

APPLICATION DEVELOPMENT USING PYTHON [(Effective from the academic year 2018 -2019) SEMESTER – V			
Course Code	18CS55	IA Marks	40
Number of Lecture Hours/Week	03	Exam Marks	60
Total Number of Lecture Hours	40	Exam Hours	03
CREDITS – 03			
Course Learning Objectives: This course (18CS55) will enable students to			
<ul style="list-style-type: none"> Learn the syntax and semantics of Python programming language. Illustrate the process of structuring the data using lists, tuples and dictionaries. Demonstrate the use of built-in functions to navigate the file system. Implement the Object Oriented Programming concepts in Python. Appraise the need for working with various documents like Excel, PDF, Word and Others. 			
Module – 1			Teaching Hours
Python Basics , Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program, Flow control , Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit(), Functions , def Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling, A Short Program: Guess the Number Textbook 1: Chapters 1 – 3 RBT: L1, L2			08
Module – 2			
Lists , The List Data Type, Working with Lists, Augmented Assignment Operators, Methods, Example Program: Magic 8 Ball with a List, List-like Types: Strings and Tuples, References, Dictionaries and Structuring Data , The Dictionary Data Type, Pretty Printing, Using Data Structures to Model Real-World Things, Manipulating Strings , Working with Strings, Useful String Methods, Project: Password Locker, Project: Adding Bullets to Wiki Markup Textbook 1: Chapters 4 – 6 RBT: L1, L2, L3			08
Module – 3			
Pattern Matching with Regular Expressions , Finding Patterns of Text Without Regular Expressions, Finding Patterns of Text with Regular Expressions, More Pattern Matching with Regular Expressions, Greedy and Nongreedy Matching, The findall() Method, Character Classes, Making Your Own Character Classes, The Caret and Dollar Sign Characters, The Wildcard Character, Review of Regex Symbols, Case-Insensitive Matching, Substituting Strings with the sub() Method, Managing Complex Regexes, Combining re .IGNORECASE, re .DOTALL, and re .VERBOSE, Project: Phone Number and Email Address Extractor, Reading and Writing Files , Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module, Saving Variables with the pprint.pformat() Function, Project: Generating Random Quiz Files, Project: Multiclipboard, Organizing Files , The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates, Project: Backing Up a Folder into a ZIP File, Debugging , Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE's Debugger. Textbook 1: Chapters 7 – 10			08
RBT: L1, L2, L3			
Module – 4			

<p>Classes and objects, Programmer-defined types, Attributes, Rectangles, Instances as return values, Objects are mutable, Copying, Classes and functions, Time, Pure functions, Modifiers, Prototyping versus planning, Classes and methods, Object-oriented features, Printing objects, Another example, A more complicated example, The init method, The __str__ method, Operator overloading, Type-based dispatch, Polymorphism, Interface and implementation, Inheritance, Card objects, Class attributes, Comparing cards, Decks, Printing the deck, Add, remove, shuffle and sort, Inheritance, Class diagrams, Data encapsulation</p> <p>Textbook 2: Chapters 15 – 18 RBT: L1, L2, L3</p>	08
Module – 5	
<p>Web Scraping, Project: MAPIT.PY with the webbrowser Module, Downloading Files from the Web with the requests Module, Saving Downloaded Files to the Hard Drive, HTML, Parsing HTML with the BeautifulSoup Module, Project: “I’m Feeling Lucky” Google Search, Project: Downloading All XKCD Comics, Controlling the Browser with the selenium Module, Working with Excel Spreadsheets, Excel Documents, Installing the openpyxl Module, Reading Excel Documents, Project: Reading Data from a Spreadsheet, Writing Excel Documents, Project: Updating a Spreadsheet, Setting the Font Style of Cells, Font Objects, Formulas, Adjusting Rows and Columns, Charts, Working with PDF and Word Documents, PDF Documents, Project: Combining Select Pages from Many PDFs, Word Documents, Working with CSV files and JSON data, The csv Module, Project: Removing the Header from CSV Files, JSON and APIs, The json Module, Project: Fetching Current Weather Data</p> <p>Textbook 1: Chapters 11 – 14 RBT: L1, L2, L3</p>	08
Course Outcomes: After studying this course, students will be able to	
<ul style="list-style-type: none"> • Demonstrate proficiency in handling of loops and creation of functions. • Identify the methods to create and manipulate lists, tuples and dictionaries. • Discover the commonly used operations involving regular expressions and file system. • Interpret the concepts of Object-Oriented Programming as used in Python. • Determine the need for scraping websites and working with CSV, JSON and other file formats. 	
Question paper pattern:	
<ul style="list-style-type: none"> • The question paper will have ten questions. • Each full Question consisting of 20 marks • There will be 2 full questions (with a maximum of four sub questions) from each module. • Each full question will have sub questions covering all the topics under a module. • The students will have to answer 5 full questions, selecting one full question from each module. 	
Text Books:	
<ol style="list-style-type: none"> 1. Al Sweigart, “Automate the Boring Stuff with Python”, 1st Edition, No Starch Press, 2015. (Available under CC-BY-NC-SA license at https://automatetheboringstuff.com/) (Chapters 1 to 18) 2. Allen B. Downey, “Think Python: How to Think Like a Computer Scientist”, 2nd Edition, Green Tea Press, 2015. (Available under CC-BY-NC license at http://greenteapress.com/thinkpython2/thinkpython2.pdf) (Chapters 13, 15, 16, 17, 18) (Download pdf/html files from the above links) 	
Reference Books:	
<ol style="list-style-type: none"> 1. Gowrishankar S, Veena A, “Introduction to Python Programming”, 1st Edition, CRC Press/Taylor & Francis, 2018. ISBN-13: 978-0815394372 	

PYTHON BASICS

The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

You will, however, have to learn some basic programming concepts before you can do anything. Like a wizard in training, you might think these concepts seem arcane and tedious, but with some knowledge and practice, you'll be able to command your computer like a magic wand and perform incredible feats.

This chapter has a few examples that encourage you to type into the *interactive shell*, also called the *REPL* (Read-Evaluate-Print Loop), which lets you run (or *execute*) Python instructions one at a time and instantly shows you the results. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

ENTERING EXPRESSIONS INTO THE INTERACTIVE SHELL

You can run the interactive shell by launching the Mu editor, which you should have downloaded when going through the setup instructions in the Preface. On Windows, open the Start menu, type "Mu," and open the Mu app. On macOS, open your Applications folder and double-click **Mu**. Click the **New** button and save an empty file as *blank.py*. When you run this blank file by clicking the **Run** button or pressing F5, it will open the interactive shell, which will open as a new pane that opens at the bottom of the Mu editor's window. You should see a `>>>` prompt in the interactive shell.

Enter `2 + 2` at the prompt to have Python do some simple math. The Mu window should now look like this:

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

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as 2) and *operators* (such as +), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, 4. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

ERRORS ARE OKAY!

Programs will crash if they contain code the computer can't understand, which will cause Python to show an error message. An error message won't break your computer, though, so don't be afraid to make mistakes. A *crash* just means the program stopped running unexpectedly.

If you want to know more about an error, you can search for the exact error message text online for more information. You can also check out the resources at <https://nostarch.com/automatestuff2/> to see a list of common Python error messages and their meanings.

You can use plenty of other operators in Python expressions, too. For example, [Table 1-1](#) lists all the math operators in Python.

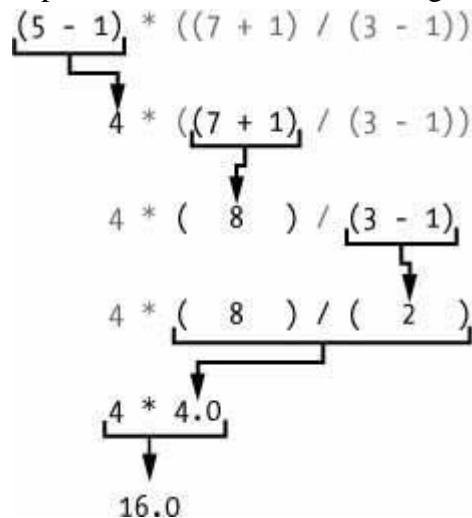
Table 1-1: Math Operators from Highest to Lowest Precedence

Operator Operation		Example Evaluates to . . .	
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

The *order of operations* (also called *precedence*) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Whitespace in between the operators and values doesn't matter for Python (except for the indentation at the beginning of the line), but a single space is convention. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2 + 2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it down to a single value. Python will keep evaluating parts of the expression until it becomes a single value, as shown here:



These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you enter a bad Python instruction, Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```

>>> 5 +
      File "<stdin>", line 1
        5 +
          ^
      SyntaxError: invalid syntax
>>> 42 + 5 + * 2
      File "<stdin>", line 1
        42 + 5 + * 2
              ^
      SyntaxError: invalid syntax

```

You can always test to see whether an instruction works by entering it into the interactive shell. Don't worry about breaking the computer: the worst that could happen is that Python responds with an error message. Professional software developers get error messages while writing code all the time.

THE INTEGER, FLOATING-POINT, AND STRING DATA TYPES

Remember that expressions are just values combined with operators, and they always evaluate down to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most common data types in Python are listed in [Table 1-2](#). The values -2 and 30, for example, are said to be *integer* values. The integer (or *int*) data type

indicates values that are whole numbers. Numbers with a decimal point, such as 3.14, are called *floating-point numbers* (or *floats*). Note that even though the value 42 is an integer, the value 42.0 would be a floating-point number.

Table 1-2: Common Data Types

Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

Python programs can also have text values called *strings*, or *strs* (pronounced “stirs”). Always surround your string in single quote (') characters (as in 'Hello' or 'Goodbye cruel world!') so Python knows where the string begins and ends. You can even have a string with no characters in it, "", called a *blank string* or an *empty string*. Strings are explained in greater detail in [Chapter 4](#).

If you ever see the error message `SyntaxError: EOL while scanning string literal`, you probably forgot the final single quote character at the end of the string, such as in this example:

```
>>> 'Hello,                                     world!
SyntaxError: EOL while scanning string literal
```

STRING CONCATENATION AND REPLICATION

The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used on two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Alice' + 42
TypeError: can only concatenate str (not "int") to str
```

The error message `can only concatenate str (not "int") to str` means that Python thought you were trying to concatenate an integer to the string 'Alice'. Your code will have to explicitly convert the integer to a string because Python cannot do this automatically. (Converting data types will be explained in “[Dissecting Your Program](#)” on [page 13](#) when we talk about the `str()`, `int()`, and `float()` functions.)

The `*` operator multiplies two integer or floating-point values. But when the `*` operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original string a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The `*` operator can be used with only two numeric values (for multiplication), or one string value and one integer value (for string replication). Otherwise, Python will just display an error message, like the following:

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'

>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

It makes sense that Python wouldn't understand these expressions: you can't multiply two words, and it's hard to replicate an arbitrary string a fractional number of times.

STORING VALUES IN VARIABLES

A *variable* is like a box in the computer's memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, then a variable named `spam` will have the integer value 42 stored in it.

Think of a variable as a labeled box that a value is placed in, as in [Figure 1-1](#).

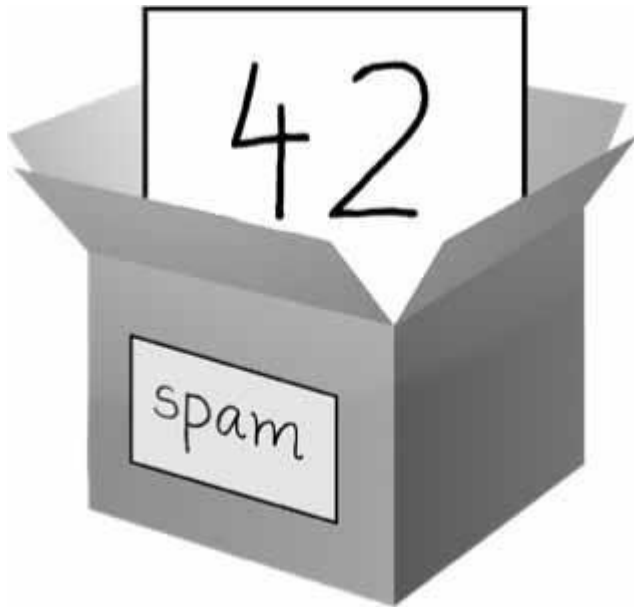


Figure 1-1: `spam = 42` is like telling the program, “The variable `spam` now has the integer value 42 in it.”

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
>>> spam
40
>>> eggs = 2
❷ >>> spam + eggs
42
>>> spam + eggs + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to 42 instead of 40 at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in [Figure 1-2](#), the `spam` variable in this example stores 'Hello' until you replace the string with 'Goodbye'.

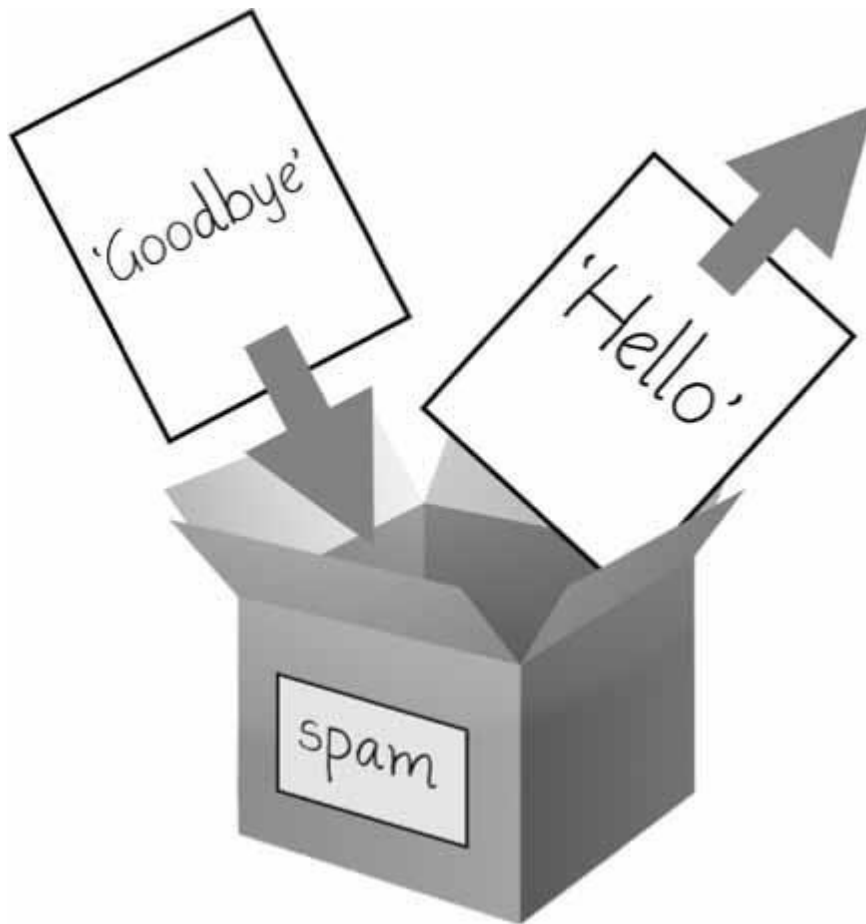


Figure 1-2: When a new value is assigned to a variable, the old one is forgotten.

Variable Names

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! Most of this book's examples (and Python's documentation) use generic variable names like *spam*, *eggs*, and *bacon*, which come from the Monty Python "Spam" sketch. But in your programs, a descriptive name will help make your code more readable.

Though you can name your variables almost anything, Python does have some naming restrictions. [Table 1-3](#) has examples of legal variable names. You can name a variable anything as long as it obeys the following three rules:

- It can be only one word with no spaces.
- It can use only letters, numbers, and the underscore (`_`) character.
- It can't begin with a number.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
<code>current_balance</code>	<code>current-balance</code> (hyphens are not allowed)
<code>currentBalance</code>	<code>current balance</code> (spaces are not allowed)
<code>account4</code>	<code>4account</code> (can't begin with a number)
<code>_42</code>	<code>42</code> (can't begin with a number)

Valid variable names Invalid variable names

TOTAL_SUM	TOTAL_\$UM (special characters like \$ are not allowed)
hello	'hello' (special characters like ' are not allowed)

Variable names are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four different variables. Though Spam is a valid variable you can use in a program, it is a Python convention to start your variables with a lowercase letter.

This book uses *camelcase* for variable names instead of underscores; that is, variables lookLikeThis instead of looking_like_this. Some experienced programmers may point out that the official Python code style, PEP 8, says that underscores should be used. I unapologetically prefer camelcase and point to the “A Foolish Consistency Is the Hobgoblin of Little Minds” section in PEP 8 itself:

Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn’t apply. When in doubt, use your best judgment.

YOUR FIRST PROGRAM

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs, you’ll type the instructions into the file editor. The *file editor* is similar to text editors such as Notepad or TextMate, but it has some features specifically for entering source code. To open a new file in Mu, click the **New** button on the top row.

The window that appears should contain a cursor awaiting your input, but it’s different from the interactive shell, which runs Python instructions as soon as you press ENTER. The file editor lets you type in many instructions, save the file, and run the program. Here’s how you can tell the difference between the two:


- The interactive shell window will always be the one with the >>> prompt.
- The file editor window will not have the >>> prompt.

Now it’s time to create your first program! When the file editor window opens, enter the following into it:

```
❶ # This program says hello and asks for my name.

❷ print('Hello, world!')
   print('What is your name?')  # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
   print(len(myName))
❻ print('What is your age?')  # ask for their age
   myAge = input()
   print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Once you’ve entered your source code, save it so that you won’t have to retype it each time you start Mu. Click the **Save** button, enter *hello.py* in the File Name field, and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit Mu, you won't lose the code. As a shortcut, you can press CTRL-S on Windows and Linux or -S on macOS to save your file.

Once you've saved, let's run our program. Press the **F5** key. Your program should run in the interactive shell window. Remember, you have to press **F5** from the file editor window, not the interactive shell window. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Python 3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018, 19:02:22) [MSC v.1913 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART
=====
>>>
Hello, world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.)

You can close the file editor by clicking the X at the top of the window. To reload a saved program, select **File ▶ Open...** from the menu. Do that now, and in the window that appears, choose **hello.py** and click the **Open** button. Your previously saved **hello.py** program should open in the file editor window.

You can view the execution of a program using the Python Tutor visualization tool at <http://pythontutor.com/>. You can see the execution of this particular program at <https://autbor.com/hello.py/>. Click the forward button to move through each step of the program's execution. You'll be able to see how the variables' values and the output change.

DISSECTING YOUR PROGRAM

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*.

```
❶ # This program says hello and asks for my name.
```

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program isn't working. You can remove the # later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This can make your code easier to read, like paragraphs in a book.

The print() Function

The print() function displays the string value inside its parentheses on the screen.

```
❷ print('Hello, world!')  
   print('What is your name?') # ask for their name
```

The line print('Hello, world!') means “Print out the text in the string 'Hello, world!'.” When Python executes this line, you say that Python is *calling* the print() function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

NOTE

You can also use this function to put a blank line on the screen; just call print() with nothing in between the parentheses.

When you write a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book, you'll see print() rather than print. [Chapter 3](#) describes functions in more detail.

The input() Function

The input() function waits for the user to type some text on the keyboard and press ENTER.

```
❸ myName = input()
```

This function call evaluates to a string equal to the user's text, and the line of code assigns the myName variable to this string value.

You can think of the input() function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to myName = 'Al'.

If you call input() and see an error message, like NameError: name 'Al' is not defined, the problem is that you're running the code with Python 2 instead of Python 3.

Printing the User's Name

The following call to print() actually contains the expression 'It is good to meet you, ' + myName between the parentheses.

```
❹ print('It is good to meet you, ' + myName)
```

Remember that expressions can always evaluate to a single value. If 'Al' is the value stored in myName on line ③, then this expression evaluates to 'It is good to meet you, Al'. This single string value is then passed to print(), which prints it on the screen.

The len() Function

You can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

```
⑤ print('The length of your name is:')  
  print(len(myName))
```

Enter the following into the interactive shell to try this:

```
>>> len('hello')  
5  
>>> len('My very energetic monster just scarfed nachos.')  
46  
>>> len('')  
0
```

Just like those examples, len(myName) evaluates to an integer. It is then passed to print() to be displayed on the screen. The print() function allows you to pass it either integer values or string values, but notice the error that shows up when you type the following into the interactive shell:

```
>>> print('I am ' + 29 + ' years old.')  
Traceback (most recent call last):  
  File "<pyshell#6>", line 1, in <module>  
    print('I am ' + 29 + ' years old.')  
TypeError: can only concatenate str (not "int") to str
```

The print() function isn't causing that error, but rather it's the expression you tried to pass to print(). You get the same error message if you type the expression into the interactive shell on its own.

```
>>> 'I am ' + 29 + ' years old.'  
Traceback (most recent call last):  
  File "<pyshell#7>", line 1, in <module>  
    'I am ' + 29 + ' years old.'  
TypeError: can only concatenate str (not "int") to str
```

Python gives an error because the + operator can only be used to add two integers together or concatenate two strings. You can't add an integer to a string, because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

The str(), int(), and float() Functions

If you want to concatenate an integer such as 29 with a string to pass to `print()`, you'll need to get the value '29', which is the string form of 29. The `str()` function can be passed an integer value and will evaluate to a string value version of the integer, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because `str(29)` evaluates to '29', the expression 'I am ' + `str(29)` + ' years old.' evaluates to 'I am ' + '29' + ' years old.', which in turn evaluates to 'I am 29 years old.'. This is the value that is passed to the `print()` function.

The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions and watch what happens.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the `str()`, `int()`, and `float()` functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The `str()` function is handy when you have an integer or float that you want to concatenate to a string. The `int()` function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the `input()` function always returns a string, even if the user enters a number. Enter `spam = input()` into the interactive shell and enter **101** when it waits for your text.

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside spam isn't the integer 101 but the string '101'. If you want to do math using the value in spam, use the `int()` function to get the integer form of spam and then store this as the new value in spam.

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the spam variable as an integer instead of a string.

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to `int()` that it cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

The `int()` function is also useful if you need to round a floating-point number down.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

You used the `int()` and `str()` functions in the last three lines of your program to get a value of the appropriate data type for the code.

```
❹ print('What is your age?') # ask for their age
   myAge = input()
   print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

TEXT AND NUMBER EQUIVALENCE

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.


```
>>> 42 == '42'  
False  
>>> 42 == 42.0  
True  
>>> 42.0 == 0042.000  
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.

The `myAge` variable contains the value returned from `input()`. Because the `input()` function always returns a string (even if the user typed in a number), you can use the `int(myAge)` code to return an integer value of the string in `myAge`. This integer value is then added to 1 in the expression `int(myAge) + 1`.

The result of this addition is passed to the `str()` function: `str(int(myAge) + 1)`. The string value returned is then concatenated with the strings 'You will be ' and ' in a year.' to evaluate to one large string value. This large string is finally passed to `print()` to be displayed on the screen.

Let's say the user enters the string '4' for `myAge`. The string '4' is converted to an integer, so you can add one to it. The result is 5. The `str()` function converts the result back to a string, so you can concatenate it with the second string, ' in a year.', to create the final message. These evaluation steps would look something like the following:

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')  
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')  
print('You will be ' + str(    4 + 1    ) + ' in a year.')  
print('You will be ' + str(        5        ) + ' in a year.')  
print('You will be ' +          '5'          + ' in a year.')  
print('You will be 5'                + ' in a year.')  
print('You will be 5 in a year.')
```

1. Which of the following are operators, and which are values?

```
*  
'hello'  
-88.8  
-  
/
```

+
5

2. Which of the following is a variable, and which is a string?

spam
'spam'

3. Name three data types.

4. What is an expression made up of? What do all expressions do?

5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?

6. What does the variable `bacon` contain after the following code runs?

`bacon = 20`
`bacon + 1`

7. What should the following two expressions evaluate to?

`'spam' + 'spamspam'`
`'spam' * 3`

8. Why is `eggs` a valid variable name while `100` is invalid?

9. What three functions can be used to get the integer, floating-point number, or string version of a value?

10. Why does this expression cause an error? How can you fix it?

`'I have eaten ' + 99 + ' burritos.'`

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled “Built-in Functions.” Skim the list of other functions Python has, look up what the `round()` function does, and experiment with it in the interactive shell.

FLOW CONTROL

So, you know the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another like a weekend errand list. Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. [Figure 2-1](#) shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

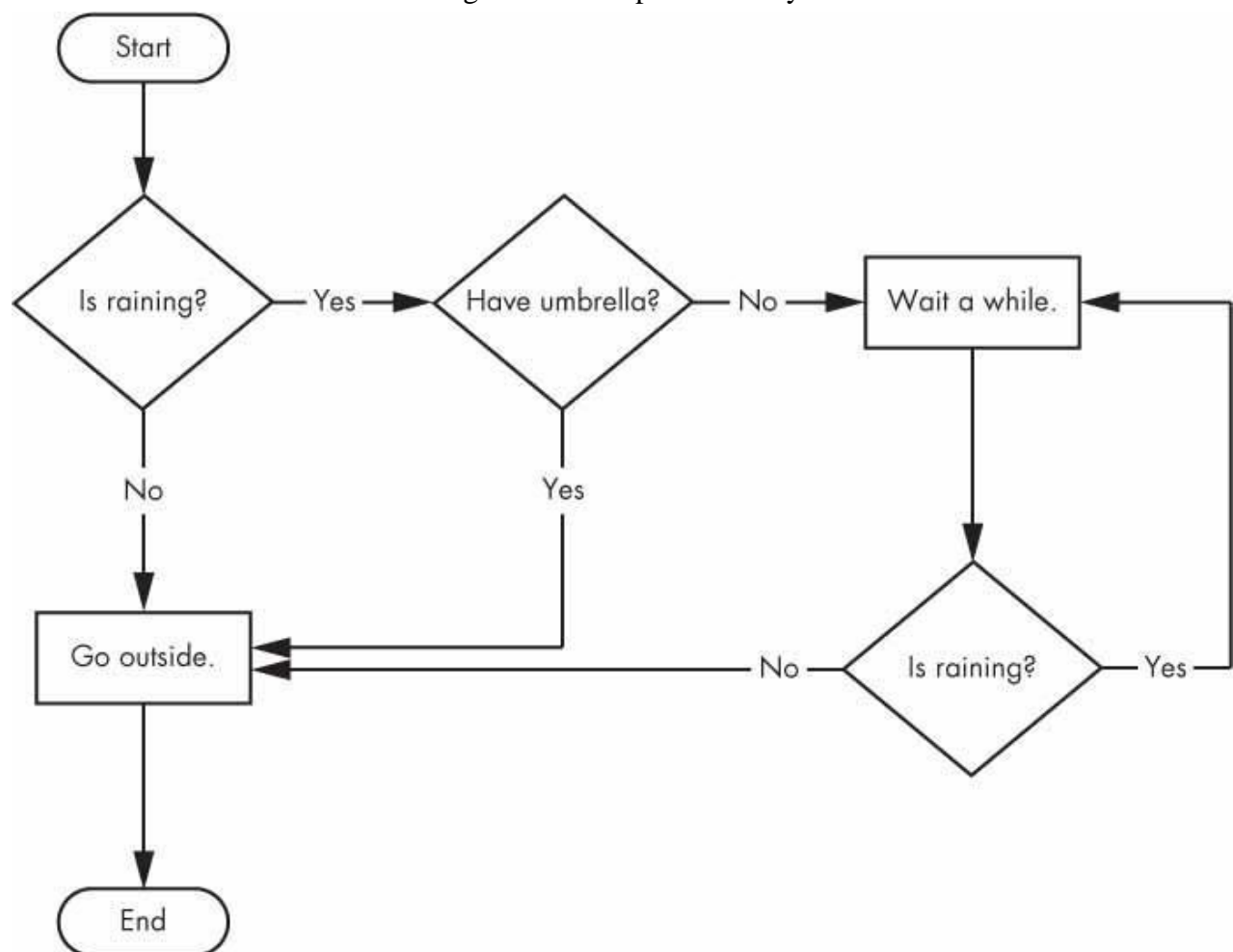


Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

BOOLEAN VALUES

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: True and False. (Boolean is capitalized because the data type is named after mathematician George Boole.) When entered as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Enter the following into the interactive shell. (Some of these instructions are intentionally incorrect, and they'll cause error messages to appear.)

```
❶ >>> spam = True
>>> spam
True
❷ >>> true
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    true
NameError: name 'true' is not defined
❸ >>> True = 2 + 2
SyntaxError: can't assign to keyword
```

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use True and False for variable names ❸, Python will give you an error message.

COMPARISON OPERATORS

Comparison operators, also called *relational operators*, compare two values and evaluate down to a single Boolean value. [Table 2-1](#) lists the comparison operators.

Table 2-1: Comparison Operators

Operator Meaning	
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

These operators evaluate to True or False depending on the values you give them. Let's try some operators now, starting with == and !=.

```
>>> 42 == 42
True
>>> 42 == 99
```

```
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, == (equal to) evaluates to True when the values on both sides are the same, and != (not equal to) evaluates to True when the two values are different. The == and != operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` ❶ evaluates to False because Python considers the integer 42 to be different from the string '42'.

The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the == operator (equal to) has two equal signs, while the = operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The == operator (equal to) asks whether two values are the same as each other.
- The = operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the == operator (equal to) consists of two characters, just like the != operator (not equal to) consists of two characters.

You'll often use comparison operators to compare a variable's value to some other value, like in the `eggCount <= 42` ❶ and `myAge >= 10` ❷ examples. (After all, instead of entering `'dog' != 'cat'` in your code, you could have just entered `True`.) You'll see more examples of this later when you learn about flow control statements.

BOOLEAN OPERATORS

The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the and operator.

Binary Boolean Operators

The and and or operators always take two Boolean values (or expressions), so they're considered *binary* operators. The and operator evaluates an expression to True if *both* Boolean values are True; otherwise, it evaluates to False. Enter some expressions using and into the interactive shell to see it in action.

```
>>> True and True
True
>>> True and False
False
```

A *truth table* shows every possible result of a Boolean operator. [Table 2-2](#) is the truth table for the and operator.

Table 2-2: The and Operator's Truth Table

Expression	Evaluates to . . .
True and True	True
True and False	False
False and True	False
False and False	False

On the other hand, the or operator evaluates an expression to True if *either* of the two Boolean values is True. If both are False, it evaluates to False.

```
>>> False or True
True
>>> False or False
False
```

You can see every possible outcome of the or operator in its truth table, shown in [Table 2-3](#).

Table 2-3: The or Operator's Truth Table

Expression	Evaluates to . . .
True or True	True
True or False	True
False or True	True
False or False	False

The not Operator

Unlike and and or, the not operator operates on only one Boolean value (or expression). This makes it a *unary* operator. The not operator simply evaluates to the opposite Boolean value.

```
>>> not True
False
❶ >>> not not not not True
True
```

Much like using double negatives in speech and writing, you can nest not operators ❶, though there's never not no reason to do this in real programs. [Table 2-4](#) shows the truth table for not.

Table 2-4: The not Operator's Truth Table

Expression	Evaluates to . . .
not True	False
not False	True

MIXING BOOLEAN AND COMPARISON OPERATORS

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the and, or, and not operators are called Boolean operators because they always operate on the Boolean values True and False. While expressions like $4 < 5$ aren't Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for $(4 < 5)$ and $(5 < 6)$ as the following:


```
(4 < 5) and (5 < 6)
  ↓
True and (5 < 6)
  ↓
True and True
  ↓
True
```

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the not operators first, then the and operators, and then the or operators.

ELEMENTS OF FLOW CONTROL

Flow control statements often start with a part called the *condition* and are always followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

- Blocks begin when the indentation increases.
- Blocks can contain other blocks.
- Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
    ❶ print('Hello, Mary')
    if password == 'swordfish':
        ❷ print('Access granted.')
```

else:

❸ print('Wrong password.')

You can view the execution of this program at <https://autbor.com/blocks/>. The first block of code ❶ starts at the line `print('Hello, Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access Granted.')`. The third block ❸ is also one line long: `print('Wrong password.')`.

PROGRAM EXECUTION

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. The *program execution* (or simply, *execution*) is a term for the current instruction being executed. If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find yourself jumping around the source code based on conditions, and you'll probably skip entire clauses.

FLOW CONTROL STATEMENTS

Now, let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in [Figure 2-1](#), and they are the actual decisions your programs will make.

if Statements

The most common type of flow control statement is the if statement. An if statement's clause (that is, the block following the if statement) will execute if the statement's condition is True. The clause is skipped if the condition is False.

In plain English, an if statement could be read as, "If this condition is true, execute the code in the clause." In Python, an if statement consists of the following:

- The if keyword
- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the if clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend name was assigned some value earlier.)

```
if name == 'Alice':  
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This if statement's clause is the block with `print('Hi, Alice.')`. [Figure 2-2](#) shows what a flowchart of this code would look like.

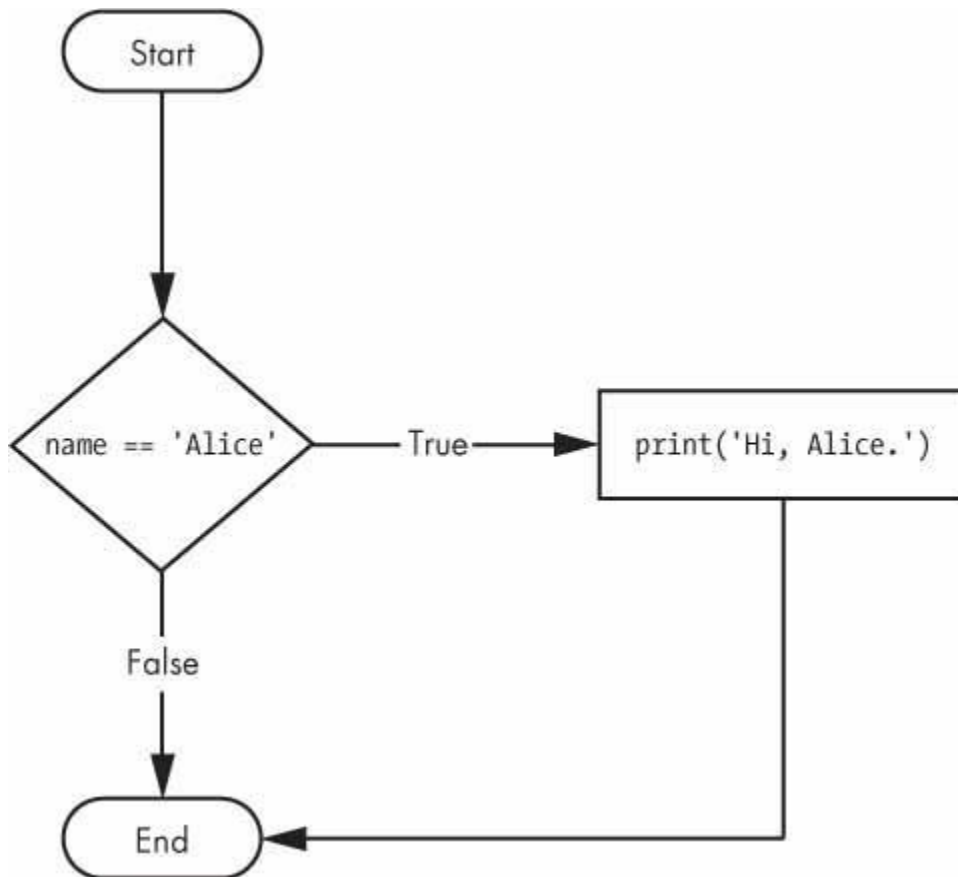


Figure 2-2: The flowchart for an if statement

else Statements

An if clause can optionally be followed by an else statement. The else clause is executed only when the if statement's condition is False. In plain English, an else statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An else statement doesn't have a condition, and in code, an else statement always consists of the following:

- The else keyword
- A colon
- Starting on the next line, an indented block of code (called the else clause)

Returning to the Alice example, let's look at some code that uses an else statement to offer a different greeting if the person's name isn't Alice.

```
if name == 'Alice':  
    print('Hi, Alice.')  
else:  
    print('Hello, stranger.')
```

Figure 2-3 shows what a flowchart of this code would look like.

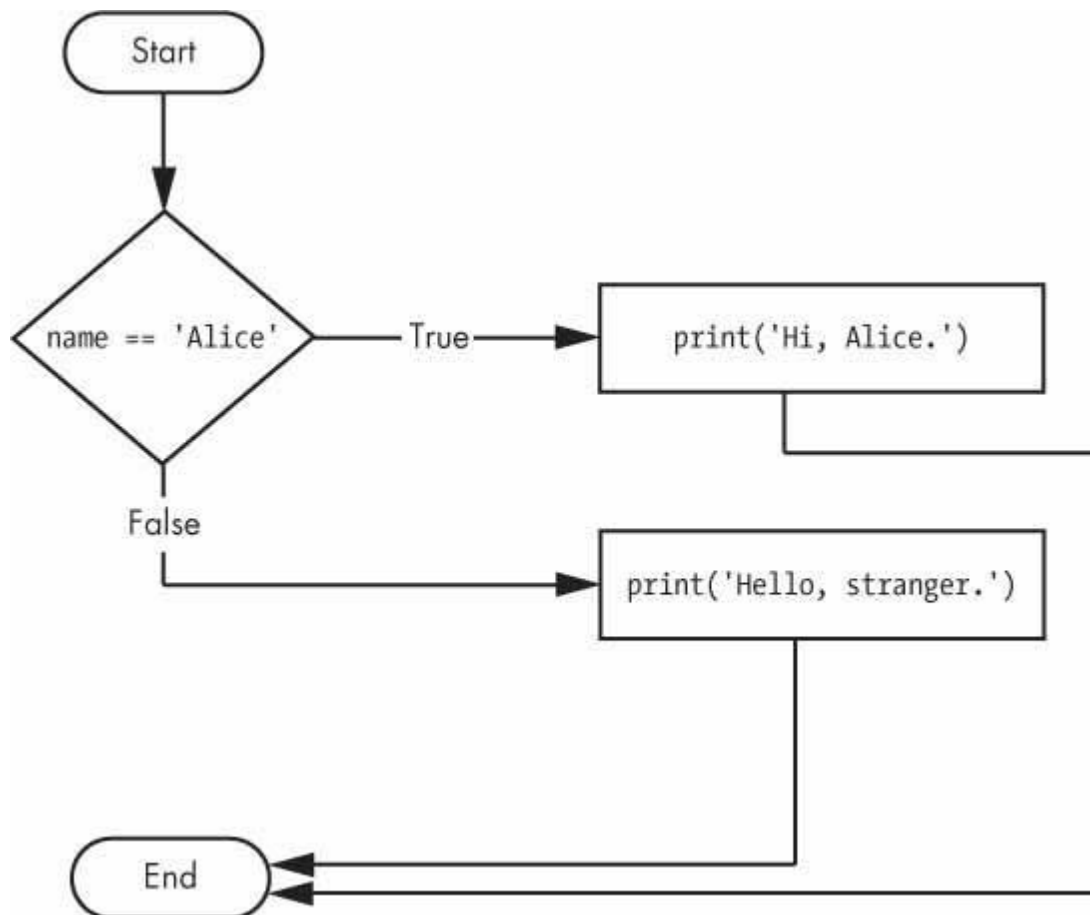


Figure 2-3: The flowchart for an else statement

elif Statements

While only one of the if or else clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The elif statement is an “else if” statement that always follows an if or another elif statement. It provides another condition that is checked only if all of the previous conditions were False. In code, an elif statement always consists of the following:

- The elif keyword
- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the elif clause)

Let’s add an elif to the name checker to see this statement in action.

```
if name == 'Alice':  
    print('Hi, Alice.')  
elif age < 12:  
    print('You are not Alice, kiddo.')
```

This time, you check the person’s age, and the program will tell them something different if they’re younger than 12. You can see the flowchart for this in [Figure 2-4](#).

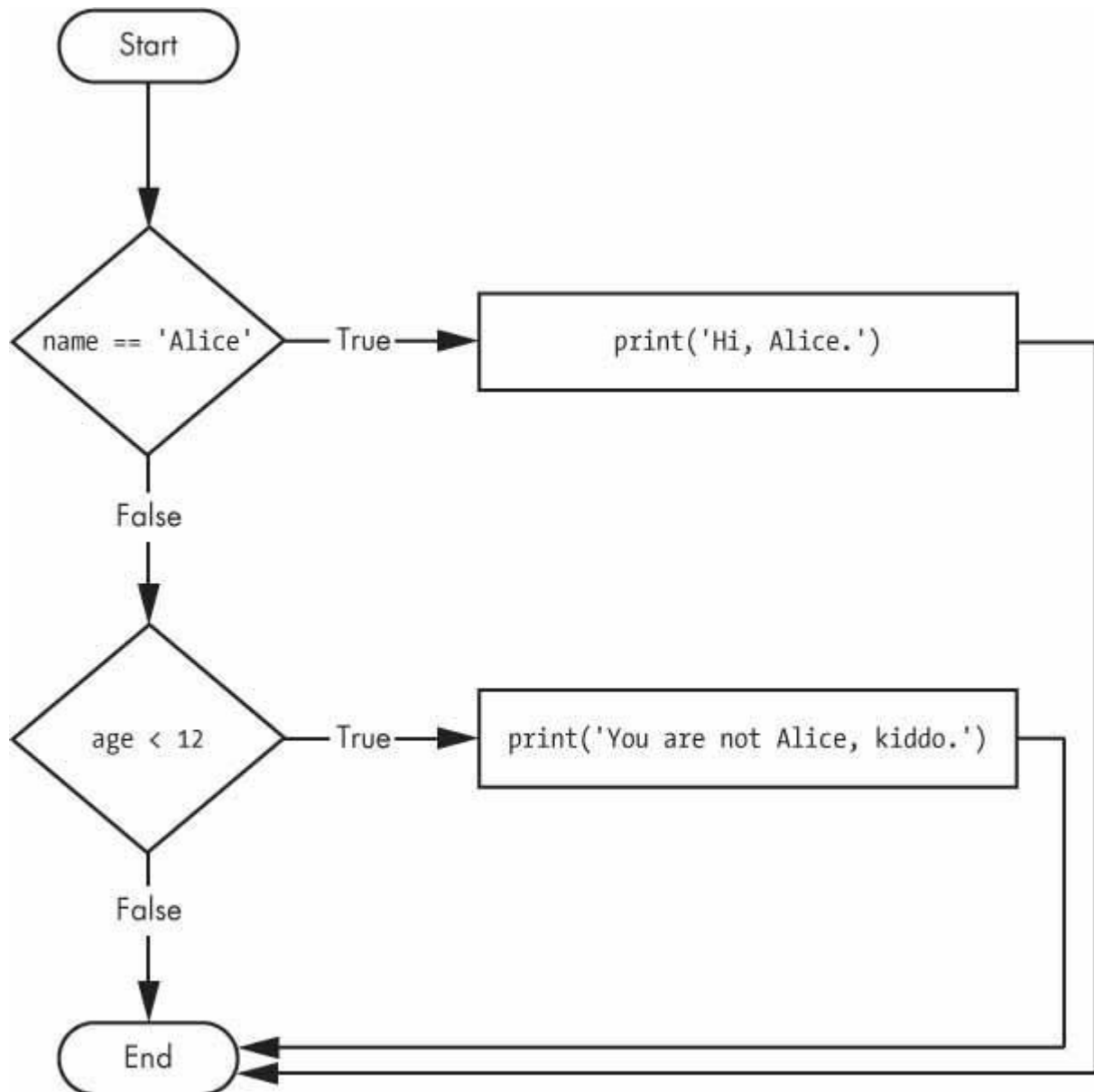


Figure 2-4: The flowchart for an elif statement

The elif clause executes if `age < 12` is True and `name == 'Alice'` is False. However, if both of the conditions are False, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of elif statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be True, the rest of the elif clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as *vampire.py*:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
```

```
print('Unlike you, Alice is not an undead, immortal vampire.')  
elif age > 100:  
    print('You are not Alice, grannie.')
```

You can view the execution of this program at <https://autbor.com/vampire/>. Here, I've added two more elif statements to make the name checker greet a person with different answers based on age. [Figure 2-5](#) shows the flowchart for this.

