the string, you can use the `lstrip` or `rstrip` methods, respectively.

**Question:** Why do we have to close the file we're reading?

**Answer:** Reading from a file will never change the contents of the file, so forgetting to close the file won't mean your program will damage any data. However, you should still close a file after you've finished using it so the file is available for other programs to use.

You might also find that your computer refuses to shut down if it thinks files are open.

**Question:** What would happen if I tried to write to a file that I had opened for reading?

**Answer:** This will result in an exception being raised. However, you can use the mode string '`r+`' to open a file for both reading and writing. Reading and writing a file from the same program is quite hard to do. You must make sure that writing the file doesn't corrupt data that's already there. If the program writes a line longer than the one in the file, it will corrupt information on the following line. A program will not normally change the data in a file. Instead it will load data from the file, update the data, and then re-write all the data into the file.

**Question:** Can a program read an entire file at once?

**Answer:** Yes. An input file object provides a `read` method that reads the entire contents of the file in one go. Python strings can hold very large amounts of text, so you can read large files this way. The line endings will be preserved in the string that is read. We can use the `read` method to create a very simple file printing program:

[Click here to view code image](#)

```
input_file=open('test.txt','r')
total_file=input_file.read()
print(total_file)
input_file.close()
```

You might use this method, for example, if you were creating a file copying program.

## Read the sales figures

We can now fill in the `load_sales` function.

[Click here to view code image](#)

```
1. def load_sales(file_path):
2. """
3. Loads the sales list from a file
4. file_path gives the path to the file to
load
5. Raises file exceptions if the load fails
6. """
7. print('Load the sales from:', file_path)
8. # Clear the sales list 9. sales.clear()
10. # Open the file for input
11. input_file=open(file_path,'r')
12. for line in input_file:
13. line=line.strip()
14. sales.append(int(line))
15. input_file.close()
```



## CODE ANALYSIS

### The `load_sales` function

This method is like the reverse of the `save_sales` function, which worked through the sales list adding sales figures to the file. The `load_sales` function works through the input file adding figures to an empty sales list.

**Question:** What does the `int` function do?

**Answer:** The `int` function is used on line 14. You can think of it as being the reverse of the `str` function that was used in

`save_sales`. The `str` function can convert a number into a string. The `int` function converts a string into a number. We've used it before when we took strings entered by the user and converted them into numbers.

**Question:** What would happen if the input file was empty?

**Answer:** It turns out that this would work correctly, in that the statements in the `for` loop would not be performed, so the code would create an empty sales list.

## Deal with file errors

Programs that deal with files also need to deal with the possibility that things might go wrong. A file might not be found, a USB drive might be unplugged, or the user might enter the wrong file name. Two things are very important to us if an error occurs:

1. No files should be left open.
2. The user must be made aware that something has gone wrong.

When a program action involving a file fails, the failure will raise an exception. We first saw exceptions when we converted strings of text into numbers using the `int` function. We discovered that if the string doesn't contain digits that make up a number, the `int` function fails and raises an exception. The same mechanism is used to alert a program to failure when using files. A program can deal with exceptions by using the `try...except` construction, so we can write code such as the following version of the file saving code from the sales program. It works through the sales list and saves each item, but any exceptions raised when using the file are caught and cause a message to be printed.

```
1. try: Start of the try...except construction
2. output_file=open(file_path, 'w') Code that might raise exceptions
3. for sale in sales:
4. output_file.write(str(sale)+'\n')
5. output_file.close()
6. print('File written successfully') This statement is reached only
 if no exceptions are raised
7. except:
8. print('Something went wrong writing the file') Code that handles exceptions
```



## CODE ANALYSIS

### Dealing with file handling exceptions

The code that performs the file writing is enclosed in a `try... except` construction. If any of the file actions raise an exception, the `except` part of the `try` construction is performed. This looks like it might solve our problems, but we need to take a closer look.

**Question:** In what circumstances will code in the exception part be executed?

**Answer:** If any of the file functions on statements 2, 4, or 5 raise an exception, the code in the `except` part of the construction will be obeyed. So, we see the error message only when an error occurs.

**Question:** In what circumstances will the “File written successfully” message be printed?

**Answer:** This message is printed only if every step in the file writing—including closing the file—completed successfully.

**Question:** I can see that the error message is always printed if a file error occurs, but will the output file always be closed if an error occurs?

**Answer:** No. That is the problem with this code. If a write action fails, the execution will switch straight to the `except` behavior,

leaving the file open. This is a problem. One way to deal with this would be to close the file in the exception handler code as well, but a better way is to add a **finally** part to the construction.

```
try:
 output_file=open(file_name,'w')
 for sale in sales:
 output_file.write(str(sale)+'\n')
except:
 print('Something went wrong writing the file')
finally:————— Statements in the finally part are always obeyed
 output_file.close()
```

The **finally** part is an additional item that we can add to a **try** construction. It contains code that is always obeyed, no matter what happens. In the above code, we can be sure that the output file is closed regardless of whether an exception was raised in any part of the **try** construction.

## Use the **with** construction to tidy up file access

The **try...except...finally** construction is one way to deal with file errors. However, I don't think it's perfect because I still must remember to make my program close a file when I've finished with it. It turns out that Python will get around to closing a file that my program leaves open, but I can't be sure when this will happen.

A program that forgets to close a file could exhibit the worst kind of bad behavior. It might fail, but only every now and then. The user might find that if they tried to reopen a file that they had just written, their program would fail because Python had not gotten around to closing it. At other times, however, the program would work perfectly.



### PROGRAMMER'S POINT

#### Intermittent faults are the worst kind to fix

If someone calls me and tells me that the program I wrote for them has completely failed, the solution I must implement probably won't be that time consuming. That kind of fault is often

surprisingly easy to fix. However, I dread getting the message that my program sometimes goes wrong. That means that before I can fix the fault, I must make it reoccur. I'm prepared to put a lot of extra effort into a design to try to remove the possibility of any intermittent faults.

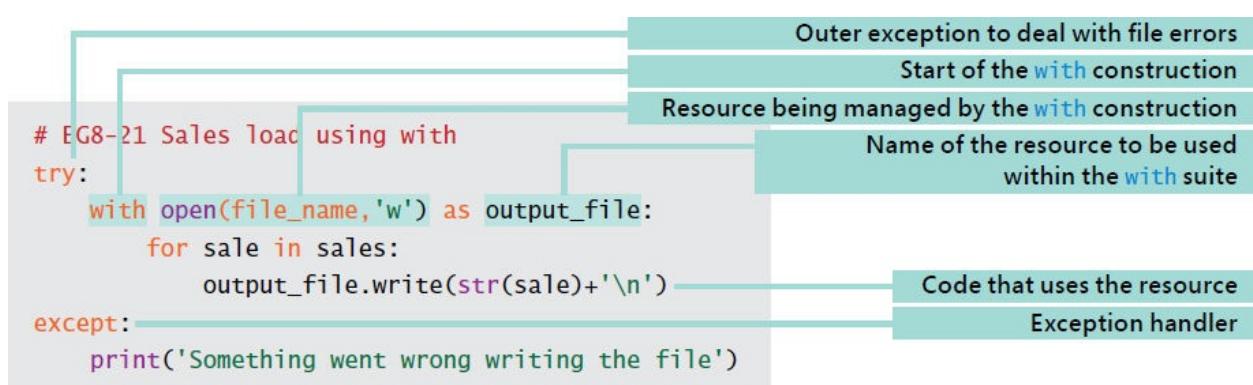
The designers of Python were concerned about this. They wanted a way that programmers could make sure that resources used by programs are obtained and released in a reliable way. So, they added the **with** construction to the language.

The **with** construction, shown in **Figure 8-3**, provides a protocol for obtaining and releasing resources. A given service can be written to work with the **with** construction so that the instruction to release the resource is automatically performed without the programmer needing to do anything. For now, we don't need to know how to make services that can be managed by the **with** construction, we just must understand how to use the **with** construction when working with files.

|                                                |                                                                         |    |                                                           |   |                                                   |
|------------------------------------------------|-------------------------------------------------------------------------|----|-----------------------------------------------------------|---|---------------------------------------------------|
| with<br>(start of<br>the with<br>construction) | expression<br>(expression that<br>generates the<br>resource to be used) | as | name<br>(name to represent<br>the resource being<br>used) | : | suite<br>(statements<br>that use the<br>resource) |
|------------------------------------------------|-------------------------------------------------------------------------|----|-----------------------------------------------------------|---|---------------------------------------------------|

**Figure 8-3** Anatomy of a **with** construction

A program uses the **with** construction to obtain an object that will provide a service. In the case of the following code sample, the object being obtained has the name **output\_file**. The **with** construction will activate an “enter” behavior for the object it is working with when it obtains the resource. In the case of a file object, this behavior will cause the file to be opened.



When the program exits, the statements inside the `with` construction automatically activate the “exit” behavior on the resource the `with` construction is managing. This is used by the file object to close the file it’s using. The `with` construction ensures that the exit behavior is always performed, which means that the programmer doesn’t need to remember to close a file; it closes automatically.

In the above code example, the whole `with` construction is enclosed in a `try...except` construction. This is because `with` doesn’t deal with raised exceptions; it just manages an object. If the file write raises an exception, the `with` construction will first perform the exit behavior (which will close the file) and then pass on the exception to be picked up by the error-handling code.



### MAKE SOMETHING HAPPEN

#### Record a list with a `save` function

Add a `save` function to your party guest program so that you can record a list of people who attended your party.

## Store tables of data

A list is a one-dimensional data structure. In other words, it has only a length. However, sometimes a program needs to store more than one dimension of data. For example, let’s say that the customer for the ice-cream sales analysis program has come back and told you how pleased she is with the code and that she’s thought of some improvements. She would like to be able to store sales for different days of the week so she can keep track of sales over time. She has drawn out a table that shows how the data would look.

| MONDAY | TUESDAY | WEDNESDAY | ... |
|--------|---------|-----------|-----|
|        |         |           |     |

|        |    |    |    |  |
|--------|----|----|----|--|
| Stand1 | 50 | 80 | 10 |  |
| Stand2 | 54 | 98 | 7  |  |
| Stand3 | 29 | 40 | 80 |  |
| ...    |    |    |    |  |

You can think of the sales list you've used up to this point as one column in the table (for example, the sales for Monday). The user can enter sales figures for that day, but what the customer now wants is a way for the program to store successive columns of sales figures for subsequent days.

One way to do this would be to have multiple lists, called Monday, Tuesday, Wednesday, and so on. However, this arrangement seems a bit like using individual variables for each sales figure, the problem we addressed earlier by using a list. Working with the data stored in this way would be difficult. For example, it would be very hard for a program to find the highest sales for the week because the program would have to consider each list individually.

We can solve this problem by creating a list that contains other lists:

```
mon_sales=[50,54,29,33,22,100,45,54,89,75]
tue_sales=[80,98,40,43,43,80,50,60,79,30]
wed_sales=[10,7,80,43,48,82,33,55,83,80]
thu_sales=[15,20,38,10,36,50,20,26,45,20]
fri_sales=[20,25,47,18,56,70,30,36,65,28]
sat_sales=[122,140,245,128,156,163,90,140,150,128]
sun_sales=[100,130,234,114,138,156,107,132,134,148] Lists for each day of the week
```

The Python statements above create seven lists of sales figures, one for each day of the week. I've created some sample data for each week that we can use to illustrate how the solution might work.

```
week_sales=[mon_sales,tue_sales,wed_sales,thu_sales,fri_sales,sat_sales,sun_sales]
```

List containing the entire week's sales.

The Python statement above creates a list called `week_sales` that contains all these lists. It is a list of lists. You can think of each individual list as a row. You can think of a list of lists as collection of rows. When a program wants to refer to an individual sales value, it must specify the row, followed by the position in that row of that value.

[Click here to view code image](#)

```
print(week_sales[1][0])
```

This statement would print the value 80, which is the Tuesday sales for stand 1. (Remember that list index values always start counting from zero.)

In the statement above, we created the `week_sales` list using a single statement to add all the sales values at the same time. We could have created this “list of lists” by appending each list of sales figures.



## CODE ANALYSIS

### Inadequate index values

**Question:** Which of the following statements would fail when the program runs?

[Click here to view code image](#)

```
Statement 1: week_sales[0][0] = 50
```

[Click here to view code image](#)

```
Statement 2: week_sales[8][7] = 88;
```

[Click here to view code image](#)

```
Statement 3: week_sales[7][10] = 100;
```

**Answer:** Statement 1 is completely correct (as it should be; it is used in the text). Statement 2 will fail because the first index (the day of the week) has the value 8. The `week_sales` list, however, contains seven items, one for each day of the week, so this statement is trying to access a nonexistent item. Statement 3 is also invalid. Because items are indexed starting at zero, this statement attempts to go beyond both lists, and the program will fail as a result. If we really want to access the item at the bottom right corner of the table, we should access the item `week_sales[6][9]`.



## PROGRAMMER'S POINT

### Make it easy to test your programs

My experience as a programmer has been that if testing your program is very difficult, you just don't do it. Unless the tests are really easy, or better yet completely automatic, you won't bother with them.

It took me just a few minutes to create the test data above. In the finished program, I would create a function called `make_test_data` that I could call to create test data with which to easily test my program. If I was serious about testing, I'd even get creative with the random number generator to create large amounts of data to test my programs.

Making things easy to test even extends as far as video games. Rather than having to play for half an hour to get to the level you want to test, you should have some way of skipping levels.

Whenever you find yourself repeating a pattern of steps in order to test your program, consider how you can automate this action.

## Use loops to work with tables

The Python `for` loop construction can work through lists of lists just as easily as it can work through lists of individual values. If we want a program that will calculate the total sales for a week, we can do this as follows:

[Click here to view code image](#)

The outer loop works through the entire week, pulling out the list for each day. The inner loop works through the list for the day. Each successive value is added to the total.

Using a loop like this is called *nesting*. (We've put loops inside one another before, which is how the program repeatedly reads and acts on commands.) Here we have an outer loop that goes around seven times (once for each day), and an inner loop that is performed 10 times (once for each ice-cream stand). When the loop has completed, the program will have put all the values into the list.



## CODE ANALYSIS

### Loop counting

**Question:** How many times will the statements inside the two loops be obeyed?

**Answer:** They will be obeyed 70 times. The outer loop is obeyed 7 times, the inner loop is obeyed 10 times. To get the total number of times around the loop, you multiply one by the other, giving 70 times around the loop.

**Question:** How would you change this program so that it could handle more than one week's worth of sales?

**Answer:** We can add more days to the list. From the point of view of the table, this would be equivalent to adding more rows.

**Question:** How would we add a day's worth of sales to the weekly list?

**Answer:** To do this, we would need to read in a list of values and then add it to the weekly list:

[Click here to view code image](#)

```
Read in a set of sales values
read_sales(10)
Add the daily sales figures to the week
week_sales.append(sales)
```

You could store the sales figures for an entire year in a single list rather than just seven days' worth. You could also add extra loops to the `save` and `load` functions so that the data could be saved to a file and loaded.

## More than two dimensions

If you ever need to represent a large number of tables, you can move up to a list that contains a list of lists. The best way to visualize this type of list is as a pile of pages, with one page for each week. The third dimension would be the number of the page containing the results for that week.

The following statement shows how a program would add a week's worth of sales to a list that held a series of entries, one for each week.

[Click here to view code image](#)

```
annual_sales.append(week_sales)
```



### PROGRAMMER'S POINT

## Keep your dimensions low

In all my years of programming, I've never had to use any more than three dimensions, and I've only ever used three dimensions a couple times (and one of those occasions was to create a "3-D Tic-tac-toe" game).

If you find yourself having lists that contain lists of lists, I would suggest that you're trying to do things the wrong way and that you should step back from the problem and think about how your data fits together. Later in the book, you'll see ways to build classes that contain a number of related data items. It's often much easier to make a one-dimensional list from such structures rather than move into multiple dimensions.

The computer is quite happy to work in very large numbers of dimensions as long as it doesn't run out of memory. However, I've found that the same can't be said for programmers.

## Use lists as lookup tables

Now that you know how to store data in the program, you can discuss with the customer again how the program is supposed to be used. The customer is quite impressed with the data storage plans, but she now raises an interesting issue. She is concerned that when sales figures are entered, the program doesn't show the user the day the sales figures are for. The program will work perfectly correctly, but it might be confusing to use. What she would like is for the program to display the day being entered.

[Click here to view code image](#)

Enter the Monday sales figures for stand 2:

To do this, the program must display a message that identifies the day of the week. A program could use a variable called `day_number` to count through the days as they are read. The variable could start at 0 for Monday and then count to 6 for Sunday. A collection of `if` conditions could be used to convert the day number to a string:

[Click here to view code image](#)

```
EG8-23 Day Name If
if day_number==0:
 day_name='Monday'
elif day_number==1:
 day_name='Tuesday'
elif day_number==2:
 day_name='Wednesday'
elif day_number==3:
 day_name='Thursday'
elif day_number==4:
 day_name='Friday'
```

```
elif day_number==5:
 day_name='Saturday'
elif day_number==6:
 day_name='Sunday'
```

This code would work fine, but it would be tedious to type in, and there's a good chance that you would make a mistake. Python provides a much easier way to do this. You can create a preset list and use it as a lookup table.

[Click here to view code image](#)

```
EG8-24 Day Name List
day_names=
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', '

day_name=day_names[day_number]
```

When the program runs, the list is created with the preset contents. There will be an item for each day of the week. The program can now directly convert a day value in the range 0–6 to the matching day.

Lookup tables are very useful. They can be used to create *data-driven* applications—programs that work by using built-in data rather than hard-wired behaviors.

## Tuples

A list is a very powerful thing. It provides a complete set of behaviors that allow programs to append new items to the list and change its contents. However, to decode day numbers into day names we don't need anything as powerful as a list. Once we create the lookup table, we don't want it to be changed. In fact, it would be very useful if we could prevent changes to the list of names. We don't want a rogue programmer to be able to do something like this:

[Click here to view code image](#)

---

```
EG8-25 Day Name Tuple
day_names[5]='Splatterday'
```

This change would mean that the program would now refer to “Saturday” as “Splatterday,” which some people might think is hilarious but our customer would not like very much. It turns out that a Python program can contain data collections that cannot be modified after creation.

In [Chapter 6](#), you were introduced to a data collection called a *tuple*. A tuple is much like a list, but with one significant difference: It is not possible to change the contents of a tuple. Python has a special word for this behavior: *immutable*. We say that “tuples are immutable.”

For our purposes, we don’t want the day decode list to change. An attempt to change the contents of a tuple containing the `day_names` would cause the program to fail.

[Click here to view code image](#)

```
Traceback (most recent call last):
 File "C:/Ch 08 Collections/code/samples/EB8-26
 Day Name Tuple.py", line 9, in <module>
 day_names[5]='Splatterday'
TypeError: 'tuple' object does not support item
assignment
```

A tuple is created as a list of items enclosed in parentheses.

[Click here to view code image](#)

```
day_names=
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', '
```

A program can read values from the tuple, and work through it using a `for` loop in the same way as a program would use a list. However, the contents of the

tuple cannot be changed.

Tuples are very useful when a program wants to work with values more complicated than simple integers or floating-point values. For example, you might have a function in your program that could be used by a pirate to tell you where the treasure is buried. The function might need to return the name of a landmark and the number of paces north and east that we need to walk to find the place to dig a hole. We have seen that a function can return only one value, but a function can also return a tuple.

[Click here to view code image](#)

```
def get_treasure_location():
 # get the location from the pirate
 return ('The old oak tree', 20, 30)
```

The function `get_treasure_location` returns a tuple that contains three values. The first is a string, the following two values are integers. A program can use the return value from the function to display the position to dig:

[Click here to view code image](#)

```
location=get_treasure_location()
print ('Start at',location[0],
 'walk',location[1], 'paces north and',
 location[2], 'paces east')
```

The index values identify the particular items in the tuple. The item with index 0 is the first item in the tuple, in this case the name of the landmark, the old oak tree. The second two items are the north and east number of paces values, respectively, and they are index values 1 and 2. A tuple is a terrific way to return data like this, as it also stops a program from being able to tamper with any of the items in the data.

Many times, you need a quick way of creating a value comprising a few items (for example coordinate values x and y, or color values that contain amounts of red, green, and blue). A tuple is very useful in these situations.



## WHAT COULD GO WRONG

### Take care with your tuple indexes

It's important that Python programs use the result of the `get_treasure_location` correctly. There is a kind of "contract" between the function and any code that calls the function. The contract states that "The first item of the tuple is the description, the second the distance north, and the third the distance east." If a program calling the `get_treasure_location` function uses the wrong index values, it will display incorrect instructions.

[Click here to view code image](#)

```
EG8-26 Pirate Treasure Tuple
def get_treasure_location():
 """
 Get the location of the treasure
 returns a tuple:
 [0] is a string naming the landmark to
 start
 [1] is the number of paces north
 [2] is the number of paces east
 ''' # get the location from the pirate

 return ('The old oak tree', 20, 30)
```

Above you can see that I have added details of the parameters to the description of the function. I don't generally use tuples to return values that are more complex than a few items. Later in the book, we'll discover how to return objects that have a better-defined structure.

If a function returns a tuple or a list, we can use a different format to call the

function, which makes it slightly easier to unravel the results it returns.

[Click here to view code image](#)

```
EG8-27 Pirate Treasure Tuple Function
landmark, north, east = get_treasure_location()
print ('Start at',landmark, 'walk', north,'paces
north and', east,'paces east')
```

This call of `get_treasure_location` would place the three values returned in the tuple into the variables `landmark`, `north`, and `east`, respectively.

## What you have learned

In this chapter, you've learned how to store large amounts of data in a Python program using lists. A list can hold several different values. Each value in the list is called an item. A program can add items to a list by using the `append` method, or it can create a list containing a set of items. The `len` function can be used to determine the number of items in a list, and a `for` loop can work through the items in a list. The items in a list can be of the same type of data or many different types.

A program can locate an item in a list by using an index value, enclosed in brackets, to identify the position of the item in the list. The item at the start of the list has the index 0, and successive items are numbered sequentially up to the limit of the list. For example, a five-item list would have items numbered 0,1,2,3,4. The index of a list item can be expressed as a fixed value or by using the value of a variable. If a program uses an index value for which there is no item (for example, trying to access an item with the index value 5 in a five-item list) the program will fail with an exception.

A list is a one-dimensional storage device. To store two-dimensional data (for example a table of numbers) you can create a list that contains other lists.

Lists (and other items) can be written into files. The `open` function can be used to create an object that represents a connection to a file for reading or writing.

The object exposes methods that can be used to interact with the file. It can also be used with a `for` construction to work through each line in the file.

When writing to a file, the program must explicitly add new line (`'\\n'`) characters to the end of each line in the file. When the lines are read back in, a program can use the `strip` function to remove new lines from lines that are read. When a program has finished interacting with a file, it must use the `close` method on the file object to complete any outstanding actions on the file and make the file accessible for other programs.

Using files may result in programs raising exceptions. These must be dealt with so that the user is aware when something fails. The exception handlers must also ensure that all open files are closed in the event of an error. The `with` construction makes it easier to create code that ensures files are closed in the event of an error.

The Python language provides a collection storage mechanism called a tuple. A tuple can hold several items, but it is immutable, which means that the elements in a given tuple cannot be changed. Tuples can be used to create lookup tables and can also be used by functions wishing to return more than one value.

Here are some points to ponder about lists.

### **Do we really need lists?**

Yes. There are many situations where it would be impossible to create a program if lists were not available. Very simple programs can use single variables, but to process substantial amounts of data you need to have a list.

### **Do we really need tuples?**

No. Tuples are very useful and make it possible for a programmer to prevent a data item from being changed when it should not be, but we can write programs without using a tuple at all.

### **How does a list actually work?**

When the program creates a list, a block of memory is reserved that is big

enough for a few list items. The block of memory also holds the current number of active items in the list (that is, those items that actually have something in them). When an item is appended to the list, one of the items is “filled in” with the item being added. If there is no room in the list for another item, it is automatically extended. When a program accesses a list item, the program first checks to see whether the requested item exists (that is, it makes sure that the index value doesn’t refer to an item that does not exist). If the item can’t be found, the program is terminated with an exception. If the item is inside the list bounds, the program finds the item in the list and returns it.

## **Why are tuples called tuples?**

Tuple is a mathematical term meaning “an ordered list of elements.” Python must have gotten the name tuple from mathematics.

## **Should the sales program use a list to store the sales figures or a tuple?**

This is a very complicated question. It really depends on what we want to do with the sales figures. One part of me reckons that the sales figures should be stored in a tuple because the values in a tuple can’t be changed. From a security point of view, this is a good thing. We don’t want programmers to be able to alter sales figures they’re not supposed to change.

However, using a tuple would make the program more complicated because it would be harder to “build up” the items in a tuple as they are read in. This is because, as we said, a tuple cannot be changed once it has been created. This would mean that the program would need to create a new tuple each time a new value is read in.

There’s also the possibility that the user of the program might want us to add an “edit” function so that she can correct sales values that were entered incorrectly. If we had used tuples to store the values, this would not be easy to do.

## **Can functions return lists, instead of tuples?**

Yes, they can. It’s best to regard the result of a function as something that cannot be changed, which means that returning a tuple from a function is a good idea. But a list could be returned instead.

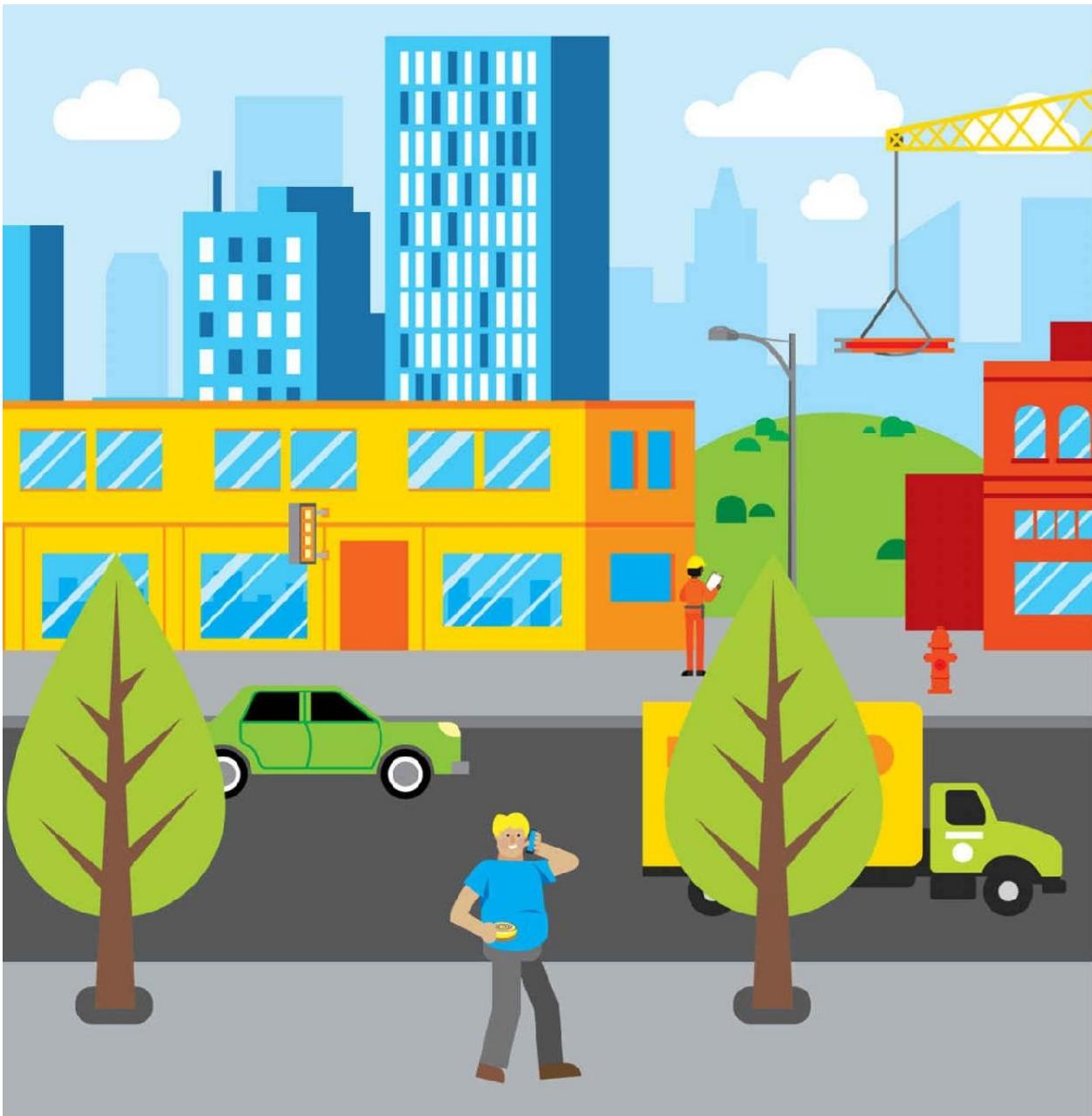
## **Will my program run faster if I use tuples to store all the data in it?**

Yes, it will. This is because tuples themselves are simpler to implement when the program runs. However, the speed improvement would be very hard to detect, so it's not worth the extra effort.

## **Does the `with` construction stop objects from throwing exceptions?**

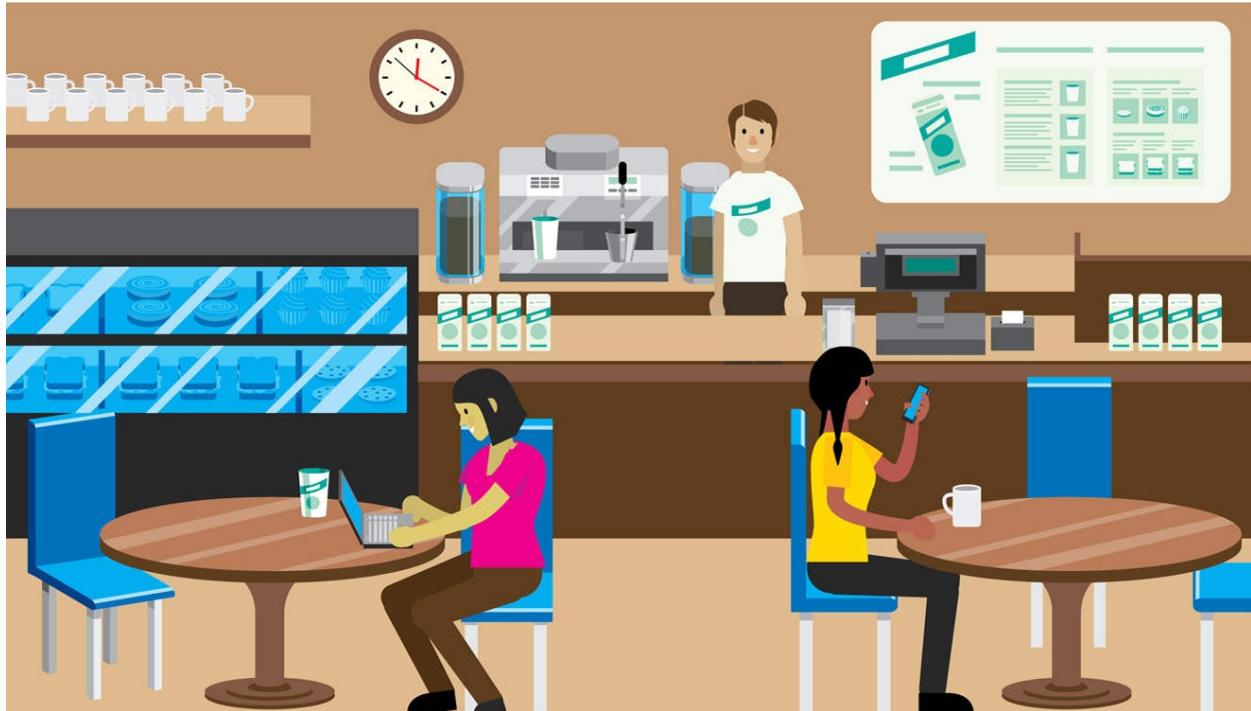
No. The principle behind the `with` construction is that it ensures that if an object throws an exception, the managed object is still closed down correctly. In other words, using the `with` construction to manage a file object will not hide exceptions that the object might produce, but it will make sure that if the file throws an exception the close behavior performs.

## **Part 2 Advanced programming**



In this second part, we'll look at advanced features of the Python language that build on the fundamental program abilities we picked up in [Part 1](#). These features are designed to make it easier to create larger programs and map the program to the problem. You'll also find out how to create libraries of reusable code and how to download and install libraries that others have created.

## 9 Use classes to store data



### What you will learn

We know that a computer program is something that takes data in, does something with it, and then produces output. Programming is the creation of instructions that tell the computer what to do with incoming data. Data can be stored in single values or in collections of data. In this chapter, we'll examine features of the Python programming language that let programmers design data storage that maps to the needs of a specific program. We'll learn how to create *classes* that can be used to treat complex collections of data as single *objects*. We'll learn how to use objects to make programs easier to write and understand. Finally, we'll look at the Python dictionary.

[Make a tiny contacts app](#)

[Dictionaries](#)

## What you have learned

# Make a tiny contacts app

Suppose one of your clients is a lawyer who wants someone to create a personal, confidential contacts app. The client wants a tiny “lightweight” application to provide a quick way of storing contact details—names, addresses, and telephone numbers—for her important clients. You start by drawing up a storyboard for the program. Below, you see the first menu the program will display.

[Click here to view code image](#)

Tiny Contacts

1. New Contact
2. Find Contact
3. Exit program

Enter your command:

The user enters a command number and presses **Enter**. If the user enters **1**, the program asks for the contact’s name, address, and phone number, and then creates a new contact for that name.

[Click here to view code image](#)

Create new contact

Enter the contact name: Rob Miles

Enter the contact address: 18 Pussycat Mews,  
London, NE1 410S

Enter the contact phone: +44(1234) 56789

Contact record stored for Rob Miles

If the user enters **2**, the program asks for a name, and then prints out the contact details for the name:

[Click here to view code image](#)

```
Find contact
Enter the contact name: Rob Miles
Name: Rob Miles
Address: 18 Pussycat Mews, London, NE1 410S
Phone: +44(1234) 56789
```

If the name is not found, the program prints out a message:

[Click here to view code image](#)

```
Find contact
Enter the contact name: Fred Bloggs
This name was not found.
```

If the user enters command 3, the program finishes.

## Make a prototype

The best way to show the lawyer what her program will look like is to create a prototype program that behaves in the same way as the finished product. For this Tiny Contacts program, we can do this by using some simple code that prints messages and accepts input. The program below uses functions from the Begin to Code input module that we created at the end of [Chapter 7](#). We need to make sure that the Python file **BTCInput.py** is in the same folder as this program when we run it.

```

EG9-01 Tiny Contacts Prototype

from BTCInput import * We're using the BTC input functions

def new_contact(): Called to create and store a new contact
 print('Create new contact')
 read_text('Enter the contact name: ') None of these values are stored when they have been read
 read_text('Enter the contact address: ')
 read_text('Enter the contact phone: ')

def find_contact(): Called to find a new contact
 print('Find contact')
 name = read_text('Enter the contact name: ')
 if name=='Rob Miles': Only recognize contact Rob Miles
 print('Name: Rob Miles')
 print('Address: 18 Pussycat Mews, London, NE1 410S')
 print('Phone: +44(1234) 56789')
 else: If the name is not Rob Miles, print an error
 print('This name was not found.')

menu='''Tiny Contacts

1. New Contact
2. Find Contact
3. Exit program

Enter your command: '''

while True: Main command loop
 command=read_int_ranged(prompt=menu,min_value=1,max_value=3)
 if command==1:
 new_contact()
 elif command==2:
 find_contact()
 elif command==3:
 break

```

You can use this program to enter and search for contact information. However, the program will only display output if you search for a contact named **Rob Miles**, and it doesn't store any entered data. However, for demonstrating how the program will work, it's very useful. Your client agrees that the program can work like this, and you can start building it.



## CODE ANALYSIS

# The contacts application prototype

There are a few questions you should consider about this code.

**Question:** Is this code familiar?

**Answer:** Yes. A lot of the behaviors have been taken straight from the ice-cream sales program that we wrote in [Chapter 8](#). The structure of the menu used to select different functions is the same. This structure is a very good template for any kind of menu-driven program.

**Question:** The values returned by the `read_text` functions are ignored by the program. Is this legal?

**Answer:** Yes. It is legal. The `read_text` function is from the BTCInput library that we created in [Chapter 7](#). The function asks a user for a string of text and then returns that text. You might find this surprising, but a Python program is not forced to use the value returned by `read_text`. In this case, I haven't decided how to store the data, so there's no point in doing anything with it.

**Question:** How does the program stop?

**Answer:** The main command loop repeatedly reads in command numbers and acts on them. When the user enters command number 3, this causes the program to break out of the loop, which stops it.

**Question:** Isn't the prototype a bit basic? Why don't you make it store the data?

**Answer:** My prototype is intentionally useless. There are two reasons for this. First, there is always the chance that the customer might take a look at the prototype and decide that they don't want the program after all. In that case, all the work I've performed on

the prototype is a waste of time. So, I try to do the minimum work possible.

Second, if the prototype works very well, there's always the danger that the customer might decide that they want the program there and then, rather than waiting for the polished, definitive version. This can be dangerous because the prototype might be poorly written and may not perform well in practice.

**Question:** How is the telephone number stored?

**Answer:** The telephone number will be stored as a string of text. This is how it should be. Although we refer to them as numbers, a telephone number will often contain other characters such as the + character to denote international numbers and parentheses and spaces to denote area codes.

## Store contact details in separate lists

First, we must write the part of the program that reads in the contact details. In the preceding program, this means filling in the contents of the `new_contact` function. The function will request the name, address, and phone number of the contact and then store that data somewhere. When we wrote the program to store ice-cream sales, we used a list to hold the sales values:

```
sales=[]
```

The ice-cream sales analysis program appended each sales value to the sales list as it was read in:

[Click here to view code image](#)

```
sales.append(read_int(prompt))
```

We could do something similar for our address book. Perhaps we could create

three separate lists, one for each of the contact items we want to store in the program:

[Click here to view code image](#)

```
Create the lists to store contact information
names=[]
addresses=[]
telephones=[]
```

Then the `new_contact` function could read and store the information in each list:

[Click here to view code image](#)

```
def new_contact():
 """
 Reads in a new contact and stores it
 """
 print('Create new contact')
 names.append(read_text('Enter the contact name: '))
 addresses.append(read_text('Enter the contact address: '))
 telephones.append(read_text('Enter the contact phone: '))
```

Then when we want to find contact information for a specific person, we must find the index of that person in the name list and then use that index to obtain the rest of their contact information.

```

EG9-02 Tiny Contacts Three Lists
def find_contact():
 ...
 Reads in a name to search for and then displays
 the content information for that name or a
 message indicating that the name was not found
 ...

 print('Find contact')
 search_name = read_text('Enter the contact name: ') Read in the name to search
 search_name = search_name.strip()
 search_name = search_name.lower() Allow for capitalization variations
 name_position=0 Count the names
 for name in names: Work through the names list
 name=name.strip()
 name = name.lower() Tidy up the name we're comparing
 if name==search_name: See if the names match
 break Break out of the loop
 name_position=name_position+1 Move the counter on so that it refers
 to the next contact position
 if name_position < len(names): If we didn't reach the end of the
 names list, we had a match
 print('Name: ',names[name_position])
 print('Address: ',addresses[name_position])
 print('Telephone: ',telephones[name_position])
 else:
 print('This name was not found.')

```



## CODE ANALYSIS

### The `find_contact` function

There are a few questions we might consider about this code.

**Question:** How does this code work?

**Answer:** If you've ever searched your clothes drawer for a matching sock, then you've used the same algorithm as the `find_contact` function. Sock searching involves working through the socks in the drawer to find the one that matches the one

you are holding. As soon as you find the sock, you can give up the search, put your socks on, and head for breakfast (or lunch or dinner, depending on what time it is).

In the case of the `find_contact` function, the user enters the search name (the first sock) and then the function uses a `for` loop to work through all the names in the name list (the other socks) looking for a match for that name.

**Question:** What is the `name_position` variable used for?

**Answer:** The `name_position` variable is used to count through the names during the `for` loop during the search for a matching name. The `for` loop will go around once for each name in the list. The function needs to know the position of the name that matches the search name so that it can use that position value to get the address and telephone details from the lists that hold them.

Consider what would happen if the name we were searching for was the second name in the list. The first time around the loop, the value of `name_position` would be 0. The name would not match, the value of `name_position` would be increased by 1, and the loop would go around again. This second time, the name would match, and so the program breaks out of the loop, leaving the value 1 in `name_position`. This value can be used to index the lists and retrieve the data for the contact.

**Question:** How does the function know if a name has not been found?

**Answer:** If you're searching for a matching sock, you know that there is no match when you've looked at all the socks in the drawer and haven't found a match. The `find_contact` function works the same way. If the `for` loop looks at all the names in the list and doesn't find a match, this means the name being searched for was not found. In this case, the variable `name_position` will have been incremented for each of the names in the list, meaning it will contain the length of the name list. The `if` construction after the `for` loop tests for this situation and displays an appropriate message.

**Question:** What do the calls of `strip` and `lower` do?

**Answer:** The user of the program wouldn't like it if the program failed to recognize a name because you typed "rob" rather than "Rob" or accidentally entered a space at the beginning of a name as it was entered. The program "sanitizes" the search name by removing any "white space" at the beginning or end of the search name and then converting it to lowercase. Each of the names in the list is given the same treatment before it is compared to the search name.

**Question:** Can we save the user from having to type in all the names when they search?

**Answer:** Yes, we can. Currently, the search process checks to see whether the search name matches the entire name in the storage. The `startswith` function provided by the string type returns `True` if a string starts with a given string. We can complete the search when it finds a name that starts with the search string.

[Click here to view code image](#)

```
EG9-03 Tiny Contacts Quick Search
if name.startswith(search_name):
 # if the names match, end the loop
 break
```

This means that a search for "Rob" will now find the entry for "Rob Miles." However, there is a problem with this approach. If the user searches for "Rob" and the name list contains both "Rob" and "Robin," the program will only display the first matching entry. If the user wants to find "Robin," they will have to type in a greater number of matching characters. However, using `startsWith` to match names will reduce the amount of typing the user must do when searching for a contact.

## Use a class to store contact details

As you've seen, we can create a perfectly workable Tiny Contacts application by using a list for each piece of information we want to store for our contact.

However, working with data stored in this way is not as easy as we might like. If the customer asks us to make the program print out a list of contacts sorted in alphabetical order by name we could do this (after all, we know about Bubble Sort) but it would be harder to write the sort function because the program would have to swap the elements in all the lists each time it found a pair of contacts that were in the wrong order.

If we ever add a new data item for a contact (perhaps we want to add their email address), we would need to add a new list and then make sure that items in it were managed correctly. Otherwise, we might find that a sorted list of contacts included incorrect email addresses.

We want a way of holding all information about one contact. Some kind of "container" could hold the name, address, phone number, and any other items we want to store. One possible solution might be to use a list to store information about each customer, but this wouldn't make it very easy to access specific detail items. Instead, we'll use a Python *class*.

You'll hear a lot about Python classes over the next few chapters, as they are one of the fundamental building blocks that underpin the language. You might have heard the term "object-oriented programming." Classes are the program constructions you use to create objects. Another way to look at this is that an object is an *instance* of a class. You can think of a class as the plans to make a tree-house and an instance of a class (or object) as a tree-house made from those plans.

In this chapter, we'll focus on how you use a class to store data—how to declare a class and then create instances of that class.



**MAKE SOMETHING HAPPEN**

## Creating a class

We can use the Python Shell to investigate how a class is created. Open the IDLE command shell and enter the statements below.

[Click here to view code image](#)

```
>>> class Contact:
 pass

>>>
```

The first statement `class Contact:` starts the definition of a Python class called `Contact`. When you press **Enter** at the end of the statement, you'll find that IDLE automatically indents the next line because it's now expecting you to enter statements that are part of the class. You saw this behavior in [Chapter 7](#) when you entered Python functions into the Command Shell.

We'll create an empty class and then fill it in later, so just give the statement `pass` as the first and only statement in the `Contact` class. Then enter an empty line to tell the Command Shell that you've completed your definition of the class.

**Question:** Why does the name `Contact` begin with a capital letter?

**Answer:** We have seen that Python variable and function names begin with a lowercase letter. However, by convention, Python programs use initial caps for class names. Our program will run regardless of whatever we call our classes, but we should try to follow language conventions when creating our programs.

**Question:** Why does the `Contact` class contain a Python `pass` statement?

**Answer:** We first saw the `pass` statement in [Chapter 8](#) when we used it to create empty “placeholder” functions. We are creating an empty `Contact` class and then adding attributes to each `Contact`

as the program runs. A class must contain at least one statement, so we put a `pass` statement in `Contact`.

Now we can create an instance of the `Contact` class. Type the following:

[Click here to view code image](#)

```
>>> x=Contact()
>>>
```

**Question:** This looks like a function call. Are we calling a function here?

**Answer:** We are not actually calling a function in this statement, but you can regard `Contact()` as a call to a function that generates an instance of the `Contact` class and then returns it. One reason why class names should start with a capital letter is that an experienced programmer can look at the above statement and know instantly that it is creating an *instance* of a class, not calling a function.

**Question:** What's an instance?

**Answer:** An instance is the *realization* of a class. A class is a bit like a design, whereas an instance is something built from that design. A program can contain instances of many different types of classes. These instances are all called objects. In other words, an object is an instance of a class.

The class design for `Contact` is presently empty. Not all classes are created empty; we'll look at some more complex classes in [Chapter 10](#).

Now that we have our instance, we can add *data attributes* to it.

[Click here to view code image](#)

```
>>> x.name='Rob Miles'
```

We have seen that when a Python statement assigns a value to a new variable name, this variable is created automatically. The same is true of data attributes. The instance `x` now contains an attribute called `name`.

**Question:** What's a data attribute?

**Answer:** A data attribute provides information about a class instance. In the English language, the word “attribute” refers to a quality or feature of something we are describing. I have many attributes. I’m tall, devastatingly good looking, and prone to telling lies about myself.

In the case of the `Contact` instance above, it now has data attributes called `name`, `address`, and `phone`, which describe this contact.

Classes can also contain *methods attributes*. These are behaviors that an instance can be asked to perform. We’ll create a method attribute later in this chapter when we create an `__init__` method for the `Contact` class.

A program can ask an object to provide the value of an attribute. Try the following:

[Click here to view code image](#)

```
>>> x.name
'Rob Miles'
>>> x.name = x.name + ' is a star'
>>> x.name
'Rob Miles is a star'
>>>
```

These statements display the value of the `name` attribute and then update it by adding a true message to the end of the name. A program

can use an attribute of a specific type (in this case a string) anywhere it can use a variable of that type.



## PROGRAMMER'S POINT

### Attributes in Python classes can be confusing

If you already know some Java, C#, or C++, you might find a Python program's ability to add attributes to an instance rather confusing.

In Java, C#, or C++, a detailed class design is required before any instances of a class can be created. In other words, in these languages, you would have to specify that the `Contact` class contains the name, address, and telephone number attributes before your program creates any `Contact` instances. In some languages, a class design is fixed at the start of a program, and it is not possible to add new attributes to an object while a program is running.

You should not regard this difference as evidence that these languages are better or worse than Python, any more than you would say that cars are better than motorcycles for getting you around. Both have their disadvantages and advantages.

## Use the `Contact` class in the Tiny Contacts program

We can use the `Contact` class to simplify the Tiny Contacts program. The program now only needs a single list to hold all the contacts.

```
contacts=[]
```

The `new_contact` function creates a `Contact` instance, sets the attributes to the contact information, and adds the new contact to the list of contacts.

```
def new_contact():
 """
 Reads in a new contact and stores it
 """

 print('Create new contact')
 # create a new instance
 new_contact=Contact()
 # add the data attributes
 new_contact.name=read_text('Enter the contact name: ')
 new_contact.address=read_text('Enter the contact address: ')
 new_contact.telephone=read_text('Enter the contact phone: ')
 # add the new contact to the contact list
 contacts.append(new_contact)
```

**Create a new contact**

**Add the data attributes to the contact**

**Append the contact to the list of contacts**

This function is very similar to the `new_contact` function that stores contacts in separate lists. The information about the new contact is read in and assigned to data attributes on a `Contact` instance. This instance is then appended to the contact list.

```

EG9-04 Tiny Contacts Class
def find_contact():
 """
 Reads in a name to search for and then displays
 the contact information for that name or a
 message indicating that the name was not found
 """

 print('Find contact')
 search_name = read_text('Enter the contact name: ')
 search_name = search_name.strip()
 search_name = search_name.lower() Convert the search name to lowercase

 # Set the result to indicate nothing found
 result=None Set result to None to indicate nothing found

 for contact in contacts: Work through the contacts
 name=contact.name Get the name from the contact for testing
 name=name.strip()
 name = name.lower() Convert the name we are
 comparing with to lowercase

 # see whether the names match
 if name.startswith(search_name): Do we have a match to the name?
 # if the names match, set the contact
 result = contact Record the contact that was found

 # end the loop
 break Stop searching

 if result!=None: Test whether the result has been
 set to a contact
 # Found a name
 print('Name: ',result.name) Display the details
 print('Address: ',result.address)
 print('Telephone: ',result.telephone)
 else:
 print('This name was not found.') Tell the user nothing was found

```



## CODE ANALYSIS

The class-based `find_contact` function

**Question:** How does this code work?

**Answer:** This code looks for contacts that match the search name.

In this respect, it is very similar to the `find_contact` function that searched through a list of names. However, rather than using an integer to hold the index of the result information in the various lists, this function uses a variable called `result` that is set to the `Contact` instance that has a matching name.

**Question:** What does the value `None` mean?

**Answer:** The `find_contact` function needs a way to represent the situation in which there is no contact with the name being sought. Python provides the symbol `None` to represent a value meaning nothing. The `result` variable is initially set to `None` (because no matching contact has been found). If the `for` loop finds a matching `Contact`, it will set `result` to this contact, replacing the `None` value. If the `find_contact` function reaches the end of the list of contacts without finding a matching name, the value of `result` will still be `None`. The function tests for this, and either displays the found contact or a message indicating that nothing was found.



## WHAT COULD GO WRONG

### Duplicate names

There is a serious bug in the system we've created. It's possible to create a new contact with the same name as an existing one. Because the "replacement" contact will be stored further down the array than the original contact, it will never be used. The program will always find the original customer first, which would result in one of our array elements being wasted. You might like to consider how you could modify the program to eliminate this problem.



## PROGRAMMER'S POINT

### Look for problems when you receive the specifications

When you talk to your lawyer client about the Tiny Contacts application (see “[Make a tiny contacts app](#)” earlier in this chapter), there is no guarantee that problems such as duplicate contact names will ever be discussed. It is your job as a programmer to consider the ways a system can go wrong and to add extra behaviors to deal with these. There are some ways to handle duplicate account names, but you need to find out how the client wants the program to behave. The worst thing you can do in a situation like this is to assume you know what the customer wants the system to do, because doing so will almost certainly mean your solution will behave incorrectly when things go wrong.

## Edit contacts

Let’s assume your customer is pleased with her Tiny Contacts data storage program and starts using it a lot. However, as is often the case with systems like these, she soon discovers a limitation that she hadn’t thought of when she agreed to the specification. She would like a way to edit the contact information. Currently, if the telephone number of a contact changes, there is no way she can update the information in the contact store. Once entered, a record cannot be changed. Here’s how you might write out the specification for the new behaviors:

[Click here to view code image](#)

### Tiny Contacts

1. New Contact
2. Find Contact
3. Edit Contact
4. Exit program

This is the new menu for the Tiny Contacts program. It has acquired an additional menu item, Edit Contact. When the user selects this item, she can then search for a contact and edit it in the following way.

[Click here to view code image](#)

```
Edit contact
Enter the contact name:Rob
Name: Robert Miles
Enter new name or . to leave unchanged: .
Enter new address or . to leave unchanged: .
Enter new telephone or . to leave unchanged: +44
(1482) 465079
```

Once the contact has been found, the user can enter a new value for each of the data items about the contact, or she can enter a period (.) to indicate that the entry is to be left unchanged. In the above example, the name and address were unchanged and a new telephone number was entered. If no contact is found with the required name, the edit function displays an appropriate message:

[Click here to view code image](#)

```
This name was not found.
```

## Refactor the Tiny Contacts program

In [Chapter 8](#), we restructured the Ice Cream analysis program and created functions that let us reuse their behaviors in the program. At the time, I said that you refactor a program when your understanding of a program improves, and you hit upon a better way of arranging the code. You might also need to refactor your solution if the program specifications change.

We now have two features in our program that search for a contact by name. The program searches for a contact so that it can be displayed, and it also searches for a contact in order to edit that contact. You might think that an effective way to create the `edit_contact` function would be to copy the `find_contact` function and just change the print behavior into an edit behavior. This would work, but it is *not a good idea*, because both functions would contain a contact search behavior. If the user asks for an improvement to the way that search works, or if we find a bug in the way the search works, we must remember to

change the search code in both functions. Otherwise, your customer will complain that the behavior of her program is inconsistent.

The refactoring we'll do involves creating a new function that we'll call `find_contact`. We'll then rename the original function `find_contact` to `display_contact` because that is a better description of what it actually does.

```
def find_contact(search_name):
 ...
 Finds the contact with the matching name
 Returns a contact instance or None if there is
 no contact with the given name
 ...
 search_name = search_name.strip()
 search_name = search_name.lower() -> Convert the name we're searching for to lowercase
 for contact in contacts: -> Work through the contacts
 name=contact.name
 name=name.strip()
 name = name.lower() -> Convert this contact name to lowercase for searching
 # see whether the names match
 if name.startswith(search_name): -> See whether we have a matching contact
 # return the contact that was found
 return contact -> Return the matching contact
 # if we get here, no contact was found
 # with the given name
 return None -> If no matching name was found, return None
```



## CODE ANALYSIS

### The refactored `find_contact` function

If you look closely at this version of the `find_contact` function, you'll find that it is very similar to the previous version. However, some things are different.

**Question:** Why does the function contain two `return` statements?

**Answer:** Although the function contains two returns, only one of them is obeyed for a given name search. Either the code in the `for` loop finds a match for the name of a contact, in which case the function returns the contact that was located, or the name supplied as a parameter is not matched by any of the contacts in the list. If the name is not matched, the `for` loop completes, and the program returns a `None` value to indicate that nothing was found.

**Question:** What would happen if another program tried to use the return value of the `find_contact` function, and the `find_contact` function had returned `None`?

**Answer:** The `find_contact` function returns `None` if no contact is found with a matching name. If a program tries to use this value, an exception is raised.

[Click here to view code image](#)

```
c=find_contact('Mysterious X')
print(c.address)
```

The code above tries to print the address of the contact called `Mysterious X`. If this contact doesn't exist, the function `find_contact` will return `None`. When the second statement tries to print out the `address` attribute of `c`, the program will fail with an exception.

[Click here to view code image](#)

```
print(c.address)
AttributeError: 'NoneType' object has no
attribute 'address'
```

It's important that users of the `find_contact` function check to

see whether it has returned a contact.

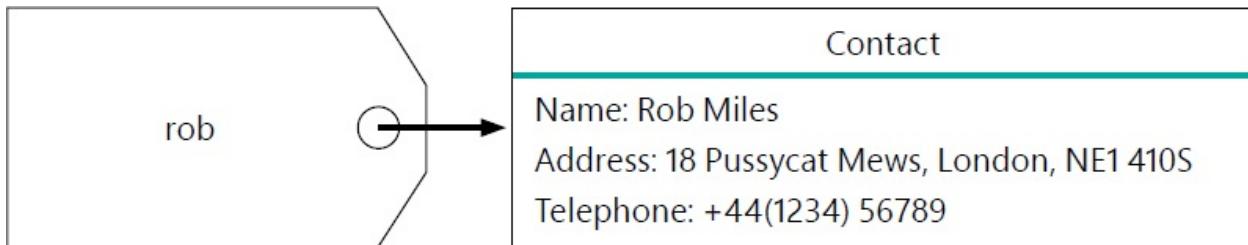
## Contact objects and references

Now that we have a `find_contact` function, we need to consider just what it returns to the caller. In other words, we need to understand just what happens in this statement:

[Click here to view code image](#)

```
rob=find_contact('Rob Miles')
```

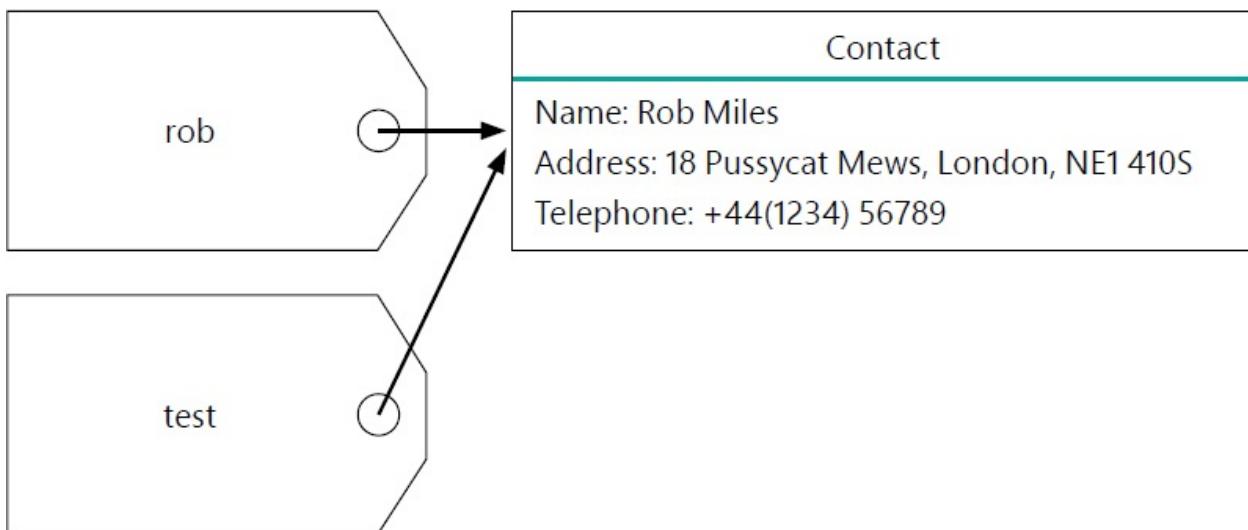
The statement calls `find_contact`, which will search for a contact with the name Rob Miles. If a contact with this name is present in the contacts list, it is returned. However, what gets returned by `find_contact` is a *reference* to the contact that contains the name Rob Miles. You can think of a reference as a tag that is tied to a particular object in memory. **Figure 9-1** shows how this might look.



**Figure 9-1** Contact class and reference

The tag and the object in memory are separate things. A tag can be connected to a different object by an assignment. We can also create multiple tags (references) that all refer to the same object, simply by assigning a new variable to the reference. **Figure 9-2** shows what happens if we create a new variable called `test` and assign it to the variable `rob`.

```
test=rob
```

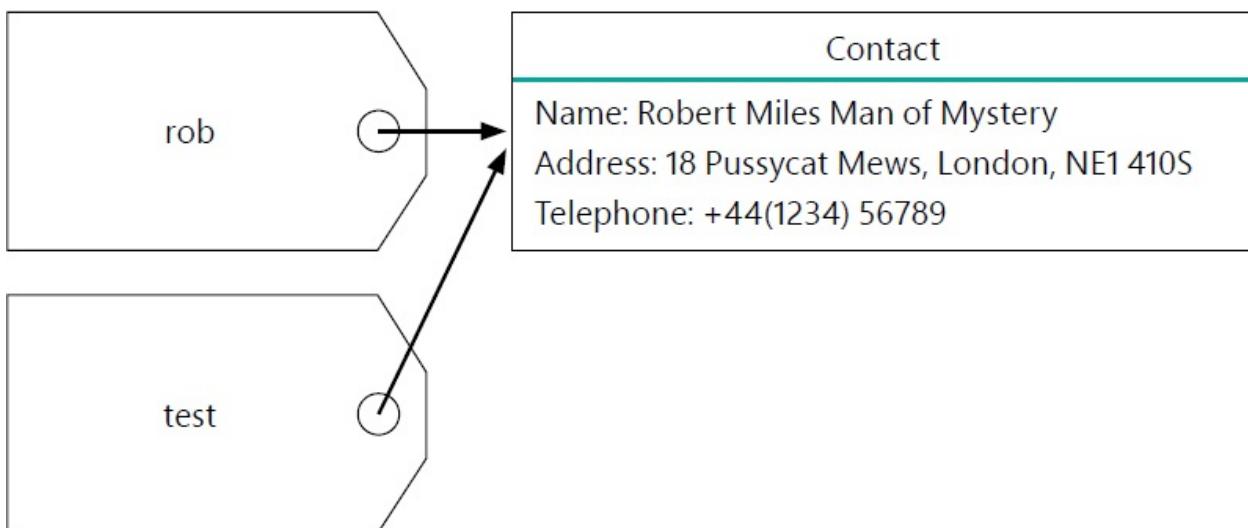


**Figure 9-2** Two references to the same contact

We now have two references that both refer to the same object in memory. This means that changes to the variable `test` will affect the variable `rob` because they are both the same object. **Figure 9-3** shows the effect of the change performed by the statement below. The name attribute of the contact held in memory has been updated to the new name.

[Click here to view code image](#)

```
test.name='Robert Miles Man of Mystery'
```



### **Figure 9-3** Changing the contents of an object

Now that we understand how references work, we can start to see how lists work. An item is never held “in” a list. Instead, the list holds a collection of references to list items.



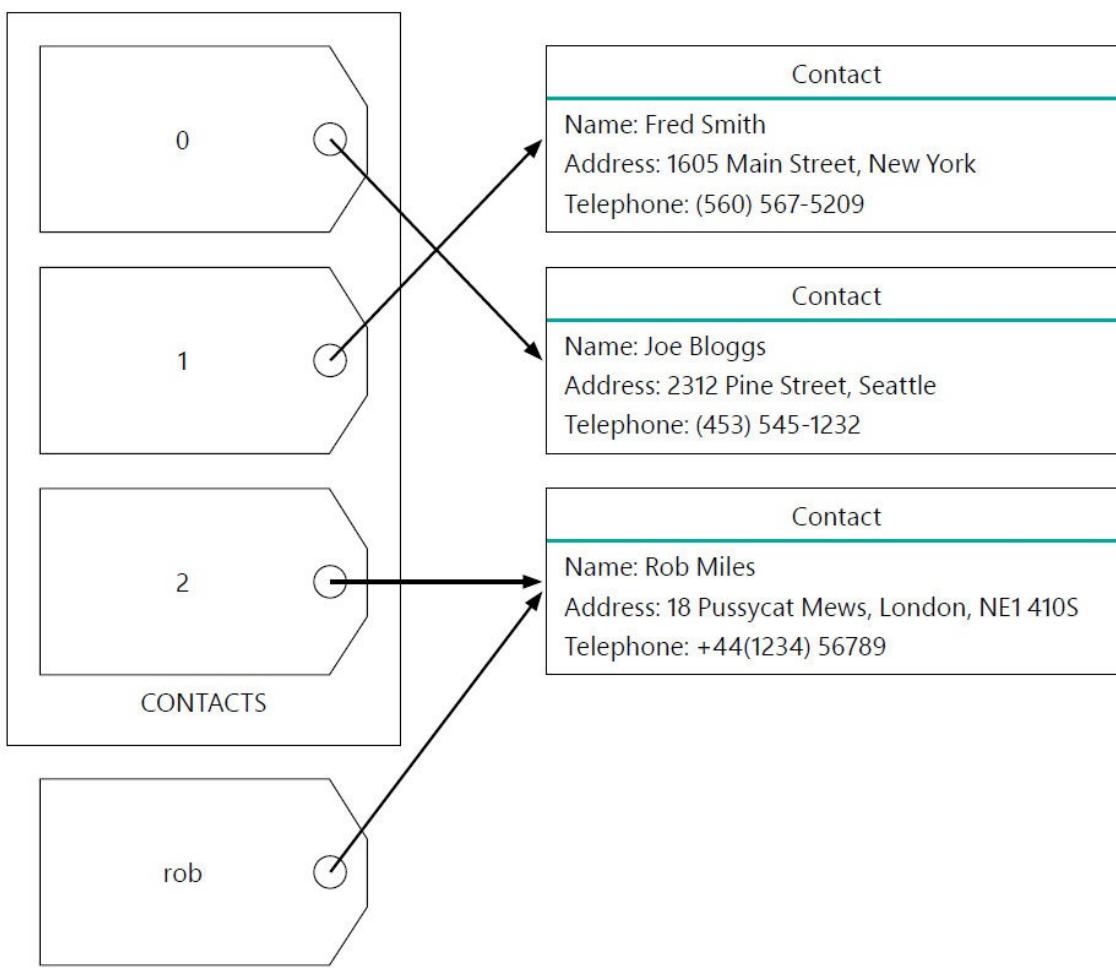
### **CODE ANALYSIS**

## Understanding lists and references

**Figure 9-4** illustrates how lists and references work. It shows a Tiny Contacts data store with three contacts registered. Each of the tags in the contacts list refers to a different Contact instance that is held somewhere in memory. We can build our understanding of lists and references by considering some questions about this arrangement.

**Question:** The diagram contains four references. How many data objects does it contain?

**Answer:** There are three data objects for “Rob Miles,” “Joe Bloggs,” and “Fred Smith.” There are four references, but two of the references refer to the same object.



**Figure 9-4** Lists and references

**Question:** What is the name of the contact at the beginning of the list?

**Answer:** The contact at the start of the list has the index value 0 (because that is the index of the element at the beginning of a list). If you look at the top left tag in the list in [Figure 9-4](#), you can follow the arrow from this tag to find that the object it refers to has the name “Joe Bloggs.” We don’t really know where this contact is held in the memory of the computer, but we do know that the tag at location 0 in the contacts list contains a reference referring to a contact object with the name attribute “Joe Bloggs.”

**Question:** What would happen if the program performed the following statement?

[Click here to view code image](#)

```
contacts[0]=contacts[1]
```

**Answer:** Remember that a number in brackets after a list name is the index number to an element in the list. So, we are making the element at the beginning of the list (the one with index 0 and the name of “Joe Bloggs”) equal to the element with index 1 (the one with the name “Fred Smith”).

Both these elements would now refer to the contact with the name “Fred Smith.” If a program worked through the contacts list using a for loop, it would find the “Fred Smith” contact twice.

However, I can’t think of a reason why you would ever do this. Not only does it make an item in the list appear twice, but it also has the effect of making it impossible to use the contact with the name “Joe Bloggs,” because there is no longer a reference to this object. The Python system will notice that there are no references to the object and it will be automatically removed. This process is called *Garbage Collection*.

References make it much easier to work with large data objects. If we decide to produce a list of contacts sorted in alphabetical order by contact name, the program will not move any objects in memory; instead, the program just has to sort the list of references. If a function wants to return an object to a caller (as the `find_contact` does) there’s no need to copy a large lump of data; instead, a reference to the object can be returned.

**Health warning:** This next part covers one of the most painful things about Python that you must understand. It might take several attempts to grasp. If you find it difficult to understand, you are not alone. If it starts to get confusing, just back off, have a cup of coffee (or a good night’s sleep), and then come back to it.



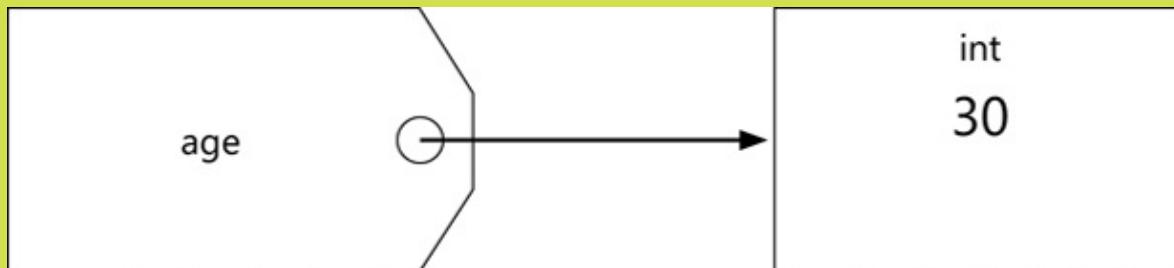
## MAKE SOMETHING HAPPEN

### Discovering “Immutable”

Everything in Python is an object. In other words, everything is an instance of a class. The value `30` is an instance of the `int` class. Open IDLE, and we'll investigate how this works.

```
>>> age=30
>>>
```

Type in the above statement and then consider what you have made. We've said that such a statement creates a *variable* called `age` and that the variable is set to the value `30`. Now, we can visualize what is happening.



If you're unsure, you can always ask Python:

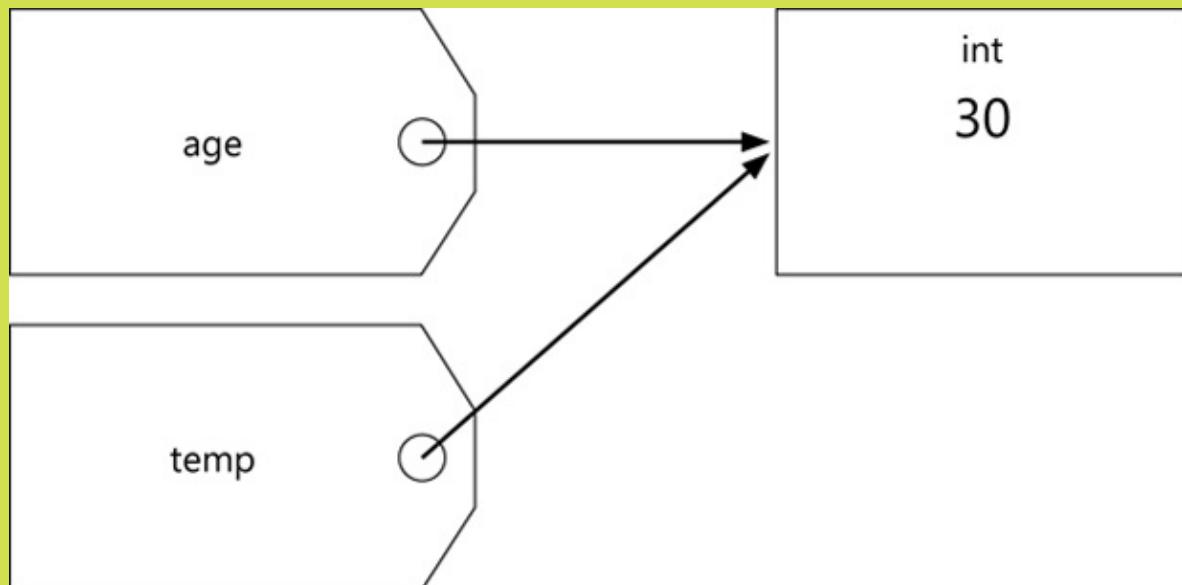
```
>>> type(age)
<class 'int'>
>>>
```

The built-in function `type` accepts a reference as an argument and then returns the type of object to which the reference refers. So, we can be sure that the variable `age` refers to an instance of the type `int`. Now,

let's do some more Python:

```
>>> temp=age
>>>
```

The above statement creates a new variable called `temp`, which is equal to `age`.



From the Tiny Contacts diagram in [Figure 9-4](#), you know that the variables `age` and `temp` now refer to the same instance of `int`. So far, so good. However, what happens when we assign a new value to the variable `temp`?

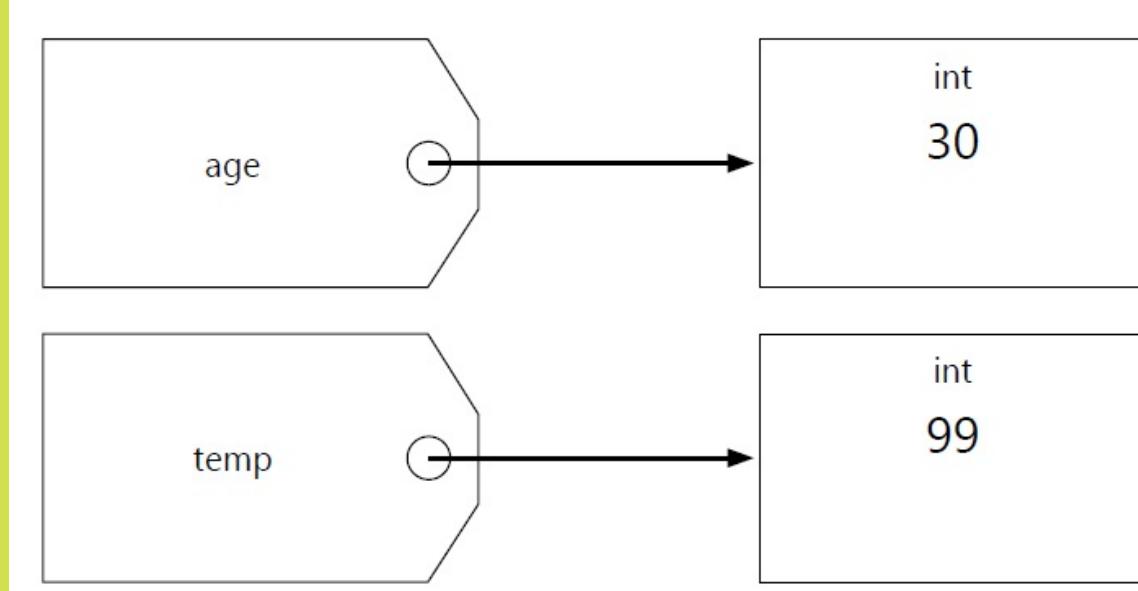
```
>>> temp=99
>>>
```

We've stored a new value in `temp`. Because `temp` and `age` both refer to the same thing, you might think that this would change the value in `age`. Let's see:

```
>>> age
```

```
30
>>>
```

The value of `age` has *not* changed. However, a new instance of `int` has been created, and `temp` has been made to refer to it:



This happens because the `int` data type is *immutable*. Rather than changing the contents of an instance of an immutable data type, Python instead creates a new instance of that type with the changed value. The Python string type is also immutable. If we perform the same actions with string values, we'll see exactly the same behavior as above:

```
>>> name='Rob'
>>> temp=name
>>> temp='Fred'
>>> name
'Rob'
>>>
```

The above sequence shows that assigning the string '`Fred`' to the variable `temp` does not affect the value of `name` because these variables are of type string, and the string type is immutable.

**Question:** Why does Python use immutable data types?

**Answer:** For some procedures—such as working with simple numbers—it's best to have variables that behave as values. Consider the following statements.

```
pi=3.1415
x=pi
x=99.99
```

We don't want this sequence of statements to change the value of pi, which is what would happen if the floating-point data type was not immutable.



## PROGRAMMER'S POINT

### Programming languages work with values differently

If you look at other programming languages, you'll find that many of them address this issue in some manner. Programmers like using references because references make it very easy to work with large data objects without having to move the objects around in memory. However, programmers also like the ability to perform simple data manipulation with types such as `int`, `bool`, `float`, and `string`.

If you're a C# programmer, you know about *value types*. If you have learned some Java, you'll have heard of *primitive types*. The Python language makes the `int`, `bool`, `float`, and `string` types immutable so that they can be manipulated as if they were simple values.

## Edit a contact

After that digression about references and immutable types, we can now consider how to edit contact information. We know that once a program has a reference to a contact object, it can read and modify any of the data attributes of the contact.

The program we created is designed to make it easy for the user to change just one element of the contact.

[Click here to view code image](#)

```
Edit contact
Enter the contact name:Rob
Name: Robert Miles
Enter new name or . to leave unchanged: .
Enter new address or . to leave unchanged: .
Enter new telephone or . to leave unchanged: 123-
456-7890
```

Our program must read a string from the user for each of the items in the contact. If the content of the string is not a single period (.), the string replaces the item in the contact with the text typed by the user (in this case 123-456-7890). In the above exchange, only the telephone number of the contact would be changed. The name and address would be left as they were.

```

EC9-05 Tiny Contacts with Editor
def edit_contact():
 """
 Reads in a name to search for and then allows
 the user to edit the details of that contact.
 If there is no contact, the function displays a
 message indicating that the name was not found
 """

 print('Edit contact')
 search_name=read_text('Enter the contact name:')
 contact=find_contact(search_name)
 if contact!=None:
 # Found a contact
 print('Name: ',contact.name)
 new_name=read_text('Enter new name or . to leave unchanged: ')
 if new_name!='.':
 contact.name=new_name
 new_address=read_text('Enter new address or . to leave unchanged: ')
 if new_address!='.':
 contact.address=new_address
 new_phone=read_text('Enter new telephone or . to leave unchanged: ')
 if new_phone!='.':
 contact.telephone=new_phone
 else:
 print('This name was not found.')

```

Read the name to search

Search for the name in the contacts

find\_contact returns None if  
the contact was not found

If the name is new, store it

If the address is new, store it

If the phone number is new, store it

If we get here, find\_contact returned None

One very important thing to take away from this discussion of editing data is that editing does not remove a contact from the list, edit it, and then “put it back.” The `find_contact` function returns a reference to the actual data object itself. Any changes to this object will occur on the “live” data. If you want to create a “cancel” feature that allows the user to abandon changes before saving them, you’ll need to make the `edit` function work on a copy of the data that can be restored to the contact if the user abandons any changes they’ve made.



## WHAT COULD GO WRONG

Missing attributes

The `edit_contact` function calls the `find_contact` function to find a contact with the given name. We've seen that `find_contact` will return `None` if it can't find a contact with a matching name, but what would happen if the object returned did not have an expected attribute? For example, `find_contact` might return an object that has a `name` attribute but not an `address` attribute. In this case, the program will fail with an exception when it runs:

[Click here to view code image](#)

```
AttributeError: 'Contact' object has no attribute 'address'
```

Some programming languages, such as Java, C#, Visual Basic, and C++, check for this kind of mistake before the program runs. Python does not. If you incorrectly type an attribute name (for example, you try to use an attribute called `adress`), then the program will start running and then fail at the statement that uses the incorrectly named attribute.

## Save contacts in a file using pickle

Currently, the Tiny Contacts program does not save the contacts to a file. This means that when the program stops, all the data is lost. In [Chapter 8](#), we saw how to read and write lines of text to a file, and we used the `load` and `save` behaviors for the ice-cream sales figures. We could use the same commands to write out names, addresses, and telephone numbers. However, Python provides a much better method of storing large data structures, called *pickling*.

If you have a large amount of vegetables, you can preserve them with pickling. Python provides commands that let you “pickle” the contents of a variable. Pickling is a clever process because it works with complicated structures such as lists. It stores the data in a binary file, and binary files contain values that make sense to the program that created them. The functions that perform pickling and unpickling are held in the `pickle` library. First, the program needs to import this library.

```
import pickle
```

Pickled data is held in a *binary* file. Because you've used a computer, you're accustomed to working with different kinds of files, such as JPEGs for images, MP3s for music, ZIP files for compressed data, and so on. Your computer's operating system can identify the type of the file by the *file extension* part of the file name, which is expressed as a set of extra letters on the end of the file name. A picture might be called "myhouse.jpg," and a music file might be called "track1.mp3." The operating system contains a list of file extensions and the associated programs that can work with them so that when you select a file with the file extension ".jpg," it will be opened by a picture viewer program.

The file extension ".txt" means *text file*. The ".py" extension also means text, but the text is a Python program. Text files contain values that map to specific characters. We saw how this mapping works in "Data and information" in [Chapter 2](#). The important thing to remember about files, as far as the computer operating system is concerned, is that all files are treated similarly.

A program tells Python to treat a file as a binary file by adding a "b" to the mode string that controls how the file is to be opened.

[Click here to view code image](#)

```
output_file = open('contacts.pickle', 'wb')
```

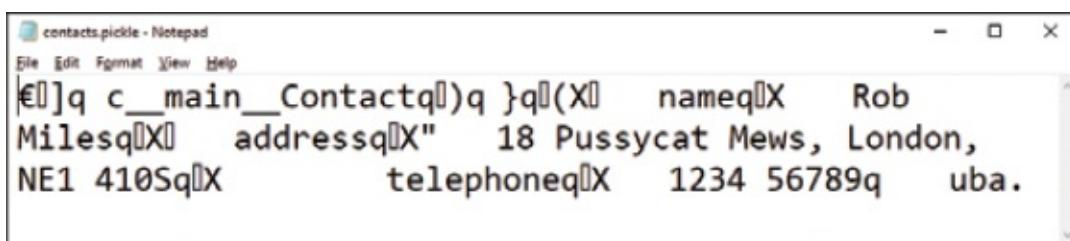
The above statement opens a file called `contacts.pickle`. The file will be opened for use as a binary file (that's what the `b` on the end of the mode string means). The file name has been given the `.pickle` extension to identify it as a file that contains pickled Python data. A program can use the `dump` function to pickle the contacts list and store it in a file:



The above statement writes the entire contacts list to the specified file

connection. All the names, addresses, and telephone numbers are stored, and this statement will work correctly whether there are 10 contacts or 100 contacts.

You might be interested to see what the contents of a pickled file look like if we open them with a text editor. **Figure 9-5** shows the contents of a pickle file that contains a single contact entry. You can see that some of the elements are recognizable text, but some are just strange characters. If we changed any of the characters in the pickled file, these changes might be detected and Python may refuse to load the file, but it's important to note that pickling data doesn't store the data securely, any more than writing data into a text file does.



```
[q c __main__Contactq)q }q(X nameqX Rob MilesqX addressqX" 18 Pussycat Mews, London, NE1 410SqX telephoneqX 1234 56789q uba.
```

**Figure 9-5** Pickled text

The `save_contacts` function saves the contacts into a file. The name of the file to use is supplied as a parameter to the function. The function uses the `with` construction we saw in [Chapter 8](#), so there's no need for the program to close the file, as that will happen automatically.

[Click here to view code image](#)

```
def save_contacts(file_name):
 """
 Saves the contacts to the given file name
 Contacts are stored in binary as a pickled file
 Exceptions will be raised if the save fails
 """
 print('save contacts')
 with open(file_name, 'wb') as out_file:
 pickle.dump(contacts, out_file)
```

This function does not deal with any exceptions that might be raised if the save

process fails, but I like this. I prefer that a program crashes trying to save something rather than leaving me with the impression that the save has worked when it has not. The code that calls `save_contacts` should deal with exceptions that the `save_contacts` function might raise.

## Load contacts from a file using pickle

The pickle library provides a function called `load`, which we'll use to read back the pickled data. The load process is the reverse of save. The binary file is opened, and then the data is reconstructed. The `load` function in the pickle library needs only to be given the connection to the file.

```
def load_contacts(file_name):
 ...
 Loads the contacts from the given file name
 Contacts are stored in binary as pickled file
 Exceptions will be raised if the load fails
 ...
 global contacts
 print('Load contacts')
 with open(file_name, 'rb') as input_file:
 contacts=pickle.load(input_file)
```

Changes will be made to the value of contacts



### CODE ANALYSIS

## Loading data using pickle

A couple questions about the `load_contacts` function are worth considering.

**Question:** What does the “global contacts” statement do? Why do we need it only in the `load` function and not the `save` function?

**Answer:** The `load_contacts` function will be used to change the value in the `contacts` variable. The `contacts` variable

refers to the list of all the contacts being held in the Tiny Contacts program. The `save_contacts` function must use the reference to find the list. However, the `load_contacts` function will be changing this value. As we saw in [Chapter 7](#), a function can always read a global variable, but if it wants to write into a global variable, it must explicitly identify that variable by declaring it as global.

**Question:** How does the pickle `load` function know what kind of data to make when loading?

**Answer:** If you look closely at the pickled text in Figure 9.4, you'll see that it contains both the data for the attributes ('`Rob Miles`') but also the names of the attributes. The file also contains the name of the class being created. When the pickled file is loaded, the `load` function looks for classes with the matching name in the program that's loading the data and uses those matches to create the new instances. It is therefore important that the `Contact` class has been defined before pickle is used to load any files that contain contact data.



## WHAT COULD GO WRONG

### Version control

Programmers call tools like pickle *serializers* because they convert a data structure into a serial stream that can be sent to another computer or stored in a file. However, there is a problem with this process that you must keep in mind: version control. If I change the design of the Contact class (perhaps to add an email attribute to each contact), this might break all the pickled files that I have stored previously because the older files would not include the email attribute. The only way to resolve this is to store version numbers along with the pickled data so that your program can migrate the data from one version to another.

## Add save and load to Tiny Contacts

Now that we have the functions to perform the `save` and `load`, we must add them to the Tiny Contacts program. We could load the contacts data when the program starts and save it when the user exits the program.

```
EC9-06 Tiny Contacts with Load and Save
file_name='contacts.pickle' This is the name of our contacts file
try:
 load_contacts(file_name) Try to load the contacts
except:
 print('Contacts file not found') Deal with exceptions raised if the load fails
 contacts=[]
while True: Command loop
 command=read_int_ranged(prompt=menu,min_value=1,max_value=4)
 if command==1:
 new_contact()
 elif command==2:
 display_contact()
 elif command==3:
 edit_contact()
 elif command==4:
 save_contacts(file_name) Save the contacts if the user gives the exit command
 break
```



### CODE ANALYSIS

#### Saving and loading contacts

A few questions about this code are worth considering.

**Question:** What happens if the `load_contacts` function raises an exception?

**Answer:** The `load_contacts` function will raise an exception if the contacts file can't be found, or if the `load` function in pickle

fails. If this happens, the program catches the exception, prints a message, and then creates an empty contacts list. This should only happen once, when the program first runs and finds no contact file.

**Question:** Why does the program not catch exceptions raised by `save_contacts`?

**Answer:** Perhaps it should. My thinking is that if the program crashes, the user will have no doubt that the contacts have not been saved. You might find it useful to modify this code to display a message rather than just fail in this way.

**Question:** Why does the program use a variable for the file name of the pickled file?

**Answer:** The contacts are held in a file called `contacts.pickle`. This file name is used in calls to `save_contacts` and `load_contacts`. Rather than inserting a string literal (`'contacts.pickle'`) into each call, I've created a variable called `file_name` that is used instead. The thinking behind this is that if I want to use a different file to hold the contacts, I only need to change the value of the `file_name` variable, rather than having to find the places in the program where the string is used.

## Set up class instances

Currently, the Tiny Contacts program adds data attributes to a `Contact` instance after the instance is created.

```
new_contact=Contact()
add the data attributes
new_contact.name=read_text('Enter the contact name: ')
new_contact.address=read_text('Enter the contact address: ')
new_contact.telephone=read_text('Enter the contact phone: ')
```

Create an "empty" Contact

Add each data attribute to the contact

This is the sequence of code in the `new_contact` function that creates a new

`Contact` instance and then adds the name, address, and telephone attributes to that instance. This code works, but there is the potential for problems if one of the attributes is not added to the class or an attribute name is misspelled. The program would contain `Contact` values that would not work correctly. It would be nice if we could create a `Contact` and initialize it at the same time.

It turns out that we can do this by adding an *initializer* method to the `Contact` class. A class can contain method attributes as well as data attributes. You can regard a method attribute as a function that is held as part of the class. Later, we'll use method attributes that will allow objects to do things for us. For now, however, we'll add an initializer method to the `Contact` class and use it to set up each instance.

## The Python initializer method

The Python initializer method is held inside a class and has the name `__init__`. This is a special name that Python uses specifically for the initializer in a class.



### MAKE SOMETHING HAPPEN

#### Create an initializer

Open IDLE and create a class that contains an initializer to find out how it works.

[Click here to view code image](#)

```
>>> class InitPrint:
 def __init__(self):
 print('you made an InitPrint
instance')

>>>
```

Type in the above class definition. It creates a class called `InitPrint` which contains an initializer that just prints a message. Be careful to put two underscores before and after the word `init`, and to add a single parameter called `self`, exactly as shown above. The last line of the class is an empty line.

**Question:** The initializer looks remarkably like a function. Why is that?

**Answer:** You can think of the initializer as a function that is called when an instance of a class is created.

[Click here to view code image](#)

```
>>> x=InitPrint()
you made an InitPrint instance
>>>
```

Type in the code above, which makes an instance of the `InitPrint` class and sets the variable `x` to refer to it. When the instance is created, the `__init__` method runs. Currently, this just prints a message.

**Question:** How is the `__init__` method made to run?

**Answer:** I don't really know. All I know is that the method runs each time I make a new instance of the `InitPrint` class. If I used a loop to create 100 instances of `InitPrint`, I would find that message printed out 100 times. My program will never call the `__init__` method directly; instead, it runs as a consequence of the creation of an object. If you create another instance of the `InitPrint` class, you'll find that the message is printed again.

So now we know that the `__init__` method runs when an instance of a class is created. Now we need to figure out how we can send information into an instance when we created it.

[Click here to view code image](#)

```
>>> class InitName:
 def __init__(self, new_name):
 self.name=new_name

>>>
```

The initializer in the class `InitName` has an additional parameter, called `new_name`. The initializer no longer prints a message. Instead it performs what looks like an assignment, using the parameter called `self` to identify the target of the assignment. Understanding what `self` means is key to understanding how methods in classes work.

You can think of `self` as “a reference to the object that is running this method.” In other words, when the initializer begins running, the value of `self` is set to refer to the instance that is being created. So, the assignment statement takes the value of `new_name` and assigns it to an attribute called `name`, which is added to the instance.

The `self` parameter is always the first parameter of a method in a class. We never set this parameter ourselves; instead, we use it in methods to get a reference to the object in which the method is running. Any other parameters work in the same way as parameters to functions. We can see this in action when we create a new instance of the `InitName` class.

[Click here to view code image](#)

```
>>> x=InitName('fred')
>>>
```

When we create a new instance of the `InitName` class, we can pass in an argument, which is the name to be assigned to this instance. This value is used to set the `name` attribute of the new `Contact` instance. After the initializer has run, we will find that the variable `x` (which is referring to an instance of the `InitName` class) now has an attribute

called `name`.

```
>>> x.name
'fred'
>>>
```

Once we have provided an initializer for a class, this becomes the only way that we can create an instance of that class. If we try to create a new `InitName` instance without a name argument, we will find that Python will generate an error:

[Click here to view code image](#)

```
>>> y=InitName()
Traceback (most recent call last):
 File "<pyshell#48>", line 1, in <module>
 y=InitName()
TypeError: __init__() missing 1 required
positional argument: 'name'
```

This is a great way of making sure that an object is only ever created with an appropriate set of attributes.

We can create an initializer for the `Contact` class that accepts three parameters along with `self`.

```
class Contact:
 def __init__(self, name, address, telephone):
 self.name=name
 self.address=address
 self.telephone=telephone
```

Initialize with four parameters  
Create attributes for the parameters



## CODE ANALYSIS

### Parameters and the `__init__` method

If you've read the above code sample properly, you should have at least one or two questions about it.

**Question:** It looks like you've written the assignments in the initializer so that a value is assigned to itself. What's going on?

**Answer:** Consider statements like this:

```
self.telephone=telephone
```

It seems that I'm assigning the value of `telephone` to itself. But this is not the case. The item on the right side is the parameter (which I've called `telephone`), and the item on the left is an attribute of the object referred to by `self` (which is called `self.telephone`). These are two different variables.

To understand why these are two different variables, you must know that Python uses *namespaces* when finding variables. A namespace is “a space in which names have a unique meaning”. One namespace is the namespace of parameters and variables local to the `__init__` method. Another namespace is that of the attributes of an instance. Two different namespaces can contain variables with the same name, just like two books can each have a page called “Contents.” If we specify the namespace (like saying “The contents page in ‘Begin to Code with Python’”), Python can work out the location of the variable. The name `self.telephone` refers to a variable called `telephone` in the object referred to by `self`. The name `telephone`, in the `__init__` method refers to the parameter `telephone`.

When I created the class `InitName`, I broke this rule so that you could see how the value of the parameter `new_name` was transferred into the `name` attribute of the instance being initialized. However, it makes sense to give the parameters the same name as the attributes in a class, and you'll see why in the next section.

**Question:** What happens if the user of the constructor supplies silly arguments to it?

**Answer:** Currently, the initializer doesn't perform any validation of the parameters it receives. In other words, the value of the `name` parameter could be an empty string, or a number, or even the value `None` and the `Contact` would still be initialized. If you wanted, you could add data validation to the initializer so that it checks the validity of the parameters being used to create the object and raises an exception if it doesn't like them.

You might do this if you're writing a super-secure banking application. However, for this program, I think it's reasonable to assume that users of the class will behave themselves. In any development, you must balance the benefits of the error checking against the cost of writing extra code. You can tell the bank that you'd be happy to write a version with super-secure classes, but you must make sure you get paid for adding the extra security.

Now our program can create a new contact and set all the values of that contact at the same time:

[Click here to view code image](#)

```
rob=Contact(name='Rob Miles', address='18 Pussycat
Mews, London, NE1 410S',
telephone='+44(1234) 56789')
```

The above statement creates a new `Contact` and sets the reference `rob` to refer to it. I've used keyword arguments to explicitly identify the different values

being set up in the class. Each keyword maps directly onto the attribute in the class, which makes it easy to understand what is being set up.

[Click here to view code image](#)

```
EG9-07 Tiny Contacts with initializer
def new_contact():
 """
 Reads in a new contact and stores it
 """
 print('Create new contact')
 # add the data attributes
 name=read_text('Enter the contact name: ')
 address=read_text('Enter the contact address:
')
 telephone=read_text('Enter the contact phone:
')
 # create a new instance

 new_contact=Contact(name=name, address=address, telepho
 # add the new contact to the contact list
 contacts.append(new_contact)
```

This version of `new_contact` reads in the name, address, and telephone values from the user and then uses them to create a new `Contact` value that is appended to the list of contacts in the Tiny Contacts application.

## Use default arguments in an initializer

We saw default arguments in [Chapter 7](#) when we created a text input function that used a default prompt (“Please enter some text”) if the caller didn’t specify a prompt when the function was called. We can do something similar with the initializer.

[Click here to view code image](#)

```
class Contact:
 def __init__(self, name, address, telephone='No
telephone'):
 self.name=name
 self.address=address
 self.telephone=telephone
```

It is now possible to create a `Contact` instance without giving a telephone number.

[Click here to view code image](#)

```
rob=Contact(name='Rob Miles', address='18 Pussycat
Mews, London, NE1 410S')
```

The telephone number for the address would be set to '`No telephone`'.

## Dictionaries

In the previous chapter, we discovered how to use lists and tuples to create variables that can store collections of values. Python has another collection mechanism called a *dictionary*. A program can store a collection of data in a dictionary and then easily locate a particular item in the dictionary by using a *key*. The name dictionary is very appropriate, as this is exactly the tool we use when we look up the meaning of a word we haven't heard before. The word is the key, and the dictionary description is the item for which we are searching.

From a Python programming perspective, you can think of a dictionary as a list that is indexed by a *key* rather than a letter or number. To access an element in a dictionary, we could say "Give me the element with the key of 'rob.'" To access an element in a Python list, we could say "Give me the element with the index of 2."



## MAKE SOMETHING HAPPEN

### Creating a dictionary

We can use the Python Shell to investigate how a dictionary is created. Open the IDLE command shell. We'll create a dictionary that a coffee shop could use to keep track of prices. The key will be the name of the drink, and the item we are seeking is the price of that drink.

```
>>> prices={}
>>>
```

Enter the statement above. It creates a dictionary with the name **prices**. Remember to use braces, { and }, to tell Python that a dictionary is being created.

Currently, the dictionary is empty. We can add an item to the dictionary by giving the key and the value to be stored.

```
>>> prices['latte']=3.5
>>>
```

This statement adds an element to the **prices** dictionary. The element has the key '**latte**' and the value **3.5**. The program can use the key '**latte**' to find the value:

```
>>> prices['latte']
3.5
>>>
```

The use of square brackets in the above statements might make you

think of a list. When we use a list, we provide an index value to identify the element we want. In a dictionary, we use the key to find the element we want. We can change the value in a dictionary at any time:

```
>>> prices['latte']=3.6
>>>
```

This statement slightly increases the price of a latte.

The key must be given exactly. Try misspelling the key to see what happens:

[Click here to view code image](#)

```
>>> prices['Latte']
Traceback (most recent call last):
 File "<pyshell#3>", line 1, in <module>
 prices['Latte']
KeyError: 'Latte'
```

As you might expect, an exception is raised. We can check to see whether a dictionary contains a key:

```
>>> 'latte' in prices
True
>>>
```

We can add extra items to the dictionary at any time, and we can print the entire dictionary:

[Click here to view code image](#)

```
>>> prices['espresso']=3.0
>>> prices['tea']=2.5
```

```
>>> prices
{'latte': 3.6, 'espresso': 3.0, 'tea': 2.5}
>>>
```

The first two statements add two more drinks to the dictionary. The third statement views the dictionary. We have seen how to add items to a dictionary, but we can also create one with a single statement.

[Click here to view code image](#)

```
>>> prices={'latte': 3.6, 'espresso': 3.0,
'tea': 2.5, 'americano':2.5}
>>>
```

This statement creates a prices dictionary for four types of drinks.

## Dictionary management

The form of a dictionary element is “**key:item**”. The value on the left is the key that will be used to locate the item. In this case, the key is a string and the item is a number, but we can use other types, too.

[Click here to view code image](#)

```
access_control={1234:'complete', 1111:'limited',
4342:'limited'}
```

This statement creates a dictionary to control access to a burglar alarm system. The user will enter their access code and the program will use the code as a key to the **access\_control** dictionary. The item that matches the key will determine whether the user has complete or limited access. The access code 1234 gives the user complete access. The access codes 1111 and 4342 give the user limited access. If the key is not found in the **access\_control** dictionary, the user is not allowed any access to the system.

```
EG9-09 Alarm access control
access_code=read_int('Enter your access code: ')
if access_code in access_control:
 print('You have', access_control[access_code], 'access')
else:
 print('You are not allowed access')
```

Read in the access code  
Check whether the dictionary contains this code value  
Display level of access  
No access allowed

If we need to delete a dictionary entry (perhaps we want to remove an alarm code value because we want to change that code to a different one) the Python `del` statement can be used to delete an entry from the dictionary:

```
del(access[1111])
```

This will delete the access code entry with the key 1111. The `del` statement can also be used to delete an entry from a list. If the entry being deleted is not found, the `del` statement will raise an exception.

## Return a dictionary from a function

A program can use a dictionary as a simple lookup table, but a dictionary is useful in a great many other contexts, too. Rather than returning a tuple as a result, the “Pirate Treasure” function from [Chapter 8](#) could instead return a dictionary:

[Click here to view code image](#)

```
EG9-08 Pirate Treasure Dictionary
def get_treasure_location():
 """
 Get the location of the treasure
 returns a dictionary:
 ['start'] is a string naming the landmark to
 start
 ['n'] is the number of paces north
 ['e'] is the number of paces east
 ...
 # get the location from the pirate
```

```

 return {'start':'The old oak
tree','n':20,'e':30}

location=get_treasure_location()
print ('Start at',location['start'],
'walk',location['n'],'paces north',
'and', location['e'],'paces east')

```

The `get_treasure_location` function above returns a dictionary that contains three elements, one for each item being returned. This result is easier to understand than a tuple because each of the elements in the dictionary has a name (the value of the key string) rather than an index value.

## Use a dictionary to store contacts

We could use a dictionary to store the contacts in the Tiny Contacts program. The key used to locate a `Contact` held in the dictionary would be the name of that `Contact`.

```

contact_dictionary = {} Create an empty dictionary
rob = Contact(name='Rob Miles',address='18 Pussycat Mews',
telephone='+44(1234) 56789') Create a new contact
contact_dictionary[rob.name] = rob Add the contact to the dictionary
using the name as the key

```

The program above shows how to create a dictionary and add a contact to it. A program can then find a contact by using their name as the key in the dictionary. The statement below finds the contact that has just been stored.

[Click here to view code image](#)

```
c = contact_dictionary['Rob Miles']
```

This would work, but the user would have to type in the full name of a contact for the name to be used as a key. On the other hand, the `find_contact` function we saw above uses the `startswith` method to match the search string with a name in the list. Our user likes the way she can find a name that

starts with a string, rather than requiring a complete match. She can enter `ro` as the search term and the program will return the contact details for `Rob Miles`. That would not be possible with a dictionary.

However, if we have an application in which the key can be a unique value—for example, a bank account number—we can use a dictionary to quickly locate bank account records.



## MAKE SOMETHING HAPPEN

### Create a data storage app

The Tiny Contacts program is a useful template for any kind of program that stores data and lets a user work with it. You can even add some of the sorting and data-processing features from the ice-cream sales program to make applications that not only store data but let you do interesting things with it.

You could create a music track storage program that lets you search for tracks based on the length of the track. The program could suggest tracks that could be combined to fill an exact amount of time or give the total play time of a specific playlist. You'd have to create a class that could hold the track information, store the information in a list, and then create some behaviors that would search through and process the data.

Or, you could make a recipe storage program that stores lists of ingredients and preparation details. Remember that one of the items in a class could be a list of strings, which could be the steps performed to prepare the recipe.

## What you have learned

In this chapter, you've created a genuinely useful application. It can store contact information, and it could be modified to store and manage any kind of data. You created your first Python class to hold contact information, and you added attributes to the class to hold specific information. You discovered that Python variables are references to objects in memory, and that some objects are immutable. The contents of an immutable object cannot be changed; instead, a new version of the object is created with the changed contents. This immutable behavior applies to simple data objects such as `int`, `float`, and `string` so that they can be manipulated as values by programs that process data.

You used the Python pickle library to save an entire collection of contacts, and you simplified the initialization of a contact by adding an `init` method to set initial values into the contact. Finally, you explored the Python dictionary mechanism that allows objects to be located by using a unique key value that identifies them.

Here are some points to ponder about classes.

**If an object has name, address, and telephone attributes, can a program treat it as a Contact instance?**

Yes. This is a very good question. It goes to the heart of how objects work in Python. In the Tiny Contacts program, we created a class called `Contact` and then added `name`, `address`, and `telephone` attributes to instances of that class. The Tiny Contacts program then displayed these attributes and allowed the user to edit them. If another program made an object (perhaps called `Customer`) which has the same attributes, then the Tiny Contacts program would work with the `Customer` object too. Some languages—for example Java, C++ and C#—have what is called “strong typing.” In these languages, each variable is given a specific type and can work only with values of that type. In these programs, an attempt to treat a `Customer` as a `Contact` would cause the program to be rejected as incorrect before it ever got to run.

Programmers refer to the way Python works as “duck typing” because Python takes the approach that “If it walks like a duck, and quacks like a duck, it is a duck.” In other words, anything that behaves as a `Contact` can be used as a contact. If a programmer gets this wrong—for example, by creating a `Customer` instance that doesn’t have a `telephone` attribute—the Python

program will raise an exception when it tries to use that attribute. Later in this book, we'll see how a Python program can check the type of objects it is using as it runs.

### **Can an object contain a reference to itself?**

Yes, although this might not be a good idea. A better idea would be to "daisy chain" objects together to form something called a "linked list," in which an object in the list contains a reference to the next item in the list. You can also use references in objects to build more complex "tree-like" data structures.

### **Can we find out exactly where in memory an object is stored?**

Everything in a program is stored somewhere in the memory of the computer. Each memory location in the computer has a unique numeric address. You can think of memory as a very large list of byte values, with an index number going from 0 to several billions. When Python creates an object, it places it at some location in memory. It's possible to discover what this location is, but this information is not particularly useful to our programs. For now, it's best to just regard objects as being "out there" on the end of references.

### **Is an object forced to have an initializer method?**

No. The very first Contact we made didn't have an initializer method. Instead, we added the attribute values after the `Contact` object had been created. The `__init__` method made it easier to set the initial values, and an initializer also ensures that the attributes are set when an instance is created, but you don't always have to make one.

### **Can you stop a program from adding new attributes to an object?**

No. We could add a new attribute to an instance of `Contact` at any time. Doing so would make that object a bit of a mutant, in that it would have an attribute that other `Contact` instances did not. However, there's nothing to stop us from doing this.

### **Can you remove attributes from an object?**

No. Once attributes have been added, they are present for the lifetime of the object.

## **What is immutable again?**

Immutable means unchangeable. Think of an immutable object as being held in a sealed box with a glass window. We can look inside the box to see what's there, but we can't change what's in the box. Whenever a program tries to change an immutable object, the Python system creates a new box with new contents, and uses that instead. Making some types immutable allows Python to simplify data storage.

Consider a program that contained a story as a list of words. Each word in the story is held in a string (which is immutable). Each entry in the list is a different word in the story. We know that lists are implemented as references, so each element of the list would refer to a string object. The word "the" is quite popular, so there might be many references to a single string instance holding the word "the." From Python's point of view, many strings containing "the" saves memory because the word "the" need be stored only once.

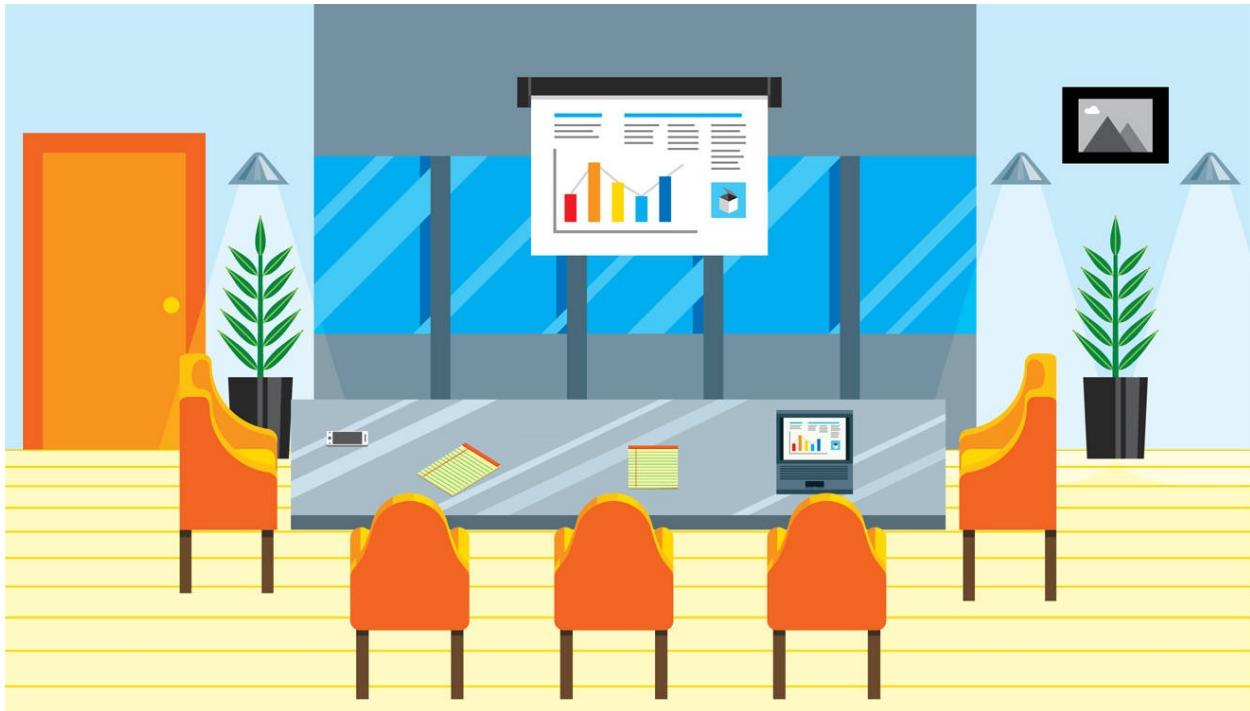
## **How does an operating system know it's storing a binary file?**

It doesn't. A computer's file system doesn't really know or care what's in the files it's managing in the same way that a librarian doesn't have to care what's inside the book that she fetches for you. It is the program using the file that imposes meaning on the contents. Files have "file extensions" on their names so that the operating system can choose an appropriate program to work with a file, but the operating system is completely unaware whether a given file is binary or text.

## **Can two items in a dictionary have the same key?**

No. If you think about it, adding a second item with the same key would make it impossible for the original to be located.

## 10 Use classes to create active objects



### What you will learn

From what we've learned up to now, we can regard a Python class as a way of designing objects that hold a set of data. We've seen that a program can add data attributes to a class instance so that it can assemble related items. We explored this by creating a class-based application that stores contact details. However, we know that we can use a class in any situation in which we want to gather some related information.

In this chapter, we'll look at the other kind of class attribute: the method. We'll discover how to create active objects that can manage the data they hold and provide behaviors for use in programs. We'll look at techniques we can use to ensure that objects have integrity and resilience, how an object can raise exceptions if invalid actions are attempted, and how a program deals with these exceptions. We'll also find out how to protect data inside objects to prevent

accidental damage to their contents, and how to use object properties to make it easy to access protected data.

We'll also look at how Python manages the process of iterating through large amounts of data, and we'll sample some powerful functions that make use of iteration. There's a lot of ground to cover, so, let's get started.

[Create a Time Tracker](#)

[Create class properties](#)

[Evolve class design](#)

[The `\_\_str\_\_` method in a class](#)

[Session tracking in Time Tracker](#)

[Make music with Snaps](#)

[What you have learned](#)

## Create a Time Tracker

Programs have a habit of growing larger. Sometimes this occurs because you underestimate the scope of the problem, which is bad. However, it can also happen because your customer likes your first program and comes back to you with additional requests (which is good). In this chapter, the news is good. We heard back from our friend the lawyer, who's been using the tiny contact book we created in [Chapter 9](#). She now wants you to add capability to track the amount of time she spends working for a client so that she has this information handy for billing. You've worked out the following user interface:



Command 4 is used to add the length of a work session that was performed for a contact. The user can find the required contact and add the length of the session

to the details for that contact: [Click here to view code image](#)

```
Enter your command: 4
add hours
Enter the contact name: Rob
Name: Rob Miles
Previous hours worked : 0
Session length : 3
Updated hours worked : 3.0
```

When command 2 is selected, the information displayed now includes the time spent: [Click here to view code image](#)

```
Find contact
Enter the contact name: Rob Miles
Name: Rob Miles
Address: 18 Pussycat Mews, London, NE1 410S
Telephone: 1234 56789
Hours worked : 3.0
```

The number of hours spent working for each contact is displayed along with their other details.

## Add a data attribute to a class

The Time Tracker application will need a way of storing the time the lawyer has worked for each client. Each contact in the Tiny Contact book is represented by a **Contact** object, which contains attributes that hold the name, address, and telephone number of that contact. To create a Time Tracker application that stores the number of hours worked for a contact, we can add an additional attribute to the **Contact** class, which will be the number of hours the lawyer has been working for that contact. When we create a new **Contact**, we need to set this value to **0**. The best place to do this is in the **\_\_init\_\_** method, which is called to set up an instance of the class when it is created:

```
class Contact:
 def __init__(self, name, address, telephone):
 self.name = name
 self.address = address
 self.telephone = telephone
 self.hours_worked = 0
```

init called to initialize a Contact

Set the initial value of hours to 0

The `__init__` method now creates an `hours_worked` attribute and sets it to `0` when a new `Contact` is created. Remember that the first parameter received by a class method call is a reference called `self`. This is a reference to the object on which the method is being called. In the case of the `__init__` method, this reference refers to the newly created `Contact`. The `__init__` method uses the `self` reference to find this newly created object and set up each of the data attributes on this object.

Now that the `hours_worked` attribute has been set up, we can create an `add_session` function to our application that will find a `Contact` object and then increase the hours worked for that contact. The lawyer will use this when she completes a session working for a client so that she can enter the time spent and make sure she gets paid. The `add_session_to_contact` function will have a very similar behavior to the `edit_contact` function, but rather than allowing the user to edit the details of a contact, the `add_session_to_contact` function will instead increase the value of the `hours_worked` attribute by an amount specified by the user.

```

EG10-01 Time Tracker

def add_session_to_contact():
 """
 Reads in a name to search for and then allows
 the user to add a session spent working for
 that contact
 """
 print('add session')
 search_name = read_text('Enter the contact name: ')
 contact = find_contact(search_name) Find the contact
 if contact != None:
 # Found a contact We found the contact
 print('Name:', contact.name)
 print('Previous hours worked :', contact.hours_worked)
 session_length = read_float_ranged(prompt='Session length : ',
 min_value=0.5, max_value=3.5)
 contact.hours_worked = contact.hours_worked+session_length Add the hours
 worked
 print('Updated hours worked:', contact.hours_worked)
 else:
 print('This name was not found.') Contact not found

```

When we add this function to our application, it finds a contact, prints out the current hours worked, and then requests a `session_length` value from the user. This value is added to the `hours_worked` attribute for that contact. The updated `hours_worked` value is then printed, and the function ends. If you run the example program **EG10-01 Time Tracker** in the code samples for this chapter, you'll find that you can use it to create and store `Contact` objects and keep track of the hours worked for each contact.

## Create a cohesive object

There's nothing wrong with our implementation of the Time Tracker application. After all, it works, and it does what the customer wants. However, from a software design point of view, there's room for improvement. Good software design is important. We want to make our software design as clear and simple as possible, making it easy for other programmers to work with it.

If we were constructing a house, we would use bricks to make the walls and cables to send electric power around the building. The job of a brick is to hold up the roof. The job of a cable is to send power from one place to another.

Builders can work with bricks and cables and know that the behavior of one will not affect the other. In other words, whether the lights in my house will work is not affected by the color of the bricks used to build it.

We can use object-oriented design to create software objects that are as individual and self-contained as bricks and cables. We want a programmer to be able to use a **Contact** object in their application in the same way that a builder would use a brick to build a house. When considering software quality, software developers talk about the amount of *cohesion* shown in the design of an object. A lot of cohesion in an object an object is a good thing. In the case of the Tiny Contacts application, a cohesive **Contact** object should provide all the data and method attributes needed to work with contact information.

Currently, the design of our **Contact** object does not show very high cohesion. The Time Tracker program works by acting directly on the data attributes held by the **Contact** object. The lawyer has told us that the shortest time she can spend on a case is half an hour, and the longest time she can spend is three and a half hours. In the Time Tracker application, this restriction is enforced in the **add\_session\_to\_contact** function, which restricts the range of the session length value read in from the user.

[Click here to view code image](#)

```
session_length = read_float_ranged(prompt='Session
length : ', min_value=0.5,
max_value=3.5)
contact.hours_worked =
contact.hours_worked+session_length
```

These two statements update the **hours\_worked** value for a contact. The first statement reads the session length (a number in the range 0.5 to 3.5), and the second statement adds this session length value to the **hours\_worked** attribute of the contact. If the lawyer tells us that she has changed her working practices and can now work for up to four hours for a contact, we must find the statements that read the session length and update the maximum value allowed.

[Click here to view code image](#)

```
session_length = read_float_ranged(prompt='Session
length : ', min_value=0.5,
max_value=4.0)
contact.hours_worked =
 contact.hours_worked+session_length
```

This would work, but in [Chapter 13](#) we'll create another version of the Time Tracker application that uses a graphical interface. This program will perform its own validation of session length. If the maximum length of a session is changed by our lawyer client, we must make sure that the session length tests in the graphical interface version of the program are updated, too. Otherwise, our client would become annoyed that she cannot enter four-hour sessions when using a graphical interface.

We have this problem because some of our “business rules”—things our customer has asked the system to do—are outside of the “business objects”—the things we have created to implement the system.