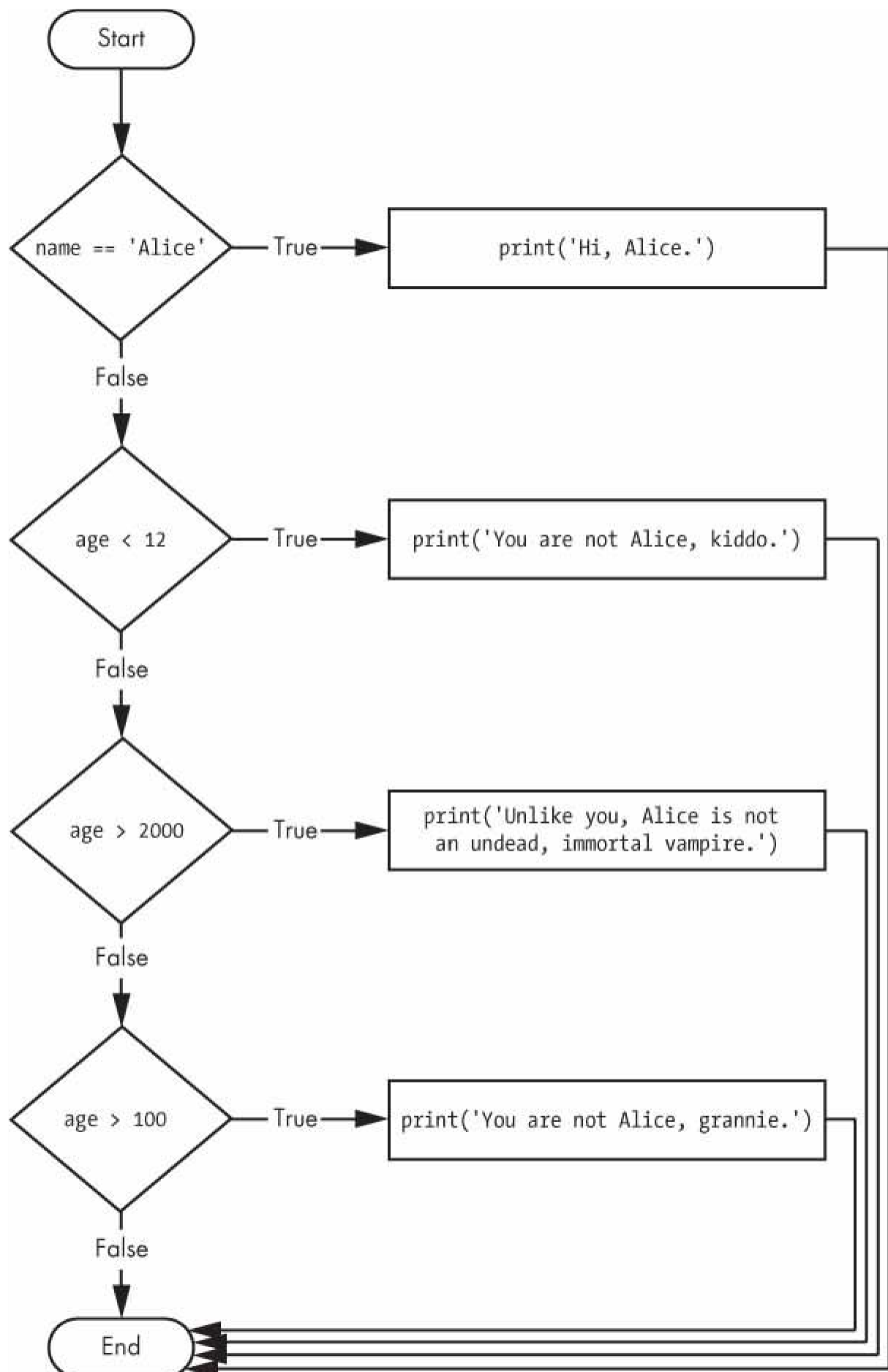


Figure 2-5: The flowchart for multiple *elif* statements in the *vampire.py* program

The order of the *elif* statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the *elif* clauses are automatically skipped once a *True* condition has been found, so if you swap around some of the clauses in *vampire.py*, you run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
❶ elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

You can view the execution of this program at <https://autbor.com/vampire2/>. Say the age variable contains the value 3000 before this code is executed. You might expect the code to print the string 'Unlike you, Alice is not an undead, immortal vampire.'. However, because the *age > 100* condition is *True* (after all, 3,000 *is* greater than 100) ❶, the string 'You are not Alice, grannie.' is printed, and the rest of the *elif* statements are automatically skipped. Remember that at most only one of the clauses will be executed, and for *elif* statements, the order matters!

Figure 2-6 shows the flowchart for the previous code. Notice how the diamonds for *age > 100* and *age > 2000* are swapped.

Optionally, you can have an *else* statement after the last *elif* statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every *if* and *elif* statement are *False*, then the *else* clause is executed. For example, let's re-create the Alice program to use *if*, *elif*, and *else* clauses.

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

You can view the execution of this program at <https://autbor.com/littlekid/>. Figure 2-7 shows the flowchart for this new code, which we'll save as *littleKid.py*.

In plain English, this type of flow control structure would be "If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else." When you use *if*, *elif*, and *else* statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-6. First, there is always exactly one *if* statement.

Any elif statements you need should follow the if statement. Second, if you want to be sure that at least one clause is executed, close the structure with an else statement.

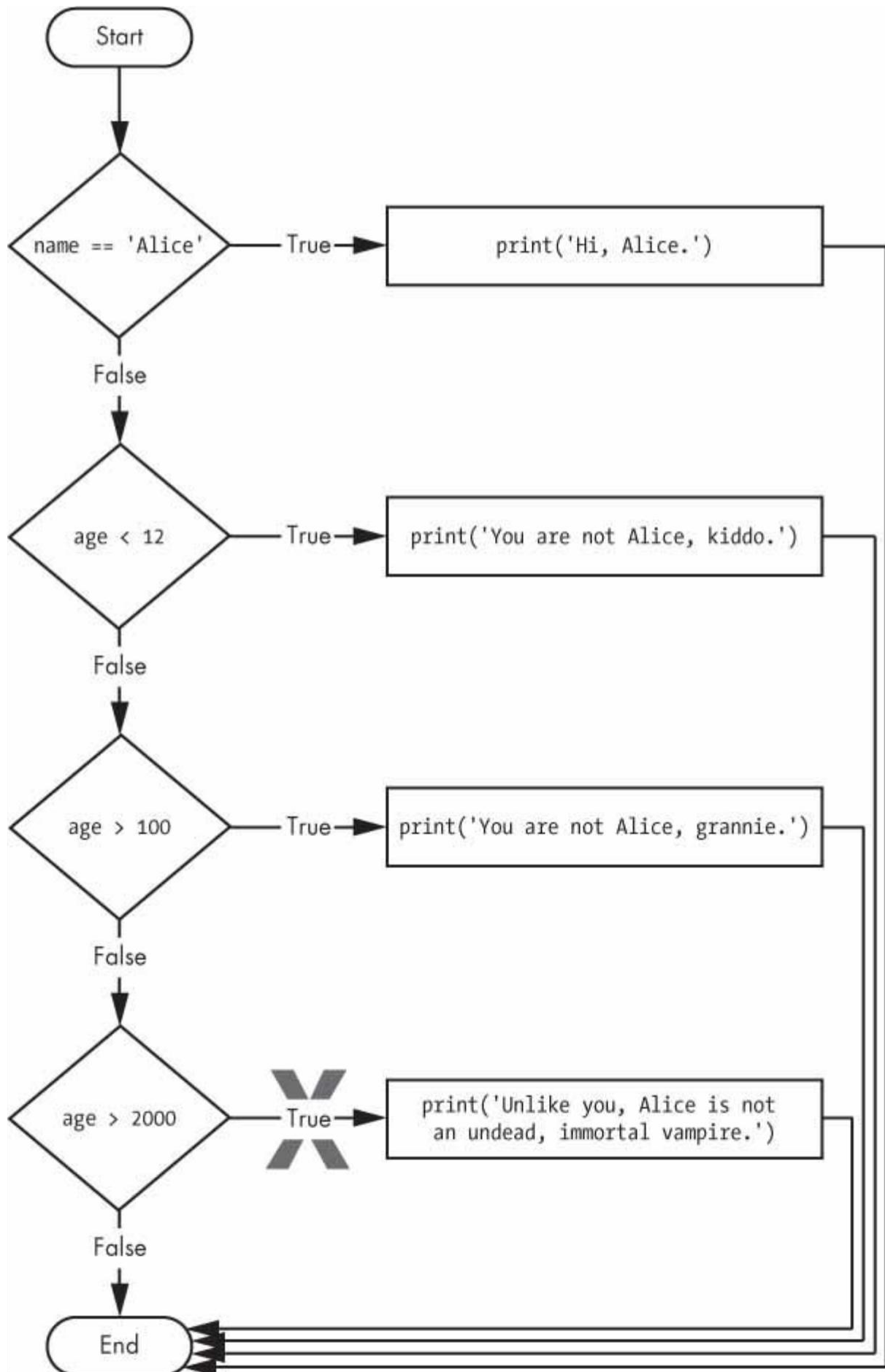


Figure 2-6: The flowchart for the vampire2.py program. The X path will logically never happen, because if age were greater than 2000, it would have already been greater than 100.

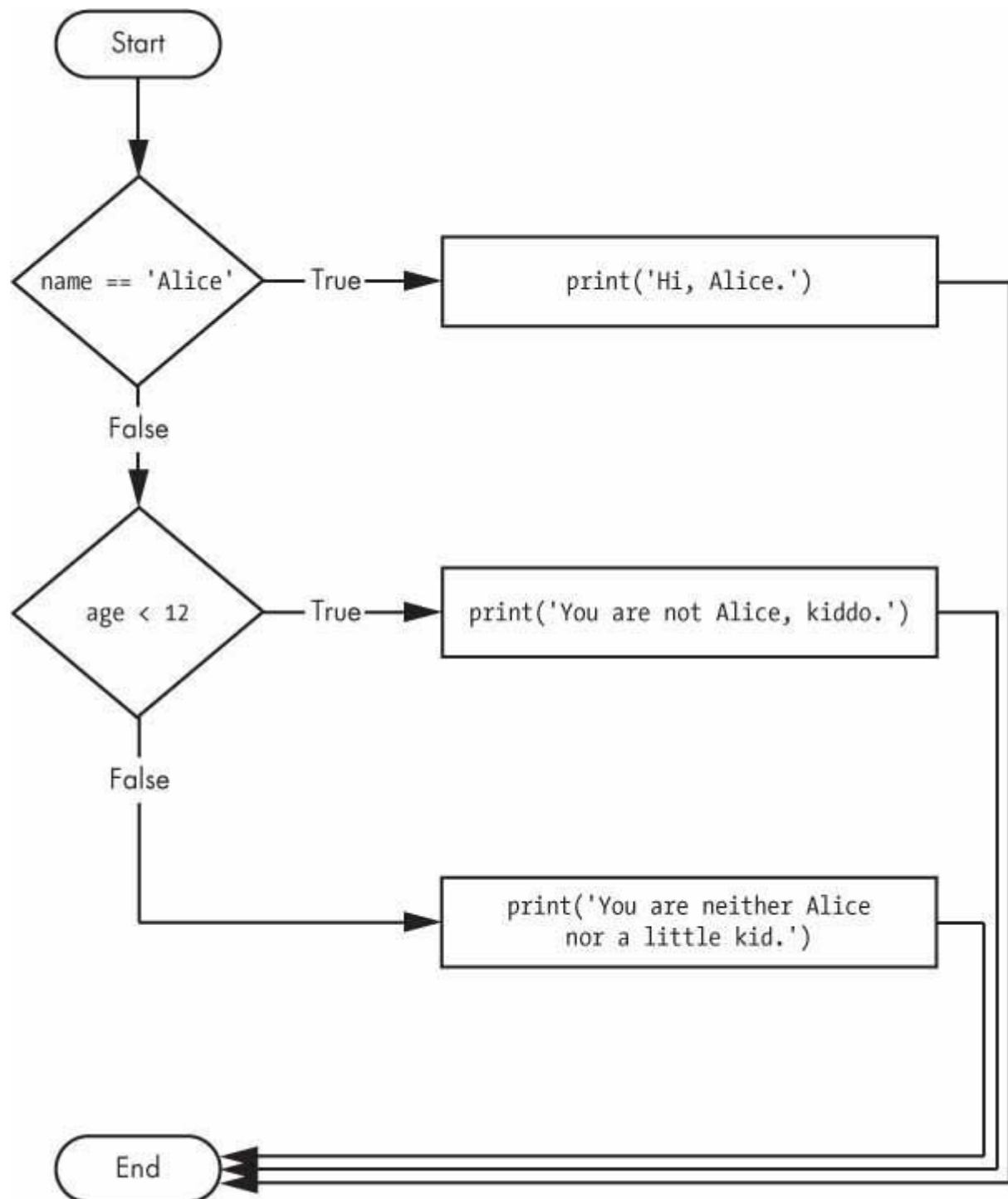


Figure 2-7: Flowchart for the previous littleKid.py program

while Loop Statements

You can make a block of code execute over and over again using a while statement. The code in a while clause will be executed as long as the while statement's condition is True. In code, a while statement always consists of the following:

- The while keyword

- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the while clause)

You can see that a while statement looks similar to an if statement. The difference is in how they behave. At the end of an if clause, the program execution continues after the if statement. But at the end of a while clause, the program execution jumps back to the start of the while statement. The while clause is often called the *while loop* or just the *loop*.

Let's look at an if statement and a while loop that use the same condition and take the same actions based on that condition. Here is the code with an if statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a while statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar—both if and while check the value of spam, and if it's less than 5, they print a message. But when you run these two code snippets, something very different happens for each one. For the if statement, the output is simply "Hello, world.". But for the while statement, it's "Hello, world." repeated five times! Take a look at the flowcharts for these two pieces of code, [Figures 2-8](#) and [2-9](#), to see why this happens.

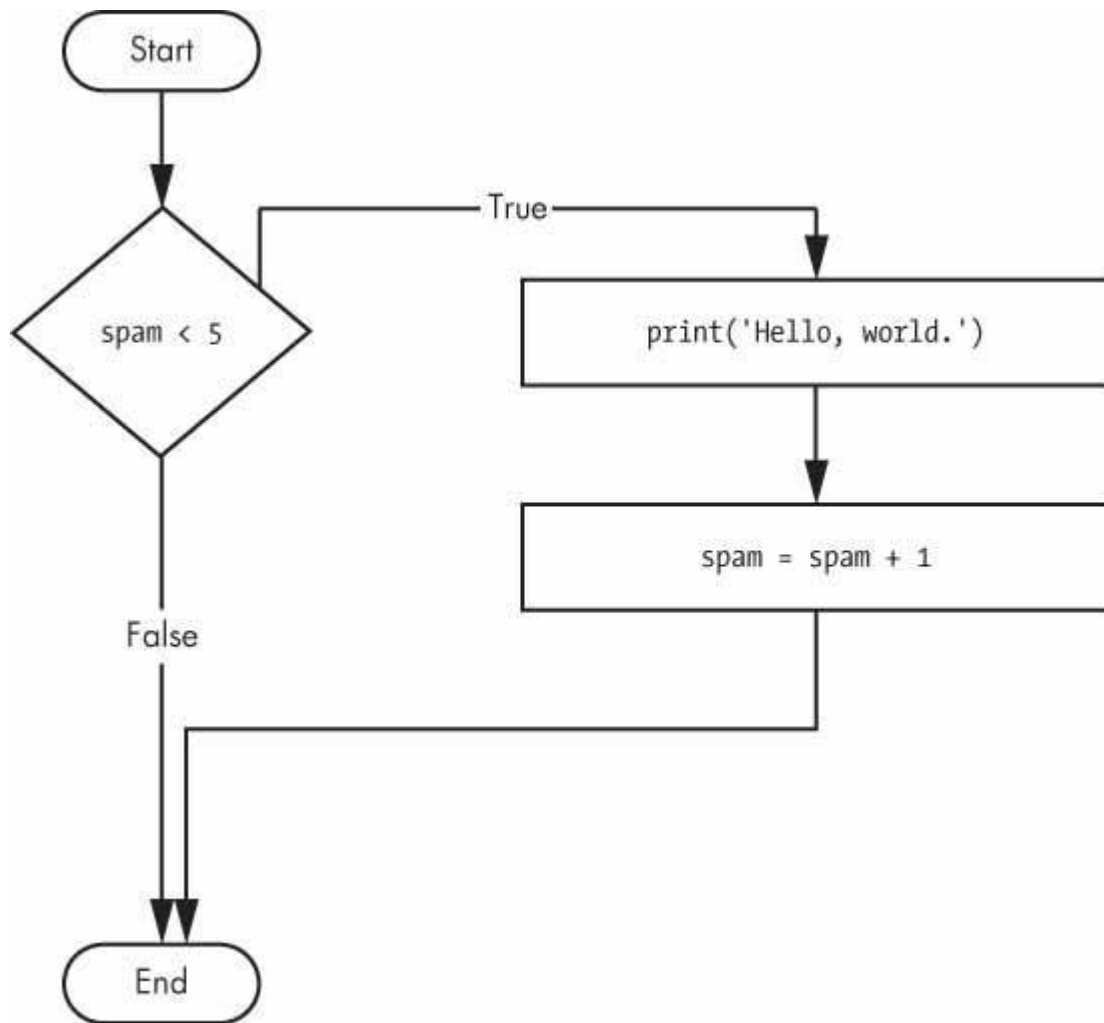


Figure 2-8: The flowchart for the if statement code

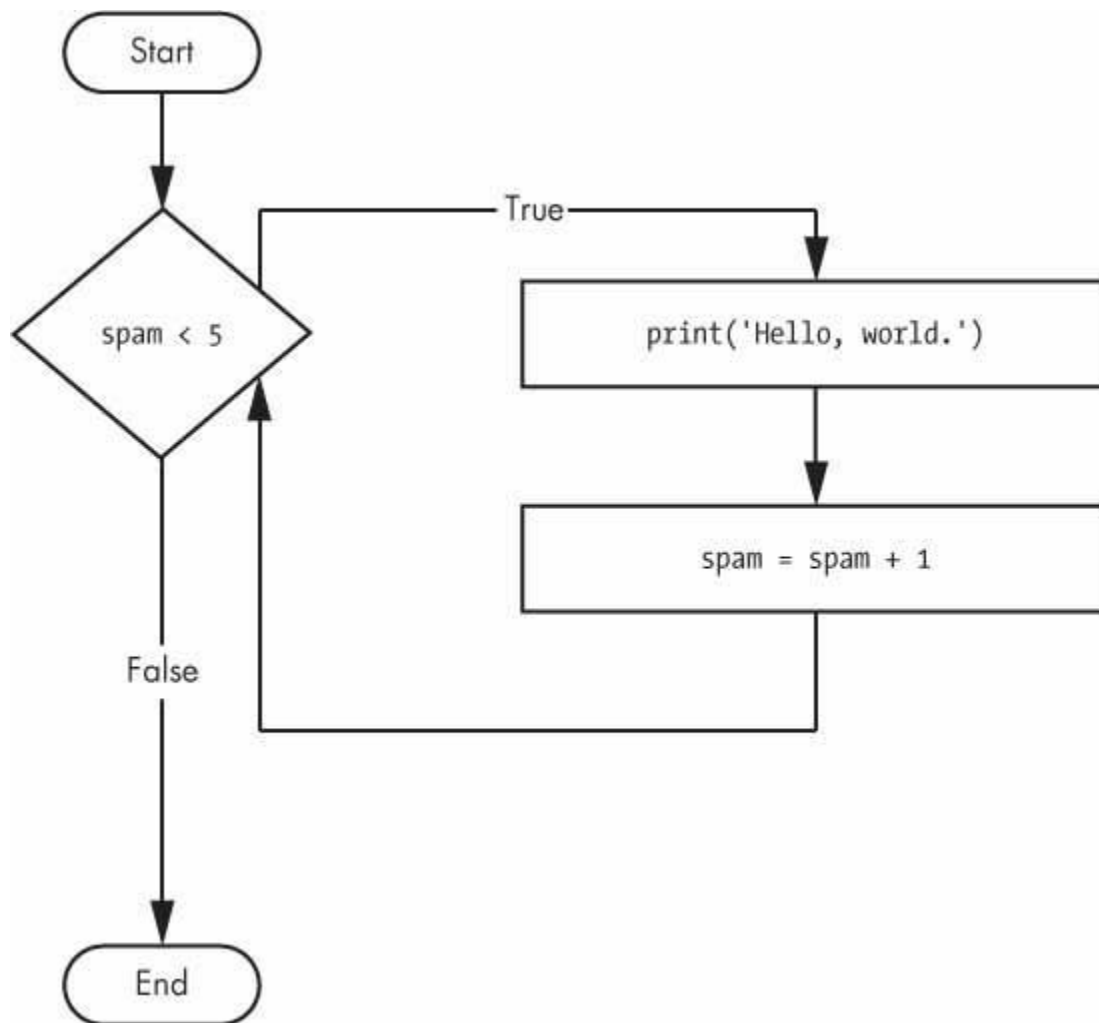


Figure 2-9: The flowchart for the while statement code

The code with the if statement checks the condition, and it prints Hello, world. only once if that condition is true. The code with the while loop, on the other hand, will print it five times. The loop stops after five prints because the integer in spam increases by one at the end of each loop iteration, which means that the loop will execute five times before spam < 5 is False.

In the while loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is True, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be False, the while clause is skipped.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, your name. Select **File ▶ New** to open a new file editor window, enter the following code, and save the file as *yourName.py*:

```
❶ name = ""
❷ while name != 'your name':
    print('Please type your name.')
```



```
❸ name = input()
❹ print('Thank you!')
```

You can view the execution of this program at <https://autbor.com/yourname/>. First, the program sets the name variable ❶ to an empty string. This is so that the name != 'your name' condition will evaluate to True and the program execution will enter the while loop's clause ❷.

The code inside this clause asks the user to type their name, which is assigned to the name variable ❸. Since this is the last line of the block, the execution moves back to the start of the while loop and reevaluates the condition. If the value in name is *not equal* to the string 'your name', then the condition is True, and the execution enters the while clause again.

But once the user types your name, the condition of the while loop will be 'your name' != 'your name', which evaluates to False. The condition is now False, and instead of the program execution reentering the while loop's clause, Python skips past it and continues running the rest of the program ❹. Figure 2-10 shows a flowchart for the *yourName.py* program.

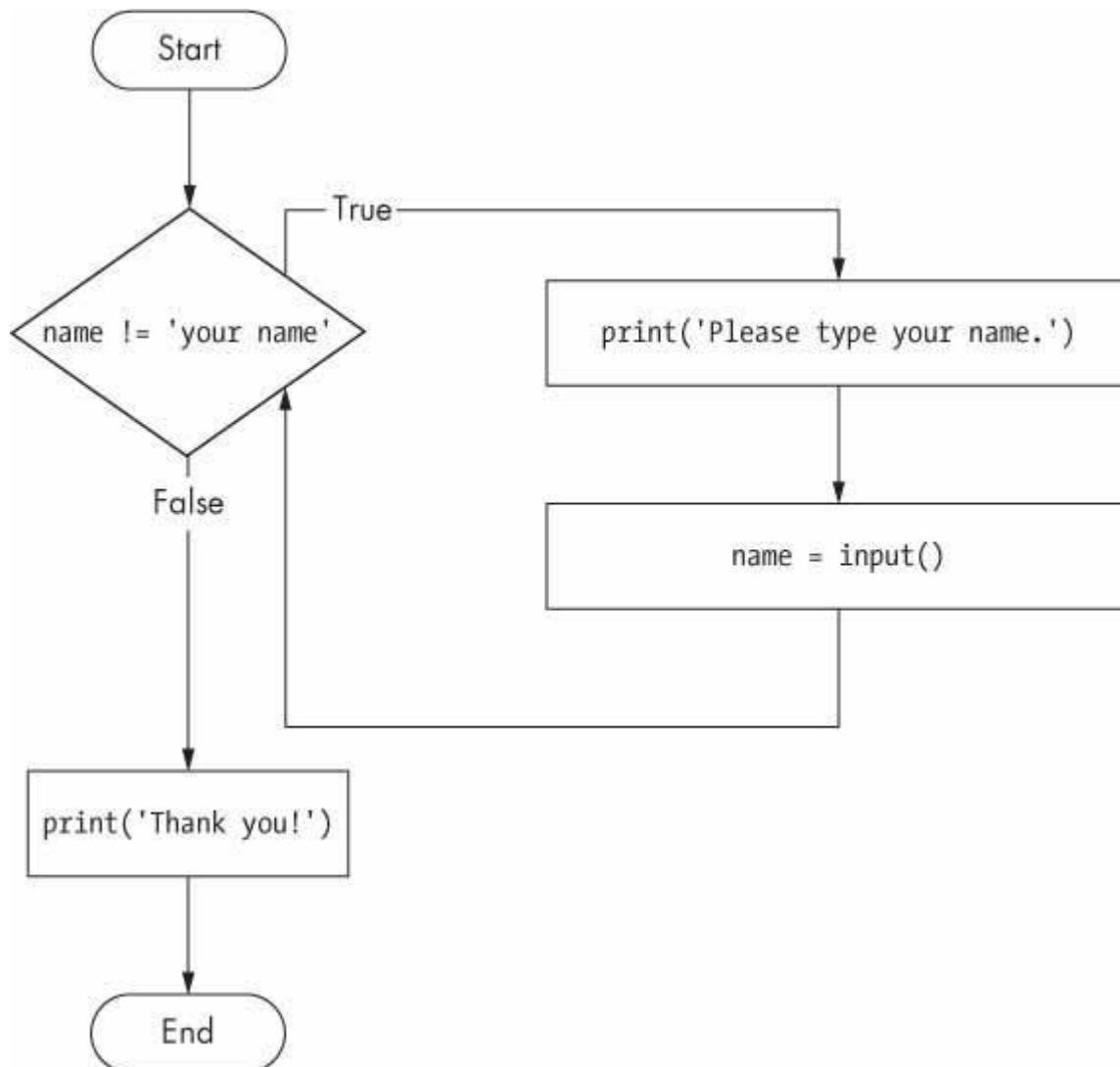


Figure 2-10: A flowchart of the *yourName.py* program

Now, let's see *yourName.py* in action. Press **F5** to run it, and enter something other than your name a few times before you give the program what it wants.

Please type your name.

Al

Please type your name.

Albert

Please type your name.

%#@#%*(^&!!!

Please type your name.

your name

Thank you!

If you never enter your name, then the while loop's condition will never be False, and the program will just keep asking forever. Here, the input() call lets the user enter the right string to make the program move on. In other programs, the condition might never actually change, and that can be a problem. Let's look at how you can break out of a while loop.

break Statements

There is a shortcut to getting the program execution to break out of a while loop's clause early. If the execution reaches a break statement, it immediately exits the while loop's clause. In code, a break statement simply contains the break keyword.

Pretty simple, right? Here's a program that does the same thing as the previous program, but it uses a break statement to escape the loop. Enter the following code, and save the file as *yourName2.py*:

```
❶ while True:
    print('Please type your name.')
    ❷ name = input()
    ❸ if name == 'your name':
        ❹ break
❺ print("Thank you!")
```

You can view the execution of this program at <https://autbor.com/yourname2/>. The first line ❶ creates an *infinite loop*; it is a while loop whose condition is always True. (The expression True, after all, always evaluates down to the value True.) After the program execution enters this loop, it will exit the loop only when a break statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to enter your name ❷. Now, however, while the execution is still inside the while loop, an if statement checks ❸ whether name is equal to 'your name'. If this condition is True, the break statement is run ❹, and the execution moves out of the loop to print("Thank you!") ❺. Otherwise, the if statement's clause that contains the break statement is skipped, which puts the execution at the end of the while loop. At this point, the program execution jumps back to the start of the while statement ❶ to recheck the condition. Since this condition is merely the True Boolean value, the execution enters the loop to ask the user to type your name again. See [Figure 2-11](#) for this program's flowchart.

Run *yourName2.py*, and enter the same text you entered for *yourName.py*. The rewritten program should respond in the same way as the original.

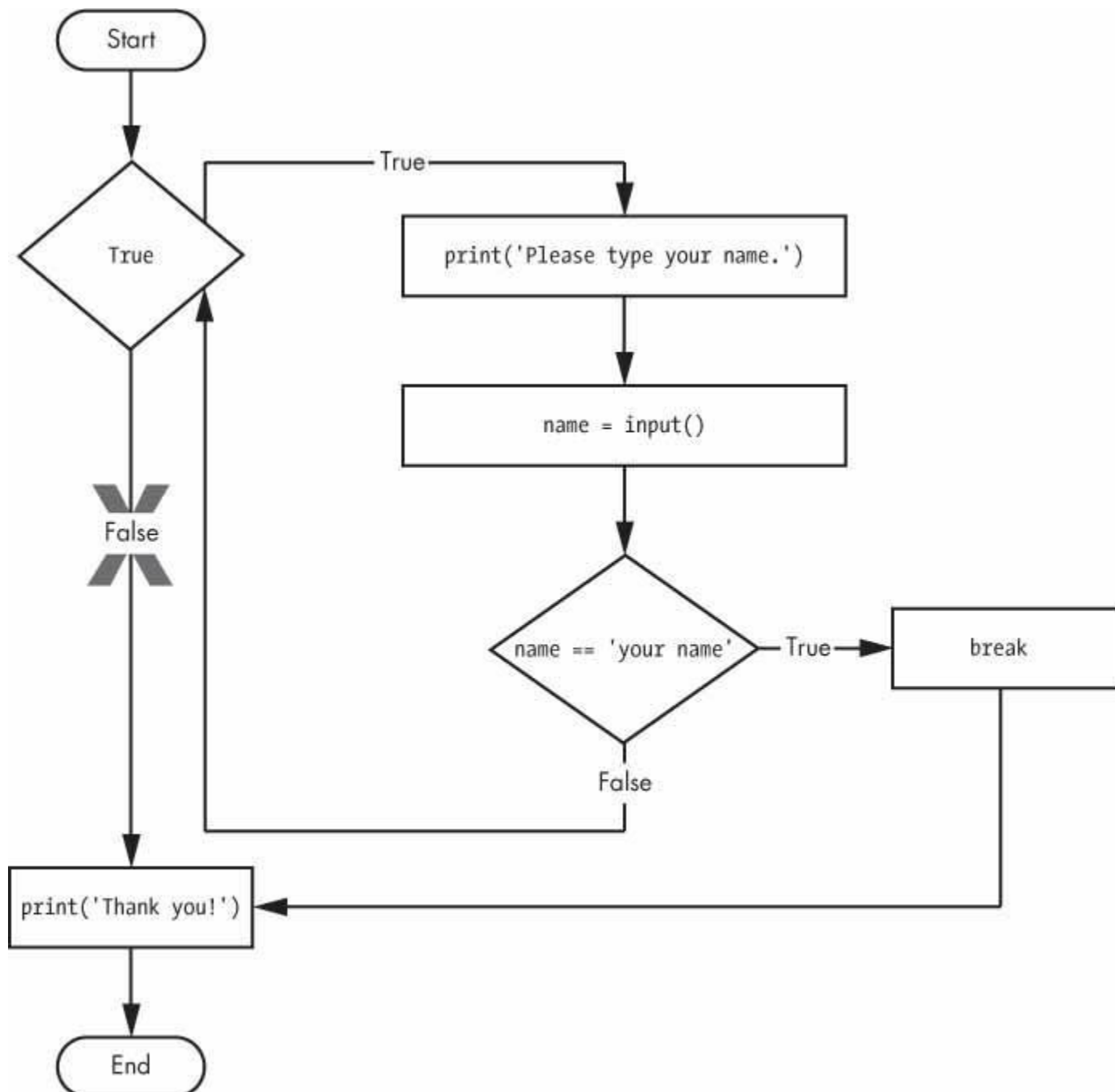
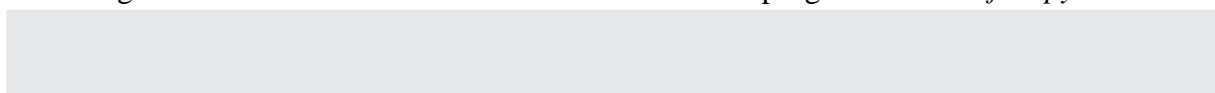


Figure 2-11: The flowchart for the yourName2.py program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always True.

continue Statements

Like break statements, continue statements are used inside loops. When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

Let's use continue to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as *swordfish.py*.



TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C or select **Shell ► Restart Shell** from IDLE's menu. This will send a KeyboardInterrupt error to your program and cause it to stop immediately. Try stopping a program by creating a simple infinite loop in the file editor, and save the program as *infiniteLoop.py*.

```
while True:
    print('Hello, world!')
```

When you run this program, it will print Hello, world! to the screen forever because the while statement's condition is always True. CTRL-C is also handy if you want to simply terminate your program immediately, even if it's not stuck in an infinite loop.

```
while True:
    print('Who are you?')
    name = input()
    ❶ if name != 'Joe':
        ❷ continue
    print('Hello, Joe. What is the password? (It is a fish.)')
    ❸ password = input()
    if password == 'swordfish':
        ❹ break
    ❺ print('Access granted.')
```

If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop. When the program reevaluates the condition, the execution will always enter the loop, since the condition is simply the value True. Once the user makes it past that if statement, they are asked for a password ❸. If the password entered is swordfish, then the break statement ❹ is run, and the execution jumps out of the while loop to print Access granted ❺. Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop. See [Figure 2-12](#) for this program's flowchart.

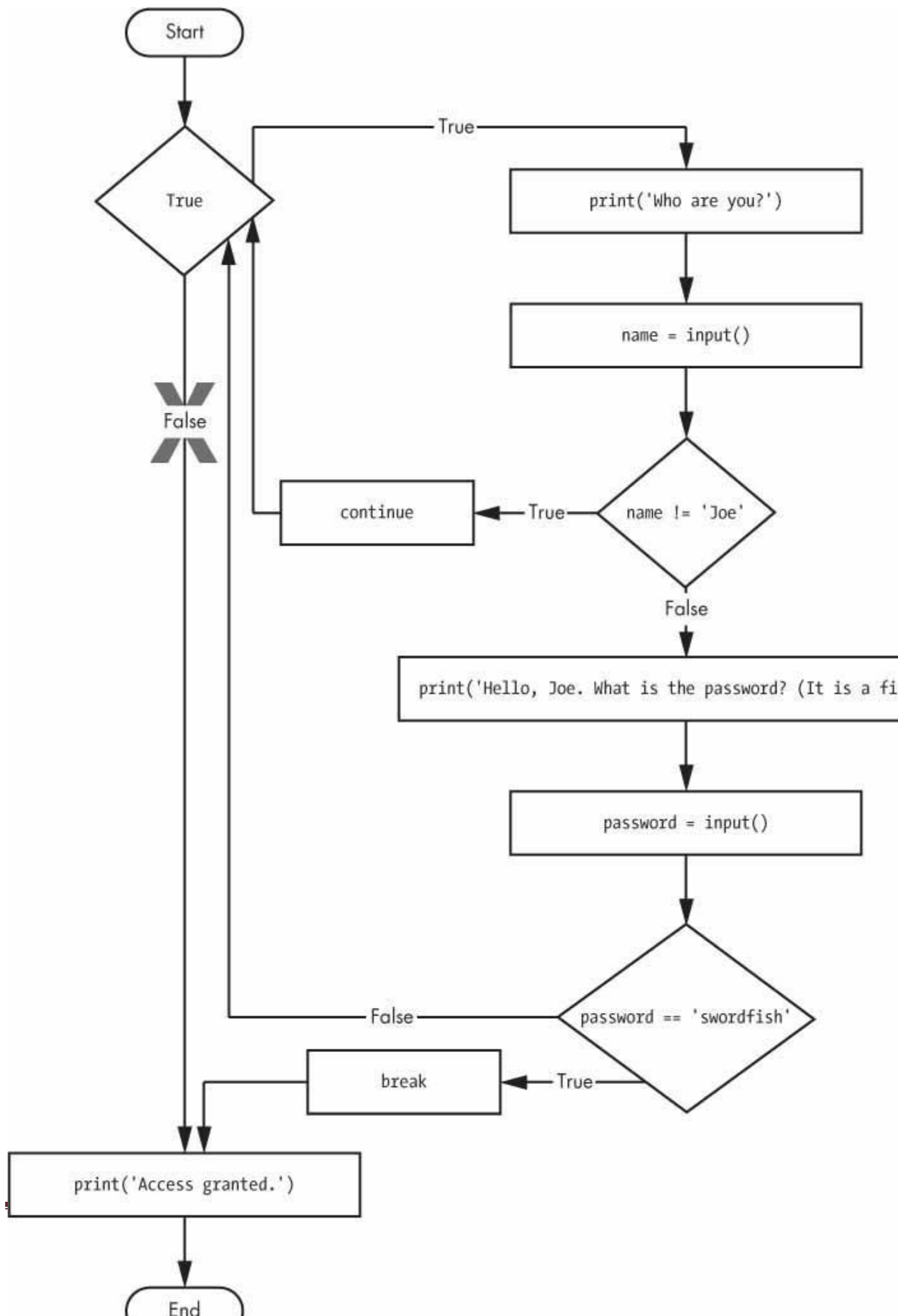


Figure 2-12: A flowchart for swordfish.py. The X path will logically never happen, because the loop condition is always True.

“TRUTHY” AND “FALSEY” VALUES

Conditions will consider some values in other data types equivalent to True and False. When used in conditions, 0, 0.0, and "" (the empty string) are considered False, while all other values are considered True. For example, look at the following program:

```

name = ""
❶ while not name:
    print('Enter your name: ')
    name = input()
print('How many guests will you have?')
numOfGuests = int(input())
❷ if numOfGuests:
    ❸ print('Be sure to have enough room for all your guests.')
print('Done')

```

You can view the execution of this program at <https://autbor.com/howmanyguests/>. If the user enters a blank string for name, then the while statement's condition will be True ❶, and the program continues to ask for a name. If the value for numOfGuests is not 0 ❷, then the condition is considered to be True, and the program will print a reminder for the user ❸.

You could have entered not name != "" instead of not name, and numOfGuests != 0 instead of numOfGuests, but using the truthy and falsey values can make your code easier to read.

Run this program and give it some input. Until you claim to be Joe, the program shouldn't ask for a password, and once you enter the correct password, it should exit.

```

Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.

```

You can view the execution of this program at <https://autbor.com/hellojoe/>.

for Loops and the range() Function

The while loop keeps looping while its condition is True (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a for loop statement and the range() function.

In code, a for statement looks something like for i in range(5): and includes the following:

- The for keyword

- A variable name
- The in keyword
- A call to the range() method with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the for clause)

Let's create a new program called *fiveTimes.py* to help you see a for loop in action.

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

You can view the execution of this program at <https://autbor.com/fivetimesfor/>. The code in the for loop's clause is run five times. The first time it is run, the variable *i* is set to 0. The print() call in the clause will print Jimmy Five Times (0). After Python finishes an iteration through all the code inside the for loop's clause, the execution goes back to the top of the loop, and the for statement increments *i* by one. This is why range(5) results in five iterations through the clause, with *i* being set to 0, then 1, then 2, then 3, and then 4. The variable *i* will go up to, but will not include, the integer passed to range(). [Figure 2-13](#) shows a flowchart for the *fiveTimes.py* program.

When you run this program, it should print Jimmy Five Times followed by the value of *i* five times before leaving the for loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

NOTE

You can use break and continue statements inside for loops as well. The continue statement will continue to the next value of the for loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use continue and break statements only inside while and for loops. If you try to use these statements elsewhere, Python will give you an error.