We can address this problem by making the `Contact` object more cohesive and implementing the business rules inside the `Contact` class. If we put the `Contact` object in charge of validating the length of a session, we simply must change the `Contact` object to reflect the new business rules; systems that use the `Contact` object will just keep working.

## Create method attributes for a class

Currently, any Python code has direct access to the `hours_worked` data attribute in the `Contact` object. However, programs don't need to be able to access the `hours` attribute at all. In fact, they only need to do two things with the `hours_worked` value in a `Contact`:

- Get the `hours_worked` value (to display time spent with a contact).
- Add the length of a session to the `hours_worked` value (to record a session).

Python objects can contain method attributes that can be used to ask an object to

do something. We first saw method attributes in [Chapter 5](#) when we discovered that a Python string object provides a method called `upper()`, which can be used to ask a string to generate a version of itself with the lowercase characters converted to uppercase. We used the `upper()` method to make sure that a name recognition program would recognize a name entered in any case.

We'll create two method attributes for the `Contact` class. These will manage the hours worked for a contact and remove any need for programs to access the `hours_worked` data attribute. We can start with a method to get the hours worked for a contact.

[Click here to view code image](#)

```
class Contact:
 def __init__(self, name, address, telephone):
 self.name = name
 self.address = address
 self.telephone = telephone
 self.hours_worked = 0

 def get_hours_worked(self):
 """
 Gets the hours worked for this contact
 """
 return self.hours_worked
```



## CODE ANALYSIS

The `get_hours_worked` method There are a few questions we might like to consider about this code.

**Question:** What is the parameter `self` used to accomplish?

**Answer:** A method is a function that is part of an object. The first thing a method needs to know is which object it is part of. The `self` reference is provided as the first parameter of the method and refers to the object within which the method is running. We first saw `self` when we considered the initializer (`__init__`) method in [Chapter 9](#). In the case of the initializer method, the value of the `self` parameter is a reference to the object being initialized. The code in the initializer follows this reference to add the name, address, and telephone attributes to the object being initialized.

In the case of the `get_hours_worked` method, the value of `self` is a reference to the `Contact` that the method is running “inside.” Consider the following Python code: [Click here to view code image](#)

```
rob_work = rob.get_hours_worked()
jim_work = jim.get_hours_worked()
if rob_work > jim_work:
 print('More work for rob')
else:
 print('More work for jim')
```

The code compares the number of hours worked for contacts `rob` and `jim`. The first time that `get_hours_worked` is called, the value of `self` in the method call is a reference to the object referred to by `rob`. The second time that `get_hours_worked` is called, the value of `self` in the method call is a reference to the object referred to by `jim`.

**Question:** Is the `get_hours_worked` method stored when we save contact information in a file?

**Answer:** No. If we use pickle (see [Chapter 9](#) for details) to store a contact list, the method attributes in the class are not stored. Pickle stores only the data attributes of an object.

This means we must make sure that the `Contact` class has been defined in our program *before* we load any `Contact` values using pickle. This allows method attributes on `Contact` instances to be used in the program.

**Question:** Can a program still access the `hours_worked` attribute of a `Contact` class?

**Answer:** Yes, it can. Using method attributes to get the value of a data attribute held in a class doesn't stop a program from accessing the data attribute directly. Our aim is to remove the need for programs to access data attributes. Later in the chapter, we'll discover ways that we can flag the `hours_worked` attribute as "private" to the `Contact` class.

To add the hours worked in a session, we can create a method attribute that takes a session length and adds it to the number of hours worked for that `Contact` object.

[Click here to view code image](#)

```
EG10-02 Time Tracker with method attributes

class Contact:

 def add_session(self, session_length):
 """
 Adds the value of the parameter
 onto the hours worked for this contact
 """
 self.hours_worked = self.hours_worked +
session_length
```

The `add_session` method in the `Contact` class has two parameters. The first is `self`, which refers to the `Contact` object being updated, and the second is `session_length`, which is the length of the session to be added.

This is a very simple version of the `add_session` method, which just adds the value of the parameter to the `hours_worked` attribute.

## Add validation to methods

The `add_session` method we just created is a good start, but we need to add the validation of the session length. Consider the following statement: [Click here to view code image](#)

```
rob.add_session(-10)
```

This is a completely legal call of `add_session` on the `Contact` variable `rob`, which reduces the number of hours worked by 10. This happens because the `add_session` method uses the value of its parameter and adds that value to the `hours_worked` attribute. We can make the argument a negative number, and the `add_session` method will happily reduce the number of hours assigned to that contact. This is good news for the contact, but bad news for our lawyer who has just lost some money. If you run the example **EG10-02 Time Tracker with method attributes**, you'll find that you can add negative session lengths.

To fix this, we can add validation to the `add_session` method so that it rejects an attempt to add an invalid session length. When we began writing the program, our customer told us that the smallest billable amount of time she spends on a case is half an hour (0.5) and the longest time is three and a half hours (3.5). Any attempt to add a session with a length outside this range should fail.

I regard the values 0.5 and 3.5 as “magic numbers,” in that these values that have a special meaning. However, it’s not obvious from reading the program text what that meaning is. It would be really useful if we could give these values names so that anyone reading the program can understand the intent of the code. It turns out that there is a way we can do this in Python by using *class data variables*.

## Create class variables

A class variable is a data attribute that is not part of any specific instance of the class. Instead, the variable is part of the class itself. We can use class variables to

store the maximum and minimum session length values.

[Click here to view code image](#)

```
class Contact:

 min_session_length = 0.5
 max_session_length = 3.5
```

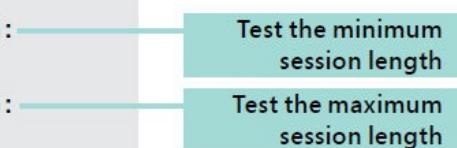
The two variables `min_session_length` and `max_session_length` are declared as part of the `Contact` class. They are not part of any `Contact` object; they are part of the `Contact` class. A program can use the name of the class to access these variables:

```
EG10-03 Time Tracker with class variables

class Contact:

 min_session_length = 0.5
 max_session_length = 3.5

 def add_session(self, session_length):
 """
 Adds the value of the parameter
 onto the hours worked for this contact
 Invalid session length values are
 ignored.
 """
 if session_length < Contact.min_session_length:
 return
 if session_length > Contact.max_session_length:
 return
 self.hours_worked = self.hours_worked + session_length
```



This version of `add_session` will ignore invalid `session_length` values by returning if it is given a session value outside the valid range. If you experiment with the sample program **EG10-03 Time Tracker with class variables**, you'll find that invalid session lengths are not added to the `hours_worked` value for an object.

A class variable is created the first time Python encounters the class. A program

does not need to create an instance of the `Contact` class to be able to use the values of `max_session_length` and `min_session_length`; the variables are attached to the `Contact` class, not to an object that is an instance of the class.



## CODE ANALYSIS

**Using class variables** We can build our understanding of class variables by considering some situations in which we might like to use them.

**Question:** Should I use a class variable to hold the age of a contact?

**Answer:** No. Each contact will have an age, so the age must be a data attribute added to a `Contact` object, probably by code running in the `__init__` method.

**Question:** Should I use a class variable to hold the maximum age of a contact?

**Answer:** Yes. There is no need to store this value for each contact; it can be held as part of the class.

**Question:** Should I use class variables to hold the price per hour that the lawyer will charge for her services?

**Answer:** If the lawyer charges exactly the same amount for all her clients, then it would be reasonable to store the price as a class variable because the value will be stored only once for all contacts. However, if the lawyer wants to be able to charge different amounts for different customers, the price must be stored as an attribute of each `Contact` object.

However, you could store the maximum and minimum prices that could be charged as class variables.

## Create a static method to validate values

When we considered cohesion earlier in this chapter, we decided that it is best if an object doesn't expose attributes for use by other programs. Ideally, users of a `Contact` object should just interact with the object via calls to methods inside the object. We created the `get_hours_worked` and `add_session` methods so that users of the `Contact` class would not have to interact with the `hours_worked` data attribute.

We could extend this policy to validation of the session length value because we don't want people to interact with the `max_session_length` and `min_session_length` values in the `Contact` class. We could create a method called `validate_session_length` that accepts a session length value and returns `True` if the session length is valid and `False` if it is invalid.

The best place for validation behavior is as part of the `Contact` class, rather than as part of any given `Contact` object, which means any program could validate a session length without needing to have an actual Contact. Python lets us do this by creating a static method. You can think of a static method as one that is part of a class. If the class is there, the method is always there. We can create a static method that could be used to validate a session length:

```

EG10-04 Time Tracker with static method

class Contact:

 min_session_length = 0.5
 max_session_length = 3.5

 @staticmethod
 def validate_session_length(session_length):
 """
 Validates a session length and returns
 True if the length is valid or False if invalid
 """

 if session_length < Contact.min_session_length:
 return False
 if session_length > Contact.max_session_length:
 return False
 return True

 def add_session(self, session_length):
 """
 Adds the value of the parameter
 onto the hours worked for this contact
 """

 if not Contact.validate_session_length(session_length):
 return
 self.hours_worked = self.hours_worked + session_length

```

Decorator indicates this is a static method  
Static method is part of the `Contact` class

Call the validation method

We tell Python that the `validate_session_length` function is a static method by preceding the declaration of `validate_session_length` with a *decorator*. A decorator wraps extra code around a function and modifies the way the function works. You add a decorator to a Python program by using the `@` character followed by the name of the decorator you want to use. The `@staticmethod` decorator is built into the Python language. It was created to convert a method into a static method that can exist without the need for an instance of the class of which it is a part. A static method can be called directly from the `Contact` class: [Click here to view code image](#)

```
print(Contact.validate_session_length(5))
```

This statement would print `False` because 5 is not a valid session length.



## CODE ANALYSIS

Creating static validation methods Input validation allows us to make very good use of static methods. Here are some questions you might consider about input validation.

**Question:** Why does the `valid_session_length` method not have a `self` parameter?

**Answer:** This is a very good question. The `self` reference is used in a method to refer to the particular object running that method. In the case of a static method, there is no object. The method is running as part of the class, not as an instance of the class. There can be no `self` reference because there is no object to which it can refer.

**Question:** Why does the `valid_session_length` method not print a message to the user communicating that a session length is invalid?

**Answer:** I've said repeatedly that it is important that the user of a program is kept informed when things go wrong. In this case, you might think it would be sensible for the `valid_session_length` method to print a message if it decides a session length is invalid so the user would always know when she had entered an incorrect value.

However, this is not a good idea. To understand why, you must consider how the `Contact` object will be used in the future. Currently, we are creating a Time Tracker that's being used from the Python console. The user types in commands, and the Time Tracker application prints messages in response to these commands.

In Chapter 13, we'll discover how to create an application that uses a graphical user interface. I plan to use this Contact class in a graphical version of Time Tracker. If methods in the `Contact` class printed messages, these could not be displayed by a graphical version of the Time Tracker program because there will be no Python console open to display them.

Software developers talk about “a separation of concerns” between objects in a program. They would say that the `Contact` class should contain all the code that manages a contact. However, it is not the job of a contact to interact with the user.

In Chapter 1, we compared the Python Command Shell in IDLE with a waiter in a restaurant. You type your commands into the shell, which then passes them on to the Python engine. The Python engine produces a result that it passes back to the Python Shell for display.

We compared the Python engine to a chef in a restaurant. The chefs in a restaurant never deal directly with the customers. They are simply given instructions to prepare dishes. Where the dishes go and how they are used is not their concern. It is the waiter who provides the “user interface” for the restaurant, which allows the chef to focus entirely on the cooking; the waiter can focus entirely on the customer experience.

You can regard the `Contact` class as rather like a chef. A program that provides the user interface will call methods on a contact to ask it to do things (for example, add a work session). Each method will return a result that can be displayed to the user, but how the result is displayed is not the responsibility of the `Contact` class. In the Python Command Shell version of the Time Tracker application, an invalid session length will result in a printed message in the Command Shell. In the graphical version of the Time Tracker, an invalid session length will be displayed in a window on the screen.

**Question:** What does a decorator do?

**Answer:** In real life, a decorator is someone who takes something

and adds things to it. For example, a decorated version of a picture might have a nice wooden frame around it. You can think of a Python decorator as a function that sets up an environment for another function to work in, runs the function, and then tidies up afterward.

**Question:** Can I create my own decorators?

**Answer:** Yes, you can, but creating decorators is a bit beyond the scope of this text.

**Question:** How do I know when to create a static method in a class?

**Answer:** You use a static method if you want to create a behavior that is independent of any instance of a class. The `validate_session_length` method is not attached to any `Contact` object because it “speaks for” the entire class.

## Return status messages from a validation method

The `add_session` method above will prevent invalid session length values from being added to a `Contact`, but it doesn’t indicate whether the session information was stored correctly. If our lawyer client mistypes an hour value, it’s possible she might not notice that the value was invalid, and this might cause session records to be lost. We need to add some way that the `add_session` method can indicate whether the `Contact` was updated correctly.

One programming technique is for a method to return a value that indicates whether it worked. Up until now, the `add_session` method hasn’t returned anything. Now we’ll make it return a Boolean value that indicates whether it worked.

[Click here to view code image](#)

```
def add_session(self, session_length):
 ...
 Adds the value of the parameter
```

```

 onto the hours spent with this contact
 Returns True if it works,
 or False if the session length is invalid
 '''
if not
Contact.validate_session_length(session_length):
 return False
 self.hours_worked = self.hours_worked +
session_length
 return True

```

When this method is called, a program can check to see whether it worked by inspecting the value of the result of the method. The `add_session_to_contact` function in the Time Tracker application finds a contact and adds the length of the new session to it. The following code shows how this function can test the result of the `add_session` method and display an appropriate message.

```

EB10-05 Time Tracker with status reporting

session_length=read_float(prompt='Session Length: ')
if contact.add_session(session_length): Add the session to the contact
 print('Updated hours succeeded:', contact.get_hours_worked()) Display success message
else:
 print('Add hours failed') Display failure message

```

This code tests the result returned by a call to `add_session` for the contact. If `add_session` returns `True`, then all is well. Otherwise, the code tells the user that the update has failed.

This works well, but it has one major problem, which is that the caller of `add_session` does not need to take any notice of the result the method returns. This makes it possible to write a version of the Time Tracker in which an attempt to add hours might fail. However, the user will never know that it has failed.

Raise an exception to indicate an error

If we want to force fellow programmers to deal with a failure in the `add_session` method, we could make the `add_session` method raise an exception rather than returning the result `False`. This will stop the program unless the exception is handled. Programs raise exceptions when something has gone wrong, and it would be meaningless for the program to perform any more statements. We've seen this behavior when we converted strings to numbers. The `int` function raises an exception when it is given a string that doesn't contain a number: [Click here to view code image](#)

```
>>> x=int('rob')
Traceback (most recent call last):
 File "<pyshell#5>", line 1, in <module>
 x=int('rob')
ValueError: invalid literal for int() with base 10:
'rob'
```

The `int` method converts a string of digits into a number. However, if you give text to the `int` method (as above), it cannot perform a conversion. Instead, the `int` method raises a `ValueError` exception to indicate that it is unhappy. If this happens in a program, the program is stopped. We'll make an `add_session` method that raises an exception when it is given an invalid input.

```
EG10-05 Time Tracker with exception

def add_session(self, session_length):
 ...
 Adds the value of the parameter
 onto the hours spent with this contact
 Raises an exception if the session length is invalid
 ...
 if not Contact.validate_session_length(session_length):
 raise Exception('Invalid session length') Raise an exception if invalid
 # only reach this statement if no exceptions were raised
 self.hours_worked = self.hours_worked+session_length Add the hours
```

The `add_session` method above raises an exception if the value of the `session_length` parameter is invalid. You can think of an exception as a

message sent to explain why the program couldn't continue. This message object is created and then "raised" to the attention of the Python system.

We'll raise an exception object. The `Exception` class is designed to deliver messages about exceptions. The initializer for the `Exception` class accepts a string that we can use to describe what went wrong.

[Click here to view code image](#)

```
if not
Contact.validate_session_length(session_length):
 raise Exception('Invalid session length')
```

This code in `add_session` deals with an invalid session length. Once the exception has been raised, the current sequence of program execution is interrupted and the program either stops with an error or control passes to an `except` handler if the statement is running inside a `try` construction. In the case of the `add_session` method above, the method will only reach the statement that updates the `hours_worked` value if no exceptions were thrown.



## MAKE SOMETHING HAPPEN

Raising exceptions from code We can investigate how exceptions are raised by using one of the example programs.

Start the IDLE editor and open the demo program **EG10-06 Time Tracker with exception** and run it.

Select option 1 and enter a new contact: [Click here to view code image](#)

Time Tracker

1. New Contact
2. Find Contact
3. Edit Contact
4. Add Session
5. Exit Program

Enter your command: 1

Create new contact

Enter the contact name: Rob Miles

Enter the contact address: 18 Pussycat Mews,  
London, NE1 410S

Enter the contact phone: 1234 56789

Now add a new session lasting 2 hours to the contact using option 4:

[Click here to view code image](#)

Enter your command: 4

add session

Enter the contact name: Rob Miles

Name: Rob Miles

Previous hours worked: 0

Session length: 2

Updated hours worked: 2.0

This works correctly because 2 is a valid session length. Now try adding a session length of 4, which is too large: [Click here to view code image](#)

Enter your command: 4

add session

Enter the contact name: Rob Miles

Name: Rob Miles

Previous hours worked: 2.0

Session length: 4

Traceback (most recent call last):

```
File "C:/Users/Rob/EG10-06 Time Tracker
with exception.py", line 197, in <module>
 add_session_to_contact()
 File "C:/Users/Rob/EG10-06 Time Tracker
with exception.py", line 145, in
 add_session_to_contact
 if contact.add_session(session_length):
 File "C:/Users/Rob/EG10-06 Time Tracker
with exception.py", line 45, in add_session
 raise Exception('Invalid session length')
Exception: Invalid session length
```

The `add_session` method raises an exception that stops our program.

## Extract an exception error message

Now we need to discover how to deal with exceptions and extract error messages from them.

```
1. # EG10-07 Time Tracker with exception handler
2. hours_worked = read_float(prompt='Enter hours spent : ') Read in the hours
3. try:
4. contact.add_session(hours_worked) This might raise an exception
5. print('Updated hours succeeded:', contact.get_hours_worked())
6. except Exception as e:
7. print('Add failed:', e)
```

This statement is reached only if `add_session` didn't throw an exception

Display failure message

This code shows how to deal with an exception and extract the `Exception` object that was raised when the error occurred. It's part of the `add_hours_to_contact` function in the Time Tracker application. We've written code that deals with exceptions before, but this version obtains the `Exception` object and then displays the message from it. The important statement here is the one on line 6, which defines the code that will run if an `Exception` object is thrown. This statement sets the reference `e` to refer to the

**Exception** object that was raised. The statement that follows (on line 7) is the code that handles the exception. It prints an error message and then the value of the exception, **e**, which causes the error text to be displayed.

[Click here to view code image](#)

```
Enter the contact name: Rob Miles
Name: Rob Miles
Previous hours worked: 2.0
Session length : -1
Add failed: Invalid session length
```

Above, you can see the output from this code if we try to add an invalid session length. In this case, the session length is too small, which is reflected by the printed message. A program can raise different types of the exception object, and we can create custom exception types if needed.



## MAKE SOMETHING HAPPEN

### Catching exceptions

We can repeat our previous experiment with the sample program **EG10-07 Time Tracker with exception handler**. You'll find that the program now runs correctly, and no errors are produced if you enter invalid session lengths.



## CODE ANALYSIS

Raising and dealing with exceptions There are a few questions we might consider about how programs deal with exceptions.

**Question:** Why does this version of the program not check the result returned by `add_session`?

**Answer:** In the previous version of our Time Tracker, the method `add_session` returned `False` if it found that the length of the session to be added was too long or too small. This version of `add_session` doesn't do that. Instead, it raises an exception if the session length is invalid. We don't need to test the result returned by this method as our program will stop if an invalid session length is given.

**Question:** Isn't raising an exception and stopping the program when something goes wrong a bit harsh?

**Answer:** I don't think so. My primary concern in situations like these is that I want to avoid "silent" errors. I'd hate for my program to leave the user with the impression that something had worked when it hadn't. Raising an exception reduces this possibility. If a programmer wants to avoid the possibility of a call to `add_session` raising an exception, they can always use the `validate_session_length` method to check a session length before adding it to a contact. In other words, I've provided a means by which session lengths can be validated, so there should be no need for `add_session` to ever throw an exception because it should only be called when we know it will work.

**Question:** Can a method be resumed once it has raised an exception?

**Answer:** No. Raising an exception is a one-way trip out of running code. If the `add_hours_to_contact` method raises an exception, the only way to repeat the behavior in the method is to call the method again, hopefully with a more sensible value to add.

**Question:** Why would we want to create our own kinds of exceptions?

**Answer:** Whenever we write some code that could fail, we should think about creating our exception type to describe what went wrong. For example, if our program is trying to read a file, it might be useful to record the name of the file and the position that's been reached in the file.

These kinds of design decisions should be made at the start of development to create an error management and reporting strategy. If you get a job as a programmer, you'll spend at least as much time writing programs to deal with fault conditions as you will spend writing the code to do the job.

**Question:** Should I always use exceptions to indicate that something has gone wrong?

**Answer:** I like exceptions because they ensure that errors are dealt with, but they don't force the errors to be handled in a particular way. In the case of the `add_session` method we've been discussing, the program that calls the method could print a message, display a dialog box, or write a line in a log file when an exception is raised.

**Question:** Why have we made `add_session` work like this? Our program was perfectly fine before because it ensured that the hours value entered was in the valid range.

**Answer:** This is a very good point. We seem to have worked very hard to solve a problem that we didn't have in the first place. However, I think we have vastly improved the program. In previous versions of the `Contact` class, some of the knowledge about how a contact works (in this case, the valid ranges for the hours we can add) was held outside the class. In other words, users of the `Contact` class had to know that they are not supposed to add hours values less than 0.5 or greater than 3.5.

I very much like putting all the knowledge about good contact behavior inside the `Contact` class. That way, if we decide to change the allowed range of hours, we know exactly where to look. Rather than having to change every piece of code that interacts with

the contact, we need only change one behavior inside the `Contact` class.

## Protect a data attribute against damage

We have now completely removed the need for a programmer to interact directly with the `hours_worked` attribute of the `Contact` class. However, the attribute is still easily accessible. A programmer could accidentally (or maliciously) alter the value of this attribute, and change the number of hours worked for a contact (which might cost our lawyer some money). So, let's look at how we can provide some protection for this important information.



### PROGRAMMER'S POINT

Python protects against mistakes, not attacks. The features we'll explore are very useful and can help protect data attributes against accidental damage. However, they don't provide any protection against malicious code. In other words, if another programmer decides to add some code to the Time Tracker application that changes the `hours_worked` information in a `Contact` object, there's nothing in the Python language I can use to stop this. The only way I can detect and prevent such attacks is by inspecting the running Python code and making sure that it runs as intended.

One of Python's conventions dictates that an attribute with a name beginning with the underscore character should not be used by code running outside the class. That is, only methods in a class should use attributes that have names

beginning with an underscore. By adding an underscore to the beginning of the attribute name, we can mark it as internal to the **Contact** class.

[Click here to view code image](#)

```
def get_hours_worked(self):
 """
 Gets the hours spent with this contact
 """
 return self._hours_worked
```

Above, you see the `get_hours_worked` method returning the value of the `_hours_worked` attribute. The snag with this approach is that it doesn't provide any protection for the variable `_hours_worked`. If a malicious programmer decides to fiddle with the value of `_hours_worked` outside the **Contact** class, Python will not stop him.

You can achieve a higher level of security by preceding the name of the attribute with two underscore characters to create a variable called `__hours_worked`. Doing so tells Python to do some "name mangling" on the attribute name, making it slightly harder to access from outside the class. We can see how this works by performing some experiments.



## MAKE SOMETHING HAPPEN

Protecting data attributes in a class We can find out how these naming conventions help to make programs more secure by using the Python Command Shell in IDLE. Open it and enter the statements below.

[Click here to view code image](#)

```
>>> class Secret:
 def __init__(self):
 self._secret=99
 self.__top_secret=100

>>>
```

These statements create a class called `Secret` that has an `__init__` method that creates two data attributes. One attribute is called `_secret` and is set to `99`. The other data attribute is called `__top_secret` and is set to `100`.

We use the same technique to create the name, address, and telephone number attributes of a new contact.

Now, let's create an instance of the `Secret` class and try to access these attributes. Enter the following statement:

```
>>> x=Secret()
```

This creates a new `Secret` instance and sets the variable `x` to refer to that instance. Now, we can try to access the `_secret` data attribute. Type the following and press **Enter**.

```
>>> x._secret
```

This tries to access the `_secret` data attribute. It works because adding a leading underscore to an attribute name does not protect it.

```
>>> x._secret
99
```

It seems that the attribute called `_secret` is not protected at all. How about the `__top_secret` attribute? Type the following and press **Enter**:

```
>>> x.__top_secret
```

This time, we are not successful; it seems that the `__top_secret` data attribute has been hidden from us.

[Click here to view code image](#)

```
>>> x.__top_secret
Traceback (most recent call last):
 File "<pyshell#43>", line 1, in <module>
 x.__top_secret
AttributeError: 'Secret' object has no attribute
'__top_secret'
```

However, Python has performed some “name mangling” to the name `__top_secret`. Inside the `Secret` class, the attribute `__top_secret` can be referred to as `__top_secret`. However, outside the class, the variable name is appended to the name of the class of which it is a part, meaning that to the outside world, the attribute is called `_Secret__top_secret`. To prove this, we can try to access an attribute with this name. Type in the following and press **Enter**: [Click here to view code image](#)

```
>>> x._Secret__top_secret
```

This time, we get access to the attribute of the class: [Click here to view code image](#)

```
>>> x._Secret__top_secret
100
```

The “name mangling” provides very good protection against accidental use of attributes inside a class, but it doesn’t completely prevent a determined person from changing data values that should be private.

The good news is that there are programs available that can check Python source

files for this kind of naughty behavior. A good example is a program called Pylint ([www.pylint.org](http://www.pylint.org)), which is a free download. Pylint will also make sure your code conforms to Python's layout conventions.

The example program **EG10-08 Time Tracker with protected attributes** contains a version of the Time Tracker that contains a protected version of the `hours_worked` attribute.

## Protected methods

So far, all the methods we've added to the `Contact` class have been intended for use by code outside the class. Methods such as `add_session` are called to store session details. Our Time Tracker application calls these public methods to perform the options the user selects from the application menus.

It is also possible to use this mechanism to protect methods held inside a class. By putting a double underscore in front of the method name, we mark it as being private to the class and not for use by code running outside the class. Note that the `__init__` method is already flagged to indicate that it is not for use outside of the `Contact` class.



### PROGRAMMER'S POINT

Writing secure code is all about workflow Making a secure program is not about doing a single thing; it's about creating a workflow that generates quality code. You can think of the program-writing process as a bit like a data processor. Problems go into the processor, and working solutions come out of the other end. We've already seen the best way to handle the inputs to the process; we use things like prototypes to make sure that the customer agrees that we are building the right thing.

Now we're considering how to make high-quality code by using sensible design and tools, such as Pylint, to make sure we're correctly building the program. Solving problems for a customer is about building a process that will generate a quality output, not just write a program. The things we're learning in this chapter play a big part in making quality, professional applications.

## Create class properties

We've spent a lot of time and effort protecting the `hours_spent` data attributes of the `Contact` class, but we haven't done anything to protect any of the other contact data. Currently, we can enter anything for a contact's name, address, and telephone number, including a single letter.

We should invent some more business rules to make sure our contact objects have sensible contents. A simple rule would be to insist that name, address, and telephone number items must be at least four characters long. We would, of course, discuss these requirements with our customer to make sure that she agrees that they are a good idea. We can then create a static method that will validate text entered into the contact and add it to the `Contact` class:

```
class Contact:

 __min_text_length = 4 Class data variable giving the minimum name length

 @staticmethod
 def valid_text(text): Decorator that makes the following method static
 validate_text takes a single text parameter
 """
 Validates text to be stored in the contact
 storage.
 True if the text is valid, False if not
 """
 if len(text) < Contact.__min_text_length:
 return False Test the length of the text against
 the minimum length
 else:
 return True Return False if the text is too small
 Return True if the text is too large
```

This method would be used in the same way as the `valid_session_length` method. It would be called to validate any text to be stored in the program. We could manage the name, address, and phone number attributes in the same way as the `hours_spent` data attribute. Also, we could provide methods to get and set these data attributes, just as we have the `get_hours_worked` and `add_session` methods for managing

`hours_spent`. We could create methods called `set_name` and `get_name` to manage the name of each contact. The `set_name` method could use the `valid_text` method above to ensure that a contact can only have names that are at least four characters long.

However, Python has a better way of providing simple read and write access to protected data held inside a class. It's called a property. Properties let us preserve the simple access to data attributes held in an object while allowing us to validate the actions performed on a data attribute.



## CODE ANALYSIS

### Properties in classes

```
class Contact:

 @property
 def name(self):
 return self.__name

 @name.setter
 def name(self, name):
 if not Contact.validate_text(name):
 raise Exception('Invalid name')
 self.__name = name
```

The code above shows how we would implement a property for the name value in the `Contact` class. The property performs validation and will reject an attempt to set the name of a string to fewer than four characters.

**Question:** How does the value being set in the property get into the `setter`?

**Answer:** The `setter` method has two parameters when it is called. The first is `self`, a reference to the object on which the `setter` is running. The second is the value to be set in the property. In the setter above, the value being set is the `name` attribute of the `Contact`.

**Question:** How does the program know which `setter` method to call for a particular property?

**Answer:** The decorator for the `setter` name contains the name of the property being set.

**Question:** Must the `setter` method raise an exception if the value being set is not valid?

**Answer:** No. There is no need to raise an exception. The `setter` could ignore invalid values or set the value to a default. However, I've decided that it's important that the user of the object be informed when a set operation fails, so I've made this version of the `setter` raise an exception if the entered text is invalid.

**Question:** Do we need to perform the same validation for all the properties in a class?

**Answer:** No. The validation for the telephone number could test that the value being set does not contain text, and the validation for the address could check for a properly formed address. I've just used the same validation method to keep the code simple.

**Question:** Must a property have a `setter`?

**Answer:** No. If we leave out the `setter` method, we have created a “read only” property. We could use this to remove the need for the `get_hours_worked` method. We could just create a property called `hours_worked` that returns the value.



MAKE SOMETHING HAPPEN

Investigating properties We can find out how properties work by using the Python Command

Shell in IDLE. Open it and enter the statements below. End the class definition with an empty line.

[Click here to view code image](#)

```
>>> class Prop:
 @property
 def x(self):
 print('property x get')
 return self.__x
 @x.setter
 def x(self,x):
 print('property x set:', x)
 self.__x = x

>>>
```

This creates a new class called `Prop` that contains a property called `x`. Now, enter a statement to create an instance of the class.

```
>>> test = Prop()
```

We can now put a value into the `x` property in the `test` class. Enter the following to set the `x` property to `99`.

```
>>> test.x=99
```

When Python performs this statement, it runs the `setter` method for the property. This method prints a message to tell us it has been called:

```
>>> test.x=99
property x set: 99
```

The `setter` method prints the value being set in `x`; in this case, the value `99`.

We can now try to read the property. Enter the following statement, which should print the value of the property.

```
>>> print(test.x)
```

When Python reads the property, it runs the property method to get the value. This method prints a message on the console:

```
>>> print(test.x)
property x get
```

We can see the getting and setting in action when we work with properties:

```
>>> test.x = test.x + 1
property x get
property x set: 100
```

In the above statement, which adds `1` to the value of the `x` property, you can see that Python first fetches the property, adds `1` to it, and then stores the result.

Note that I added the `print` statements to the `x` property so we could see how the properties are called. We would not normally put `print` statements in property code.

We can use properties for the name, address, and telephone number of a contact. Each property will have a pair of methods to get and set the value of that property:

```

EG10-09 Time Tracker with properties

class Contact:

 @property
 def name(self):
 return self.__name

 @name.setter
 def name(self, name):
 if not Contact.validate_text(name):
 raise Exception('Invalid name')
 self.__name = name

 @property
 def address(self):
 return self.__address

 @address.setter
 def address(self, address):
 if not Contact.validate_text(address):
 raise Exception('Invalid address')
 self.__address = address

```

Setter decorator name includes the name of the property

Setter decorator name includes the name of the property



## WHAT COULD GO WRONG

Failures in property code can be confusing. The example program **EG10-09 Time Tracker with properties** implements the name, address, and telephone number elements of a contact as properties. An attempt to set a property to an invalid value will cause an exception. The initializer for the **Contact** in this example program looks like this: [Click here to view code](#)

## image

```
def __init__(self, name, address, telephone):
 self.name = name
 self.address = address
 self.telephone = telephone
 self.__hours_worked = 0
```

It doesn't look like any of these statements would cause a program to fail. The `__init__` method uses the values of the parameters to set the values of the data attributes in the object. However, the following statement would fail: [Click here to view code image](#)

```
rob = Contact(name='Rob', address='18
Pussycat Mews, London, NE1 410S',
telephone='1234 56789')
```

This attempt to construct a `Contact` with the name `Rob` would raise an exception because the name `Rob` is only three characters long. The `__init__` method would try to set the name property to '`Rob`', and the property code would raise an exception. A programmer investigating this problem would find that it was caused by the statement: Programmers might expect a method or function to throw an exception, but they might not expect a simple attribute assignment to cause a program to fail. If we're going to implement properties, we need to be very clear about how the properties work and what will happen when they fail. The example program **EG10-10 Time Tracker with properties and exception handlers** contains exception handlers that will deal appropriately with incorrect assignments to properties.

## Evolve class design

Our lawyer customer has had another idea to improve her program. She wants to

use it for billing. Along with keeping track of hours spent on a case for a customer, she now wants the Time Tracker program to track the billing amount in dollars owed by each contact for her services. She has a simple way of calculating her prices. For every session working for a contact, the lawyer charges \$30 just to open the contact's case (the "open fee"), plus an additional \$50 per hour (the "hourly fee"). As an example, a one-hour session would cost \$80 (that's \$30 open fee plus \$50 for the hour).

She wants us to modify the behavior of the [Add Session](#) menu item so that each time a work session is added, the program also updates the billing amount for that contact. Then, when she prints out the customer details, the program will display the billing amount as well as the working hours: [Click here to view code image](#)

```
Name: Rob Miles
Address: 18 Pussycat Mews, London, NE1 410S
Telephone: 1234 56789
Hours on the case: 2.0
Billing amount: 130.0
```

This is the output she'd like to see. She has had a single two-hour session with Rob Miles, and the billing amount is \$130 (\$30 to open the case and \$100 for two hours).



## CODE ANALYSIS

Managing the billing amount This code analysis is a bit different, because we will consider how to design our code rather than look at program code that has already been written.

**Question:** How would we store the billing amount for a contact?

**Answer:** This would be held as a data attribute in the `Contact` class. We should store and manage this value in a very similar manner to the `__hours_worked` value. Each `Contact` will contain a data attribute to hold the hours worked and another to hold the billing amount.

Once we've decided on a need for an attribute, we then pick a name for the attribute. The name `__billing_amount` should work well.

**Question:** Why does `__billing_amount` have two leading underscores in the name?

**Answer:** A class attribute with a name beginning with two underscores is intended to be private within that class and not for direct use by code outside that class. In other words, only methods inside the `Contact` class should use the value in `__billing_amount`, and the leading underscores are there to indicate this. We decided that `__hours_worked` should not be changed outside the class, and `__billing_amount` should be managed in the same way. We'll provide access to the `__billing_amount` value in the `Contact` class by creating a read-only property: [Click here to view code image](#)

```
@property
def billing_amount(self):
 return self.__billing_amount
```

It's easy to create a read-only property. We just omit the setter method. Now, users of the `Contact` class can read the billing amount, but they can't change it.

[Click here to view code image](#)

```
print('Rob owes:', rob.billing_amount)
```

The above statement would print the billing amount for a **Contact** referred to by the variable **rob**.

**Question:** What would the statement calculating the billable amount for a session look like?

**Answer:** A Python statement calculating the billable amount would look like this: [Click here to view code image](#)

```
amount_to_bill = 30 + (50 * session_length)
```

The **session\_length** value is multiplied by **50** (the hourly fee) and then added to **30** (the open fee). We can then add this amount to the billing amount for this customer.

[Click here to view code image](#)

```
self.__billing_amount =
self.__billing_amount+amount_to_bill
```

**Question:** Is it sensible to just use the values **30** and **50** in this code?

**Answer:** No. The program will work, but one might have difficulty remembering which value is the open fee and which value is the hourly fee. We can improve the program a lot by using class variables to hold these values, as we did for the **maximum** and **minimum** values of **session\_length**. We declare these as part of the **Contact** class because there's no need to store them for each contact because the lawyer has told us that she charges all her customers the same amount.

[Click here to view code image](#)

```
class Contact:
 __open_fee = 30
```

```
__hourly_fee = 50
```

Note that I've flagged these two attributes as private (by beginning the names with two underscore characters) because we don't want them to be changed from outside the class.

We can then use these class attributes to calculate the amount to bill for a session.

[Click here to view code image](#)

```
amount_to_bill = Contact.__open_fee +
(Contact.__hourly_fee * session_length)
```

**Question:** Where should the statement above go into the program?

**Answer:** This is a very good question. The best place to put this code is in the same place in which we add a session to a `Contact` instance—the `add_session` method inside the `Contact` class.

[Click here to view code image](#)

```
def add_session(self, session_length):
 '''
 Adds the value of the parameter
 onto the hours spent with this contact
 Raises an exception if the session length
 is invalid
 '''
 if not
 Contact.validate_session_length(session_length):
 raise Exception('Invalid session
length')
 self.__hours_worked = self.__hours_worked
 + session_length
 amount_to_bill = Contact.__open_fee +
```

```
(Contact.__hourly_fee * session_length)
 self.__billing_amount =
self.__billing_amount + amount_to_bill
 return
```

Once the `hours_worked` value has been updated, the `add_session` method calculates the amount to bill the contact and adds it to the billing amount for that contact.

We just change the `display_contact` method so that it prints the billing amount, and the new feature is complete.

[Click here to view code image](#)

```
def display_contact():
 """
 Reads in a name to search for and then displays
 the content information for that name or a
 message indicating that the name was not found
 """
 print('Find contact')
 search_name = read_text('Enter the contact
name: ')
 contact = find_contact(search_name)
 if contact != None:
 # Found a contact
 print('Name:', contact.name)
 print('Address:', contact.address)
 print('Telephone:', contact.telephone)
 print('Hours on the case:',
contact.hours_worked)
 print('Amount to bill:',
contact.billing_amount)
 else:
 print('This name was not found.')
```

This `display_contact` method finds the contact and then displays the contact details. Note that both the hours worked and the billing amount information about a contact are now provided as properties. You can find the modified program in the sample **EG10-11 Time Tracker with Billing Amount**.

## Manage class versions

Adding the new data seems to have gone very well, but there is a problem with our new program that our lawyer customer will find very quickly. The new program does not work with any of her existing contact data. She will notice that while the program can be started, the program will fail whenever she tries to add a new session or find a contact, and she'll see the following error message: [Click here to view code image](#)

```
Traceback (most recent call last):
 File "C:/Users/Rob/EG10-11 Time Tracker with
Billing Amount.py", line 257,
 in <module>
 display_contact()
 File "C:/Users/Rob/EG10-11 Time Tracker with
Billing Amount.py", line 160,
 in display_contact
 print('Amount to bill:',
contact.billing_amount)
 File "C:/Users/Rob/ EG10-11 Time Tracker with
Billing Amount.py", line 79,
 in billing_amount
 return self.__billing_amount
AttributeError: 'Contact' object has no attribute
'_Contact__billing_amount'
```

The new program contains a read-only property in the `Contact` class that returns the billing amount for the contact. Unfortunately, this fails because the property tries to use the `__billing_amount` attribute, which doesn't exist in a contact loaded from an old file.

## Add a version attribute to a class

The best way to solve this problem is to have a version number attribute in each contact that we store. If the application loads a contact with an old version number, it can detect the old version and upgrade the old contact into a new one. The version number will be just another data attribute stored in the class and set when the class is created.

```
def __init__(self, name, address, telephone):
 ...
 Initializes a version 1 contact
 ...
 self.name = name
 self.address = address
 self.telephone = telephone
 self.__hours_worked = 0
 self.__version = 1
```

Set the version number to 1

This is the `__init__` method for a “version managed” `Contact` that doesn’t perform session billing. It sets the name, address, and telephone number to the parameters supplied and then sets the `__hours_worked` attribute to `0`. It also sets the `__version` attribute to `1` to indicate that this is a version 1 `Contact` object.

## Check version numbers

We can then create a method to check the version of a contact and make sure it’s up to date, which would be called after a `Contact` has been loaded: [Click here to view code image](#)

```
def check_version(self):
 ...
 Checks the version number of this instance of
 Contact and upgrades the object if required.
 ...
 pass
```

This `check_version` method doesn’t do anything now because version 1 is the first version of our `Contact`. However, we need to add it at this point

because it will be used each time a contact is loaded:

```
def load_contacts(file_name):
 ...
 Loads the contacts from the given file name
 Contacts are stored in binary as a pickled file
 Exceptions will be raised if the load fails
 ...

 global contacts
 print('Load contacts')
 with open(file_name, 'rb') as input_file:
 contacts = pickle.load(input_file)
 # Now update the versions of the loaded contacts
 for contact in contacts: Work through all the loaded contacts and check their versions
 contact.check_version() Ask this contact to check its version
```

When the contacts have been loaded, the `for` loop at the end of the `load_contacts` function will work through the contacts and call `check_version` to check the version of each contact. The `check_version` method will make sure that each contact is up to date.

## Upgrade a class

Now, let's change the application and add the `__billing_amount` attribute to it. To do this, we must create a new version of the `Contact`, which contains an extra data value. The initializer of this version of the `Contact` will set the billing amount of the new contact to `0`, and it will set the version number to `2`.

```
def __init__(self, name, address, telephone,email):
 ...
 Initializes a version 2 contact
 ...

 self.name = name
 self.address = address
 self.telephone = telephone
 self.__hours_worked=0
 self.__billing_amount=0 Billing amount attribute
 self.__version = 2 Set the version number to 2
```

If my upgraded program tries to use an old contacts file, it will fail because there is no `__billing_amount` attribute in the old file. This is what caused the

error our lawyer saw when she tried to open an old file using the modified Time Tracker. However, now that our program is tracking the versions of the data it is working with, we can add some code to the `check_version` to fix this problem.

```
def check_version(self):
 ...
 Checks the version number of this instance of
 Contact and upgrades the object if required.
 ...
 if self.__version == 1: Check the version of this Contact
 # version 1 of this class does not have a billing amount
 # create a billing amount attribute of zero
 self.__billing_amount = 0 Set the billing amount to 0
 # upgrade the contact to version 2
 self.__version = 2 Upgrade the version number
```

The `check_version` method is called after the contact has been loaded. It tests the version number of this contact. If the version is 1, a `__billing_amount` data attribute set to `0` is added to the object. Once the `__billing_amount` data attribute has been added, the class is compatible with version 2 contacts, so the version number is increased to reflect this. When the class is stored, the updated version number will be stored in the contact, along with the billing amount.



## MAKE SOMETHING HAPPEN

Explore version management To get a better understanding of what we have just done, we can use two of the sample programs: Start IDLE and load the programs **EG10-12 Time Tracker with version management** and **EG10-13 Time Tracker with version managed billing**

Run the program **EG10-12 Time Tracker with version management** and use menu option **1** to create a new contact.

[Click here to view code image](#)

```
Enter your command: 1
Create new contact
Enter the contact name: Rob Miles
Enter the contact address: 18 Pussycat Mews,
London, NE1 410S
Enter the contact phone: 1234 56789
```

Then use menu option **2** to find that contact and view it.

[Click here to view code image](#)

```
Enter your command: 2
Find contact
Enter the contact name: Rob
Version: 1
Name: Rob Miles
Address: 18 Pussycat Mews, London, NE1 410S
Telephone: 1234 56789
Hours on the case: 0
```

This program prints the version of the contact along with other contact information. You can see that this is version 1 of the contact. Now, stop the program and save the data by using command **5**.

[Click here to view code image](#)

```
Enter your command: 5
save contacts
```

Now, run the program **EG10-13 Time Tracker with version managed**

**billing.** It will load the contacts list and upgrade it. Enter command number 2 to view a contact, and view the one you just created.

[Click here to view code image](#)

```
Enter your command: 2
Find contact
Enter the contact name: Rob
Version: 2
Name: Rob Miles
Address: 18 Pussycat Mews, London, NE1 410S
Telephone: 1234 56789
Hours on the case: 0
Billing amount: 0
```

As you can see above, the contact is now version 2, and it has a billing amount that was set up by the `check_version` method when the contacts were loaded.



## PROGRAMMER'S POINT

Add version management when you design data storage Whenever I start a project for a customer, I consider which items I'm storing will require version management. In the case of the Time Tracker, there was a good chance that our customer would want to add features to the system, and so we should have considered version management at the very beginning of the project.

The above process is followed every time a new version of an application is installed. New features usually mean changes to the underlying data, and the application manufacturers have very well-developed processes for doing this.

When you're trying to work out how long it will take to write a program for a customer, it is very important that you allow for the time you will spend writing code to deal with updates to the data. This explains why programs that seem to be trivial can actually involve a lot of work.

## The `__str__` method in a class

One thing we've noticed is that each time we add a new attribute to the Contact object, we must update the `display_contact` method in our application. We also need to make sure it prints the value of the new attribute. It would be nice if we could just print the contact, rather than having to print each data attribute in turn.

```
def display_contact():
 """
 Reads in a name to search for and then displays
 the content information for that name or a
 message indicating that the name was not found
 """

 print('Find contact')
 search_name = read_text('Enter the contact name: ')
 contact=find_contact(search_name)
 if contact!=None:
 # Found a contact
 print(contact) -> Just print the contact
 else:
 print('This name was not found.')
```

This version of `display_contact` prints the contact rather than printing each data attribute from the contact. Unfortunately, just printing the contact doesn't work:

## Time Tracker

1. New Contact
2. Find Contact
3. Edit Contact
4. Add Session
5. Exit Program

Enter your command: 2

Find contact

Enter the contact name: Rob

<\_\_main\_\_.Contact object at 0x0000018E5E9EBB70>

Output from the default  
object printing method

Above, you can see the result of using the new `display_contact` method. The default print behavior for an object simply prints the type of the object being printed and the physical address of the object in memory. However, we can replace this behavior with a new version by adding a new method into the `Contact` class:

```
class Contact:

 def __str__(self):
 return 'Name: ' + self.name + '\n' + \
 'Address: ' + self.address + '\n' + \
 'Telephone: ' + self.telephone + '\n' + \
 'Hours on the case: ' + str(self.hours_worked) + '\n' + \
 'Amount to bill: ' + str(self.billing_amount)
```

New `__str__` method  
Continuation character  
on the end of the line  
Convert the  
hours\_worked  
number into  
a string

Whenever Python needs the string version of an object, it calls the `__str__` method provided by that object. The classes that represent numeric objects, such as `int` and `float`, have `__str__` methods that return their value expressed as a string; this is how we can print numeric values in our programs. Classes that we create inherit their `__str__` behavior from the object on which they are based. We'll discuss inheritance in detail in the next chapter.

The default `__str__` behavior returns the simple description we saw above. However, we can provide an object with its own `__str__` method that the object can use to return a string describing its contents. In the example above, you can see that the method assembles a string and returns it.

The `__str__` method uses something we haven't seen before. The expression that combines all the various string elements to create the description to return is very long. We use a "continuation character" on the end of each line of the expression to tell Python that the expression continues on the next line. The continuation character is a single backslash (\), as you can see above.

If we add this `__str__` method to our `Contact` class, the print behavior works correctly: [Click here to view code image](#)

```
Name: Rob Miles
Address: 18 Pussycat Mews, London, NE1 410S
Telephone: 1234 56789
Hours on the case: 3.0
Amount to bill: 180
```

## Python string formatting

The expression that creates the string to be returned by the `__str__` method is very long and rather tedious for us to create. We must remember to use the `str` function to convert all the numeric values for hours worked and amount to bill into strings so that they can be assembled into a result. Python has a way to make this much easier. A program can use the `format()` method to create a formatted string. We've seen how strings can expose methods, such as `upper()`, which returns an uppercase version of the text in the string. The `format()` method is given a set of values and inserts them into the string, which serves as a template for the output that we want. The positions for the insertions are given by placeholders.

```
EG10-15 Time Tracker with formatted string

class Contact:

 def __str__(self):
 template = '''Name: {0} _____ Format string
Address: {1} _____ Placeholder for the address
Telephone: {2}
Hours on the case: {3}
Amount to bill: {4} '''
 return template.format(self.name, self.address, self.telephone,
 self.hours_worked, self.billing_amount) _____ Format method
```

Above, you can see how this would be used to format the string that describes a contact object. The values in the call to the format method are inserted at the points marked by the placeholders for each value. A placeholder is expressed as `{n}`, where `n` is the position of the argument in the call of format. The argument at the start of the list of values is numbered `0`.



## MAKE SOMETHING HAPPEN

Adventures with string formatting We can find out how string formatting works by using the Python Command Shell in IDLE. Open it and enter the statements below.

[Click here to view code image](#)

```
>>> name = 'Rob Miles'
>>> age = 21
```

These statements create two variables that hold my name and my age. Now we can create a template string to be formatted.

[Click here to view code image](#)

```
>>> template = 'My name is {0} and my age is {1}'
```

This creates a new string value called `template`. We can then call the `format` method on the template string. Type the following and press **Enter**.

[Click here to view code image](#)

```
>>> template.format(name, age)
```

The `format` method returns a string that contains the parameter values inserted in it: [Click here to view code image](#)

```
'My name is Rob Miles and my age is 21'
```

The `format` method converts items into strings before printing them, but we can add more formatting information if we wish. (I'm just showing the templates and their outputs in these examples.) [Click here to view code image](#)

```
template = 'My name is {0:20} and my age is {1:10}'
'My name is Rob Miles' and my age
is 21'
```

The placeholder can be followed by a width value as shown above, in which case, the item is printed in that width. Spaces are added if required, which is very useful if you want to print things in columns.

Strings are normally aligned on the left edge when they are printed, and numbers are aligned on the right. We can select which alignment to use

by adding > or < characters as shown below: [Click here to view code image](#)

```
template = 'My name is {0:>20} and my age is
{1:<10}'
'My name is Rob Miles and my age
is 21 '
```

If you're printing a floating-point number, you can set the number of decimal places to be printed: [Click here to view code image](#)

```
template = 'My name is {0:20} and my age is
{1:10.2f}'
'My name is Rob Miles and my age
is 21.00'
```

This template prints the age value as a floating-point value with two decimal places, in a width of 10 characters. There are other formatting options you can use to center text and to control how numbers are displayed. They are described in the Python documentation here:  
<https://docs.python.org/3.6/library/string.html>

## Session tracking in Time Tracker

The Time Tracker application is turning into a bit of a monster. Our customer is getting very enthusiastic about the program and keeps having new ideas. This is good news for us because it keeps us busy. Her latest idea is a very good one. She has decided that it would be very useful to be able to record exactly when a given session for a client took place. She's drawn up a specification of what she wants to see: [Click here to view code image](#)

Time Tracker

1. New Contact
2. Find Contact
3. Edit Contact
4. Add Session
5. Exit Program

Enter your command: 2

Enter the contact name: Rob

Name: Rob Miles

Address: 18 Pussycat Mews, London, NE1 410S

Telephone: 1234 56789

Hours on the case: 10.0

Amount to bill: 470.0

Sessions

Date: Mon Jul 10 11:30:00 2017 Length: 1.0

Date: Tue Jul 12 11:30:00 2017 Length: 2.0

Date: Wed Jul 19 11:30:00 2017 Length: 2.5

Date: Wed Jul 26 10:30:20 2017 Length: 2.5

Date: Mon Jul 31 16:51:45 2017 Length: 1.0

Date: Mon Aug 14 16:51:45 2017 Length: 1.0

The Find Contact command now shows a list of sessions, when each took place, and the length of each session in hours. This looks like it might be difficult to add to the Time Tracker, but it's a good way for us to explore class design and look at some interesting features of the Python language.



## CODE ANALYSIS

Creating a session class In this Code Analysis, we'll design some code and then take a look at how it works.

**Question:** How will we store information about a session?

**Answer:** Whenever we need to store a set of related information, we should think about creating a class to hold that information. We should give the class a name (I suggest `Session`) and then identify the data attributes that the class should contain. In this case, we are storing two items: the length of the session and the date and time that the session ended. We can initialize these values in an `__init__` method for the `Session` class: [Click here to view code image](#)

```
class Session:

 __min_session_length = 0.5
 __max_session_length = 3.5

 @staticmethod
 def validate_session_length(session_length):
 """
 Validates a session length and
 returns
 True if the session is valid or False
 if not
 ...
 if session_length <
Session.__min_session_length:
 return False
 if session_length >
Session.__max_session_length:
 return False
 return True

 def __init__(self, session_length):
 if not
Session.validate_session_length():
 raise Exception('Invalid session
length')
 self.__session_length =
session_length
```

```
 self.__session_end_time =
 time.localtime()
 self.__version = 1
```

The Time Tracker application can now create an object that describes a particular session: [Click here to view code image](#)

```
session_record = Session(session_length)
```

This statement will create a `Session` with the session length supplied as a parameter. The `validate_session_length` method has been moved inside the `Session` class. It is used to validate the session length when a new `Session` object is created. If the session length is invalid, the `__init__` method raises an exception. The `__init__` method uses the `time` library to read the local time when the `Session` object is created. This is stored in the `__session_end_time` attribute of the `Session` object.

**Question:** Are we using version control for the `Session` class?

**Answer:** Yes, we are using version control. We want to be able to keep the lawyer happy if she suggests new things she wants to store about each session. Perhaps she will want to be able to make a note of the location of a session or who was present at a meeting. Adding version control now will make it possible for us to add extra features to our session records without breaking existing stored data. This means that the `Session` class will also contain a `check_version` method that can be used to update a session object if required.

[Click here to view code image](#)

```
def check_version(self):
 pass
```

Currently, this method does nothing because we are creating version 1 of the **Session** class.

If you look at the pattern for construction of the **Session** object, you'll find that it is heavily based on the construction of the **Contact** object. This is not accidental. It is very sensible to use a particular format for the design of an object and then repeat it across an application.

**Question:** How will we allow users of the **Session** class to get the session length and session end time items from a **Session** object?

**Answer:** We can expose these as properties of the class, but we won't provide a setter method for the properties. This makes it possible for programs to read the values but not change them. We used the same technique with the billing amount and hours worked values of the **Contact** class.

[Click here to view code image](#)

```
@property
def session_length(self):
 return self.__session_length

@property
def session_end_time(self):
 return self.__session_end_time
```

**Question:** Will the **Session** class have an **\_\_str\_\_** method?

**Answer:** Yes, it will. It will return a string that describes the contents of the **Session**.

```
def __str__(self):
 template = 'Date: {0} Length: {1}'
 date_string = time.asctime(self.__session_end_time) ————— Convert the time into a string
 return template.format(date_string, self.__session_length)
```

The time library contains a function called **asctime()** that takes a

`localtime` value and returns a string containing the time. This is used to get a date string, which is then used in a template to create the string to be returned.

Now that we have our `Session` class, the next thing to do is incorporate this into the Time Tracker application. Each `Contact` object will contain a list of sessions. The list will be created when the `Contact` is initialized:

```
class Contact:

 def __init__(self, name, address, telephone):
 self.name = name
 self.address = address
 self.telephone = telephone
 self.__hours_worked = 0
 self.__billing_amount = 0
 self.__sessions = [] Create a list to hold the sessions for this contact
 self.__version = 3 This is version 3 of the Contact object
```

The `__init__` method above creates the list of sessions for this contact. Note that the `Contact` class is now at version 3. Contact version 1 was the original contact. Contact version 2 added the billing amount to each contact. Version 3 adds session tracking. The `check_version` for a version 3 method will add a session list to an older version `Contact` object.

```

class Contact:

 def check_version(self):
 """
 Checks the version number of this instance of
 Contact and upgrades the object if required.
 """

 if self.__version == 1: Check for a version 1 Contact
 # version 1 of this class does not have a billing amount
 # create a billing amount attribute of zero
 self.__billing_amount = 0 Add a billing amount to a version 1 Contact
 # upgrade the contact to version 2
 self.__version = 2 Upgrade the version of the Contact to version 2

 if self.__version == 2: Check for version 2 of the Contact
 # Version 2 of this class does not have a session list
 # Add an empty session list
 self.__sessions = [] Add an empty session list to a version 2 Contact
 # upgrade the contact to version 3
 self.__version = 3 Upgrade the version of the Contact to version 3

 # Now check the versions of each of the sessions
 for session in self.__sessions: Update all the sessions in this Contact
 session.check_version()

```

If the Time Tracker application opens a very old file of version 1 contacts, you will see that the contacts will first be upgraded to version 2 and then upgraded to version 3 right away. Note that the `check_version` method also calls a `check_version` method on each of the sessions in the contact, using a `for` loop to work through the sessions.

We add a new session record to the `Contact` in the `add_session` method, which is part of the `Contact` class. Previously, this method just updated the values of the hours worked and amount to bill data attributes. Now, it creates a new `Session` record and adds it to the list of sessions held in the `Contact`.

```

class Contact:

 def add_session(self, session_length):
 """
 Adds the value of the parameter
 onto the hours spent with this contact
 Raises an exception if the session length is invalid
 """
 if not Session.validate_session_length(session_length):
 raise Exception('Invalid session length')
 self.__hours_worked = self.__hours_worked + session_length
 amount_to_bill = Contact.__open_fee + (Contact.__hourly_fee *
 session_length)
 self.__billing_amount = self.__billing_amount + amount_to_bill
 session_record = Session(session_length) Create a session record for this session
 self.__sessions.append(session_record) Add it to the list of sessions

```

The final thing we must consider is how we'll get a list of the sessions out of a `Contact`. We must create a string that contains a line for each session. This is the format that the lawyer specified when she suggested the new feature. The starting point for this string is the list of session objects in a `Contact`. This must be converted into a string, which can be printed in a report.

```

@property
def session_report(self):
 # Convert the list of sessions into a list of strings
 report_strings = map(str, self.__sessions) Use map to convert each Session in
 the list to a string
 # Convert the list of strings into one string
 # separated by newline characters
 result = '\n'.join(report_strings) Use join to convert the list of strings
 into a single string
 return result

```

I'm quite proud of this method. It returns a string that contains a list of sessions. It uses the Python functions `map` and `join`, which are worth knowing. However, if it looks strange to you, don't worry. We'll explore in detail how we can use these functions to go from a list of sessions to a long string that contains a report.

## The Python `map` function

We want to convert a list of `Session` objects into a list of strings. The `str`

function can be applied to an object to get the string version of that object, and we can use the `map` function to apply this function to all the `Session` objects in the `__sessions` list.

The `map` function is a great example of the power of Python. It accepts two arguments when called. The first argument is the name of a function that accepts a single parameter and returns a result. The second argument is a list of items on which the function can work. Python allows us to use function names just like any other value in a program. Functions can be stored in variables and passed as arguments to method calls. We'll investigate how to do this in [Chapter 11](#). For now, just work on the basis that the first argument to a call of `map` is the name of the function that you apply to each item in the list.



## MAKE SOMETHING HAPPEN

Investigating the `map` function and iteration Please note that this is an important, and rather long, piece of investigation. At the end of it, you will have learned not only how the `map` function is used, but also some fundamental things about the way Python works.

We can find out about the `map` function by using the Python Command Shell in IDLE. We'll use `map` to indent a list of text strings. Indenting is a large part of how Python programs are structured. The IDLE program editor even includes a command you can use to indent a block of text (**Format, Indent**). We'll build our indenter using the `map` function. Open the IDLE Command Shell and enter the statement below.

[Click here to view code image](#)

```
>>> code = ['line1', 'line2', 'line3']
```

```
>>>
```

This statement creates a list that contains three string values. If we just enter the name of the list, Python will show us the contents.

```
>>> code
```

The Python Command Shell will show us the value of any expression entered, so the contents of the code list will now be displayed.

[Click here to view code image](#)

```
>>> code
['line1', 'line2', 'line3']
>>>
```

Next, we need to create a function that will indent a string for us. We can indent a string just by adding four spaces to the beginning of the string. Enter the following Python code to create a function that will indent a string provided as a parameter. Remember to enter a blank line after the return to end the definition of the function.

[Click here to view code image](#)

```
>>> def indent(x):
 return ' '+x

>>>
```

We can test the `indent` function by giving it a string and seeing what the function returns. Enter the following statement.

```
>>> indent('Rob')
```

This will call the function `indent` and pass it to the argument '`Rob`'. The result of the call of the function will be displayed.

```
>>> indent('Rob')
' Rob'
>>>
```

We would like to apply the `indent` function to every string in the `code` list to produce indented lines of code. We could create a `for` loop to do this, but instead we'll use the `map` function to apply the `indent` function to each of the items of code. Type in the following statement: [Click here to view code image](#)

```
>>> indented_code = map(indent, code)
>>>
```

Enter the above statement to set the variable `indented_code` to the result of the `map` function. You might think that when we print `indented_code`, you'll see a list of indented code. View the contents of the `indented_code` variable by entering its name.

```
>>> indented_code
```

When you press **Enter**, Python will show you the contents of the `indented_code` variable.

[Click here to view code image](#)

```
>>> indented_code
<map object at 0x00000211E6FCBA58>
>>>
```

This is very confusing. Instead of a list of strings, we have a thing

called a [map object](#). What is happening here?

What we see here is a splendid example of the cleverness of Python. Rather than giving us a processed list of objects, the [map](#) function instead returns something called an iterator. An iterator is an object that a program can “work through” one item at a time.

The usual way of working through an iterator is to use a [for](#) loop. List objects are also iterators, which is how we have written [for](#) loops that work through items in a list. The [range](#) function also returns an iterator as a result so that we can write loops that can count.

We can write a [for](#) loop to work through the values returned by the [indented\\_code](#) iterator and print each item.

[Click here to view code image](#)

```
>>> for s in indented_code:
 print(s)
```

Enter the above [for](#) loop and the [print](#) statement. Enter an empty line after the [print](#) statement to cause the loop to run.

[Click here to view code image](#)

```
>>> for s in indented_code:
 print(s)

line1
line2
line3
```

This is the list of strings that we were expecting. Each time around the loop, the value in [s](#) is the next item returned by the iterator. Note that each line has been indented by four spaces.

Iterators are a way of saving memory. The designers of Python said,

“There’s no need for the `map` function to produce a list. Instead, it could just provide us with an iterator object that can provide each list element in turn when we ask it.”

You can think of the map iterator as a little factory. Each time the loop iterating the map needs another item, it asks the map iterator for it. The map iterator gets the next value out of the source, applies the function it has been told to use to that item and then returns it. When the map iterator runs out of values to return, it raises a “`StopIteration`” exception to tell the loop there are no more items available. This stops the loop.

We can explore this by recreating the iteration and then doing what a `for` loop would do with the iteration. Repeat the statement that creates the map.

[Click here to view code image](#)

```
>>> indented_code = map(indent, code)
```

Now, we can ask the `indented_code` iterator to give us the next value in the iteration by calling the method `__next__` on the iterator object. Type the statement below and press **Enter** to run it.

[Click here to view code image](#)

```
>>> indented_code.__next__()
```

This is the point at which the `indent` method will be called to produce the next value from the iteration.

[Click here to view code image](#)

```
>>> indented_code.__next__()
'line1'
>>>
```

This is the first line in the indented list. We can view successive lines by calling the `__next__` method again. Perform three more calls and see what happens.

[Click here to view code image](#)

```
>>> indented_code.__next__()
'line1'
>>> indented_code.__next__()
'line2'
>>> indented_code.__next__()
'line3'
>>> indented_code.__next__()
Traceback (most recent call last):
 File "<pyshell#67>", line 1, in <module>
 indented_code.__next__()
StopIteration >>>
```

After the third call of `__next__` the iteration runs out of items, so it raises the `StopIteration` exception to indicate that there are no items left. Once an iteration has been completed, it can't be used again. You must create a new iteration if you want to make another pass through the data. Let's do that now. Enter the following statement:

[Click here to view code image](#)

```
>>> indent_iterator = map(indent, code)
```

This creates a new iterator called `indent_iterator`, which will iterate through the code list and apply the `indent` function to each element.

We can use the Python function `list` to create a new list from this iterator. The `list` function creates an empty list and then adds successive iterations from the iterator to that list. Enter the following statement to do this: [Click here to view code image](#)

```
>>> indented_code = list(indent_iterator)
```

```
>>>
```

We can now view the contents of the `indented_code` list and see that it is now a list of strings. Type in the name and press **Enter**.

```
>>> indented_code
```

The Python Shell will now display the contents of the `indented_code` list: [Click here to view code image](#)

```
>>> indented_code
[' line1', ' line2', ' line3']
>>>
```

This shows that we now have our indented lines of code.

One mind-bending possibility is that the input of a `map` function is actually an iterator. Enter the following statements to explore this: [Click here to view code image](#)

```
>>> i1 = map(indent, code)
>>> i2 = map(indent, i1)
```

The first statement creates an iteration called `i1` that applies the `indent` function to all the items in `code` list. The output of this iteration will be code lines indented by four characters.

The second statement creates an iteration called `i2` that applies the `indent` function to all items in the `i1` iteration. We can then use the `list` function to convert the `i2` iteration into a list and look at it.

```
>>> list(i2)
```

The `list` function creates a list of items generated by the `i2` iterator. These items are then displayed by the Python Command Shell: [Click here to view code image](#)

```
>>> list(i2)
[' line1', ' line2', '
line3']
```

As you might expect, each line has been indented twice. Once by the `i1` iteration, and again by the `i2` iteration. Python makes it very easy to create “chains” of iterators to work through data. Note that if nothing ever iterates through the iterator returned by a map, none of the items in the iterator will ever be generated.

Now that we know what `map` does, and a lot of other useful things about how Python processes data, we can re-visit the statement in the `session_report` method that generates a list of report strings.

[Click here to view code image](#)

```
report_strings = map(str, self.__sessions)
```

Remember that the starting point for our report is a list of `Session` objects in a list in the variable `self.__sessions`. These `Session` objects need to be converted into strings to be used in the report. The `map` function will create an iterator that will apply the `str` function to each element in the `self.__sessions` list. The `str` function acts on an object to return the string that describes that object. In other words, the `str` function calls the `__str__` method in an object. The `Session` class contains a `__str__` method, which we discussed in the “[Code Analysis: Creating a session class](#)” section earlier in this chapter. It generates a session description in the format our lawyer wants to see.

The next phase of the conversion of our session list into a reported string will

work through the `report_strings` iterator and generate the string result that will be printed.

## The Python `join` method

I hope that by now you're beginning to see that the elements of Python that we're using are objects with method attributes. Just like a `Contact` object provides methods such as `add_session`, a string object provides methods such as `lower()` (to return a lowercase version of the string). Python will allow programs to call string methods directly on strings of text in a program.

```
'FRED'.lower()
```

This is completely legal Python and would create the string '`fred`'. When Python runs the program, it converts the string '`FRED`' into an object of type string and then calls the `lower()` method on that object.

Another method provided by a string object is called `join(iterator)`. The `lower()` function doesn't accept any arguments, but the `join(iterator)` function is supplied with something to iterate through. It does what its name implies. It works through the iteration, adding each successive value to a string and joining each value to the next with a copy of itself.

[Click here to view code image](#)

```
report_result = '\n'.join(report_strings)
```

The statement above creates a single string called `report_result`, which consists of all the elements returned by the iterator `report_strings`, joined by a newline character. This gives the report format that we want.



MAKE SOMETHING HAPPEN

Investigating the `join` function This will be a slightly shorter investigation than the previous one. We can find out about the `join` function by using the IDLE Command Shell. Open the IDLE Command Shell and enter the statement below.

[Click here to view code image](#)

```
>>> report_strings = ['report1', 'report2',
'report3', 'report4']
>>>
```

This creates a list called `report_strings`, which contains four strings. As we know, a list can be used as an iterator, so we can use this list in a `join` function.

[Click here to view code image](#)

```
>>> '**'.join(report_strings)
```

This statement iterates through each of the elements in `report_strings`, adding them together and inserting the `**` character sequence between each element. The Python Command Shell will show us the result of the expression.

[Click here to view code image](#)

```
>>> '**'.join(report_strings)
'report1**report2**report3**report4'
>>>
```

This is one long string with two asterisks between each line. If we use `\n` (newline) as the joining character, we can get each line of the report

on a separate line.

[Click here to view code image](#)

```
>>> print('\n'.join(report_strings))
report1
report2
report3
report4
>>>
```

If we want to just concatenate the items in the list of strings we can use `join` on an empty string.

[Click here to view code image](#)

```
>>> ''.join(report_strings)
'report1report2report3report4'
>>>
```

The example program **EG10-16 Time Tracker with session history** contains a complete Time Tracker application that records individual sessions for each contact. It also automatically upgrades older versions of the `Contact` class.

This application is a very good starting point for any program that you might like to write that stores and manages information. You could replace the sessions and contacts with albums and music tracks, salesman and sales, artists and pictures—or anything else that you want to track.

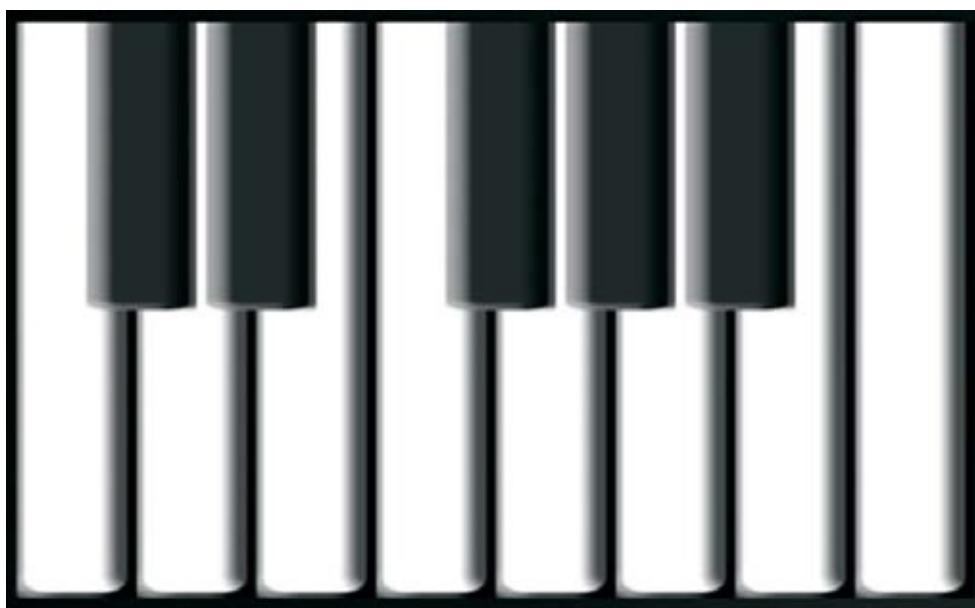
## Make music with Snaps

We have spent a while building a Time Tracker application. Now we can have some fun and play some music. We'll create a simple music player and then look at how we can use Python language features to make it easier to manage music

playback.

The snaps library is supplied with a folder of musical note samples that can be used to play tunes using the snaps `play_sound` function. The name of each sample corresponds to a particular musical note. The snaps function `play_note` can be used to play one of the notes.

**Figure 10-1** shows how the note numbers are mapped onto piano keys.



**Figure 10-1** Note numbers

The sound **samples** are held in a folder called **MusicalNotes**. It is *very important* that this file is present in the same folder as the snaps framework. Otherwise your program will not play sounds correctly. If you run the sample programs for this chapter from their original folder, they will be loaded correctly.

```
EG10-17 Play notes
import time
import snaps

for note in range(0,13):
 snaps.play_note(note)
 time.sleep(0.5)
```

Work through all the note values  
Play the note  
Pause to allow the note to sound

The example program **EG10-17 Play notes** will play all the notes one after the

other, with a half-second delay between each note. We can use the `play_note` method to make a program that will play a tune.

[Click here to view code image](#)

```
EG10-18 Twinkle Twinkle
import time
import snaps

snaps.play_note(0)
time.sleep(0.4)
snaps.play_note(0) time.sleep(0.4)
snaps.play_note(7)
time.sleep(0.4)
snaps.play_note(7)
time.sleep(0.4)
snaps.play_note(9)
time.sleep(0.4)
snaps.play_note(9)
time.sleep(0.4)
snaps.play_note(7)
time.sleep(0.8)
```

This listing program will print the first part of “Twinkle, Twinkle Little Star.” The example program **EG10-18 Twinkle Twinkle** prints a slightly longer song. This program uses a sequence of method calls that play each note in turn. If we want to play a longer tune, we must add more lines to the program.

A better way to play the music would be to make the program *data driven*. Rather than expressing the required notes as values in the method calls, we could instead express the note and duration values as tuples.

```
EG10-19 Twinkle Twinkle Tuples
```

```
import time
import snaps

tune = [(0, 0.4), (0, 0.4), (7, 0.4), (7, 0.4),
 (9, 0.4), (9, 0.4), (7, 0.8), (5, 0.4),
 (5, 0.4), (4, 0.4), (4, 0.4), (2, 0.4),
 (2, 0.4), (0, 0.8)]
```

Create a list of tuples that contain the tune

```
for note in tune:
 snaps.play_note(note[0])
 time.sleep(note[1])
```

Work through the notes in the tune

snaps.play\_note(note[0])

The first element in the tuple holds the note number

time.sleep(note[1])

The second element in the tuple holds the note duration

Recall that a tuple is a collection of values enclosed in brackets. We create tuples to hold related values. In the case of the `tune` list above, each tuple in the list holds two values. The first is an integer that specifies the note to be played; the second is the duration of the note. This version of the music player is smaller, and we can now create longer tunes just by adding more note information, but it's a little hard for other programmers to create tunes as they must know how to create the tuples and play them.

Perhaps we can make the code easier to understand by storing the note information in a class:

```

EG10-20 Twinkle Twinkle class

import time
import snaps

class Note:
 def __init__(self, note, duration):
 self.__note = note
 self.__duration = duration

 def play(self):
 snaps.play_note(self.__note)
 time.sleep(self.__duration)

tune = [Note(note=0, duration=0.4), Note(note=0, duration=0.4),
 Note(note=7, duration=0.4), Note(note=7, duration=0.4),
 Note(note=9, duration=0.4), Note(note=9, duration=0.4),
 Note(note=7, duration=0.8), Note(note=5, duration=0.4),
 Note(note=5, duration=0.4), Note(note=4, duration=0.4),
 Note(note=4, duration=0.4), Note(note=2, duration=0.4),
 Note(note=2, duration=0.4), Note(note=0, duration=0.8)]

for note in tune:
 note.play()

```

Annotations from top to bottom:

- Create a new Note instance
- Set the note to play
- Set the duration of the note
- Play the note
- Play the note sound
- Pause the program while the note sounds
- Create a list of note instances
- Work through the notes in the list
- Get each note to play itself

This version of the tune player uses a class called **Note**. The **Note** class holds the note number and duration of the note. These are set into each note by the `__init__` method. The program creates a list of **Note** instances to make up the tune. The program uses keyword arguments to identify each of the values used to create each note.



### CODE ANALYSIS

The **Note** class I'm quite happy with the design of the **Note** class. However, there are some aspects of the design that are worth considering in detail.

**Question:** Why does the `Note` class contain a `Play` method?

**Answer:** This is all to do with “cohesion.” The process of playing a note should be managed by a `Note` itself, not by something external to the `Note`. For code outside the `Note` class to be able to play a note, the code would have to have access to the note and duration values, which should be private to a `Note`. There are other advantages to structuring the program this way. If we need to change the way a note is played, we just have to change the `Play` method in the note and any programs that use a `Note` to play tunes would just work.

**Question:** Could the `Note` have a `__str__` method?

**Answer:** This would be a good idea. It would make printing notes very easy.

[Click here to view code image](#)

```
def __str__(self):
 template = 'Note: {0} Duration: {1}'
 return template.format(self.__note,
self.__duration)
```

If we add this to the `Note` class, we can then print the tune very easily: [Click here to view code image](#)

```
tune_strings = map(str,tune)
print('\n'.join(tune_strings))
```

This code uses the same structure we used to print the Sessions in the Time Tracker application. You will find sample code that uses this in the application **EG10-21 Twinkle Twinkle printer**.



## MAKE SOMETHING HAPPEN

### Make your own music

You can modify the sample programs to make your own tunes. You can even replace the note samples with other WAV files to change the musical instruments that play each note.

## What you have learned

You've learned a lot in this chapter. You learned how to create a class containing data attributes that allow it to hold data values. When a new instance of the class is created, these values are stored inside the object (remember that an object is an instance of a class). The data attributes can be initialized by the `__init__` method in the class, which can be given parameter values used to set the value of data attributes in the class.

You've seen how Python classes can contain method attributes associated with an instance of the class. A method attribute allows an object to be asked to perform a specific action by calling that method. A good example of a class method is the `add_session` method of the `Contact` class in the Time Tracker application. This class method asks the Contact to store details of a new work session performed for that contact.

Methods in classes are very similar to Python functions, but a method is provided with a reference (usually called `self`) as the first parameter of the method. The `self` parameter is set automatically when the method is called and refers to the object running the method. Python statements in the method can then access attributes of that object by using the `self` reference.

You also discovered that methods are fundamental to creating “cohesive” classes, with no need to use elements of other classes to function. A cohesive object contains all the data attributes required by that object and provides a set of method attributes so that the object can perform the task for which it was created.

Making self-contained objects that provide behaviors for others to use is a good way to create solutions that are easy to manage and update. Self-contained objects can also perform validation of actions they are asked to perform, and reject invalid requests, either by returning error messages or by raising exceptions. If a method in an object raises an exception, the program will stop unless the caller of the method takes steps to catch and deal with the exception.

Python provides features that can be used to protect data attributes from accidental damage, but there's no way we can prevent a determined programmer from making changes to data attributes in a Python object. However, we can use source code analysis programs such as Pylint ([www.pylint.org](http://www.pylint.org)) to audit a Python program and detect attempts to change protected data values.

You've seen that a class can contain "static" methods that can be used without the need to create a class instance. Static methods are useful for things such as validation. They make it possible to determine whether potential data attributes hold valid values before a program tries to use them to create an instance of the class. You also encountered properties, which provide easy access to a data attribute in a class but also give programmers the ability to get control and validate changes to the attributes.

You discovered the importance of version management and giving a class the ability to automatically upgrade the data stored inside it when new versions of the class are created. You've added an `__str__` method to a class so it can return a string that describes the contents of the object, and you found that Python string formatting is a useful way to create strings that contain the values of variables.

Finally, you took a close look at the very powerful iteration feature of Python, which make it easy to perform an action on a large amount of data. An *iterator* is an object that can be asked for successive elements in an iteration. The source of an iteration can be a list of items or even another iteration. The Python `map` function can be used to generate an iteration that applies a specific function to all the elements in an iteration. We used this to add spaces to the beginning of strings in a list of text, thereby indenting the text. We also used this process to convert elements in a list into strings by using the Python `str` function in a map.

You also discovered the `join` function, which allows a string to join a list of strings to produce a larger string. We used `join` to create a single string that contains session reports from the Time Tracker application.

Here are some points to ponder about what we have learned.

## **Why doesn't Python provide a way for a programmer to completely protect data attributes in objects?**

This is an interesting question. If you come from other programming languages (for example, Java, C++, and C#), you know that they have protection mechanisms that can mark important attributes of a class as private to that class. In these languages, private means exactly that. Code that is not part of a class is not allowed any access to private attributes. Python seems rather half-hearted in the way it provides some protection, but this can be circumvented. I think the reason that Python doesn't provide private class attributes is that the designers were concerned that programmers might think that just because something is private means it can't be accessed from outside the class. However, it would be easy for a determined programmer to add a public method to a class, or change the behavior of an existing method to corrupt the contents of a class. The key to making secure code is not the writing of the code, but the process of review you use to check the code to make sure it is secure. Python would like programmers to engage with this review process, rather than assume that making things private will make a program secure.

## **When would we use a property in our programs?**

A property provides a way that a class can control access to a data attribute in that class. One way to control access to a data attribute is to provide get and set methods for the attribute. As an example, we could have methods called `get_name` and `set_name` to manage the name attribute in a class. The `get_name` method would return the name, and the `set_name` method would accept a new name value and then set the name attribute to this value if the new value was valid.

This would work, but it makes the code that accesses the name attribute rather long-winded. A property binds methods to the get and set actions of a class data attribute, but the property can be used in the same way as a data attribute. When

code assigns a value to the property, the setter behavior runs. When code accesses the property, the get behavior runs. It is possible to create “read only” properties that don’t have the set behavior.

I use properties when I want to manage access to data in a class, but I don’t want the user of my class to keep calling get and set methods to access that data.

## **When would we create static class attributes?**

The word *static* can be a bit confusing. It’s best to regard it as meaning “always there.” I create static data attributes in classes when I want to store a value that gives information about the class, rather than about an instance of the class. Static attributes are well suited for data validation values. For example, the minimum length of a session we spend with our lawyer client is not a property of any individual session; it is a property of the session itself, and should therefore be stored as a class attribute. Validation methods—for example, a method that checks a session length value for validity—should also use static attributes, as these methods do not apply to any specific session object and may need to be used before any sessions are created.

## **Must all our objects be highly cohesive?**

Not necessarily. If I’m making a program to process a single set of data, and I know it will be used only once, and only by me, I’ll write the code in a way that is probably very poorly designed. I’ll make everything public and do whatever it takes to get the program working with a minimum of effort. However, if I’m making a program that I know will be subject to change and maintenance, and that other programmers will be working on, I’ll spend a lot of time making sure that the code is easy to understand and modify. I’ll design it so that changes to one part of the program don’t affect the behavior of another part, and I’ll create a pattern of use and naming that is easy to understand. I consider the code we have written for the Time Tracker application as close to “professional” quality, so if you’re looking for a standard, you have it in the programs in this chapter.

## **What is an iterator again?**

An *iterator* is an object that provides methods we can use to make it do things for us. We can ask an iterator to give us the next value in an iteration by calling

the `__next__` method on the iterator object. Some Python objects—for example, the list type—behave as iterators so that we can work through the elements in the list. Other objects, such as `range` and `map` objects, also behave as iterators.

It's important to remember that the Python construction consuming the iterator doesn't know where the data has come from. A Python `for` loop will just work through whatever iterator the loop is supplied with. The loop only knows that each time it calls `__next__`, it will be given the next object in the iteration and that an exception will be raised when there are no more elements to iterate.

# 11 Object-based solution design



## What you will learn

In the previous chapter, we discovered how to create genuinely useful objects. We saw how to use Python to create classes that can hold and manage data values and provide methods that allow programs to make use of them.

In this chapter, we'll build on our class knowledge to learn how to build systems that work with large numbers of different, but related data items. We'll also explore how we can connect objects via the methods they expose. Alongside all this, we'll discover more features and functions of Python, including sets.

[Fashion Shop application](#)

[Create a FashionShop component](#)

[Design with classes](#)

Python sets

What you have learned

## Fashion Shop application

Your lawyer client is very happy with her Time Tracker application. She's been showing it to her friends, and they've been very impressed—particularly a friend who runs a fashion shop and has been looking for an application to help her manage her stock. She sells a large range of clothing items and needs help tracking inventory. Stock arrives from suppliers, and she enters the details in the system. When she sells an item, she wants to remove it from stock. She would also like a way of producing reports that will show her how many of each item she has in stock. She's keen to get your help, and she's offering discounted prices, or even free clothing, in exchange.

Free fashion sounds like an interesting idea, so you sit down with your new client and talk about what she wants to do. She shows you her stock folder, which you can see in **Figure 11-1**.



**Figure 11-1** Fashion shop stock file

She tells you that each item she receives from her suppliers has a unique stock reference that she uses to track that item. She has a large binder with a page per stock item. When she gets something she hasn't stocked before, such as a new style of dress, she creates a new page for that type of item and adds it to the folder. Then, when she receives stock deliveries, she can look up the stock reference in the binder and update the stock level for that item. **Figure 11-2** shows a page in the folder.

# Mary's Fashion Shop

**Figure 11-2** Fashion shop stock page

This page holds details of a dress that she sells. Each item of clothing in stock has a page in the binder that is updated as stock arrives and is sold. Currently, Mary would be happy with the ability to just print out her entire stock list, but

later she wants to be able to do things such as determine which item has the lowest stock levels so that she can place orders for new stock. You agree on the following main menu for the application: [Click here to view code image](#)

### Mary's Fashion Shop

- 1: Create new stock item
- 2: Add stock to existing item
- 3: Sell stock
- 4: Stock report
- 5: Exit

Enter your command:

There are five options. The first menu item is used to create a new stock item. This is equivalent to adding a new page to the stock binder to describe a new item being stocked in the shop. The second menu item is used to add stock to an existing item type. This updates the page for an item and increases the number in stock. The third menu item is used when an item is sold; the fourth item produces a stock report when selected.

## Application data design

The shop sells a range of different clothing items, and each item has a particular set of information that describes that item. For every item of clothing, she needs to store the stock reference, the price, the color, and the number of units in stock. For a dress, she wants to store the size, the style, and the pattern. For pants, she wants to store the length, waist size, style, and pattern. For hats, she just stores the size. For blouses, she wants to store size, style, and pattern. Some typical descriptions look like this: [Click here to view code image](#)

```
Dress: stock reference: 'D0001' price: 100.0 color:
red pattern: swirly size: 12
Pants: stock reference: 'TR12327' price:50 color:
black pattern: plain length: 30 waist: 30
```

We can do some *data design* to identify how we'll store the stock data. Data design is performed at an early stage in application design. It is where we identify and specify how we will represent the data with which the application will work.

## Object-oriented design

It would make sense to create a class to hold each kind of data we wish to store. Programmers call this *object-oriented programming*. The idea is that elements in a solution are represented by software “objects.” The first step in creating an application is to identify these objects.

In the English language, words that identify things are called *nouns*. When trying to work out what classes a system should contain, it's a good idea to look through the description of a system and find all the nouns. As an example, consider the following description of a fast-food delivery application.

“The **customer** will select a **dish** from the **menu** and add it to his **order**.”

I've identified four nouns in the description, each of which will map to a specific class in the application. If I were working for the fast-food delivery company, I would next ask them what data they stored about customers, dishes, menus, and orders.



### PROGRAMMER'S POINT

Don't write any code before you have completed your data design For a commercial project, you would spend a lot of time on the design of the classes in your system before you wrote a single line of code. This is because design mistakes are much easier to fix at the beginning of the project, rather than after code has been written.

In the case of our fast-food management example above, we would want to make sure that the customer class holds all the information required to make the business work. We would do this by

creating “paper” versions of the classes and then working through all the usage scenarios (creating an order, cooking an order, delivering an order) to make sure that all the data the application needs is being captured.

If the application must store a customer telephone number so that the delivery driver can call for directions if needed, it is best to discover this at the beginning of the project, rather than after the entire user interface has been created.

We will write code and discuss it as we go along because we are learning about data design and Python programming. However, if I were creating a professional solution, I’d spend a lot of time away from Python working out the design before I created any classes.

When we talk to our fashion shop customer, she’ll talk about the dresses, pants, hats, blouses, and other items that she wants the application to manage. Each of these could be objects in the application and can be represented by a Python class. Each class will contain the data attributes that describe that item of clothing. Let’s start by considering just the information for dresses and pants and create some classes for these objects.

[Click here to view code image](#)

```
EG11-01 Separate classes

class Dress:
 def __init__(self, stock_ref, price, color,
pattern, size):
 self.stock_ref = stock_ref
 self.__price = price
 self.__stock_level = 0
 self.color = color
 self.pattern = pattern
 self.size = size

 @property
 def price(self):
 return self.__price
 @property
 def stock_level(self):
 return self.__stock_level

class Pants:
 def __init__(self, stock_ref, price, color,
```

```

 pattern, length, waist):
 self.stock_ref = stock_ref
 self.__price = price
 self.__stock_level = 0
 self.color = color
 self.pattern = pattern
 self.length = length
 self.waist = waist

@property
def price(self):
 return self.__price

@property
def stock_level(self):
 return self.__stock_level

x = Dress(stock_ref='D0001', price=100,
color='red', pattern='swirly', size=12)
y = Pants(stock_ref='TR12327', price=50,
color='black', pattern='plain', length=30,
waist=25)
print(x.price)
print(y.stock_level)

```

The code above defines a `Dress` class and a `Pants` class. Each class contains an `__init__` method that a program can use to set up the contents of that class. At the end of the code sample, there are two statements that create a `Dress` and a `Pants` instance. The price and stock level data attributes have been made private because the application will have to carefully manage the price and stock level of items in the shop. I've given their names two leading underscores to indicate that they are private to their class.

Each class contains properties to provide access to their `price` and `stock_level` attributes. We saw properties in [Chapter 10](#) and used them to store the name, address, and telephone number of a contact. Here we're using properties to provide access to the `price` and `stock_level` attributes of the stock items in the classes. The idea is that these attributes will be set when an

item is created, and we'll create some more methods to manage the price and stock level later, once we have decided how the data will be stored.

I could have made all the other data attributes (for example `stock_ref`, `color`, and `pattern`) private in the same way, but the fashion shop owner and I can't think of a reason why it would be dangerous to have these attributes accessible outside the class.

When I wrote this sample code, I found myself using a lot of block-copy commands in the editor. This is not a good thing.



## PROGRAMMER'S POINT

Block copy is not your friend The IDLE editor will let you select a block of Python statements and copy them into another point in your program. I call this action “block copy.” And it is not your friend.

When writing the code for the Dress and Pants classes, you might think it is efficient programming to just block copy the repeated elements from one class to another. However, this is not a good idea. If you are copying the same code from one part of your program to another, you are not programming most efficiently. A good programmer will try to write a piece of code exactly once. If the code is used more than once in an application, a good programmer will convert the code into a method or function and then call the method each time it's needed.

However, this is not about making sure that our programs are as small as we can make them. It's about self-preservation. If you block copy a piece of code into lots of different places in your application, you'll have a real problem if you find a bug in the copied code. You'll need to go through your entire application and fix all the broken copies of that code. On the other hand, if you find a bug in a method, you can fix it just once, and it is fixed for every situation in which that method is used. Fortunately, there is a way we can remove the need for numerous copies of the same code, which we will discuss now.

If I find myself copying program text from one place to another, I take this as a trigger to step back from the problem and think about different ways of structuring my solution.

## Creating superclasses and subclasses

Python classes support a mechanism called *inheritance*. This is another aspect of object-oriented design. Inheritance lets us base one class on an existing

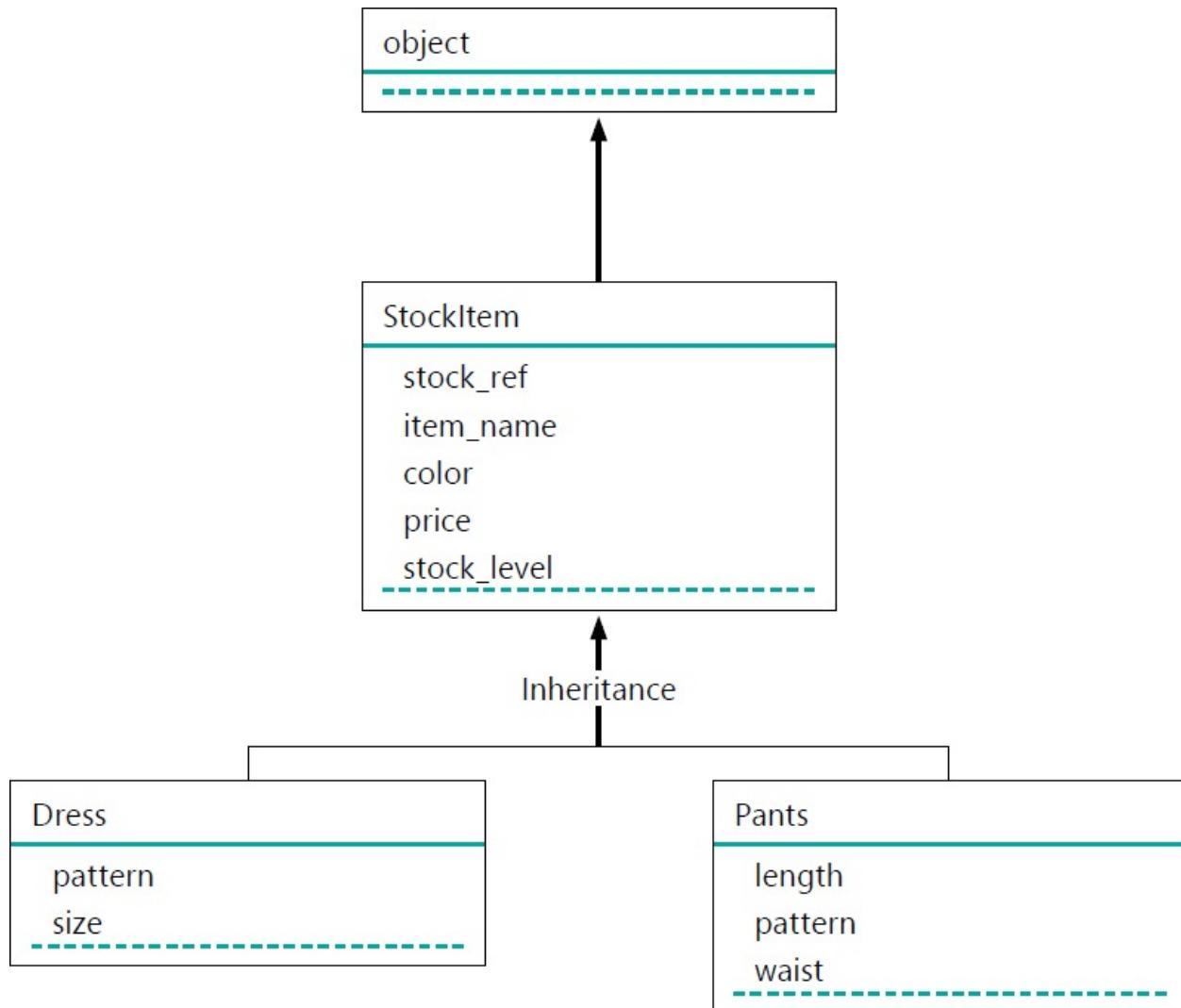
superclass. This is called *extending the superclass*. In fact, we've been doing this already, every time we created a new class in our Python programs. If we don't specify otherwise, all Python classes extend the `object` class, which is the class on which all Python objects are based. The explicit way of stating this is as follows, which we do when we create a new class:

```
class Contact(object):
```

The type in parentheses is the superclass being extended. The above definition for a class called `Contact` is explicitly extending the `object` class. If you leave out the superclass type, Python assumes that you're extending the `object` class.

We can greatly simplify the design of our classes for the Fashion Shop program by creating a superclass, which we can call `StockItem`.

The `StockItem` class will store all the attributes common to all the data items in the shop. These are the stock reference, price, color, and stock level. The `Dress` and `Pants` classes will extend the `StockItem` class and add the attributes particular to dresses and pants. **Figure 11-3** shows the arrangement of the classes we're creating. In software design terms, this is called a *class diagram*.



**Figure 11-3** Fashion Shop class diagram

The class diagram shows the relationship between classes in a system. Figure 11-3 shows that both `Pants` and `Dress` are *subclasses* of the `StockItem` class (meaning they are based on that class). We could also say that the `StockItem` class is the *superclass* of `Dress` and `Pants`.

In real life, inheritance means stuff that you get from people who are older than you. In Python terms, *inheritance* means the attributes a subclass gets from its superclass. Some programmers call the superclass the parent class and the subclass the child class.

The key to understanding inheritance is to focus on the problem we're using it to solve. We're working with a collection of related data items. The related items

have some attributes in common. We want to implement the shared attributes in a superclass and then use this superclass as the basis of subclasses that will hold data specific to their item type. That way, we only need to implement the common attributes once, and any faults in the implementation of those attributes need only be fixed once.

Working in this way has another advantage. If the fashion shop owner decides that she would find it useful to be able to store the manufacturer of the items she's selling, we can add a manufacturer attribute to the `StockItem` class, and all the subclasses will inherit that attribute, too. This will be much easier than adding the attribute to each class.

## Abstraction in software design

Another way to think of this is to consider what we are doing regarding *abstraction*. Abstraction is another term that has a particular meaning when we are talking about object-oriented design. It means “stepping back” from the objects in an application and taking a more general, or abstract, view of them.

In our conversations with the fashion shop owner, we would like to talk in general terms about the things she would like to do with the stock in her shop. She will want to add stock items, sell stock items, find out what stock items she has, and so on. We can talk to her about her stock in general terms and then later go back and fill in the specific details about each type of stock and give them appropriate behaviors.

Programmers use abstraction a lot. They talk about things like stock items, customers, and orders without considering specific details. Later, they can go back and “fill in the details” and decide what particular kinds of stock items, customers, and orders with which the application will work. We’ll create different kinds of stock items in our Fashion Shop program, and the `StockItem` class will contain the fundamental attributes for all the stock, and the subclasses will represent more specific items.

The diagram in [Figure 11-3](#) is called a *class hierarchy*. It shows the superclass at the top and subclasses below. When you travel down a class hierarchy, you should find that you move from the abstract toward the more concrete. The most abstract class in [Figure 11-3](#) is the `object` class, which is the superclass of every object in the Python program. The least abstract classes are `Pants` and

`Dress` because these represent actual physical objects in our application.



## CODE ANALYSIS

Understanding inheritance Here are some questions about object-oriented design and inheritance. Try to come up with your own answers before reading the answers I've provided.

**Question:** Why don't we put all the data attributes in one class and not bother with subclasses?

**Answer:** This is a very good question. Rather than having `Dress` and `Pants` classes, we could add `length`, `pattern`, `size`, and `waist` data attributes to the `StockItem` class and then store everything as an instance of the `StockItem` class. As we find new kinds of stock items, we just need to add new data attributes to describe them.

However, this would be hard to manage. When I wanted to print the details of a pair of pants, the application would need to know to print the length and waist data attributes and not the size. This means that the `StockItem` class would need to hold a "stock type" data attribute and use this to decide what to do when asked to perform actions. This would be difficult to implement and manage.

Later in this chapter, we'll discover a feature of object-oriented design called *polymorphism*, which allows an object to provide behaviors appropriate to its object type. For now, just accept that putting everything in one class would be a bad idea.

**Question:** Why is the superclass called super?

**Answer:** This is another good question and one that has confused

me for a long time. The word *super* usually implies something better, or more powerful. A “superhero” has special powers that ordinary people do not. However, in the case of a superclass, this doesn’t seem to be the case. The superclass has fewer powers (fewer attributes) than the subclass that extends it. The ultimate superclass in Python is the `object` class. This, by definition, has the fewest attributes because everything else adds to it.

I think the word *super* makes sense if you consider it as something from which classes descend. The super object is above the sub object, just like superscript text is above subscript text. The `object` is the superclass because it is above everything else.

**Question:** Which is most abstract, a superclass or a subclass?

**Answer:** If you can work out the answer to this question, you can start to consider yourself an “object-oriented ninja.” Remember that we use abstraction as a way of “stepping back” from the elements in a system. We’ll say “receipt” rather than “cash receipt” or “StockItem” rather than “Pants.”

If you look at the class diagram in [Figure 11-3](#), you will see that the higher up the diagram you go, the more abstract things get, until we reach the most abstract class of all, which is an object. Objects are the superclass of all classes, and also the most abstract (more so than a subclass).

**Question:** Can you extend a subclass?

**Answer:** Yes, you can extend a subclass. In fact, we have already done this. In [Figure 11-3](#), you can see that the `Dress` class extends the `StockItem` class, which itself extends the `object` class. In Python, there is no limit to how many times you can extend classes, although I try to keep my class diagrams fairly shallow, with no more than two or three subclasses.

**Question:** Why is the `pattern` attribute not in the `StockItem` class?

**Answer:** Most impressive. Well spotted. The `pattern` attribute is in both the `Dress` and `Pants` classes. It might seem sensible to move the attribute into the `StockItem` class with the `color`, `stock_level`, and `price` attributes.

The reason I haven't done this is that I think that the fashion shop might sell some stock items that have no pattern—for example, items of jewelry. I want to avoid a class having data attributes that aren't relevant to that item type, so I've put pattern values into the `Dress` and the `Pants` class instead.

I'm not particularly happy with this, in that ideally an attribute should appear only in one class, but in real-world design, you come across these issues quite often. One possible way to resolve the issue would be to create a subclass called "PatternedStock" that is the superclass for `Dress` and `Pants`, but I think that would be too confusing.

**Question:** Will our system ever create a `StockItem` object?

**Answer:** The Python system will allow the creation of a `StockItem` object (an instance of the `StockItem` class), but it's unlikely that we would ever actually create a `StockItem` on its own.

Some programming languages, for example, C++, Java, and C# allow you to specify that a class definition is *abstract*, which stops a program from making instances of that class. In these languages, an abstract class exists solely as the superclass for subclasses. However, Python does not provide this feature.

**Question:** The owner of the fashion shop thinks that one day she might like to keep track of which customer has bought which item of stock. That way she can look at their past purchases and make recommendations for future purchases. Here are three ways to do this. Which would make the most sense?

1. Extend the `StockItem` class to make a `Customer` subclass that contains the customer details because customers buy

`StockItems`.

2. Add `Customer` details to each `StockItem`.
3. Create a new `Customer` class that contains a list of the `StockItems` that the `Customer` has bought.

**Answer:** Option 1 is a bad idea because a class hierarchy should hold items that are in the same “family.” In other words, they should all be different versions of the same fundamental type. We can see that there is some association between a `Customer` and a `StockItem`, but making a `Customer` a subclass of `StockItem` is a bad idea because they’re different kinds of objects. The `StockItem` holds attributes such as `price` and `stock_level`, which are meaningless when applied to a `Customer`.

Option 2 is a bad idea because several customers might buy the same `StockItem`. The customer details cannot be stored inside the `StockItem`.

Option 3, adding a new `Customer` class, is the best way to do this. Remember that because objects in Python are managed by references, the list of clothing items in the `Customer` class (the items the customers have bought) will just be a list of references, not copies of `StockItem` information.

## Store data in a classes hierarchy

Now that we’ve decided using inheritance is a good idea, we need to consider how to make it work with our classes.

```

class StockItem(object):
 """
 Stock item for the fashion shop
 """

 def __init__(self, stock_ref, price, color):
 self.stock_ref = stock_ref
 self.__price = price
 self.color = color
 self.__stock_level = 0

 @property
 def price(self):
 return self.__price

 @property
 def stock_level(self):
 return self.__stock_level

```

**StockItem class explicitly extends the object class**

**Initializer for the StockItem class**

**Initial stock level for the item is zero**

**Price property**

**Stock level property**

This is the **StockItem** class file. It contains an **\_\_init\_\_** method to set up an instance of the class. The **StockItem** class will be the superclass of all the objects that the fashion shop will be selling. The **StockItem** class extends the **object** class. We can create a **Dress** class that is a subclass of the **StockItem** class to hold information about dresses that the fashion shop will be selling.

```

EG11-02 Stock Item class failed

class Dress(StockItem):
 def __init__(self, price, color, pattern, size):
 self.pattern = pattern
 self.size = size

```

**Dress is a subclass of StockItem**

The **Dress** class is a subclass of the **StockItem** class, which we indicated by providing the superclass name when we declared the class. However, we have a problem if we try to use this version of the **Dress** class: [Click here to view code image](#)

```
x = Dress(stock_ref='D0001', price=100,
```

```
color='red', pattern='swirly', size=12)
```

The statement above creates an instance of the `Dress` class with the name `x`. This statement will not generate any errors when it runs, but we will have problems if we try to use some of the properties of the `Dress` object created by the statement.

```
print(x.pattern)
swirly
print(x.price)
Traceback (most recent call last):
 File "<pyshell#103>", line 1, in <module>
 print(x.price)
 File "C:/Users/Rob/ EG11-02 Stock Item class failed.py", line 16, in price
 return self._price
AttributeError: 'Dress' object has no attribute '_StockItem_price'
```

Pattern prints correctly  
Price does not print

The `pattern` attribute prints correctly, but if I try to print the `price` property of the `Dress` instance, the program fails with an error saying that there is no `price` attribute. In fact, the message says that there is no `_StockItem_price` attribute. If you think about it, this is a reasonable error.

The `__init__` method in the `Dress` has set up the `pattern` and `size` attributes in the `Dress`, but nothing has set up the `stock_level`, `price`, `stock_level` and `color` data attributes in the `StockItem` on which the `Dress` is based.

If we want the `StockItem` part of the `Dress` object to have `stock_level`, `price`, `stock_level`, and `color` data attributes, we need to call the `__init__` method for the `StockItem` class and pass these values into it.

```
EG11-03 Stock Item class super init

class Dress(StockItem):
 def __init__(self, stock_ref, price, color, pattern, size):
 super().__init__(stock_ref=stock_ref, price=price, color=color)
 self.pattern = pattern
 self.size = size
```

Dress is a subclass of StockItem  
Call the \_\_init\_\_ method in the super object

This is a version of the `Dress` `__init__` method that calls the `__init__` method in the `StockItem` class. Python provides a function called `super()` that can be used to obtain a reference to the superclass in an object. The `super` method returns a reference to the super object. We can then call the `__init__` method on that reference, feeding in the price and color values. If this seems confusing, we can break this single statement down into two:

```
super_object = super() Get a reference to the super object
super_object.__init__(stock_ref=stock_ref, price=price, color=color) Call the
__init__
method in the
super object
```

The first statement gets a reference to the `super` object. The second statement calls the `__init__` method on the object to which the reference refers. The parameter values of `stock_ref`, `price`, and `color` are passed into this call, and used to set up the attributes in the `StockItem`. This looks a little confusing because we're using keyword arguments in the call to the `__init__` method. It looks like we're doing things like setting price to price (`price=price`), whereas we are really copying the price value received as a parameter into the price argument we're sending to the `__init__` method.

When you create a subclass, you have to make sure that the initialization process of the subclass explicitly initializes the superclass as well.



## WHAT COULD GO WRONG

### The super function in Python 2.7

The use of the `super` function will work in Python 3.6, but in Python 2.7 it will generate an error. The `super` function in Python 2.7 needs to be provided with two arguments:

- The class in which the `super` function is running
- A reference to the object being initialized

[Click here to view code image](#)

```
super(Dress, self).__init__(stock_ref, price,
color)
```

This format will work in Python 3.6 as well as Python 2.7.

## Manage the item name in the Fashion Shop program

We've created classes to represent the stock items in the fashion shop. The name of each class matches the type of stock being stored. However, the fashion shop owner might not want to give her stock items names that match Python class names. We would not be able to create a class called "Evening Dress" because the string contains a space and is therefore not a valid Python name. The fashion shop owner will not want to see an item described as an "Evening\_Dress" just because that is a valid Python identifier, so we must devise a way of providing a "friendly name" for each of the classes in our hierarchy.

It turns out that I've already thought of this. If you look carefully at the class diagram in [Figure 11-3](#), you'll see that the `StockItem` class contains something called `item_name`. This is intended to hold the name of this type of object. It will contain a string that provides a "friendly" name for the item of that type of data. The best way to provide this information is as a property of the class.

[Click here to view code image](#)

```
class StockItem(object):

 @property
 def item_name(self):
 return 'Stock Item'
```

The code above shows the `item_name` property in the `StockItem` class. It looks like the `price` and `stock_level` properties, except that it returns the string '`Stock Item`'. Each of the child classes can override this property to

return their name: [Click here to view code image](#)

```
class Dress(StockItem):

 @property
 def item_name(self):
 return 'Dress'
```

Remember, if you override an attribute in a subclass, that attribute is used instead of the superclass attribute. An attempt to get the `item_name` property of a `Dress` object would result in the `item_name` code for `Dress` being obeyed, and the `Dress` string being returned.

## Add an `__str__` method to classes

Objects can produce a string description of themselves. The class describing an object can contain an `__str__` method that is called to return a string description of the contents of the object. We added `__str__` methods to the `Contact` and `Session` classes to make it easy to view the contents of these objects. Now, we'll add `__str__` methods to the objects in our Fashion Shop application, which will allow us to view the contents of those objects.

Adding an `__str__` method to a class replaces the method in the object on which the class is based. Programmers talk about *overriding* a method in a superclass. Let's investigate how this works.



MAKE SOMETHING HAPPEN

Method overriding in classes Overriding means, “superceding a method in a superclass with one in the subclass.” We can find out how it works by using the IDLE Command Shell. Open it and enter

this statement:

```
>>> o = object()
```

This statement creates an `object` instance referred to by the variable `o`. We've created `Contact` and `Session` objects in previous chapters, but we can also create instances of the `object` class if we wish. Now we can print the value in the object referred to by `o`. Type in the call of the `print` function below and press Enter.

```
>>> print(o)
```

The `print` function uses the method `str` to convert an object to a string before printing it. The `str` function calls the `__str__` method on an object to return a description of the contents of the object. So, this statement will show us what the `__str__` method in the `object` class returns.

[Click here to view code image](#)

```
>>> print(o)
<object object at 0x0000020B57A59070>
```

The `__str__` method in the `object` class returns a string that indicates, “This is an object of type `object`” and gives the object’s address stored in memory. We’ve seen this behavior before when we tried to print the contents of a `Contact`: [Click here to view code image](#)

```
<__main__.Contact object at 0x0000018E5E9EBB70>
```

We saw this because the `Contact` value was using the `__str__` method of its superclass, which is the `object` class. Now let’s create a class of our own that overrides the `__str__` method. Enter the following definition of a class called `StrTest`. Remember to enter a blank line at the end of the definition to mark the end of the class.

[Click here to view code image](#)

```
>>> class StrTest(object):
 def __str__(self):
 return 'string from StrTest'
```

```
>>>
```

The `StrTest` class contains an `__str__` method that returns the string `string from StrTest`. This `__str__` method *overrides* the `__str__` method in the object superclass. Attempts to get the string representation of an `StrTest` object will use the `__str__` method in `StrTest`. Prove this by creating an `StrTest` instance and then printing it.

[Click here to view code image](#)

```
>>> t1 = StrTest()
>>> print(t1)
string from StrTest
```

When the `print` method gets the string description of an `StrTest` object, the `string from StrTest` string is returned (just like with `Contact` and `Session` classes). Now, let's add another class.

[Click here to view code image](#)

```
>>> class StrTestSub(StrTest):
 def __str__(self):
 return super().__str__() + '..with
sub'
```

```
>>>
```

This class is a subclass of `StrTest`. In other words, it extends the superclass. It contains a `__str__` method that overrides the `__str__` method in the `StrTest` superclass. However, this method doesn't return a fixed string. Instead, it returns the result of the `__str__` method in the superclass, as well as some extra text. We can see this behavior if we make an instance of the new class and print it: [Click here to view code image](#)

```
>>> t2 = StrTestSub()
>>> print(t2)
string from StrTest..with sub
```

The first part of the printed string was generated by the `__str__` method in the `StrTest` class. The second part of the string was generated by the `__str__` method in the `StrTestSubclass`, which illustrates an important aspect of method overriding: an overriding method in a subclass can call the method that it is overriding.

Now that we know how to override the `__str__` method, we can provide these methods for the `StockItem`, `Dress`, and `Pants` classes.

The diagram shows the `__str__` method for the `StockItem` class. The code is as follows:

```
class StockItem(object):
 def __str__(self):
 template = '''Stock Reference: {0}
Type: {1}
Price: {2}
Stock Level: {3}
Color: {4}'''
 return template.format(self.stock_ref, self.item_name,
 self.price, self.stock_level, self.color)
```

Annotations explain the steps:

- `StockItem` based on the `object` class
- Definition of the `__str__` method for `StockItem`
- Create a string template
- Insert the values into the template

Above, you can see the `__str__` method for the `StockItem` class. The `__str__` method for the `StockItem` class looks remarkably like the `__str__` methods in the `Contact` and `Session` classes that we created in [Chapter 10](#). It creates a template string and then fills this in with data attributes and properties from the object.

You might be wondering why we are making a `__str__` method for the `StockItem` class when this class is the superclass of the other classes in our system. The program will create instances of classes, such as `Dress`, but will not create `StockItem` objects. Why would we give this class a `__str__` method?

The answer lies in something we discovered in the previous “Make Something Happen.” A method that overrides another can still call the overridden method. The best way for a subclass of `StockItem` to get a string describing the contents of a `StockItem` is for the `__str__` method in the subclass to use the `__str__` method in the superclass.

```
class Dress(StockItem):
 def __str__(self):
 stock_details = super().__str__()
 template = '{0}
Pattern: {1}
Size: {2}'''
 return template.format(stock_details, self.pattern, self.size)

Dress based on the StockItem class
__str__ method for Dress
Get the StockItem description text
Template location for StockItem description
```

We've used the `super()` function before to locate the super object of a clothing class so that we could call the `__init__` method in that object to set it up. Here, we're using the `super()` function to locate the super object so that we can ask it for a string description of its contents. Then, we can add the resulting string to the description for this class.

[Click here to view code image](#)

```
EG11-04 Stock Items with str

x = Dress(stock_ref='D001', price=100, color='red',
 pattern='swirly', size=12)
print(x)
Stock Reference: D001
Price: 100
Stock level: 0
Color: red
```

Pattern: swirly

Size: 12

Above, you can see the results of creating a `Dress` instance and printing it. The first three lines of the description string are produced by the `StockItem` class; the last two lines are added by the `__str__` method in the `Dress` class.



## CODE ANALYSIS

Understanding method overriding Here are some questions about method overriding that you might be pondering.

**Question:** How does method overriding work?

**Answer:** When Python wants to call a method on an object, it looks first at the object itself to see if it has a method with the specified name. If the method isn't found, Python searches the superclass to find that method. If the method is found in the superclass, Python runs the method. If the method still isn't found, Python will look in the superclass above that class, and so on, until either the method is found or Python runs out of superclasses to search, and an `AttributeError` is raised.

We can get a description for our original `Dress` class (which didn't contain a `__str__` method) because Python finds a `__str__` method in the super `object` class.

**Question:** Is an overriding method forced to call the method it is overriding?

**Answer:** No. It should only do this if there is an advantage in doing so. The `__str__` method in the `Dress` class should call the

`__str__` method in the `super` object. If I change the data attributes in the `StockItem` class, I can change the behavior of the `__str__` method for `StockItem`. Then, all the classes based on it receive an updated description.

## Version management in the Fashion Shop program

When we created the `Contact` and `Session` classes for the Time Tracker application in [Chapter 10](#), we spent some time ensuring that changes to these classes did not prevent the application from working with older data files. Each time an object is loaded into the Time Tracker application, the version number of the object is checked, and if required the object is updated to the latest version. We need to consider how versions of classes will be managed in the Fashion Shop application, too.