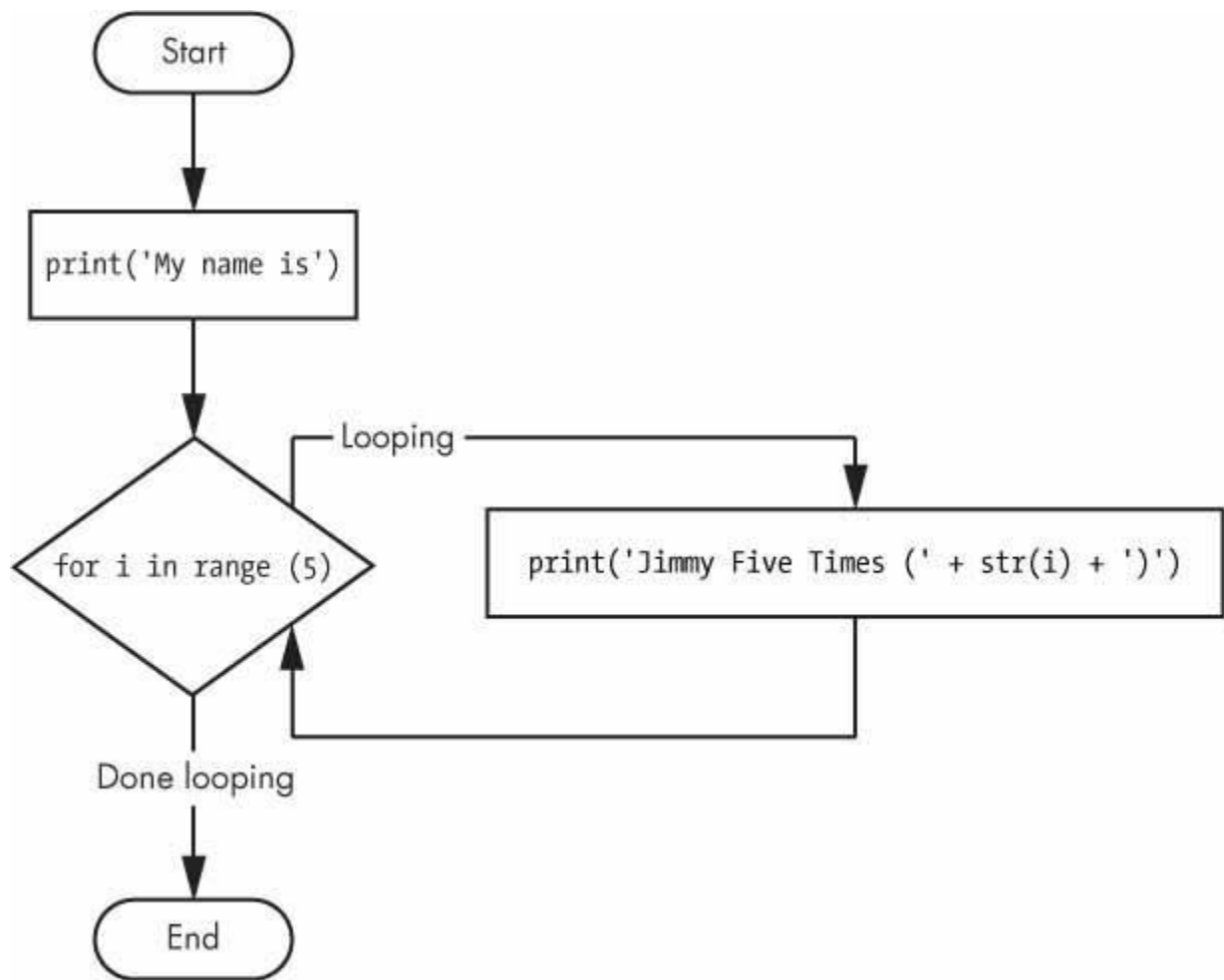


Figure 2-13: The flowchart for fiveTimes.py

As another for loop example, consider this story about the mathematician Carl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a for loop to do this calculation for you.

-
- ❶ `total = 0`
 - ❷ `for num in range(101):`
 - ❸ `total = total + num`
 - ❹ `print(total)`
-

The result should be 5,050. When the program first starts, the total variable is set to 0 ❶. The for loop ❷ then executes `total = total + num` ❸ 100 times. By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to total. At this point, total is printed to the screen ❹. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out a way to solve the problem in seconds. There are 50 pairs of numbers that add up to 101: 1 + 100, 2 + 99, 3 + 98, and so on, until 50 + 51. Since 50×101 is 5,050, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

An Equivalent while Loop

You can actually use a while loop to do the same thing as a for loop; for loops are just more concise. Let's rewrite *fiveTimes.py* to use a while loop equivalent of a for loop.

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

You can view the execution of this program at <https://autbor.com/fivetimeswhile/>. If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a for loop.

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12, 16):
    print(i)
```

The first argument will be where the for loop's variable starts, and the second argument will be up to, but not including, the number to stop at.

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

So calling `range(0, 10, 2)` will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

The range() function is flexible in the sequence of numbers it produces for for loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the for loop count down instead of up.

```
for i in range(5, -1, -1):  
    print(i)
```

This for loop would have the following output:

```
5  
4  
3  
2  
1  
0
```

Running a for loop to print i with range(5, -1, -1) should print from five down to zero.

IMPORTING MODULES

All Python programs can call a basic set of functions called *built-in functions*, including the print(), input(), and len() functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the math module has mathematics-related functions, the random module has random number-related functions, and so on.

Before you can use the functions in a module, you must import the module with an import statement. In code, an import statement consists of the following:

- The import keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the random module, which will give us access to the random.randint() function.

Enter this code into the file editor, and save it as *printRandom.py*:

```
import random  
for i in range(5):  
    print(random.randint(1, 10))
```

DON'T OVERWRITE MODULE NAMES

When you save your Python scripts, take care not to give them a name that is used by one of Python's modules, such as *random.py*, *sys.py*, *os.py*, or *math.py*. If you accidentally name one of your programs, say, *random.py*, and use an import random statement in another program, your program would import your *random.py* file instead of Python's random module. This can lead to errors such as AttributeError: module 'random' has no attribute 'randint', since your *random.py* doesn't have the functions that the real random module has. Don't use the names of any built-in Python functions either, such as print() or input().

Problems like these are uncommon, but can be tricky to solve. As you gain more programming experience, you'll become more aware of the standard names used by Python's modules and functions, and will run into these problems less frequently.

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

You can view the execution of this program at <https://autbor.com/printrandom/>. The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Since `randint()` is in the `random` module, you must first type **random.** in front of the function name to tell Python to look for this function inside the `random` module.

Here's an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. We'll learn more about them later in the book.

from import Statements

An alternative form of the import statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star; for example, `from random import *`.

With this form of import statement, calls to functions in `random` will not need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the `import random` form of the statement.

ENDING A PROGRAM EARLY WITH THE `SYS.EXIT()` FUNCTION

The last flow control concept to cover is how to terminate the program. Programs always terminate if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or `exit`, before the last instruction by calling the `sys.exit()` function. Since this function is in the `sys` module, you have to import `sys` before your program can use it.

Open a file editor window and enter the following code, saving it as *exitExample.py*:

```
import sys

while True:
    print("Type exit to exit.")
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in IDLE. This program has an infinite loop with no break statement inside. The only way this program will end is if the execution reaches the `sys.exit()` call. When response is equal to exit, the line containing the `sys.exit()` call is executed. Since the response variable is set by the `input()` function, the user must enter exit in order to stop the program.

A SHORT PROGRAM: GUESS THE NUMBER

The examples I've shown you so far are useful for introducing basic concepts, but now let's see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple "guess the number" game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

Enter the following source code into the file editor, and save the file as *guessTheNumber.py*:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + '')
```

```
guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

You can view the execution of this program at <https://autbor.com/guessthenumber/>. Let's look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the random module so that it can use the random.randint() function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable secretNumber.

```
print('I am thinking of a number between 1 and 20.')
```

```
# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code that lets the player enter a guess and checks that guess is in a for loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Since input() returns a string, its return value is passed straight into int(), which translates the string into an integer value. This gets stored in a variable named guess.

```
if guess < secretNumber:
    print('Your guess is too low.')
elif guess > secretNumber:
    print('Your guess is too high.')
```

These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
else:
    break # This condition is the correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number—in which case, you want the program execution to break out of the for loop.

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
```

else:

```
print('Nope. The number I was thinking of was ' + str(secretNumber))
```

After the for loop, the previous if...else statement checks whether the player has correctly guessed the number and then prints an appropriate message to the screen. In both cases, the program displays a variable that contains an integer value (guessesTaken and secretNumber). Since it must concatenate these integer values to strings, it passes these variables to the str() function, which returns the string value form of these integers. Now these strings can be concatenated with the + operators before finally being passed to the print() function call.

A SHORT PROGRAM: ROCK, PAPER, SCISSORS

Let's use the programming concepts we've learned so far to create a simple rock, paper, scissors game. The output will look like this:

ROCK, PAPER, SCISSORS

0 Wins, 0 Losses, 0 Ties

Enter your move: (r)ock (p)aper (s)cissors or (q)uit

p

PAPER versus...

PAPER

It is a tie!

0 Wins, 1 Losses, 1 Ties

Enter your move: (r)ock (p)aper (s)cissors or (q)uit

s

SCISSORS versus...

PAPER

You win!

1 Wins, 1 Losses, 1 Ties

Enter your move: (r)ock (p)aper (s)cissors or (q)uit

q

Type the following source code into the file editor, and save the file as *rpsGame.py*:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0

while True: # The main game loop.
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True: # The player input loop.
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        playerMove = input()
        if playerMove == 'q':
```

```

    sys.exit()
    #
    if playerMove == 'r' or playerMove == 'p' or playerMove == 's':
        break # Break out of the loop.
    print('Type one of r, p, s, or q.')

# Display what the player chose:
if playerMove == 'r':
    print('ROCK versus...')
elif playerMove == 'p':
    print('PAPER versus...')
elif playerMove == 's':
    print('SCISSORS versus...')

# Display what the computer chose:
randomNumber = random.randint(1, 3)
if randomNumber == 1:
    computerMove = 'r'
    print('ROCK')
elif randomNumber == 2:
    computerMove = 'p'
    print('PAPER')
elif randomNumber == 3:
    computerMove = 's'
    print('SCISSORS')

# Display and record the win/loss/tie:
if playerMove == computerMove:
    print('It is a tie!')
    ties = 1
elif playerMove == 'r' and computerMove == 's':
    print('You win!')
    wins = 1
elif playerMove == 'p' and computerMove == 'r':
    print('You win!')
    wins = 1
elif playerMove == 's' and computerMove == 'p':
    print('You win!')
    wins = 1
elif playerMove == 'r' and computerMove == 'p':
    print('You lose!')
    losses = 1
elif playerMove == 'p' and computerMove == 's':
    print('You lose!')
    losses = 1
elif playerMove == 's' and computerMove == 'r':
    print('You lose!')
    losses = 1
    losses = losses + 1

```

Let's look at this code line by line, starting at the top.

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0
```

First, we import the random and sys module so that our program can call the random.randint() and sys.exit() functions. We also set up three variables to keep track of how many wins, losses, and ties the player has had.

```
while True: # The main game loop.
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True: # The player input loop.
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        playerMove = input()
        if playerMove == 'q':
            sys.exit() # Quit the program.
        if playerMove == 'r' or playerMove == 'p' or playerMove == 's':
            break # Break out of the player input loop.
        print('Type one of r, p, s, or q.')
```

This program uses a while loop inside of another while loop. The first loop is the main game loop, and a single game of rock, paper, scissors is played on each iteration through this loop. The second loop asks for input from the player, and keeps looping until the player has entered an r, p, s, or q for their move. The r, p, and s correspond to rock, paper, and scissors, respectively, while the q means the player intends to quit. In that case, sys.exit() is called and the program exits. If the player has entered r, p, or s, the execution breaks out of the loop. Otherwise, the program reminds the player to enter r, p, s, or q and goes back to the start of the loop.

```
# Display what the player chose:
if playerMove == 'r':
    print('ROCK versus...')
elif playerMove == 'p':
    print('PAPER versus...')
elif playerMove == 's':
    print('SCISSORS versus...')
```

The player's move is displayed on the screen.

```
# Display what the computer chose:
randomNumber = random.randint(1, 3)
if randomNumber == 1:
    computerMove = 'r'
```

```
print('ROCK')
elif randomNumber == 2:
    computerMove = 'p'
    print('PAPER')
elif randomNumber == 3:
    computerMove = 's'
    print('SCISSORS')
```

Next, the computer's move is randomly selected. Since `random.randint()` can only return a random number, the 1, 2, or 3 integer value it returns is stored in a variable named `randomNumber`. The program stores a 'r', 'p', or 's' string in `computerMove` based on the integer in `randomNumber`, as well as displays the computer's move.

```
# Display and record the win/loss/tie:
if playerMove == computerMove:
    print('It is a tie!')
    ties = ties + 1
elif playerMove == 'r' and computerMove == 's':
    print('You win!')
    wins = wins + 1
elif playerMove == 'p' and computerMove == 'r':
    print('You win!')
    wins = wins + 1
elif playerMove == 's' and computerMove == 'p':
    print('You win!')
    wins = wins + 1
elif playerMove == 'r' and computerMove == 'p':
    print('You lose!')
    losses = losses + 1
elif playerMove == 'p' and computerMove == 's':
    print('You lose!')
    losses = losses + 1
elif playerMove == 's' and computerMove == 'r':
    print('You lose!')
    losses = losses + 1
```

Finally, the program compares the strings in `playerMove` and `computerMove`, and displays the results on the screen. It also increments the wins, losses, or ties variable appropriately. Once the execution reaches the end, it jumps back to the start of the main program loop to begin another game.

SUMMARY

By using expressions that evaluate to True or False (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to True. The `break` and `continue` statements are useful if you need to exit a loop or jump back to the loop's start.

These flow control statements will let you write more intelligent programs. You can also use another type of flow control by writing your own functions, which is the topic of the next chapter.

PRACTICE QUESTIONS

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Write code that prints Hello if 1 is stored in spam, prints Howdy if 2 is stored in spam, and prints Greetings! if anything else is stored in spam.
10. What keys can you press if your program is stuck in an infinite loop?
11. What is the difference between break and continue?
12. What is the difference between range(10), range(0, 10), and range(0, 10, 1) in a for loop?
13. Write a short program that prints the numbers 1 to 10 using a for loop. Then write an equivalent program that prints the numbers 1 to 10 using a while loop.
14. If you had a function named bacon() inside a module named spam, how would you call it after importing spam?

Extra credit: Look up the `round()` and `abs()` functions on the internet, and find out what they do. Experiment with them in the interactive shell.

FUNCTIONS

You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a miniprogram within a program.

To better understand how functions work, let's create one. Enter this program into the file editor and save it as *helloFunc.py*:

```
❶ def hello():  
    ❷ print('Howdy!')  
    print('Howdy!!!')  
    print('Hello there.')  
  
❸ hello()  
hello()  
hello()
```

You can view the execution of this program at <https://autbor.com/hellofunc/>. The first line is a `def` statement ❶, which defines a function named `hello()`. The code in the block that follows the `def` statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `hello()` lines after the function ❸ are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Howdy!  
Howdy!!!  
Hello there.  
Howdy!  
Howdy!!!  
Hello there.  
Howdy!  
Howdy!!!  
Hello there.
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time, and the program would look like this:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

In general, you always want to avoid duplicating code because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

As you get more programming experience, you'll often find yourself *deduplicating* code, which means getting rid of duplicated or copy-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

DEF STATEMENTS WITH PARAMETERS

When you call the `print()` or `len()` function, you pass them values, called *arguments*, by typing them between the parentheses. You can also define your own functions that accept arguments. Type this example into the file editor and save it as *helloFunc2.py*:

```
❶ def hello(name):
    ❷ print('Hello, ' + name)

❸ hello('Alice')
   hello('Bob')
```

When you run this program, the output looks like this:

```
Hello, Alice
Hello, Bob
```

You can view the execution of this program at <https://autbor.com/hellofunc2/>. The definition of the `hello()` function in this program has a parameter called `name` ❶. *Parameters* are variables that contain arguments. When a function is called with arguments, the arguments are stored in the parameters. The first time the `hello()` function is called, it is passed the argument 'Alice' ❸. The program execution enters the function, and the parameter name is automatically set to 'Alice', which is what gets printed by the `print()` statement ❷.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you

a `NameError` because there is no variable named `name`. This variable is destroyed after the function call `hello('Bob')` returns, so `print(name)` would refer to a `name` variable that does not exist.

This is similar to how a program's variables are forgotten when the program terminates. I'll talk more about why that happens later in the chapter, when I discuss what a function's local scope is.

Define, Call, Pass, Argument, Parameter

The terms *define*, *call*, *pass*, *argument*, and *parameter* can be confusing. Let's look at a code example to review these terms:

```
❶ def sayHello(name):  
    print('Hello, ' + name)  
❷ sayHello('Al')
```

To *define* a function is to create it, just like an assignment statement like `spam = 42` creates the `spam` variable. The `def` statement defines the `sayHello()` function ❶. The `sayHello('Al')` line ❷ *calls* the now-created function, sending the execution to the top of the function's code. This function call is also known as *passing* the string value `'Al'` to the function. A value being passed to a function in a function call is an *argument*. The argument `'Al'` is assigned to a local variable named `name`. Variables that have arguments assigned to them are *parameters*.

It's easy to mix up these terms, but keeping them straight will ensure that you know precisely what the text in this chapter means.

RETURN VALUES AND RETURN STATEMENTS

When you call the `len()` function and pass it an argument such as `'Hello'`, the function call evaluates to the integer value 5, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the *return value* of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The return keyword
- The value or expression that the function should return

When an expression is used with a return statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what number it is passed as an argument. Enter this code into the file editor and save it as *magic8Ball.py*:

```
❶ import random  
  
❷ def getAnswer(answerNumber):  
    ❸ if answerNumber == 1:  
        return 'It is certain'  
    elif answerNumber == 2:  
        return 'It is decidedly so'  
    elif answerNumber == 3:
```

```
    return 'Yes'
elif answerNumber == 4:
    return 'Reply hazy try again'
elif answerNumber == 5:
    return 'Ask again later'
elif answerNumber == 6:
    return 'Concentrate and ask again'
elif answerNumber == 7:
    return 'My reply is no'
elif answerNumber == 8:
    return 'Outlook not so good'
elif answerNumber == 9:
    return 'Very doubtful'
```

④ `r = random.randint(1, 9)`

⑤ `fortune = getAnswer(r)`

⑥ `print(fortune)`

You can view the execution of this program at <https://autbor.com/magic8ball/>. When this program starts, Python first imports the random module ①. Then the `getAnswer()` function is defined ②. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments: 1 and 9 ④. It evaluates to a random integer between 1 and 9 (including 1 and 9 themselves), and this value is stored in a variable named `r`.

The `getAnswer()` function is called with `r` as the argument ⑤. The program execution moves to the top of the `getAnswer()` function ③, and the value `r` is stored in a parameter named `answerNumber`. Then, depending on the value in `answerNumber`, the function returns one of many possible string values. The program execution returns to the line at the bottom of the program that originally called `getAnswer()` ⑤. The returned string is assigned to a variable named `fortune`, which then gets passed to a `print()` call ⑥ and is printed to the screen.

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

to this single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

Remember, expressions are composed of values and operators. A function call can be used in an expression because the call evaluates to its return value.

THE NONE VALUE

In Python, there is a value called None, which represents the absence of a value. The None value is the only value of the NoneType data type. (Other programming languages might call this value null, nil, or undefined.) Just like the Boolean True and False values, None must be typed with a capital N.

This value-without-a-value can be helpful when you need to store something that won't be confused for a real value in a variable. One place where None is used is as the return value of print(). The print() function displays text on the screen, but it doesn't need to return anything in the same way len() or input() does. But since all function calls need to evaluate to a return value, print() returns None. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds return None to the end of any function definition with no return statement. This is similar to how a while or for loop implicitly ends with a continue statement. Also, if you use a return statement without a value (that is, just the return keyword by itself), then None is returned.

KEYWORD ARGUMENTS AND THE PRINT() FUNCTION

Most arguments are identified by their position in the function call. For example, random.randint(1, 10) is different from random.randint(10, 1). The function call random.randint(1, 10) will return a random integer between 1 and 10 because the first argument is the low end of the range and the second argument is the high end (while random.randint(10, 1) causes an error).

However, rather than through their position, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for *optional parameters*. For example, the print() function has the optional parameters end and sep to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

If you ran a program with the following code:

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two outputted strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed. However, you can set the end keyword argument to change the newline character to a different string. For example, if the code were this:

```
print('Hello', end='')  
print('World')
```

the output would look like this:

HelloWorld

The output is printed on a single line because there is no longer a newline printed after 'Hello'. Instead, the blank string is printed. This is useful if you need to disable the newline that gets added to the end of every `print()` function call.

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')  
cats dogs mice
```

But you could replace the default separating string by passing the `sep` keyword argument a different string. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')  
cats,dogs,mice
```

You can add keyword arguments to the functions you write as well, but first you'll have to learn about the list and dictionary data types in the next two chapters. For now, just know that some functions have optional keyword arguments that can be specified when the function is called.

THE CALL STACK

Imagine that you have a meandering conversation with someone. You talk about your friend Alice, which then reminds you of a story about your coworker Bob, but first you have to explain something about your cousin Carol. You finish your story about Carol and go back to talking about Bob, and when you finish your story about Bob, you go back to talking about Alice. But then you are reminded about your brother David, so you tell a story about him, and then get back to finishing your original story about Alice. Your conversation followed a *stack*-like structure, like in [Figure 3-1](#). The conversation is stack-like because the current topic is always at the top of the stack.

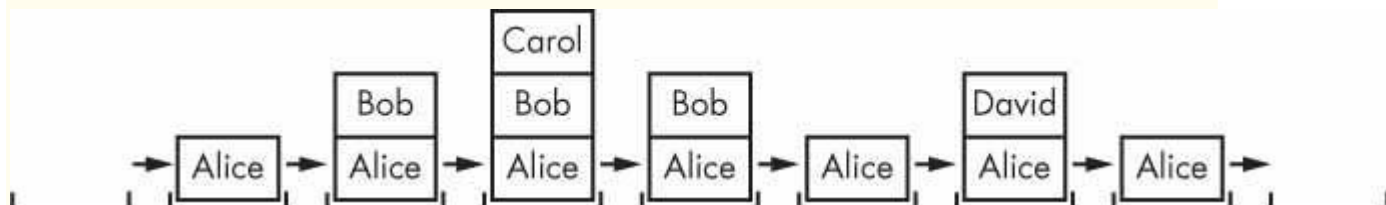


Figure 3-1: Your meandering conversation stack

Similar to our meandering conversation, calling a function doesn't send the execution on a one-way trip to the top of a function. Python will remember which line of code called the function so that the execution can return there when it encounters a `return` statement. If that

original function called other functions, the execution would return to *those* function calls first, before returning from the original function call.

Open a file editor window and enter the following code, saving it as *abcdCallStack.py*:

```
def a():  
    print('a() starts')  
    ❶ b()  
    ❷ d()  
    print('a() returns')  
  
def b():  
    print('b() starts')  
    ❸ c()  
    print('b() returns')  
  
def c():  
    ❹ print('c() starts')  
    print('c() returns')  
  
def d():  
    print('d() starts')  
    print('d() returns')  
  
❺ a()
```

If you run this program, the output will look like this:

```
a() starts  
b() starts  
c() starts  
c() returns  
b() returns  
d() starts  
d() returns  
a() returns
```

You can view the execution of this program at <https://autbor.com/abcdcallstack/>. When a() is called ❺, it calls b() ❶, which in turn calls c() ❸. The c() function doesn't call anything; it just displays c() starts ❹ and c() returns before returning to the line in b() that called it ❸. Once execution returns to the code in b() that called c(), it returns to the line in a() that called b() ❶. The execution continues to the next line in the b() function ❷, which is a call to d(). Like the c() function, the d() function also doesn't call anything. It just displays d() starts and d() returns before returning to the line in b() that called it. Since b() contains no other code, the execution returns to the line in a() that called b() ❷. The last line in a() displays a() returns before returning to the original a() call at the end of the program ❺.

The *call stack* is how Python remembers where to return the execution after each function call. The call stack isn't stored in a variable in your program; rather, Python handles it behind the scenes. When your program calls a function, Python creates a *frame object* on the top of the call stack. Frame objects store the line number of the original function call so that Python can remember where to return. If another function call is made, Python puts another frame object on the call stack above the other one.

When a function call returns, Python removes a frame object from the top of the stack and moves the execution to the line number stored in it. Note that frame objects are always added and removed from the top of the stack and not from any other place. [Figure 3-2](#) illustrates the state of the call stack in *abcdCallStack.py* as each function is called and returns.

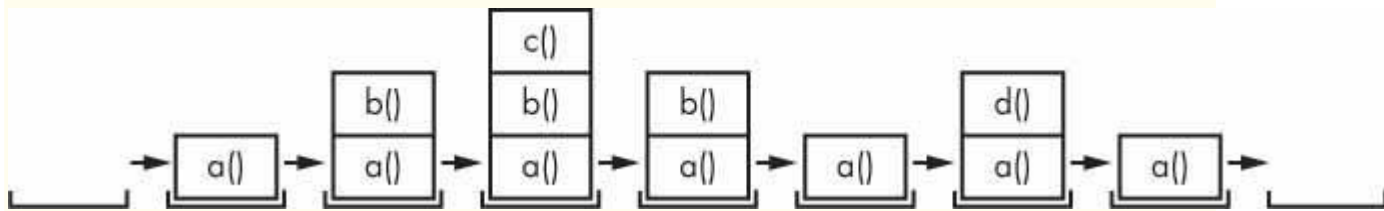


Figure 3-2: The frame objects of the call stack as *abcdCallStack.py* calls and returns from functions

The top of the call stack is which function the execution is currently in. When the call stack is empty, the execution is on a line outside of all functions.

The call stack is a technical detail that you don't strictly need to know about to write programs. It's enough to understand that function calls return to the line number they were called from. However, understanding call stacks makes it easier to understand local and global scopes, described in the next section.

LOCAL AND GLOBAL SCOPE

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call the function, the local variables will not remember the values stored in them from the last time the function was called. Local variables are also stored in frame objects on the call stack.

Scopes matter for several reasons:

- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named *spam* and a global variable also named *spam*.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the number of lines of code that may be causing a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program, and your program could be hundreds or thousands of lines long! But if the bug is caused by a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():  
    ❶ eggs = 31337  
spam()  
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):  
  File "C:/test1.py", line 4, in <module>  
    print(eggs)  
NameError: name 'eggs' is not defined
```

The error happens because the eggs variable exists only in the local scope created when spam() is called ❶. Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs. So when your program tries to run print(eggs), Python gives you an error saying that eggs is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def spam():  
    ❶ eggs = 99  
    ❷ bacon()  
    ❸ print(eggs)  
  
def bacon():  
    ham = 101  
    ❹ eggs = 0
```

5 spam()

You can view the execution of this program at <https://autbor.com/otherlocalscopes/>. When the program starts, the spam() function is called **5**, and a local scope is created. The local variable eggs **1** is set to 99. Then the bacon() function is called **2**, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is different from the one in spam()’s local scope—is also created **4** and set to 0.

When bacon() returns, the local scope for that call is destroyed, including its eggs variable. The program execution continues in the spam() function to print the value of eggs **3**. Since the local scope for the call to spam() still exists, the only eggs variable is the spam() function’s eggs variable, which was set to 99. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():  
    print(eggs)  
eggs = 42  
spam()  
print(eggs)
```

You can view the execution of this program at <https://autbor.com/readglobal/>. Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

Local and Global Variables with the Same Name

Technically, it’s perfectly acceptable to use the same variable name for a global variable and local variables in different scopes in Python. But, to simplify your life, avoid doing this. To see what happens, enter the following code into the file editor and save it as *localGlobalSameName.py*:

```
def spam():  
    1 eggs = 'spam local'  
    print(eggs) # prints 'spam local'  
  
def bacon():  
    2 eggs = 'bacon local'  
    print(eggs) # prints 'bacon local'  
    spam()  
    print(eggs) # prints 'bacon local'  
  
3 eggs = 'global'
```

```
bacon()  
print(eggs)      # prints 'global'
```

When you run this program, it outputs the following:

```
bacon local  
spam local  
bacon local  
global
```

You can view the execution of this program at <https://autbor.com/localglobalsamename/>. There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

- ❶ A variable named eggs that exists in a local scope when spam() is called.
- ❷ A variable named eggs that exists in a local scope when bacon() is called.
- ❸ A variable named eggs that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

THE GLOBAL STATEMENT

If you need to modify a global variable from within a function, use the global statement. If you have a line such as `global eggs` at the top of a function, it tells Python, “In this function, eggs refers to the global variable, so don’t create a local variable with this name.” For example, enter the following code into the file editor and save it as *globalStatement.py*:

```
def spam():  
    ❶ global eggs  
    ❷ eggs = 'spam'  
  
eggs = 'global'  
spam()  
print(eggs)
```

When you run this program, the final print() call will output this:

```
spam
```

You can view the execution of this program at <https://autbor.com/globalstatement/>. Because eggs is declared global at the top of spam() ❶, when eggs is set to 'spam' ❷, this assignment is done to the globally scoped eggs. No local eggs variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
 - If there is a global statement for that variable in a function, it is a global variable.
-

- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

To get a better feel for these rules, here's an example program. Enter the following code into the file editor and save it as *sameNameLocalGlobal.py*:

```
def spam():
    ❶ global eggs
    eggs = 'spam' # this is the global

def bacon():
    ❷ eggs = 'bacon' # this is a local

def ham():
    ❸ print(eggs) # this is the global

eggs = 42 # this is the global
spam()
print(eggs)
```

In the `spam()` function, `eggs` is the global `eggs` variable because there's a global statement for `eggs` at the beginning of the function ❶. In `bacon()`, `eggs` is a local variable because there's an assignment statement for it in that function ❷. In `ham()` ❸, `eggs` is the global variable because there is no assignment statement or global statement for it in that function. If you run *sameNameLocalGlobal.py*, the output will look like this:

```
spam
```

You can view the execution of this program at <https://autbor.com/sameNameLocalGlobal/>. In a function, a variable will either always be global or always be local. The code in a function can't use a local variable named `eggs` and then use the global `eggs` variable later in that same function.

NOTE

If you ever want to modify the value stored in a global variable from in a function, you must use a global statement on that variable.

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, enter the following into the file editor and save it as *sameNameError.py*:

```
def spam():
    print(eggs) # ERROR!
    ❶ eggs = 'spam local'

❷ eggs = 'global'
spam()
```

If you run the previous program, it produces an error message.

Traceback (most recent call last):

File "C:/sameNameError.py", line 6, in <module>

spam()

File "C:/sameNameError.py", line 2, in spam

print(eggs) # ERROR!

UnboundLocalError: local variable 'eggs' referenced before assignment

You can view the execution of this program at <https://autbor.com/sameNameError/>. This error happens because Python sees that there is an assignment statement for eggs in the spam() function ❶ and, therefore, considers eggs to be local. But because print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn't exist. Python will *not* fall back to using the global eggs variable ❷.

FUNCTIONS AS “BLACK BOXES”

Often, all you need to know about a function are its inputs (the parameters) and output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating a function as a “black box.”

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

EXCEPTION HANDLING

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a divide-by-zero error. Open a file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divideBy):  
    return 42 / divideBy
```

```
print(spam(2))  
print(spam(12))  
print(spam(0))  
print(spam(1))
```

We've defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

21.0

3.5

Traceback (most recent call last):

File "C:/zeroDivide.py", line 6, in <module>

print(spam(0))

File "C:/zeroDivide.py", line 2, in spam

return 42 / divideBy

ZeroDivisionError: division by zero

You can view the execution of this program at <https://autbor.com/zerodivide/>. A ZeroDivisionError happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the return statement in spam() is causing an error.

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens.

You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
```

```
    try:
```

```
        return 42 / divideBy
```

```
    except ZeroDivisionError:
```

```
        print('Error: Invalid argument.')
```

```
print(spam(2))
```

```
print(spam(12))
```

```
print(spam(0))
```

```
print(spam(1))
```

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

21.0

3.5

Error: Invalid argument.

None

42.0

You can view the execution of this program at <https://autbor.com/tryexceptzerodivide/>. Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

```
def spam(divideBy):
```

```
    return 42 / divideBy
```

```
try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

You can view the execution of this program at <https://autbor.com/spamintry/>. The reason `print(spam(1))` is never executed is because once the execution jumps to the code in the `except` clause, it does not return to the `try` clause. Instead, it just continues moving down the program as normal.

A SHORT PROGRAM: ZIGZAG

Let's use the programming concepts you've learned so far to create a small animation program. This program will create a back-and-forth, zigzag pattern until the user stops it by pressing the Mu editor's Stop button or by pressing CTRL-C. When you run this program, the output will look something like this:

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

Type the following source code into the file editor, and save the file as *zigzag.py*:

```
import time, sys
indent = 0 # How many spaces to indent.
indentIncreasing = True # Whether the indentation is increasing or not.

try:
    while True: # The main program loop.
        print(' ' * indent, end=")
        print('*****')
        time.sleep(0.1) # Pause for 1/10 of a second.
```

```
if indentIncreasing:
    # Increase the number of spaces:
    indent = indent + 1
    if indent == 20:
        # Change direction:
        indentIncreasing = False

else:
    # Decrease the number of spaces:
    indent = indent - 1
    if indent == 0:
        # Change direction:
        indentIncreasing = True
except KeyboardInterrupt:
    sys.exit()
```

Let's look at this code line by line, starting at the top.

```
import time, sys
indent = 0 # How many spaces to indent.
indentIncreasing = True # Whether the indentation is increasing or not.
```

First, we'll import the time and sys modules. Our program uses two variables: the indent variable keeps track of how many spaces of indentation are before the band of eight asterisks and indentIncreasing contains a Boolean value to determine if the amount of indentation is increasing or decreasing.

```
try:
    while True: # The main program loop.
        print(' ' * indent, end="")
        print('*****')
        time.sleep(0.1) # Pause for 1/10 of a second.
```

Next, we place the rest of the program inside a try statement. When the user presses CTRL-C while a Python program is running, Python raises the KeyboardInterrupt exception. If there is no try-except statement to catch this exception, the program crashes with an ugly error message. However, for our program, we want it to cleanly handle the KeyboardInterrupt exception by calling sys.exit(). (The code for this is in the except statement at the end of the program.)

The while True: infinite loop will repeat the instructions in our program forever. This involves using ' ' * indent to print the correct amount of spaces of indentation. We don't want to automatically print a newline after these spaces, so we also pass end="" to the first print() call. A second print() call prints the band of asterisks. The time.sleep() function hasn't been covered yet, but suffice it to say that it introduces a one-tenth-second pause in our program at this point.

```
if indentIncreasing:
    # Increase the number of spaces:
```

```
indent = indent + 1
if indent == 20:
    indentIncreasing = False # Change direction.
```

Next, we want to adjust the amount of indentation for the next time we print asterisks. If `indentIncreasing` is `True`, then we want to add one to `indent`. But once `indent` reaches 20, we want the indentation to decrease.

```
else:
    # Decrease the number of spaces:
    indent = indent - 1
    if indent == 0:
        indentIncreasing = True # Change direction.
```

Meanwhile, if `indentIncreasing` was `False`, we want to subtract one from `indent`. Once `indent` reaches 0, we want the indentation to increase once again. Either way, the program execution will jump back to the start of the main program loop to print the asterisks again.

```
except KeyboardInterrupt:
    sys.exit()
```

If the user presses CTRL-C at any point that the program execution is in the `try` block, the `KeyboardInterrupt` exception is raised and handled by this `except` statement. The program execution moves inside the `except` block, which runs `sys.exit()` and quits the program. This way, even though the main program loop is an infinite loop, the user has a way to shut down the program.

SUMMARY

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions. This limits what code could be changing the values of your variables, which can be helpful when it comes to debugging your code.

Functions are a great tool to help you organize your code. You can think of them as black boxes: they have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

PRACTICE QUESTIONS

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?

5. How many global scopes are there in a Python program? How many local scopes?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a return statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?
10. What is the data type of None?
11. What does the import areallyourpetsnamederic statement do?
12. If you had a function named bacon() in a module named spam, how would you call it after importing spam?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the try clause? What goes in the except clause?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then write a program that lets the user type in an integer and that keeps calling `collatz()` on that number until the function returns the value 1. (Amazingly enough, this sequence actually works for any integer—sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called “the simplest impossible math problem.”)

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value.

Hint: An integer number is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

The output of this program could look something like this:

Enter number:

3
10
5
16
8
4
2
1

Input Validation

Add try and except statements to the previous project to detect whether the user types in a noninteger string. Normally, the `int()` function will raise a `ValueError` error if it is passed a

noninteger string, as in `int('puppy')`. In the except clause, print a message to the user saying they must enter an integer.

LISTS

One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes writing programs that handle large amounts of data easier. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then I'll briefly cover the sequence data types (lists, tuples, and strings) and show how they compare with each other. In the next chapter, I'll introduce you to the dictionary data type.

THE LIST DATA TYPE

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: ['cat', 'bat', 'rat', 'elephant']. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, []. Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The spam variable ❶ is still assigned only one value: the list value. But the list value itself contains other values. The value [] is an empty list that contains no values, similar to "", the empty string.

Getting Individual Values in a List with Indexes

Say you have the list ['cat', 'bat', 'rat', 'elephant'] stored in a variable named spam. The Python code spam[0] would evaluate to 'cat', and spam[1] would evaluate to 'bat', and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index 0, the second value is at index 1, the third value is at index 2, and so on. [Figure 4-1](#) shows a list value assigned to spam, along with what the index expressions would evaluate to. Note that because the first index is 0, the last index is one less than the size of the list; a list of four items has 3 as its last index.

```
spam = ["cat", "bat", "rat", "elephant"]
```

```

      ↗       ↗       ↗       ↗
spam[0]  spam[1]  spam[2]  spam[3]

```

Figure 4-1: A list value stored in the variable *spam*, showing which value each index refers to

For example, enter the following expressions into the interactive shell. Start by assigning a list to the variable *spam*.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello, ' + spam[0]
❷ 'Hello, cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notice that the expression `'Hello, ' + spam[0]` ❶ evaluates to `'Hello, ' + 'cat'` because `spam[0]` evaluates to the string `'cat'`. This expression in turn evaluates to the string value `'Hello, cat'` ❷.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a `TypeError` error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers or slices, not float
>>> spam[int(1.0)]
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints 'bat', the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at 0 and go up, you can also use negative integers for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '!'
'The elephant is afraid of the bat.'
```

Getting a List from Another List with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with the len() Function

The len() function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

Normally, a variable name goes on the left side of an assignment statement, like spam = 42. However, you can also use an index of a list to change the value at that index. For example, spam[1] = 'aardvark' means -Assign the value at index 1 in the list spam to the string 'aardvark'. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

Lists can be concatenated and replicated just like strings. The + operator combines two lists to create a new list value and the * operator can be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
```

```
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

The del statement can also be used on a simple variable to delete it, as if it were an `--unassignment` statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists. In practice, you almost never need to delete simple variables. The del statement is mostly used to delete values from lists.

WORKING WITH LISTS

When you first begin writing programs, it's tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

It turns out that this is a bad way to write code. (Also, I don't actually own this many cats, I swear.) For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
```

```
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print("The cat names are:")
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, enter the following source code and save it as *allMyCats2.py*:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
        ' (Or enter nothing to stop.):')
    name = input()
    if name == "":
        break
    catNames = catNames + [name] # list concatenation
print("The cat names are:")
for name in catNames:
    print(' ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop.):
```

The cat names are:

Zophie
Pooka

Simon
Lady Macbeth
Fat-tail
Miss Cleo

You can view the execution of these programs at <https://autbor.com/allmycats1/> and <https://autbor.com/allmycats2/>. The benefit of using a list is that your data is now in a structure, so your program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

In [Chapter 2](#), you learned about using for loops to execute a block of code a certain number of times. Technically, a for loop repeats the code block once for each item in a list value. For example, if you ran this code:

```
for i in range(4):  
    print(i)
```

the output of this program would be as follows:

```
0  
1  
2  
3
```

This is because the return value from `range(4)` is a sequence value that Python considers similar to `[0, 1, 2, 3]`. (Sequences are described in [-Sequence Data Types](#) on [page 93](#).) The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:  
    print(i)
```

The previous for loop actually loops through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

A common Python technique is to use `range(len(someList))` with a for loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']  
>>> for i in range(len(supplies)):  
...     print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

```
Index 0 in supplies is: pens  
Index 1 in supplies is: staplers  
Index 2 in supplies is: flamethrowers  
Index 3 in supplies is: binders
```

Using `range(len(supplies))` in the previously shown for loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`).

Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as `myPets.py`:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

The output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

You can view the execution of this program at <https://autbor.com/mypets/>.

The Multiple Assignment Trick

The *multiple assignment trick* (technically called *tuple unpacking*) is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a ValueError:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: not enough values to unpack (expected 4, got 3)
```

Using the enumerate() Function with Lists

Instead of using the range(len(*someList*)) technique with a for loop to obtain the integer index of the items in the list, you can call the enumerate() function instead. On each iteration of the loop, enumerate() will return two values: the index of the item in the list, and the item in the list itself. For example, this code is equivalent to the code in the [-Using for Loops with Lists](#) on page 84:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)
```

```
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

The enumerate() function is useful if you need both the item and the item's index in the loop's block.

Using the random.choice() and random.shuffle() Functions with Lists

The random module has a couple functions that accept lists for arguments. The random.choice() function will return a randomly selected item from the list. Enter the following into the interactive shell:

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Dog'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
```

You can consider `random.choice(someList)` to be a shorter form of `someList[random.randint(0, len(someList) - 1)]`.

The `random.shuffle()` function will reorder the items in a list. This function modifies the list in place, rather than returning a new list. Enter the following into the interactive shell:

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)
>>> people
['Alice', 'David', 'Bob', 'Carol']
```

AUGMENTED ASSIGNMENT OPERATORS

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable `spam`, you would increase the value in `spam` by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator `+=` to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the `+`, `-`, `*`, `/`, and `%` operators, described in [Table 4-1](#).

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
<code>spam += 1</code>	<code>spam = spam + 1</code>
<code>spam -= 1</code>	<code>spam = spam - 1</code>
<code>spam *= 1</code>	<code>spam = spam * 1</code>
<code>spam /= 1</code>	<code>spam = spam / 1</code>
<code>spam %= 1</code>	<code>spam = spam % 1</code>

The `+=` operator can also do string and list concatenation, and the `*=` operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello,'
>>> spam += ' world!'
>>> spam
'Hello world!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

METHODS

A *method* is the same thing as a function, except it is –called on a value. For example, if a list value were stored in `spam`, you would call the `index()` list method (which I’ll explain shortly) on that list like so: `spam.index('hello')`. The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the `index()` Method

List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’t in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that `index()` returns 1, not 3:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

Adding Values to Lists with the `append()` and `insert()` Methods

To add new values to a list, use the `append()` and `insert()` methods. Enter the following into the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
```

```
>>> spam
['cat', 'dog', 'bat', 'moose']
```

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Notice that the code is `spam.append('moose')` and `spam.insert(1, 'chicken')`, not `spam = spam.append('moose')` and `spam = spam.insert(1, 'chicken')`. Neither `append()` nor `insert()` gives the new value of `spam` as its return value. (In fact, the return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place*. Modifying a list in place is covered in more detail later in [–Mutable and Immutable Data Types](#) on page 94.

Methods belong to a single data type. The `append()` and `insert()` methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the `AttributeError` error messages that show up:

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with the `remove()` Method

The `remove()` method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error that is displayed:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The del statement is good to use when you know the index of the value you want to remove from the list. The remove() method is useful when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the sort() method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order. Enter the following into the interactive shell:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

There are three things you should note about the sort() method. First, the sort() method sorts the list in place; don't try to capture the return value by writing code like spam = spam.sort().

Second, you cannot sort lists that have both number values *and* string values in them, since Python doesn't know how to compare these values. Enter the following into the interactive shell and notice the TypeError error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Third, `sort()` uses `-ASCIIbetical order` rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase Z. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the `key` keyword argument in the `sort()` method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Reversing the Values in a List with the `reverse()` Method

If you need to quickly reverse the order of the items in a list, you can call the `reverse()` list method. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()
>>> spam
['moose', 'dog', 'cat']
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines does not matter; Python knows that the list is not finished until it sees the ending square bracket. For example, you can have code that looks like this:

```
spam = ['apples',
        'oranges',
```

```
'bananas',  
'cats']  
print(spam)
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in the Magic 8 Ball program.

You can also split up a single instruction across multiple lines using the *\ line continuation character* at the end. Think of ** as saying, "This instruction continues on the next line." The indentation on the line after a *\ line continuation* is not significant. For example, the following is valid Python code:

```
print('Four score and seven ' + \  
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

Like the `sort()` list method, `reverse()` doesn't return a list. This is why you write `spam.reverse()`, instead of `spam = spam.reverse()`.

EXAMPLE PROGRAM: MAGIC 8 BALL WITH A LIST

Using lists, you can write a much more elegant version of the previous chapter's Magic 8 Ball program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*.

```
import random  
  
messages = ['It is certain',  
            'It is decidedly so',  
            'Yes definitely',  
            'Reply hazy try again',  
            'Ask again later',  
            'Concentrate and ask again',  
            'My reply is no',  
            'Outlook not so good',  
            'Very doubtful']  
  
print(messages[random.randint(0, len(messages) - 1)])
```

You can view the execution of this program at <https://author.com/magic8ball2/>.

When you run this program, you'll see that it works the same as the previous *magic8Ball.py* program.

Notice the expression you use as the index for messages: `random.randint(0, len(messages) - 1)`. This produces a random number to use for the index, regardless of the size of messages. That is, you'll get a random number between 0 and the value of `len(messages) - 1`. The benefit of this approach is that you can easily add and remove strings to the messages list without changing other lines of code. If you later update your

code, there will be fewer lines you have to change and fewer chances for you to introduce bugs.

SEQUENCE DATA TYPES

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar if you consider a string to be a -list of single text characters. The Python sequence data types include lists, strings, range objects returned by range(), and tuples (explained in the [The Tuple Data Type](#) on page 96). Many of the things you can do with lists can also be done with strings and other values of sequence types: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
...     print('* * * ' + i + ' * * *')

* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a *mutable* data type: it can have values added, removed, or changed. However, a string is *immutable*: it cannot be changed. Trying to reassign a single character in a string results in a TypeError error, as you can see by entering the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to *mutate* a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string. Enter the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> newName
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified, because strings are immutable.

Although a list value *is* mutable, the second line in the following code does not modify the list eggs:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

The list value in eggs isn't being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]). This is depicted in [Figure 4-2](#).

If you wanted to actually modify the original list in eggs to contain [4, 5, 6], you would have to do something like this:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

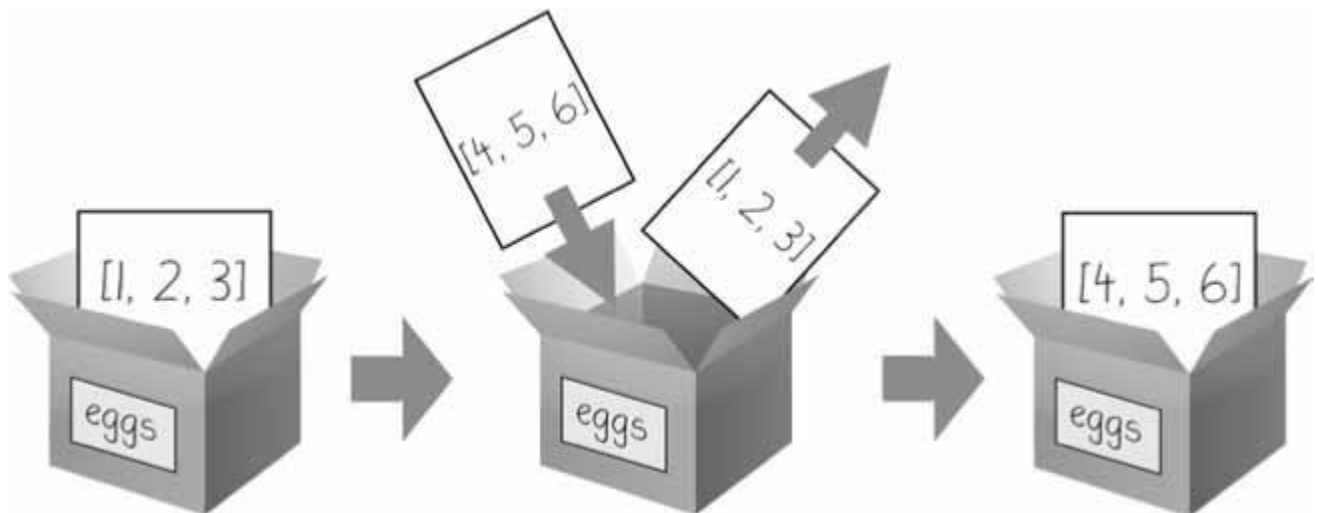


Figure 4-2: When `eggs = [4, 5, 6]` is executed, the contents of `eggs` are replaced with a new list value.

In the first example, the list value that `eggs` ends up with is the same list value it started with. It's just that this list has been changed, rather than overwritten. [Figure 4-3](#) depicts the seven changes made by the first seven lines in the previous interactive shell example.

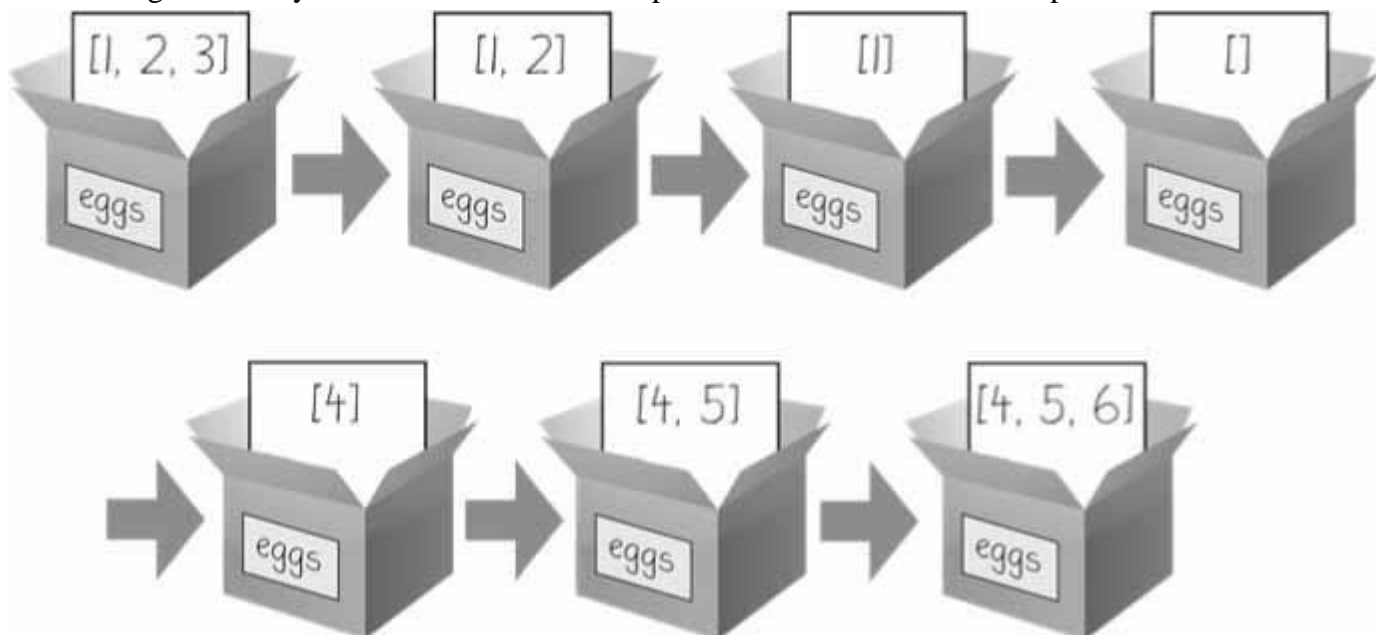


Figure 4-3: The `del` statement and the `append()` method modify the same list value in place.

Changing a value of a mutable data type (like what the `del` statement and `append()` method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.

Mutable versus immutable types may seem like a meaningless distinction, but [–Passing References](#) on [page 100](#) will explain the different behavior when calling functions with mutable arguments versus immutable arguments. But first, let's find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

The *tuple* data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and]. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed. Enter the following into the interactive shell, and look at the `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, it's fine to have a trailing comma after the last item in a list or tuple in Python.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))
<class 'tuple'>
>>> type('hello')
<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Converting Types with the list() and tuple() Functions

Just like how `str(42)` will return `'42'`, the string representation of the integer 42, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:


```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a tuple to a list is handy if you need a mutable version of a tuple value.

REFERENCES

As you've seen, variables store strings and integer values. However, this explanation is a simplification of what Python is actually doing. Technically, variables are storing references to the computer memory locations where the values are stored. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

When you assign 42 to the spam variable, you are actually creating the 42 value in the computer's memory and storing a *reference* to it in the spam variable. When you copy the value in spam and assign it to the variable cheese, you are actually copying the reference. Both the spam and cheese variables refer to the 42 value in the computer's memory. When you later change the value in spam to 100, you're creating a new 100 value and storing a reference to it in spam. This doesn't affect the value in cheese. Integers are *immutable* values that don't change; changing the *spam* variable is actually making it refer to a completely different value in memory.

But lists don't work this way, because list values can change; that is, lists are *mutable*. Here is some code that will make this distinction easier to understand. Enter this into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam # The reference is being copied, not the list.
❸ >>> cheese[1] = 'Hello!' # This changes the list value.
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese # The cheese variable refers to the same list.
[0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code touched only the cheese list, but it seems that both the cheese and spam lists have changed.

When you create the list ❶, you assign a reference to it in the spam variable. But the next line ❷ copies only the list reference in spam to cheese, not the list value itself. This means

the values stored in spam and cheese now both refer to the same list. There is only one underlying list because the list itself was never actually copied. So when you modify the first element of cheese ❸, you are modifying the same list that spam refers to.

Remember that variables are like boxes that contain values. The previous figures in this chapter show that lists in boxes aren't exactly accurate, because list variables don't actually contain lists—they contain *references* to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, [Figure 4-4](#) shows what happens when a list is assigned to the spam variable.

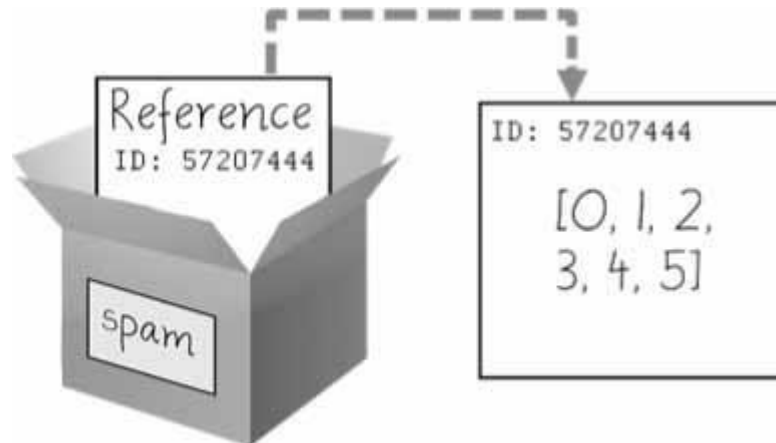


Figure 4-4: `spam = [0, 1, 2, 3, 4, 5]` stores a reference to a list, not the actual list.

Then, in [Figure 4-5](#), the reference in spam is copied to cheese. Only a new reference was created and stored in cheese, not a new list. Note how both references refer to the same list.

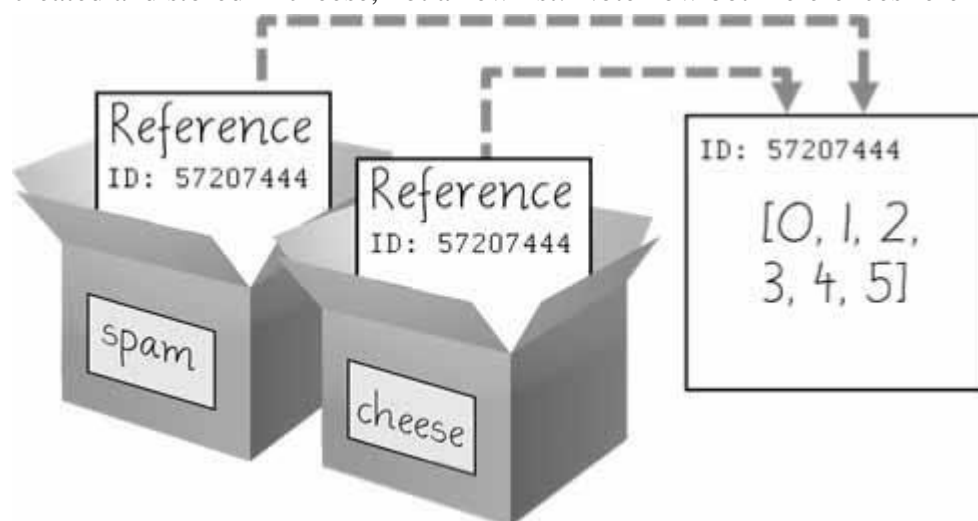


Figure 4-5: `spam = cheese` copies the reference, not the list.

When you alter the list that cheese refers to, the list that spam refers to is also changed, because both cheese and spam refer to the same list. You can see this in [Figure 4-6](#).

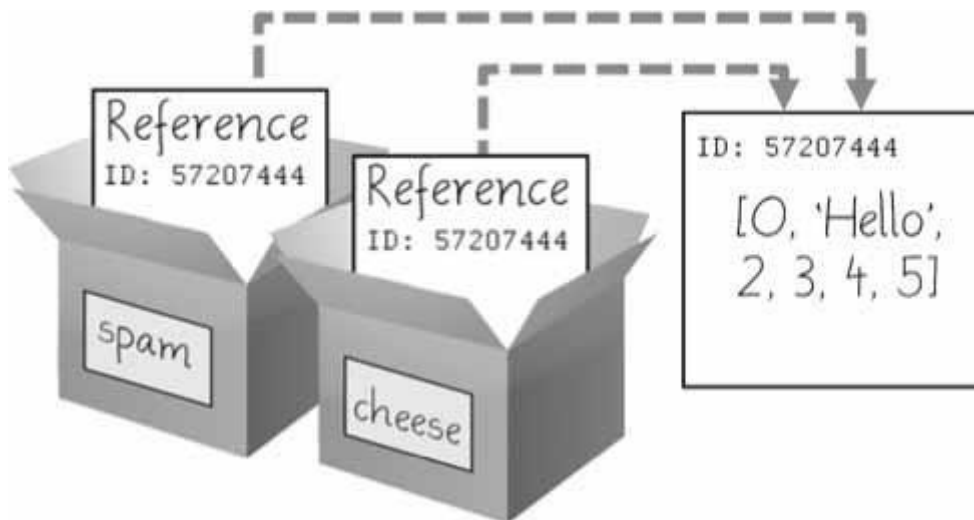


Figure 4-6: `cheese[1] = 'Hello!'` modifies the list that both variables refer to.

Although Python variables technically contain references to values, people often casually say that the variable contains the value.

Identity and the `id()` Function

You may be wondering why the weird behavior with mutable lists in the previous section doesn't happen with immutable values like integers or strings. We can use Python's `id()` function to understand this. All values in Python have a unique identity that can be obtained with the `id()` function. Enter the following into the interactive shell:

```
>>> id('Howdy') # The returned number will be different on your machine.
44491136
```

When Python runs `id('Howdy')`, it creates the 'Howdy' string in the computer's memory. The numeric memory address where the string is stored is returned by the `id()` function. Python picks this address based on which memory bytes happen to be free on your computer at the time, so it'll be different each time you run this code.

Like all strings, 'Howdy' is immutable and cannot be changed. If you `reassign` the string in a variable, a new string object is being made at a different place in memory, and the variable refers to this new string. For example, enter the following into the interactive shell and see how the identity of the string referred to by `bacon` changes:

```
>>> bacon = 'Hello'
>>> id(bacon)
44491136
>>> bacon += ' world!' # A new string is made from 'Hello' and ' world!'.
>>> id(bacon) # bacon now refers to a completely different string.
44609712
```

However, lists can be modified because they are mutable objects. The `append()` method doesn't create a new list object; it changes the existing list object. We call this *modifying the object in-place*.

```
>>> eggs = ['cat', 'dog'] # This creates a new list.
>>> id(eggs)
35152584
>>> eggs.append('moose') # append() modifies the list "in place".
>>> id(eggs) # eggs still refers to the same list as before.
35152584
>>> eggs = ['bat', 'rat', 'cow'] # This creates a new list, which has a new
identity.
>>> id(eggs) # eggs now refers to a completely different list.
44409800
```

If two variables refer to the same list (like `spam` and `cheese` in the previous section) and the list value itself changes, both variables are affected because they both refer to the same list. The `append()`, `extend()`, `remove()`, `sort()`, `reverse()`, and other list methods modify their lists in place.

Python's *automatic garbage collector* deletes any values not being referred to by any variables to free up memory. You don't need to worry about how the garbage collector works, which is a good thing: manual memory management in other programming languages is a common source of bugs.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries, which I'll describe in the next chapter), this means a copy of the reference is used for the parameter. To see the consequences of this, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when `eggs()` is called, a return value is not used to assign a new value to `spam`. Instead, it modifies the list in place, directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

The copy Module's copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value. For this, Python provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. The first of these, `copy.copy()`, can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> id(spam)
44684232
>>> cheese = copy.copy(spam)
>>> id(cheese) # cheese is a different list with different identity.
44685832
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

Now the `spam` and `cheese` variables refer to separate lists, which is why only the list in `cheese` is modified when you assign 42 at index 1. As you can see in [Figure 4-7](#), the reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

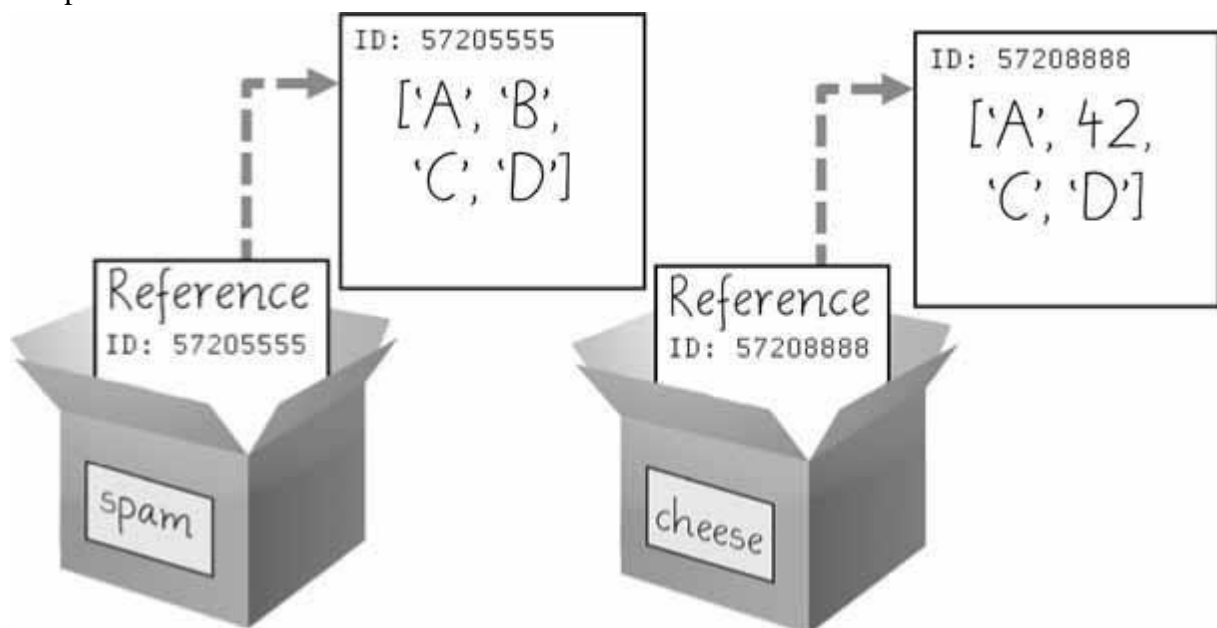


Figure 4-7: `cheese = copy.copy(spam)` creates a second list that can be modified independently of the first.

If the list you need to copy contains lists, then use the `copy.deepcopy()` function instead of `copy.copy()`. The `deepcopy()` function will copy these inner lists as well.

A SHORT PROGRAM: CONWAY'S GAME OF LIFE

Conway's Game of Life is an example of *cellular automata*: a set of rules governing the behavior of a field made up of discrete cells. In practice, it creates a pretty animation to look at. You can draw out each step on graph paper, using the squares as cells. A filled-in square will be -alive- and an empty square will be -dead-. If a living square has two or three living neighbors, it continues to live on the next step. If a dead square has exactly three living neighbors, it comes alive on the next step. Every other square dies or remains dead on the next step. You can see an example of the progression of steps in [Figure 4-8](#).

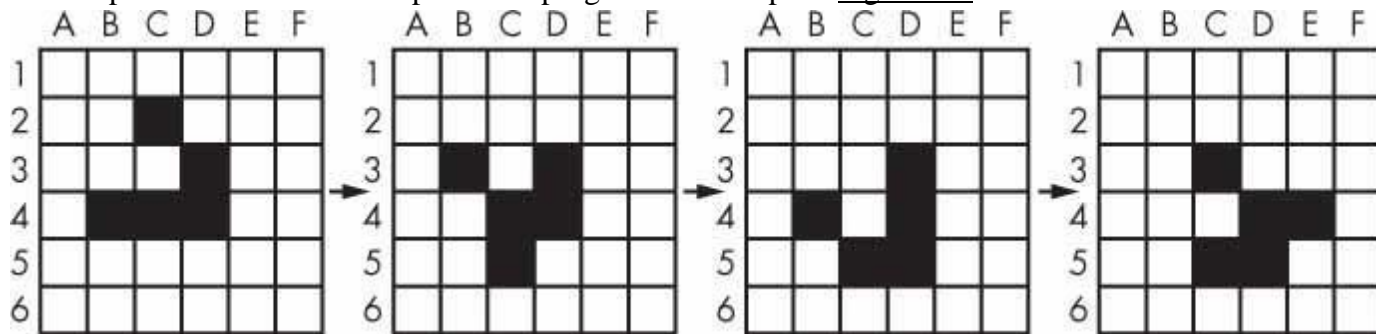


Figure 4-8: Four steps in a Conway's Game of Life simulation

Even though the rules are simple, there are many surprising behaviors that emerge. Patterns in Conway's Game of Life can move, self-replicate, or even mimic CPUs. But at the foundation of all of this complex, advanced behavior is a rather simple program.

We can use a list of lists to represent the two-dimensional field. The inner list represents each column of squares and stores a '#' hash string for living squares and a ' ' space string for dead squares. Type the following source code into the file editor, and save the file as *conway.py*. It's fine if you don't quite understand how all of the code works; just enter it and follow along with comments and explanations provided here as close as you can:

```
# Conway's Game of Life
import random, time, copy
WIDTH = 60
HEIGHT = 20

# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
        else:
            column.append(' ') # Add a dead cell.
    nextCells.append(column) # nextCells is a list of column lists.

while True: # Main program loop.
    print('\n\n\n\n\n') # Separate each step with newlines.
    currentCells = copy.deepcopy(nextCells)

    # Print currentCells on the screen:
    for y in range(HEIGHT):
```

```
for x in range(WIDTH):
    print(currentCells[x][y], end=") # Print the # or space.
    print() # Print a newline at the end of the row.

# Calculate the next step's cells based on current step's cells:
for x in range(WIDTH):
    for y in range(HEIGHT):
        # Get neighboring coordinates:
        # `% WIDTH` ensures leftCoord is always between 0 and WIDTH - 1
        leftCoord = (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT

        # Count number of living neighbors:
        numNeighbors = 0
        if currentCells[leftCoord][aboveCoord] == '#':
            numNeighbors += 1 # Top-left neighbor is alive.
        if currentCells[x][aboveCoord] == '#':
            numNeighbors += 1 # Top neighbor is alive.
        if currentCells[rightCoord][aboveCoord] == '#':
            numNeighbors += 1 # Top-right neighbor is alive.
        if currentCells[leftCoord][y] == '#':
            numNeighbors += 1 # Left neighbor is alive.
        if currentCells[rightCoord][y] == '#':
            numNeighbors += 1 # Right neighbor is alive.
        if currentCells[leftCoord][belowCoord] == '#':
            numNeighbors += 1 # Bottom-left neighbor is alive.
        if currentCells[x][belowCoord] == '#':
            numNeighbors += 1 # Bottom neighbor is alive.
        if currentCells[rightCoord][belowCoord] == '#':
            numNeighbors += 1 # Bottom-right neighbor is alive.

        # Set cell based on Conway's Game of Life rules:
        if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
            # Living cells with 2 or 3 neighbors stay alive:
            nextCells[x][y] = '#'
        elif currentCells[x][y] == '.' and numNeighbors == 3:
            # Dead cells with 3 neighbors become alive:
            nextCells[x][y] = '#'
        else:
            # Everything else dies or stays dead:
            nextCells[x][y] = '.'
    time.sleep(1) # Add a 1-second pause to reduce flickering.
```

Let's look at this code line by line, starting at the top.


```
# Conway's Game of Life
import random, time, copy
WIDTH = 60
HEIGHT = 20
```

First we import modules that contain functions we'll need, namely the random.randint(), time.sleep(), and copy.deepcopy() functions.

```
# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
        else:
            column.append(' ') # Add a dead cell.
    nextCells.append(column) # nextCells is a list of column lists.
```

The very first step of our cellular automata will be completely random. We need to create a list of lists data structure to store the '#' and ' ' strings that represent a living or dead cell, and their place in the list of lists reflects their position on the screen. The inner lists each represent a column of cells. The random.randint(0, 1) call gives an even 50/50 chance between the cell starting off alive or dead.

We put this list of lists in a variable called nextCells, because the first step in our main program loop will be to copy nextCells into currentCells. For our list of lists data structure, the x-coordinates start at 0 on the left and increase going right, while the y-coordinates start at 0 at the top and increase going down. So nextCells[0][0] will represent the cell at the top left of the screen, while nextCells[1][0] represents the cell to the right of that cell and nextCells[0][1] represents the cell beneath it.

```
while True: # Main program loop.
    print("\n\n\n\n") # Separate each step with newlines.
    currentCells = copy.deepcopy(nextCells)
```

Each iteration of our main program loop will be a single step of our cellular automata. On each step, we'll copy nextCells to currentCells, print currentCells on the screen, and then use the cells in currentCells to calculate the cells in nextCells.

```
# Print currentCells on the screen:
for y in range(HEIGHT):
    for x in range(WIDTH):
        print(currentCells[x][y], end=" ") # Print the # or space.
    print() # Print a newline at the end of the row.
```

These nested for loops ensure that we print a full row of cells to the screen, followed by a newline character at the end of the row. We repeat this for each row in nextCells.

```
# Calculate the next step's cells based on current step's cells:
for x in range(WIDTH):
    for y in range(HEIGHT):
        # Get neighboring coordinates:
        # `% WIDTH` ensures leftCoord is always between 0 and WIDTH - 1
        leftCoord = (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT
```

Next, we need to use two nested for loops to calculate each cell for the next step. The living or dead state of the cell depends on the neighbors, so let's first calculate the index of the cells to the left, right, above, and below the current x- and y-coordinates.

The % mod operator performs a -wraparound. The left neighbor of a cell in the leftmost column 0 would be 0 - 1 or -1. To wrap this around to the rightmost column's index, 59, we calculate (0 - 1) % WIDTH. Since WIDTH is 60, this expression evaluates to 59. This mod-wraparound technique works for the right, above, and below neighbors as well.

```
# Count number of living neighbors:
numNeighbors = 0
if currentCells[leftCoord][aboveCoord] == '#':
    numNeighbors += 1 # Top-left neighbor is alive.
if currentCells[x][aboveCoord] == '#':
    numNeighbors += 1 # Top neighbor is alive.
if currentCells[rightCoord][aboveCoord] == '#':
    numNeighbors += 1 # Top-right neighbor is alive.
if currentCells[leftCoord][y] == '#':
    numNeighbors += 1 # Left neighbor is alive.
if currentCells[rightCoord][y] == '#':
    numNeighbors += 1 # Right neighbor is alive.
if currentCells[leftCoord][belowCoord] == '#':
    numNeighbors += 1 # Bottom-left neighbor is alive.
if currentCells[x][belowCoord] == '#':
    numNeighbors += 1 # Bottom neighbor is alive.
if currentCells[rightCoord][belowCoord] == '#':
    numNeighbors += 1 # Bottom-right neighbor is alive.
```

To decide if the cell at nextCells[x][y] should be living or dead, we need to count the number of living neighbors currentCells[x][y] has. This series of if statements checks each of the eight neighbors of this cell, and adds 1 to numNeighbors for each living one.

```
# Set cell based on Conway's Game of Life rules:
if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
    # Living cells with 2 or 3 neighbors stay alive:
    nextCells[x][y] = '#'
elif currentCells[x][y] == ' ' and numNeighbors == 3:
    # Dead cells with 3 neighbors become alive:
```

```
        nextCells[x][y] = '#'
    else:
        # Everything else dies or stays dead:
        nextCells[x][y] = ''
time.sleep(1) # Add a 1-second pause to reduce flickering.
```

Now that we know the number of living neighbors for the cell at `currentCells[x][y]`, we can set `nextCells[x][y]` to either `'#'` or `' '`. After we loop over every possible `x`- and `y`-coordinate, the program takes a 1-second pause by calling `time.sleep(1)`. Then the program execution goes back to the start of the main program loop to continue with the next step.

Several patterns have been discovered with names such as `-glider`,^{||} `-propeller`,^{||} or `-heavyweight spaceship`.^{||} The glider pattern, pictured in Figure 4-8, results in a pattern that `-moves`^{||} diagonally every four steps. You can create a single glider by replacing this line in our `conway.py` program:

```
if random.randint(0, 1) == 0:
```

with this line:

```
if (x, y) in ((1, 0), (2, 1), (0, 2), (1, 2), (2, 2)):
```

You can find out more about the intriguing devices made using Conway's Game of Life by searching the web. And you can find other short, text-based Python programs like this one at <https://github.com/asweigart/pythonstdiogames>.

SUMMARY

Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you will see programs using lists to do things that would be difficult or impossible to do without them.

Lists are a sequence data type that is mutable, meaning that their contents can change. Tuples and strings, though also sequence data types, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the `append()` or `remove()` methods do on lists.

Variables do not store list values directly; they store *references* to lists. This is an important distinction when you are copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

PRACTICE QUESTIONS

1. What is `[]`?
2. How would you assign the value `'hello'` as the third value in a list stored in a variable named `spam`? (Assume `spam` contains `[2, 4, 6, 8, 10]`.)

For the following three questions, let's say `spam` contains the list `['a', 'b', 'c', 'd']`.

3. What does `spam[int(int('3' * 2) // 11)]` evaluate to?
4. What does `spam[-1]` evaluate to?
5. What does `spam[:2]` evaluate to?

For the following three questions, let's say `bacon` contains the list `[3.14, 'cat', 11, 'cat', True]`.

6. What does `bacon.index('cat')` evaluate to?
7. What does `bacon.append(99)` make the list value in `bacon` look like?
8. What does `bacon.remove('cat')` make the list value in `bacon` look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the `append()` and `insert()` list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you type the tuple value that has just the integer value 42 in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that contain list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between `copy.copy()` and `copy.deepcopy()`?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous `spam` list to the function would return `'apples, bananas, tofu, and cats'`. But your function should be able to work with any list value passed to it. Be sure to test the case where an empty list `[]` is passed to your function.

Coin Flip Streaks

For this exercise, we'll try doing an experiment. If you flip a coin 100 times and write down an `H` for each heads and a `T` for each tails, you'll create a list that looks like `-T T T T H H H H T T`. If you ask a human to make up 100 random coin flips, you'll probably end up with alternating head-tail results like `-H T H T H H T H T T`, which looks random (to humans), but isn't mathematically random. A human will almost never write down a streak of six heads or six tails in a row, even though it is highly likely to happen in truly random coin flips. Humans are predictably bad at being random.

Write a program to find out how often a streak of six heads or a streak of six tails comes up in a randomly generated list of heads and tails. Your program breaks up the experiment into two parts: the first part generates a list of randomly selected 'heads' and 'tails' values, and the second part checks if there is a streak in it. Put all of this code in a loop that repeats the experiment 10,000 times so we can find out what percentage of the coin flips contains a streak of six heads or tails in a row. As a hint, the function call `random.randint(0, 1)` will return a 0 value 50% of the time and a 1 value the other 50% of the time.

You can start with the following template:

```
import random
numberOfStreaks = 0
for experimentNumber in range(10000):
    # Code that creates a list of 100 'heads' or 'tails' values.

    # Code that checks if there is a streak of 6 heads or tails in a row.
print('Chance of streak: %s%%' % (numberOfStreaks / 100))
```

Of course, this is only an estimate, but 10,000 is a decent sample size. Some knowledge of mathematics could give you the exact answer and save you the trouble of writing a program, but programmers are notoriously bad at math.

Character Picture Grid

Say you have a list of lists where each value in the inner lists is a one-character string, like this:

```
grid = [['.', '.', '.', '.', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['.', 'O', 'O', 'O', 'O', 'O'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]
```

Think of `grid[x][y]` as being the character at the x- and y-coordinates of a –picture– drawn with text characters. The (0, 0) origin is in the upper-left corner, the x-coordinates increase going right, and the y-coordinates increase going down.

Copy the previous grid value, and write code that uses it to print the image.

```
..OO.OO..
.OOOOOOO.
.OOOOOOO.
..OOOOO..
...OOO...
....O....
```

Hint: You will need to use a loop in a loop in order to print `grid[0][0]`, then `grid[1][0]`, then `grid[2][0]`, and so on, up to `grid[8][0]`. This will finish the first row, so then print a newline. Then your program should print `grid[0][1]`, then `grid[1][1]`, then `grid[2][1]`, and so on. The last thing your program will print is `grid[8][5]`.

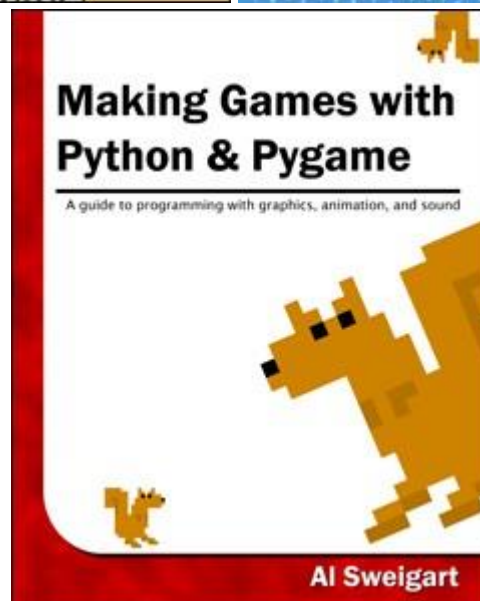
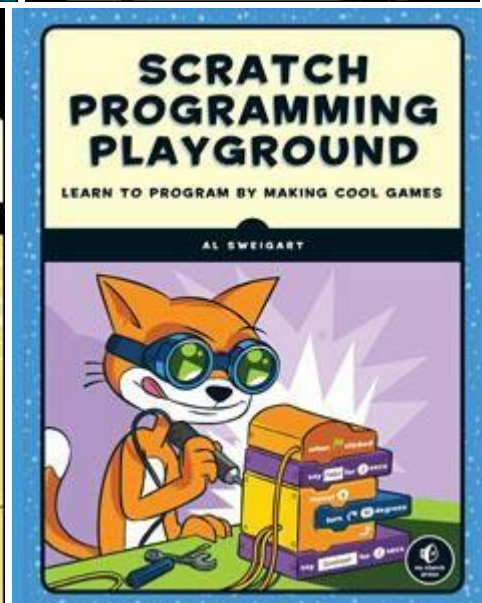
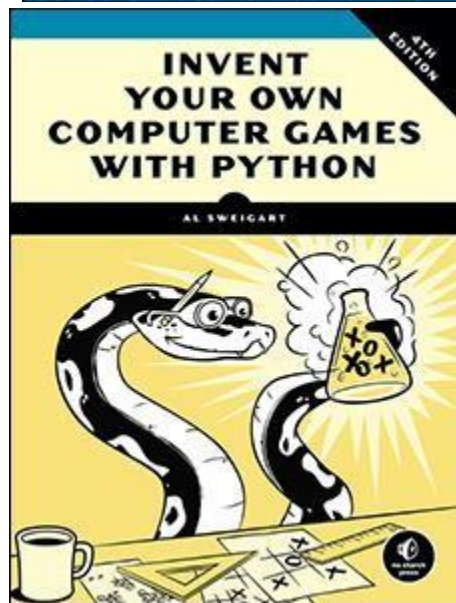
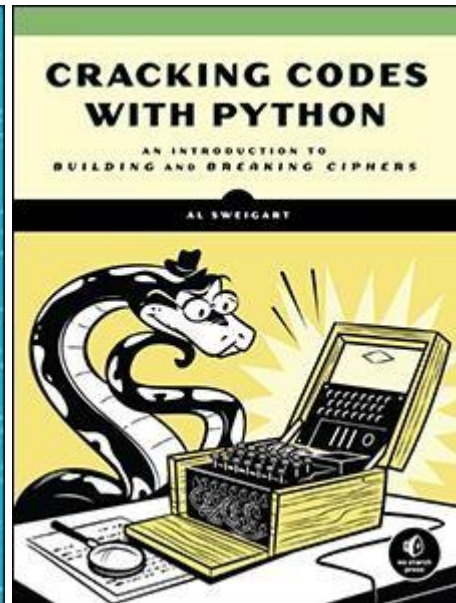
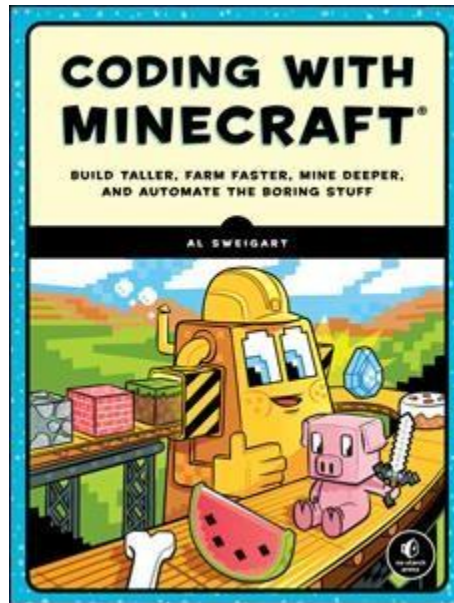
Also, remember to pass the end keyword argument to `print()` if you don't want a newline printed automatically after each `print()` call.

[Home](#) | [Buy on No Starch Press \(comes with free ebook\)](#) | [Buy on Amazon](#) | [@AlSweigart](#) |

Support the author by purchasing the print/ebook bundle from [No Starch Press](#) or separately on [Amazon](#).



Read the author's other Creative Commons licensed Python books.





DICTIONARIES AND STRUCTURING DATA



In this chapter, I will cover the dictionary data type, which provides a flexible way to access and organize data. Then, combining dictionaries with your knowledge of lists from the previous chapter, you'll learn how to create a data structure to model a tic-tac-toe board.