The first question we need to consider is, “Where should the version numbers be stored if we’re using a class hierarchy?” In the Time Tracker application, the `Session` and `Contact` classes both contained a version number attribute. In the Fashion Shop application, the `Dress` class is a subclass of the `StockItem` class. We put the version number in the `StockItem` class and in the `Dress` class. Each class will have a `check_version` method. The `check_version` method in the `Dress` class will use `super` to call the `check_version` in the super object, which will ensure that the `StockItem` class is kept up to date.

```

class StockItem(object):
 ...
 Stock item for the fashion shop
 ...

 def __init__(self, stock_ref, price, color): Initialize the StockItem value
 self.stock_ref = stock_ref
 self.__price = price
 self.__stock_level = 0
 self.__StockItem_version = 1 Store the version number in a private
 self.color = color variable in the StockItem class

 def check_version(self): Called to check the version of the StockItem
 # This is version 1 - no need to update anything
 pass

```

These are the `__init__` and `check_version` methods for the `StockItem` class. Because this is version 1 of the `StockItem`, the `check_version` method doesn't need to check for upgrades because there are none.

```

EG11-05 Stock Items with version

class Dress(StockItem):

 def __init__(self, stock_ref, price, color, pattern, size):
 super().__init__(stock_ref, price, color)
 self.pattern = pattern
 self.size = size
 self.__Dress_version = 1

 def check_version(self): Called to check the version of the Dress
 # This is version 1 - no need to update anything
 super().check_version() Check the version of the superclass StockItem
 pass

```

These are the `__init__` and `check_version` methods for the `Dress` class. They manage version numbers in a comparable way, except that the `check_version` method in the `Dress` class contains a call of the `check_version` method in the parent `StockItem` class. We can treat the superclass and the subclass as separate regarding version control.

## Polymorphism in software design

The next thing I want to talk about has the most impressive name in the entire book. The word *polymorphism* comes from the Greek language and means “the condition of occurring in multiple forms.”

Overriding the `__str__` method in a class is a great example of polymorphism at work. All objects in a Python program can be asked to provide a text description of their contents. This works because each object has an `__str__` method that provides details of the contents of that class. In other words, when Python says to an object, “Give me a string version of yourself” the object can be made to do the right thing, whatever type of object it is.

An application may contain many `__str__` methods in different classes, each of which will provide a string self-description for the object to which it applies. The `__str__` in a `Contact` will deliver the description of a `Contact`, whereas the `__str__` in a `Dress` will describe a dress.

Software frequently uses polymorphism. Many programs include a Play button, which starts playback of content, such as music, video, or a slide show, which is another good example of how a program can be polymorphic.

Polymorphism is a powerful design tool, and it goes hand in hand with abstraction. We’ve seen that abstraction means “stepping back from a system” so that we can talk in general terms about an account rather than specifically a credit card account or a savings account. We can create a method (perhaps called `withdraw_funds`) in our superclass to denote a need for an action, such as withdrawing money from an account. Each subclass that extends the superclass can override that method with the behavior that works for that subclass type.



### CODE ANALYSIS

Understanding polymorphism Here are some questions about polymorphism. Try to answer

these questions yourself before reading the answers that I've provided.

**Question:** Is polymorphism all about providing methods in a class hierarchy?

**Answer:** This is a very good question. The answer is no, but this requires some explanation. In our examples from this chapter, we have a class hierarchy that has a superclass called `StockItem`. We have identified an action—check what version of data you contain and upgrade it if necessary—that all objects in the hierarchy need to perform. We've assigned the action to a method (`check_version`), and we've added `check_version` methods to all the objects in our class hierarchy. We can ask any of the subobjects of `StockItem` to check its version. The result depends on the object itself.

However, we can add a `check_version` method to any class in an application; the `check_version` method doesn't have to be in the `StockItem` class hierarchy. If the Fashion Shop application starts storing customer records using a class called `Customer`, it would make good sense to add a `check_version` method to that class as well. The `check_version` method can be used in any object it has loaded.

Therefore, polymorphism can be used across an entire range of classes.

**Question:** How do I know which methods in my application should be polymorphic?

**Answer:** During the design phase, you'll identify behaviors that will be performed by different objects but must work differently for each object. For example, you might be writing a video game that includes many types of attacking aliens. You might decide to create a class hierarchy for each type of alien even though each type will share behaviors, such as “attack” and “take damage.” The actions taken by each type of alien will differ depending on the alien, but it

is very useful for the game to be able to regard aliens purely regarding their ability to attack and take damage.

## Protect data in a class hierarchy

In Chapter 10, we developed the `Contact` and `Session` classes in the Time Tracker application. We decided that some attributes of these classes would benefit from special protection within the application. For example, we made the `__hours_worked` value in the `Contact` class private to that class. It is worth considering how the privacy attribute is managed in a class hierarchy.

```
class StockItem(object):
 ...
 Stock item for the fashion shop
 ...

 def __init__(self, stock_level, price, color):
 self.stock_level = stock_level
 self.__price = price
 self.__stock_level = 0
 self.color = color
```

Stock reference has been made public  
Price has been made private  
Stock level has been made private  
Color is public

This is part of the `__init__` method for the `StockItem` class. In Python, a class attribute with a name beginning with two underscores is regarded as private to the class where it was declared. In other words, the `__price` and `__stock_level` attributes created in the `StockItem` class can only be used by methods in the `StockItem` class.

However, the `color` attribute has been made public (there are no leading underscore characters), so that subclasses can use this value. When you design a class hierarchy, you need to think about how data will be used in classes at each level. I've decided that it is important that the price and stock level of items be given some protection so that we can control how they are changed. However, an item's color is less important.

## Data design recap

At this point, let's recap our Fashion Shop application development: Your customer is the owner of a fashion shop that sells several types of clothing.

She wants you to create an application to manage information about the dresses, pants, blouses, and hats she sells.

All clothing items have a stock reference, price, stock level (how many of that item are in the shop), and color attributes.

Also, each clothing item has a specific set of attributes (for example, dresses have size and pattern attributes, while pants have pattern, waist, and length attributes).

To help us avoid having to duplicate the code that manages price, color, and stock level attributes (which are common to all the stock items), we have created a `StockItem` class to manage these attributes.

The `Dress`, `Pants`, `Hat` and `Blouse` classes are subclasses of `StockItem`, which means the subclasses inherit the price, color, and stock level attributes from `StockItem` and can then add their own attributes to these.

Each of the classes we've created has an `__init__` method that is called to initialize instances of that object. The `__init__` methods in the subclasses contain a call to the `__init__` method of the `StockItem` class, which sets that up in the “super” object. These methods use the function `super()`, which returns a reference to the super object. Once the `__init__` method in a sub class has this reference it can call `__init__` on this reference to set up the super object.

We've added a `check_version` method to all the classes in the Fashion Shop application, as we did for the `Contact` and `Session` classes in the Time Tracker application in [Chapter 10](#).



## CODE ANALYSIS

## Data design

Here are some questions about our data design. Try to answer these questions yourself before reading the answers that I've provided.

**Question:** Is the data design now complete?

**Answer:** No. However, I'm not too worried about this. In addition to data design, we've also developed a strategy (using the `check_version` methods) that will make it very easy for us to add new attributes to our data storage design, even after we've released the first version of our application. If we find that some classes need to be modified, we have a framework for doing so.

**Question:** What happens if the fashion shop owner decides to sell a new kind of stock item? Suppose that she decides to start stocking Jeans. Jeans are managed in the same way as `pants`, except that they have a `style` property that can be `flared`, `bootleg`, or `straight`. What's the best way to do this?

**Answer:** The good news is that we can do this by just adding a new subclass to the hierarchy. It would seem sensible to extend the `Pants` class to make a new subclass that just holds the extra information about `Jeans`.

```

class Jeans(Pants):
 Extend the Pants class

 def __init__(self, stock_level, price, color, pattern, length, waist, style):
 Initialize the parent Pants
 super().__init__(stock_level, price, color, pattern, length, waist)
 self.style = style
 Set the style for the Jeans
 self.Jeans_version = 1
 Add version management

 @property
 Property to return the name of the item
 def item_name(self):
 return 'Jeans'
 Return "Jeans" as the item name

 def check_version(self):
 Version management for Jeans
 # This is version 1 - no need to update anything
 super().check_version()
 Check the version of the parent Pants
 pass

 def __str__(self):
 Get string description of some Jeans
 pants_details = super().__str__()
 Get the parent description string
 template = '{0}'
 Create the template
 Style: {1}'''
 return template.format(pants_details, self.style)
 Return the string

```

**Question:** What happens if the fashion shop owner decides to store something new about her stock? Suppose she decides to add a “location” attribute to her stock items. The location would be a string, such as “Hanging rail at the front of the shop.” When she adds stock to the system, she would like to enter the location string for that item. In the future, she would like to create a “fashion assistant” application that will allow customers to search for items they might like to buy. When the application suggests items to wear, it can also tell the customer where they are located in the shop. How would we add this attribute, and to which class would it be added?

**Answer:** Because all items in the shop would need this location property, we could add it to the `StockItem` class `__init__` method so that it is set whenever a stock item is created.

```
class StockItem(object):
 """
 Stock item for the fashion shop
 """

 def __init__(self, stock_ref, price, color, location):— Added location parameter
to __init__ method
 self.stock_ref = stock_ref
 self.__price = price
 self.__stock_level = 0
 self.color = color
 self.location = location Set the location attribute
```

This is part of the updated `__init__` method for the `StockItem` class, which adds a new attribute called `location`. We can now specify the location of an item of stock when we create it. The problem with this change is that it breaks our application: [Click here to view code image](#)

```
Traceback (most recent call last):
 File "C:/Users/Rob/FashionShop.py", line
198, in <module>
 new_item = get_new_item()
 File "C:/Users/Rob//FashionShop.py ", line
165, in get_new_item
 pattern=pattern, size=size)
 File "C:/Users/Rob//FashionShop.py ", line
42, in __init__
 super().__init__(price, color)
TypeError: __init__() missing 1 required
positional argument: 'location'
```

This is what happens if we try to create a new `Dress` object after we've modified the `__init__` method for the `StockItem` class. The problem is that the `__init__` method for the `Dress` class doesn't supply a location value when it calls the `__init__` method for its `StockItem` super object. I now must modify the `Dress` class to add the location argument to the call of `__init__` in the super object:

```
class Dress(StockItem):
 def __init__(self, stock_ref, price, color, location, pattern, size):
 super().__init__(stock_ref, price, color, location)
 self.pattern = pattern
 self.size = size
 self.Dress_version = 1
```

`__init__` for `Dress` now contains location  
Add the location value to the `__init__` of the parent object

This is the modified `__init__` method for the `Dress` class. It now accepts a location string and passes this string into the call to `__init__` for the superclass. Next, we must go through all the subclasses (`Pants`, `Hat`, and `Blouse`) in the application and change their `__init__` methods, too.

Class hierarchies are very *brittle*. In other words, they're easy to break. Changes to classes at the top of the hierarchy may result in you having to modify code in many subclasses. Therefore I strongly encourage programmers to design their classes on paper before they try to write any program code. Otherwise, a lot of time can be wasted updating classes to reflect changes to the design.

In the case of the location information for a stock item, perhaps a better solution would be to add a `location` property to the `StockItem`. We've seen properties before. We used properties to manage the name, address, and telephone number attributes of the `Contact` class.

```
class StockItem(object):
 @property
 def location(self):
 result = getattr(self, '_location', None)
 return result
 @location.setter
 def location(self, location):
 self._location = location
```

Get method for the location property  
Get the attribute, or return `None` if location has not been set  
Set method for the location property

The code above shows how the `location` property can be added to a `StockItem`. The first method gets the `location` property; the second method sets it. The first method is interesting because it uses a Python function that we haven't seen before.

[Click here to view code image](#)

```
result = getattr(self, '_location', None)
```

The `getattr()` function is provided with three arguments. The first is a reference to an object. In this case, the reference is `self`, because we want to get the argument out of the object that is running the method that's reading the property. The second argument to the call of `getattr()` is a string containing the name of the attribute for which we want to get the value. In this case, we want to get the value of the `_location` attribute. The third argument is a value to be returned if the attribute is not present in the object. This would be the case if the location attribute had not been set. If we try to get the location of a `StockItem` that has no location attribute, the property reading code above will return the value `None`. We've seen the value `None` before in [Chapter 7](#) (see “[Code Analysis: Functions and Return](#)”). We saw that `None` is a special value used by Python to indicate that a variable is not set to a useful value. A program reading the location attribute can test for this value and behave sensibly if it has not been set.

We use the `getattr` function to read the `_location` attribute because the property must deal with the situation in which the `_location` attribute has not been defined. If a program just tries to use the property, it would raise an `AttributeError`, which will stop the program.

We have seen two ways you can add location information to an object in the Fashion Shop. A third way would be to just add the `location` attribute to a particular object when we want to store the location of that object.

[Click here to view code image](#)

```
new_dress = Dress(price=100, color='red',
pattern='swirly', size=12)
new_dress.location = 'Front of shop'
```

The statements above create a new `dress` instance and then set the location of that `dress` object to `Front of shop`. This will work for any object

because a Python program can add a new attribute to an object at any time. We can then use a Python function called `hasattr` to test whether an object has a location attribute: The `hasattr()` function takes two arguments. The first argument is a reference to an object, and the second argument is the name of the attribute you’re looking for. The function returns `True` if the attribute is present in the object. The code below shows how this would work. It only displays the location of a stock item if that item has a location property.

[Click here to view code image](#)

```
if hasattr(new_dress, 'location'):
 print('The dress is located: ',
new_dress.location)
else:
 print('The dress does not have location
information')
```

We’ve come up with three ways of adding a `location` attribute to a `StockItem` class. The first way involves creating the `location` attribute in the `__init__` method that’s called when a `StockItem` is created. The second involves adding a `location` property to the `StockItem` class. The third is the simplest, just adding a new `Location` attribute to the class.

Setting a `location` attribute of a stock item in the `__init__` method guarantees that location information is stored for all objects. Adding a `location` property to a stock item manages the setting of the `location` attribute and provides a sensible behavior if the location has not been set. The third approach—just adding the `location` attribute to an object—is extremely open. It provides a way that location information can be added to any object at any time.

I find the third approach tempting because it is very easy to write, but if I were creating a “professional” application (that is, one that I intended to sell), I would not use it. This is because adding attributes in numerous places in an object can make an application hard to work with. Another programmer reading the program source code would have to look through the entire program to discover

where the location information is added to a stock item. If I set everything up in the `__init__` method, this provides a single place to look for attribute setups.

As an analogy, I try to keep all my tools in my toolbox, so that if I need my screwdriver, I can just go to the toolbox and get it. However, as I do jobs around the house, my tools tend to get spread all over the place. It takes an effort of will for me to put my tools back in the box when I've finished with them, but it greatly speeds up the next job. You can think of the `__init__` method as a "toolbox" in which all the settings are entered into a new object. Using the `__init__` method may mean extra work, particularly if we add attributes to the superclass, but it makes it much easier for anyone looking at the classes to discover what attributes they contain.



### MAKE SOMETHING HAPPEN

Instrumented stock items In [Chapter 6](#), we saw how to use the IDLE debugger to execute a Python program one statement at a time and observe the sequence in which the statements are performed. Another way to find out what a program does is to add some instrumentation to the code, which is quite simple to do. The instrumentation we will add is print statements so that show what happens when the program runs. I've created a special set of Fashion Shop classes that contain `print` statements at the beginning of all methods inside the class.

```
class StockItem(object):
 """
 Stock item for the fashion shop
 """

 show_instrumentation = True Set to False to turn off the instrumentation output

 def __init__(self, stock_ref, price, color, location):
 if StockItem.show_instrumentation:
 print('**StockItem __init__ called') Instrumentation print statement
 self.stock_ref = stock_ref
 self.__price = price
 self.__stock_level = 0
 self.StockItem_version = 1
 self.color = color
 self.location = location
```

Above, you can see the `__init__` method for the `StockItem` class in my instrumented version of the Fashion Shop application. It contains a `print` statement as the first line of the method, as do all other methods in the `StockItem` class and every other class in the program. All the instrumentation messages begin with the string `**`, so we can distinguish them from any prints from the program itself. The instrumentation is printed only if the value of the `StockItem show_instrumentation` attribute is `True`.

We can work with the instrumented classes and find out how they work by using the IDLE Command Shell. **Open IDLE** and then use **File, Open** to open the file **EG11-06 Instrumented Stock Items** in the IDLE program editor. If we run this program, it will set up all the classes for us to experiment with in the command shell. Run the program by selecting **Run, Run Module** from the editor. The program will run, create the fashion shop classes, print a message, and then return the Python prompt.

[Click here to view code image](#)

```
RESTART: C:/Users/Rob/EG11-06 Instrumented
Stock Items.py
Instrumented classes ready for use
>>>
```

We can now issue Python statements that create and work with Fashion Shop classes. **Note that you must have run the example program before trying to create objects.** Type in the following statement: [Click here to view code image](#)

```
>>> new_dress = Dress('D001', 100, 'red',
'swirly', 12, 'shop window')
```

When you press Enter, the `Dress` is constructed.

[Click here to view code image](#)

```
>>> new_dress = Dress('D001',100, 'red',
'swirly', 12, 'shop window')
** Dress __init__ called
** StockItem __init__ called
>>>
```

The instrumentation in the class methods show that the `__init__` method in `Dress` is called, which then calls the `__init__` method in the `StockItem`. Now, let's construct some jeans.

[Click here to view code image](#)

```
>>> new_jeans = Jeans('J1',50, 'blue',
'plain', 30, 30, 'flared', 'shop window')
** Jeans __init__ called
** Pants __init__ called
** StockItem __init__ called
>>>
```

Constructing jeans is slightly more complicated than dresses. The `Jeans` class extends the `Pants` class, so to make a `Jeans` instance, we first make some `Pants`. The `__init__` method in the `Pants` class then calls the `__init__` method in the `StockItem` class.

Now, let's see what happens when we print a pair of jeans: [Click here to view code image](#)

```
>>> print(new_jeans)
** Jeans __str__ called
** Pants __str__ called
** StockItem __str__ called
** Jeans get item_name called
** StockItem get price called
** StockItem get stock_level called
Stock Reference: J1
Price: 50
Stock level: 0
Color: blue
Location: shop window
Pattern: plain
Length: 30
Waist: 30
Style: flared
>>>
```

This set of messages shows how the `__str__` methods in each of the classes in the hierarchy are called to build up the message that's printed. The instrumentation also shows when the `price` and `stock_level` properties in the `StockItem` class were accessed.

We can see how our `check_version` methods have been chained together so that an application can check the version of all the elements that make up an object: [Click here to view code image](#)

```
>>> new_dress.check_version()
** Dress check_version called
** StockItem check_version called
>>>
```

The code above shows how the `Dress` part of the object is checked

first followed by the `StockItem` part.

Finally, we can investigate how the instrumentation itself works.

[Click here to view code image](#)

```
>>> StockItem.show_instrumentation = False
>>> print(new_jeans)
Stock Reference: J1
Price: 50
Stock level: 0
Color: blue
Location: shop window
Pattern: plain
Length: 30
Waist: 30
Style: flared
>>>
```

If the `StockItem.show_instrumentation` flag is set to `False`, the instrumentation is turned off. Code instrumentation is very useful in situations in which it would be very difficult to use an interactive debugger to step through a program. The problem with code instrumentation is that you must add the print statements to the methods. If we were concerned about the performance of our program, we could add additional instrumentation that could give timing information.

Also, you can use *logging* to determine what a program is doing. Logging is like instrumentation, but the details of which methods are called are stored in a file that you can view, which is very useful if your programs are running on a server in another location.

## Implement application behaviors

This is the main menu for the Fashion Shop application.

[Click here to view code image](#)

## Mary's Fashion Shop

- 1: Create new stock item
- 2: Add stock to existing item
- 3: Sell stock
- 4: Stock report
- 5: Exit

Enter your command:

We have used this style of menu several times. The user enters a number to select the command he wants to perform. Now, we must create the stock item behaviors for each item.

## Create new stock item

You have created the classes that will represent the items in the shop. Now, you need to work out how to create the stock item records. In the manual version of the Fashion Shop application, this was performed by adding a new page to the file that holds all the stock items and then filling in all the details of the new stock item being created. The application needs to read in all the details for the new stock item and then create an object of the required type. The code sample below shows how we can do this.

```
EG11-07 Creating Stock Items
```

```
menu = '''
Create new stock item
```

```
1: Dress
2: Pants
3: Hat
4: Blouse
5: Jeans
```

```
What kind of item do you want to add: '''
```

Menu string

```
item = read_int_ranged(prompt=menu, min_value=1, max_value=5)
```

Select the item to add

```
if item == 1:
```

```
 print('Creating a Dress')
 stock_ref = read_text('Enter stock reference: ')
 price = read_float_ranged(prompt='Enter price: ',
 min_value=StockItem.min_price,
 max_value=StockItem.max_price)
 color = read_text('Enter color: ')
 location = read_text('Enter location: ')
 pattern = read_text('Enter pattern: ')
 size = read_text('Enter size: ')
 stock_item = Dress(stock_ref=stock_ref,
```

Adding a dress

Create the dress instance

```
 price=price,
 color=color,
 location=location,

 pattern=pattern,
 size=size)
```

```
elif item == 2:
```

```
 print('Creating a pair of Pants')
 stock_ref = read_text('Enter stock reference: ')
 price = read_float_ranged(prompt='Enter price: ',
 min_value=StockItem.min_price,
 max_value=StockItem.max_price)
 color = read_text('Enter color: ')
 location = read_text('Enter location: ')
 pattern = read_text('Enter pattern: ')
 length = read_text('Enter length: ')
 waist = read_text('Enter waist: ')
 stock_item = Pants(stock_ref=stock_ref,
 price=price,
 color=color,
 location=location,
 pattern=pattern,
 length=length,
 waist=waist)
```

```
print(stock_item)
```

This code allows the user to select the stock item she wants to create, reads in the attributes for that item, creates an instance of that type, and then prints that item. The code sample above only creates `Dress` and `Pants` objects; the sample in the file **EG11-07 Creating Stock Items** contains code for all the different types of clothing objects. You can run this example, select the type of item to create, and the item will be created and then printed. Note that this code will not be part of any stock item class; instead, it will be part of the user interface class `FashionShopShellApplication`, which we'll discuss later.

## Add stock to an existing item

When items arrive at the shop from the suppliers, the fashion shop owner must add their details to the stock records. In the paper-based system, the owner would have to find the page for that particular stock item and then update the records on that page in her binder. [Figure 11-2](#) shows a handwritten entry indicating that stock has arrived. The `StockItem` class contains an attribute called `__stock_level`. This is a private attribute—indicated by the two underscore characters in the name—which is set to zero when an instance of a stock item is created. We can add a method to the `StockItem` class that will allow this to be updated:

```

EG11-08 Adding to stock levels

class StockItem(object):
 ...
 Stock item for the fashion shop
 ...

 max_stock_add = 10 Maximum number that can be added

 def add_stock(self, count): Add items to stock
 ...
 Adds stock for this item.
 count gives the amount of stock to add
 Raises an exception if the amount is invalid
 ...
 if count < 0 or count > StockItem.max_stock_add: Check that the amount to be added is valid
 raise Exception('Invalid add amount') Raise an exception if invalid

 self.__stock_level = self.__stock_level + count Update the stock level with
 the count

```

The `add_stock` method manages the `__stock_level` and adds to it. The method parameter `count` provides the number of items to add to the stock level for this item.

Note that I've added an attribute called `StockItem.max_stock_add` to the `StockItem` class. The value of this attribute determines the maximum number of items that can be added in a call to the `add_stock` method. For example, let's say the fashion shop owner is concerned that she might type `50` rather than `5`, which would cause her records to be incorrect. The present value of `StockItem.max_stock_add` is `10`, which means values above `10` would be rejected.

```

d = Dress(stock_ref='D01', price=100, color='red', pattern='swirly',
 size=12, location='Shop Window') Create a dress
d.add_stock(5) Add 5 dresses to stock
print(d) Print the dress

```

These three statements create a new `Dress` object, add `5` dresses to the stock for this item, and then print the dress details. The `add_stock` method is added to the `StockItem` class because this means that all the subclasses (`Dress`,

[Pants](#), [Hat](#), [Jeans](#), and [Blouse](#)) now have this method too.

## Sell a stock item

The final behavior that we must implement for our stock item is selling something. We can add another method to the [StockItem](#) class to do this:

[Click here to view code image](#)

```
EG11-09 Selling stock

class StockItem(object):
 """
 Stock item for the fashion shop
 """

 def sell_stock(self, count):
 if count < 1:
 raise Exception('Invalid number of
items to sell')

 if count > self.__stock_level:
 raise Exception('Not enough stock to
sell')

 self.__stock_level = self.__stock_level -
count
```

This method is called to sell a given number of stock items. The number of items to sell is given in the parameter [count](#). The method will raise an exception if the value of count is less than 1. It will also raise an exception if there are not enough items in stock to sell. The following sequence of statements creates a [Dress](#), adds [5](#) to the stock for the dress, sells [1](#) dress, and then prints the details of the dress.

```

d = Dress(stock_ref='D001', price=100, color='red', pattern='swirly',
 size=12, location='Shop Window')
Create a dress
d.add_stock(5)
Add 5 to stock
d.sell_stock(1)
Sell 1 dress
print(d)
Print the dress details again

```

## Objects as components

We have now completed development of the `StockItem` object. It contains all the behaviors that we will need for the first version of the Fashion Shop application. The `StockItem` object is completely self-contained, and we've been able to check that it works by performing actions on it and viewing the results. In the next chapter, we'll look at ways that we can test our objects automatically.

An object like `StockItem` is sometimes called a *component*, which is a self-contained part of a system. A car production line is another example of component-based manufacturing. A car contains several large components, such as the motor, quarter panels, axles, and transmission, which are created separately and then assembled to create a car. The car body contains fittings into which the motor is bolted, and then the motor output is connected to the wheels.

We've done something very similar with our `StockItem` class. You can think of the `StockItem` as the “motor” and the Fashion Shop application as the car in this analogy.



### PROGRAMMER'S POINT

Self-contained components are a great way to build software A great way to attack a large software project is to break it down into individual components. If we were working as a team of programmers to build the Fashion Shop application, we could have sat down at the beginning and decided that the `StockItem` class must hold a

particular set of attributes and provide a particular set of methods. Then, one programmer could build the **StockItem** component while others work on other parts of the system.

## Create a FashionShop component

We now have a fully working and tested **StockItem** component. In terms of the owner's original, paper-based system, we have implemented a single page to put in the binder. Now we need to decide how to keep track of the substantial number of **StockItem** objects that our application will manage. We can create a **FashionShop** component to do this. The **FashionShop** component must be able to do the following:

- Create a new fashion shop.
- Save the fashion shop stock data to a file.
- Load the data from a file.
- Store a new stock item.
- Find a particular stock item.
- Provide a listing of all stock items.

We can map each of these onto a method in the **FashionShop** class: [Click here to view code image](#)

```
EG11-10 FashionShop template

class FashionShop:

 def __init__(self):
 pass
```

```
def save(self, filename):
 """
 Saves the fashion shop item to the given
 file name
 Exceptions will be raised if the save fails
 """
 pass

@staticmethod
def load(filename):
 """
 Loads a fashion shop item from the given
 file name
 Exceptions will be raised if the load fails
 """
 return None

def store_new_stock_item(self, item):
 """
 Create a new fashion shop item
 The item is indexed on the stock_ref
 attribute
 Raises an exception if the item is already
 stored in the fashion shop
 """
 pass

def find_stock_item (self, stock_ref):
 """
 Gets an item from the stock
 Returns None if there is no item for
 this stock reference
 """
 return None

def __str__(self):
 return ''
```

You can think of this class as a template for the development of the finished object. We could give this class to a Python programmer and ask her to “fill in the blanks.” Each method has a documentation string that describes what the method does. Some of the methods return “placeholder” values that we’ll fill in later. For example, the `find_stock_item` method always returns the value `None`. When the method completes, it will return the stock item with the matching reference.

Note that once we have determined how the `FashionShop` component will be used, the precise way the methods work does not concern a program using the `FashionShop` class. You can think of these methods as “buttons” that can be pressed to make things happen. As long as the right thing happens when the button is pressed, we don’t care how the class works.

## Create a `FashionShop` object

We will create a version of `FashionShop` that uses a Python dictionary to store the stock items. However, a different implementation of the `FashionShop` class could use a database instead. We first saw the Python dictionary in [Chapter 9](#). A dictionary lets you find items in a collection by using a key value to index it. We have the perfect dictionary key in our application. Each item in the store is identified by a unique stock reference. We create the dictionary when we create a new `FashionShop` instance:

```
class FashionShop:

 def __init__(self):
 self.__stock_dictionary = {}
```

Create an empty stock dictionary

This is the `__init__` method in the `FashionShop` class. It creates an empty dictionary attribute called `__stock_dictionary`, which is part of the `FashionShop` class. It’s important that the contents of `stock_dictionary` are not changed by mistake, so we’ve marked it as private to the `FashionShop` class by beginning the attribute name with two underscore characters. If we want to store other information in the `FashionShop` class (perhaps a future version might store a customer list), we would create those attributes here, too. The Python statement below will create a new `FashionShop` instance and set the variable `shop` to refer to it.

```
shop = FashionShop()
```

## Save the FashionShop object

We'll save the data by asking a `FashionShop` object to save itself. It will do this by using the Python pickle mechanism. We have used the Python pickle mechanism to store items in previous programs. The contacts list in the Time Tracker application we wrote in [Chapter 10](#) was stored using Python pickle.

```
class FashionShop:

 def save(self, filename):
 ...
 Saves the fashion shop item to the given file name
 Data is stored in binary as a pickled file
 Exceptions will be raised if the save fails
 ...
 with open(filename, 'wb') as out_file: Open the file for binary output
 pickle.dump(self, out_file) Save the object to the file
```

We've seen the `with` construction before in [Chapter 8](#) (see “[Use the with construction to tidy up file access](#)”). The `with` construction ensures that the file is always closed correctly after it's been used. The method uses the self reference supplied to `save` to identify the object to be pickled (which is the `FashionShop` object itself). The two Python statements below create an empty `FashionShop` and then store it in a file called `FashionShop.pickle`.

[Click here to view code image](#)

```
shop = FashionShop()
shop.save('FashionShop.pickle')
```

## Load the FashionShop object

The `load` method in the **EG11-10 FashionShop template** sample code above has a `@staticmethod` decorator. We saw this decorator when we wrote the

Time Tracker application in [Chapter 10](#). It makes the `load` method part of the `FashionShop` class, rather than part of a `FashionShop` instance.

If you find this confusing, think about the problem we're using `@staticmethod` to solve. Methods like `find_stock_item` search for a stock item in a `FashionShop` object. However, we can't use the `load` method from a `FashionShop` object because at the time we're loading the `FashionShop`, we don't have a `FashionShop` object in place. Making the `load` method static means that it is part of the `FashionShop` class, not part of a `FashionShop` object.

```
class FashionShop:

 @staticmethod
 def load(filename):
 ...
 Loads the fashion shop from the given file name
 Data are stored in binary as a pickled file
 Exceptions will be raised if the load fails
 ...
 with open(filename,'rb') as input_file:
 result = pickle.load(input_file)
 return result
```

Make the `load` method part of the class  
Open the file for binary input  
Load the `FashionShop` using pickle  
Return the loaded `FashionShop`

The `load` method returns a reference to the object it has loaded. The statement below sets the variable `loaded_shop` to refer to a fashion shop item that's been loaded from a file called `FashionShop.pickle`.

[Click here to view code image](#)

```
loaded_shop =
FashionShop.load('FashionShop.pickle')
```

## Store a new stock item

The `FashionShop` class acts as a container for all the information concerned with running the fashion shop. It provides a method that can be used to store a new stock item.

```
class FashionShop:

 def store_new_stock_item(self, stock_item):
 """
 Create a new item in the fashion shop
 The item is indexed on the stock_ref value
 Raises an exception if the item already
 exists
 """
 if stock_item.stock_ref in self.__stock_dictionary: Check for an existing item
 raise Exception('Item already present') Raise an exception if the item is found
 self.__stock_dictionary[stock_item.stock_ref] = stock_item Add the item to
 the dictionary
```

The `store_new_stock_item` method adds a new stock item to the fashion shop. It is the Python equivalent of adding a new page to the file binder that holds the paper stock records. Before the item is added to the dictionary, the method checks to see whether it is already present. That way, if the fashion shop owner mistypes a stock reference she will not overwrite any existing stock items with new ones.

If the item is already present, the method will raise an exception; otherwise, the stock item is added to the stock dictionary using the stock reference value as the key. The following statements create a new dress and then add it to the stock of a fashion shop.

[Click here to view code image](#)

```
dress = Dress(stock_ref='D001', price=100,
 color='red', pattern='swirly', size=12,
 location='front')
shop = FashionShop()
shop. store_new_stock_item(dress)
```

## Find a stock item

The fashion shop owner searched for her stock items by flicking through the pages in her binder, looking for the stock reference of the item. The `FashionShop` class provides a method that can find a stock item based on its

stock reference:

```
class FashionShop:

 def find_stock_item(self, stock_ref):
 """
 Gets an item from the stock dictionary
 Returns None if there is no item for
 this key
 """
 if stock_ref in self.__stock_dictionary:
 return self.__stock_dictionary[stock_ref]
 else:
 return None
```

The code defines a `find_stock_item` method within a `FashionShop` class. It takes a `stock_ref` parameter. Inside the method, it checks if the `stock_ref` is in the `__stock_dictionary`. If it is, it returns the item from the dictionary. If it's not, it returns `None`.

The method returns the value `None` if no stock item matches the stock reference supplied. It's up to the code that uses this method to check whether a stock item was found. The statements below search for a stock item with the reference '`'D001'`'. If this is found, it is printed out; otherwise, a message is printed.

[Click here to view code image](#)

```
item = shop.find_stock_item('D001')
if item == None:
 print('Item not found')
else:
 print(item)
```

## List the stock data

The final behavior we need to implement is listing the stock data. This method should provide a string that describes the items in stock. Since each stock item class already has an `__str__` method that provides a string description of that item, we just need to get the string descriptions of all the items in stock and join them into a single string. We did this before in [Chapter 10](#), when we created a string that contained all the session information for a contact.

```

class FashionShop:

 def __str__(self):
 stock = map(str, self.__stock_dictionary.values())
 Get all the stock items into an iterator
 stock_list = '\n'.join(stock)
 Join the stock items to form a single string
 template = '''Items in Stock
{0}
...
{1}'''
 Template for the string to return

 return template.format(stock_list)
 Build the string to return

```

This code uses the `map` and `join` functions. If you're not sure how they are used, look again at the section “Session Tracking in Time Tracker” in Chapter 10 for details of how the string is built. The statements below create a new dress and add it to a `FashionShop` object. They then print the contents of the shop. I've included the output so you can see what is produced.

[Click here to view code image](#)

```

dress = Dress(stock_ref='D001', price=100,
color='red', pattern='swirly', size=12,
location='front')
shop = FashionShop()
shop.store_new_stock_item(dress)
print(shop)
Items in Stock

Stock Reference: D001
Type: Dress
Location: front
Price: 100
Stock level: 0
Color: red
Pattern: swirly
Size: 12

```

The example program **EG11-11 FashionShop class** contains the above sample code. You can use this to explore how the fashion items are stored. The

`StockItem` and `FashionShop` classes provide instrumentation that you can turn on to view how the program works. See the “[Make Something Happen - Instrumented stock items](#)” section earlier in this chapter to refresh your understanding of instrumentation.

The `FashionShop` class is much more than a fashion shop storage manager. You can use it in any application where you want to store items managed by a key. If we’re asked to create a program to implement a bank, manage a doll collection, or track entries in a competition, we can use exactly the same code.

## Create a user interface component

The final component of our Fashion Shop application is the user interface. This is the onscreen part with which the user will interact. Because we’re using object-oriented design, we’ll create a class called `FashionShopShellApplication` that will implement the user interface. This class just provides two behaviors:

- Initialize the application by loading a `FashionShop` object from a file (or creating a new `FashionShop` if this is the first time the program has been run).
- Display the menu for the user.

This class will create a text-based user interface using the command shell. Later, we’ll create a graphical user interface.

## Initialize the user interface class

The `__init__` method for the `FashionShopShellApplication` class starts by attempting to load the specified fashion shop file into a `FashionShop` object. If the load fails, the `__init__` method creates an empty `FashionShop` instance and tells the user that an empty shop has been set up. The `__init__` method sets the value of an attribute called `__shop` which refers to the `FashionShop` object that is being used.

```
class FashionShopShellApplication:

 def __init__(self, filename):
 ...
 Manages the fashion shop data
 Displays a message if the load fails and creates a new shop
 ...

 FashionShopShellApplication.__filename = filename Store the file name for use
 try:
 self.__shop = FashionShop.load(filename) Attempt to load a
 except: FashionShop object
 print('Fashion shop not loaded.') Tell the user the load failed
 print('Creating an empty fashion shop')
 self.__shop = FashionShop() Create an empty FashionShop object
```

The `__shop` attribute refers to the `FashionShop` instance with which the program is working. It will be used by all the other methods in the `FashionShopShellApplication` class. The statement below creates a new `FashionShopShellApplication` instance, which is using `FashionShop` data held in a file called `fashionshop.pickle`.

[Click here to view code image](#)

```
ui =
FashionShopShellApplication('fashionshop.pickle')
```

## Implement the user menu

The other method in the `FashionShopShellApplication` class repeatedly displays a menu of commands and lets the user choose what to do. This method will continue looping until the user decides to exit the program.

```

class FashionShopShellApplication:

 def main_menu(self):

 prompt = '''Mary's Fashion Shop

1: Create new stock item
2: Add stock to existing item
3: Sell stock
4: Stock report
5: Exit

Enter your command: '''

```

```

while(True):
 command = read_int_ranged(prompt, 1, 5)
 if command == 1:
 self.create_new_stock_item()
 elif command == 2:
 self.add_stock()
 elif command == 3:
 self.sell_stock()
 elif command == 4:
 self.do_report()
 elif command == 5:
 self.__shop.save(FashionShopShellApplication.__filename)
 print('Shop data saved')
 break

```

The diagram illustrates the flow of the code within the `main_menu` function. It shows the main loop starting with `while(True)`, followed by the assignment of `command` to `read_int_ranged(prompt, 1, 5)`. The code then branches into five `if` statements based on the value of `command`. Each `if` statement leads to a specific method call: `self.create_new_stock_item()` for command 1, `self.add_stock()` for command 2, `self.sell_stock()` for command 3, `self.do_report()` for command 4, and `self.__shop.save(FashionShopShellApplication.__filename)` for command 5. After command 5, a `print('Shop data saved')` statement is executed before the loop exits with a `break` statement.

This code should be familiar because it's the same format we've used for previous applications. The number the user enters is used to select a method that will perform that function. The two statements below show how a user interface is created and then the main menu displayed.

[Click here to view code image](#)

```

ui =
FashionShopShellApplication('dressshop1.pickle')
ui.main_menu()

```

## Implement the user interface behaviors

The methods called by the `main_menu` method are all members of the `FashionShopShellApplication` class. Each user interface method sends commands to the `FashionShop` instance to which the user interface is connected. As an example, consider the `sell_stock` method, which finds the item being sold and then updates the stock level for that item.

```

class FashionShopShellApplication:

 def sell_stock(self):
 ...
 Sells stock. Searches for the item and then reads the
 number of items being sold.
 Will not allow more items to be sold than are in stock
 ...
 print('Sell item')

 item_stock_ref = read_text('Enter the stock reference: ') Get the stock
 reference

 item = self.__shop.find_stock_item(item_stock_ref) Find the item we
 are selling

 if item == None: If the item wasn't found, display a message
 print('This item was not found')
 return

 print('Selling')
 print(item) Show the item that is being sold

 if item.stock_level == 0: Abandon the sale if there are none to sell
 print('There are none in stock')
 return

 number_sold = read_int_ranged('How many sold (0 to abandon): ',
 0,
 item.stock_level)

 if number_sold == 0: Abandon the sale if the user entered 0
 print('Sell item abandoned')
 return

 item.sell_stock(number_sold) Update the item with the number sold

 print('Items sold')

```

This method takes the user through the sales process. The user is first asked to identify the stock item to be sold. The method then asks for the number of items that have been sold and calls the `sell_stock` method on that item to update the stock levels. All the other methods are implemented in the same way.



## PROGRAMMER'S POINT

You will spend a lot of your time writing code to deal with failure. The `sell_stock` method does the following:

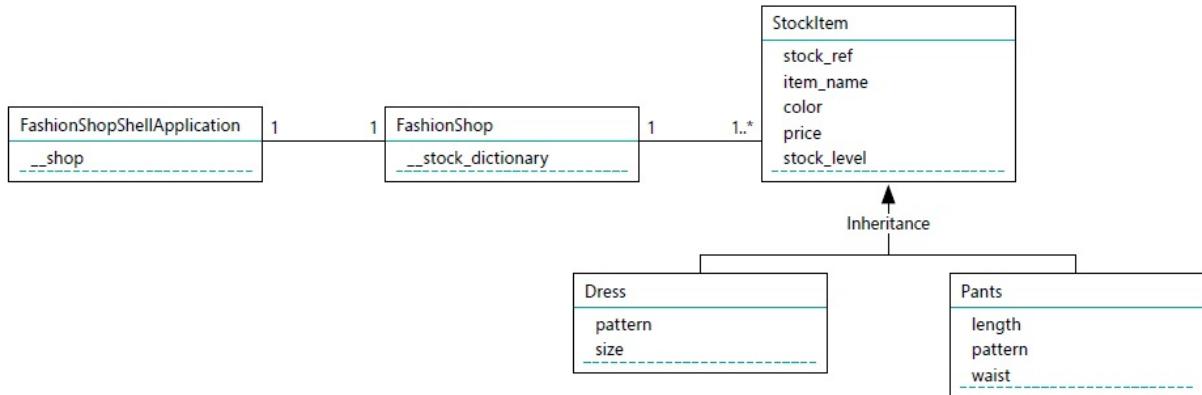
- It handles invalid stock references entered by the user.
- It ensures that it's not possible to sell more items than exist in stock.
- It provides a way that the user can abandon the sale if they find that they have selected the wrong item.

Software developers talk about the “happy path” through a program, which is the path followed when everything works and nobody makes any mistakes. The “happy path” code is often a very small fraction of the final application, and has to deal with all the ways that actions can go wrong. The happy path is not a problem as such, but it is something to remember when you try to decide how much work will be involved in creating a system.

The sample program **EG11-12 Complete Fashion Shop** contains a completely working and useable fashion shop manager that you can explore. You could actually use this to manage an actual fashion shop. You could also use it as the basis of any program you write that will allow the user to manage a set of related data items.

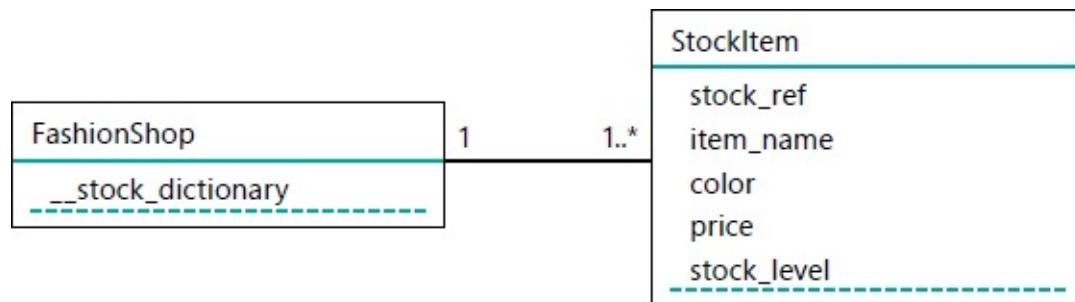
## Design with classes

**Figure 11-4** shows the classes in the finished Fashion Shop application. It is not quite complete. It shows only the `Dress` and `Pants` subclasses of `StockItem` and no longer shows that the `object` class is the superclass of all the other classes. However, it does show how the objects are related. The diagram shows that the `StockItem` class is the superclass of `Dress` and `Pants`, and it also shows the associations between the two other classes in the solution.



**Figure 11-4** Complete **FashionShop** class diagram

**Figure 11-5** lets us examine in detail how an association is described on this diagram. The line shows that **FashionShop** and **StockItem** are associated in some way. The numbers give the *multiplicity* of the association. The multiplicity values give the number of items on each side of an association. The number 1 next to the **FashionShop** class means that there is one **FashionShop** object on the left-hand side of the association. The string **1..\*** means that there are “one to many” **StockItem** objects on the right-hand side of the association. This is how, in software design, we express that a fashion shop can contain many items of stock. The **FashionShopShellApplication** is only associated with a single **FashionShop**, so multiplicity values on each side of that association are **1**.



**Figure 11-5** Associations between objects

Using class diagrams like this is a good way to start designing a system. The diagrams are also a good way to describe to other programmers how the elements of your program fit together.

# Python sets

Python lets us create variables that can hold sets of data. A set is a collection of values. However, unlike a list, each value in a set is unique. Sets are extremely powerful and allow us do things that our fashion shop owner would really like. However, first we must find out how they work.



## MAKE SOMETHING HAPPEN

### Investigating sets

We can experiment with sets by using the IDLE Command Shell. Open it, type the statement below, and press Enter.

```
>>> set1=set()
```

This statement creates an empty set. Enter the statement below and press Enter to view the contents of the set.

```
>>> set1
set()
```

We can add things to the set in a similar way as we append things to a list. A set object provides an [add](#) method.

```
>>> set1.add(1)
```

This statement adds the integer value [1](#) to the list. If we now view the set, we can see that it contains the value [1](#).

```
>>> set1
{1}
```

Currently, you probably can't see much difference between a set and a list. However, sets can hold only one copy of each possible value. We can prove this by trying to add the value `1` to the list again.

```
>>> set1.add(1)
```

If the variable `set1` was referring to a list, this would have added a second value of `1` to the list. However, if we view the contents of `set1` we'll find that it has not changed:

```
>>> set1
{1}
```

If we add a different value, it is added to the set. Enter the following two statements to add the value `2` to the set and then view the contents of the set.

```
>>> set1.add(2)
>>> set1
{1, 2}
```

A quick way to make a set is to use curly braces to enclose a set of values to be added. Let's create another set with the original name of `set2`.

```
>>> set2={2, 3, 4, 5}
>>> set2
{2, 3, 4, 5}
```

A set has methods that can be used to work with other sets. Let's look at a few, starting with the `difference` method.

```
>>> set1.difference(set2)
{1}
```

In the statement above, the `difference` method is running on the object referred to by `set1`. It returns a set that contains all the items in `set1` but not in `set2`. We can run the same method on `set2` to find all the items in `set2` that are not in `set1`.

```
>>> set2.difference(set1)
{3, 4, 5}
```

The `union` method returns a set that contains all the elements of both sets:

```
>>> set1.union(set2)
{1, 2, 3, 4, 5}
```

The `intersection` method returns a set that contains all the elements the sets have in common:

```
>>> set1.intersection(set2)
{2}
```

The only element in `set1` and `set2` is the element `2`.

We can also use methods on sets to compare their contents. The `isdisjoint` method returns `True` if the two sets have no elements in common:

```
>>> set1.isdisjoint(set2)
False
```

The two sets are not disjoint because they both contain the value `2`.

The `issubset` method returns `True` if one set is a subset of the other (meaning one set is contained entirely within another set). Let's create a new set to experiment with this.

```
>>> set3={2,3}
>>> set3.issubset(set2)
True
```

This is **True** because **set2** (which contains {2,3,4,5}) contains all the elements in **set3**.

The **issuperset** method returns **True** if one set is a superset of the other:

```
>>> set3.issuperset(set2)
False
```

This is **False** because not all the elements in **set3** are in **set2**.

Now that we know something about sets and their methods, you might be wondering when they are useful.

Sets can be used to remove duplicate items from a collection: [Click here to view code image](#)

```
set('hello world')
{'o', 'l', ' ', 'h', 'r', 'w', 'd', 'e'}
```

A set created from the string **hello world** contains all the unique letters in the string. You may notice that the order of the letters in the set is not alphabetic. Python does not store the elements of a set in any particular order. If you want to sort a set (or any other collection), you can use the Python function called **sorted**.

[Click here to view code image](#)

```
sorted('Rob Miles')
[' ', 'M', 'R', 'b', 'e', 'i', 'l', 'o', 's']
sorted(set('hello world'))
[' ', 'd', 'e', 'h', 'l', 'o', 'r', 'w']
```

We can also use sets to manage the inventory of a player in a video game: [Click here to view code image](#)

```
pocket = {'axe', 'apple', 'herbs', 'flashlight'}
```

This player is carrying a nice selection of items in their pocket. It is now easy to check for particular combinations of items in this set: [Click here to view code image](#)

```
apple_potion = {'apple', 'herbs'}
if apple_potion.issubset(pocket):
 print('You have the ingredients for the apple
potion')
```

The apple potion is made from apple and herbs. The above statement checks to see whether the player can make this potion. The minus operator can be used between set operands to subtract the contents of one set from another. The following two statements remove the ingredients from the pocket and then add the apple potion into the pocket of the player.

[Click here to view code image](#)

```
pocket = pocket - apple_potion
pocket.add('apple potion')
```

The statements above work because the subtract (-) operator can be applied between two sets. In the above statement, the elements in the `apple_potion` set (the `apple` and the `herbs` elements) will be removed from the `pocket` set.

## Sets and tags

Sets are one of those things you don't use very often, but when you do, they're very useful. The fashion shop owner has been in touch and has an idea for her

program that she wants you to implement. She's noticed that often her customers are pursuing a specific "look" or "style" when buying their clothes. They might be buying items for work, outdoors, or a formal occasion. She wants to add "tags" to her stock items so that she can easily search for the "formal" or "outdoors" (or even "formal outdoors") clothing items that she has in stock. Then she can direct customers to appropriate items.

Many items of data are tagged in this way. My blog posts are tagged "Python," "C#," "gadgets," and so on. I can tag my photographs as "sunset," "landscape," or "people." Sets are a very good way to represent tags on an item.

## Create a set from a string of text

We can give each `StockItem` in our fashion shop a `tags` attribute that is a set of tag values for that attribute. The tags will be entered when the stock item is created. We can enter them as a comma-separated list that we can then convert into a set of tags. In other words, the user would enter the following: [Click here to view code image](#)

```
Enter tags (separated by commas):
outdoor, spring, informal, short
```

The part of the program that reads in a `StockItem` would then convert the comma-separated list into a set of tags for this stock item. Let's consider how we would do this. We can use one of our text reading methods to read in the tag list.

[Click here to view code image](#)

```
tag_string = read_text('Enter tags (separated by
commas): ')
```

This statement reads in the tags and stores them in a variable called `tag_string`. The first thing we need to do is make sure that all the letters in this string are lowercase. As far as Python sets are concerned, the tags "Outdoor" and "outdoor" are different, and we want these to match up when we search for them.

[Click here to view code image](#)

```
tag_string = str.lower(tag_string)
```

This statement uses the `lower()` method provided by the `str` type to convert `tag_string` into lowercase. We've used string methods like this before when we converted names to uppercase in [Chapter 5](#).

We can create a set from a list of items, so next we need to convert the string of tags into a list. You can think of this process as the reverse of the `join` function that we used to create long strings of output from lists of stock item details. The `split` method provided by the `str` type will do this for us.

[Click here to view code image](#)

```
tag_list = str.split(tag_string, sep=',')
```

The `split` method takes a string and converts it into a list of items. We can use the keyword argument `sep` to specify the separator between each item that we want to split. In this case, we want to separate items using the comma character, so we give that as the value of `sep`.

Once this statement has completed, the variable `tag_list` contains a list of the tags that the user has entered. You might think that we could just use this list to create a set, but this might give us problems. Consider this input from our fashion shop owner: [Click here to view code image](#)

```
Enter tags (separated by commas): outdoor, spring,
informal, short
```

It looks very similar to the original list above, but there are some important differences between this list of tags and the previous one. This time, the owner has entered some spaces in front of the tag names. This is a problem for the program because the tags `spring` (without a space before the word) and `spring` (with a space before the word) would be regarded as different elements

in a set. This would result in some items not appearing in searches, which would result in complaints from the fashion shop owner. And pointing out that she had typed in the tags wrong would not improve the situation. We need a way to “tidy up” the tag names before we create the set.

In [Chapter 8](#), we learned about a string method called `strip`, which we used to strip non-printable characters from the start and end of a string. We want to apply the `strip` function to every string in the list of tags.

This turns out to be very easy. We first saw the `map` function in [Chapter 10](#), when we used it to apply the `str` function to a list of items. We can use `map` to apply the `strip` function from the `str` type to all the items in the tag list:  
[Click here to view code image](#)

```
tag_list = map(str.strip, tag_list)
```

If this is difficult to understand, consider what we’re trying to do here. We want to strip every string in the list of any non-printing characters (for example, space characters). We have a method that we know can do this. The job of the `map` function is to take a function and “map” it onto an entire collection of items.

Now that we have our tag list, we can use it to create a set:

```
tags = set(tag_list)
```

The best way to provide this behavior is as a method. The method takes a string and then returns the set that’s created from that string.

[Click here to view code image](#)

```
@staticmethod
def get_tag_set_from_text(tag_text):
 """
 Converts a comma-separated list into a set
 of individual items
 Converts the text to lowercase and trims each
 word
```

```
'''
Convert the string to lowercase
tag_text = str.lower(tag_text)

Make a list of all the words in the string
separated by the comma character
tag_list = str.split(tag_text, sep=', ')

Remove any spaces at the start or
end of each string in the list
tag_list = map(str.strip, tag_list)

return a set created from the list
return set(tag_list)
```

This is the method I've added to the Fashion Shop application. It is a static method, since the behavior that it provides is not tied to any particular instance of a class.

## Filter on tags

Now that we've added tags to the stock items, next we need a way to select the items in the stock list that contain the tags we're searching for. In other words, the owner of the fashion shop will type in a list of tags she wants to search for, and the program will display items it finds that match these tags.

[Click here to view code image](#)

```
Enter the tags to look for (separated by commas):
outdoor,spring
Stock Reference: BL343
Type: Blouse
Location: blouse rail
Price: 100
Stock level: 0
Color: pink
Tags: {'spring', 'friendly', 'city', 'outdoor'}
```

Size: 14  
Style: plain  
Pattern: check

The output above shows how this will work. The owner is searching for outdoor wear for spring, and the program has found a blouse that matches these criteria. We can use the method `get_tag_set_from_text` to create the search set, and now we need to work through all the stock items and find those that “match.”

In this case, the word match means “is the search set a *subset*” of the tags on this item. In the example above, the search set is `{ outdoor, spring }` and the tags on the matched item are `{ spring, friendly, city, outdoor }`. The item has been selected because its tags contain all the tags in the search set.

We can use the method `issubset` to test whether the search tags are a subset of the tags on the item.

[Click here to view code image](#)

```
def match_tags(item):
 """
 Returns True if the tags in the item
 contain the search tags
 """
 return search_tags.issubset(item.tags)
```

The function `match_tags` will return `True` if the search tags are a subset of the tags on the item passed as a parameter to the method. I’ve written this function because I want to use it in a Python function we haven’t seen before. The function is called `filter`. It works in the same way as the `map` function. The `map` function returns the result of applying a function to all the elements in a collection. This is how we applied `trim` to all the strings in the list of tag strings the user had entered.

The `filter` function returns the result of testing all the elements in a collection

with a filter. If the function returns `True`, the resulting collection contains that item.

[Click here to view code image](#)

```
filtered_list = filter(match_tags, stock_list)
```

The statement above sets the variable `filtered_list` to refer to a collection of stock items that have been filtered by the `match_tags` method. If we want to change the filter behavior (perhaps to return items that don't match the given tags), we just need to create a different filter function and use that instead.

[Click here to view code image](#)

```
def find_matching_with_tags(self, search_tags):
 """
 Returns the stock items that contain
 the search_tags as a subset of their tags
 """

 def match_tags(item):
 """
 Returns True if the tags in the item
 contain the search tags
 """
 return search_tags.issubset(item.tags)

 return filter(match_tags,
self.__stock_dictionary.values())
```

This is the `find_matching_with_tags` method from the `FashionShop` class. It returns a list of stock items that contain the search tags supplied as a parameter. It uses a Python feature that we haven't seen before. The `match_tags` function is declared inside the `find_matching_with_tags` method. Python allows functions to be declared inside program code. We'll discuss this in more detail in the next chapter, when we discover how to create

libraries of Python code.

The sample code file **EG11-13 Fashion Shop with Tags** contains a version of the Fashion Shop application that allows you to tag stock items and search for them.

## Sets versus class hierarchies

One of the wonderful things about writing programs for people is finding out what they do with them. You've delivered your "tagged" version of the Fashion Shop application to the owner, and she's very pleased with it. She starts using the new version of the program. After a while, she gives you a call to discuss some changes she'd like to make to the code. This is a worrying thing to hear from a customer, so you go along to find out what's wrong.

It turns out that nothing is broken in the code, but the fashion shop owner would like to make some big changes to the way the application works. She really likes using tags to identify elements in her stock and is finding it a pain to have to enter things as pants, dresses, jeans, and so on. Instead, she wants to index all stock using tags. Dresses would have the **dress** tag, pants could have the **pants** tag, and so on. In other words, she wants to change from this: [Click here to view code image](#)

```
What kind of item do you want to add: 1
Creating a Dress
Enter stock reference: D001
Enter price: 120
Enter color: red
Enter location: shop window
Enter tags (separated by commas): evening, long
Enter pattern: swirly
Enter size: 12
```

To this:

[Click here to view code image](#)

```
Enter stock reference: D001
Enter price: 120
Enter tags (separated by commas):
dress,color:red,location:shop
window,pattern:swirly,size:12,evening,long
```

She finds it very easy to search for the different tags, and the tags themselves remind her of the notes she used to write on the stock pages. If she starts to stock a new kind of item, or if she decides on a new way of describing items she has, then she just has to invent a new tag. The only change that she really needs to make this work is for us to add the ability to edit the tag information on an item. That way, she can add new tags. We tell her of the dangers of working this way: If she mistypes a tag, it will mean that searches for that item will fail. But she doesn't care about that. She just loves tags.