THE DICTIONARY DATA TYPE

Like a list, a *dictionary* is a mutable collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is typed with braces, `{}`. Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively. You can access these values through their keys:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at 0 and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named `spam` would be `spam[0]`. But there is no `-first` item in a dictionary. While the order of items matters for

determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's `-out-of-range` `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code. Save it as *birthdays.py*.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}
```

```
while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == "":
        break

    ❷ if name in birthdays:
        ❸ print(birthdays[name] + ' is the birthday of ' + name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
        ❹ birthdays[name] = bday
        print('Birthday database updated.')
```

You can view the execution of this program at <https://autbor.com/bdaydb>. You create an initial dictionary and store it in birthdays ❶. You can see if the entered name exists as a key in the dictionary with the in keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in [Chapter 9](#).

ORDERED DICTIONARIES IN PYTHON 3.7

While they're still not ordered and have no `-first` key-value pair, dictionaries in Python 3.7 and later will remember the insertion order of their key-value pairs if you create a sequence value from them. For example, notice the order of items in the lists made from the eggs and ham dictionaries matches the order in which they were entered:

```
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> list(eggs)
['name', 'species', 'age']
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> list(ham)
['species', 'age', 'name']
```

The dictionaries are still unordered, as you can't access items in them using integer indexes like `eggs[0]` or `ham[2]`. You shouldn't rely on this behavior, as dictionaries in older versions of Python don't remember the insertion order of key-value pairs. For example, notice how the list doesn't match the insertion order of the dictionary's key-value pairs when I run this code in Python 3.5:

```
>>> spam = {}
>>> spam['first key'] = 'value'
>>> spam['second key'] = 'value'
>>> spam['third key'] = 'value'
```

```
>>> list(spam)
['first key', 'third key', 'second key']
```

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods are not true lists: they cannot be modified and do not have an `append()` method. But these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
...     print(v)

red
42
```

Here, a for loop iterates over each of the values in the `spam` dictionary. A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():
...     print(k)

color
age
>>> for i in spam.items():
...     print(i)

('color', 'red')
('age', 42)
```

When you use the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the `list()` function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

The `list(spam.keys())` line takes the `dict_keys` value returned from `keys()` and passes it to `list()`, which then returns a list value of `['color', 'age']`.

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
...     print('Key: ' + k + ' Value: ' + str(v))
```

```
Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

In the previous example, notice that `'color' in spam` is essentially a shorter version of writing `'color' in spam.keys()`. This is always the case: if you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the `in` (or `not in`) keyword with the dictionary value itself.

The get() Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no `'eggs'` key in the `picnicItems` dictionary, the default value `0` is returned by the `get()` method. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
```

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

KeyError: 'eggs'

The.setdefault() Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'`, because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as *characterCount.py*:

```
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = { }
```

for character in message:

❶ `count.setdefault(character, 0)`

❷ `count[character] = count[character] + 1`

```
print(count)
```

You can view the execution of this program at <https://autbor.com/setdefault>. The program loops over each character in the message variable's string, counting how often each character appears. The `setdefault()` method call ❶ ensures that the key is in the count dictionary (with a default value of 0) so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed ❷. When you run this program, the output will look like this:

```
{ ' ': 13, ',': 1, '.': 1, 'A': 1, 'T': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2,
'i': 6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1 }
```

From the output, you can see that the lowercase letter *c* appears 3 times, the space character appears 13 times, and the uppercase letter *A* appears 1 time. This program will work no matter what string is inside the message variable, even if the string is millions of characters long!

PRETTY PRINTING

If you import the `pprint` module into your programs, you'll have access to the `pprint()` and `pformat()` functions that will -pretty print a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what `print()` provides. Modify the previous *characterCount.py* program and save it as *prettyCharacterCount.py*.

import pprint

```
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = { }
```

```
for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1
```

pprint.pprint(count)

You can view the execution of this program at <https://autbor.com/pprint/>. This time, when the program is run, the output looks much cleaner, with the keys sorted.

```
{ ' ': 13,
  ',': 1,
  '.': 1,
  'A': 1,
  'T': 1,
  '--snip--
  't': 6,
  'w': 2,
  'y': 1 }
```

The `pprint.pprint()` function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call `pprint.pformat()` instead. These two lines are equivalent to each other:

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

USING DATA STRUCTURES TO MODEL REAL-WORLD THINGS

Even before the internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in [Figure 5-1](#).

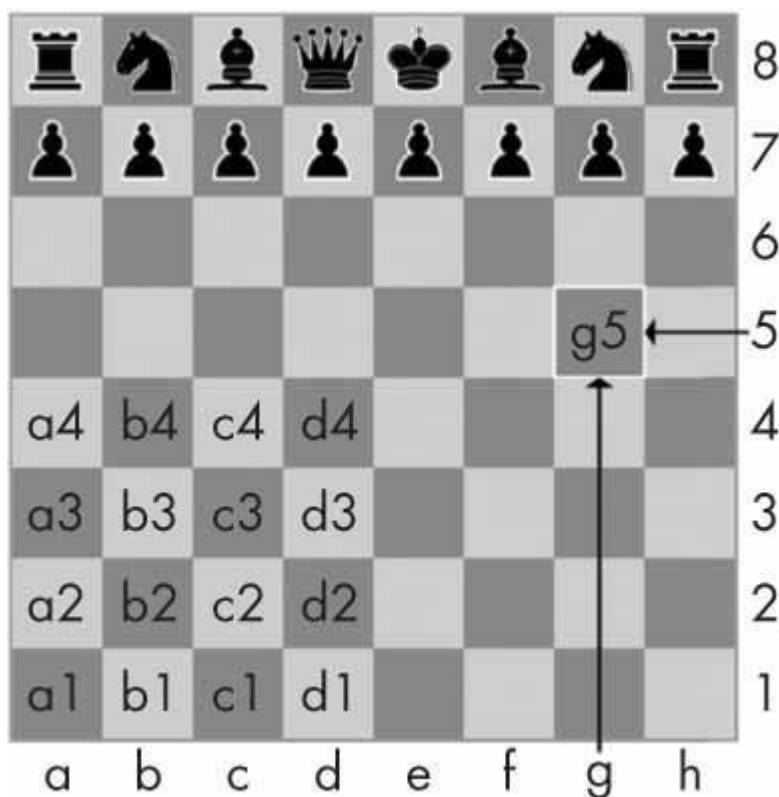


Figure 5-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation 2. *Nf3 Nc6* indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: you can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings like '2. Nf3 Nc6'. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model.

This is where lists and dictionaries can come in. For example, the dictionary {'1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'} could represent the chess board in [Figure 5-2](#).

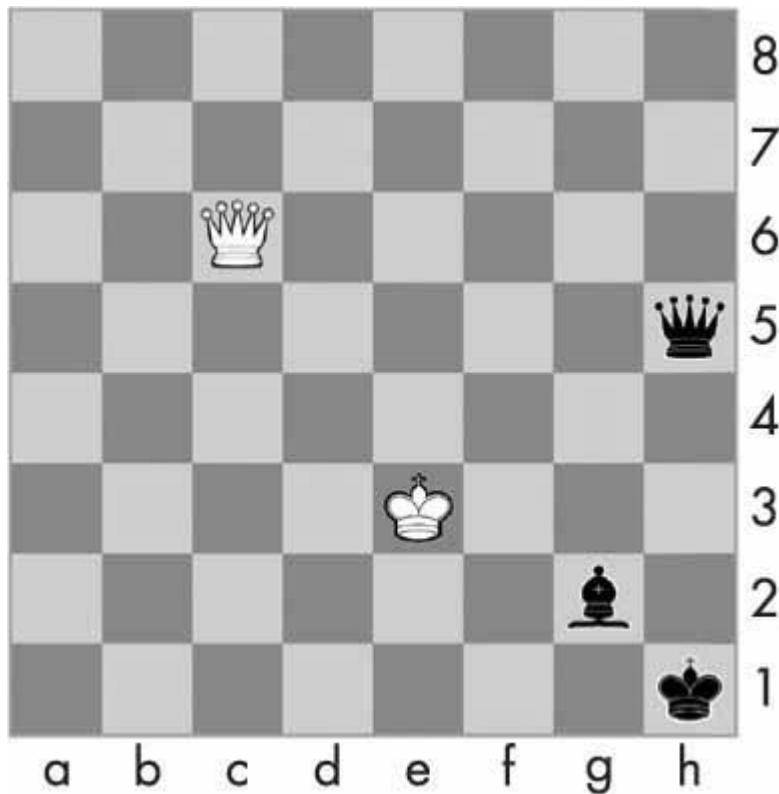


Figure 5-2: A chess board modeled by the dictionary '1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'}

But for another example, you'll use a game that's a little simpler than chess: tic-tac-toe.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an X, an O, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in [Figure 5-3](#).

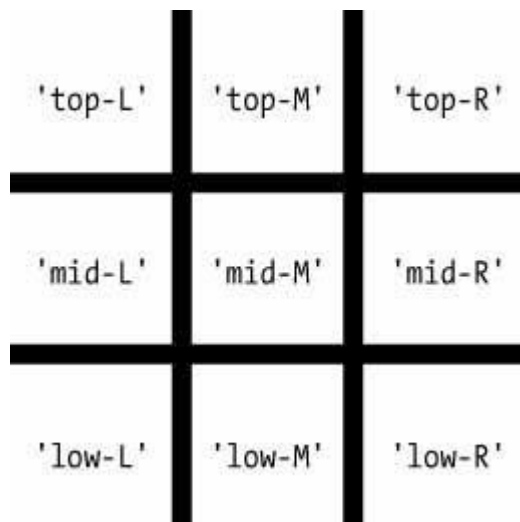


Figure 5-3: The slots of a tic-tac-toe board with their corresponding keys

You can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key 'top-R' can represent the top-right corner, the string value with the key 'low-L' can represent the bottom-left corner, the string value with the key 'mid-M' can represent the middle, and so on.

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named `theBoard`. Open a new file editor window, and enter the following source code, saving it as *ticTacToe.py*:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',  
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',  
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure stored in the `theBoard` variable represents the tic-tac-toe board in [Figure 5-4](#).

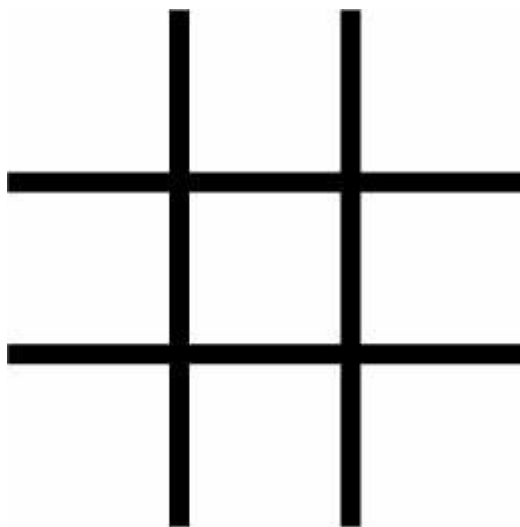


Figure 5-4: An empty tic-tac-toe board

Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': '', 'top-M': '', 'top-R': '',  
            'mid-L': '', 'mid-M': 'X', 'mid-R': '',  
            'low-L': '', 'low-M': '', 'low-R': ''}
```

The data structure in theBoard now represents the tic-tac-toe board in [Figure 5-5](#).

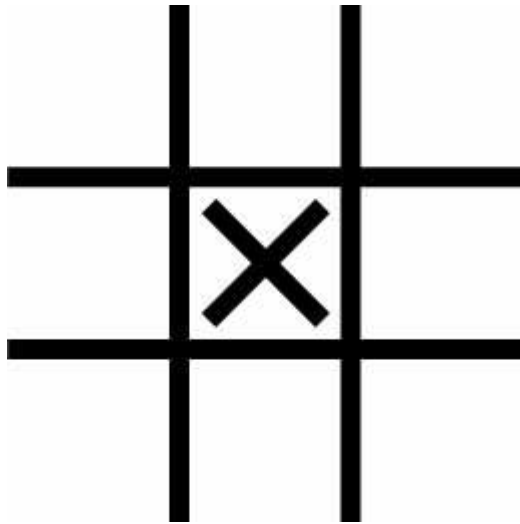


Figure 5-5: The first move

A board where player O has won by placing Os across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',  
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': '',  
            'low-L': '', 'low-M': '', 'low-R': 'X'}
```

The data structure in theBoard now represents the tic-tac-toe board in [Figure 5-6](#).

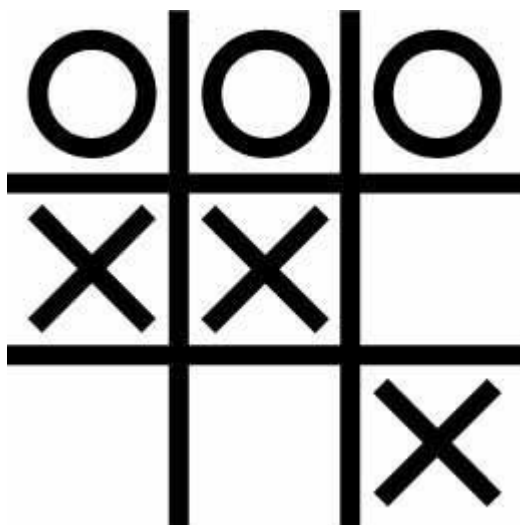


Figure 5-6: Player O wins.

Of course, the player sees only what is printed to the screen, not the contents of variables. Let's create a function to print the board dictionary onto the screen. Make the following addition to *ticTacToe.py* (new code is in bold):

```
theBoard = {'top-L': '', 'top-M': '', 'top-R': '',
            'mid-L': '', 'mid-M': '', 'mid-R': '',
            'low-L': '', 'low-M': '', 'low-R': ''}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

You can view the execution of this program at <https://autbor.com/tictactoe1/>. When you run this program, printBoard() will print out a blank tic-tac-toe board.

```
||
-+-+-
||
-+-+-
||
```

The printBoard() function can handle any tic-tac-toe data structure you pass it. Try changing the code to the following:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
            'X', 'mid-R': '', 'low-L': '', 'low-M': '', 'low-R': 'X'}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

You can view the execution of this program at <https://autbor.com/tictactoe2/>. Now when you run this program, the new board will be printed to the screen.

```
O|O|O
-+-+-
X|X|
-+-+-
|X|
```

Because you created a data structure to represent a tic-tac-toe board and wrote code in `printBoard()` to interpret that data structure, you now have a program that models the tic-tac-toe board. You could have organized your data structure differently (for example, using keys like 'TOP-LEFT' instead of 'top-L'), but as long as the code works with your data structures, you will have a correctly working program.

For example, the `printBoard()` function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots. If the dictionary you passed was missing, say, the 'mid-L' key, your program would no longer work.

```
O|O|O
```

```
--+-
```

```
Traceback (most recent call last):
```

```
File "ticTacToe.py", line 10, in <module>
```

```
    printBoard(theBoard)
```

```
File "ticTacToe.py", line 6, in printBoard
```

```
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
```

```
KeyError: 'mid-L'
```

Now let's add code that allows the players to enter their moves. Modify the *ticTacToe.py* program to look like this:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M':
            ' ', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

```
def printBoard(board):
```

```
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
```

```
    print('--+-')
```

```
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
```

```
    print('--+-')
```

```
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
```

```
turn = 'X'
```

```
for i in range(9):
```

```
    ❶ printBoard(theBoard)
```

```
    print('Turn for ' + turn + '. Move on which space?')
```

```
    ❷ move = input()
```

```
    ❸ theBoard[move] = turn
```

```
    ❹ if turn == 'X':
```

```
        turn = 'O'
```

```
    else:
```

```
        turn = 'X'
```

```
printBoard(theBoard)
```

You can view the execution of this program at <https://autbor.com/tictactoe3/>. The new code prints out the board at the start of each new turn ❶, gets the active player's move ❷, updates the game board accordingly ❸, and then swaps the active player ❹ before moving on to the next turn.

When you run this program, it will look something like this:

```

||
--++--
||
--++--
||
Turn for X. Move on which space?

```

mid-M

```

||
--++--
|X|
--++--
||

```

--snip--

```

O|O|X
--++--
X|X|O
--++--
O| |X
Turn for X. Move on which space?

```

low-M

```

O|O|X
--++--
X|X|O
--++--
O|X|X

```

This isn't a complete tic-tac-toe game—for instance, it doesn't ever check whether a player has won—but it's enough to see how data structures can be used in programs.

NOTE

If you are curious, the source code for a complete tic-tac-toe program is described in the resources available from <https://nostarch.com/automatestuff2/>.

Nested Dictionaries and Lists

Modeling a tic-tac-toe board was fairly simple: the board needed only a single dictionary value with nine key-value pairs. As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary that contains other dictionaries of what items guests are bringing to a picnic. The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

```

allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

```

```

def totalBrought(guests, item):

```

```
numBrought = 0
❶ for k, v in guests.items():
    ❷ numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples      ' + str(totalBrought(allGuests, 'apples'))))
print(' - Cups        ' + str(totalBrought(allGuests, 'cups'))))
print(' - Cakes        ' + str(totalBrought(allGuests, 'cakes'))))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches'))))
print(' - Apple Pies   ' + str(totalBrought(allGuests, 'apple pies'))))
```

You can view the execution of this program at <https://autbor.com/guestpicnic/>. Inside the `totalBrought()` function, the for loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, its value (the quantity) is added to `numBrought` ❷. If it does not exist as a key, the `get()` method returns 0 to be added to `numBrought`.

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1
```

This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `totalBrought()` function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items. Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about the -right way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program's needs.

SUMMARY

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another (the value), as opposed to lists, which simply contain a series of values in order. Values inside a dictionary are accessed using square brackets just as with lists. Instead of an integer index, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program's values into data structures, you can create representations of real-world objects. You saw an example of this with a tic-tac-toe board.

PRACTICE QUESTIONS

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key 'foo' and a value 42 look like?
3. What is the main difference between a dictionary and a list?
4. What happens if you try to access spam['foo'] if spam is {'bar': 100}?
5. If a dictionary is stored in spam, what is the difference between the expressions 'cat' in spam and 'cat' in spam.keys()?
6. If a dictionary is stored in spam, what is the difference between the expressions 'cat' in spam and 'cat' in spam.values()?
7. What is a shortcut for the following code?

```
if 'color' not in spam:
    spam['color'] = 'black'
```

8. What module and function can be used to —pretty print dictionary values?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Chess Dictionary Validator

In this chapter, we used the dictionary value {'1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'} to represent a chess board. Write a function named isValidChessBoard() that takes a dictionary argument and returns True or False depending on if the board is valid.

A valid board will have exactly one black king and exactly one white king. Each player can only have at most 16 pieces, at most 8 pawns, and all pieces must be on a valid space from '1a' to '8h'; that is, a piece can't be on space '9z'. The piece names begin with either a 'w' or 'b' to represent white or black, followed by 'pawn', 'knight', 'bishop', 'rook', 'queen', or 'king'. This function should detect when a bug has resulted in an improper chess board.

Fantasy Game Inventory

You are creating a fantasy video game. The data structure to model the player's inventory will be a dictionary where the keys are string values describing the item in the inventory and the value is an integer value detailing how many of that item the player has. For example, the dictionary value {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12} means the player has 1 rope, 6 torches, 42 gold coins, and so on.

Write a function named displayInventory() that would take any possible —inventory and display it like the following:

```
Inventory:
12 arrow
42 gold coin
1 rope
6 torch
```

1 dagger
Total number of items: 62

Hint: You can use a for loop to loop through all the keys in a dictionary.

```
# inventory.py
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}

def displayInventory(inventory):
    print("Inventory:")
    item_total = 0
    for k, v in inventory.items():
        # FILL THIS PART IN
    print("Total number of items: " + str(item_total))

displayInventory(stuff)
```

List to Dictionary Function for Fantasy Game Inventory

Imagine that a vanquished dragon's loot is represented as a list of strings like this:

```
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Write a function named `addToInventory(inventory, addedItems)`, where the `inventory` parameter is a dictionary representing the player's inventory (like in the previous project) and the `addedItems` parameter is a list like `dragonLoot`. The `addToInventory()` function should return a dictionary that represents the updated inventory. Note that the `addedItems` list can contain multiples of the same item. Your code could look something like this:

```
def addToInventory(inventory, addedItems):
    # your code goes here

inv = {'gold coin': 42, 'rope': 1}
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = addToInventory(inv, dragonLoot)
displayInventory(inv)
```

The previous program (with your `displayInventory()` function from the previous project) would output the following:

```
Inventory:
45 gold coin
1 rope
1 ruby
1 dagger
```

Total number of items: 48

PATTERN MATCHING WITH REGULAR EXPRESSIONS

They allow you to specify a *pattern* of text to search for. You may not know a business's exact phone number, but if you live in the United States or Canada, you know it will be three digits, followed by a hyphen, and then four more digits (and optionally, a three-digit area code at the start). This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but 4,155,551,234 is not.

We also recognize all sorts of other text patterns every day: email addresses have @ symbols in the middle, US social security numbers have nine digits and two hyphens, website URLs often have periods and forward slashes, news headlines use title case, social media hashtags begin with # and contain no spaces, and more.

Regular expressions are helpful, but few non-programmers know about them even though most modern text editors and word processors, such as Microsoft Word or OpenOffice, have find and find-and-replace features that can search based on regular expressions. Regular expressions are huge time-savers, not just for software users but also for programmers. In fact, tech writer Cory Doctorow argues that we should be teaching regular expressions even before programming:

Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you're a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.¹

In this chapter, you'll start by writing a program to find text patterns *without* using regular expressions and then see how to use regular expressions to make the code much less bloated. I'll show you basic matching with regular expressions and then move on to some more powerful features, such as string substitution and creating your own character classes. Finally, at the end of the chapter, you'll write a program that can automatically extract phone numbers and email addresses from a block of text.

FINDING PATTERNS OF TEXT WITHOUT REGULAR EXPRESSIONS

Say you want to find an American phone number in a string. You know the pattern if you're American: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.

Let's use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either `True` or `False`. Open a new file editor tab and enter the following code; then save the file as *isPhoneNumber.py*:

```
def isPhoneNumber(text):  
    ❶ if len(text) != 12:  
        return False  
    for i in range(0, 3):  
        ❷ if not text[i].isdecimal():  
            return False  
    ❸ if text[3] != '-':  
        return False  
    for i in range(4, 7):
```

```
    ❷ if not text[i].isdecimal():
        return False
    ❸ if text[7] != '-':
        return False
    for i in range(8, 12):
        ❹ if not text[i].isdecimal():
            return False
    ❺ return True

print('Is 415-555-4242 a phone number?')
print(isPhoneNumber('415-555-4242'))
print('Is Moshi moshi a phone number?')
print(isPhoneNumber('Moshi moshi'))
```

When this program is run, the output looks like this:

```
Is 415-555-4242 a phone number?
True
Is Moshi moshi a phone number?
False
```

The `isPhoneNumber()` function has code that does several checks to see whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`. First the code checks that the string is exactly 12 characters ❶. Then it checks that the area code (that is, the first three characters in `text`) consists of only numeric characters ❷. The rest of the function checks that the string follows the pattern of a phone number: the number must have the first hyphen after the area code ❸, three more numeric characters ❹, then another hyphen ❺, and finally four more numbers ❻. If the program execution manages to get past all the checks, it returns `True` ❼.

Calling `isPhoneNumber()` with the argument `'415-555-4242'` will return `True`. Calling `isPhoneNumber()` with `'Moshi moshi'` will return `False`; the first test fails because `'Moshi moshi'` is not 12 characters long.

If you wanted to find a phone number within a larger string, you would have to add even more code to find the phone number pattern. Replace the last four `print()` function calls in *isPhoneNumber.py* with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
    ❶ chunk = message[i:i+12]
    ❷ if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

On each iteration of the for loop, a new chunk of 12 characters from message is assigned to the variable chunk ❶. For example, on the first iteration, *i* is 0, and chunk is assigned message [0:12] (that is, the string 'Call me at 4'). On the next iteration, *i* is 1, and chunk is assigned message[1:13] (the string 'all me at 41'). In other words, on each iteration of the for loop, chunk takes on the following values:

- 'Call me at 4'
- 'all me at 41'
- 'll me at 415'
- 'l me at 415-'
- ... and so on.

You pass chunk to isPhoneNumber() to see whether it matches the phone number pattern ❷, and if so, you print the chunk.

Continue to loop through message, and eventually the 12 characters in chunk will be a phone number. The loop goes through the entire string, testing each 12-character piece and printing any chunk it finds that satisfies isPhoneNumber(). Once we're done going through message, we print Done.

While the string in message is short in this example, it could be millions of characters long and the program would still run in less than a second. A similar program that finds phone numbers using regular expressions would also run in less than a second, but regular expressions make it quicker to write these programs.

FINDING PATTERNS OF TEXT WITH REGULAR EXPRESSIONS

The previous phone number-finding program works, but it uses a lot of code to do something limited: the isPhoneNumber() function is 17 lines but can find only one pattern of phone numbers. What about a phone number formatted like 415.555.4242 or (415) 555-4242? What if the phone number had an extension, like 415-555-4242 x99?

The isPhoneNumber() function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.

Regular expressions, called *regexes* for short, are descriptions for a pattern of text. For example, a `\d` in a regex stands for a digit character—that is, any single numeral from 0 to 9. The regex `\d\d\d-\d\d\d-\d\d\d\d` is used by Python to match the same text pattern the previous isPhoneNumber() function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `\d\d\d-\d\d\d-\d\d\d\d` regex.

But regular expressions can be much more sophisticated. For example, adding a 3 in braces (`{3}`) after a pattern is like saying, "Match this pattern three times." So the slightly shorter regex `\d{3}-\d{3}-\d{4}` also matches the correct phone number format.

Creating Regex Objects

All the regex functions in Python are in the `re` module. Enter the following into the interactive shell to import this module:

```
>>> import re
```

NOTE

Most of the examples in this chapter will require the re module, so remember to import it at the beginning of any script you write or any time you restart Mu. Otherwise, you'll get a NameError: name 're' is not defined error message.

Passing a string value representing your regular expression to `re.compile()` returns a Regex pattern object (or simply, a Regex object).

To create a Regex object that matches the phone number pattern, enter the following into the interactive shell. (Remember that `\d` means –a digit character and `\d\d\d-\d\d\d-\d\d\d\d` is the regular expression for a phone number pattern.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the `phoneNumRegex` variable contains a Regex object.

Matching Regex Objects

A Regex object's `search()` method searches the string it is passed for any matches to the regex. The `search()` method will return `None` if the regex pattern is not found in the string. If the pattern *is* found, the `search()` method returns a Match object, which have a `group()` method that will return the actual matched text from the searched string. (I'll explain groups shortly.) For example, enter the following into the interactive shell:

```
>>> phoneNumRegex=re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo=phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found:'+mo.group())
Phone number found: 415-555-4242
```

The `mo` variable name is just a generic name to use for Match objects. This example might seem complicated at first, but it is much shorter than the earlier `isPhoneNumber.py` program and does the same thing.

Here, we pass our desired pattern to `re.compile()` and store the resulting Regex object in `phoneNumRegex`. Then we call `search()` on `phoneNumRegex` and pass `search()` the string we want to match for during the search. The result of the search gets stored in the variable `mo`. In this example, we know that our pattern will be found in the string, so we know that a Match object will be returned. Knowing that `mo` contains a Match object and not the null value `None`, we can call `group()` on `mo` to return the match. Writing `mo.group()` inside our `print()` function call displays the whole match, 415-555-4242.

Review of Regular Expression Matching

While there are several steps to use regular expressions in Python, each step is fairly simple.

1. Import the regex module with `import re`.
 2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
 3. Pass the string you want to search into the Regex object's `search()` method. This returns a Match object.
 4. Call the Match object's `group()` method to return a string of the actual matched text.
-

NOTE

While I encourage you to enter the example code into the interactive shell, you should also make use of web-based regular expression testers, which can show you exactly how a regex matches a piece of text that you enter. I recommend the tester at <https://pythex.org/>.

MORE PATTERN MATCHING WITH REGULAR EXPRESSIONS

Now that you know the basic steps for creating and finding regular expression objects using Python, you're ready to try some of their more powerful pattern-matching capabilities.

Grouping with Parentheses

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create *groups* in the regex: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Then you can use the `group()` match object method to grab the matching text from just one group.

The first set of parentheses in a regex string will be group 1. The second set will be group 2. By passing the *integer* 1 or 2 to the `group()` match object method, you can grab different parts of the matched text. Passing 0 or nothing to the `group()` method will return the entire matched text. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `areaCode, mainNumber = mo.groups()` line.

Parentheses have a special meaning in regular expressions, but what do you do if you need to match a parenthesis in your text? For instance, maybe the phone numbers you are trying to match have the area code set in parentheses. In this case, you need to escape the `(` and `)` characters with a backslash. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

The `\(` and `\)` escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters. In regular expressions, the following characters have special meanings:

```
. ^ $ * + ? { } [ ] \ | ( )
```

If you want to detect these characters as part of your text pattern, you need to escape them with a backslash:

```
\. \^ \$ \* \+ \? \{ \} \[ \] \\ \| \(\ \)
```

Make sure to double-check that you haven't mistaken escaped parentheses `\(` and `\)` for parentheses `(` and `)` in a regular expression. If you receive an error message about `-missing)` or `-unbalanced parenthesis`, you may have forgotten to include the closing unescaped parenthesis for a group, like in this example:

```
>>> re.compile(r'\(Parentheses\))
Traceback (most recent call last):
  --snip--
re.error: missing ), unterminated subpattern at position 0
```

The error message tells you that there is an opening parenthesis at index 0 of the `r'\(Parentheses\)` string that is missing its corresponding closing parenthesis.

Matching Multiple Groups with the Pipe

The `|` character is called a *pipe*. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'.

When *both* Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object. Enter the following into the interactive shell:

```
>>> heroRegex = re.compile(r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman')
>>> mo2.group()
'Tina Fey'
```

NOTE

You can find all matching occurrences with the `findall()` method that's discussed in "[The `findall\(\)` Method](#)"

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once. This can be done with parentheses. Enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

The method call `mo.group()` returns the full matched text 'Batmobile', while `mo.group(1)` returns just the part of the matched text inside the first parentheses group, 'mobile'. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like `\|`.

Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match regardless of whether that bit of text is there. The `?` character flags the group that precedes it as an optional part of the pattern. For example, enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

The `(wo)?` part of the regular expression means that the pattern `wo` is an optional group. The regex will match text that has zero instances or one instance of `wo` in it. This is why the regex matches both 'Batwoman' and 'Batman'.

Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code. Enter the following into the interactive shell:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
```

```
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

You can think of the `?` as saying, –Match zero or one of the group preceding this question mark.¶

If you need to match an actual question mark character, escape it with `\?`.

Matching Zero or More with the Star

The `*` (called the *star* or *asterisk*) means –match zero or more¶—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Let’s look at the Batman example again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

For 'Batman', the `(wo)*` part of the regex matches zero instances of `wo` in the string; for 'Batwoman', the `(wo)*` matches one instance of `wo`; and for 'Batwowowowoman', `(wo)*` matches four instances of `wo`.

If you need to match an actual star character, prefix the star in the regular expression with a backslash, `*`.

Matching One or More with the Plus

While `*` means –match zero or more,¶ the `+` (or *plus*) means –match one or more.¶ Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear *at least once*. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
```

```
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

The regex `Bat(wo)+man` will not match the string 'The Adventures of Batman', because at least one `wo` is required by the plus sign.

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: `\+`.

Matching Specific Repetitions with Braces

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in braces. For example, the regex `(Ha){3}` will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the braces. For example, the regex `(Ha){3,5}` will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

You can also leave out the first or second number in the braces to leave the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the (Ha) group, while `(Ha){,5}` will match zero to five instances. Braces can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

And these two regular expressions also match identical patterns:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Enter the following into the interactive shell:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Here, `(Ha){3}` matches 'HaHaHa' but not 'Ha'. Since it doesn't match 'Ha', `search()` returns `None`.

GREEDY AND NON-GREEDY MATCHING

Since `(Ha){3,5}` can match three, four, or five instances of `Ha` in the string `'HaHaHaHaHa'`, you may wonder why the `Match` object's call to `group()` in the previous brace example returns `'HaHaHaHaHa'` instead of the shorter possibilities.

After all, `'HaHaHa'` and `'HaHaHaHa'` are also valid matches of the regular expression `(Ha){3,5}`.

Python's regular expressions are *greedy* by default, which means that in ambiguous situations they will match the longest string possible. The *non-greedy* (also called *lazy*) version of the braces, which matches the shortest string possible, has the closing brace followed by a question mark.

Enter the following into the interactive shell, and notice the difference between the greedy and non-greedy forms of the braces searching the same string:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a non-greedy match or flagging an optional group. These meanings are entirely unrelated.

THE FINDALL() METHOD

In addition to the `search()` method, `Regex` objects also have a `findall()` method. While `search()` will return a `Match` object of the *first* matched text in the searched string, the `findall()` method will return the strings of *every* match in the searched string. To see how `search()` returns a `Match` object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

On the other hand, `findall()` will not return a `Match` object but a list of strings—as long as there are no groups in the regular expression. Each string in the list is a piece of the searched text that matched the regular expression. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then `findall()` will return a list of tuples. Each tuple represents a found match, and its items are the matched strings for each group in the

regex. To see `findall()` in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '9999'), ('212', '555', '0000')]
```

To summarize what the `findall()` method returns, remember the following:

- When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d\d`, the method `findall()` returns a list of string matches, such as `['415-555-9999', '212-555-0000']`.
- When called on a regex that has groups, such as `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, the method `findall()` returns a list of tuples of strings (one string for each group), such as `[('415', '555', '9999'), ('212', '555', '0000')]`.

CHARACTER CLASSES

In the earlier phone number regex example, you learned that `\d` could stand for any numeric digit. That is, `\d` is shorthand for the regular expression `(0|1|2|3|4|5|6|7|8|9)`. There are many such *shorthand character classes*, as shown in [Table 7-1](#).

Table 7-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents
<code>\d</code>	Any numeric digit from 0 to 9.
<code>\D</code>	Any character that is <i>not</i> a numeric digit from 0 to 9.
<code>\w</code>	Any letter, numeric digit, or the underscore character. (Think of this as matching <code>-word\w</code> characters.)
<code>\W</code>	Any character that is <i>not</i> a letter, numeric digit, or the underscore character.
<code>\s</code>	Any space, tab, or newline character. (Think of this as matching <code>-space\w</code> characters.)
<code>\S</code>	Any character that is <i>not</i> a space, tab, or newline.

Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers 0 to 5; this is much shorter than typing `(0|1|2|3|4|5)`. Note that while `\d` matches digits and `\w` matches digits, letters, and the underscore, there is no shorthand character class that matches only letters. (Though you can use the `[a-zA-Z]` character class, as explained next.)

For example, enter the following into the interactive shell:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7 swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6 geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression `\d+\s\w+` will match text that has one or more numeric digits (`\d+`), followed by a whitespace character (`\s`), followed by one or more letter/digit/underscore characters (`\w+`). The `findall()` method returns all matching strings of the regex pattern in a list.

MAKING YOUR OWN CHARACTER CLASSES

There are times when you want to match a set of characters but the shorthand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. Enter the following into the interactive shell:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the `.`, `*`, `?`, or `()` characters with a preceding backslash. For example, the character class `[0-5.]` will match digits 0 to 5 and a period. You do not need to write it as `[0-5\.]`.

By placing a caret character (`^`) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class. For example, enter the following into the interactive shell:

```
>>> consonantRegex = re.compile(r'^[aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'C', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', ' ', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', ' ', '']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

THE CARET AND DOLLAR SIGN CHARACTERS

You can also use the caret symbol (`^`) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text. Likewise, you can put a dollar sign (`$`) at the end of the regex to indicate the string must *end* with this regex pattern. And you can use the `^` and `$` together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example, the `r'^Hello'` regular expression string matches strings that begin with 'Hello'. Enter the following into the interactive shell:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello, world!')
<re.Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9. Enter the following into the interactive shell:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<re.Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<re.Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

The last two `search()` calls in the previous interactive shell example demonstrate how the entire string must match the regex if `^` and `$` are used.

I always confuse the meanings of these two symbols, so I use the mnemonic –Carrots cost dollars| to remind myself that the caret comes first and the dollar sign comes last.

THE WILDCARD CHARACTER

The `.` (or *dot*) character in a regular expression is called a *wildcard* and will match any character except for a newline. For example, enter the following into the interactive shell:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Remember that the dot character will match just one character, which is why the match for the text `flat` in the previous example matched only `lat`. To match an actual dot, escape the dot with a backslash: `\.`

Matching Everything with Dot-Star

Sometimes you will want to match everything and anything. For example, say you want to match the string `'First Name: '`, followed by any and all text, followed by `'Last Name: '`, and then followed by anything again. You can use the dot-star `(.*)` to stand in for that –anything. Remember that the dot character means –any single character except the newline, and the star character means –zero or more of the preceding character.

Enter the following into the interactive shell:

```
>>> nameRegex = re.compile(r'First Name: (.*?) Last Name: (.*?)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
```

```
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

The dot-star uses *greedy* mode: It will always try to match as much text as possible. To match any and all text in a *non-greedy* fashion, use the dot, star, and question mark (`.*`). Like with braces, the question mark tells Python to match in a non-greedy way.

Enter the following into the interactive shell to see the difference between the greedy and non-greedy versions:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Both regexes roughly translate to –Match an opening angle bracket, followed by anything, followed by a closing angle bracket. But the string `'<To serve man> for dinner.>'` has two possible matches for the closing angle bracket. In the non-greedy version of the regex, Python matches the shortest possible string: `'<To serve man>'`. In the greedy version, Python matches the longest possible string: `'<To serve man> for dinner.>'`.

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

Enter the following into the interactive shell:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

The regex `noNewlineRegex`, which did not have `re.DOTALL` passed to the `re.compile()` call that created it, will match everything only up to the first newline character, whereas `newlineRegex`, which *did* have `re.DOTALL` passed to `re.compile()`,

matches everything. This is why the `newlineRegex.search()` call matches the full string, including its newline characters.

REVIEW OF REGEX SYMBOLS

This chapter covered a lot of notation, so here's a quick review of what you learned about basic regular expression syntax:

- The `?` matches zero or one of the preceding group.
- The `*` matches zero or more of the preceding group.
- The `+` matches one or more of the preceding group.
- The `{n}` matches exactly *n* of the preceding group.
- The `{n,}` matches *n* or more of the preceding group.
- The `{,m}` matches 0 to *m* of the preceding group.
- The `{n,m}` matches at least *n* and at most *m* of the preceding group.
- `{n,m}?` or `*?` or `+?` performs a non-greedy match of the preceding group.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively.
- `[abc]` matches any character between the brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the brackets.

CASE-INSENSITIVE MATCHING

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`. Enter the following into the interactive shell:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so
```

```
much?').group()  
'robocop'
```

SUBSTITUTING STRINGS WITH THE SUB() METHOD

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The `sub()` method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The `sub()` method returns a string with the substitutions applied.

For example, enter the following into the interactive shell:

```
>>> namesRegex = re.compile(r'Agent \w+')  
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')  
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can type `\1`, `\2`, `\3`, and so on, to mean –Enter the text of group 1, 2, 3, and so on, in the substitution.

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1****'` as the first argument to `sub()`. The `\1` in that string will be replaced by whatever text was matched by group 1—that is, the `(\w)` group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')  
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')  
A**** told C**** that E**** knew B**** was a double agent.'
```

MANAGING COMPLEX REGEXES

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. This –verbose model can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3})|(\d{3}))?(\s|-|.)?\d{3}(\s|-|.)\d{4}  
(\s*(ext|x|ext.)\s*\d{2,5})?')
```

you can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''(  
  (\d{3})|(\d{3}))?      # area code  
  (\s|-|.)?             # separator  
  \d{3}                 # first 3 digits
```

```
(\s|-\|\. )      # separator
\d{4}            # last 4 digits
(\s*(ext|x|ext.)\s*\d{2,5})? # extension
)", re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (") to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as regular Python code: the # symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so it's easier to read.

COMBINING re.IGNORECASE, re.DOTALL, AND re.VERBOSE

What if you want to use re.VERBOSE to write comments in your regular expression but also want to use re.IGNORECASE to ignore capitalization? Unfortunately, the re.compile() function takes only a single value as its second argument. You can get around this limitation by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the *bitwise or* operator.