The sample code file **EG11-14 Tag only fashion shop** holds a version of the Fashion Shop application that works entirely with tags.

## Advantages of using sets and tags

Using sets to implement tags like this brings a few advantages:

- The client is now completely in control of how she organizes her stock.
- She can just continue adding tags and searching for them. She might find this a bit unwieldy if the number of tags gets very large, but that is something she seems happy to live with.
- From a programming point of view, this greatly simplifies the program, in that there is now no need for a class hierarchy. Everything in the system can be a stock item.

## Advantages of using classes

The major advantage of using classes to manage data like this is that the application can enforce strict rules on the objects created. If we use classes, we can ensure that every dress in the system has a size, pattern, and color attribute,

because all this information is required to create a dress. If the shop owner uses tags to describe a dress, she runs the risk of missing data from stock items. In other words, if we use tags the shop owner is under no obligation to specify the color of a dress.

Classes also allow polymorphism. We can make a dress object behave differently from pants when it performs a particular action. We used this when we wrote the `__str__` methods to provide a string description of the contents of a stock item. Each stock item could give a description of its attributes that matched that type of item. If we just tag all the items, we don't have an easy way to make different kinds of items behave in different ways.



## PROGRAMMER'S POINT

What's important to the programmer might not be important to the customer. This is one of the hardest lessons that you must learn when writing software for a customer. Sometimes things that you think are great, important, or useful are of no interest to the customer. And just as often, programmers don't recognize things that are important to a customer. In the case of the Fashion Shop application, we made the assumption that it was very important to store all details of all stock items all the time. A class hierarchy is a great way to achieve this, and so we used that mechanism.

However, it turns out that as long as the system rigorously manages the stock reference, price, and the number of items in stock, the customer doesn't really care about the other information in her stock items. She finds it much more useful to be able to add and modify tags and organize her stock as she goes along.

Finding out what's important to the customer is a crucial part of creating a project with a happy ending.



## CODE ANALYSIS

### Design decisions

A good way to explore these issues is by looking at some scenarios and deciding whether tags or a class hierarchy makes sense.

**Question:** We've been asked to create a set of diverse kinds of bank accounts by a local bank. The bank offers savings, credit, and checking accounts. Should we use a class hierarchy for this?

**Answer:** A class hierarchy would work best in this situation. We need control over all the attributes in the items we're storing. Some elements of each account will be common (for example, the name and address of the account holder), but other items will be different for each account. For example, a credit account will need a credit limit attribute. A superclass called account that holds the common items, and subclasses that add their own attributes would make sense.

Furthermore, a class hierarchy would provide polymorphism. The process of withdrawing funds will be different for each kind of account, so each subclass can provide its own method for performing this function.

**Question:** We've been asked to create an artwork tracking system for a local gallery. The gallery holds pictures, sculptures, and manuscripts. Should we use a hierarchy for this?

**Answer:** It might not be a good idea to use a hierarchy in this situation. Each item in the gallery will have different properties, and the gallery may well want to evolve and change the way that they categorize their stock. Sets and tags would work well in this situation. In fact, we could probably suggest that they use the Fashion Shop application directly, since it's flexible enough to

allow the gallery owner to store any kind of information about exhibits.

## What you have learned

In this chapter, you created another useful application. The application was built from components and uses inheritance to allow it to manage a large number of related items, each of which has slightly different characteristics. Inheritance lets programmers create subclasses (children) based on superclasses (parents). The subclass inherits all the attributes of the superclass and can use overriding to replace behaviors in the superclass with behaviors more appropriate to the subclass. As an example, the Fashion Shop application contains a `StockItem` class, which is the superclass of the `Dress` and `Pants` classes. The `StockItem` contains all the information common to any item of stock, and the `Dress` and `Pants` subclasses hold information specific to that type of clothing. Each of the subclasses overrides the `__str__` behavior of the superclass, so that a `Dress` can generate a correct string description of itself.

You discovered how breaking down systems into components makes it easy to develop and test each component separately. A component can provide a set of method attributes that make the component do the job it was created to perform. As an example, the `StockItem` class contains a method called `add_stock`, which can be used to increase the stock levels. Other parts of a system can regard a particular component purely in terms of the methods it provides, without concern for how the methods work. The `FashionShop` class in the example version of the Fashion Shop application uses a dictionary to store the stock items, but this could be replaced by a `FashionShop` class that uses a database without any need to change the other classes in the system. Components can be developed and tested individually once their method attributes have been established.

You were introduced to sets. A set is a collection of unique values that can be managed by Python, which provides functions that can manage the contents of a set and also determine relationships between sets. You found that you can use sets to allow objects to be “tagged” with the contents of a set. We can then search and filter collections of objects based on their tags. We used this to

provide a simpler Fashion Shop application, which replaced various types of objects with tags that hold information about different stock items.

Finally, you discovered (or at least were told) that it's very important for a programmer to make sure they fully understand what's important to a user of the system they're creating.

Here are some points to ponder about what we have learned.

### **Do I have to use a class hierarchy if I want to store many related items?**

No. I tend to use class hierarchies when I really want each of the related items to use polymorphism to behave in a unique way. We'll see this ability used to very good effect when we create a video game in [Chapter 16](#). Each of the elements of our video game will have an "update" method, but the action of the update method will vary for diverse types of elements. The update method for a player will read the game controls and move the player around the screen, the update method for an alien will make the alien chase the player, and so on.

### **What turns an object into a software component?**

We decided that creating software components is a good thing. A software component is *cohesive*. It can perform all its functions without needing to make use of other objects. If you consider the `StockItem` class from the point of view of cohesion, you'll find that every action that I ask the `StockItem` to perform is performed by that object alone, without reference to any other object. A system with poor cohesion would have the methods `create_stock_item` and `sell_stock` located in different classes.

A good test of how cohesive an object is would be to consider the difficulty of swapping that component for another that worked in the same way. In the case of `StockItem`, I could change the `StockItem` class for a different version that worked in the same way, and I would not have to change any other classes.

It's important that the way you interact with a component is very well defined. The `StockItem` class provides a set of methods that can store and manage data about a stock item. It doesn't provide any other methods for any other purpose.

The most important thing about component-based design is that you design the

components first. You decide what data attributes each component will have. You then decide on the method attributes and how the individual components are related. Then you can work on these components individually and then integrate them to create the finished solution.

## **Do I have to use object-oriented design to make my programs?**

No. A great thing about Python is that we can achieve polymorphism (making objects behave in a manner appropriate to them) just by adding a method with a specific name to each object. When we call that method on the object, it will provide its own, characteristic action.

There's no need to create a framework of classes and design their attributes before we write any code. We can just dive in and start writing the program. This is great fun, and sometimes it even results in a working solution. However, programs written in this way are extremely hard to fix or upgrade, because a programmer will have a hard time figuring out how the program works before they can make any changes to it.

You might think that you write programs for computers, and I suppose that this is true, in that the program will be obeyed by a computer. But I try to write programs for other people, specifically those who will come after me and have to work on programs that I've created. My priority when writing code is to try to make it as easy to understand as possible. This can sometimes mean that my code runs a bit slower than it might if I focused on speed alone, but modern computers are so powerful that this is rarely an issue.

## **Why is the relationship between a subclass and a superclass so confusing?**

If it isn't confusing for you, then move on to the next question. However, I have always found this confusing for the simple reason that the superclass is the one with the fewest abilities. Because the subclass is an extension of the superclass, it will always be able to do at least as much as the superclass. The reason the superclass is called super is that it is at the top of the tree of classes, and the class from which other things are descended. If you're having difficulty remembering how this works, remember that if a superhero was at the top of a class diagram containing people, it would mean that all the people in the class diagram would have super powers.

## **What exactly happens when a method in a class is overridden?**

Overriding a method means providing a behavior that is more appropriate than the behavior in the superclass. When Python calls a method attribute on a class, it first looks in that class for the method. If it doesn't find the method in the class, it looks for the method in the superclass of that object. It repeats this process until it either finds the method or runs out of objects in the hierarchy. Allowing methods to be overridden does slightly slow down a Python program because a method call might involve this searching process, but the Python system is very good at searching very quickly.

## **What things in my class should be made static?**

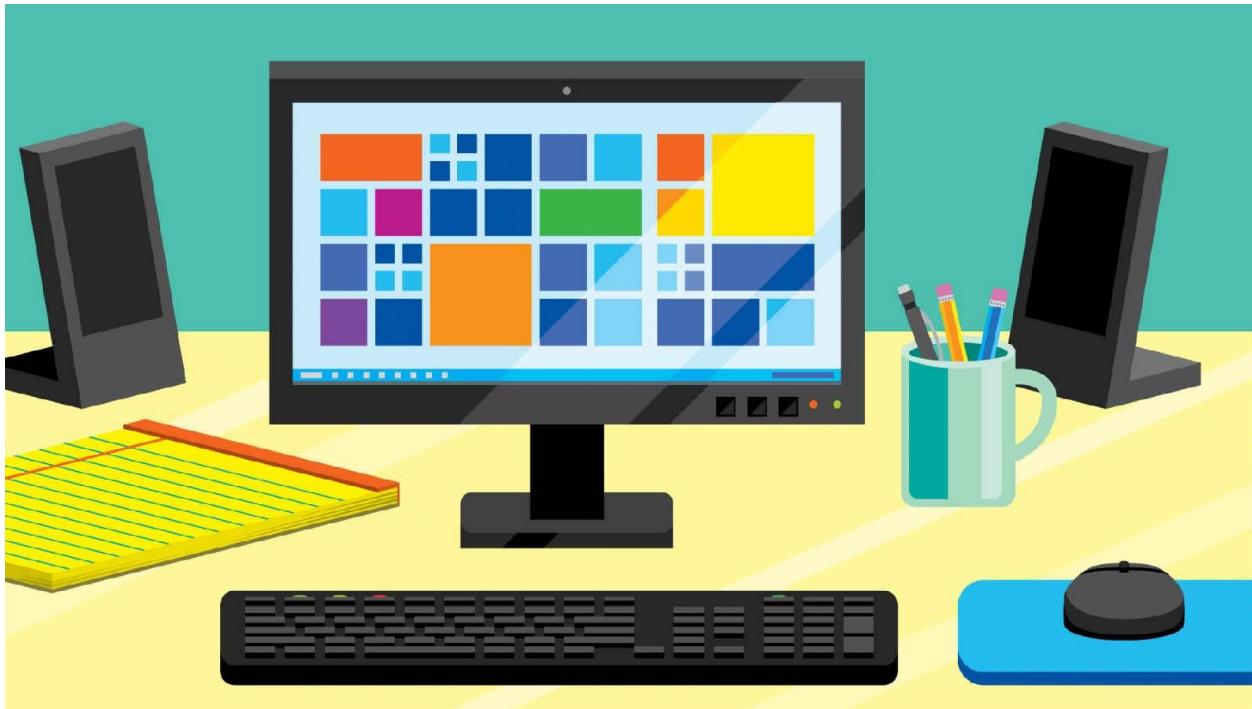
Static items are always present. We don't need to create any objects to use static items because they are part of classes, rather than objects. You use static data attributes to store values for class-wide use. For example, the maximum price of a stock item can be made a static value. A program might want to find the maximum price of a stock item without having a stock item to ask, and the maximum price should be stored only for the class, not for every stock item.

You use static method attributes to create a method that can be used without needing an instance of the class. The `load` method is a very good example of a `static` method because it can't be part of an object because the object hasn't been loaded yet.

## **When do I use abstraction?**

You use abstraction when you're thinking about the objects in the system you're creating. For example, abstraction would help you avoid getting distracted by the various kinds of customers with which your sales management system might need to work. Abstraction says that you regard them all as "customers," decide what customers need to do, and then create more concrete (less abstract) classes as you develop your design.

# 12 Python applications



## What you will learn

In this chapter, you'll discover how to create full-fledged Python applications. We'll start by discovering a full set of extra Python function features that provide added flexibility and power in program design. Then we'll learn to package programs as modules and packages. Finally, we'll look at how to test and document a Python application.

[Advanced functions](#)

[Modules and packages](#)

[Program testing](#)

[View program documentation](#)

## What you have learned

# Advanced functions

In this section, we'll build on our understanding of how Python functions work and discover some really useful and powerful features of the language.

## References to functions

We've already seen references to functions in our programs. In [Chapter 7](#), we discovered that we could create variables that refer to functions. In [Chapter 10](#), we saw how the Python `map` function is given a reference to a function and then applies (or maps) that function to all items in a collection. In [Chapter 11](#), we saw how the `filter` function could use a supplied function reference to select items from a list. Now, we'll explore function references in more detail. Let's start by considering the following code:

```
EG12-01 Simple Function References

def func_1():
 print('hello from function 1')

def func_2():
 print('hello from function 2')

x = func_1
x()
x = func_2
x()
```

The diagram illustrates the execution flow of the code. It highlights the creation of functions `func_1` and `func_2`, and the assignment of variable `x` to each. It also shows the subsequent calls to `x` which result in the execution of the correct function based on the current assignment.

- `def func_1():` Create a function called `func_1`
- `def func_2():` Create a function called `func_2`
- `x = func_1` Set the variable `x` to refer to `func_1`
- `x()` Call the function to which `x` refers
- `x = func_2` Set the variable `x` to refer to `func_2`
- `x()` Call the function to which `x` refers

This code contains two functions called `func_1` and `func_2`. The variable `x` is made to refer to each function in turn and is called after it has been assigned. In the program above, when the variable `x` is used as the basis of a function call, a call is made to the function to which variable `x` refers. As a result, the program will print the following:

[Click here to view code image](#)

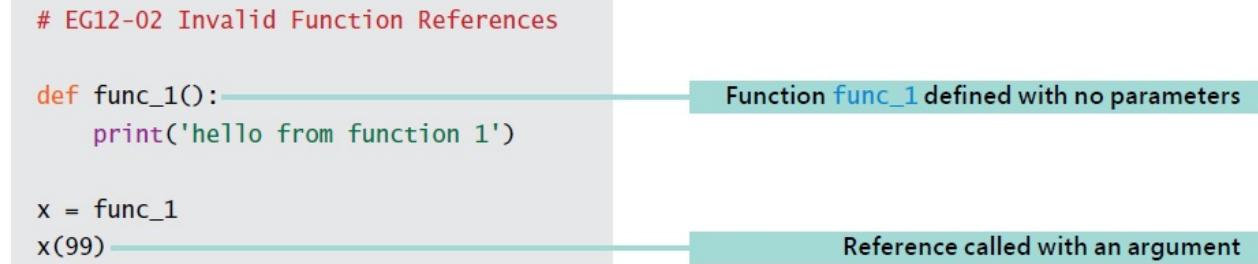
```
hello from function 1
hello from function 2
```

You can think of a variable that refers to a function as a direct replacement for the function name. If we use the reference incorrectly, we'll see the same error that's produced when we call the function incorrectly.

```
EG12-02 Invalid Function References

def func_1():
 print('hello from function 1')

x = func_1
x(99)
```



The above program will generate an error, because the call to reference `x` is given an argument (`99`) that the function to which it refers cannot accept because `func_1` was defined as having no arguments.

[Click here to view code image](#)

```
Traceback (most recent call last):
 File "C:/Users/Rob/EG12-02 Invalid Function
References.py", line 7, in <module>
 x(99)
TypeError: func_1() takes 0 positional arguments
but 1 was given
```

Above, you see the error that's produced, but the error output doesn't mention the variable `x`. Although the function call was made via the variable `x` (which contains the reference to the function), the error report simply states that the function to which `x` refers—in this case `func_1`—was not called correctly.

## Use function references in the BTCInput module

We can see a good example of the power of function references when we

consider the number input functions in the BTCInput module. This module contains functions based on ones we developed in [Chapter 7](#) to help our programs read input from the user. It contains a set of functions for reading numbers, including a function that can read an integer.

```
def read_int(prompt): Function supplied with a string to use as a prompt
 while True:
 try:
 number_text = read_text(prompt) Read in the text of the number the user enters
 result = int(number_text) # read the input Use the int function to
 # if we get here, no exception was raised
 # break out of the loop
 break
 except ValueError:
 # if we get here, the user entered an invalid number
 print('Please enter a number')

 # return the result
 return result
```

This code shows the `read_int` function in the BTCInput module. Programs can use it to read an integer from the user. It deals with any exceptions that would normally be raised if the user enters something like “`twelve`” rather than “`12`”. The `read_int` function can be used in a program as follows:

[Click here to view code image](#)

```
age = BTCInput.read_int('Enter your age: ')
```

The statement above calls `read_int`, which reads an integer value and assigns the result to a variable called `age`. The BTCInput library also contains a function called `read_float` that reads-in a floating-point number. The `read_float` function is the same code as `read_int` except for one difference. The `read_float` function creates a floating-point number from the text the user enters rather than creating an integer. We can remove the need for duplication of the code by creating a generic `read_number` function that can be used to read any value that can be converted from a string of text into a numeric value:

```
def read_number(prompt, number_converter):
 """
 Displays a prompt and returns a value obtained
 by applying the number_conversion function to the text
 entered by the user.
 If the number_conversion function raises an exception,
 the read is retried.
 Keyboard interrupts (Ctrl+C) are ignored
 """

 while True:
 try:
 number_text = read_text(prompt)
 result = number_converter(number_text) # read the input
 # if we get here, no exception was raised
 # break out of the loop
 break
 except ValueError:
 # if we get here, the user entered an invalid number
 print('Please enter a number')

 # return the result
 return result
```

The function to perform number conversion is now passed as a parameter

Use the `number_converter` parameter to perform the number conversion

This code looks very similar to the `read_int` function, but the `read_number` function has an extra parameter, called `number_converter`. This parameter is a reference to the number conversion function that will be used to take in a string of text and generate the method's result.

We can now tell the `read_number` function the process that converts the user entered string into a result. We can then create a `read_float` function, which calls `read_number` and provides `float` as the number converter function that converts the text into a number.

[Click here to view code image](#)

```
def read_float(prompt):
 return read_number(prompt=prompt,
 number_converter=float)
```

The above code is a version of the `read_float` method that accepts a prompt

string as a parameter and returns a floating-point result. It calls the `read_number` function to do this. Note that the `number_converter` argument is given as the `float` function because `float` is the function we want `read_number` to use to convert the text input into a value.



## CODE ANALYSIS

### Function references

Function references are complicated. I'd be surprised if you didn't have some questions.

#### Question: What is a function reference?

**Answer:** You can think of a function reference as one piece of a program telling another piece of the program what to do. People tell each other what to do all the time. When we're cooking, my wife will sometimes give me some potatoes and say, "Please peel and roast these. Other times, she'll give me the potatoes and say, "Please put these in the oven and bake them." I'm getting two inputs:

- The potatoes
- Instructions telling me what to do with them

The `read_number` function is given a prompt string for the user followed by a reference to the function it will call to convert the string into a numeric value. You can think of a function reference parameter as a way of telling a function what to do.

#### Question: Why is using function references like this a good idea?

**Answer:** The `read_float` and `read_int` functions do almost the same thing. They both read numbers. The only difference between the two functions is that one function uses the `int` method

to generate the result from the input string and the other uses the `float` method. It's not good programming practice to have two functions that are identical except for a single statement. If you find yourself writing code like this, try to work out how to use a parameter to modify the behavior of a single function to make it do the job of both. The best way to tell the `read_number` function what to do is to pass it a reference to the function you want it to use.

Note that I'm not using a single function because I want to save space in my program by only having one `read_number` function in place of two. I'm using one function so that if I find a mistake in my number reader, or I need to modify it, I need only change one function.

**Question:** If I wrote a function that converted Roman numerals into a numeric result, could I use `read_number` to read Roman numbers?

**Answer:** Yes. I once wrote a function that accepted Roman number strings such as “`XVI`” and calculated the number they represent (in this case, 16). You might want to try doing this. The number conversion function you create must accept a string and return a number. If your function is given as a string that it can't convert into a number, it must raise an exception. This is exactly how the `int` and `float` functions work. You could call your function `roman_converter`. We could then use your function with `read_number`:

[Click here to view code image](#)

```
age = read_number(prompt='Enter your age in
roman numerals',
number_converter=roman_converter)
```

The `read_number` function has a wide range of possible uses. If we write a function to convert date strings (for example, '`12/10/2017`') into date values, we could use `read_number` to repeatedly ask for dates until the user entered a valid one.

Using a function as a parameter, as we have done with the `number_converter` function parameter, is a form of abstraction. Abstraction is a way of “stepping back” from a problem and coming up with general solutions that we modify to make more specific. We’ve written a method that repeatedly asks the user a question, uses a function that we’ve supplied to process the answer, and returns a result only when the process works. We can use the `read_number` function to request and process any user input.

## Build lists of function references

We can use function references as we would any other kind of data. We can even create collections containing function reference values.

```
EG12-3 Robot Dancer

import time

def forward():
 print('robot moving forward')
 time.sleep(1)

def back():
 print('robot moving back')
 time.sleep(1)

def left():
 print('robot moving left')
 time.sleep(1)

def right():
 print('robot moving right')
 time.sleep(1)

dance_moves = [forward, back, left, right] List of dance move function references

print('Dance starting')
for move in dance_moves: Work through each dance move
 move() Call the dance move function
print('Dance over')
```

This code could be used to control a dancing robot. The robot has four moves it can perform, each of which is represented by a function. In the example above, the functions just print a message and pause the program for a second; we can go back and fill in the actual behaviors later. The `dance_moves` list contains a list of references to the robot functions. The `for` loop at the end of the program works through this list and calls each function in turn. The output from the program would look like this:

[Click here to view code image](#)

```
Dance starting
robot moving forward
robot moving back
robot moving left
robot moving right
Dance over
```

We could make the dance longer by adding elements to the list. We could write a program that lets the user select each move and build up a dance sequence of their own.

## Use lambda expressions

Lambda expressions can be very useful, though some people think they're quite hard to understand (perhaps because the name sounds rather mathematical and complicated). However, a lambda expression is a means of creating a piece of functional behavior that we can use as a value in our programs. That doesn't sound complicated at all, does it?

A function takes something, does something with it, and then returns a result. You can think of a lambda expression as a tiny function that we can write on one line. [Figure 12-1](#) shows the anatomy of a lambda expression.



**Figure 12-1** Anatomy of a lambda expression

The colon character serves to separate the list of arguments (stuff going in) from the expression that is evaluated (stuff coming out). As an example, look at the following Python code:

```
def increment(x):
 return x+1
```

This code defines a function called `increment`, which takes in a value (the parameter `x`) and returns a result (the value of `x + 1`). The same thing written as a lambda expression would look like this:

[Click here to view code image](#)

```
increment = lambda x : x+1
```

This expression also defines a function called `increment` which returns the value of its parameter, plus one. On the left side of the colon is the parameter supplied to the lambda expression; on the right is the expression (`x+1`) that provides the result.



MAKE SOMETHING HAPPEN

## Make a lambda expression

The best way to learn how to use lambda expressions is to work with

them, so let's do that. Open the Python Command Shell in IDLE and enter this statement:

[Click here to view code image](#)

```
>>> numbers = [1,2,3,4,5,6,7,8]
```

This statement creates a list called `numbers`, which contains the numbers 1 to 8. These numbers will be the data which our lambda expression will work on. Perhaps we might want to create a list that contains the numbers 1 to 8 with one added to them. What we need is a way of creating a new list with the updated values.

We can use the `map` function to do this. As we know from [Chapter 10](#), `map` is a built-in Python function that applies a function to all the elements in a collection and outputs a collection with the updated values. We used `map` to convert all the Fashion Shop stock item objects into strings by using the `str` function.

We can create a function called `increment` and use `map` to apply the `increment` function to all the values in the `numbers` list. The code would look like this (there's no need to type this code in):

[Click here to view code image](#)

```
def increment(x):
 return x+1

new_numbers = map(increment, numbers)
```

When the program runs, the `map` function generates an iterator that will apply the `increment` function to each of the elements in the `numbers` list. This program works, but it is rather tedious to have to type in the `increment` function and give it a name. We can do this much more simply by creating a lambda expression that describes the increment behavior. We can use the value of this expression as an argument to the

**map** call. Enter the statement below:

[Click here to view code image](#)

```
>>> new_numbers = map(lambda x : x+1,
numbers)
```

The call to the **increment** method has been replaced by a **lambda** function that accepts a single parameter and returns the value of that parameter plus one. Remember that the **map** function returns an iterator that works through the given collection and returns a new collection. We can view the contents of the **new\_numbers** iterator by using the Python **list** function to convert this iterator into a list. Enter the following statement and press Enter:

[Click here to view code image](#)

```
>>> list(new_numbers)
[2, 3, 4, 5, 6, 7, 8, 9]
```

As you can see, the **new\_numbers** list holds the incremented values from the list. You can find this example code in the sample file **EG12-04 Lambda Example**.

We can do another “lambda” investigation by creating a lambda expression and assigning it to a variable. Type in the following and press Enter:

[Click here to view code image](#)

```
>>> adder = lambda x, y : x+y
```

The expression on the right side of this assignment is a lambda expression that accepts two parameters (which I’ve called **x** and **y**) and returns their sum. This function is then assigned to the variable **adder**.

We can now use `adder` as a function in our programs. Type in the following and press Enter:

```
>>> adder(1, 2)
```

When you press Enter, Python follows the `adder` reference to the lambda expression and evaluates it.

```
>>> adder(1, 2)
3
```



## CODE ANALYSIS

# Lambda expressions

You might have some questions about lambda expressions.

**Question:** What are lambda expressions?

**Answer:** You can think of a lambda expression as a lump of data that describes an operation to be performed on something. We already explored the idea of passing method references into functions. In the previous section, we used this technique to tell the `read_number` function which number decoding function to use. We tell `read_number` to use the `float` or `int` functions when it reads a number by passing the `read_number` function an argument that specifies the function to use.

In the case of the `read_number` function, we had to give it the names of the functions (`int` or `float`) that we wanted it to use. In

the case of a lambda expression, we create a value that contains the behavior itself.

**Lambda** expressions are sometimes called *anonymous functions* because they are little bits of functional behavior (they do something with an input to produce a result) that don't have a name.

**Question:** Must I use lambda expressions in my programs?

**Answer:** No. Lambda functions don't allow us to do new things, but they do make writing programs easier and quicker. Lambda functions are particularly useful when used in conjunction with Python's `map` and `filter` functions.

**Question:** Can a lambda function contain more than one action?

**Answer:** No. The active part of a lambda function is a single expression that calculates the value to be returned.

**Question:** Can a lambda expression accept multiple arguments?

**Answer:** Yes, it can. The first lambda expression we wrote, which we used to add one to the values in a list, had only a single argument. The second expression, which returned the sum of two values, had two arguments. You can use as many arguments as you like.

**Question:** Can a lambda expression make decisions?

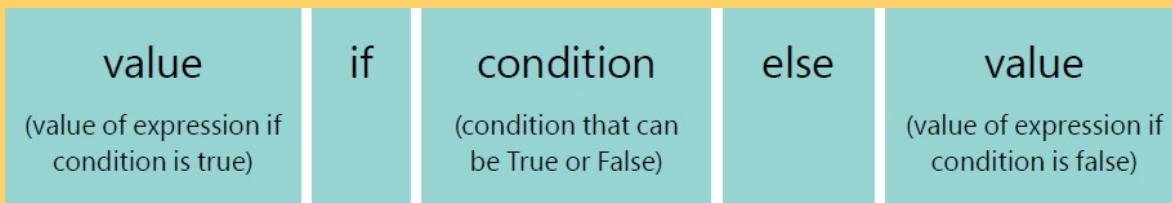
**Answer:** A lambda function can return `True` or `False` as its result, which can be used in conditions in programs. It can also use an expression called a *conditional expression* to allow it to choose a value to return. We haven't seen conditional expressions before. They allow a single expression to provide a result controlled by a Boolean expression.

[Click here to view code image](#)

```
hour = 8 # set the value of hour to the
```

```
current hour of the time
print('morning' if hour < 12 else
'afternoon')
```

The `print` statement above prints the result of a conditional expression that generates the value `morning` if the hour is less than 12 and the value `afternoon` if the hour is greater than or equal to 12. You can see the anatomy of a conditional expression in the diagram below.



You can use conditional expressions anywhere you can use an expression in a Python program. A lambda expression can return the value of a conditional expression:

[Click here to view code image](#)

```
day_prompt = lambda hour: 'morning' if hour < 12 else 'afternoon'
```

The statement above creates a function called `day_prompt` that accepts a single parameter and returns `morning` if the value of that parameter is less than 12. The variable `day_prompt` is set to refer to this function. We can use this as follows:

```
print(day_prompt(5))
```

The statement above calls the lambda expression referred to by the variable `day_prompt`. The value `5` is supplied as an argument to the lambda expression, which produces the value `morning`. So, the statement above would print `morning`.



## PROGRAMMER'S POINT

Don't worry if you don't get lambda expressions the first time you see them

It took me a while to understand lambda expressions. Don't worry if they seem difficult to understand at first. They represent a blurring between data and program code. Up until now, our applications have been made up of data (which holds values) and code (which gives the operations to be performed on the data). A lambda expression is a piece of data that specifies an action to be performed. Sometimes, it's useful to be able to create a piece of program code that you can pass around like a lump of data, and lambda expressions provide a way of doing this.

## Iterator functions and the `yield` statement

In Chapter 10, when we first used the `map` function, we came across the Python *iterator*. If you're not clear on what an iterator is, go back through the “[Make Something Happen: Investigating the `map` function and iteration](#)” section.

Up until now, the programs we've written have used iterators by consuming the values they produce. In other words, the programs have worked through the results that an iterator generates. One of the first iterators we encountered was generated by a built-in Python function called `range`. We used the `range` function to create a sequence of numbers:

[Click here to view code image](#)

```
for i in range(1, 5):
 print(i)
```

The `range` function in the code above returns a result that is an iterator that produces the values 1 to 4. The above statements would consume the iterator and print the values 1 to 4. Remember that the upper value of the range represents the value that terminates the range, not the last value in the range.

```
1
2
3
4
```

In this section, we'll discover how to create our own iterators. We can make an iterator by using a Python statement that we haven't seen before: `yield`. In English, the word "yield" can be used in more than one way. It can be used when two knights are fighting for the hand of the princess. At some point in the fight, one knight will say, "I yield" and let the victorious knight get the girl. Another way of using `yield` is to say that an apple tree might "yield" (produce) some fruit that you can take home and use in a pie. The latter meaning of `yield` best represents what `yield` does in Python programs.

A `yield` statement allows a function to "yield" a result. When a function yields a result, the value of the result is returned to the caller, but Python remembers the position reached in the function. The next time the function is called, the function resumes from the statement following the `yield`. A function that contains `yield` statements can return a sequence of values, each of which is returned by the next `yield` encountered when it runs. In other words, a function that contains `yield` statements can act as an iterator in a Python program.



## MAKE SOMETHING HAPPEN

### Investigating `yield`

The best way to find out about `yield` is to use it in some functions. Open the Python Command Shell in IDLE and enter the statements below followed by an empty line after the last `yield` statement.

[Click here to view code image](#)

```
>>> def mr_yield():
```

```
yield 1
yield 2
yield 3
yield 4
```

```
>>>
```

We've created a function called `mr_yield` that contains four yield statements. Each yield returns a value. We can use this function in a loop. Enter the following statements to create a loop that uses `mr_yield` as an iterator.

[Click here to view code image](#)

```
>>> for i in mr_yield():
 print(i)
```

When you enter an empty line after the `print` statement, the loop is performed:

[Click here to view code image](#)

```
>>> for i in mr_yield():
 print(i)

1
2
3
4
```

We can also use the results generated by `mr_yield` to create a list of the values that the function yields. Enter the following statement:

```
>>> list(mr_yield())
```

We've used the `list` function before. It accepts an iterator as an argument and uses the argument to generate a list of results produced by the iterator. Press Enter to call the `list` function and generate the output:

```
>>> list(mr_yield())
[1, 2, 3, 4]
```

## Create a test data generator using `yield`

We can use the `yield` keyword to make a very flexible test data generator for our Time Tracker application. Suppose that our customer for the Time Tracker application we developed in [Chapter 10](#) wants to see our application work with hundreds of contacts before she begins using it. This is a very sensible request for a customer to make. We might provide a program that will work well with ten people but will fail when we add a few more. We don't want to spend a week typing in fake contacts, so we decide to use our programming skills to create some test data.



### PROGRAMMER'S POINT

#### Using a program to create test data is a great idea

Many years ago, we set up a student project that used GPS (global positioning system) enabled mobile phones. The students had to make a program that would tell them when they were late for a lab session. The idea was that their application would tell the user when to start walking toward their lab. The application knew the locations of the phone and the lab.

Some students tested their programs by walking around the campus, but the smarter ones just walked around once, recorded the GPS locations of a few points, and then fed their programs the recorded locations during testing. Using this method saved them a lot of walking around. If you find yourself entering lots of test data into your programs, you should write a program to do that work for you. Remember that computers were created to spare us from drudgery, not create more of it.

```

EC12-05 Test contact generator

class Contact:

 def __init__(self, name, address, telephone):
 self.name = name
 self.address = address
 self.telephone = telephone
 self._hours_worked=0

 @staticmethod
 def create_test_contacts():
 phone_number = 1000000
 hours_worked = 0
 for first_name in ('Rob', 'Mary', 'Jenny', 'David', 'Chris', 'Imogen'):
 for second_name in ('Miles', 'Brown'):
 full_name = first_name + ' ' + second_name
 address = full_name + "'s house"
 telephone = str(phone_number)
 phone_number = phone_number + 1
 contact = Contact(full_name, address, telephone)
 contact._hours_worked = hours_worked
 hours_worked = hours_worked + 1
 yield contact

```

The diagram illustrates the execution flow of the `create_test_contacts` method. It starts with the assignment of `phone_number` to 1000000 and `hours_worked` to 0. It then enters a nested loop where it iterates through first names and second names. For each combination, it creates a full name, adds "'s house" to it, converts the phone number to a string, and creates a `Contact` object. This object has its `_hours_worked` attribute set to the current `hours_worked` value. After creating the contact, it increments the `hours_worked` value by 1. Finally, it yields the created contact. The annotations provide detailed explanations for each step:

- `create_test_contacts` is a static method
- Start the test phone numbers at 1000000
- Start the test number of hours at 0
- Loop through second names
- Create the full name
- Create the address
- Move on to the next phone number
- Add 1 for the next contact hours worked value
- Set the `hours_worked` for the result
- Create a Contact as the result
- Create a string for the telephone number
- Loop through first names
- Yield the contact that's been created

The code sample above shows a `Contact` class that contains a static function called `make_test_contacts`. The function creates 12 test contacts with names ranging from “Rob Miles” to “Imogen Brown.” It uses two tuples; one contains some first names and the other contains some last names. The function creates a contact using each combination of names and delivers it to the caller using a `yield` statement. Each contact created has an `hours_worked` value that is one hour larger than the previous one. The function uses a similar technique to generate different phone numbers for each of the contacts. The address string for each contact is created by adding `'s house` to the end of the name string. These contacts will provide a useful test set that we can use to demonstrate the program. If we want to create more contacts, we simply need to add more names to the first and second name tuples:

```
for first_name in
('Rob', 'Mary', 'Jenny', 'David', 'Chris', 'Imogen', 'Marilyn', 'Julie', 'Tim'):
 for second_name in ('Miles', 'Brown', 'Smith', 'Jones'):
```

Loop through  
second names

These two loops would create 36 contacts. We can use the `create_test_contacts` to generate an iterator that our programs can use.

```
for contact in Contact.create_test_contacts():
 print(contact.name)
```

Work through each contact  
Print some contact information

The two statements above would print out the name attribute of each test contact.

[Click here to view code image](#)

```
contacts = list(Contact.create_test_contacts())
```

This statement creates a list of contacts containing all the test contacts.



## CODE ANALYSIS

### Using `yield` to generate test contacts

You might have some questions about the preceding code.

**Question:** Why is `create_test_contacts` a static method?

**Answer:** We would not ask a specific `Contact` object to give us a set of test data. We don't want to have to create an instance of the `Contact` class just so we can get some test contacts. Instead, we should ask the `Contacts` class to do this.

**Question:** What is the difference between `yield` and `return`?

**Answer:** Good question. The `yield` statement causes Python to note the position reached in the function before it delivers the value to the caller. This position is then used to continue the function if another value is requested from the iterator.

A `return` statement just returns the value and informs the caller that the function is complete. As an example, consider the following:

[Click here to view code image](#)

```
def yield_return():
 yield 1
 yield 2
 return 3
 yield 4

for i in yield_return():
 print(i)
```

The function `yield_return` uses `yield` to return the first two values and then uses a `return` statement to return the third value. The use of `return` at this point has the effect of ending the iteration, which means the `yield` that delivers the value `4` is never reached. When we run the above program, it prints the following:

```
1
2
```

Note that the value `3` (which is returned by the `return`) is not used as part of the iteration.

**Question:** Does a function using `yield` have to `return`?

**Answer:** An iterator function using `yield` can run forever. Consider the following function:

[Click here to view code image](#)

```
def forever_tens():
 result = 0
 while True:
 yield result
 result = result + 10
```

The `forever_tens` function creates an iterator that will return an infinitely long iteration. Each value that it returns will be ten larger than the previous one.

[Click here to view code image](#)

```
for result in forever_tens():
 print(result)
 if result > 100:
 break
```

This loop displays the values in the iterator until the value of the result exceeds 100. It uses a condition and a break to stop the loop when this point is reached.

**Question:** What happens to local variables in a yielded function?

**Answer:** The variables local to the function are retained when the function yields a result. You see this in the function `forever_tens` above. Each time the iterator is asked for the next number in the sequence, the value of `result` has been retained from the previous time the method was active.

## Functions with an arbitrary number of arguments

We now know a lot about functions, but there might still be some things you find confusing about them. One thing we've used throughout this book is the ability

of the `print` function to accept different numbers of arguments:

[Click here to view code image](#)

```
print('Hello world')
print('The answer is', 42)
```

Both of these calls to the `print` function will work perfectly well, even though the first call has a single parameter and the second has two. When we've created and used functions, we've been very careful to make sure that the number of arguments given to a function call matches the number of parameters that the function has been defined to accept. In other words, when we've defined a function that accepts two parameters, we've made sure to provide two arguments when the function is called. However, the `print` function has been defined to accept an *arbitrary number* of arguments. In this context, the word *arbitrary* means a value that is not set when the function is defined. Let's find out how this works.



## MAKE SOMETHING HAPPEN

### Investigating arbitrary arguments

Let's see what we mean by arbitrary arguments by creating some functions. Open the Python Command Shell in IDLE and enter the statements below. Enter an empty line after the `return` statement.

[Click here to view code image](#)

```
>>> def add_function(x, y):
 return x + y
>>>
```

We've created a function called `add_function` that returns the sum

of two arguments. We can use this function to add two numbers. Enter the following text:

```
>>> add_function(1, 2)
```

When you press Enter, the function is called, and it returns the result we expect.

```
>>> add_function(1, 2)
3
```

Unfortunately, it is not possible to use this function to add more than two numbers. Try adding three numbers. Type in the following text:

```
>>> add_function(1, 2, 3)
```

When you press Enter, Python tries to match the arguments that were supplied to the parameters in the function definition. Unfortunately, they do not match, so an error is displayed.

[Click here to view code image](#)

```
>>> add_function(1, 2, 3)
Traceback (most recent call last):
 File "<pyshell#29>", line 1, in <module>
 add_function(1, 2, 3)
TypeError: add_function() takes 2 positional
arguments but 3 were given
```

Python makes sure that the function call matches the function definition and will produce an error if the two don't match. We can tell Python that a function accepts an arbitrary number of arguments by adding an \* (meaning multiple) to

the parameter description for the function. When the function is called, Python takes the supplied arguments and creates a list that is supplied to the method when it runs.

```
def add_function(*values):
 total = 0
 for value in values:
 total = total + value
 return total
```

A \* in front of the name means "arbitrary number"  
Set the total to the start value  
Work through the parameters supplied in the list  
Add the value to the total  
Return the total

The above version of `add_function` can accept any number of arguments, including no arguments at all. The function contains a `for` loop that works through the list supplied as a parameter and adds each value to the total. Below you can see the function being called with three arguments.

```
>>> add_function(1, 2, 3)
6
```

When the `add_function` is called, it can be given as many parameters as required. You can find a demonstration of `add_function` in the sample file **EG12-06 Arbitrary Arguments**.

A parameter denoting a list of arbitrary arguments can be preceded by a parameter that must be supplied to the function call. Perhaps we want to force a programmer to supply at least one value to a call to `add_function`. We can do this by adding a parameter, perhaps called `start`, which starts the list of values. This parameter is followed by a list of arbitrary length. The version of `add_function` below must be supplied with at least one parameter.

```
def add_function(start, *values):
 total = start
 for value in values:
 total = total + value
 return total
```

A \* in front of the name means "arbitrary number"  
Set the total to the value of the first parameter  
Work through the parameters supplied in the list  
Add the value to the total  
Return the total

The final situation we need to address is when a programmer already has a list of values she wants to feed into the method call.

```
numbers = (1, 2, 3, 4)
```

The tuple `numbers` defined above can't be given as an argument to `add_function` because it is already a list. Fortunately, we can ask Python to "unpack" the tuple and supply each value as an individual argument when the function is called. We do this by adding an asterisk (\*) character before the argument name:

[Click here to view code image](#)

```
>>> add_function(*numbers)
10
```

When `add_function` is called, each value in `numbers` is converted into an individual argument, which is passed into the function call.



### WHAT COULD GO WRONG

The \* character in function arguments can confuse C and C++ programmers

Python uses the \* character in a function argument to mean, "Please unpack the elements in this collection and pass them as individual arguments to the function call." Unfortunately, other programming languages, notably C and C++, use the \* character in a very different way, involving the use of pointers. If you already use C or C++, the \* character usage will likely confuse you in Python. Unfortunately, this is just something you'll have to sort out.

A much easier way of adding things in a list is accomplished with the `sum` function. We could rewrite the `add_function` as follows:

[Click here to view code image](#)

```
def add_function(*values):
 return sum(values)
```

## Modules and packages

In this section, we'll investigate how to build large Python applications out of individual source files. We've already used modules in our programs; now we'll discover more module features and how to use packages.

### Python modules

In [Chapter 7](#), we created some functions that made it very easy for us to read text and numbers. We used these functions extensively in the programs we've written since [Chapter 7](#). We stored the program code for these functions in a file called `BTCInput.py`. Programs that wanted to use these functions had to import that `BTCInput` file to use them. In Python, these files are called *modules*. Below, you can see the `BTCInput` module being used in a program that reads an age value from the user. The `read_int_ranged` function is defined in the `BTCInput` module. This function is given a prompt, a minimum value, and a maximum value. The `read_int_ranged` function then ensures that the value entered by the program user is in the specified range.

```
EG12-01 Using a BTCInput function

import BTCInput -> Import the BTCInput module

age = BTCInput.read_int_ranged(prompt='Enter age: ', min_value=5, max_value=95)
print('Your age is: ', age) -> Call a function from the module
```

Python obeys all the statements in a module when the module is imported. In the `BTCInput` module, these statements define the functions that the module provides. Modules are very useful. Anytime you want to create some code that can be used in several different programs, you can create a module file.

## Add a `readme` function to BTCInput

The BTCInput.py source file contains a set of functions provided by the module. The following code adds a `readme` function to these that describes the library itself:

[Click here to view code image](#)

```
def readme():
 print('''Welcome to the BTCInput functions
version 1.0

You can use these to read numbers and strings in
your programs.
The functions are used as follows:
text = read_text(prompt)
int_value = read_int(prompt)
float_value = read_float(prompt)
int_value = read_int_ranged(prompt, max_value,
min_value)
float_value = read_float_ranged(prompt, max_value,
min_value)

Have fun with them.

Rob Miles''')
```

The `readme` function above displays a string that describes each of the functions in the module. A programmer using this module can call the `readme` function to find out what the module does. Currently, the description is just a fixed string of text that's printed. In the section "Using pydoc" later in this chapter, we'll find out how to generate this description text from the function documentation strings we added to the beginning of each function, as described in [Chapter 7, "Add help information to functions."](#)

## Run a module as a program

There's nothing special about a Python module file. We can treat the BTCInput.py file as a Python program if we wish and run it. When we run the BTCInput.py module as a program, the Python statements in it are executed. We could add a call to the `readme` function to the module so that if the module is executed, the module will introduce itself.

[Click here to view code image](#)

```
all the BTCInput functions go here

Have the BTCInput module introduce itself once
the
functions have been defined
readme()
```

However, this statement will also be obeyed when the module is imported, which is something we don't want. Below, you can see the result of running **EG12-07 Using a BTCImport function** with our new “friendly” version of the BTCInput module. The `readme` message is displayed because when a module is imported, all the statements in the module are obeyed, including the `readme` call. This behavior can be useful if a module needs to perform some statements to set itself up, but in this case, the module is printing a message when we don't want it to do so.

```
RESTART: C:/Users/Rob/ EG12-01 Using a BTCImport function.py
Welcome to the BTCInput functions version 1.0
```

You can use these to read numbers and strings in your programs.

The functions are used as follows:

```
text = read_text(prompt)
int_value = read_int(prompt)
float_value = read_float(prompt)
int_value = read_int_ranged(prompt, max_value, min_value)
float_value = read_float_ranged(prompt, max_value, min_value)
```

Have fun with them.

Rob Miles

This is the output from the `readme` function

Enter age: 25

This is the output from the `read_int_ranged` function

Your age is: 25

## Detect whether a module is executed as a program

Creating a module that displays a friendly message when it's executed as a program is a good thing to do, but we don't want the message to be displayed when a program imports the module.

Fortunately, Python provides a special variable called `__name__` that a program can use to determine the name of the context in which the code is running. If a statement is running within an imported module, the value of `__name__` is set to the name of the module. If a statement is running within code started as a program, the value of `__name__` is set to the string '`__main__`'. We can use the `__name__` variable in the BTCInput module to make the module only display the readme information when it is run as a program.

[Click here to view code image](#)

```
if __name__ == '__main__':
 # Have the BTCInput module introduce itself
 readme()
```

Now the BTCInput module introduces itself only when it's running as a

program. If it's imported into another program, the readme is not displayed.



## MAKE SOMETHING HAPPEN

### Checking program context

We can investigate this module execution by using the BTCInput module and one of the example programs. You can find the BTCInput module and an example program in the code samples folder for this chapter. Start the IDLE program and use the **File, Open** commands to open the files **BTCInput.py** and **EG12-07 Using a BTCImport function.**

First, we'll run the BTCInput file as a program. Select the edit window for this file and select **Run, Run Module** to run the module as a program. You will see the friendly message displayed.

Now select the edit window for the sample program and run that. You'll see that the `read_int_ranged` method runs, but the friendly message is not displayed.

It is often very useful to make a module also function as a program. If a module performs a mathematical calculation, it could request some numbers and then display the result. Alternatively, the module could perform a set of demonstrations to show off what it can do.

## Create a Python package

A very large Python program might contain many different program files. In this section, we'll explore how Python packages work, and we'll learn some very powerful Python features along the way.

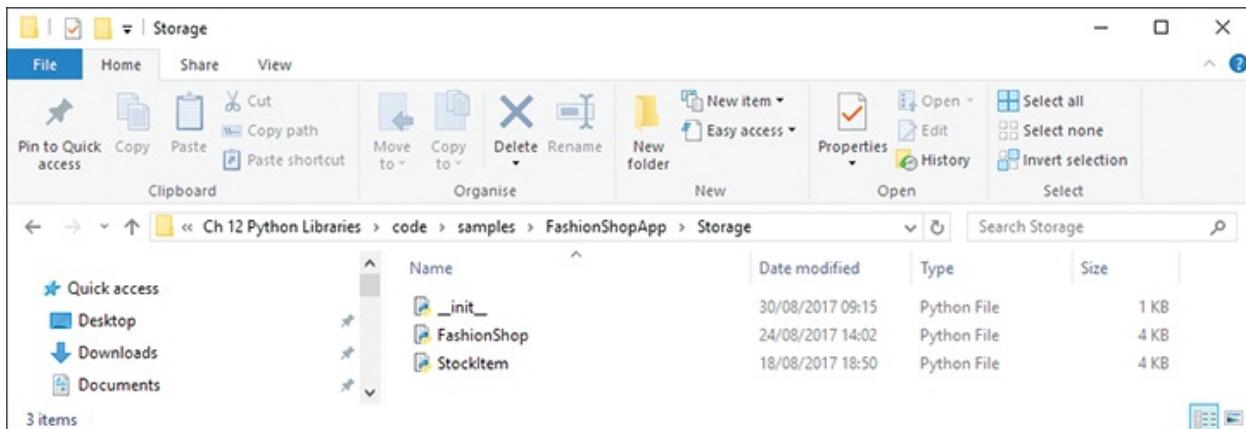
Until now, every Python program we've written has been stored in a single Python file. In [Chapter 8](#), we started using the module called BTCInput that we

used in many of our programs, but the programs themselves have been single files of code. It makes sense to put each Python class in an application into a different Python source file; doing so is convenient for multiple programmers working on a single application. Each class in the application imports the classes it needs to use, just like our programs have imported the BTCInput module whenever they needed to read information from the user.

As we learned in [Chapter 11](#) with our Fashion Shop application, we can divide the classes that implement the application into two distinct kinds. One kind of class stores the data, and the other talks to the program user. We can think of these as two different *packages* of classes.

Next, we'll learn how to create a Python package, which is quite easy. A Python package is simply a subfolder (or directory) on a storage device that contains a set of Python source files, one of which must be called `__init__.py`.

**Figure 12-2** shows a package I've created called Storage. The Storage package contains the `FashionShop` and `StockItem` classes for the Fashion Shop application.



**Figure 12-2** A Python package

The complete set of packages that I've created for the Fashion Shop application look like this:

[Click here to view code image](#)

```
FashionShopShellUI.py
```

```
ShellUI
 BTCInput.py
 FashionShopShell.py
 __init__.py
```

```
Storage
 FashionShop.py
 StockItem.py
 __init__.py
```

The ShellUI package contains the classes that generate the Python Command Shell user interface version of the Fashion Shop. Note that this package contains the shell program and the BTCInput package. The Storage package contains the data storage classes. Note that both folders also contain an `__init__.py` file. Currently, these are empty files.



## CODE ANALYSIS

### Making modules

You might have some questions about modules.

**Question:** How do I decide which module goes in which package?

**Answer:** It makes sense to put related classes into a single package. The two obvious distinctions we've used in the Fashion Shop include:

- User interface modules, which talk to the user, go into one package.
- Data storage modules, which hold the actual Fashion Shop data, go into another package.

**Question:** What does the `__init__.py` file in a package do?

**Answer:** The `__init__.py` file provides a way for a programmer to gain control when a package is loaded. The Python statements in the `__init__.py` are obeyed when the package folder is first opened. You can think of this as an initializer for packages. The `__init__.py` file can set up resources and could contain a help string that describes the package contents.

## Import modules from packages

A program can import a module from a package. The statement below imports the `FashionShop` module from the `Storage` package:

[Click here to view code image](#)

```
from Storage import FashionShop
```

The effect of this statement is to make the items in the `FashionShop` module available for use by our program. Now that we have the `FashionShop` module available in our program, we can use the classes that are defined within it:

[Click here to view code image](#)

```
shop = FashionShop.FashionShop
```

This statement looks a bit confusing. It's an assignment, but what is it assigning? Above, we're using the `FashionShop` class from the `FashionShop` module. We are then setting the variable `shop` to refer to this class. If we had placed the `FashionShop` class in a module called `DiskStorage` (something we could have done), the statement would look like this:

[Click here to view code image](#)

```
shop = DiskStorage.FashionShop
```

The value being assigned to `shop` is a reference to the `FashionShop` class, which is something we've never done before. We've assigned strings, integers, and functions to variables, but we have never assigned a class to a variable. Let's take a closer look at how this works, and what it means.



## MAKE SOMETHING HAPPEN

### Use classes as values

Using classes as values made my head hurt the first time I heard about it, but I discovered that it's worth learning. We can do some experiments to see how this works using the Python Command Shell in IDLE. Enter the statements below, and enter an empty line after the `print` statement.

[Click here to view code image](#)

```
>>> class VarTest:
 def __init__(self):
 print('making a VarTest')

>>>
```

We've created a class called `VarTest` that contains an initializer, which just prints a message. Let's see what happens if we create an instance of `VarTest`. Type in the following statement that creates an instance of `VarTest` and makes the variable `x` refer to it.

```
>>> x = VarTest()
```

When you press Enter, the `__init__` method in the `VarTest` class is performed, and it prints the message we would expect.

```
>>> x = VarTest()
making a VarTest
>>>
```

This result is not terribly surprising. We created a class and then made an object from that class. However, let's do something slightly different. Type in the following statement and press Enter.

```
>>> y = VarTest
```

This code looks very much like the assignment of the variable `x` that we just performed, but with one significant difference. There are no parentheses after the name of the class. In this assignment, the *value* of `VarTest` is assigned to `y`, which means `y` is now a variable that refers to a class. We can use the variable `y` in the same way as we would use `VarTest`. For example, we can make a new instance of `y`.

```
>>> z = y()
making a VarTest
>>>
```

When the reference `y` is followed, it sends Python to the `VarTest` class, and by creating an instance of `y`, makes a `VarTest` instance.

A program can treat class references in the same way as any other type of data. We can make lists of class references, store them in dictionaries, and pass them as arguments to function calls. If we revisit the statement below, we see that the variable `shop` has been set to refer to the `FashionShop` class that manages the Fashion Shop data items.

[Click here to view code image](#)

```
shop = FashionShop.FashionShop
```

Next, we'll show Python's power by passing this `shop` reference into the class that controls our Fashion Shop user interface. Using Python in this way is the ultimate expression of component-based software design. We have created a complete component—the `FashionShop` class—that manages all the data storage needs of the fashion shop. We can give this component to the user interface component that provides a means for the user to interact with the data that this component manages.



## CODE ANALYSIS

### References to classes

References to classes is very strong, and potentially confusing, programming practice.

**Question:** What are the benefits of using class references?

**Answer:** We don't have to use class references in this way. We could create an application that has the data storage and the user interface in a single class. However, in the next chapter, we'll create a Fashion Shop manager program that has a GUI (graphical user interface). We can pass the GUI a `FashionShop` class to use for data storage.

If we create a version of the `FashionShop` class that uses a database to store the data, we can pass that class into the user interface to make a version of the program that works with databases. We've talked about objects as components, and these abilities are how, with the proper design, we can swap elements of a system very easily.

You can think of the association between the `FashionShop` class and a user interface class as a bit like the connection between a monitor and a PC. Because both the PC and the monitor use the same connection standard, we can swap the device at either end of the connection, and the system will still work.

Now we need to work out how to use the classes in these packages to make our application work. The file `FashionShopShellUI.py` is the program we'll run to make the Fashion Shop work. The file `FashionShopShellUI.py` will create a user interface and display it on the screen. Below, you can see the first part of the program. It creates two variables. The variable `ui` refers to the user interface class we'll use; the variable `shop` refers to the `FashionShop` storage class we'll use.

[Click here to view code image](#)

```
Loads a user interface class and a data manager
class
and then uses these to create an application

Get the module containing the user interface
class
from the ShellUI package
from ShellUI import FashionShopShell # Get the user
interface manager class from this module
ui = FashionShopShell.FashionShopShell

Get the module containing the data storage class
from the Storage package
from Storage import FashionShop
Get the data manager class from the storage
module
shop = FashionShop.FashionShop
```

Now that we have our user interface and our two shop classes, we need to build an application from them, which is much like plugging a monitor into a PC.

```
app = ui(filename='simplefashionshop.pickle', storage_class=shop)
```

Create the user interface

This statement creates an instance of the class to which `ui` is referring. It passes two parameters to the initializer for this class: the name of the file that contains the shop, and the storage class that will manage this data. Remember that the variable `ui` refers to the `FashionShopShell` class, so this is the class that is actually created:

```
class FashionShopShell:

 def __init__(self, filename, storage_class):
 self.__filename = filename
 try:
 self.__shop = storage_class.load(filename)
 except:
 print('Fashion shop not loaded.')
 print('Creating an empty fashion shop')
 self.__shop = storage_class()
```

Called to set up a class instance  
Copy the file name for later use  
Call the load method on the supplied class to try and load the data  
If the load fails, create an empty fashion shop

The `FashionShopShell` initializer method `__init__` creates an attribute called `__shop` for which the `FashionShopShellUI` will provide a user interface. In the next chapter, we'll create a `FashionShopGUI` class with an initializer that is initialized in the same way but provides a graphical user interface.

Now that we have our user interface running, we must next ask it to display the main menu for the application.

[Click here to view code image](#)

```
Now call the main_menu function on the app
app.main_menu()
```

The example application in the folder **EG12-08 FashionShopApp** in the samples folder for this chapter implements the Fashion Shop application using modules and packages. You can open the Fashion Shop by running the program **FashionShopShellUI**.



## PROGRAMMER'S POINT

### Using classes as values is extremely strong programming magic

We're using a programming technique with the wonderful name of *dependency injection*. The idea behind this is that we inject something (in our case, the `FashionShop` storage class) into another class (in this case, the user interface class) to give the second class something with which to work. In the Fashion Shop application, we can change the class we're injecting (or "passing into the constructor") into the user interface class to make the user interface work with different types of storage devices.

If you're confused about all this (quite understandably), just remember why we're doing this. We want to be able to use the `FashionShop` storage class with both the command shell and GUIs. Also, we want to be able to switch our storage system for a different one (which perhaps uses databases) without having to change the behavior of the user interface classes. By using references to classes, we can create a program that "wires up" one class to another to build an application that functions the way that we want.

## Program testing

Some programmers don't seem to know what the word "testing" really means. They will say things like, "I've tested the program, and it seems to work." What this usually means is that they've run the program, entered a few things, looked at the results, and decided that everything is working properly. To me, this is a bit like saying that they've tested a new airplane by counting how many wings it has and have gotten the number they were expecting. One of the things that separates "professional" software from "ordinary" software is the amount of testing that it should have. In other words, if you want to sell your software, you should ensure that you have properly tested it.

When engineers test a new plane, they do a lot more than just count the wings. Aircraft manufacturers have a very highly developed set of tests that they perform on each plane they make. The tests are:

- Repeatable (the same tests are performed each time on each plane)
- Documented (they produce a detailed test report for each aircraft)
- Automatic (they have built machines to automate as much of the testing as

possible)

In this section, we'll look at Python features that make it possible to create tests for our applications. Let's make some tests for the `StockItem` class from the Fashion Shop application. We want to make sure that new `StockItem` objects are created with the correct initial settings, attempts to create invalid `StockItem` objects are rejected, and that the `StockItem` itself works correctly.

We could test these things by hand, entering values into the program and observing what happens. However, this would quickly get very tedious, particularly when we remember that we must perform all our tests after *every* change to the `StockItem` class. If we add a new feature or fix a fault, we must repeat all the tests because one of the main causes of faults in programs is fixes for other faults.

Writing our own code that automatically tests our applications makes more sense. The code below creates a new `StockItem` and then checks the `stock_level` value to make sure that it has been initialized with the value zero. If this is not the case, an exception is raised.

```
item = StockItem(stock_ref='Test', price=10, tags='test:tag') —— Make a test StockItem
if item.stock_level != 0: —— Is the stock level zero?
 raise Exception('Initial stock level not 0') —— If the stock level is not zero,
 raise an exception
```

The idea is that we will have lots of tiny tests like the one above, and then repeat these tests each time the program is updated. If we were making a very large application, we'd run all the tests like these on the code in the system automatically every day, perhaps overnight. In some languages, you would have to perform your testing using constructions like this. However, Python provides a few extra tricks that we can use to make testing programs as easy as possible.

## The Python `assert` statement

Python provides a language construction called `assert` that programs can use to test themselves as they run. In English, the word “assert” means “stating firmly something that you believe to be true.” I could use the `assert` mechanism to perform the previous test of the `StockItem` class:

```
item = StockItem(stock_ref='Test', price=10, tags='test:tag')
assert item.stock_level == 0
```

Make a test StockItem

Assert that the initial stock level is zero

The `assert` statement is followed by a Boolean value that you believe to be true for the program. In the above code, I'm saying that the `stock_level` attribute of the object referred to by `item` should be zero. If the assertion succeeds, (in other words, the value of `stock_level` is indeed zero), the program continues. If the assertion fails, the program stops with an error:

[Click here to view code image](#)

```
Traceback (most recent call last):
 File "C:/Users/Rob/Test Program.py", line 20, in
<module>
 assert item.stock_level == 0
AssertionError
```



## CODE ANALYSIS

# Python Assertions

**Question:** How many `assert` statements can a program contain?

**Answer:** You can have as many assertions as you like.

**Question:** Does the program continue after an assertion has failed?

**Answer:** The program will not continue if an assertion fails. When an exception is raised the program stops running. A program can use the `try...except` construction to catch assertion failures, so a program could display a message to indicate that something had failed.

## The Python unittest module

You can add `assert` statements to your programs to make you more confident that they are working properly. However, Python also provides a framework that makes it easy to create lots of small tests and run them. Each of the tests is called a *unit test* because it's designed to perform a small test on one particular unit in the program. The idea behind unit tests is that programmers develop the tests for their components at the same time they create them. In fact, there's also a form of development called *test driven development* in which the tests are created before the actual components. Code is then added to the components so that they pass the tests.

The `unittest` module is provided as part of a standard Python installation. We can import the `unittest` module into a program in the usual way:

```
import unittest
```

The `unittest` module contains a class that serves as the superclass of any test classes we create. The idea is that we create a class that is a subclass of the test class and fill it with methods that perform unit tests. Code in the `unittest` module will automatically run the unit test methods for us and report on the test results.

```
class TestShop(unittest.TestCase):

 def test_StockItem_init(self):
 item = StockItem(stock_ref='Test', price=10, tags='test:tag')
 self.assertEqual(item.stock_ref, 'Test') Make a test object
 self.assertEqual(item.price, 10) Was the name set correctly?
 self.assertEqual(item.stock_level, 0) Was the price set correctly?
 self.assertEqual(item.tags, 'test:tag') Is the initial stock level zero?
 Was the tag set correctly?
```

The `TestShop` class above is defined as a subclass of the `TestCase` class, which is found in the `unittest` module. The class contains a method called `test_StockItem_init`, which has been created to ensure that new `StockItem` values are created with the correct initial values.

The method `assertItemEqual` performs the testing of the code. It works similarly to the `assert` statement we saw above. The method is supplied with two arguments. If the arguments are equal, the test passes. The `test_StockItem_init` method ensures that the arguments are correctly passed into a new `StockItem` and that the initial stock level of the item is zero. We can run the tests simply by calling `main` in the `unittest` module:

```
unittest.main()
```

The unittest framework finds all the classes that have been created and searches each class for method attributes that have names beginning with `_unittest` and then calls each of the methods in turn. If an `assertItemEqual` method fails (in other words, the two items in the method are not equal), that test method is abandoned, and an error is reported for that test. Once the test methods complete, the framework produces a brief report:

[Click here to view code image](#)

```
.
```

---

```
Ran 1 test in 0.037s
```

OK

If you want a bit more detail in the report, you can select a more verbose level of output:

```
unittest.main(verbosity=2)
```

The report now contains the name of each method tested:

[Click here to view code image](#)

```
..
test_StockItem_init (__main__.TestShop) ... ok
```

```


Ran 1 test in 0.013s
```

```
OK
```

We can discover what happens when a test fails by creating a test that's guaranteed not to work. We could assert that `1` is equal to `0`:

[Click here to view code image](#)

```
def test_that_fails(self):
 self.assertEqual(1, 0)
```

If we add the above method to the test class and rerun the tests, we get output that identifies the failed statement and the error:

[Click here to view code image](#)

```
.F
=====

FAIL: test_that_fails (__main__.TestShop)

Traceback (most recent call last):
 File "C:/Users/Rob/OneDrive/Begin to code Python/
tinytest.py",
 line 16, in test_that_fails
 self.assertEqual(1, 0)
AssertionError: 1 != 0

```

Ran 2 tests in 0.008s

FAILED (failures=1)

The following table shows all the possible assertions that a program can use to test the behavior of statements in a program.

| TEST FUNCTION        | TEST ACTION                                   | EXPLANATION                                                                                                                                                                                               |
|----------------------|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| assertEqual(a, b)    | Asserts that a is the same as b               | Use this to test two values to see whether they are the same                                                                                                                                              |
| assertNotEqual(a, b) | Asserts that a is not the same as b           | Use this to test two values to see whether they are identical                                                                                                                                             |
| assertTrue(b)        | Asserts that the b is True                    | Use this to test whether a Boolean expression or value is True                                                                                                                                            |
| assertFalse(b)       | Asserts that b is False                       | Use this to test whether a Boolean expression or value is False                                                                                                                                           |
| assertIs(a, b)       | Asserts that a and b refer to the same object | Use this to test whether two references refer to the same object. Note that this is not the same as assertEqual, as assertEqual will return True if the objects are different but contain the same value. |

|                                       |                                                      |                                                                                                                                      |
|---------------------------------------|------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <code>assertIsNot(a, b)</code>        | Asserts that a and b do not refer to the same object | Use this to test whether references do not refer to the same object                                                                  |
| <code>assertIsNone(r)</code>          | Asserts that variable r is None                      | Use this to test whether a variable has been explicitly set to the None value                                                        |
| <code>assertIsNotNone(r)</code>       | Asserts that variable r is not None                  | Use this to test whether a variable is not set to the value None                                                                     |
| <code>assertIn(a, b)</code>           | Asserts that a is in b                               | Use this to test whether a value is in a collection (list or tuple) of items                                                         |
| <code>assertNotIn(a,b)</code>         | Asserts that a is not in b                           | Use this to test whether a value is not in a collection (list or tuple) of items                                                     |
| <code>assertIsInstance(a,b)</code>    | Asserts that a is an instance of type b              | Use this to test that a reference refers to an object of a particular type. You could test whether a reference refers to a StockItem |
| <code>assertNotIsInstance(a,b)</code> | Asserts that a is not an                             | Use this to test whether a reference does not refer to an object of a particular type                                                |

instance  
of type b

## Test for exceptions

We can use the assertions above to check values in our test functions, but there is one other thing for which we would like to test. We'd like to be able to test whether a method raises an exception when things go wrong. Consider the following code:

```
item = StockItem(stock_ref='Test', price=10, tags='test:tag')
item.add_stock(-1)
```

Create a test StockItem

Try to add -1 items to stock, which should fail

The first statement creates a `StockItem`. The second statement tries to add `-1` to the stock level of this item. This is not a valid action (we can't add -1 items to stock), and the `add_stock` method raises an `Exception` to indicate that it's being used incorrectly. We need to create a test to prove that the exception was raised, but this involves quite a few statements. Fortunately, the `unittest` framework provides an easy way to test for this behavior:

[Click here to view code image](#)

```
with self.assertRaises(Exception):
 item.add_stock(-1)
```

The test uses the Python `with` construction, which is a way of creating objects used for a specific action. We first saw the `with` construction in [Chapter 8](#) when we used it to manage file access. The `assertRaises` method uses `with` to create an action that can catch an `Exception` that should be raised. Below, you can see the result of the exception not being raised. I've modified the test so that it adds 1 item to stock, which will not cause an exception. Adding -1 to the stock will cause the test to fail.

[Click here to view code image](#)

F.

```
=====
```

```
FAIL: test_StockItem_add_stock (__main__.TestShop)
```

```

```

```

```

```
Traceback (most recent call last):
```

```
 File "C:\Users\Rob\OneDrive\Begin to code
```

```
 Python\Part 2 Final\Ch 12
```

```
 Python Libraries\code\samples\EG12-09
```

```
 TestFashionShopApp\tinytest.py",
```

```
 line 25, in test_StockItem_add_stock
```

```
 item.add_stock(1)
```

```
AssertionError: Exception not raised
```

```

```

```
Ran 2 tests in 0.049s
```

```
FAILED (failures=1)
```

## Create tests

The advantage of using the `unittest` framework is that it's very easy to add new tests at any time. Creating the tests also forces you to think about the behavior of the application in detail and ponder what should happen in certain circumstances. The results of your tests often mean that you then must ask the customer for clarification. For example, you would have to ask the fashion shop owner, “What is the largest possible number of items that you ever add to stock at once?” You need this information so that you can write a test for that behavior.

Once all the initial values of a `StockItem` have been tested, we can create some code that tests specific behaviors of the system. The `test_StockItem_sell_stock` method below creates a `StockItem`, adds ten items to stock, checks that ten items were recorded, sells two items, and makes sure that eight items remain in stock.

```

def test_StockItem_sell_stock(self):
 item = StockItem(stock_ref='Test', price=10, tags='test:tag')
 self.assertEqual(item.stock_level, 0) Create a test StockItem
 item.add_stock(10) Make sure that the initial stock level is 0
 self.assertEqual(item.stock_level, 10) Add 10 items to stock
 item.sell_stock(2) Make sure that 10 items are now in stock
 self.assertEqual(item.stock_level, 8) Sell 2 items
 Make sure there are 8 items left in stock

```

If you consider how long it would take you to perform this transaction by hand, you can start to see how useful it is to implement automatic testing like this. The `unittest` framework includes lots of other features you can explore. You can create methods to set up and “tear down” test scenarios. You can find out more about the framework here: <https://docs.python.org/3.6/library/unittest.html>.

The example FashionShop application in the folder **EG12-09 TestFashionShopApp** contains a Python program called RunTests that runs all the tests in the preceding code block. You might like to add some more tests. For example, you might want to add a test to make sure that the `StockItem` raises an exception if a program tries to sell more items than there are in stock.

Testing should be driven by the specification. In [Chapter 7](#), when we were creating the user interface for the Theme Park Ride Selector program, we spent some time considering sensible ways of testing the behavior of the program. You might find it interesting to consider how you would take the tests described in [Chapter 7](#) and perform them automatically.



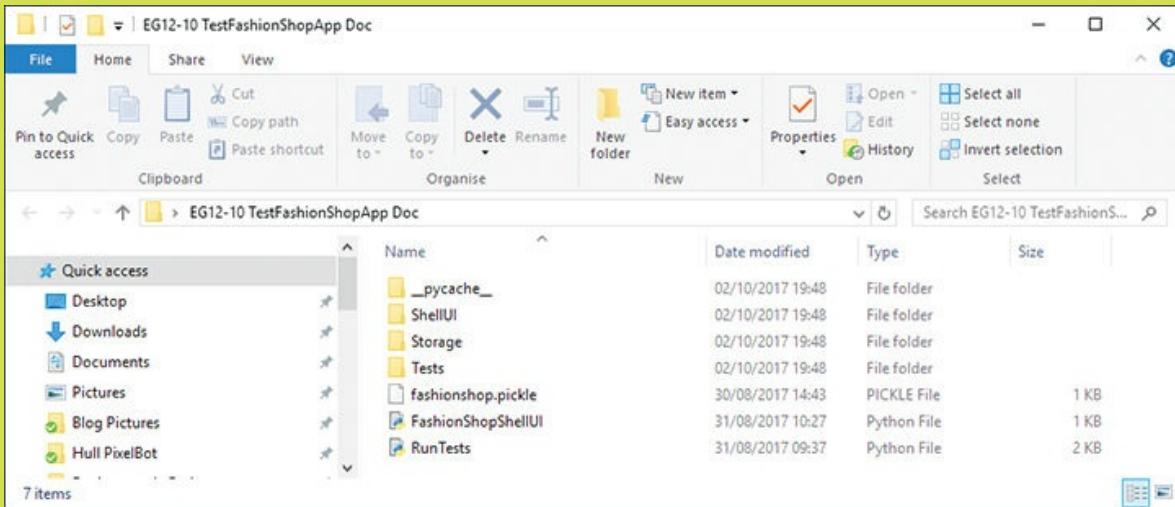
## PROGRAMMER'S POINT

### Tests prove only the existence of faults

Tests are a vital part of professional software development. If you want other people to use your program, you should make sure that you put some effort into testing it. However, passing a set of tests doesn't mean that a program is good. It just means that it passes the tests. You can never claim that your program is free of bugs just because it passes all your tests. Automated tests are a very important part of software development, but you should also make sure that you form the habit of looking through your code (called “code reviews”) and testing the program with users.

## View program documentation

When we began creating functions, we saw that we could add documentation strings to a function or method. We can also add documentation strings to classes and modules. These strings must be the first statement in the file. Python provides a tool called pydoc that can be used to work through modules and classes and extract this documentation. Pydoc can be used to create a website that allows programmers to browse the documentation. Let's look at what we can do with pydoc and the fully documented FashionShop application.



The screenshot shows a Windows File Explorer window titled "EG12-10 TestFashionShopApp Doc". The window displays a list of files and folders within the folder. The columns are "Name", "Date modified", "Type", and "Size".

| Name               | Date modified    | Type        | Size |
|--------------------|------------------|-------------|------|
| __pycache__        | 02/10/2017 19:48 | File folder |      |
| ShellUI            | 02/10/2017 19:48 | File folder |      |
| Storage            | 02/10/2017 19:48 | File folder |      |
| Tests              | 02/10/2017 19:48 | File folder |      |
| fashionshop.pickle | 30/08/2017 14:43 | PICKLE File | 1 KB |
| FashionShopShellUI | 31/08/2017 10:27 | Python File | 1 KB |
| RunTests           | 31/08/2017 09:37 | Python File | 2 KB |

The left sidebar shows a "Quick access" list with links to Desktop, Downloads, Documents, Pictures, Blog Pictures, and Hull PixelBot. The title bar of the window says "EG12-10 TestFashionShopApp Doc".

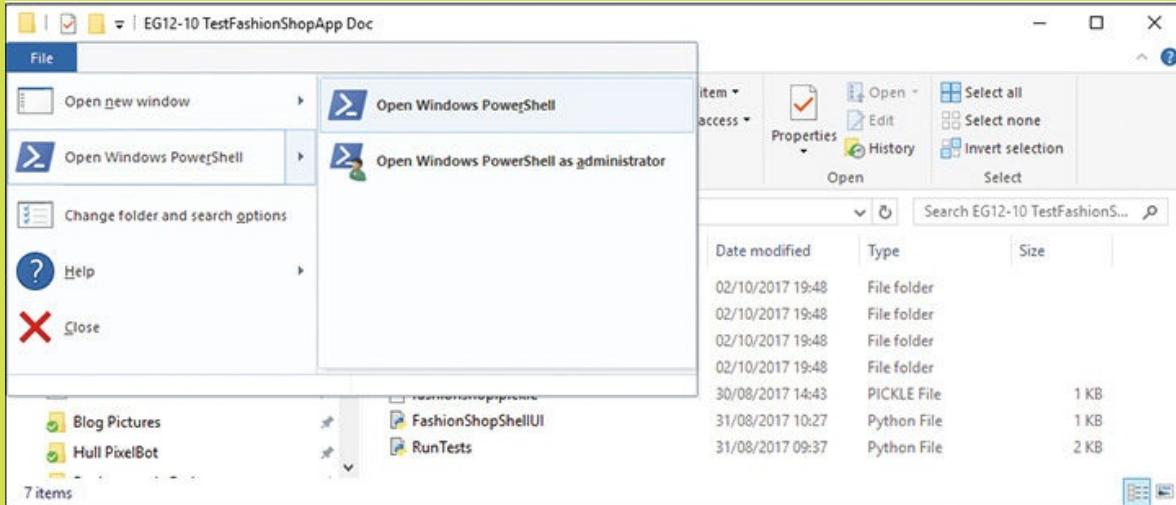
## MAKE SOMETHING HAPPEN

## Explore pydoc

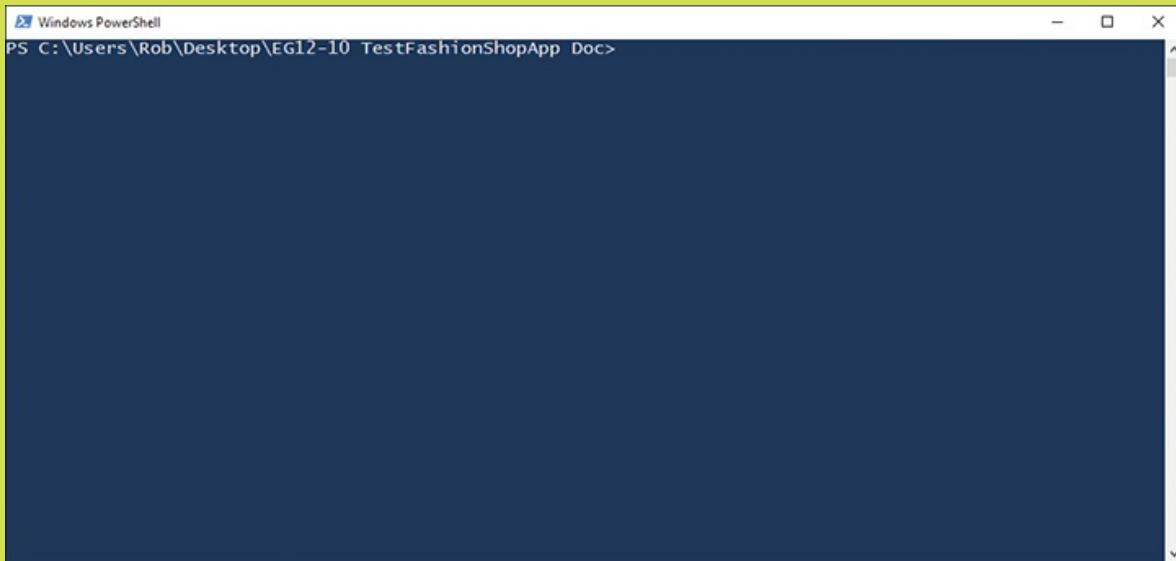
The pydoc program is written in Python. The best way to run pydoc is from the command prompt. We'll use the Windows PowerShell command prompt to run pydoc in these examples. The commands are the same on Apple and Linux machines, but you should use the Terminal to perform this exercise on those operating systems.

The first thing to do is find the folder that contains the sample application for this exercise. The folder is called **EG12-10 TestFashionShopApp Doc**. I copied this file onto the desktop of my machine for the practical work:

Now we need to open a PowerShell prompt in this folder. The best way to do this is to click **File** in the top left corner and then select **Open Windows PowerShell** from the window that appears.



This should cause the PowerShell to open on your desktop:



As its name implies, the PowerShell program is a shell like the Python Command Shell. However, this shell is wrapped around the Windows operating system rather than around Python. We can type commands and view responses from Windows. We'll use the PowerShell to ask Python to run the pydoc module. Enter the following command:

[Click here to view code image](#)

```
PS C:\Users\Rob\Desktop\EG12-10
TestFashionShopApp Doc> python -m pydoc
```

The Python command is followed by the option `-m`, which means “execute the following module.” The module we want to run is `pydoc`. If we don’t give `pydoc` any options, it tells us about itself:

[Click here to view code image](#)

```
pydoc - the Python documentation tool

pydoc <name> ...
 Show text documentation on something.
<name> may be the name of a
 Python keyword, topic, function, module,
or package, or a dotted
 reference to a class or function within a
module or module in a
 package. If <name> contains a '\', it is
used as the path to a
 Python source file to document. If name
is 'keywords', 'topics',
 or 'modules', a listing of these things
is displayed.

pydoc -k <keyword>
 Search for a keyword in the synopsis
lines of all available modules.

pydoc -p <port>
 Start an HTTP server on the given port on
the local machine. Port
 number 0 can be used to get an arbitrary
unused port.

pydoc -b
 Start an HTTP server on an arbitrary
```

unused port and open a Web browser to interactively browse documentation. The `-p` option can be used with the `-b` option to explicitly specify the server port.

`pydoc -w <name> ...`

Write out the HTML documentation for a module to a file in the current directory. If `<name>` contains a '\', it is treated as a filename; if it names a directory, documentation is written for all the contents.

We want to use pydoc to create a webpage that we can view. We can use the `-b` option to do this. Enter the following command into PowerShell:

[Click here to view code image](#)

```
PS C:\Users\Rob\Desktop\EG12-10
TestFashionShopApp Doc> python -m pydoc -b
```

This will cause pydoc to start, create a web server on a port on your machine, and then open your default browser to show you this webpage. On my machine, I get the following display:

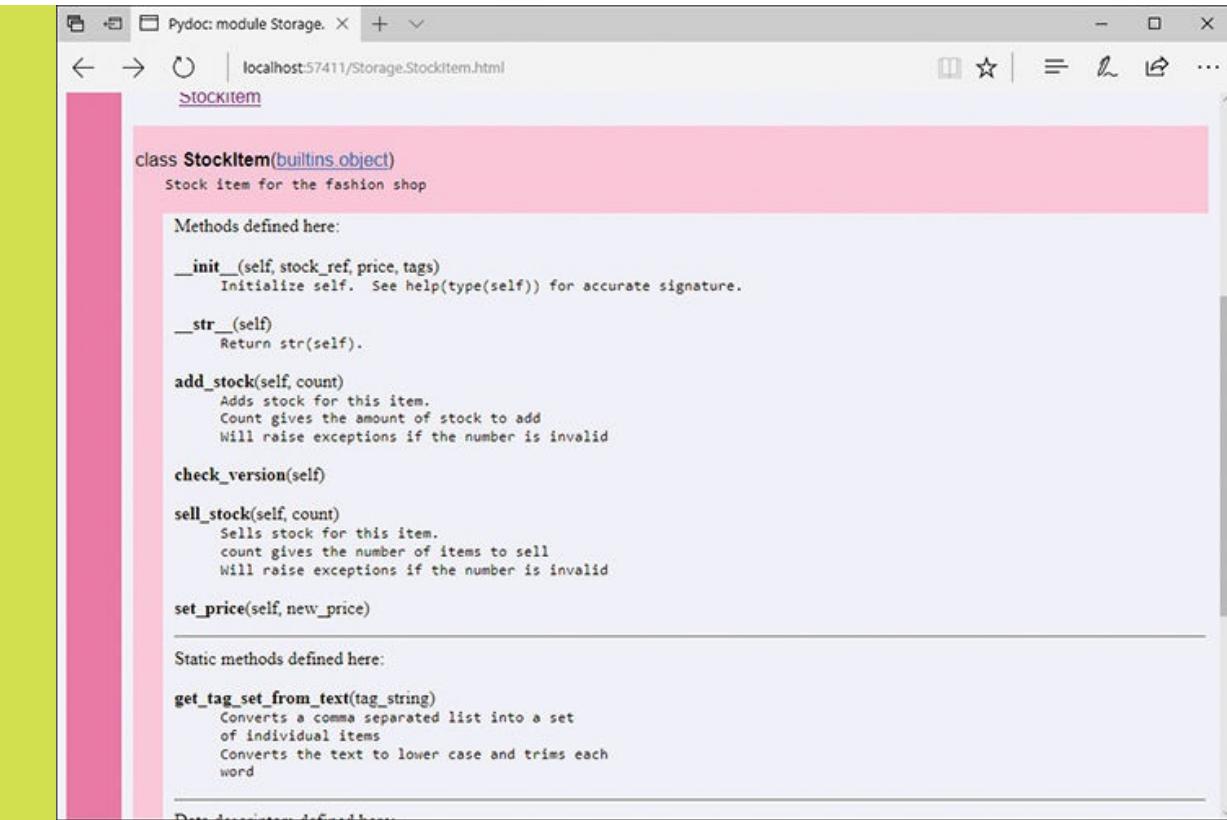
The screenshot shows a web browser window titled "Pydoc: Index of Module" at "localhost:57411". The page displays the "Index of Modules" for Python 3.6.1. The "Built-in Modules" section lists numerous standard library modules in two columns. At the bottom of the page, there are links for "FashionShopShellUI", "RunTests", "ShellUI (package)", "Storage (package)", and "Tests (package)".

| ast<br>biseect<br>blake2<br>codecs<br>codecs_cn<br>codecs_hk<br>codecs_iso2022<br>codecs_jp<br>codecs_kr<br>codecs_tw<br>collections<br>csv<br>datetime<br>functools<br>heapq<br>imp | io<br>json<br>locale<br>lsprof<br>md5<br>multibytecodec<br>opcode<br>operator<br>pickle<br>random<br>sha1<br>sha256<br>sha3<br>sha512<br>signal<br>sre | stat<br>string<br>struct<br>symtable<br>thread<br>tracemalloc<br>warnings<br>weakref<br>winapi<br>array<br>atexit<br>audioop<br>binascii<br>builtins<br>cmath<br>errno | faulthandler<br>gc<br>itertools<br>marshal<br>math<br>mmap<br>msvcrt<br>nt<br>parser<br>svs<br>time<br>winreg<br>xxsubtype<br>zipimport<br>zlib |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|

You can click on the links to view the documentation for the built-in modules, but the pyhelp program also searches the current folder for Python modules and programs. Along the bottom of the page, you can see familiar package names from the FashionShop. Click **Storage (package)** in the middle of the bottom row to open it.

The screenshot shows a web browser window titled "Pydoc: package Storage" at "localhost:57411/Storage.html". The page displays the "Storage" package contents for the Fashion Shop application. It shows the file path "c:\users\rob\desktop\eg12-123\testfashionshopapp doc\storage\\_\_init\_\_.py" and a "Package Contents" section with links for "FashionShop" and "StockItem".

Now, you navigate to the contents of this package. If you click **StockItem**, you can see the help for that module.



The help strings we created when we wrote the methods are now displayed next to the method names for the class. You can browse through the **FashionShop** object (and also the Python objects) to get help. When finished, close the browser window, go back to the PowerShell window, and enter the command **q** and press Enter to stop the pydoc server.

[Click here to view code image](#)

```
PS C:\Users\Rob\Desktop\EG12-10
TestFashionShopApp Doc> python -m pydoc -b
Server ready at http://localhost:57591/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
```

The pydoc program is a good example of a Python program that hosts a website. We'll do this ourselves in [Chapter 14](#).



## WHAT COULD GO WRONG

### Modules that can run as programs can break pydoc

Earlier in this chapter, in the section “Running a module as a program,” we saw that a module (a file of Python classes that we want to import into other programs) can contain Python statements that are executed when the module is loaded. The pydoc program loads all the modules it finds, which will result in the programs in them being executed within pydoc.

This behavior caused big problems for me when the Fashion Shop started running as the documentation was being created. In the section “[Detect whether a module is executed as a program](#),” we saw how code in a module can detect whether it is being run as a program. Make sure all your program modules contain this test. Otherwise, you might find that your programs start running when you don’t expect them to. In other words, the program that runs the Fashion Shop now looks like this:

[Click here to view code image](#)

```
'''
Starts a Fashion Shop running with a Python
Command Shell user interface

Runs only if it is started as the main
program
if __name__ == '__main__':
 # Loads a user interface class and a
 # storage manager class
 # and then uses these to create an
```

## What you have learned

This has been a difficult chapter. You learned a lot, but you've also transformed the FashionShop application from a nice piece of sample code into something that could be the basis of a professional development. You've added proper testing and discovered how to add documentation to the application and view that documentation via your browser.

From a programming perspective, you've learned how powerful function references are, and how you can use these to add flexibility to the code you write. You also learned about lambda expressions, and how these can be used to generate descriptions of actions that can be passed around as data. As an additional function feature, you learned the use of the yield statement in a function that allows the function to be used as the source of an iterator.

You also discovered how to create packages of Python modules. These packages allow multiple programmers to work on different elements in a program solution. You were also introduced to variables that contain class references. These allow for classes to be used as components in applications in a very flexible way.

Here are some points to ponder about what we have learned.

### Is everything in Python an object?

Yes, it is. Even things that you might not expect to be objects. Everything from an integer to a class containing many attributes can be treated as an object. Even a function can be treated as an object to which you can add attributes (although I'd need a very good reason to do anything that strange). The fact that we can assign anything to a variable and pass it around might seem scary, but it's also very powerful.

Problems come when your programs start to combine objects in ways that don't work. Perhaps a program will try to call a method attribute on an object that

doesn't have that attribute. In this case, your program will fail when it runs, which is why testing is such an important part of Python program development. You must test all possible behaviors of your program to ensure that all program elements are used correctly.

### **Should I feel bad if I don't yet understand things like lambda expressions and yield?**

No. It takes a while to understand the value of language features like these. The Fashion Shop and Time Tracker applications give context for these language features, but they can still be confusing if you're just learning to code.

Part of the problem with learning how to use these features is that when you're learning how to program you haven't experienced many situations in which such features would be useful. It's a bit like being told how useful gears are on a bike when you've never had to ride a bike up a steep hill. Just keep coming back to those sections and read them again and again as you write more code. At some point, you'll discover a use for one of them, and at that point, you'll understand what they are used to accomplish.

### **What happens when I move Python packages from one computer to another?**

Python is a very portable language. I would expect the programs would just work in their new home. You need to make sure that the version of Python on the new machine is the correct one, but you shouldn't have to make any changes to your program to accommodate a different host computer.

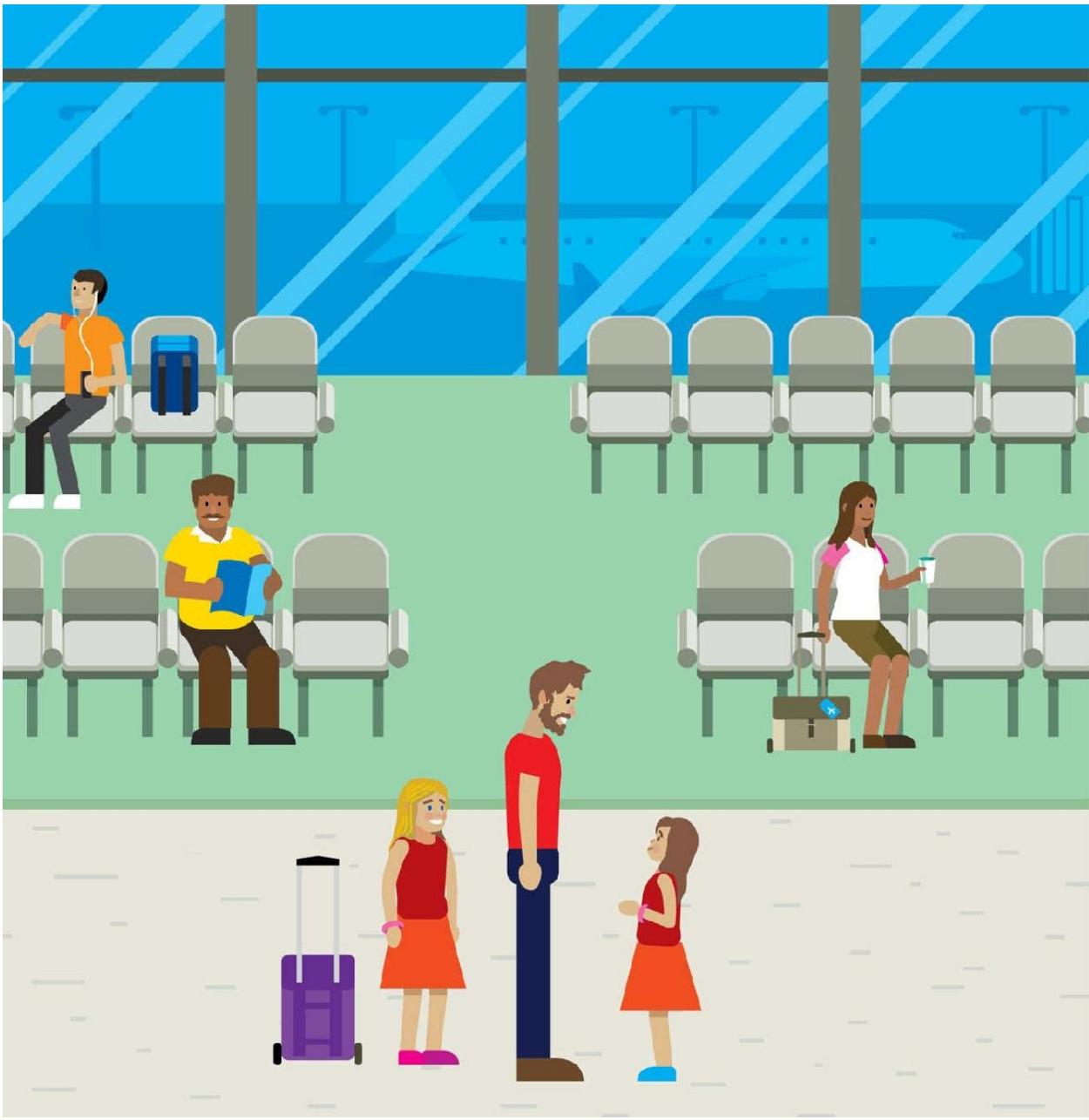
### **When should I write my documentation and tests?**

I think you know the answer to this one. The answer is "as you go along." It's very dangerous to work on the basis that you'll put the documentation in and perform the tests at the end of the project. The main reason this is a bad idea is that programs have a habit of taking longer to write than you expect, which means that you'll probably be so busy fixing faults and getting the program to work that you might not have time to document and test it if you leave those tasks to the end.

Creating documentation and tests as you go along (or even before you write the program itself) dramatically reduces the chances of the program failing. Writing good tests as you develop your code is an effective way to make sure you have a good understanding of the specification, and it also means that you are working with “good” components as you build your solution. I find that I write Python programs best if I create a very small amount of code at a time between each test. Rather than writing 500 lines of code and then running it to see what it does (which usually fails spectacularly), I’ll instead write 50 lines ten times. After each 50 lines, I’ll build some tests and run them to prove that the code works, and then I move on to the next 50 lines.

I’ve found that very large Python programs that don’t work are extremely difficult to fix, so I write very short amounts of code at a time.

# **Part 3 Useful Python**



Parts one and two gave you a good foundation in the Python language and a good understanding of software design. You've built some substantial applications, and hopefully you've built some of your own programs, too. You also know about the importance of testing and documentation and have seen the powerful Python tools that can help you with these tasks.

Now it's time to move on to the really cool stuff. In this third part, you'll learn how to make Python programs that have graphical user interfaces, talk to the Internet, and work over the network. Then, we'll round things off with an exploration of game development in Python.

In this part, the balance of the content changes slightly. There will be a bit less talking and a lot more doing. Expect to see more “Make Something Happen” sections as we explore how to build useful applications using popular Python frameworks. We’ll also have more “Make Something Happen: Development Challenges” where you can take our example code and “run with it” to create programs of your own.

# 13 Python and Graphical User Interfaces



## What you will learn

This chapter begins with a description of a new Integrated Development Environment for Python. The IDLE editor we've been using for the previous twelve chapters is a great place to learn to program, but we're starting to find limitations with the way it works, particularly now that we're creating applications that span several source files. The Visual Studio Code application provides a flexible and fast place to work with Python, so we'll look at how we can use it to develop our code.

Once we've sorted out our new workplace, we'll look at the creation of Python programs that have a Graphical User Interface (GUI). We'll use the popular Tkinter module that is supplied with the Python language and allows us to write easy-to-use programs that work on any device that supports Python. We'll discover how to create labels, text entry boxes, buttons, and list boxes. We'll

then find out about events, create a drawing program, and finish off with a GUI version of the Fashion Shop application.

## Visual Studio Code

### Create a Graphical User Interface with Tkinter

### What you have learned

## Visual Studio Code

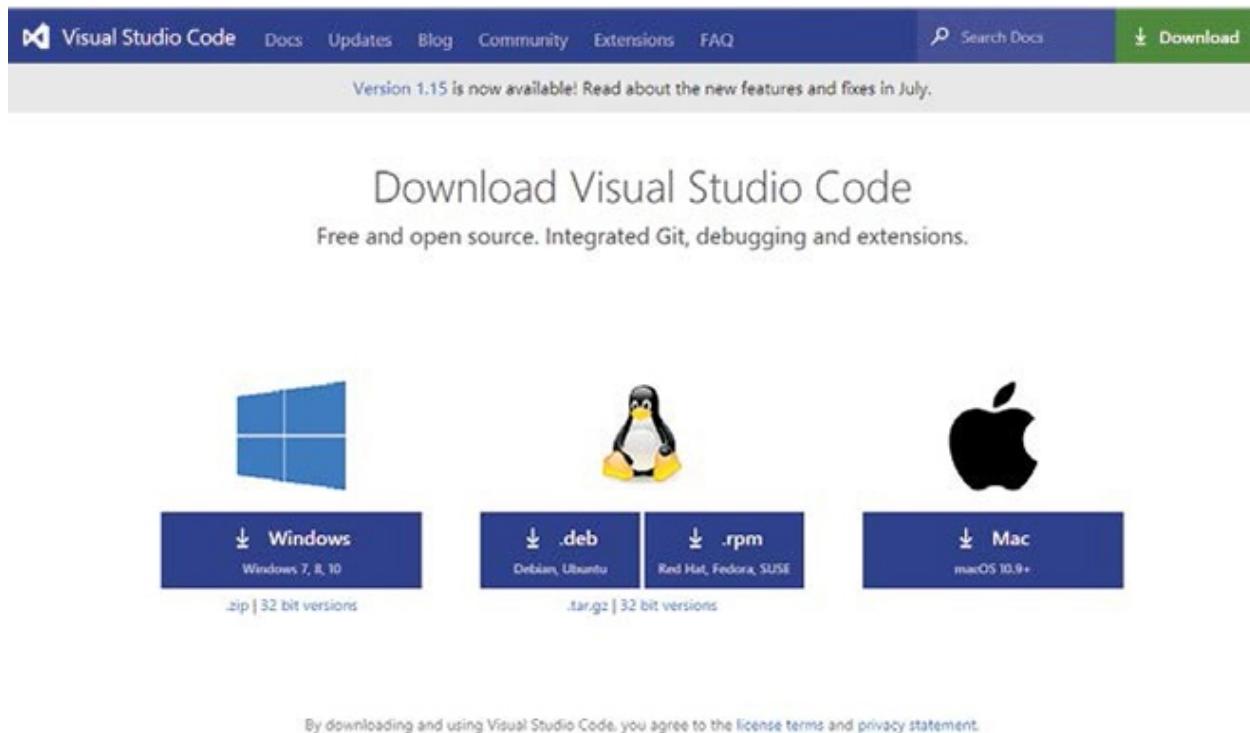
The IDLE program supplied with Visual Studio is a great place to learn how to program. However, as we begin to write larger programs, we start to notice that it has some limitations. If you want to make a program out of several Python source files (as we've begun to do now that we're using modules), the IDLE experience is not a good one. You must remember to save all open file windows before you run your program; otherwise, it might not incorporate all the latest changes to your code.

Visual Studio Code is a free and lightweight program editor from Microsoft. It's available for a wide range of operating systems, including Windows, Mac, and Linux. It's an open-source project, so you can even take a look inside the Visual Studio Code program source code and discover exactly how it works. Visual Studio Code is not tied to working with any specific programming language; it supports *plugins* that can be installed within the editor to customize it. We'll install Visual Studio Code and then use a very popular Python plugin from open-source contributor Don Jayamanne.

We'll still use IDLE from time to time, though, as it's still a great place to use the Python Command Shell.

## Install Visual Studio Code

You can download a copy of Visual Studio Code for your machine from <https://code.visualstudio.com/Download>, which you can see in **Figure 13-1**.

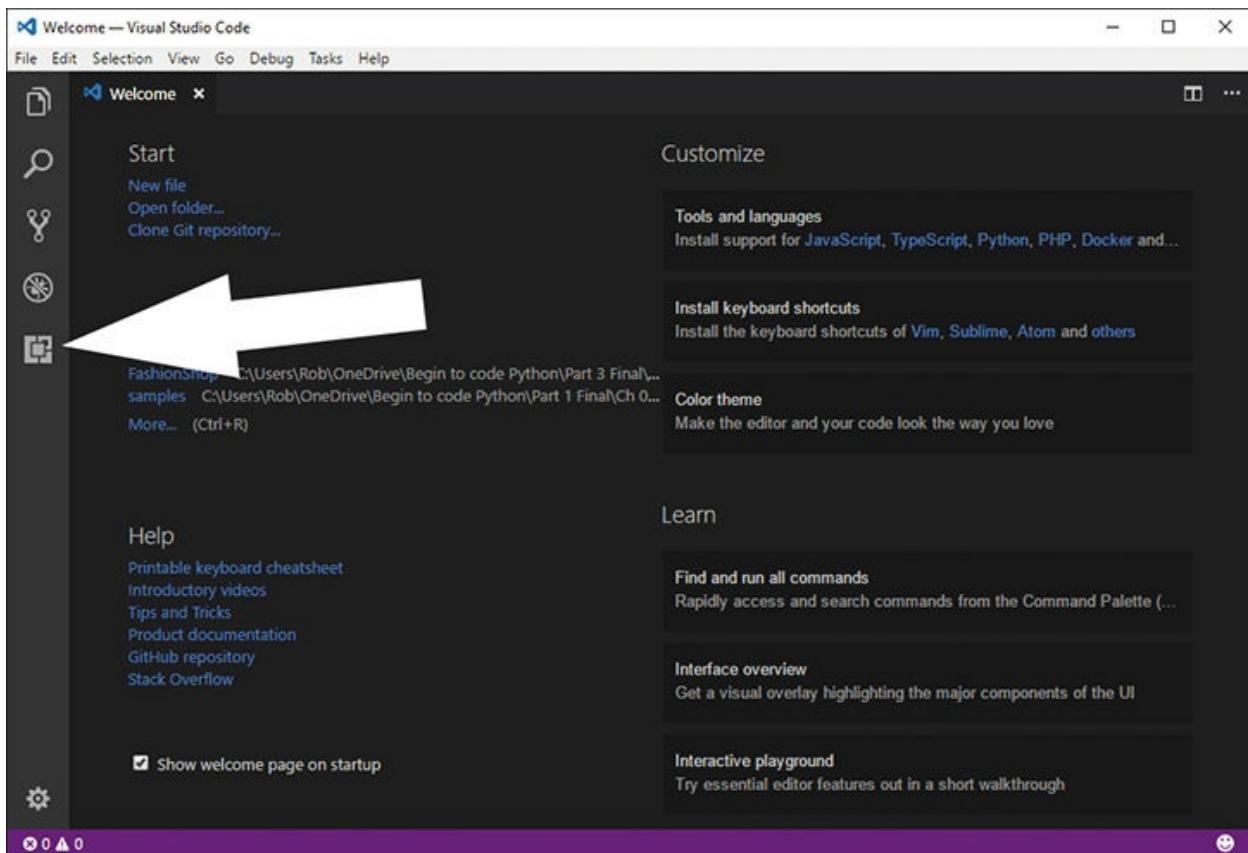


**Figure 13-1** Visual Studio Code downloads

Select the version of the program for your computer by clicking the appropriate button on the page. Follow the installation instructions to install Visual Studio Code on your machine.

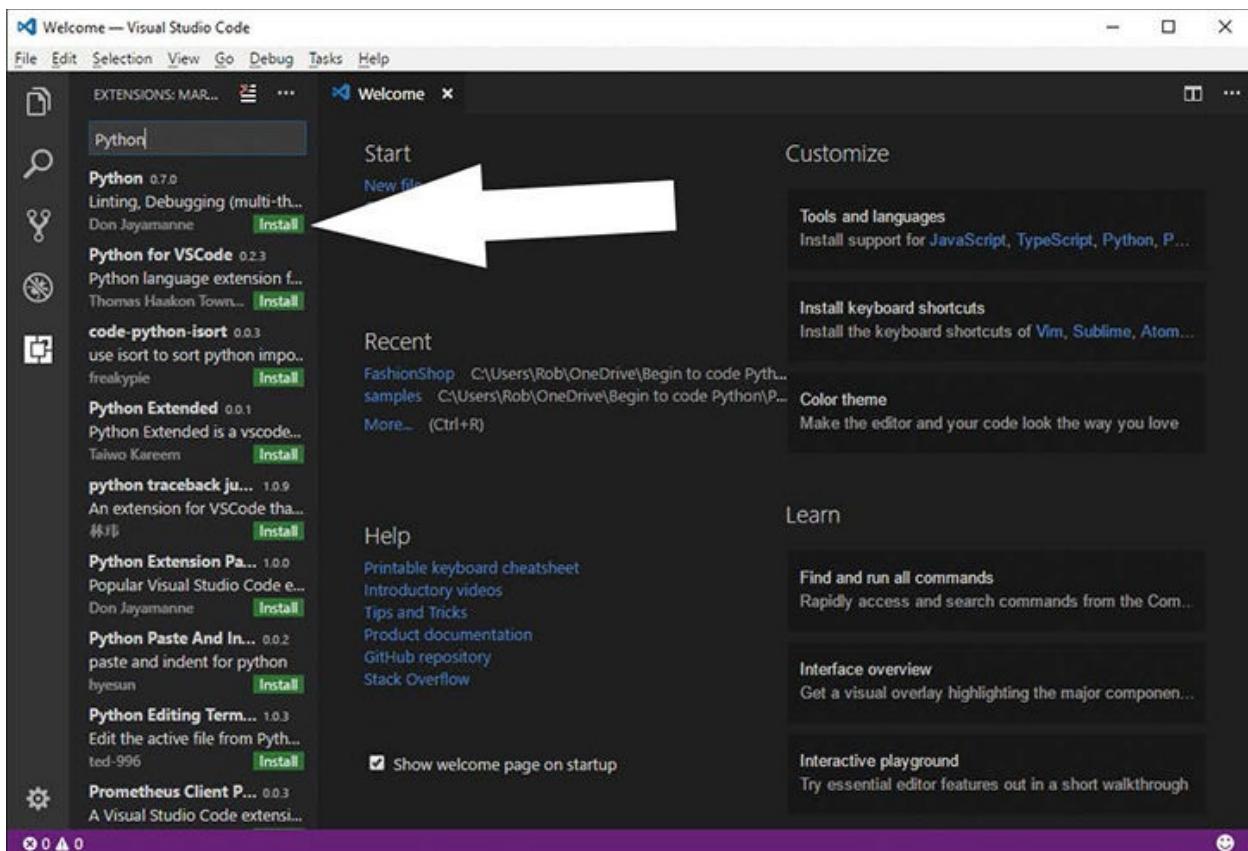
## Install the Python Extension in Visual Studio Code

Once we have Visual Studio Code installed, we next need to add the Python Extension that helps us work with Python programs. Open the Visual Studio Code application and click the Extensions icon indicated by the arrow in **Figure 13-2**.



**Figure 13-2** Extensions selector

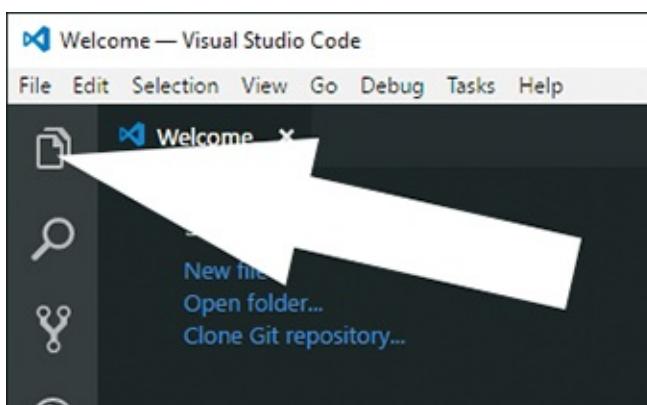
Visual Studio Code will now allow you to select extensions, showing you a list of available extensions. The Python environment we want to use should be near the top of the list, but if it's not visible, type **Python** into the search box at the top as shown in **Figure 13-3**. Once you've found the correct extension (you want the one by Don Jayamanne), click the Install button. Now that you have the extension installed, we can start writing some Python.



**Figure 13-3** Installing the Python Extension

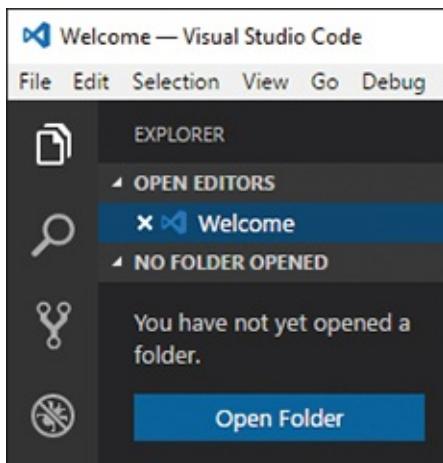
## Create a project folder

Visual Studio Code manages your work in folders. Each folder holds the program files for a specific project. When you're working on a project, you have the project's folder open in Visual Studio Code. To open the folder explorer view in Visual Studio Code, click on the folder icon as shown in **Figure 13-4**.



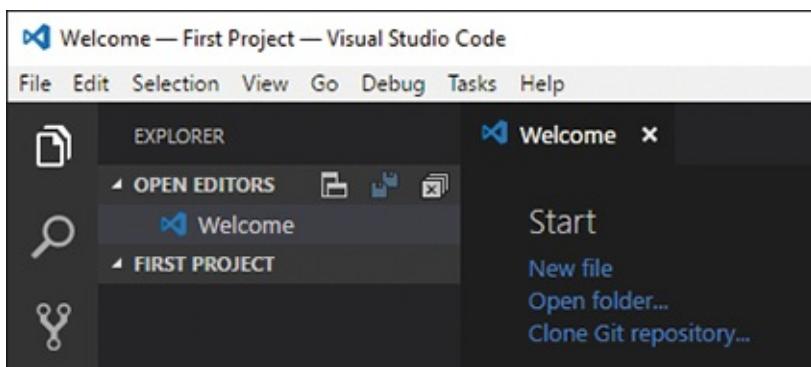
**Figure 13-4** Opening the folder explorer

Visual Studio Code will tell you that you presently have no folders open and invites you to open one by clicking the Open Folder button shown in **Figure 13-5**.



**Figure 13-5** The Open Folder button

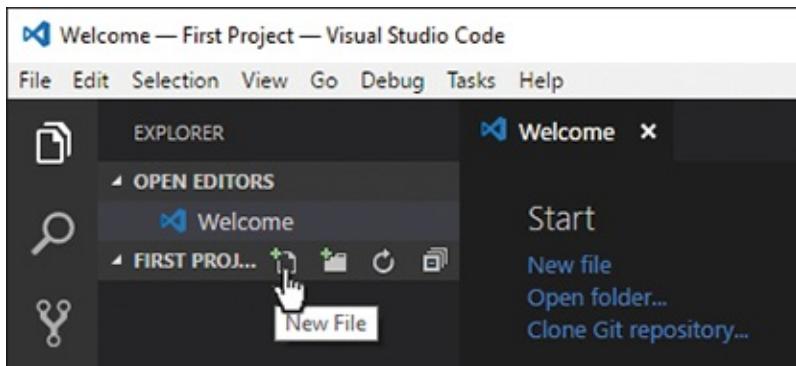
When you click **Open Folder**, a dialog appears that you can use to create or select a folder. Make a new folder called **First Project** and select it. Note that the precise dialog you see at this point will depend on which operating system you're using. Once you've opened the folder, it will appear in the folder explorer in Visual Studio Code, as shown in **Figure 13-6** below.



**Figure 13-6** First Project in Visual Studio Code

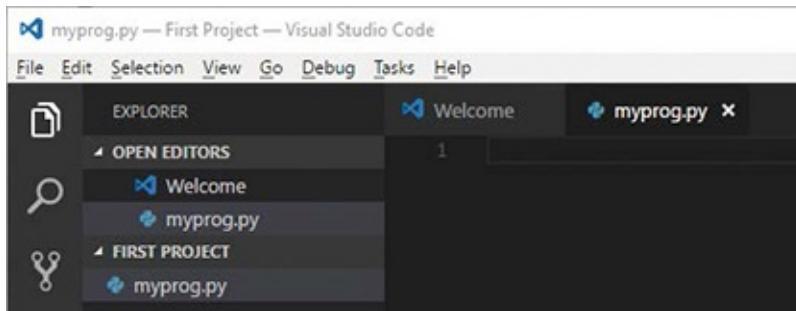
## Create a program file

Currently, the folder is empty. Now, let's make a Python program. Rest your mouse cursor over First Project in the Folder Explorer and click the New File icon that appears, as shown in **Figure 13-7**.



**Figure 13-7** New File in Visual Studio Code

Give the new program file the name **myprog.py** and press **Enter**. The file will be created in the folder and opened for editing, as shown in **Figure 13-8**.



**Figure 13-8** File Editing in Visual Studio Code

Now, type in the following tiny Python program:

[Click here to view code image](#)

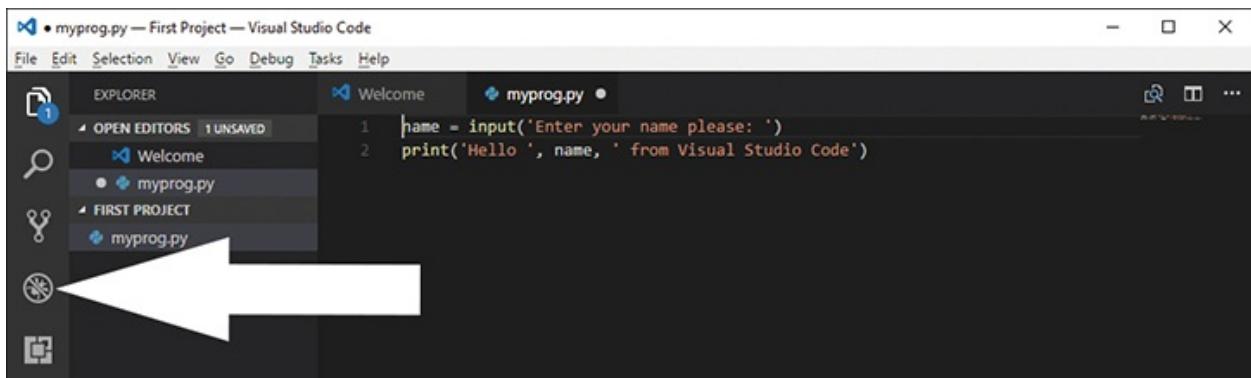
```
name = input('Enter your name please: ')
print('Hello ', name, ' from Visual Studio Code')
```

As you type the program into Visual Studio Code, you'll notice some differences in the way it works compared with the IDLE editor. The editor will suggest Python elements as you type their names. Simply press **Enter** to accept the

selected suggestion, or use the arrow keys to scroll down the suggested items to the word you want to select. You will also find help suggestions on functions that you call. If you move the cursor over particular words in the text, you'll find the same words highlighted on the page (which is very useful for seeing where you've used a variable). As you type, you'll also see a tiny map of your program appear in the top right corner of the screen. You can use this map to move rapidly through large files.

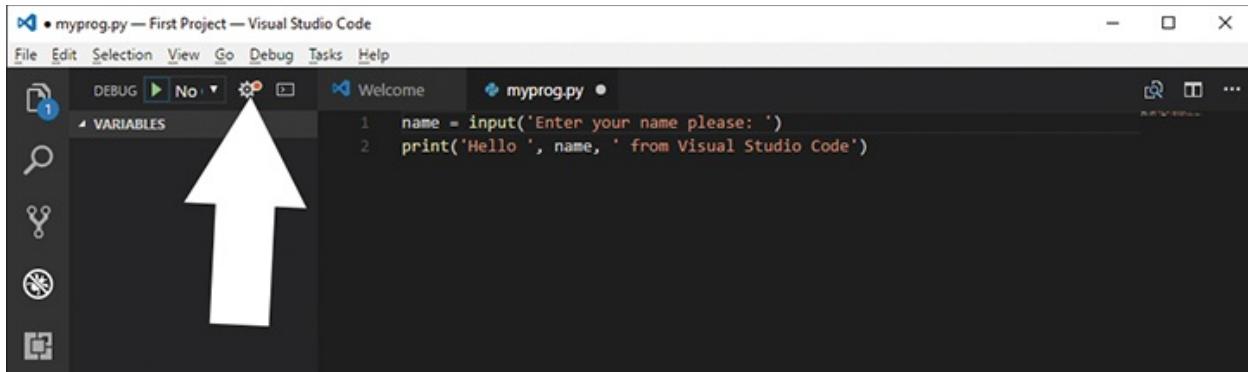
## Debug a program

Once you've typed your program, you'll want to run it. Press the **Debug** button on the left-hand menu, as shown in [Figure 13-9](#) below.