So if you want a regular expression that's case-insensitive *and* includes newlines to match the dot character, you would form your re.compile() call like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

Including all three options in the second argument will look like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL |
re.VERBOSE)
```

This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at <https://nostarch.com/automatestuff2/> for more information. You can also pass other options for the second argument; they're uncommon, but you can read more about them in the resources, too.

PROJECT: PHONE NUMBER AND EMAIL ADDRESS EXTRACTOR

Say you have the boring task of finding every phone number and email address in a long web page or document. If you manually scroll through the page, you might end up searching for a long time. But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press CTRL-A to select all the text, press CTRL-C to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

Whenever you're tackling a new project, it can be tempting to dive right into writing code. But more often than not, it's best to take a step back and consider the bigger picture. I recommend first drawing up a high-level plan for what your program needs to do. Don't think about the actual code yet—you can worry about that later. Right now, stick to broad strokes.

For example, your phone and email address extractor will need to do the following:

1. Get the text off the clipboard.
2. Find all phone numbers and email addresses in the text.
3. Paste them onto the clipboard.

Now you can start thinking about how this might work in code. The code will need to do the following:

1. Use the pyperclip module to copy and paste strings.
2. Create two regexes, one for matching phone numbers and the other for matching email addresses.
3. Find all matches, not just the first match, of both regexes.
4. Neatly format the matched strings into a single string to paste.
5. Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately. Each step is fairly manageable and expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

First, you have to create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as *phoneAndEmail.py*:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'(\d{3}|\(\d{3}\))?      # area code
(s|-\|.)?      # separator
(\d{3})      # first 3 digits
(s|-\|.)?      # separator
(\d{4})      # last 4 digits
(s*(ext|x|ext.)s*(\d{2,5}))? # extension
)', re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The TODO comments are just a skeleton for the program. They'll be replaced as you write the actual code.

The phone number begins with an *optional* area code, so the area code group is followed with a question mark. Since the area code can be just three digits (that is, `\d{3}`) *or* three digits within parentheses (that is, `\(\d{3}\)`), you should have a pipe joining those parts. You can add the regex comment `# Area code` to this part of the multiline string to help you remember what `(\d{3}|\(\d{3}\))?` is supposed to match.

The phone number separator character can be a space (\s), hyphen (-), or period (.), so these parts should also be joined by pipes. The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an optional extension made up of any number of spaces followed by ext, x, or ext., followed by two to five digits.

NOTE

It's easy to get mixed up with regular expressions that contain groups with parentheses () and escaped parentheses \(\). Remember to double-check that you're using the correct one if you get a "missing), unterminated subpattern" error message.

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''
--snip--

# Create email regex.
emailRegex = re.compile(r'''
    ❶ [a-zA-Z0-9._%+-]+ # username
    ❷ @ # @ symbol
    ❸ [a-zA-Z0-9.-]+ # domain name
    (\.[a-zA-Z]{2,4}) # dot-something
''', re.VERBOSE)

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The username part of the email address ❶ is one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen. You can put all of these into a character class: [a-zA-Z0-9._%+-].

The domain and username are separated by an @ symbol ❷. The domain name ❸ has a slightly less permissive character class with only letters, numbers, periods, and hyphens: [a-zA-Z0-9.-]. And last will be the -dot-com part (technically known as the *top-level domain*), which can really be dot-anything. This is between two and four characters.

The format for email addresses has a lot of weird rules. This regular expression won't match every possible valid email address, but it'll match almost any typical email address you'll encounter.

Step 3: Find All Matches in the Clipboard Text

Now that you have specified the regular expressions for phone numbers and email addresses, you can let Python's re module do the hard work of finding all the matches on the clipboard.

The `pyperclip.paste()` function will get a string value of the text on the clipboard, and the `findall()` regex method will return a list of tuples.

Make your program look like the following:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''
--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())

❶ matches = []
❷ for groups in phoneRegex.findall(text):
    phoneNum = '-'.join([groups[1], groups[3], groups[5]])
    if groups[8] != '':
        phoneNum += ' x' + groups[8]
    matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
    matches.append(groups[0])

# TODO: Copy results to the clipboard.
```

There is one tuple for each match, and each tuple contains strings for each group in the regular expression. Remember that group 0 matches the entire regular expression, so the group at index 0 of the tuple is the one you are interested in.

As you can see at ❶, you'll store the matches in a list variable named `matches`. It starts off as an empty list, and a couple for loops. For the email addresses, you append group 0 of each match ❸. For the matched phone numbers, you don't want to just append group 0. While the program *detects* phone numbers in several formats, you want the phone number appended to be in a single, standard format. The `phoneNum` variable contains a string built from groups 1, 3, 5, and 8 of the matched text ❷. (These groups are the area code, first three digits, last four digits, and extension.)

Step 4: Join the Matches into a String for the Clipboard

Now that you have the email addresses and phone numbers as a list of strings in `matches`, you want to put them on the clipboard. The `pyperclip.copy()` function takes only a single string value, not a list of strings, so you call the `join()` method on `matches`.

To make it easier to see that the program is working, let's print any matches you find to the terminal. If no phone numbers or email addresses were found, the program should tell the user this.

Make your program look like the following:

```
#!/python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

--snip--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

Running the Program

For an example, open your web browser to the No Starch Press contact page at <https://nostarch.com/contactus/>, press CTRL-A to select all the text on the page, and press CTRL-C to copy it to the clipboard. When you run this program, the output will look something like this:

```
Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
info@nostarch.com
```

Ideas for Similar Programs

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications. For example, you could:

- Find website URLs that begin with *http://* or *https://*.
- Clean up dates in different date formats (such as 3/14/2019, 03-14-2019, and 2015/3/19) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security or credit card numbers.
- Find common typos such as multiple spaces between words, accidentally accidentally repeated words, or multiple exclamation marks at the end of sentences. Those are annoying!!

SUMMARY

While a computer can search for text quickly, it must be told precisely what to look for. Regular expressions allow you to specify the pattern of characters you are looking for, rather than the exact text itself. In fact, some word processing and spreadsheet applications provide find-and-replace features that allow you to search using regular expressions.

The `re` module that comes with Python lets you compile Regex objects. These objects have several methods: `search()` to find a single match, `findall()` to find all matching instances, and `sub()` to do a find-and-replace substitution of text.

You can find out more in the official Python documentation at <https://docs.python.org/3/library/re.html>. Another useful resource is the tutorial website <https://www.regular-expressions.info/>.

PRACTICE QUESTIONS

1. What is the function that creates Regex objects?
2. Why are raw strings often used when creating Regex objects?
3. What does the `search()` method return?
4. How do you get the actual strings that match the pattern from a Match object?
5. In the regex created from `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`, what does group 0 cover? Group 1? Group 2?
6. Parentheses and periods have specific meanings in regular expression syntax. How would you specify that you want a regex to match actual parentheses and period characters?
7. The `findall()` method returns a list of strings or a list of tuples of strings. What makes it return one or the other?
8. What does the `|` character signify in regular expressions?
9. What two things does the `?` character signify in regular expressions?
10. What is the difference between the `+` and `*` characters in regular expressions?
11. What is the difference between `{3}` and `{3,5}` in regular expressions?
12. What do the `\d`, `\w`, and `\s` shorthand character classes signify in regular expressions?
13. What do the `\D`, `\W`, and `\S` shorthand character classes signify in regular expressions?
14. What is the difference between `.*` and `.*??`?
15. What is the character class syntax to match all numbers and lowercase letters?
16. How do you make a regular expression case-insensitive?
17. What does the `.` character normally match? What does it match if `re.DOTALL` is passed as the second argument to `re.compile()`?
18. If `numRegex = re.compile(r'\d+')`, what will `numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')` return?
19. What does passing `re.VERBOSE` as the second argument to `re.compile()` allow you to do?
20. How would you write a regex that matches a number with commas for every three digits? It must match the following:
 - '42'
 - '1,234'
 - '6,368,745'

but not the following:

- '12,34,567' (which has only two digits between the commas)
- '1234' (which lacks commas)

21. How would you write a regex that matches the full name of someone whose last name is Watanabe? You can assume that the first name that comes before it will always be one word that begins with a capital letter. The regex must match the following:

- 'Haruto Watanabe'
- 'Alice Watanabe'
- 'RoboCop Watanabe'

but not the following:

- 'haruto Watanabe' (where the first name is not capitalized)
- 'Mr. Watanabe' (where the preceding word has a nonletter character)
- 'Watanabe' (which has no first name)
- 'Haruto watanabe' (where Watanabe is not capitalized)

22. How would you write a regex that matches a sentence where the first word is either *Alice*, *Bob*, or *Carol*; the second word is either *eats*, *pets*, or *throws*; the third word is *apples*, *cats*, or *baseballs*; and the sentence ends with a period? This regex should be case-insensitive. It must match the following:

- 'Alice eats apples.'
- 'Bob pets cats.'
- 'Carol throws baseballs.'
- 'Alice throws Apples.'
- 'BOB EATS CATS.'

but not the following:

- 'RoboCop eats apples.'
- 'ALICE THROWS FOOTBALLS.'
- 'Carol eats 7 cats.'

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Date Detection

Write a regular expression that can detect dates in the *DD/MM/YYYY* format. Assume that the days range from 01 to 31, the months range from 01 to 12, and the years range from 1000 to 2999. Note that if the day or month is a single digit, it'll have a leading zero.

The regular expression doesn't have to detect correct days for each month or for leap years; it will accept nonexistent dates like 31/02/2020 or 31/04/2021. Then store these strings into variables named *month*, *day*, and *year*, and write additional code that can detect if it is a valid date. April, June, September, and November have 30 days, February has 28 days, and the rest of the months have 31 days. February has 29 days in leap years. Leap years are every year evenly divisible by 4, except for years evenly divisible by 100, unless the year is also evenly divisible by 400. Note how this calculation makes it impossible to make a reasonably sized regular expression that can detect a valid date.

Strong Password Detection

Write a function that uses regular expressions to make sure the password string it is passed is strong. A strong password is defined as one that is at least eight characters long, contains both uppercase and lowercase characters, and has at least one digit. You may need to test the string against multiple regex patterns to validate its strength.

Regex Version of the strip() Method

Write a function that takes a string and does the same thing as the strip() string method. If no other arguments are passed other than the string to strip, then whitespace characters will be removed from the beginning and end of the string. Otherwise, the characters specified in the second argument to the function will be removed from the string.

READING AND WRITING FILES

READING AND WRITING FILES

Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file. You can think of a file's contents as a single string value, potentially gigabytes in size. In this chapter, you will learn how to use Python to create, read, and save files on the hard drive.

FILES AND FILE PATHS

A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer. For example, there is a file on my Windows 7 laptop with the filename *project.docx* in the path *C:\Users\asweigart\Documents*. The part of the filename after the last period is called the file's *extension* and tells you a file's

type. *project.docx* is a Word document, and *Users*, *asweigart*, and *Documents* all refer to *folders* (also called *directories*). Folders can contain files and other folders. For example, *project.docx* is in the *Documents* folder, which is inside the *asweigart* folder, which is inside the *Users* folder. [Figure 8-1](#) shows this folder organization.

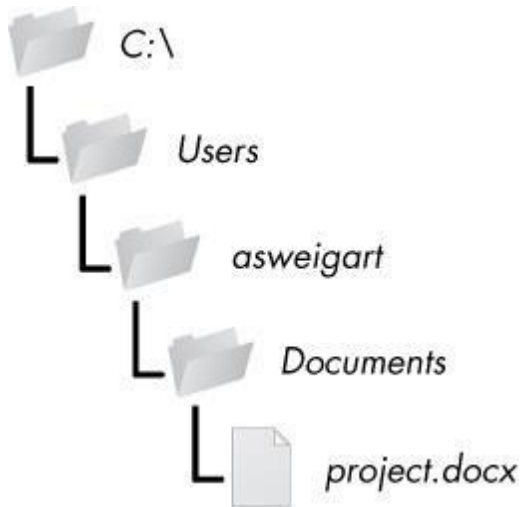


Figure 8-1. A file in a hierarchy of folders

The *C:* part of the path is the *root folder*, which contains all other folders. On Windows, the root folder is named *C:* and is also called the *C: drive*. On OS X and Linux, the root folder is */*. In this book, I'll be using the Windows-style root folder, *C:*. If you are entering the interactive shell examples on OS X or Linux, enter */* instead.

Additional *volumes*, such as a DVD drive or USB thumb drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as *D:* or *E:*. On OS X, they appear as new folders under the */Volumes* folder. On Linux, they appear as new folders under the */mnt* (–mount) folder. Also note that while folder names and filenames are not case sensitive on Windows and OS X, they are case sensitive on Linux.

BACKSLASH ON WINDOWS AND FORWARD SLASH ON OS X AND LINUX

On Windows, paths are written using backslashes (**) as the separator between folder names. OS X and Linux, however, use the forward slash (*/*) as their path separator. If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.

Fortunately, this is simple to do with the `os.path.join()` function. If you pass it the string values of individual file and folder names in your path, `os.path.join()` will return a string with a file path using the correct path separators. Enter the following into the interactive shell:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

I'm running these interactive shell examples on Windows, so `os.path.join('usr', 'bin', 'spam')` returned `'usr\\bin\\spam'`. (Notice that the backslashes are doubled because each backslash needs to be escaped by another backslash character.) If I had called this function on OS X or Linux, the string would have been `'usr/bin/spam'`.

The `os.path.join()` function is helpful if you need to create strings for filenames. These strings will be passed to several of the file-related functions introduced in this chapter. For example, the following example joins names from a list of filenames to the end of a folder's name:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
    print(os.path.join('C:\\Users\\asweigart', filename))
C:\\Users\\asweigart\\accounts.txt
C:\\Users\\asweigart\\details.csv
C:\\Users\\asweigart\\invite.docx
```

THE CURRENT WORKING DIRECTORY

Every program that runs on your computer has a *current working directory*, or *cwd*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. You can get the current working directory as a string value with the `os.getcwd()` function and change it with `os.chdir()`. Enter the following into the interactive shell:

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Here, the current working directory is set to `C:\\Python34`, so the filename `project.docx` refers to `C:\\Python34\\project.docx`. When we change the current working directory to `C:\\Windows`, `project.docx` is interpreted as `C:\\Windows\\project.docx`.

Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

NOTE

While *folder* is the more modern name for *directory*, note that *current working directory* (or *just working directory*) is the standard term, not *current working folder*.

ABSOLUTE VS. RELATIVE PATHS

There are two ways to specify a file path.

- An *absolute path*, which always begins with the root folder
- A *relative path*, which is relative to the program's current working directory

There are also the *dot* (.) and *dot-dot* (..) folders. These are not real folders but special names that can be used in a path. A single period (–dot) for a folder name is shorthand for –this directory. Two periods (–dot-dot) means –the parent folder.

Figure 8-2 is an example of some folders and files. When the current working directory is set to `C:\bacon`, the relative paths for the other folders and files are set as they are in the figure.

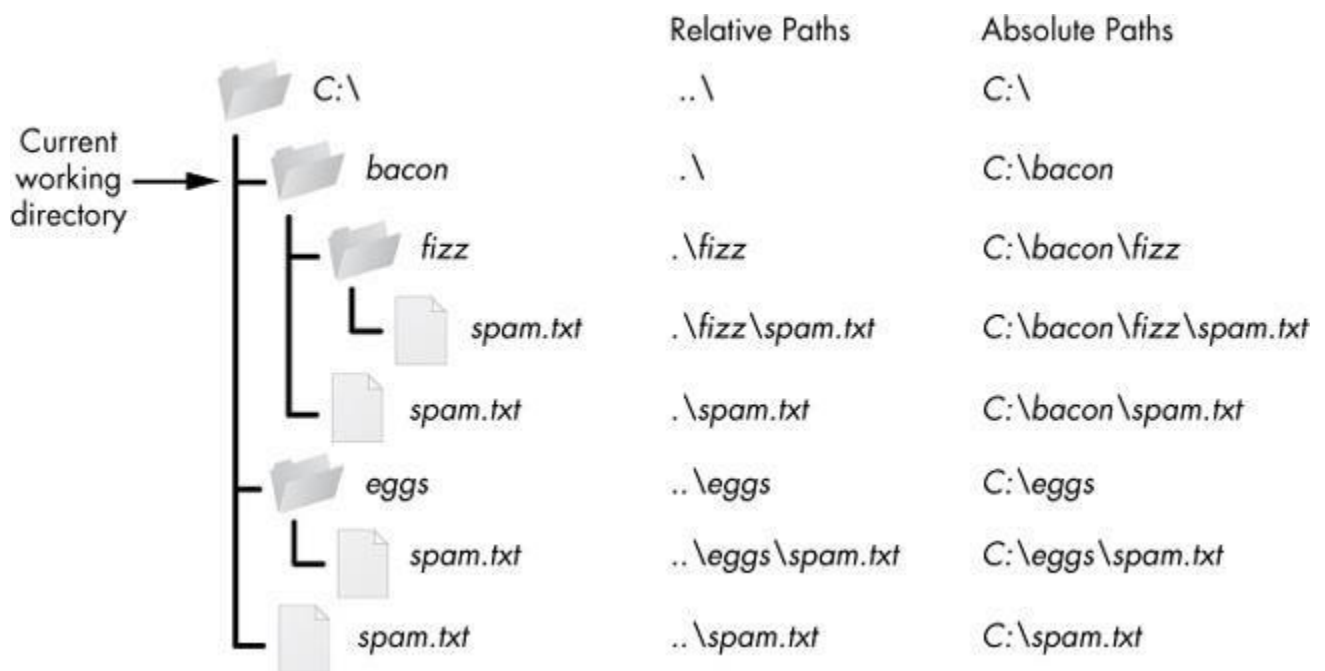


Figure 8-2. The relative paths for folders and files in the working directory `C:\bacon`

The `.\` at the start of a relative path is optional. For example, `.\spam.txt` and `spam.txt` refer to the same file.

CREATING NEW FOLDERS WITH `OS.MAKEDIRS()`

Your programs can create new folders (directories) with the `os.makedirs()` function. Enter the following into the interactive shell:

```
>>> import os
```

```
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

This will create not just the `C:\delicious` folder but also a `walnut` folder inside `C:\delicious` and a `waffles` folder inside `C:\delicious\walnut`. That is, `os.makedirs()` will create any necessary intermediate folders in order to ensure that the full path exists. [Figure 8-3](#) shows this hierarchy of folders.

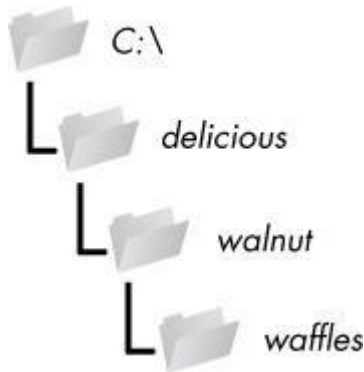


Figure 8-3. The result of `os.makedirs('C:\\delicious \\walnut\\waffles')`

THE OS.PATH MODULE

The `os.path` module contains many helpful functions related to filenames and file paths. For instance, you've already used `os.path.join()` to build paths in a way that will work on any operating system. Since `os.path` is a module inside the `os` module, you can import it by simply running `import os`. Whenever your programs need to work with files, folders, or file paths, you can refer to the short examples in this section. The full documentation for the `os.path` module is on the Python website at <http://docs.python.org/3/library/os.path.html>.

NOTE

Most of the examples that follow in this section will require the `os` module, so remember to import it at the beginning of any script you write and any time you restart IDLE. Otherwise, you'll get a `NameError: name 'os' is not defined` error message.

HANDLING ABSOLUTE AND RELATIVE PATHS

The `os.path` module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

- Calling `os.path.abspath(path)` will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
- Calling `os.path.isabs(path)` will return `True` if the argument is an absolute path and `False` if it is a relative path.

- Calling `os.path.relpath(path, start)` will return a string of a relative path from the *start* path to *path*. If *start* is not provided, the current working directory is used as the start path.

Try these functions in the interactive shell:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('..\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Since `C:\\Python34` was the working directory when `os.path.abspath()` was called, the `..` folder represents the absolute path `'C:\\Python34'`.

NOTE

Since your system probably has different files and folders on it than mine, you won't be able to follow every example in this chapter exactly. Still, try to follow along using folders that exist on your computer.

Enter the following calls to `os.path.relpath()` into the interactive shell:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd() 'C:\\Python34'
```

Calling `os.path.dirname(path)` will return a string of everything that comes before the last slash in the path argument. Calling `os.path.basename(path)` will return a string of everything that comes after the last slash in the path argument. The dir name and base name of a path are outlined in [Figure 8-4](#).

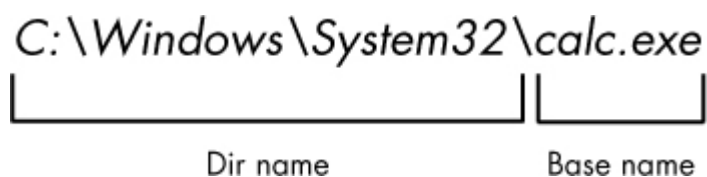


Figure 8-4. The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

For example, enter the following into the interactive shell:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

If you need a path's dir name and base name together, you can just call `os.path.split()` to get a tuple value with these two strings, like so:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Notice that you could create the same tuple by calling `os.path.dirname()` and `os.path.basename()` and placing their return values in a tuple.

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

But `os.path.split()` is a nice shortcut if you need both values.

Also, note that `os.path.split()` does *not* take a file path and return a list of strings of each folder. For that, use the `split()` string method and `split` on the string in `os.sep`. Recall from earlier that the `os.sep` variable is set to the correct folder-separating slash for the computer running the program.

For example, enter the following into the interactive shell:

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

On OS X and Linux systems, there will be a blank string at the start of the returned list:

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

The `split()` string method will work to return a list of each part of the path. It will work on any operating system if you pass it `os.path.sep`.

FINDING FILE SIZES AND FOLDER CONTENTS

Once you have ways of handling file paths, you can then start gathering information about specific files and folders. The `os.path` module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling `os.path.getsize(path)` will return the size in bytes of the file in the *path* argument.
- Calling `os.listdir(path)` will return a list of filename strings for each file in the *path* argument. (Note that this function is in the `os` module, not `os.path`.)

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

As you can see, the *calc.exe* program on my computer is 776,192 bytes in size, and I have a lot of files in *C:\\Windows\\system32*. If I want to find the total size of all the files in this directory, I can use `os.path.getsize()` and `os.listdir()` together.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
    totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32',
filename))

>>> print(totalSize)
1117846456
```

As I loop over each filename in the *C:\\Windows\\System32* folder, the `totalSize` variable is incremented by the size of each file. Notice how when I call `os.path.getsize()`, I use `os.path.join()` to join the folder name with the current filename. The integer that `os.path.getsize()` returns is added to the value of `totalSize`. After looping through all the files, I print `totalSize` to see the total size of the *C:\\Windows\\System32* folder.

CHECKING PATH VALIDITY

Many Python functions will crash with an error if you supply them with a path that does not exist. The `os.path` module provides functions to check whether a given path exists and whether it is a file or folder.

- Calling `os.path.exists(path)` will return `True` if the file or folder referred to in the argument exists and will return `False` if it does not exist.

- Calling `os.path.isfile(path)` will return `True` if the `path` argument exists and is a file and will return `False` otherwise.
- Calling `os.path.isdir(path)` will return `True` if the `path` argument exists and is a folder and will return `False` otherwise.

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `os.path.exists()` function. For instance, if I wanted to check for a flash drive with the volume named `D:` on my Windows computer, I could do that with the following:

```
>>> os.path.exists('D:\\')
False
```

Oops! It looks like I forgot to plug in my flash drive.

THE FILE READING/WRITING PROCESS

Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write. The functions covered in the next few sections will apply to plaintext files. *Plaintext files* contain only basic text characters and do not include font, size, or color information. Text files with the `.txt` extension or Python script files with the `.py` extension are examples of plaintext files. These can be opened with Windows's Notepad or OS X's TextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like in [Figure 8-5](#).

Both these commands will open the file in `-reading plaintext` mode, or *read mode* for short. When a file is opened in read mode, Python lets you only read data from the file; you can't write or modify it in any way. Read mode is the default mode for files you open in Python. But if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value `'r'` as a second argument to `open()`. So `open('/Users/asweigart/hello.txt', 'r')` and `open('/Users/asweigart/hello.txt')` do the same thing.

The call to `open()` returns a File object. A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with. In the previous example, you stored the File object in the variable `helloFile`. Now, whenever you want to read from or write to the file, you can do so by calling methods on the File object in `helloFile`.

READING THE CONTENTS OF FILES

Now that you have a File object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the File object's `read()` method. Let's continue with the *hello.txt* File object you stored in `helloFile`. Enter the following into the interactive shell:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

If you think of the contents of a file as a single large string value, the `read()` method returns the string that is stored in the file.

Alternatively, you can use the `readlines()` method to get a *list* of string values from the file, one string for each line of text. For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and write the following text in it:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Make sure to separate the four lines with line breaks. Then enter the following into the interactive shell:

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', 'I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

Note that each of the string values ends with a newline character, `\n`, except for the last line of the file. A list of strings is often easier to work with than a single large string value.

WRITING TO FILES

Python allows you to write content to a file in a way similar to how the `print()` function –writes strings to the screen. You can't write to a file you've opened in read mode, though. Instead, you need to open it in –write plaintext mode or –append plaintext mode, or *write mode* and *append mode* for short.

Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value. Pass 'w' as the second argument to `open()` to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this as appending to a list in a variable, rather than overwriting the variable altogether. Pass 'a' as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

Let's put these concepts together. Enter the following into the interactive shell:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

First, we open *bacon.txt* in write mode. Since there isn't a *bacon.txt* yet, Python creates one. Calling `write()` on the opened file and passing `write()` the string argument 'Hello world!\n' writes the string to the file and returns the number of characters written, including the newline. Then we close the file.

To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write 'Bacon is not a vegetable.' to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call `read()`, store the resulting File object in `content`, close the file, and print `content`.

Note that the `write()` method does not automatically add a newline character to the end of the string like the `print()` function does. You will have to add this character yourself.

SAVING VARIABLES WITH THE SHELVE MODULE

You can save variables in your Python programs to binary shelf files using the `shelve` module. This way, your program can restore data to variables from the hard drive. The `shelve` module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Enter the following into the interactive shell:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Call `shelve.open()` and pass it a filename, and then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelfFile`. We create a list `cats` and write `shelfFile['cats'] = cats` to store the list in `shelfFile` as a value associated with the key 'cats' (like in a dictionary). Then we call `close()` on `shelfFile`.

After running the previous code on Windows, you will see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On OS X, only a single *mydata.db* file will be created.

These binary files contain the data you stored in your shelf. The format of these binary files is not important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode—they can do both once opened. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
```

```
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering `shelfFile['cats']` returns the same list that we stored earlier, so we know that the list is correctly stored, and we call `close()`.

Just like dictionaries, shelf values have `keys()` and `values()` methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the `list()` function to get them in list form. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the `shelve` module.

SAVING VARIABLES WITH THE `PPRINT.PFORMAT()` FUNCTION

Recall from [Pretty Printing](#) that the `pprint.pprint()` function will -pretty print the contents of a list or dictionary to the screen, while the `pprint.pformat()` function will return this same text as a string instead of printing it. Not only is this string formatted to be easy to read, but it is also syntactically correct Python code. Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use. Using `pprint.pformat()` will give you a string that you can write to `.py` file. This file will be your very own module that you can import whenever you want to use the variable stored in it.

For example, enter the following into the interactive shell:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
```

```
>>> fileObj.close()
```

Here, we import pprint to let us use pprint.pformat(). We have a list of dictionaries, stored in a variable cats. To keep the list in cats available even after we close the shell, we use pprint.pformat() to return it as a string. Once we have the data in cats as a string, it's easy to write the string to a file, which we'll call *myCats.py*.

The modules that an import statement imports are themselves just Python scripts. When the string from pprint.pformat() is saved to a .py file, the file is a module that can be imported just like any other.

And since Python scripts are themselves just text files with the .py file extension, your Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a .py file (as opposed to saving variables with the shelve module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the shelve module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.

PROJECT: GENERATING RANDOM QUIZ FILES

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

- Creates 35 different quizzes.
- Creates 50 multiple-choice questions for each quiz, in random order.
- Provides the correct answer and three random wrong answers for each question, in random order.

- Writes the quizzes to 35 text files.
- Writes the answer keys to 35 text files.

This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call `open()`, `write()`, and `close()` for the quiz and answer key text files.
- Use `random.shuffle()` to randomize the order of the questions and multiple-choice options.

STEP 1: STORE THE QUIZ DATA IN A DICTIONARY

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
#!/ python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

❶ import random

# The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
❸ for quizNum in range(35):
    # TODO: Create the quiz and answer key files.
```


TODO: Write out the header for the quiz.

TODO: Shuffle the order of the states.

TODO: Loop through all 50 states, making a question for each.

Since this program will be randomly ordering the questions and answers, you'll need to import the random module **❶** to make use of its functions. The capitals variable **❷** contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with TODO comments for now) will go inside a for loop that loops 35 times **❸**. (This number can be changed to generate any number of quiz files.)

STEP 2: CREATE THE QUIZ FILE AND SHUFFLE THE QUESTION ORDER

Now it's time to start filling in those TODOs.

The code in the loop will be repeated 35 times—once for each quiz—so you have to worry about only one quiz at a time within the loop. First you'll create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period. Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Add the following lines of code to *randomQuizGenerator.py*:

```
#!/ python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
    ❶ quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
    ❷ answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Write out the header for the quiz.
    ❸ quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
    quizFile.write('\n\n')

    # Shuffle the order of the states.
```



```
states = list(capitals.keys())
```

```
④ random.shuffle(states)
```

```
# TODO: Loop through all 50 states, making a question for each.
```

The filenames for the quizzes will be *capitalsquiz<N>.txt*, where *<N>* is a unique number for the quiz that comes from *quizNum*, the for loop's counter. The answer key for *capitalsquiz<N>.txt* will be stored in a text file named *capitalsquiz_answers<N>.txt*. Each time through the loop, the *%s* placeholder in 'capitalsquiz%s.txt' and 'capitalsquiz_answers%s.txt' will be replaced by (*quizNum* + 1), so the first quiz and answer key created will be *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*. These files will be created with calls to the *open()* function at ① and ②, with 'w' as the second argument to open them in write mode.

The *write()* statements at ③ create a quiz header for the student to fill out. Finally, a randomized list of US states is created with the help of the *random.shuffle()* function ④, which randomly reorders the values in any list that is passed to it.

STEP 3: CREATE THE ANSWER OPTIONS

Now you need to generate the answer options for each question, which will be multiple choice from A to D. You'll need to create another for loop—this one to generate the content for each of the 50 questions on the quiz. Then there will be a third for loop nested inside to generate the multiple-choice options for each question. Make your code look like the following:

```
#!/ python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
```

```
--snip--
```

```
# Loop through all 50 states, making a question for each.
for questionNum in range(50):
```

```
    # Get right and wrong answers.
```

```
    ① correctAnswer = capitals[states[questionNum]]
    ② wrongAnswers = list(capitals.values())
    ③ del wrongAnswers[wrongAnswers.index(correctAnswer)]
    ④ wrongAnswers = random.sample(wrongAnswers, 3)
    ⑤ answerOptions = wrongAnswers + [correctAnswer]
    ⑥ random.shuffle(answerOptions)
```

```
    # TODO: Write the question and answer options to the quiz file.
```

TODO: Write the answer key to a file.

The correct answer is easy to get—it's stored as a value in the capitals dictionary ❶. This loop will loop through the states in the shuffled states list, from states[0] to states[49], find each state in capitals, and store that state's corresponding capital in correctAnswer.

The list of possible wrong answers is trickier. You can get it by duplicating *all* the values in the capitals dictionary ❷, deleting the correct answer ❸, and selecting three random values from this list ❹. The random.sample() function makes it easy to do this selection. Its first argument is the list you want to select from; the second argument is the number of values you want to select. The full list of answer options is the combination of these three wrong answers with the correct answers ❺. Finally, the answers need to be randomized ❻ so that the correct response isn't always choice D.

STEP 4: WRITE CONTENT TO THE QUIZ AND ANSWER KEY FILES

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
#!/ python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--
# Loop through all 50 states, making a question for each.
for questionNum in range(50):
    --snip--

    # Write the question and the answer options to the quiz file.
    quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
        states[questionNum]))
    ❶ for i in range(4):
    ❷     quizFile.write(' %s. %s\n' % ('ABCD'[i], answerOptions[i]))
    quizFile.write('\n')

    # Write the answer key to a file.
    ❸ answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[
        answerOptions.index(correctAnswer)]))
    quizFile.close()
    answerKeyFile.close()
```

A for loop that goes through integers 0 to 3 will write the answer options in the answerOptions list ❶. The expression 'ABCD'[i] at ❷ treats the string 'ABCD' as an

array and will evaluate to 'A','B', 'C', and then 'D' on each respective iteration through the loop.

In the final line ③, the expression `answerOptions.index(correctAnswer)` will find the integer index of the correct answer in the randomly ordered answer options, and `'ABCD'[answerOptions.index(correctAnswer)]` will evaluate to the correct answer's letter to be written to the answer key file.

After you run the program, this is how your *capitalsquiz1.txt* file will look, though of course your questions and answer options may be different from those shown here, depending on the outcome of your `random.shuffle()` calls:

Name:

Date:

Period:

State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?

- A. Hartford
- B. Santa Fe
- C. Harrisburg
- D. Charleston

2. What is the capital of Colorado?

- A. Raleigh
- B. Harrisburg
- C. Denver
- D. Lincoln

--snip--

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

1. D

2. C

3. A

4. C

--snip--

PROJECT: MULTICLIPBOARD

Say you have the boring task of filling out many forms in a web page or software with several text fields. The clipboard saves you from typing the same text over and over again. But only one thing can be on the clipboard at a time. If you have several different pieces of text that you need to copy and paste, you have to keep highlighting and copying the same few things over and over again.

You can write a Python program to keep track of multiple pieces of text. This `-multiclipboard` will be named `mcb.pyw` (since `-mcb` is shorter to type than `-multiclipboard`). The `.pyw` extension means that Python won't show a Terminal window when it runs this program. (See Appendix B for more details.)

The program will save each piece of clipboard text under a keyword. For example, when you run `py mcb.pyw save spam`, the current contents of the clipboard will be saved with the keyword *spam*. This text can later be loaded to the clipboard again by running `py mcb.pyw spam`. And if the user forgets what keywords they have, they can run `py mcb.pyw list` to copy a list of all keywords to the clipboard.

Here's what the program does:

- The command line argument for the keyword is checked.
- If the argument is `save`, then the clipboard contents are saved to the keyword.
- If the argument is `list`, then all the keywords are copied to the clipboard.
- Otherwise, the text for the keyword is copied to the clipboard.

This means the code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Read and write to the clipboard.
- Save and load to a shelf file.

If you use Windows, you can easily run this script from the Run... window by creating a batch file named `mcb.bat` with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

STEP 1: COMMENTS AND SHELF SETUP

Let's start by making a skeleton script with some comments and basic setup. Make your code look like the following:

```
#!/python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
```

```
# py.exe mcb.pyw <keyword> - Loads keyword to clipboard.  
# py.exe mcb.pyw list - Loads all keywords to clipboard.
```

```
❷ import shelve, pyperclip, sys
```

```
❸ mcbShelf = shelve.open('mcb')
```

```
# TODO: Save clipboard content.
```

```
# TODO: List keywords and load content.
```

```
mcbShelf.close()
```

It's common practice to put general usage information in comments at the top of the file ❶. If you ever forget how to run your script, you can always look at these comments for a reminder. Then you import your modules ❷. Copying and pasting will require the pyperclip module, and reading the command line arguments will require the sys module. The shelve module will also come in handy: Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file. Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into your program. The shelf file will be named with the prefix *mcb* ❸.

STEP 2: SAVE CLIPBOARD CONTENT WITH A KEYWORD

The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case. Make your code look like the following:

```
#!/python3  
# mcb.pyw - Saves and loads pieces of text to the clipboard.  
--snip--
```

```
# Save clipboard content.
```

```
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
```

```
❷     mcbShelf[sys.argv[2]] = pyperclip.paste()
```

```
    elif len(sys.argv) == 2:
```

```
❸     # TODO: List keywords and load content.
```

```
mcbShelf.close()
```

If the first command line argument (which will always be at index 1 of the sys.argv list) is 'save' ❶, the second command line argument is the keyword for the current content of the clipboard. The keyword will be used as the key for mcbShelf, and the value will be the text currently on the clipboard ❷.

If there is only one command line argument, you will assume it is either 'list' or a keyword to load content onto the clipboard. You will implement that code later. For now, just put a TODO comment there ❸.

STEP 3: LIST KEYWORDS AND LOAD A KEYWORD'S CONTENT

Finally, let's implement the two remaining cases: The user wants to load clipboard text in from a keyword, or they want a list of all available keywords. Make your code look like the following:

```
#!/ python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # List keywords and load content.
    ❶ if sys.argv[1].lower() == 'list':
    ❷     pyperclip.copy(str(list(mcbShelf.keys())))
    elif sys.argv[1] in mcbShelf:
    ❸     pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

If there is only one command line argument, first let's check whether it's 'list' ❶. If so, a string representation of the list of shelf keys will be copied to the clipboard ❷. The user can paste this list into an open text editor to read it.

Otherwise, you can assume the command line argument is a keyword. If this keyword exists in the mcbShelf shelf as a key, you can load the value onto the clipboard ❸.

And that's it! Launching this program has different steps depending on what operating system your computer uses. See Appendix B for details for your operating system.

Recall the password locker program you created in [Chapter 6](#) that stored the passwords in a dictionary. Updating the passwords required changing the source code of the program. This isn't ideal because average users don't feel comfortable changing source code to update their software. Also, every time you modify the source code to a program, you run the risk of accidentally introducing new bugs. By storing the data for a program in a different place than the code, you can make your programs easier for others to use and more resistant to bugs.

SUMMARY

Files are organized into folders (also called directories), and a path describes the location of a file. Every program running on your computer has a current working directory, which allows you to specify file paths relative to the current location instead of always typing the full (or absolute) path. The `os.path` module has many functions for manipulating file paths.

Your programs can also directly interact with the contents of text files. The `open()` function can open these files to read in their contents as one large string (with the `read()` method) or as a list of strings (with the `readlines()` method). The `open()` function can open files in write or append mode to create new text files or add to existing text files, respectively.

In previous chapters, you used the clipboard as a way of getting large amounts of text into a program, rather than typing it all in. Now you can have your programs read files directly from the hard drive, which is a big improvement, since files are much less volatile than the clipboard.

In the next chapter, you will learn how to handle the files themselves, by copying them, deleting them, renaming them, moving them, and more.

PRACTICE QUESTIONS

- Q: 1. What is a relative path relative to?
- Q: 2. What does an absolute path start with?
- Q: 3. What do the `os.getcwd()` and `os.chdir()` functions do?
- Q: 4. What are the `.` and `..` folders?
- Q: 5. In `C:\bacon\eggs\spam.txt`, which part is the dir name, and which part is the base name?
- Q: 6. What are the three `-mode` arguments that can be passed to the `open()` function?
- Q: 7. What happens if an existing file is opened in write mode?
- Q: 8. What is the difference between the `read()` and `readlines()` methods?
- Q: 9. What data structure does a shelf value resemble?

PRACTICE PROJECTS

For practice, design and write the following programs.

EXTENDING THE MULTICLIPBOARD

Extend the `multiclipboard` program in this chapter so that it has a `delete` <keyword> command line argument that will delete a keyword from the shelf. Then add a `delete` command line argument that will delete *all* keywords.

MAD LIBS

Create a Mad Libs program that reads in text files and lets the user add their own text anywhere the word *ADJECTIVE*, *NOUN*, *ADVERB*, or *VERB* appears in the text file. For example, a text file may look like this:

The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was unaffected by these events.

The program would find these occurrences and prompt the user to replace them.

Enter an adjective:

silly

Enter a noun:

chandelier

Enter a verb:

screamed

Enter a noun:

pickup truck

The following text file would then be created:

The silly panda walked to the chandelier and then screamed. A nearby pickup truck was unaffected by these events.

The results should be printed to the screen and saved to a new text file.

REGEX SEARCH

Write a program that opens all *.txt* files in a folder and searches for any line that matches a user-supplied regular expression. The results should be printed to the screen.

ORGANIZING FILES

In the previous chapter, you learned how to create and write to new files in Python. Your programs can also organize preexisting files on the hard drive. Maybe you've had the experience of going through a folder full of dozens, hundreds, or even thousands of files and copying, renaming, moving, or compressing them all by hand. Or consider tasks such as these:

- Making copies of all PDF files (and *only* the PDF files) in every subfolder of a folder
- Removing the leading zeros in the filenames for every file in a folder of hundreds of files named *spam001.txt*, *spam002.txt*, *spam003.txt*, and so on
- Compressing the contents of several folders into one ZIP file (which could be a simple backup system)

All this boring stuff is just begging to be automated in Python. By programming your computer to do these tasks, you can transform it into a quick-working file clerk who never makes mistakes.

As you begin working with files, you may find it helpful to be able to quickly see what the extension (*.txt*, *.pdf*, *.jpg*, and so on) of a file is. With macOS and Linux, your file browser most likely shows extensions automatically. With Windows, file extensions may be hidden by default. To show extensions, go to **Start ▶ Control Panel ▶ Appearance and Personalization ▶ Folder Options**. On the View tab, under Advanced Settings, uncheck the **Hide extensions for known file types** checkbox.

THE SHUTIL MODULE

The `shutil` (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the `shutil` functions, you will first need to use `import shutil`.

Copying Files and Folders

The `shutil` module provides functions for copying files, as well as entire folders.

Calling `shutil.copy(source, destination)` will copy the file at the path *source* to the folder at the path *destination*. (Both *source* and *destination* can be strings or `Path` objects.) If *destination* is a filename, it will be used as the new name of the copied file. This function returns a string or `Path` object of the copied file.

Enter the following into the interactive shell to see how `shutil.copy()` works:

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
❶ >>> shutil.copy(p / 'spam.txt', p / 'some_folder')
'C:\\Users\\AI\\some_folder\\spam.txt'
❷ >>> shutil.copy(p / 'eggs.txt', p / 'some_folder/eggs2.txt')
WindowsPath('C:/Users/AI/some_folder/eggs2.txt')
```

The first `shutil.copy()` call copies the file at `C:\Users\AI\spam.txt` to the folder `C:\Users\AI\some_folder`. The return value is the path of the newly copied file. Note that since a folder was specified as the destination ❶, the original *spam.txt* filename is used for the new, copied file's filename. The second `shutil.copy()` call ❷ also copies the file

at `C:\Users\Al\eggs.txt` to the folder `C:\Users\Al\some_folder` but gives the copied file the name `eggs2.txt`.