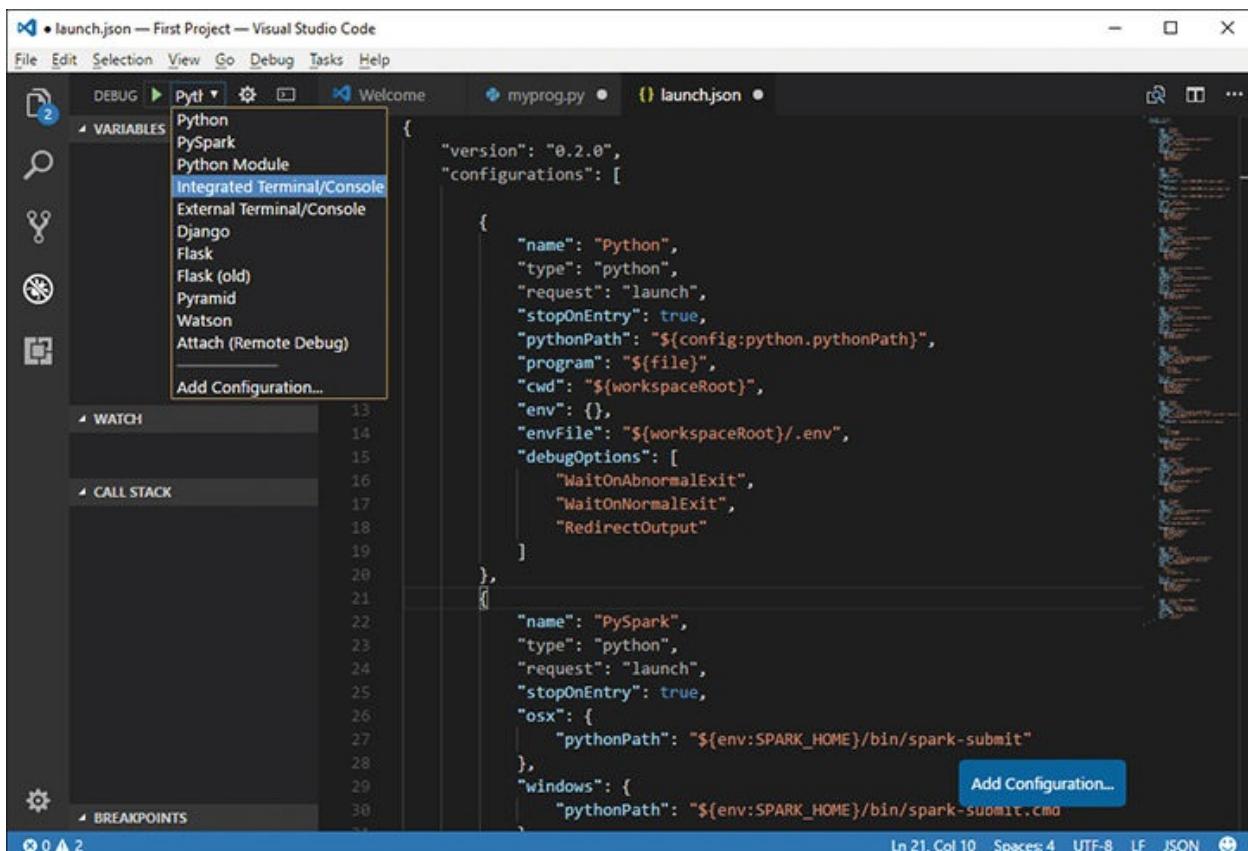
**Figure 13-9** Starting Debug in Visual Studio Code

The screen will now change to Debug mode. The Folder Explorer view changes to one that will let us see the contents of variables in the program as we run it. Now we must do some configuration. Visual Studio Code stores some configuration data in each folder with which you work. This data tells the editor the type of program you're working with, and how the editor should behave when working with this project. Currently, we don't have any configuration data set for this folder, so we need to create some. Click on the gears as indicated by the arrow in [Figure 13-10](#) below to open the configuration file for this folder.



**Figure 13-10** Set up Visual Studio Code options

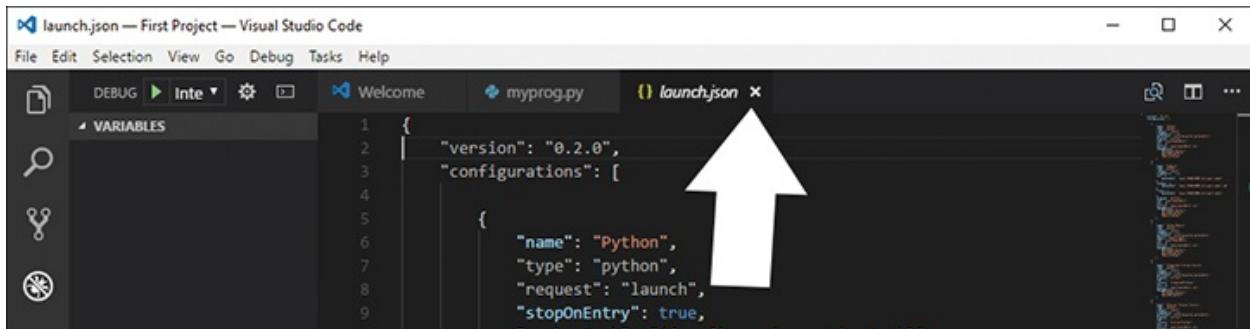
The configuration options are data files in the *JSON* format. We need to select a specific set of configuration options for this folder. Open the pull-down menu you see in **Figure 13-11** and select **Integrated Terminal/Console** from the options that appear.



**Figure 13-11** Configure Visual Studio Code options

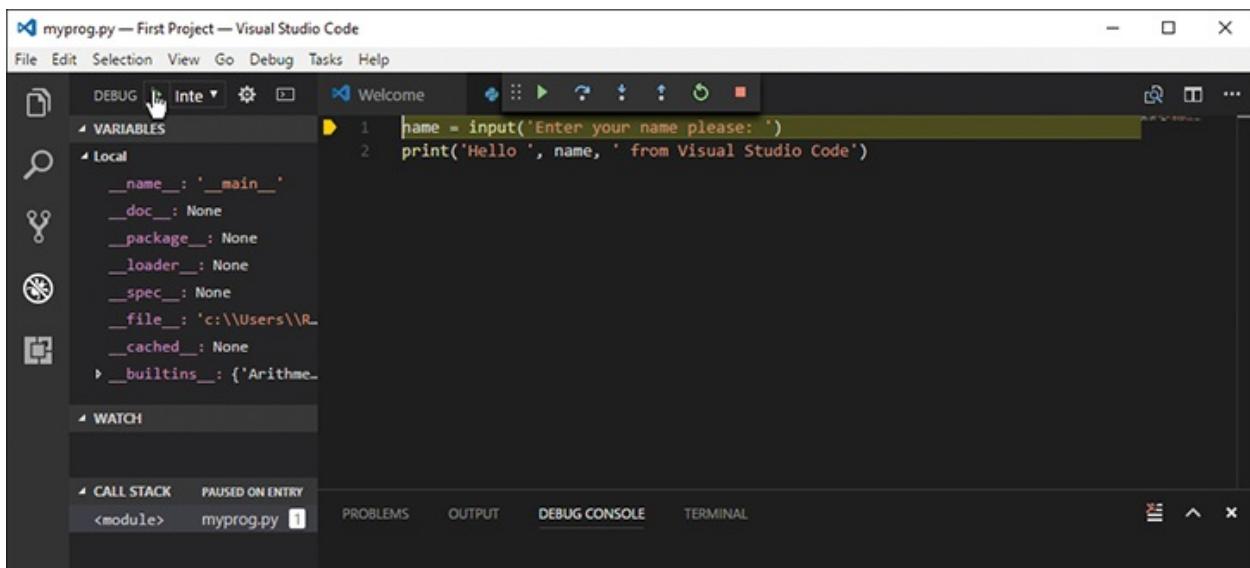
Once you've selected your options, close the configuration file by clicking the X

near the file name in the editor, as shown in **Figure 13-12** below.



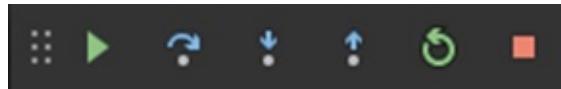
**Figure 13-12** Close the configuration file

Now we can debug our program. Click the green arrow you see next to Debug in **Figure 13-11** to start the Debug program. The program window now shows the statement about to be performed, as you can see in **Figure 13-13**.



**Figure 13-13** Visual Studio Code debugging

At the left, you see all the local variables. Currently, these are the variables set by Python for a program. The debugger allows you to watch the contents of variables, which are also shown on the left side. There is also a panel called the Call Stack that shows you the functions and methods that have been called. At the top of the screen are controls that let you debug the program, statement by statement, as shown in **Figure 13-14**.



**Figure 13-14** Run controls

From left to right, the controls are:

- **Move Panel** (six dots): Click and drag this to move the panel around.
- **Run/Continue** (green triangle): Run the program or continue from a breakpoint.
- **Step over** (Curved arrow over dot): Execute the present statement. If the statement is a method or function call, don't go into the method or function, just obey it. This is called "stepping over" a method or function.
- **Step into** (Arrow pointing down): Execute the present statement. If the statement is a method or function call, enter (or "step into") the method or function and start to step through it.
- **Step out** (Arrow pointing up): Complete the present method or function and "step out" of it.
- **Restart** (Counterclockwise arrow): Restart the program from the beginning.
- **Stop** (Square): Stop the program.

Press the Step Over control (the curved arrow over a dot), which will cause Visual Studio to perform the statement that calls the input function to read our name. The program uses the Terminal window at the bottom of the screen, so click that to open it, allowing you to enter your name, as shown in [Figure 13-15](#) below.

The screenshot shows the Visual Studio Code interface with a Python debugger session. The title bar says "myprog.py — First Project — Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Debug, Tasks, Help. The top toolbar has icons for DEBUG, Intellisense, settings, and other tools. The left sidebar has sections for VARIABLES, WATCH, CALL STACK, and BREAKPOINTS, with checkboxes for All Exceptions and Uncaught Exceptions. The main editor area shows the following Python code:

```
1 name = input('Enter your name please: ')
2 print('Hello ', name, ' from Visual Studio Code')
```

The TERMINAL tab is selected, showing a Windows PowerShell window. It displays the command PS C:\Users\Rob\Desktop\First Project> cd 'c:\Users\Rob\Desktop\First Project'; \$env:PYTHONIOENCODING='UTF-8'; \$env:PYTHONUNBUFFERED='1'; python C:\Users\Rob\.vscode\extensions\donjayama\ne.python-0.7.0\pythonFiles\PythonTools\visualstudio\_py\_launcher.py 'c:\Users\Rob\Desktop\First Project' 57070 34806ad9-833a-4524-8cd6-18ca4aa74f14 WaitOnAbnormalExit,WaitOnNormalExit 'c:\Users\Rob\Desktop\First Project\myprog.py'. The prompt "Enter your name please:" is visible, and the cursor is at the end of the line.

**Figure 13-15** Entering your name

After you've entered your name, you are returned to the debugger. If you look at the Variables display on the left side of the screen, you'll find that a new variable, `name`, has been added to the list. If you press the **Run** button in the control panel, you'll see the program run, and Visual Studio Code will say hello to you.

You might think that we've done a lot of work only to slightly improve our working conditions. However, when you start typing your programs, you'll find that this text editor is a huge improvement over the one in IDLE. Visual Studio Code is very good at suggesting things based on what you're typing. If you type a variable name in one part of the program, you'll find that the name is suggested the next time you start to type it. There's a lot to explore in the commands, too. The editor is very good at various kinds of searching and replacing. It's very easy to set breakpoints in your program. Just click to the left of the statement at which you want your program to pause. There's no need to open a special debugging window, as with IDLE.

Visual Studio Code is extremely powerful and customizable, and there are lots more useful extensions you can add. You can also integrate many Python tools that can be used to check and test your program.



## WHAT COULD GO WRONG

### Selecting the right Python interpreter

Obviously, you must have Python installed before you try using the Python extension in Visual Studio Code. However, you might have a problem if your system has multiple versions of Python installed. You might have installed Python version 3.6 to work through the examples in this book, but your computer might also have Python 2.7 installed. When the Python extension for Visual Studio Code is installed, it might pick the wrong version of the program.

You can use the Command Palette in Visual Studio Code to select the command you need to fix this. Open the Command Palette from the View menu. Then type in:

[Click here to view code image](#)

Python:Select Workspace Interpreter

You won't need to type all the text, as the palette will find matching commands from which you can choose. Select the command "Python: Select Workspace Interpreter" and then pick the Python interpreter with version 3.6.

You can also use this command to select the interpreter to use if you installed Visual Studio Code before you installed Python on your machine.

## Other Python editors

Visual Studio Code is my “weapon of choice” for writing Python. However, here are a couple other development tools you might like to check out.

### Visual Studio 2017 Community Edition

Visual Studio is a heavyweight development tool that’s very popular in the software development industry. It’s available on both Windows and Mac platforms, but unfortunately, at the time of writing, only the Windows version of Visual Studio supports Python development. If you have a Windows PC, I strongly suggest that you look into Visual Studio. The Community Edition is a free download and is extremely powerful. You can find it at [www.visualstudio.com](http://www.visualstudio.com).

### Pycharm

Pycharm is not without its charms. It provides a nice place to work, and the Community Edition is a free download from [www.jetbrains.com/pycharm](http://www.jetbrains.com/pycharm).

## Create a Graphical User Interface with Tkinter

The mainstay of our interactions with our programs has been the input and print functions provided with Python, along with the BTCInput module that we created to read numbers and text. Now we’ll find out how to use Python to create a Graphical User Interface (GUI). You should already be very familiar with GUIs, as most modern applications are controlled in this way.

A user interface is what people see when they use your program. A graphical user interface displays buttons, text fields, labels, and pictures that the user works with to get their job done. Part of the job of the programmer is to create this “front end” and then put the appropriate behaviors behind the screen that allow the user to drive the program and get what they want from it. In this section, we’ll find out how to create a program that uses a graphical user interface.

It should come as no surprise that a graphical user interface on the screen is represented by objects. When a program is working with items on the screen, it is calling methods in the object. For example, if we want to change the text displayed by a label on the screen, we would call a method on the object that is responsible for that label and tell it to change the text.

We'll use a module called Tkinter, which is shipped as part of the standard Python distribution and contains lots of different kinds of objects that represent the objects on the screen. It's also a very good example of a class hierarchy, in that particular display items (for example, buttons, blocks of text, and images) are represented by classes that are subclasses of parent items. Tkinter is actually a Python interface to a Graphical User Interface toolkit called Tk. Tk is available for many different hardware platforms including Windows, Mac, and Linux devices.



## MAKE SOMETHING HAPPEN

### Build our first user interface

The best way to find out about Tkinter is to play with it. So, let's do that. We can do so from the Python Command Shell in IDLE. So, let's start that up. The first thing we need to do is import all the resources from the Tkinter module. Give the following command and press **Enter**:

```
>>> from tkinter import *
```

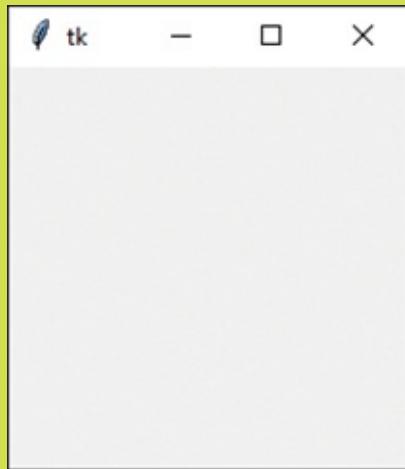
This form of input is different from others we've used recently. It's a way of using the items in a module without having to put the module name in front of each item. You can find more discussion about this in [Chapter 7](#) in the section "Convert our functions into a Python module."

Now that we've imported the module, we can use it. The first thing we'll do is create a "root" window, which will act as a container for all

the elements on our display. Type the statement below and press **Enter**:

```
>>> root = Tk()
```

The statement creates a new window on the screen and sets the variable `root` to refer to the window. You should notice that a new window has appeared on your desktop. It should look like the one below.



Let's create a new `Label` and add it to the window. `Label` items are used to display text in a window. The user can't interact with a `Label`, but a program can change the text on the label to display results on the screen. Type in the following statement and press **Enter**.

[Click here to view code image](#)

```
>>> hello = Label(root, text='hello')
```

The initializer for a `Label` takes two parameters. The first is the *parent* display object, which is the object within which the `Label` will be displayed. You can put objects inside objects so that you can build up complex displays. We won't do that just yet; instead, we'll display the `Label` in the main window so we can pass in the value of `root`. The second parameter we're giving to the initializer is a keyword argument called `text`, which is the text that we want the `Label` to display.

If you look at the window on the screen, you'll notice, perhaps to your disappointment, that the label has not appeared. This is because the graphical user interface doesn't put anything on the screen until it knows where to put it.

There are two ways you can position things within a display. You can use a mechanism called `pack`, which, as the name implies, packs the elements together in the window. You can give `pack` hints such as `"LEFT"` or `"TOP"` to tell it to put the item in that part of the display. However, I suggest that you use a mechanism called `grid`. This lets you lay out your items in a grid. This means that you'll need to plan your screen layout before you write the program, but a bit of planning is never a bad thing in my experience. We tell our label to use the grid layout method by calling the `grid` method on the label. Type in the following statement and press **Enter**.

[Click here to view code image](#)

```
>>> hello.grid(row=0, column=0)
```

This tells the `Label` referred to by `hello` to use the grid layout and to put it at grid location `(0, 0)`. This is the top left corner of the screen. If you look at the display window, you should notice two things. First, the label is now displayed. Second, you should see that the window has now been shrunk to fit the label within it.

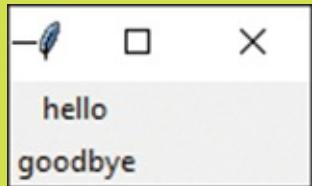


Let's add another label. Enter the following two statements:

[Click here to view code image](#)

```
>>> goodbye = Label(root, text='goodbye')
>>> goodbye.grid(row=1, column=0)
```

The display will now contain two labels.



The labels seem to be aligned at the left edge of the window. But the `hello` text is indented slightly, as it is centered about the label portion of the window. We can use some settings to improve this. We can also specify margins around items we display. But that's for later. For now, let's add a button.

Buttons are one way that a user can initiate an action in our programs. The user presses a button when they want something to happen. So, we need a way of linking a button to some code in our application. This turns out to be very easy. We create a function, tell the button the name of the function, and then when the button is clicked the function is called. So, let's make a button function. Type the following text and press **Enter** after each statement, including the empty line after the `print` statement.

[Click here to view code image](#)

```
>>> def been_clicked():
 print('click')

>>>
```

We now have a function called `been_clicked`, which we can connect to our button when we create it. Let's do that now. Enter the following statement.

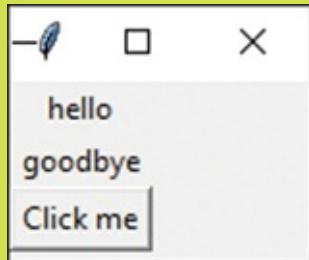
[Click here to view code image](#)

```
>>> btn = Button(root, text='Click me',
 command=been_clicked)
```

This creates a `Button` and sets the variable `btn` to refer to it. The second argument tells the `Button` to call the `been_clicked` function when the button is clicked. Now, let's place the button on the display. Enter the following statement to place the button at the bottom of the display.

[Click here to view code image](#)

```
>>> btn.grid(row=2, column=0)
```



Now, you're really going to have to click the button.

[Click here to view code image](#)

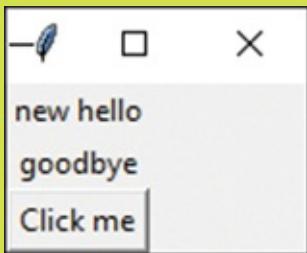
```
>>> btn.grid(row=2, column=0)
>>> click
click
click
```

I clicked the button three times, as you can see above. Each time you click the button, the function `been_clicked` is called. Functions such as `been_clicked` are called *event handlers* because they are executed in response to an external event.

Next, we'll change the content of one of the labels on the display. Display elements provide a method called `config`, which can be used to configure them. We can set the text attribute of the label by using the `config` method. Type the statement below and press **Enter**.

[Click here to view code image](#)

```
>>> hello.config(text='new hello')
```



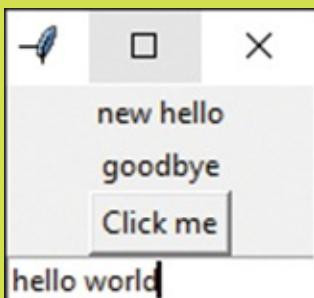
The content of the `hello` label changes to the new text.

The final thing we'll do is read some text from a display element, which is how we can read things entered by the user. If the user is only entering a single line of text, we can use the `Entry` component for this. Type in the following statements, pressing `Enter` after each of them.

[Click here to view code image](#)

```
>>> ent = Entry(root)
>>> ent.grid(row=3, column=0)
```

These statements create an `Entry` item at the bottom of our little program, which is referred to by a variable called `ent`. I've typed the universal computer greeting `hello world` into the text entry area, as you can see below. You can type in whatever text you like.



Now that we've managed to enter some text, the next thing to do is to try to read it from our program. We can use the `get` method on our text

entry object to do this. Type in the following statement. The `get` method asks an element to return the text it is holding.

```
>>> print(ent.get())
```

When you press Enter, the `get` method runs on the `Entry` object, and it returns the string that was typed in. In my case, it shows `hello world`.

[Click here to view code image](#)

```
>>> print(ent.get())
hello world
```

You can type in some more text and read it again, just to prove that it works.



## CODE ANALYSIS

## Building a graphical user interface

You might have some questions about the user interface we just created.

**Question:** What happens if we change the size of the window on the desktop?

**Answer:** We haven't given the graphical user interface any special instructions about what to do if the size of the window is changed, so if we use the mouse to grab hold of the edges of the window and change its size, we'll find that we can make the window far too big, and we can also make it smaller than the components it is

displaying. However, we can set attributes on the window to make it impossible to change its size:

[Click here to view code image](#)

```
root.resizable(width=False, height=False)
```

The `resizable` method on the `root` display element lets us determine how the user can change the size of its window on the screen. You can try it now with the window we created in the previous “Make Something Happen.”

We can also make it possible for the user to resize the window and have the size and position of components change automatically.

**Question:** What happens when I close the window we just created?

**Answer:** Because we have used the IDLE Command Shell to create the window, the window will disappear when you close it on the desktop. However, when we create a program that creates a graphical user interface, we’ll discover a way that our program can get control when the user closes the window.

**Question:** Will the window look the same on different machines?

**Answer:** Mostly. Because the Tk graphical toolkit uses the windowing system of the host computer to display its output, you’ll find that the window will look like a window on the host machine.

**Question:** What happens if an event handler function connected to a button takes a long time to complete?

**Answer:** The function connected to a button will run when the button is pressed. The button will be “stuck down” until the function completes. I actually tested this by creating a version of `been_clicked` that contained a call to the `sleep` function from the time module that made the function pause for ten seconds. When an event handler is running in response to one event, all the other controls on the application will be unresponsive.

You should take care to make sure that event handler functions are completed as quickly as possible. Fortunately, the kind of actions that we'll perform when buttons are pressed are not going to cause a problem because they all complete very quickly.

Python supports a mechanism called *threading*. An application can contain several *threads of execution* that execute simultaneously. Each thread could run a different program. An application could respond to a button press by starting a new program.

Creating and managing threads is beyond the scope of this book, but if you do want to perform an action that will take more than a second or so, you should look at how to use threading to perform the action.

**Question:** What happens if I place a large amount of text in a label?

**Answer:** The default behavior (that is, unless we specify otherwise) is for `Label` to expand to fit the text inside it. So, the window in the screen would grow to hold this text.

**Question:** What happens if I put two items in the same cell in the grid?

**Answer:** The most recently added item will be drawn in preference to the “older” one. In other words, a new item will “block out” an older one. It’s best not to do this.

**Question:** Can we update the contents of elements on the screen from within an event handler function?

**Answer:** Absolutely. In fact, this is how applications work.

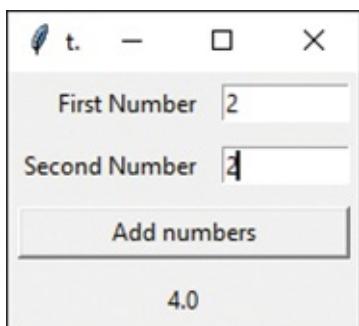
We now know nearly all we need to know to create applications that use a graphical user interface. The most important thing to remember is that events generated by the user (for example, clicking on buttons) will end up as calls to functions inside our application. In the example program above, the function `been_clicked` will never be called by any code that we write. It will be called by the button when the user clicks the button. If we create an application

that contains multiple buttons, we can connect each button to a different event handler. If we have two distinct ways to select a particular action (perhaps from a button or from a menu), we can connect both display elements to the same event handler function.

This form of application creation is a bit like “wiring up” electronic devices. We create a user interface design and then connect each of the user interface components to an event handler function. Note that events can be generated from actions such as mouse movements as well as key presses.

## Create a graphical application

Now that we know how to create a graphical user interface, we can make our first application that works this way. This application won’t do much, but it will show us how to create applications that work via a GUI. It’s a simple adding machine. **Figure 13-16** shows what it will look like. The user will type in two numbers, press the Add numbers button, and the result will magically appear underneath the button.



**Figure 13-16** Adding machine

This application looks deceptively simple, but there’s quite a lot to learn from building it. Let’s start with the application itself. I’ve created a class called **Adder** that will contain the application. The class will contain a method called **display** that will display the application:

[Click here to view code image](#)  
`class Adder(object):  
 ...`

```
 Implements an adding machine using a Tkinter
 GUI
 Call the method display to initiate the display
 """
def display(self):
 """
 Display the user interface
 Returns when the interface is closed by the
 user
 """
```

In the Adder.py source file, I've added some Python code that will run the adding machine if the Adder.py file is executed as a program:

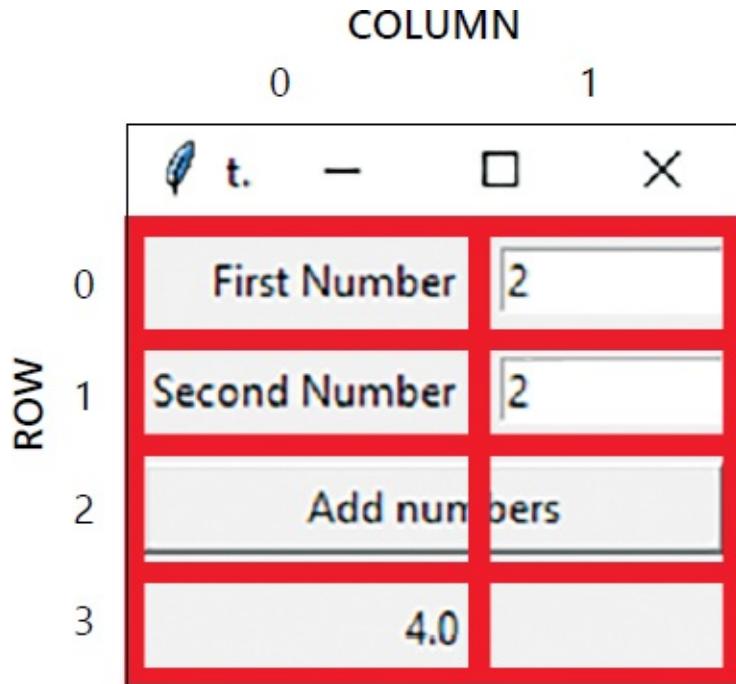
[Click here to view code image](#)

```
if __name__ == '__main__':
 app = Adder()
 app.display()
```

We've seen this arrangement of code before. The file can be opened as a module (for example, by pydoc for producing documentation), but it will only run as a program if it is the main module. Now we must create the contents of the display method that will implement our adding machine.

## Lay out a grid

We'll use the grid layout to place the elements in our display area. [Figure 13-17](#) shows the display with a grid laid over the top to show where each display element will go. The label "Second Number" is at location row=1 and column=0. Some of the items (the Add numbers button and the result value) seem to straddle two columns; we will discover how to do this in the next section.



**Figure 13-17** Adding machine layout

The items that display “First Number,” “Second Number,” and the result (in this case “4.0”) are all `Label` elements. We also have a `Button` to trigger the add numbers behavior and two `Entry` items to receive the two numbers that are typed in. Let’s start positioning components:

[Click here to view code image](#)

```
first_number_label = Label(root, text='First
Number')
first_number_label.grid(sticky=E, padx=5, pady=5,
row=0, column=0)
```

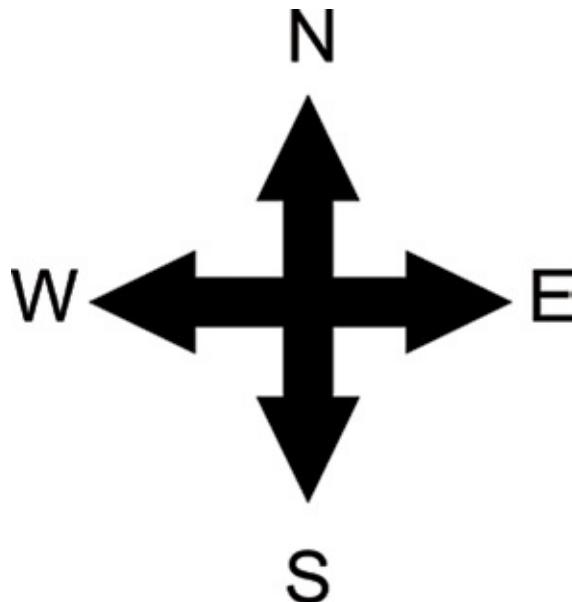
These are the statements that create and position the first number label at the top left corner of the display. This is in the element at row=0. There are a few extra arguments to the `grid` call that we haven’t seen before: `sticky`, `padx`, and `pady`. Let’s look at those.

## Use sticky formatting

Quite often, when laying things out in a grid, we’ll find two items of different

sizes in the same column. We can see this above, in that the label “Second Number” is slightly longer than the label “First Number”. The layout process will always size a row or column to the largest item in that row or column, which means there will be items placed in grid cells that are larger than they are. By default (that is, unless we state otherwise), an item is placed in the center of a larger cell.

For the `first_number_label` I want the label to be close to the `Entry` it is labeling. So, I’ve made the item “sticky” in an “easterly” direction. This means that the label will try to “stick” to an item on its east side. If I wanted to move the label all the way to the left, I’d make it sticky toward the west. If I want the item to “stretch” to fill a cell, I can make it sticky in two directions. This is how I made the “Add numbers” button stretch to fill the entire width of the display area. We’ll see this later in this section. If you’re not sure about compass directions, you can find a handy reference in **Figure 13-18** below.



**Figure 13-18** Compass points

## Use padding

Padding is extra space placed around the component to “pad” it out. Otherwise, the component will be drawn right up to the edge of the cell in which it is being drawn. I like to add around 5 pixels or so of padding around items on the screen. You can specify the amount of padding in the x and y directions. I put 5 pixels

around each of the items on the display in both directions.

## Span grid cells

The grid must be two elements wide so that I can display the label and the entry boxes for both numbers that the user will enter. However, I'd like the button and the result to be drawn across the full width of the display area. I can do this by merging the cells into which an element is to be drawn. Look at the definition of the `add_button` below.

[Click here to view code image](#)

```
add_button = Button(root, text='Add numbers',
command=do_add)
add_button.grid(sticky=E+W, row=2, column=0,
columnspan=2, padx=5, pady=5)
```

The first statement creates the `Button` instance, sets the text to be displayed on the button, and tells the button to call the function `do_add` when the button is clicked. The second statement places the `Button` in the grid in row 2, column 0. However, it also contains the argument `columnspan=2`. This means that the button will be drawn in a cell that spans two columns. This case means that the button will be the full width of the display because the display is two columns wide.

Note also that I've made the button "sticky" in both easterly and westerly directions so that it will be stretched across the entire display when it's drawn. I use the same technique to position the result label. Below are all the statements that position and set up the items on the form.

[Click here to view code image](#)

```
first_number_label = Label(root, text='First
Number')
first_number_label.grid(sticky=E, padx=5, pady=5,
row=0, column=0)
```

```
first_number_entry = Entry(root, width=10)
first_number_entry.grid(padx=5, pady=5, row=0,
column=1)

second_number_label = Label(root, text='Second
Number')
second_number_label.grid(sticky=E, padx=5, pady=5,
row=1, column=0)

second_number_entry = Entry(root, width=10)
second_number_entry.grid(padx=5, pady=5, row=1,
column=1)
add_button = Button(root ,text='Add numbers',
command=do_add)
add_button.grid(sticky=E+W, row=2, padx=5, pady=5,
column=0, columnspan=2)

result_label = Label(root, text='Result')
result_label.grid(sticky=E+W, padx=5, pady=5,
row=3, column=0, columnspan=2)
```

## Create an event handler function

We know that we can connect a function to a button so that when the button is clicked the event handler runs. For the adding machine, the event handler should read the text out of the two `Entry` objects into which the user has (hopefully) entered some numbers. The event handler should then convert the text into numbers, add the two numbers, and then display the result in the result label. Below you can see my version of the event handler for the adding machine.

```

EG13.01 First Adding machine
class Adder(object):
 """
 Implements an adding machine using a Tkinter GUI
 Call the method display to initiate the display
 """

 def display(self):
 # create all the screen elements here
 def do_add():
 first_number_text = first_number_entry.get()
 first_number = float(first_number_text)

 second_number_text = second_number_entry.get()
 second_number = float(second_number_text)

 result = first_number + second_number
 result_label.config(text = str(result))

 Method called to generate the user interface
 Event handler for the add button
 Convert the second number text
 into a floating-point number
 Calculate the result
 Convert the result into a string
 and display it
 Get the text out of the Entry for second number
 Convert the first number text into a floating-point number
 Get the text out of the Entry for the first number

```



## CODE ANALYSIS

### Writing an event handler

You might have some questions about the event handler we just created.

**Question:** Why is the event handler defined inside the display function?

**Answer:** We've seen before that Python will allow programmers to define functions inside other functions. The event handler function needs access to the `result` label and the two `Entry` variables that the user uses to enter the two numbers to be added. Code running inside a function has access to the variables in the enclosing namespace (we saw this in Chapter 7), and so the `do_add` function

can use variables declared in the `display` function.

I could have created `do_add` as a method in the `Adder` class (event handlers can be methods as well as functions), but then I would've had to make all the display elements attributes of the `Adder` class so that the `do_add` method could access them, which would have meant a bit more typing.

**Question:** What happens if the user doesn't type in a valid number before pressing the Add numbers button?

**Answer:** Good question. The answer is that the `float` function in the `do_add` event handler will be unable to convert the text in the Entry into a floating-point number. It will fail by raising an exception that will cause the `do_add` method to be abandoned when the exception is raised. You might notice the exception being raised if you're debugging the program, but when the program is running the user will not see any errors at all.

Entering invalid numbers will not stop the program from running, but the user will not see an error; they'll just notice that the result display will not be updated. In an upcoming section, we'll discover how a program can display a pop-up warning if the user does something like this.

## Create a mainloop

When we created a display window from the Python Command Shell in IDLE, we found that it just worked. This is because the shell was running. If we had exited the shell program, we'd have found that the display window disappears as well. If a program just created the display components and then ended, we'd find that the interface would flash up on the screen for a fraction of a second and then vanish when the program ended. To keep the display active, the Tkinter module provides a method called `mainloop` which a program should call once it has set up its display components:

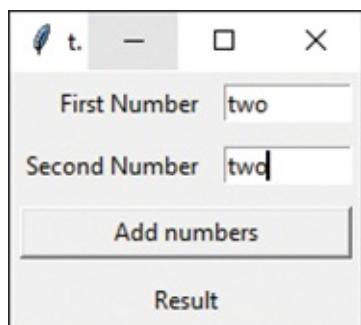
```
root.mainloop()
```

The name `mainloop` describes what this method does. It repeatedly fetches events and sends them on to functions that have been created to deal with the events. When the user closes the window on the screen (in Windows 10, they would click the X in the top right corner of the window), the `mainloop` method ends. As the `mainloop` method is frequently the last method call in a program that uses a graphical user interface when `mainloop` ends the program probably ends, too.

You can find a complete implementation of the adder program in the example file **EG13-01 First Adding machine** in the downloadable sample files for this chapter.

## Handle errors in a graphical user interface

The adding machine we've created works quite well. However, it does have problems if the user enters invalid text rather than numbers. **Figure 13-19** shows what can happen. The user has typed in two text strings. When they press the Add numbers button, the result display is not updated because the `do_add` event handler fails with an exception.



**Figure 13-19** Text as numbers

One way to deal with this would be to catch the exceptions and change the result string to reflect the issue. We saw how to catch exceptions in [Chapter 6](#). The code below is part of an improved `do_add` function that catches exceptions if either of the number conversions performed by the `float` method fail.

```
EC13-02 Exception handler with messages

def do_add():
 first_number_text = first_number_entry.get() Get the number text from the Entry on the screen
 try: Start of exception handler
 first_number = float(first_number_text) Statement that might throw an exception
 except ValueError: Handler for ValueError exceptions
 result_label.config(text='Invalid first number') Set the result to indicate
 return that an error has occurred
 Return from the method if the first number is not valid
 second_number_text = second_number_entry.get()

 try:
 second_number = float(second_number_text)
 except ValueError:
 result_label.config(text='Invalid second number')
 return
```

The code above works well, but it is not perfect. If the user enters invalid text in both `Entry` objects on the screen, the program will only tell the user about the first error, not the second. We can improve this code so that it builds an error string and displays it if any errors are detected:

```

def do_add():
 error_message = ''
 first_number_text = first_number_entry.get()
 try:
 first_number = float(first_number_text)
 except ValueError:
 error_message = 'Invalid first number\n'

 second_number_text = second_number_entry.get()
 try:
 second_number = float(second_number_text)
 except ValueError:
 error_message = error_message + 'Invalid second number'

 if error_message != '':
 result_label.config(text=error_message)
 else:
 result = first_number + second_number
 result_label.config(text = str(result))

```

The diagram illustrates the flow of the `do_add()` function. It starts by creating an empty error message string. Then, it gets the text from the first number entry. A try block is initiated to convert the first number text into a float. If the conversion fails (ValueError), an exception handler adds an error message indicating an invalid first number. Another try block is initiated for the second number entry. If the conversion fails, it adds the error message to the existing one. Finally, it checks if the error message is empty. If it is not empty, it configures the result label with the error message. Otherwise, it calculates the result (sum of both numbers) and configures the result label with the result string.

This version of the `do_add` function is much better. It uses a technique I've used many times when dealing with user errors. It starts with an empty error string. Each time code in the function finds something wrong, it will add text describing the error to the error string. If at the end of the function, the error string is empty, it means that there are no errors and the function can complete. Otherwise, it displays the error message. You can find this version of the error handler in the sample file **EG13-03 Adder with sensible messages**.

There is considerable scope for making this method even better. Tkinter provides methods to set the foreground and background colors of items on the screen, so you could make the `do_add` function indicate invalid user entries by changing the background color of invalid entries to red. The statement below shows how you can change the color of an item in the program. It configures `first_number_entry` so that the background of the `Entry` on the screen is red and the foreground (the color of the text in the entry box) is blue.

[Click here to view code image](#)

```
first_number_entry.config(background='red',
foreground='blue')
```

## Display a message box

Another way to inform the user of an error is to pop up a message box. This technique has the advantage that the user must see and acknowledge the error before they can continue. Tkinter provides a message box, and it's very easy to use. The first thing the program must do is import the `messagebox` module:

[Click here to view code image](#)

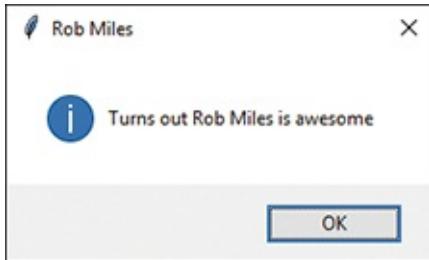
```
from tkinter import messagebox
```

The `messagebox` module contains three functions that can display messages: `showinfo`, `showwarning`, and `showerror`. All the message boxes have the same format, but a different icon is used for each. The user interface for the program displaying the message (in our case, the Adder program) will be locked until the user clears the error message by clicking OK or closing the message box. Each of the message functions accepts two arguments, a title and a message. Both are strings. Below we can see how we could use the `showinfo` function to show some information:

[Click here to view code image](#)

```
messagebox.showinfo('Rob Miles', 'Turns out Rob
Miles is awesome')
```

**Figure 13-20** shows the output of this important message. To display a warning, use the `showwarning` method. To display an error, use the `showerror` method. You can find a version of the Adder program that displays a message box to indicate user error in the sample file **EG13-04 Adder with message box**



**Figure 13-20** Important message from `showinfo`

We can make a version of the Adder program that displays a message box by replacing the statement that sets the result label to the error with one that generates a message box. The code below shows the part of `do_add` that handles errors. If the error message is not empty (in other words, something bad has happened), the message box will be displayed to indicate this.

```
if error_message != '':
 messagebox.showerror(title='Adder', message=error_message)
```

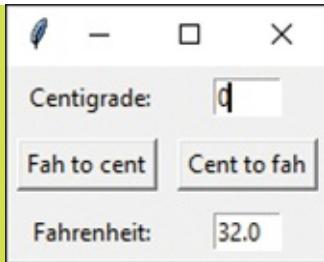
Is the error message empty? → Display an error message box



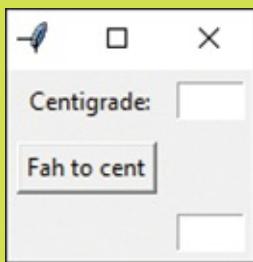
## MAKE SOMETHING HAPPEN

### Fahrenheit to centigrade. And back.

In this challenge, I'll give you a half-finished program to complete. This is something that happens surprisingly frequently in the software industry. When you get your first programming job, it's likely that you'll start by modifying an existing program rather than being asked to create an all-new program. The program you're working on is the ultimate temperature converter. The user can convert from Fahrenheit to centigrade or back. They type their conversion value into one box and, depending on which button they press, the other box will show the converted value. The program is supposed to look like this:



Unfortunately, the programmer hired to create the program has taken his job much too seriously, and has gone away to Hawaii, supposedly to test the program in higher temperatures. He has left behind a program that looks like this:



[Click here to view code image](#)

'''

Display a graphical user interface that lets users convert from temperature scales

'''

```
from tkinter import *
```

```
class Converter(object):
```

'''

Displays a Tkinter user interface to convert between Fahrenheit and centigrade

Call the display function to display the converter on the screen

'''

```
def display(self):
```

'''

Displays the converter window

```
 When the window is closed, this
method completes
 '''

 root = Tk()

 cent_label = Label(root,
text='Centigrade:')
 cent_label.grid(row=0, column=0,
padx=5, pady=5, stick=E)

 cent_entry = Entry(root, width=5)
 cent_entry.grid(row=0, column=1,
padx=5, pady=5)

 fah_entry = Entry(root, width=5)
 fah_entry.grid(row=2, column=1,
padx=5, pady=5)

 def fah_to_cent():
 '''
 Convert from Fahrenheit to
centigrade and display the result
 '''

 fah_string = fah_entry.get()
 fah_float = float(fah_string)
 result = (fah_float - 32) / 1.89
 cent_entry.delete(0, END) #
remove the old text
 cent_entry.insert(0, str(result))
insert the new text

 def cent_to_fah():
 '''
 Convert from centigrade to
Fahrenheit and display the result
 '''

 cent_string =
cent_entry.get()
 cent_float = float(cent_string)
```

```

 result = cent_float * 1.8 + 32

 fah_to_cent_button = Button(root,
text='Fah to cent', command=fah_to_cent)
 fah_to_cent_button.grid(row=1,
column=0, padx=5, pady=5)

root.mainloop()

if __name__ == '__main__':
 app = Converter()
 app.display()

```

The programmer has used a feature of Tkinter that we haven't seen before. When the program has calculated a new result, it must display it in a text entry field. Updating the text in an `Entry` is slightly more complicated than just changing the text in a `Label`. There are very powerful editing features available, but we just want to replace the text with new text. The two statements below show how this is done. The first statement deletes all the text from `cent_entry`. The first argument to the `delete` method is the position to start deleting (0 means the beginning of the string). The second argument to the `delete` method is the position to stop deleting. The variable `END` is declared in the Tkinter module and means "the end of the line."

The second statement inserts a string containing the result into `cent_entry` starting at the location 0 (the beginning of the string).

[Click here to view code image](#)

```

cent_entry.delete(0, END) # remove the old
text
cent_entry.insert(0, str(result)) # insert
the new text

```

You can find the starter code in the folder **EG13-05**

**TemperatureConverter Starter** in the sample code for this chapter.

The folder is all set up for use with Visual Studio Code. If you want to “skip to the end,” you can find a complete version of the program in the folder **EG13-06 TemperatureConverter Complete**. However, even the complete version could use some attention; currently, it doesn’t handle invalid inputs.

You can use this program as the basis for any conversion you like, such as ounces to grams, mph to kph, or dollars to euros.

## Draw on a Canvas

We can also use graphical interfaces to allow the user to draw with the mouse on the screen. We do this by creating a drawing area that sends our program an event each time the user moves their mouse. If this event performs a drawing operation, we have an instant drawing program. Let’s look at how we can get events from areas of the screen.



**MAKE SOMETHING HAPPEN**

## Investigate events and drawing

We can investigate Tkinter events from the Python Command Shell in IDLE. So, let’s start that up. As before, the first thing we need to do is import all the resources from the Tkinter module. Give the following command and press **Enter**:

```
>>> from tkinter import *
```

Next, we need to create a window on the screen. Enter the following statement and press **Enter**:

```
>>> root = Tk()
```

Now we'll create a **Canvas**. A **Canvas** is a display component that can act as a container for lots of other display elements. We can draw and position these elements inside the canvas. When you create a **Canvas**, you can tell the graphical user interface the size of the **Canvas** in pixels. Enter the following statement to create a **Canvas** that is 500 pixels square.

[Click here to view code image](#)

```
>>> c = Canvas(root, width=500, height=500)
```

To get the **Canvas** displayed, we need to specify where to place it. It will be the only item on the display, so we can place it at row 0 and column 0.

```
>>> c.grid(row=0, column=0)
```

If you look at the window that's been created, you should see that the program is now displaying a square window.



- □ ×

Now we need to connect a function to the events that Tkinter generates when a mouse is moved over the [Canvas](#). Let's write the function first. Enter the following function. Enter a blank line after the `print` statement to end the function.

[Click here to view code image](#)

```
>>> def mouse_move(event):
 print(event.x, event.y)

>>>
```

The function is supplied with a single parameter, which is a reference to an event object. This object has two attributes, which are the x and y positions of the mouse pointer at the time the event occurred. The method above just prints these positions on the screen.

Now we need to connect this function to the event generated when the

mouse is moved with a button pressed. This will give us the movement detection that will make our drawing program work, which is called *binding* the function to the event. Objects on the display provide a `bind` method that programs can use to connect functions to events. Each event has a unique name. Type in the following statement and press **Enter**. The statement calls the `bind` method on the canvas and links the `<B1-Motion>` event (that is, mouse motion with button 1 pressed down) to the function `mouse_move`. Each time the `Canvas` detects a mouse movement with the button pressed, it will call the method.

The `bind` method returns a string that describes the binding that has taken place. A program could use this string to identify the binding and disconnect the connection later, but we can ignore this string for now. Note that because the description string is supposed to be unique on a specific machine, you may find that the string displayed on your machine differs from the one shown below.

[Click here to view code image](#)

```
>>> c.bind('<B1-Motion>', mouse_move)
'2886099647752mouse_move'
>>>
```

Now for the fun bit. Move your mouse to the window displaying the canvas, hold down the left (or only) button on the mouse and drag it. Watch the Python Command Shell in IDLE. You should see a stream of numbers being generated. Below you can see some of the numbers that I saw. If you drag the mouse up to the top left corner of the canvas, you should see the numbers getting smaller. This is because the origin of the coordinates (the point 0,0) is the top left corner of the canvas. This should not come as a surprise; it is the same way that the grids are numbered.

[Click here to view code image](#)

```
>>> 283 277
```

290 297  
290 306  
289 307

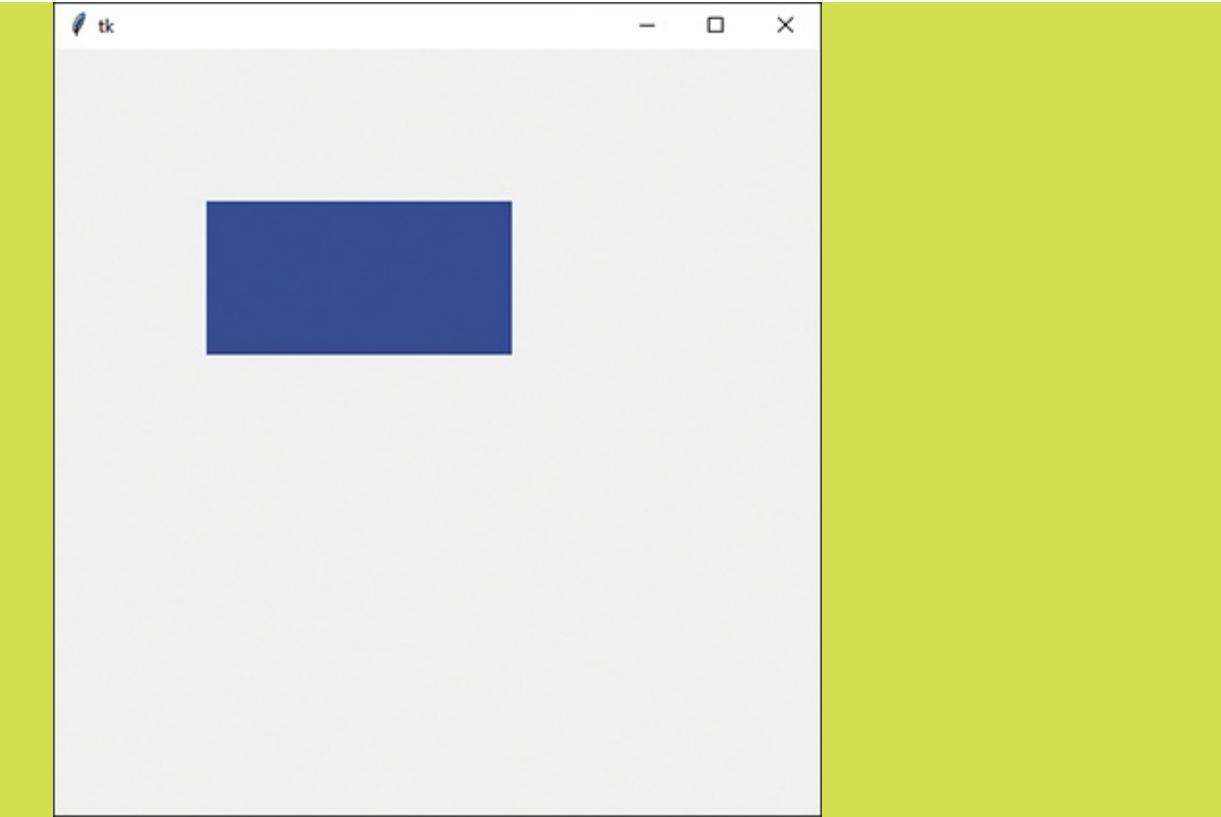
Printing coordinates is nice enough, but we are making a drawing program, and we need to draw a dot. The Canvas object provides a method called `create_rectangle` that should do the trick. Tear yourself away from dragging the mouse around your canvas and enter the following statement. This will draw a blue rectangle. The top left corner of the rectangle will be at coordinate (100,100). The bottom right corner of the rectangle will be at coordinate (300,200). The `outline` argument sets the color of the outline of the rectangle; the `fill` argument sets the color used to fill in the block. Unless you specify otherwise, the outline color will be black.

[Click here to view code image](#)

```
>>>
c.create_rectangle(100,100,300,200,outline='blue',

1
>>>
```

If you look at the output window for your program, you should see that a blue rectangle has duly appeared.



You may be wondering why the value 1 was displayed when we created the blue rectangle. This is because when you create an object on a canvas, the method that creates it will return a value that identifies this object. If we just call a method, Python will just display the value returned by it, which in this case was 1 because we have just created object number 1 on the [Canvas](#).

A canvas manages each object by its number. We can ask the canvas to remove an object from the display by using the [delete](#) method. Type the following command and press **Enter**.

```
>>> c.delete(1)
```

You should see the blue rectangle disappear. This is a very powerful feature of the [Canvas](#). Every single element on the screen is a separate object that we can find and manipulate after we've drawn it.

Now we need to use the drawing method to allow us to draw with the

mouse. We can create a new function that draws a block at the position a mouse event was detected. Enter the statements below. Add an empty line after the call of `create_rectangle` to end the function definition.

[Click here to view code image](#)

```
>>> def mouse_move_draw(event):
 c.create_rectangle(event.x-5,event.y-
5,event.x+5,event.y+5,
 fill='red', outline='red')

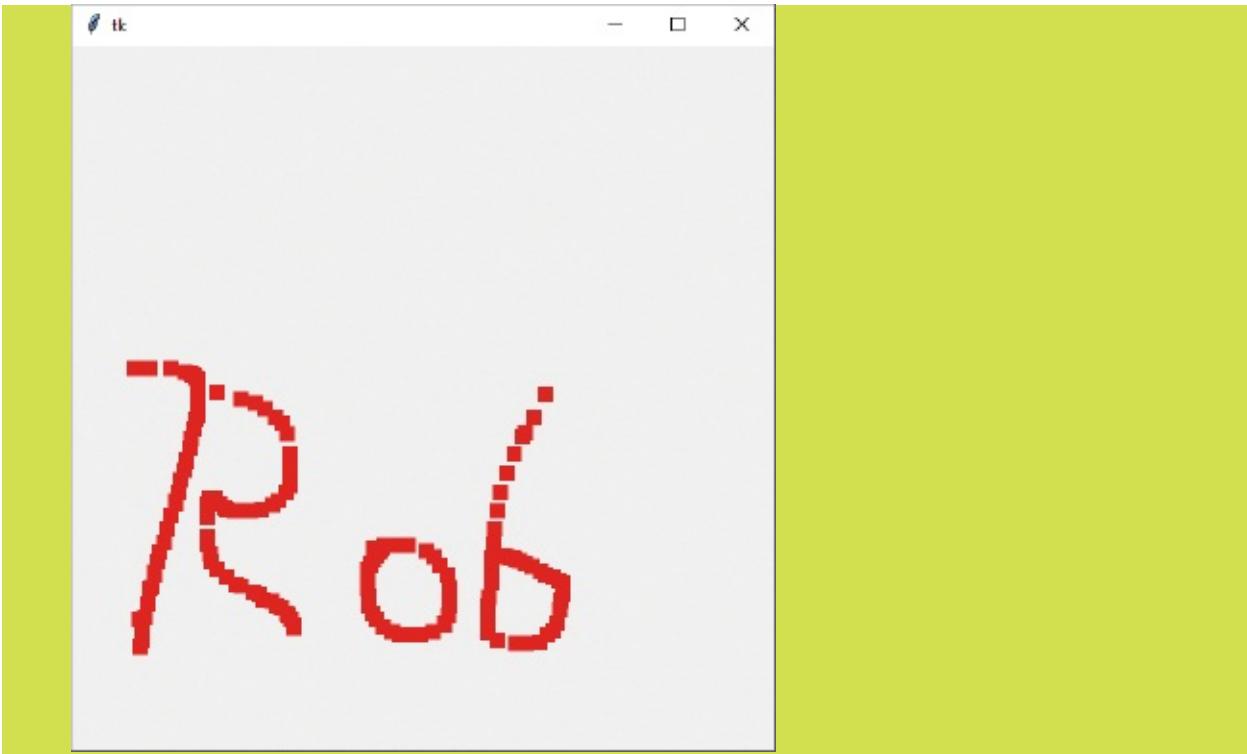
>>>
```

This method creates two points that define the rectangle to be drawn. The first point is five pixels to the left and above the mouse position, the second point is five pixels to the right and below the mouse position. The result is that the function will draw a ten-pixel square block centered around the position of the mouse. Now, all we need to do is bind this new draw function to the event generated when the mouse is moved with the pointer held down.

[Click here to view code image](#)

```
>>> c.bind('<B1-Motion>', mouse_move_draw)
'2886099651528mouse_move_draw'
```

Once you have bound the function to the event, you should be able to draw on the canvas by clicking the left mouse button and dragging it over the canvas.



Above, you can see my not very artistic attempts at drawing. You can almost certainly do better. You should also notice that the program no longer prints the mouse position in the IDLE output window, which is because only one function can be bound to a particular event.

## Tkinter events

Tkinter events are very powerful and flexible. Let's look at the event we've been using for drawing. Below is the statement we used to link the `mouse_move` function to the event where the mouse is moved with a button held down.

[Click here to view code image](#)

```
c.bind('<B1-Motion>', mouse_move)
```

The event identifier is the string '`<B1-Motion>`'. We can break this string down into two components. The first part is called the *modifier*. You can think of this as a condition that must be satisfied for the event to be generated. In our