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it. Calling `shutil.copytree(source, destination)` will copy the folder at the path *source*, along with all of its files and subfolders, to the folder at the path *destination*. The *source* and *destination* parameters are both strings. The function returns a string of the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
>>> shutil.copytree(p / 'spam', p / 'spam_backup')
WindowsPath('C:/Users/Al/spam_backup')
```

The `shutil.copytree()` call creates a new folder named *spam_backup* with the same content as the original *spam* folder. You have now safely backed up your precious, precious spam.

Moving and Renaming Files and Folders

Calling `shutil.move(source, destination)` will move the file or folder at the path *source* to the path *destination* and will return a string of the absolute path of the new location.

If *destination* points to a folder, the *source* file gets moved into *destination* and keeps its current filename. For example, enter the following into the interactive shell:

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named *eggs* already exists in the `C:\` directory, this `shutil.move()` call says, -Move `C:\bacon.txt` into the folder `C:\eggs`.¶

If there had been a *bacon.txt* file already in `C:\eggs`, it would have been overwritten. Since it's easy to accidentally overwrite files in this way, you should take some care when using `move()`.

The *destination* path can also specify a filename. In the following example, the *source* file is moved *and* renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This line says, -Move `C:\bacon.txt` into the folder `C:\eggs`, and while you're at it, rename that *bacon.txt* file to *new_bacon.txt*.¶

Both of the previous examples worked under the assumption that there was a folder *eggs* in the `C:\` directory. But if there is no *eggs* folder, then `move()` will rename *bacon.txt* to a file named *eggs*.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Here, `move()` can't find a folder named *eggs* in the `C:\` directory and so assumes that *destination* must be specifying a filename, not a folder. So the *bacon.txt* text file is renamed to *eggs* (a text file without the *.txt* file extension)—probably not what you wanted! This can be a tough-to-spot bug in your programs since the `move()` call can happily do something that might be quite different from what you were expecting. This is yet another reason to be careful when using `move()`.

Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  --snip--
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\eggs\\ham'
```

Python looks for *eggs* and *ham* inside the directory *does_not_exist*. It doesn't find the nonexistent directory, so it can't move *spam.txt* to the path you specified.

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the `os` module, whereas to delete a folder and all of its contents, you use the `shutil` module.

- Calling `os.unlink(path)` will delete the file at *path*.
- Calling `os.rmdir(path)` will delete the folder at *path*. This folder must be empty of any files or folders.
- Calling `shutil.rmtree(path)` will remove the folder at *path*, and all files and folders it contains will also be deleted.

Be careful when using these functions in your programs! It's often a good idea to first run your program with these calls commented out and with `print()` calls added to show the files that would be deleted. Here is a Python program that was intended to delete files that have the *.txt* file extension but has a typo (highlighted in bold) that causes it to delete *.rxt* files instead:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt):
    os.unlink(filename)
```

If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt):
    #os.unlink(filename)
    print(filename)
```

Now the `os.unlink()` call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted. Running this version of the program first will show you that you've accidentally told the program to delete `.txt` files instead of `.txt` files.

Once you are certain the program works as intended, delete the `print(filename)` line and uncomment the `os.unlink(filename)` line. Then run the program again to actually delete the files.

Safe Deletes with the send2trash Module

Since Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the third-party `send2trash` module. You can install this module by running `pip install --user send2trash` from a Terminal window. (See [Appendix A](#) for a more in-depth explanation of how to install third-party modules.)

Using `send2trash` is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with `send2trash` you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed `send2trash`, enter the following into the interactive shell:

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

In general, you should always use the `send2trash.send2trash()` function to delete files and folders. But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does. If you want your program to free up disk space, use the `os` and `shutil` functions for deleting files and folders. Note that the `send2trash()` function can only send files to the recycle bin; it cannot pull files out of it.

WALKING A DIRECTORY TREE

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Let's look at the `C:\delicious` folder with its contents, shown in [Figure 10-1](#).

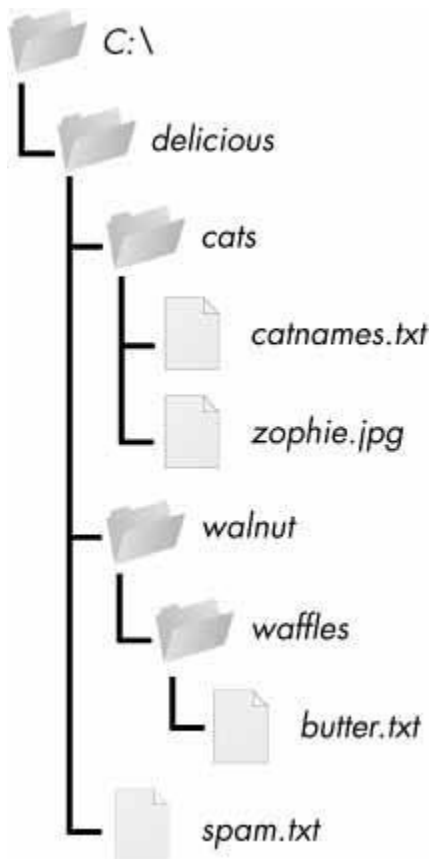


Figure 10-1: An example folder that contains three folders and four files

Here is an example program that uses the `os.walk()` function on the directory tree from [Figure 10-1](#):

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)

    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': ' + filename)

    print("")
```

The `os.walk()` function is passed a single string value: the path of a folder. You can use `os.walk()` in a for loop statement to walk a directory tree, much like how you can use the `range()` function to walk over a range of numbers. Unlike `range()`, the `os.walk()` function will return three values on each iteration through the loop:

- A string of the current folder's name
- A list of strings of the folders in the current folder
- A list of strings of the files in the current folder

(By current folder, I mean the folder for the current iteration of the for loop. The current working directory of the program is *not* changed by `os.walk()`.)

Just like you can choose the variable name `i` in the code `for i in range(10):`, you can also choose the variable names for the three values listed earlier. I usually use the names `foldername`, `subfolders`, and `filenames`.

When you run this program, it will output the following:

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt
```

```
The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg
```

```
The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles
```

```
The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since `os.walk()` returns lists of strings for the subfolder and filename variables, you can use these lists in their own for loops. Replace the `print()` function calls with your own custom code. (Or if you don't need one or both of them, remove the for loops.)

COMPRESSING FILES WITH THE ZIPFILE MODULE

You may be familiar with ZIP files (with the `.zip` file extension), which can hold the compressed contents of many other files. Compressing a file reduces its size, which is useful when transferring it over the internet. And since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an *archive file*, can then be, say, attached to an email.

Your Python programs can create and open (or *extract*) ZIP files using functions in the `zipfile` module. Say you have a ZIP file named *example.zip* that has the contents shown in [Figure 10-2](#).

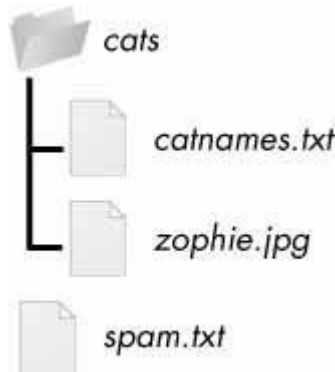


Figure 10-2: The contents of *example.zip*

You can download this ZIP file from <https://nostarch.com/automatestuff2/> or just follow along using a ZIP file already on your computer.

Reading ZIP Files

To read the contents of a ZIP file, first you must create a `ZipFile` object (note the capital letters *Z* and *F*). `ZipFile` objects are conceptually similar to the `File` objects you saw returned by the `open()` function in the previous chapter: they are values through which the program interacts with the file. To create a `ZipFile` object, call the `zipfile.ZipFile()` function, passing it a string of the *.ZIP* file's filename. Note that `zipfile` is the name of the Python module, and `ZipFile()` is the name of the function.

For example, enter the following into the interactive shell:

```
>>> import zipfile, os

>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> f'Compressed file is {round(spamInfo.file_size / spamInfo
.compress_size, 2)}x smaller!'
)
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

A `ZipFile` object has a `namelist()` method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the `getinfo()` `ZipFile` method to return a `ZipInfo` object about that particular file. `ZipInfo` objects have their own attributes, such as `file_size` and `compress_size` in bytes, which hold integers of the original file size and compressed file size, respectively. While a `ZipFile` object represents an entire archive file, a `ZipInfo` object holds useful information about a *single file* in the archive.

The command at ❶ calculates how efficiently *example.zip* is compressed by dividing the original file size by the compressed file size and prints this information.

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of *example.zip* will be extracted to *C:*. Optionally, you can pass a folder name to `extractall()` to have it extract the files into a folder other than the current working directory. If the folder passed to the `extractall()` method does not exist, it will be created. For instance, if you replaced the call at ❶ with `exampleZip.extractall('C:\\delicious')`, the code would extract the files from *example.zip* into a newly created *C:\\delicious* folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

The string you pass to `extract()` must match one of the strings in the list returned by `namelist()`. Optionally, you can pass a second argument to `extract()` to extract the file into a folder other than the current working directory. If this second argument is a folder that doesn't yet exist, Python will create the folder. The value that `extract()` returns is the absolute path to which the file was extracted.

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the `ZipFile` object in *write mode* by passing 'w' as the second argument. (This is similar to opening a text file in write mode by passing 'w' to the `open()` function.)

When you pass a path to the `write()` method of a `ZipFile` object, Python will compress the file at that path and add it into the ZIP file. The `write()` method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to `zipfile.ZIP_DEFLATED`. (This specifies the *deflate* compression algorithm, which works well on all types of data.) Enter the following into the interactive shell:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.

Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to `zipfile.ZipFile()` to open the ZIP file in *append mode*.

PROJECT: RENAMING FILES WITH AMERICAN-STYLE DATES TO EUROPEAN-STYLE DATES

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead.

Here's what the program does:

1. It searches all the filenames in the current working directory for American-style dates.
2. When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

1. Create a regex that can identify the text pattern of American-style dates.
2. Call `os.listdir()` to find all the files in the working directory.
3. Loop over each filename, using the regex to check whether it has a date.
4. If it has a date, rename the file with `shutil.move()`.

For this project, open a new file editor window and save your code as *renameDates.py*.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as TODO makes them easy to find using Mu editor's CTRL-F find feature. Make your code look like the following:

```
#!/ python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
```

❶ `import shutil, os, re`

`# Create a regex that matches files with the American date format.`

❷ `datePattern = re.compile(r"^(.*?) # all text before the date`
`((0|1)?\d)- # one or two digits for the month`
`((0|1|2|3)?\d)- # one or two digits for the day`
`((19|20)\d\d) # four digits for the year`
`(.*?)$ # all text after the date`
`""", re.VERBOSE❸)`

`# TODO: Loop over the files in the working directory.`

`# TODO: Skip files without a date.`

`# TODO: Get the different parts of the filename.`

`# TODO: Form the European-style filename.`

`# TODO: Get the full, absolute file paths.`

`# TODO: Rename the files.`

From this chapter, you know the `shutil.move()` function can be used to rename files: its arguments are the name of the file to rename and the new filename. Because this function exists in the `shutil` module, you must import that module ❶.

But before renaming the files, you need to identify which files you want to rename. Filenames with dates such as *spam4-4-1984.txt* and *01-03-2014eggs.zip* should be renamed, while filenames without dates such as *littlebrother.epub* can be ignored.

You can use a regular expression to identify this pattern. After importing the `re` module at the top, call `re.compile()` to create a `Regex` object ❷. Passing `re.VERBOSE` for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

The regular expression string begins with `^(.*?)` to match any text at the beginning of the filename that might come before the date. The `((0|1)?\d)` group matches the month. The first digit can be either 0 or 1, so the regex matches 12 for December but also 02 for February. This digit is also optional so that the month can be 04 or 4 for April. The group for the day is `((0|1|2|3)?\d)` and follows similar logic; 3, 03, and 31 are all valid numbers for days. (Yes, this regex will accept some invalid dates such as 4-31-2014, 2-29-2013, and 0-15-2014. Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

While 1885 is a valid year, you can just look for years in the 20th or 21st century. This will keep your program from accidentally matching nondate filenames with a date-like format, such as *10-10-1000.txt*.

The `(.*?)$` part of the regex will match any text that comes after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from `os.listdir()` and match them against the regex. Any files that do not have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Fill in the first three TODOs in your program with the following code:

```
#!/ python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
```

```
--snip--
```

```
# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)
```

```
    # Skip files without a date.
```

```
    ❶ if mo == None:
```

```
        ❷ continue
```

```
    ❸ # Get the different parts of the filename.
```

```
    beforePart = mo.group(1)
```

```
    monthPart = mo.group(2)
```

```
    dayPart = mo.group(4)
```

```
    yearPart = mo.group(6)
```

```
    afterPart = mo.group(8)
```

```
--snip--
```

If the Match object returned from the search() method is None ❶, then the filename in amerFilename does not match the regular expression. The continue statement ❷ will skip the rest of the loop and move on to the next filename.

Otherwise, the various strings matched in the regular expression groups are stored in variables named beforePart, monthPart, dayPart, yearPart, and afterPart ❸. The strings in these variables will be used to form the European-style filename in the next step.

To keep the group numbers straight, try reading the regex from the beginning, and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. Here's an example:

```
datePattern = re.compile(r"^(1) # all text before the date
(2 (3) )-          # one or two digits for the month
(4 (5) )-          # one or two digits for the day
(6 (7) )           # four digits for the year
(8)$              # all text after the date
"", re.VERBOSE)
```

Here, the numbers 1 through 8 represent the groups in the regular expression you wrote. Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: the date comes before the month. Fill in the three remaining TODOs in your program with the following code:

```
#!/ python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format # to
European DD-MM-YYYY.
```

```
--snip--
```

```
# Form the European-style filename.
```

```
❶ euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
    afterPart
```

```
# Get the full, absolute file paths.
```

```
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)
```

```
# Rename the files.
```

```
❷ print(f'Renaming "{amerFilename}" to "{euroFilename}"...')
```

```
❸ #shutil.move(amerFilename, euroFilename) # uncomment after testing
```

Store the concatenated string in a variable named `euroFilename` ❶. Then, pass the original filename in `amerFilename` and the new `euroFilename` variable to the `shutil.move()` function to rename the file ❸.

This program has the `shutil.move()` call commented out and instead prints the filenames that will be renamed ❷. Running the program like this first can let you double-check that the files are renamed correctly. Then you can uncomment the `shutil.move()` call and run the program again to actually rename the files.

Ideas for Similar Programs

There are many other reasons you might want to rename a large number of files.

- To add a prefix to the start of the filename, such as adding *spam_* to rename *eggs.txt* to *spam_eggs.txt*
- To change filenames with European-style dates to American-style dates
- To remove the zeros from files such as *spam0042.txt*

PROJECT: BACKING UP A FOLDER INTO A ZIP FILE

Say you're working on a project whose files you keep in a folder named *C:\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file -snapshots of the entire folder. You'd like to keep different versions, so you want the ZIP file's filename to increment each time it is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*, *AlsPythonBook_3.zip*, and so on. You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named `backupToZip()`. This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#!/ python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.
```

```
❶ import zipfile, os
```

```
def backupToZip(folder):
    # Back up the entire contents of "folder" into a ZIP file.
```

```
    folder = os.path.abspath(folder) # make sure folder is absolute
```

```
    # Figure out the filename this code should use based on
    # what files already exist.
```

```
    ❷ number = 1
```

```
    ❸ while True:
```

```

zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
if not os.path.exists(zipFilename):
    break
number = number + 1

```

④ # TODO: Create the ZIP file.

```

# TODO: Walk the entire folder tree and compress the files in each folder.
print('Done.')

```

```

backupToZip('C:\\delicious')

```

Do the basics first: add the shebang (#!) line, describe what the program does, and import the zipfile and os modules ❶.

Define a backupToZip() function that takes just one parameter, folder. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the folder folder, and add each of the subfolders and files to the ZIP file. Write TODO comments for these steps in the source code to remind yourself to do them later ❷.

The first part, naming the ZIP file, uses the base name of the absolute path of folder. If the folder being backed up is *C:\delicious*, the ZIP file's name should be *delicious_N.zip*, where *N = 1* is the first time you run the program, *N = 2* is the second time, and so on.

You can determine what *N* should be by checking whether *delicious_1.zip* already exists, then checking whether *delicious_2.zip* already exists, and so on. Use a variable named *number* for *N* ❸, and keep incrementing it inside the loop that calls *os.path.exists()* to check whether the file exists ❹. The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file. Make your program look like the following:

```

#!/python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
while True:
    zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
    if not os.path.exists(zipFilename):
        break
    number = number + 1

# Create the ZIP file.
print(f'Creating {zipFilename}...')
❶ backupZip = zipfile.ZipFile(zipFilename, 'w')

# TODO: Walk the entire folder tree and compress the files in each folder.

```

```
print('Done.')
```

```
backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the zipFilename variable, you can call zipfile.ZipFile() to actually create the ZIP file ❶. Be sure to pass 'w' as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the os.walk() function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
#!/ python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--

# Walk the entire folder tree and compress the files in each folder.
❶ for foldername, subfolders, filenames in os.walk(folder):
    print(f'Adding files in {foldername}...')
    # Add the current folder to the ZIP file.
    ❷ backupZip.write(foldername)

    # Add all the files in this folder to the ZIP file.
    ❸ for filename in filenames:
        newBase = os.path.basename(folder) + '_'
        if filename.startswith(newBase) and filename.endswith('.zip'):
            continue # don't back up the backup ZIP files
        backupZip.write(os.path.join(foldername, filename))
    backupZip.close()
print('Done.')
```

```
backupToZip('C:\\delicious')
```

You can use os.walk() in a for loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder.

In the for loop, the folder is added to the ZIP file ❷. The nested for loop can go through each filename in the filenames list ❸. Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
```

Adding files in C:\delicious\walnut\waffles...
Done.

The second time you run it, it will put all the files in *C:\delicious* into a ZIP file named *delicious_2.zip*, and so on.

Ideas for Similar Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you can write programs that do the following:

- Walk a directory tree and archive just files with certain extensions, such as *.txt* or *.py*, and nothing else.
- Walk a directory tree and archive every file except the *.txt* and *.py* ones.
- Find the folder in a directory tree that has the greatest number of files or the folder that uses the most disk space.

SUMMARY

Even if you are an experienced computer user, you probably handle files manually with the mouse and keyboard. Modern file explorers make it easy to work with a few files. But sometimes you'll need to perform a task that would take hours using your computer's file explorer.

The *os* and *shutil* modules offer functions for copying, moving, renaming, and deleting files. When deleting files, you might want to use the *send2trash* module to move files to the recycle bin or trash rather than permanently deleting them. And when writing programs that handle files, it's a good idea to comment out the code that does the actual copy/move/rename/delete and add a *print()* call instead so you can run the program and verify exactly what it will do.

Often you will need to perform these operations not only on files in one folder but also on every folder in that folder, every folder in those folders, and so on. The *os.walk()* function handles this trek across the folders for you so that you can concentrate on what your program needs to do with the files in them.

The *zipfile* module gives you a way of compressing and extracting files in *.ZIP* archives through Python. Combined with the file-handling functions of *os* and *shutil*, *zipfile* makes it easy to package up several files from anywhere on your hard drive. These *.ZIP* files are much easier to upload to websites or send as email attachments than many separate files.

Previous chapters of this book have provided source code for you to copy. But when you write your own programs, they probably won't come out perfectly the first time. The next chapter focuses on some Python modules that will help you analyze and debug your programs so that you can quickly get them working correctly.

PRACTICE QUESTIONS

1. What is the difference between *shutil.copy()* and *shutil.copytree()*?
2. What function is used to rename files?
3. What is the difference between the delete functions in the *send2trash* and *shutil* modules?
4. *ZipFile* objects have a *close()* method just like *File* objects' *close()* method.
What *ZipFile* method is equivalent to *File* objects' *open()* method?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Selective Copy

Write a program that walks through a folder tree and searches for files with a certain file extension (such as *.pdf* or *.jpg*). Copy these files from whatever location they are in to a new folder.

Deleting Unneeded Files

It's not uncommon for a few unneeded but humongous files or folders to take up the bulk of the space on your hard drive. If you're trying to free up room on your computer, you'll get the most bang for your buck by deleting the most massive of the unwanted files. But first you have to find them.

Write a program that walks through a folder tree and searches for exceptionally large files or folders—say, ones that have a file size of more than 100MB. (Remember that to get a file's size, you can use `os.path.getsize()` from the `os` module.) Print these files with their absolute path to the screen.

Filling in the Gaps

Write a program that finds all files with a given prefix, such as *spam001.txt*, *spam002.txt*, and so on, in a single folder and locates any gaps in the numbering (such as if there is a *spam001.txt* and *spam003.txt* but no *spam002.txt*). Have the program rename all the later files to close this gap.

As an added challenge, write another program that can insert gaps into numbered files so that a new file can be added.

CLASSES AND OBJECTS

Python is an object-oriented programming language, and *class* is a basis for any object oriented programming language. Class is a user-defined data type which binds data and functions together into single entity. Class is just a prototype (or a logical entity/blue print) which will not consume any memory. An object is an instance of a class and it has physical existence. One can create any number of objects for a class. A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).

Programmer-defined Types

A class in Python can be created using a keyword `class`. Here, we are creating an empty class without any members by just using the keyword `pass` within it.

```
class Point:  
    pass  
  
print(Point)
```

The output would be –

```
<class '__main__.Point'>
```

The term `__main__` indicates that the class `Point` is in the main scope of the current module.

In other words, this class is at the top level while executing the program.

Now, a user-defined data type `Point` got created, and this can be used to create any number of objects of this class. Observe the following statements –

```
p=Point()
```

Now, a reference (for easy understanding, treat reference as a pointer) to `Point` object is created and is returned.

This returned reference is assigned to the object `p`.

The process of creating a new object is called as *instantiation* and the object is *instance* of a class. When we print an object, Python tells which class it belongs to and where it is stored in the memory.

```
print(p)
```

The output would be –

```
<__main__.Point object at 0x003C1BF0>
```

The output displays the address (in hexadecimal format) of the object in the memory. It is now clear that, the object occupies the physical space, whereas the class does not.

Attributes

An object can contain named elements known as *attributes*. One can assign values to these attributes using dot operator. For example, keeping coordinate points in mind, we can assign two attributes x and y for the object p of a class Point as below –

```
p.x=10.0
```

```
p.y=20.0
```

A state diagram that shows an object and its attributes is called as *object diagram*. For the object p, the object diagram is shown in Figure 4.1.

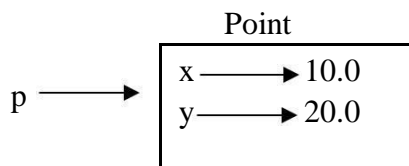


Figure 4.1 Object Diagram

The diagram indicates that a variable (i.e. object) p refers to a Point object, which contains two attributes. Each attributes refers to a floating point number.

One can access attributes of an object as shown –

```
>>> print(p.x)
>>> 10.0
>>> print(p.y)
>>> 20.0
```

Here, p.x means “Go to the object p refers to and get the value of x”. Attributes of an object can be assigned to other variables –

```
>>> a= p.x
>>> print(a)
10.0
```

Here, the variable x is nothing to do with attribute x. There will not be any name conflict between normal program variable and attributes of an object.

A complete program: Write a class Point representing a point on coordinate system. Implement following functions –

A function read_point() to receive x and y attributes of a Point object as user input.

A function distance() which takes two objects of Point class as arguments and computes the Euclidean distance between them.

A function print_point() to display one point in the form of ordered-pair.

```
import math
```

```
class Point:
```

```
    """This is a class Point representing a coordinate point"""
```

```
    def readpoint(p):
```

```
        p.x=int(input("x coordinate:"))
```

```
        p.y=int(input("y coordinate:"))
```

```
    def print_point(p):
```

```
        print("(%g,%g)"%(p.x, p.y))
```

```
    def distance(p1,p2):
```

```
        d=math.sqrt((p1.x-p2.x)**2+(p1.y-p2.y)**2)
```

```
        return d
```

```
p1=Point()
```

```
print("Enter First point:")
```

```
p1.readpoint()
```

```
p2=Point()
```

```
print("Enter Second point:")
```

```
p2.readpoint()
```

```
dist=p1.distance(p2)
```

```
print("First point is:")
```

```
p1.print_point( )
```

```
print("Second point is:")
```

```
p2.print_point()
```

```
print("Distance is: %g" %(p1.distance(p2)))
```

The sample output of above program would be –

```
Enter First point:
```

```
x coordinate:10
```

```
y coordinate:20      Enter
```

```
Second point:
```

```
x coordinate:3
```

```
y coordinate:5
```

```
First point is: (10,20)
```

Second point is:(3,5)

Distance is: 16.5529

Let us discuss the working of above program thoroughly –

The class Point contains a string enclosed within 3 double-quotes. This is known as **docstring**. Usually, a string literal is written within 3 consecutive double-quotes inside a class, module or function definition. It is an important part of documentation and is to help someone to understand the purpose of the said class/module/function. The docstring becomes a value for the special attribute viz. `__doc__` available for any class (and objects of that class). To get the value of docstring associated with a class, one can use the statements like –

```
>>> print(Point.__doc__)
```

This is a class Point representing a coordinate point

```
>>> print(p1.__doc__)
```

This is a class Point representing a coordinate point

Note that, you need to type two underscores, then the word doc and again two underscores.

In the above program, there is no need of docstring and we would have just used pass to indicate an empty class. But, it is better to understand the professional way of writing user-defined types and hence, introduced docstring.

The function `read_point()` take one argument of type Point object. When we use the statements like,

```
read_point(p1)
```

the parameter `p` of this function will act as an alias for the argument `p1`. Hence, the modification done to the alias `p` reflects the original argument `p1`. With the help of this function, we are instructing Python that the object `p1` has two attributes `x` and `y`.

The function `print_point()` also takes one argument and with the help of format-strings, we are printing the attributes `x` and `y` of the Point object as an ordered-pair `(x,y)`.

As we know, the Euclidean distance between two points (x_1, y_1) and (x_2, y_2) is

$$\sqrt{x_1^2 + x_2^2 + y_1^2 + y_2^2}$$

In this program, we have Point objects as $(p1.x, p1.y)$ and $(p2.x, p2.y)$. Apply the formula on these points by passing objects `p1` and `p2` as parameters to the function `distance()`. And then return the result.

Thus, the above program gives an idea of defining a class, instantiating objects, creating attributes, defining functions that takes objects as arguments and finally, calling (or invoking) such functions whenever and wherever necessary.

NOTE: User-defined classes in Python have two types of attributes viz. *class attributes* and *instance attributes*. Class attributes are defined inside the class (usually, immediately after class header). They are common to all the objects of that class. That is, they are shared by all the objects created from that class. But, instance attributes defined for individual objects. They are available only for that instance (or object). Attributes of one instance are not available for another instance of the same class. For example, consider the class Point as discussed earlier –

```
class Point:
    pass
p1= Point()                #first object of the class
p1.x=10.0                  #attributes for p1
p1.y=20.0
print(p1.x, p1.y)          #prints 10.0 20.0
p2= Point()                #second object of the class
print(p2.x)                #displays error as below
AttributeError: 'Point' object has no attribute 'x'
```

This clearly indicates that the attributes x and y created are available only for the object p1, but not for p2. Thus, x and y are instance attributes but not class attributes.

We will discuss class attributes late in-detail. But, for the understanding purpose, observe the following example –

```
class Point:
    x=2
    y=3
p1=Point()                 #first object of the class
print(p1.x, p1.y)          # prints 2    3
p2=Point()                 #second object of the class
print(p2.x, p2.y)          # prints 2    3
```

Here, the attributes x and y are defined inside the definition of the class Point itself. Hence, they are available to all the objects of that class.

Rectangles

It is possible to make an object of one class as an attribute to other class. To illustrate this, consider an example of creating a class called as Rectangle. A rectangle can be created using any of the following data –

By knowing width and height of a rectangle and one corner point (ideally, a bottom-left corner) in a coordinate system

By knowing two opposite corner points

Let us consider the first technique and implement the task: Write a class Rectangle containing numeric attributes width and height. This class should contain another attribute *corner* which is an instance of another class Point. Implement following functions –

A function to print corner point as an ordered-pair

A function *find_center()* to compute center point of the rectangle A

function *resize()* to modify the size of rectangle

The program is as given below –

class Point:

```
""" This is a class Point representing coordinate point """
```

class Rectangle:

```
""" This is a class Rectangle. Attributes: width, height and Corner Point """
```

```
def find_center(rect):
```

```
    p=Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

```
def resize(rect, w, h):
```

```
    rect.width +=w
    rect.height +=h
```

```
def print_point(p):
```

```
    print("(%g,%g)"%(p.x, p.y))
```

```
box=Rectangle()
```

```
box.corner=Point()
```

```
box.width=100
```

```
box.height=200
```

```
box.corner.x=0
```

```
box.corner.y=0
```

```
#create Rectangle object
```

```
#define an attribute corner for box
```

```
#set attribute          width to box
```

```
#set attribute          height to box
```

```
#corner itself          has two attributes x and y
```

```
#initialize x and y to 0
```

```
print("Original Rectangle is:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of rectangle is:")
print_point(center)

resize(box,50,70)
print("Rectangle after resize:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of resized rectangle is:")
print_point(center)
```

A sample output would be:

```
Original Rectangle is: width=100, height=200
The center of rectangle is: (50,100)
Rectangle after resize: width=150, height=270
The center of resized rectangle is: (75,135)
```

The working of above program is explained in detail here –

Two classes Point and Rectangle have been created with suitable docstrings. As of now, they do not contain any class-level attributes.

The following statement instantiates an object of Rectangle class.

```
box=Rectangle()
```

The statement

```
box.corner=Point()
```

indicates that corner is an attribute for the object box and this attribute itself is an object of the class Point. The following statements indicate that the object box has two more attributes

–

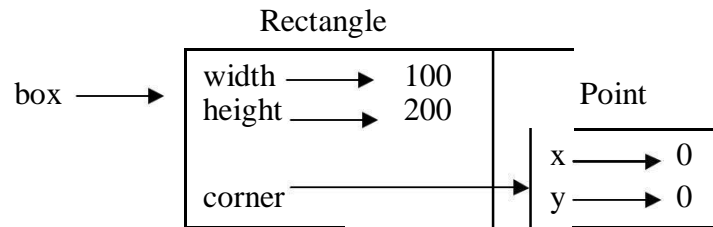
```
box.width=100           #give any numeric value
box.height=200          #give any numeric value
```

In this program, we are treating the corner point as the origin in coordinate system and hence the following assignments –

```
box.corner.x=0
box.corner.y=0
```

(Note that, instead of origin, any other location in the coordinate system can be given as corner point.)

Based on all above statements, an object diagram can be drawn as –



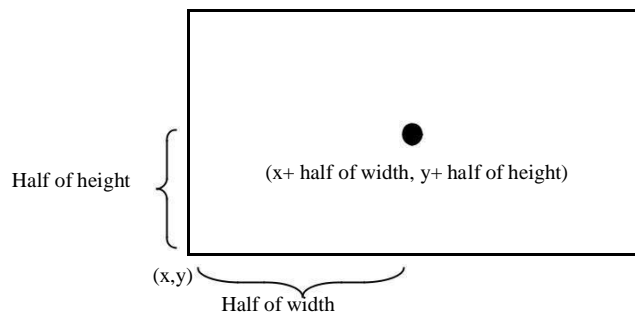
The expression `box.corner.x` means, -Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`.

The function `find_center()` takes an object `rect` as an argument. So, when a call is made using the statement –

```
center=find_center(box)
```

the object `rect` acts as an alias for the argument `box`.

A local object `p` of type `Point` has been created inside this function. The attributes of `p` are `x` and `y`, which takes the values as the coordinates of center point of rectangle. Center of a rectangle can be computed with the help of following diagram.



The function `find_center()` returns the computed center point. Note that, the return value of a function here is an instance of some class. That is, one can have an **instance as return values** from a function.

The function `resize()` takes three arguments: `rect` – an instance of `Rectangle` class and two numeric variables `w` and `h`. The values `w` and `h` are added to existing attributes `width` and `height`. This clearly shows that **objects are mutable**. State of an object can be changed by modifying any of its attributes. When this function is called with a statement –

```
resize(box,50,70)
```

the `rect` acts as an alias for `box`. Hence, width and height modified within the function will reflect the original object `box`.

Thus, the above program illustrates the concepts: *object of one class is made as attribute for object of another class, returning objects from functions* and *objects are mutable*.

Copying

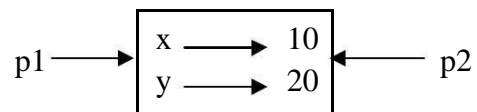
An object will be aliased whenever an object is assigned to another object of same class. This may happen in following situations –

- Direct object assignment (like p2=p1)
- When an object is passed as an argument to a function
- When an object is returned from a function

The last two cases have been understood from the two programs in previous sections. Let us understand the concept of aliasing more in detail using the following program –

```
>>> class Point: pass
>>> p1=Point()
>>> p1.x=10
>>> p1.y=20
>>> p2=p1
>>> print(p1)
<__main__.Point object at 0x01581BF0>
>>> print(p2)
<__main__.Point object at 0x01581BF0>
```

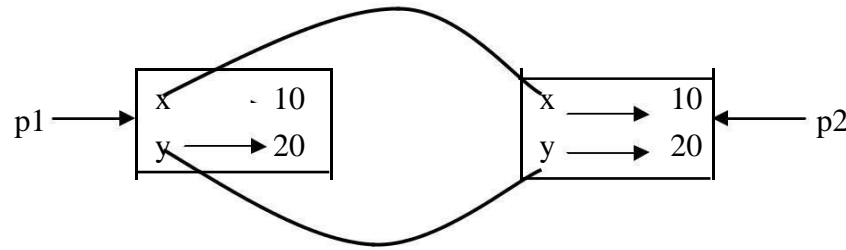
Observe that both p1 and p2 objects have same physical memory. It is clear now that the object p2 is an alias for p1. So, we can draw the object diagram as below –



Hence, if we check for equality and identity of these two objects, we will get following result.

```
>>> p1 is p2
True
>>> p1==p2
True
```

But, the aliasing is not good always. For example, we may need to create a new object using an existing object such that – the new object should have a different physical memory, but it must have same attribute (and their values) as that of existing object. Diagrammatically, we need something as below –



In short, ***we need a copy of an object, but not an alias.*** To do this, Python provides a module called ***copy*** and a method called ***copy()***. Consider the below given program to understand the concept.

```
>>> class Point:
        pass

>>> p1=Point()
>>> p1.x=10
>>> p1.y=20

>>> import copy                #import module copy
>>> p3=copy.copy(p1)           #use the method copy()
>>> print(p1)
    <__main__.Point object at 0x01581BF0>
>>> print(p3)
    <__main__.Point object at 0x02344A50>
>>> print(p3.x,p3.y) 10 20
```

Observe that the physical address of the objects p1 and p3 are now different. But, values of attributes x and y are same. Now, use the following statements –

```
>>> p1 is p3
    False
>>> p1 == p3
    False
```

Here, the is operator gives the result as False for the obvious reason of p1 and p3 are being two different entities on the memory. But, why == operator is generating False as the result, though the contents of two objects are same? The reason is – p1 and p3 are the objects of user-defined type. And, Python cannot understand the meaning of equality on the new data type. The default behavior of equality (==) is identity (is operator) itself. Hence, Python applies this default behavior on p1 == p3 and results in False.

(NOTE: If we need to define the meaning of equality (==) operator explicitly on user-defined data types (i.e. on class objects), then we need to override the method `__eq__()` inside the class. This will be discussed later in detail.)

The *copy()* method of *copy* module duplicates the object. The content (i.e. attributes) of one object is copied into another object as we have discussed till now. But, when an object itself is an attribute inside another object, the duplication will result in a strange manner. To understand this concept, try to copy Rectangle object (created in previous section) as given below –

```
import copy

class Point:
    """ This is a class Point representing coordinate
        point
    """

class Rectangle:
    """ This is a class Rectangle.
        Attributes: width, height and Corner Point
    """

box1=Rectangle()
box1.corner=Point()
box1.width=100
box1.height=200
box1.corner.x=0
box1.corner.y=0

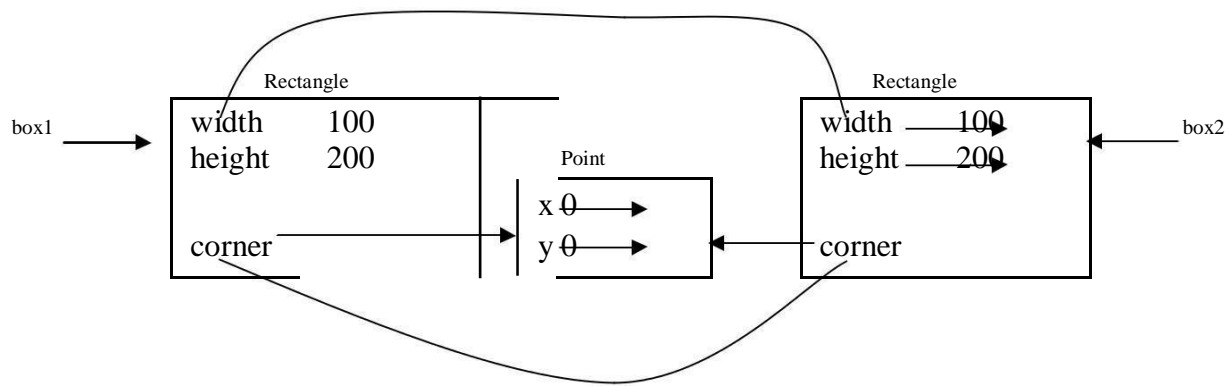
box2=copy.copy(box1)
print(box1 is box2)                                #prints    False

print(box1.corner is box2.corner)                   #prints    True
```

Now, the question is – why **box1.corner** and **box2.corner** are same objects, when **box1** and **box2** are different? Whenever the statement

```
box2=copy.copy(box1)
```

is executed, the contents of all the attributes of box1 object are copied into the respective attributes of box2 object. That is, box1.width is copied into box2.width, box1.height is copied into box2.height. Similarly, box1.corner is copied into box2.corner. Now, recollect the fact that corner is not exactly the object itself, but it is a reference to the object of type Point (Read the discussion done for Figure 4.1 at the beginning of this Chapter). Hence, the value of reference (that is, the physical address) stored in box1.corner is copied into box2.corner. Thus, the physical object to which box1.corner and box2.corner are pointing is only one. This type of copying the objects is known as *shallow copy*. To understand this behavior, observe the following diagram –



Now, the attributes width and height for two objects box1 and box2 are independent. Whereas, the attribute corner is shared by both the objects. Thus, any modification done to box1.corner will reflect box2.corner as well. Obviously, we don't want this to happen, whenever we create duplicate objects. That is, we want two independent physical objects. Python provides a method ***deepcopy()*** for doing this task. This method copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on.

```

box3=copy.deepcopy(box1)
print(box1 is box3)           #prints False
print(box1.corner is box3.corner) #prints False

```

Thus, the objects box1 and box3 are now completely independent.

Debugging

While dealing with classes and objects, we may encounter different types of errors. For example, if we try to access an attribute which is not there for the object, we will get ***AttributeError***. For example –

```

>>> p= Point()
>>> p.x = 10
>>> p.y = 20
>>> print(p.z)
AttributeError: 'Point' object has no attribute 'z'

```

To avoid such error, it is better to enclose such codes within try/except as given below – try:

```

z = p.x
except AttributeError:
    z = 0

```

When we are not sure, which type of object it is, then we can use ***type()*** as –

```

>>> type(box1)
<class '__main__.Rectangle'>

```

Another method *isinstance()* helps to check whether an object is an instance of a particular class –

```
>>> isinstance(box1, Rectangle) True
```

When we are not sure whether an object has a particular attribute or not, use a function *hasattr()* –

```
>>> hasattr(box1, 'width') True
```

Observe the string notation for second argument of the function *hasattr()*. Though the attribute width is basically numeric, while giving it as an argument to function *hasattr()*, it must be enclosed within quotes.

CLASSES AND FUNCTIONS

Though Python is object oriented programming languages, it is possible to use it as functional programming. There are two types of functions viz. *pure functions* and *modifiers*. A pure function takes objects as arguments and does some work without modifying any of the original argument. On the other hand, as the name suggests, modifier function modifies the original argument.

In practical applications, the development of a program will follow a technique called as *prototype* and *patch*. That is, solution to a complex problem starts with simple prototype and incrementally dealing with the complications.

Pure Functions

To understand the concept of pure functions, let us consider an example of creating a class called Time. An object of class Time contains hour, minutes and seconds as attributes. Write a function to print time in HH:MM:SS format and another function to add two time objects. Note that, adding two time objects should yield proper result and hence we need to check whether number of seconds exceeds 60, minutes exceeds 60 etc, and take appropriate action.

```
class Time:
    """Represents the time of a day Attributes: hour,
    minute, second """
    def printTime(t):
        print("% .2d:% .2d:% .2d"%(t.hour,t.minute,t.second))
    def add_time(t1,t2):
        sum=Time()
        sum.hour = t1.hour + t2.hour
        sum.minute = t1.minute + t2.minute
        sum.second = t1.second + t2.second
```

```
        if sum.second >= 60:
            sum.second -= 60
            sum.minute += 1
        if sum.minute >= 60:
            sum.minute -= 60
            sum.hour += 1
    return sum

t1=Time()
t1.hour=10
t1.minute=34
t1.second=25
print("Time1 is:")
printTime(t1)
t2=Time()
t2.hour=2
t2.minute=12
t2.second=41
print("Time2 is :")
printTime(t2)

t3=add_time(t1,t2)
print("After adding two time objects:")
printTime(t3)
```

The output of this program would be –

```
Time1 is: 10:34:25
Time2 is : 02:12:41
After adding two time objects: 12:47:06
```

Here, the function `add_time()` takes two arguments of type `Time`, and returns a `Time` object, whereas, it is not modifying contents of its arguments `t1` and `t2`. Such functions are called as *pure functions*.

Modifiers

Sometimes, it is necessary to modify the underlying argument so as to reflect the caller. That is, arguments have to be modified inside a function and these modifications should be available to the caller. The functions that perform such modifications are known as ***modifier function***. Assume that, we need to add few seconds to a time object, and get a new time. Then, we can write a function as below –

```
def increment(t, seconds):  
    t.second += seconds  
    while t.second >= 60:  
        t.second -= 60  
        t.minute += 1  
    while t.minute >= 60:  
        t.minute -= 60  
        t.hour += 1
```

In this function, we will initially add the argument seconds to t.second. Now, there is a chance that t.second is exceeding 60. So, we will increment minute counter till t.second becomes lesser than 60. Similarly, till the t.minute becomes lesser than 60, we will decrement minute counter. Note that, the modification is done on the argument t itself. Thus, the above function is a *modifier*.

Prototyping v/s Planning

Whenever we do not know the complete problem statement, we may write the program initially, and then keep of modifying it as and when requirement (problem definition) changes. This methodology is known as *prototype and patch*. That is, first design the prototype based on the information available and then perform patch-work as and when extra information is gathered. But, this type of incremental development may end-up in unnecessary code, with many special cases and it may be unreliable too.

An alternative is *designed development*, in which high-level insight into the problem can make the programming much easier. For example, if we consider the problem of adding two time objects, adding seconds to time object etc. as a problem involving numbers with base 60 (as every hour is 60 minutes and every minute is 60 seconds), then our code can be improved. Such improved versions are discussed later in this chapter.

CLASSES AND METHODS

The classes that have been considered till now were just empty classes without having any definition. But, in a true object oriented programming, a class contains class-level attributes, instance-level attributes, methods etc. There will be a tight relationship between the object of the class and the function that operate on those objects. Hence, the object oriented nature of Python classes will be discussed here.

Object-Oriented Features

As an object oriented programming language, Python possess following characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways objects in the real world interact.

To establish relationship between the object of the class and a function, we must define a function as a member of the class. A function which is associated with a particular class is known as a *method*. Methods are semantically the same as functions, but there are two syntactic differences:

Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

The syntax for invoking a method is different from the syntax for calling a function.

Now onwards, we will discuss about classes and methods.

Printing objects

we defined a class named Time and wrote a function named print_time:

```
class Time:
```

```
    """Represents the time of day."""
```

```
def print_time(time):
```

```
    print('%02d:%02d:%02d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a Time object as an argument:

```
>>> start = Time()
```

```
>>> start.hour = 9
```

```
>>> start.minute = 45
```

```
>>> start.second = 00
```

```
>>> print_time(start)
```

```
09:45:00
```

To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
```

```
    def print_time(time):
```

```
        print('%02d:%02d:%02d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call print_time. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)
```

```
09:45:00
```

In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
```

```
09:45:00
```

In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.

By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
```

```
    def print_time(self):
```

```
        print('%02d:%02d:%02d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, print_time(start) , suggests that the function is the

active agent. It says something like, -Hey print_time! Here's an object for you to print.¶

- In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says -Hey start! Please print yourself.¶

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite time_to_int (from Section 16.4) as a method. You might be tempted to rewrite int_to_time as a method, too, but that doesn't really make sense because there would be no object to invoke it on.

Another example

Here's a version of increment (from Section 16.3) rewritten as a method:

```
# inside class Time:
def increment(self, seconds):
    seconds += self.time_to_int()
    return int_to_time(seconds)
```

This version assumes that time_to_int is written as a method. Also, note that it is a pure function, not a modifier.

Here's how you would invoke increment:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.

This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:

```
>>> end = start.increment(1337, 460)
```

TypeError: increment() takes 2 positional arguments but 3 were given

The error message is initially confusing, because there are only two arguments in parentheses.

But the subject is also considered an argument, so all together that's three.

By the way, a **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

parrot and cage are positional, and dead is a keyword argument.

A more complicated example

Rewriting is_after (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter self and the second parameter other:

```
# inside class Time:
def is_after(self, other):
```

```
return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
```

```
True
```

One nice thing about this syntax is that it almost reads like English: —end is after start?||

The `__init__()` Method

(A method `__init__()` has to be written with two underscores before and after the word *init*)

Python provides a special method called `__init__()` which is similar to constructor method in other programming languages like C++/Java. The term *init* indicates initialization. As the name suggests, this method is invoked automatically when the object of a class is created. Consider the example given here –

```
import math

class Point:
    def __init__(self,a,b):
        self.x=a
        self.y=b

    def dist(self,p2):
        d=math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)
        return d

    def __str__(self):
        return("(%d,%d)"%(self.x, self.y))

p1=Point(10,20)           #__init__() is called automatically
p2=Point(4,5)             #__init__() is called automatically
print("P1 is:",p1)        #__str__() is called automatically
print("P2 is:",p2)        #__str__() is called automatically

d=p1.dist(p2)             #explicit call for dist()

print("The distance is:",d)
```

The sample output is –

```
P1 is: (10,20)
P2 is: (4,5)
Distance is: 16.15549442140351
```

Let us understand the working of this program and the concepts involved:

Keep in mind that every method of any class must have the first argument as *self*. The argument *self* is a reference to the current object. That is, it is reference to the object which invoked the method. (Those who know C++, can relate *self* with *this* pointer). The object which invokes a method is also known as *subject*.

The `__init__()` inside the class is an initialization method, which will be invoked automatically when the object gets created. When the statement like –

```
p1=Point(10,20)
```

is used, the `__init__()` method will be called automatically. The internal meaning of the above line is –

```
p1.__init__(10,20)
```

Here, `p1` is the object which is invoking a method. Hence, reference to this object is created and passed to `__init__()` as *self*. The values 10 and 20 are passed to formal parameters `a` and `b` of `__init__()` method. Now, inside `__init__()` method, we have statements

```
self.x=10
self.y=20
```

This indicates, `x` and `y` are instance attributes. The value of `x` for the object `p1` is 10 and, the value of `y` for the object `p1` is 20.

When we create another object `p2`, it will have its own set of `x` and `y`. That is, memory locations of instance attributes are different for every object.

Thus, state of the object can be understood by instance attributes.

The method `dist()` is an ordinary member method of the class `Point`. As mentioned earlier, its first argument must be *self*. Thus, when we make a call as –

```
d=p1.dist(p2)
```

a reference to the object `p1` is passed as *self* to `dist()` method and `p2` is passed explicitly as a second argument. Now, inside the `dist()` method, we are calculating distance between two point (Euclidian distance formula is used) objects. Note that, in this method, we cannot use the name `p1`, instead we will use *self* which is a reference (alias) to `p1`.

The `__str__` method

The next method inside the class is `__str__()`. ***It is a special method used for string representation of user-defined object.*** Usually, `print()` is used for printing basic types in Python. But, user-defined types (class objects) have their own meaning and a way of representation. To display such types, we can write functions or methods like `print_point()` as we did in Section 4.1.2. But, more polymorphic way is to use `__str__()` so that, when we write just `print()` in the main part of the program, the `__str__()` method will be invoked automatically. Thus, when we use the statement like –

```
print("P1 is:",p1)
```

the ordinary `print()` method will print the portion `-P1 is:` and the remaining portion is taken care by `__str__()` method. In fact, `__str__()` method will return the string

format what we have given inside it, and that string will be printed by print() method.

Operator Overloading

- *Ability of an existing operator to work on user-defined data type (class)* is known as operator overloading.
- It is a polymorphic nature of any object oriented programming.
- Basic operators like +, -, * etc. can be overloaded.
- To overload an operator, one needs to write a method within user-defined class.
- Python provides a special set of methods which have to be used for overloading required operator. The method should consist of the code what the programmer is willing to do with the operator. Following table shows gives a list of operators and their respective Python methods for overloading.

Operator	Special Function in Python	Operator	Special Function in Python
+	__add__()	<=	__le__()
-	__sub__()	>=	__ge__()
*	__mul__()	==	__eq__()
/	__truediv__()	!=	__ne__()
%	__mod__()	in	__contains__()
<	__lt__()	len	__len__()
>	__gt__()	str	__str__()

Let us consider an example of Point class considered earlier. Using operator overloading, we can try to add two point objects. Consider the program given below – p3=Point()

```
class Point:
    def __init__(self,a=0,b=0):
        self.x=a
        self.y=b

    def __add__(self, p2):
        p3=Point()
        p3.x=self.x+p2.x
        p3.y=self.y+p2.y
        return p3

    def __str__(self):
        return("(%d,%d)"%(self.x, self.y))
```

```
p1=Point(10,20)
p2=Point(4,5)
print("P1 is:",p1)
print("P2 is:",p2)
p4=p1+p2 #call for add__() method print("Sum is:",p4)
#p1._add_(p2)
```

The output would be –

```
P1 is: (10,20)
P2 is: (4,5)
Sum is: (14,25)
```

In the above program, when the statement `p4 = p1+p2` is used, it invokes a special method `_add_()` written inside the class. Because, internal meaning of this statement is–

```
p4 = p1._add_(p2)
```

Here, `p1` is the object invoking the method. Hence, `self` inside `_add_()` is the reference (alias) of `p1`. And, `p2` is passed as argument explicitly.

In the definition of `_add_()`, we are creating an object `p3` with the statement –

```
p3=Point()
```

The object `p3` is created without initialization. Whenever we need to create an object with and without initialization in the same program, we must set arguments of `__init__()` for some default values. Hence, in the above program arguments `a` and `b` of `__init__()` are made as default arguments with values as zero. Thus, `x` and `y` attributes of `p3` will be now zero. In the `_add_()` method, we are adding respective attributes of `self` and `p2` and storing in `p3.x` and `p3.y`. Then the object `p3` is returned. This returned object is received as `p4` and is printed.

NOTE that, in a program containing operator overloading, the overloaded operator behaves in a normal way when basic types are given. That is, in the above program, if we use the statements

```
m= 3+4
print(m)
```

it will be usual addition and gives the result as 7. But, when user-defined types are used as operands, then the overloaded method is invoked.

Let us consider a more complicated program involving overloading. Consider a problem of creating a class called `Time`, adding two `Time` objects, adding a number to `Time` object etc. that we had considered in previous section. Here is a complete program with more of OOP concepts.

```
class Time:
    def __init__(self, h=0,m=0,s=0):
        self.hour=h
        self.min=m
        self.sec=s

    def time_to_int(self):
        minute=self.hour*60+self.min
        seconds=minute*60+self.sec
        return seconds

    def int_to_time(self, seconds):
        t=Time()
        minutes, t.sec=divmod(seconds,60)
        t.hour, t.min=divmod(minutes,60)
        return t

    def __str__(self):
        return "%.2d:%.2d:%.2d"%(self.hour,self.min,self.sec)

    def __eq__(self,t):
        return self.hour==t.hour and self.min==t.min and self.sec==t.sec

    def __add__(self,t):
        if isinstance(t, Time):
            return self.addTime(t)
        else:
            return self.increment(t)

    def addTime(self, t):
        seconds=self.time_to_int()+t.time_to_int()
        return self.int_to_time(seconds)

    def increment(self, seconds):
        seconds +=
        self.time_to_int() return self.int_to_time(seconds)

    def __radd__(self,t):
        return self._add_(t)

T1=Time(3,40)
T2=Time(5,45)
print("T1 is:",T1)
print("T2 is:",T2)                #call for __eq__()
print("Whether T1 is same as T2?",T1==T2)
T3=T1+T2                          #call for __add__()
print("T1+T2 is:",T3)
T4=T1+75
print("T1+75=",T4)
#call for __add__()
T5=130+T1
print("130+T1=",T5)                #call for __radd__()
```

```
T6=sum([T1,T2,T3,T4])
print("Using sum([T1,T2,T3,T4]):",T6)
```

The output would be –

```
T1 is: 03:40:00
T2 is: 05:45:00
Whether T1 is same as T2? False
T1+T2 is: 09:25:00
T1+75= 03:41:15
130+T1= 03:42:10
Using sum([T1,T2,T3,T4]): 22:31:15
```

Working of above program is explained hereunder –

The class Time has `__init__()` method for initialization of instance attributes hour, min and sec. The default values of all these are being zero.

The method `time_to_int()` is used convert a Time object (hours, min and sec) into single integer representing time in number of seconds.

The method `int_to_time()` is written to convert the argument seconds into time object in the form of hours, min and sec. The built-in method *divmod()* gives the quotient as well as remainder after dividing first argument by second argument given to it.

Special method `__eq__()` is for overloading equality (`==`) operator. We can say one Time object is equal to the other Time object if underlying hours, minutes and seconds are equal respectively. Thus, we are comparing these instance attributes individually and returning either True or False.

When we try to perform addition, there are 3 cases –

- o Adding two time objects like `T3=T1+T2`.
- o Adding integer to Time object like `T4=T1+75`
- o Adding Time object to an integer like `T5=130+T1`

Each of these cases requires different logic. When first two cases are considered, the first argument will be T1 and hence self will be created and passed to `__add__()` method. Inside this method, we will check the type of second argument using `isinstance()` method. If the second argument is Time object, then we call `addTime()` method. In this method, we will first convert both Time objects to integer (seconds) and then the resulting sum into Time object again. So, we make use `time_to_int()` and `int_to_time()` here.

When the 2nd argument is an integer, it is obvious that it is number of seconds. Hence, we need to call `increment()` method.

Thus, based on the type of argument received in a method, we take appropriate action. This is known as *type-based dispatch*.

In the 3rd case like `T5=130+T1`, Python tries to convert first argument 130 into self, which is not possible. Hence, there will be an error. This indicates that for Python, `T1+5` is not same as `5+T1` (Commutative law doesn't hold good!!). To avoid the possible error, we need to implement *right-side addition* method `_radd_()`. Inside this method, we can call overloaded method `__add__()`.

The beauty of Python lies in surprising the programmer with more facilities!! As we have implemented `__add__()` method (that is, overloading of + operator), the built-in `sum()` will be capable of adding multiple objects given in a sequence. This is due to *Polymorphism* in Python. Consider a list containing Time objects, and then call `sum()` on that list as –

```
T6=sum([T1,T2,T3,T4])
```

The `sum()` internally calls `__add__()` method multiple times and hence gives the appropriate result.

Note down the square-brackets used to combine Time objects as a list and then passing it to `sum()`.

Thus, the program given here depicts many features of OOP concepts.

Inheritance

Inheritance is an object oriented programming language feature. Inheritance is the ability to define a new class that is a modified version of an existing class.





















































The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2. use integers to **encode** the ranks and suits. In this context, -encode means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be -encryption).

Example set of 52 playing cards; 13 of each suit clubs, diamonds, hearts, and spades

	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
Diamonds													
Hearts													
Spades													

For example, this table shows the suits and the corresponding integer codes:

Spades - 3

Hearts - 2

Diamonds - 1

Clubs - 0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The class definition for Card looks like this:

```
class Card:
    """Represents a standard playing card."""
    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```


the `_init` method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call Card with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings.

We assign these lists to **class attributes**:

```
# inside class Card:
class Card:
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']
    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank], Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance attributes** because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation.

For example, in `__str__`, `self`, `Card` is a class object, and `Card.rank_names` is a list of strings associated with the class.

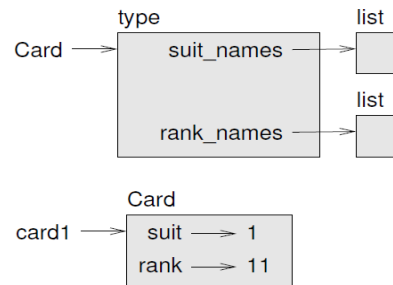


Figure 18.1: Object diagram.

Every card has its own suit and rank, but there is only one copy of `suit_names` and `rank_names`.

Putting it all together, the expression `Card.rank_names[self.rank]` means -use the attribute `rank` from the object `self` as an index into the list `rank_names` from the class `Card`, and select the appropriate string.

The first element of `rank_names` is `None` because there is no card with rank zero. By including `None` as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string '2', and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

Figure 18.1 is a diagram of the `Card` class object and one `Card` instance. `Card` is a class object; its type is `type`. `card1` is an instance of `Card`, so its type is `Card`.

Comparing cards

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named `__lt__`, which stands for -less than. `__lt__` takes two parameters, `self` and `other`, and returns `True` if `self` is strictly less than `other`.

The correct ordering for cards is not obvious.

For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__lt__`:

```
# inside class Card:
class Card:
    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False
        # suits are the same... check ranks
        return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:
class Card:
    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

As an exercise, write an `__lt__` method for `Time` objects. You can use tuple comparison, but you also might consider comparing integers.

Decks

Now that we have `Cards`, the next step is to define `Decks`.

Since a deck is made up of cards, it is natural for each `Deck` to contain a list of cards as an attribute.

The following is a class definition for `Deck`. The `init` method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13.

Each iteration creates a new `Card` with the current suit and rank, and appends it to `self.cards`.

Printing the deck

Here is a `__str__` method for Deck:

#inside class Deck:

class Card:

```
def __str__(self):
    res = []
    for card in self.cards:
        res.append(str(card))
    return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method `join`.

The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
>>> print(deck)
O/P
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it.

The list method `pop` provides a convenient way to do that:

#inside class Deck:

```
class Deck:
    def pop_card(self):
        return self.cards.pop()
```

Since `pop` removes the last card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method `append`:

```
#inside class Deck:
class Deck:
    def add_card(self, card):
        self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a **veneer**. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case `add_card` is a `-thin||` method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a Deck method named shuffle using the function shuffle from the random module:

```
import random
# inside class Deck:
class Deck:
    def shuffle(self):
        random.shuffle(self.cards)
```

Don't forget to import random.

As an exercise, write a Deck method named sort that uses the list method sort to sort the cards in a Deck. sort uses the `__lt__` method we defined to determine the order.

Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class.

As an example, let's say we want a class to represent a `Hand`, that is, the cards held by one player.

A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck.

For example, in poker we might compare two hands to see which one wins.

In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance.

To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

Syntax:

class classname(base class):

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for `Hands` as well as `Decks`.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

In this example, `Hand` inherits `__init__` from `Deck`, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the `init` method for `Hands` should initialize cards with an empty list.

If we provide an `init` method in the `Hand` class, it overrides the one in the `Deck` class:

```
# inside class Hand:
class Hand:
    def __init__(self, label=""):
        self.cards = []
        self.label = label
```

When you create a `Hand`, Python invokes this `init` method, not the one in `Deck`.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
```

'new hand'

The other methods are inherited from Deck, so we can use pop_card and add_card to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called move_cards:

```
#inside class Deck:
class Deck:
    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

move_cards takes two arguments, a Hand object and the number of cards to deal. It modifies both self and hand, and returns None.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use move_cards for any of these operations: self can be either a Deck or a Hand, and hand, despite the name, can also be a Deck.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it.

Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them.

In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read.

When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values.

These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed.

A class diagram is a more abstract representation of the structure of a program.

Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in, —a Rectangle has a Point.¶
- One class might inherit from another. This relationship is called **IS-A**, as in, —a Hand is a kind of a Deck.¶
- One class might depend on another in the sense that objects in one class take objects in the second

class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

A **class diagram** is a graphical representation of these relationships.

For example, Figure 18.2 shows the relationships between Card, Deck and Hand.

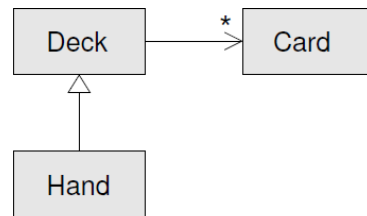


Figure 18.2: Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a **multiplicity**; it indicates how many Cards a Deck has.

A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a list of Cards, but built-in types like list and dict are usually not included in class diagrams.

Data encapsulation

The previous chapters demonstrate a development plan we might call –object-oriented design.

We identified objects we needed—like Point, Rectangle and Time—and defined classes to represent them.

In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact.

In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from Section 13.8, provides a good example. If you download my code from <http://thinkpython2.com/code/markov.py>, you’ll see that it uses two global variables—suffix_map and prefix—that are read and written from several functions.

```

suffix_map = {}
prefix = ()
  
```

Because these variables are global, we can only run one analysis at a time.

If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here’s what that looks like:

```

class Markov:
    def __init__(self):
  
```

```
self.suffix_map = {}  
self.prefix = ()
```

Next, we transform the functions into methods. For example, here's `process_word`:

```
def process_word(self, word, order=2):  
    if len(self.prefix) < order:  
        self.prefix += (word,)   
        return  
  
    try:  
        self.suffix_map[self.prefix].append(word)  
    except KeyError:  
        # if there is no entry for this prefix, make one  
        self.suffix_map[self.prefix] = [word]  
        self.prefix = shift(self.prefix, word)
```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring.

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from <http://thinkpython2.com/code/markov.py>, and follow the steps described above to encapsulate the global variables as attributes of a new class called Markov. Solution: <http://thinkpython2.com/code/markov2.py>.

GENERAL OOP CONCEPTS

At the earlier age of computers, the programming was done using assembly language. Even though, the assembly language can be used to produce highly efficient programs, it is not easy to learn or to use effectively. Moreover, debugging assembly code is quite difficult. At the later stage, the programming languages like BASIC, COBOL and FORTRAN came into existence. But, these languages are non-structured and consisting of a mass of tangled jumps and conditional branches that make a program virtually impossible to understand.

To overcome these problems, the structured or procedural programming methodology was developed. Here, the actual problem is divided into small independent tasks/modules. Then the programs are written for each of these tasks and they are grouped together to get the final solution for the given problem. Thus, the solution design technique for this method is known as *top-down approach*. C is the one successful language that adopted structured programming style. However, even with the structured programming methods, once a project/program reaches a certain size, its complexity exceeds what a programmer can manage.

The new approach – object oriented programming was developed to overcome the problems with structured approach. In this methodology, the actual data and the operations to be performed on that are grouped as a single entity called object. The objects necessary to get the solution of a problem are identified initially. The interactions between various objects are then identified to achieve the solution of original problem. Thus, it is also known as *bottom-up approach*. Object oriented concepts inhabits many advantages like re-usability of the code, security etc.

In structured programming approach, the programs are written around *what is happening* rather than *who is being affected*. That is, structured programming focuses more on the process or operation and not on the data to be processed. This is known as *process oriented model* and this can be thought of as *code acting on data*. For example, a program written in C is defined by its functions, any of which may operate on any type of data used by the program.

But, the real world problems are not organized into data and functions independent of each other and are tightly closed. So, it is better to write a program around *'who is being affected'*. This kind of data-centered programming methodology is known as *object oriented programming (OOP)* and this can be thought of as *data controlling access to code*. Here, the behavior and/or characteristics of the data/objects are used to determine the function to be written for applying them. Thus, the basic idea behind OOP language is to combine both data and the function that operates on data into a single unit. Such a unit is called as an *object*. A function that operates on data is known as a *member function* and it is the only means of accessing an object's data.

Elements of OOP: The object oriented programming supports some of the basic concepts as building blocks. Every OOPs language normally supports and developed around these features. They are

- Class
- Object
- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

Class: A class is a user defined data type which binds data and functions together into a single entity.

Class is a building block of any object oriented language.

As it is discussed earlier, object oriented programming treats data and the code acting on that data as a connected component.

That is, data and code are not treated separately as procedure oriented languages do.

Thus, OOPs suggests to wrap up the data and functions together into a single entity.

Normally, a class represents the prototype of a real world entity.

Hence, a class, by its own, is not having any physical existence.

It can be treated as a user-defined data type.

Since a class is a prototype or blueprint of a real world entity, it consists of number of properties (known as data members) and behavior (known as member functions).

To illustrate this, consider an example of a class representing a human being shown in the following Figure –

HumanBeing	
-	Hair Color
-	Number of legs
-	Number of eyes
-	Skin Color
-	Gender
+	Walking
+	Talking
+	Eating
#	Sleeping

Class diagram for Human Being

Few of the properties of human can be *number of legs, number of eyes, gender, number of hands, hair color, skin color etc.*

And the functionality or behavior of a human may be *walking, talking, eating, thinking etc.*

Object : An object is an instance of a class.

A class is just a prototype, representing a new data type.

To use this new data type, we need to create a variable of this type. Such a variable is known as an object.

Thus, an object is a physical entity, which consumes memory.

In OOPs, the object represents a real world data which defines the properties and behavior of any real world entity.

Every object has its unique existence and it is different from the other objects of a same class.

Let us refer to the class of human being discussed in previous section.

Assume that there are two persons: Ramu and Radha.

Now, properties of Ramu and Radha may be having different values as shown in the following Table.

Properties of Objects

Property/Attribute	Objects	
	Ramu	Radha
Skin color	Wheatish	Fair
Hair	gray	black
Number of legs	2	2
Number of eyes	2	2

<u>HumanBeing:Ramu</u>
Skinkcolor:Wheatish(string) Hair:gray(string) Numberoflegs:2(int)

Also, the walking and talking style of both of them may be different. Hence there are two different objects of the same class.

Thus, two or more objects of the same class will differ in the values of their properties and way they behave. But, they share common set of types of properties and behavior.

Encapsulation: The process of binding data and code together into a single entity is called encapsulation.

It is the mechanism that binds code and the data it manipulates together and keeps both safe from misuse and unauthorized manipulation. In any OOP language, the basis of encapsulation is the *class*. Class defines the structure and behavior (i.e. data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class. So, object is also referred to as an *instance of a class* and it is thought just as a variable of user-defined data type. Thus, class is a logical construct and an object is a physical entity.

The data defined by the class are known as *member variables* and the functions written to operate on that data are known as *member functions or methods*. The member variables and functions can be *private* or *public*.

If a particular member is private, then only the other members of that class can access that member. That is, a private member of the class can't be accessed from outside the class.

But, if any member is public, then it can be accessed from anywhere in the program.

Thus, through encapsulation, an OOP technique provides high security for user's data and for the entire system.

Data Abstraction: Hiding the implementation details from the end user.

Many a times, abstraction and encapsulation are used interchangeably.

But, actually, they are not same.

In OOPs, the end user of the program need not know how the actual function works, or what is being done inside a function to make it work.

The user must know only the abstract meaning of the task, and he can freely call a function to do that task.

The internal details of function may not be known to the user.

Designing the program in such a way is called as abstraction.

To understand the concept of abstraction, consider a scenario:

When you are using Microsoft Word, if you click on Save icon, a dialogue box appears which allows you to save the document in a physical location of your choice.

Similarly, when you click Open icon, another dialogue box appears to select a file to be opened from the hard disk.

You, as a user will not be knowing how the internal code would have written so as to open a dialogue box when an icon is being clicked.

As a user, those details are not necessary for you.

Thus, such implementation details are hidden from the end user.

This is an abstraction.

Being an OOPs programmer, one should design a class (with data members and member functions) such a way that, the internal code details are hidden from the end user.

OOPs provide a facility of having member functions to achieve this technique and the external world (normally, a main() function) needs to call the member function using an object to achieve a particular task.

Inheritance: Making use of existing code.

It is a process by which one object can acquire the properties of another object.

It supports the concept of hierarchical (top-down) classification.

For example, consider a large set of animals having their own behaviors.

In that, mammals are of one kind having all the properties of animals with some additional behaviors that are unique to them.

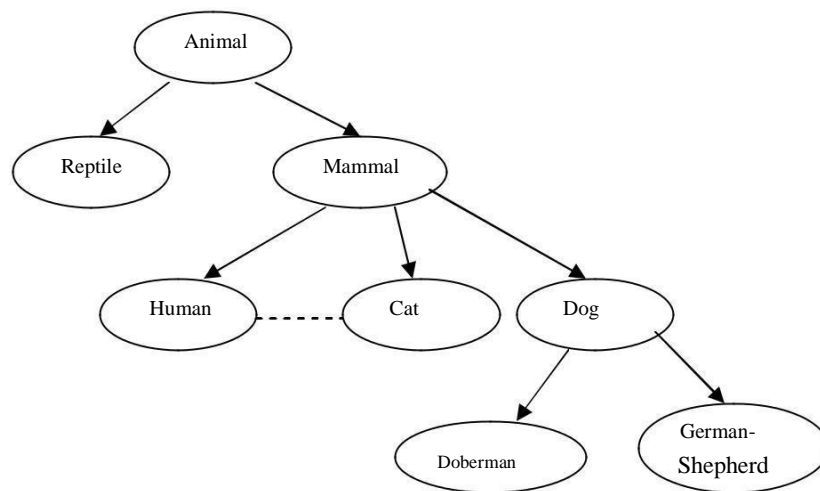
Again, we can divide mammals class into various mammals like dogs, cats, human etc. Again among the dogs, differentiation is there like

Doberman, German-shepherd, Labrador etc.

Thus, if we consider a German-shepherd, it is having all the qualities of a dog along with its own special features.

Moreover, it exhibits all the properties of a mammal, and in turn of an animal. Hence it is inheriting the properties of animals, then of mammals and then of dogs along with its own specialties.

We can depict it as shown in the Figure given below.



Example of Inheritance

Normally, inheritance of this type is also known as -is-all relationship.

Because, we can easily say -Doberman *is a* dog, -Dog *is a* mammal etc.

Hence, inheritance is termed as **Generalization to Specialization** if we consider from top-to-bottom level.

On the other hands, it can be treated as **Specialization to Generalization** if it is bottom-to-top level.

This indicates, in inheritance, the topmost base class will be more generalized with only properties which are common to all of its derived classes (various levels) and the bottom-most class is most specialized version of the class which is ready to use in a real-world.

If we apply this concept for programming, it can be easily understood that a code written is reusable. Thus, in this mechanism, it is possible for one object to be a specific instance of a more general case. Using inheritance, an object need only define those qualities that make it unique object within its class. It can inherit its general attributes from its parent.

Polymorphism: This can be thought of as one interface, multiple methods.

It is a feature that allows one interface to be used for a general class of actions.

The specific action is determined by the exact nature of the situation.

Using this mechanism, function overloading and operator overloading can be done.

Consider an example of performing stack operation on three different types of data viz. integer, floating-point and characters.

In a non-object oriented programming, we have to write functions with different name for push and pop operations for all these types of data even though the logic is same for all the data types. But in OOP languages, we can use the same function names with the data types of the parameters being different.

This is an example for function overloading.

We know that the `++` operator is used for adding two numbers.

Conceptually, the concatenation of two strings is also an addition.

But, in non-object oriented programming language, we cannot use `++` operator to concatenate two strings.

This is possible in object oriented programming language by overloading the `++` operator for string operands.

Polymorphism is also meant for *economy of expression*.

That is, the way you express things is more economical when you use polymorphism.

For example, if you have a function to add two matrices, you can use just a `+` symbol as:

`m3 = m1 + m2;`

here, `m1`, `m2` and `m3` are objects of matrix class and `+` is an overloaded operator. In the same program, if you have a function to concatenate two strings, you can write an overloaded function for `+` operator to do so –

`s3 = s1 + s2;` where `s1`, `s2` and `s3` are strings.

Moreover, for adding two numbers, the same `+` operator is used with its default behavior. Thus, one single operator `+` is being used for multiple purposes without disturbing its abstract meaning – addition. But, type of data it is adding is different. Hence, the way you have expressed your statements is more economical rather than having multiple functions like –

`addMat(m1, m2);`

`concat(s1, s2);` etc.

MODULE-V

WEB SCRAPING

In those rare, terrifying moments when I'm without Wi-Fi, I realize just how much of what I do on the computer is really what I do on the internet. Out of sheer habit I'll find myself trying to check email, read friends' Twitter feeds, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *RoboCop*?"¹

Since so much work on a computer involves going on the internet, it'd be great if your programs could get online. **Web scraping** is the term for using a program to download and process content from the web. For example, Google runs many web scraping programs to index web pages for its search engine. In this chapter, you will learn about several modules that make it easy to scrape web pages in Python.

webbrowser Comes with Python and opens a browser to a specific page.

requests Downloads files and web pages from the internet.

bs4 Parses HTML, the format that web pages are written in.

selenium Launches and controls a web browser. The selenium module is able to fill in forms and simulate mouse clicks in this browser.

PROJECT: MAPIT.PY WITH THE WEBBROWSER MODULE

The webbrowser module's `open()` function can launch a new browser to a specified URL. Enter the following into the interactive shell:

```
>>> import webbrowser
>>> webbrowser.open('https://inventwithpython.com/')
```

A webbrowser tab will open to the URL <https://inventwithpython.com/>. This is about the only thing the webbrowser module can do. Even so, the `open()` function does make some interesting things possible. For example, it's tedious to copy a street address to the clipboard and bring up a map of it on Google Maps. You could take a few steps out of this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you only have to copy the address to a clipboard and run the script, and the map will be loaded for you.

This is what your program does:

1. Gets a street address from the command line arguments or clipboard
2. Opens the web browser to the Google Maps page for the address

This means your code will need to do the following:

1. Read the command line arguments from `sys.argv`.
2. Read the clipboard contents.
3. Call the `webbrowser.open()` function to open the web browser.

Open a new file editor tab and save it as *mapIt.py*.

Step 1: Figure Out the URL

Based on the instructions in Appendix B, set up *mapIt.py* so that when you run it from the command line, like so . . .

```
C:\> mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the script will use the command line arguments instead of the clipboard. If there are no command line arguments, then the program will know to use the contents of the clipboard.

First you need to figure out what URL to use for a given street address. When you load <https://maps.google.com/> in the browser and search for an address, the URL in the address bar looks something like this: <https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1!4b1!4m2!3m1!1s0x808f7e3dad07a37:0xc86b0b2bb93b73d8>.

The address is in the URL, but there's a lot of additional text there as well. Websites often add extra data to URLs to help track visitors or customize sites. But if you try just going to <https://www.google.com/maps/place/870+Valencia+St+San+Francisco+CA/>, you'll find that it still brings up the correct page. So your program can be set to open a web browser to 'https://www.google.com/maps/place/your_address_string' (where *your_address_string* is the address you want to map).

Step 2: Handle the Command Line Arguments

Make your code look like this:

```
#!/python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys
if len(sys.argv) > 1:
    # Get address from command line.
    address = ''.join(sys.argv[1:])

# TODO: Get address from clipboard.
```

After the program's `#!/` shebang line, you need to import the `webbrowser` module for launching the browser and import the `sys` module for reading the potential command line arguments. The `sys.argv` variable stores a list of the program's filename and command line

arguments. If this list has more than just the filename in it, then `len(sys.argv)` evaluates to an integer greater than 1, meaning that command line arguments have indeed been provided.

Command line arguments are usually separated by spaces, but in this case, you want to interpret all of the arguments as a single string. Since `sys.argv` is a list of strings, you can pass it to the `join()` method, which returns a single string value. You don't want the program name in this string, so instead of `sys.argv`, you should pass `sys.argv[1:]` to chop off the first element of the array. The final string that this expression evaluates to is stored in the address variable.

If you run the program by entering this into the command line . . .

```
mapIt 870 Valencia St, San Francisco, CA 94110
```

. . . the `sys.argv` variable will contain this list value:

```
['mapIt.py', '870', 'Valencia', 'St, ', 'San', 'Francisco, ', 'CA', '94110']
```

The address variable will contain the string '870 Valencia St, San Francisco, CA 94110'.

Step 3: Handle the Clipboard Content and Launch the Browser

Make your code look like the following:

```
#!/python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.
import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Get address from command line.
    address = ''.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()

webbrowser.open('https://www.google.com/maps/place/' + address)
```

If there are no command line arguments, the program will assume the address is stored on the clipboard. You can get the clipboard content with `pyperclip.paste()` and store it in a variable named `address`. Finally, to launch a web browser with the Google Maps URL, call `webbrowser.open()`.

While some of the programs you write will perform huge tasks that save you hours, it can be just as satisfying to use a program that conveniently saves you a few seconds each time you perform a common task, such as getting a map of an address. [Table 12-1](#) compares the steps needed to display a map with and without *mapIt.py*.

Table 12-1: Getting a Map with and Without *mapIt.py*

Manually getting a map	Using mapIt.py
Highlight the address.	Highlight the address.
Copy the address.	Copy the address.
Open the web browser.	Run <i>mapIt.py</i> .
Go to https://maps.google.com/ .	
Click the address text field.	
Paste the address.	
Press enter.	

See how *mapIt.py* makes this task less tedious?

Ideas for Similar Programs

As long as you have a URL, the webbrowser module lets users cut out the step of opening the browser and directing themselves to a website. Other programs could use this functionality to do the following:

- Open all links on a page in separate browser tabs.
- Open the browser to the URL for your local weather.
- Open several social network sites that you regularly check.

DOWNLOADING FILES FROM THE WEB WITH THE REQUESTS MODULE

The requests module lets you easily download files from the web without having to worry about complicated issues such as network errors, connection problems, and data compression. The requests module doesn't come with Python, so you'll have to install it first. From the command line, run **pip install --user requests**. ([Appendix A](#) has additional details on how to install third-party modules.)

The requests module was written because Python's urllib2 module is too complicated to use. In fact, take a permanent marker and black out this entire paragraph. Forget I ever mentioned urllib2. If you need to download things from the web, just use the requests module.

Next, do a simple test to make sure the requests module installed itself correctly. Enter the following into the interactive shell:

```
>>> import requests
```

If no error messages show up, then the requests module has been successfully installed.

Downloading a Web Page with the requests.get() Function

The requests.get() function takes a string of a URL to download. By calling type() on requests.get()'s return value, you can see that it returns a Response object, which contains the response that the web server gave for your request. I'll explain

the Response object in more detail later, but for now, enter the following into the interactive shell while your computer is connected to the internet:

```
>>> import requests
❶ >>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> type(res)
<class 'requests.models.Response'>
❷ >>> res.status_code == requests.codes.ok
True
>>> len(res.text)
178981
>>> print(res.text[:250])
The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare
```

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Proje

The URL goes to a text web page for the entire play of *Romeo and Juliet*, provided on this book's site ❶. You can tell that the request for this web page succeeded by checking the `status_code` attribute of the Response object. If it is equal to the value of `requests.codes.ok`, then everything went fine ❷. (Incidentally, the status code for “OK” in the HTTP protocol is 200. You may already be familiar with the 404 status code for “Not Found.”) You can find a complete list of HTTP status codes and their meanings at https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

If the request succeeded, the downloaded web page is stored as a string in the Response object's `text` variable. This variable holds a large string of the entire play; the call to `len(res.text)` shows you that it is more than 178,000 characters long. Finally, calling `print(res.text[:250])` displays only the first 250 characters.

If the request failed and displayed an error message, like “Failed to establish a new connection” or “Max retries exceeded,” then check your internet connection. Connecting to servers can be quite complicated, and I can't give a full list of possible problems here. You can find common causes of your error by doing a web search of the error message in quotes.

Checking for Errors

As you've seen, the Response object has a `status_code` attribute that can be checked against `requests.codes.ok` (a variable that has the integer value 200) to see whether the download succeeded. A simpler way to check for success is to call the `raise_for_status()` method on the Response object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded. Enter the following into the interactive shell:

```
>>> res = requests.get('https://inventwithpython.com/page_that_does_not_exist')
```

```
>>> res.raise_for_status()
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "C:\Users\AI\AppData\Local\Programs\Python\Python37\lib\site-packages\requests\models

.py", line 940, in raise_for_status

raise HTTPError(http_error_msg, response=self)

requests.exceptions.HTTPError: 404 Client Error: Not Found for url:

https://inventwithpython

.com/page_that_does_not_exist.html

The `raise_for_status()` method is a good way to ensure that a program halts if a bad download occurs. This is a good thing: You want your program to stop as soon as some unexpected error happens. If a failed download *isn't* a deal breaker for your program, you can wrap the `raise_for_status()` line with try and except statements to handle this error case without crashing.

```
import requests
```

```
res = requests.get('https://inventwithpython.com/page_that_does_not_exist')
```

```
try:
```

```
    res.raise_for_status()
```

```
except Exception as exc:
```

```
    print('There was a problem: %s' % (exc))
```

This `raise_for_status()` method call causes the program to output the following:

```
There was a problem: 404 Client Error: Not Found for url: https://
```

```
inventwithpython.com/page_that_does_not_exist.html
```

Always call `raise_for_status()` after calling `requests.get()`. You want to be sure that the download has actually worked before your program continues.

SAVING DOWNLOADED FILES TO THE HARD DRIVE

From here, you can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. There are some slight differences, though. First, you must open the file in *write binary* mode by passing the string `'wb'` as the second argument to `open()`. Even if the page is in plaintext (such as the *Romeo and Juliet* text you downloaded earlier), you need to write binary data instead of text data in order to maintain the *Unicode encoding* of the text.

To write the web page to a file, you can use a for loop with the Response object's `iter_content()` method.

```
>>> import requests
>>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
>>>     playFile.write(chunk)

100000
78981
>>> playFile.close()
```

The `iter_content()` method returns “chunks” of the content on each iteration through the loop. Each chunk is of the *bytes* data type, and you get to specify how many bytes each chunk will contain. One hundred thousand bytes is generally a good size, so pass 100000 as the argument to `iter_content()`.

The file *RomeoAndJuliet.txt* will now exist in the current working directory. Note that while the filename on the website was *rj.txt*, the file on your hard drive has a different filename. The requests module simply handles downloading the contents of web pages. Once the page is downloaded, it is simply data in your program. Even if you were to lose your internet connection after downloading the web page, all the page data would still be on your computer.

UNICODE ENCODINGS

Unicode encodings are beyond the scope of this book, but you can learn more about them from these web pages:

- Joel on Software: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): <https://www.joelonsoftware.com/articles/Unicode.html>
- Pragmatic Unicode: <https://nedbatchelder.com/text/unipain.html>

The `write()` method returns the number of bytes written to the file. In the previous example, there were 100,000 bytes in the first chunk, and the remaining part of the file needed only 78,981 bytes.

To review, here's the complete process for downloading and saving a file:

1. Call `requests.get()` to download the file.
2. Call `open()` with 'wb' to create a new file in write binary mode.
3. Loop over the Response object's `iter_content()` method.
4. Call `write()` on each iteration to write the content to the file.
5. Call `close()` to close the file.

That's all there is to the requests module! The for loop and iter_content() stuff may seem complicated compared to the open()/write()/close() workflow you've been using to write text files, but it's to ensure that the requests module doesn't eat up too much memory even if you download massive files. You can learn about the requests module's other features from <https://requests.readthedocs.org/>.

HTML

Before you pick apart web pages, you'll learn some HTML basics. You'll also see how to access your web browser's powerful developer tools, which will make scraping information from the web much easier.

Resources for Learning HTML

Hypertext Markup Language (HTML) is the format that web pages are written in. This chapter assumes you have some basic experience with HTML, but if you need a beginner tutorial, I suggest one of the following sites:

- <https://developer.mozilla.org/en-US/learn/html/>
- <https://htmldog.com/guides/html/beginner/>
- <https://www.codecademy.com/learn/learn-html>

A Quick Refresher

In case it's been a while since you've looked at any HTML, here's a quick overview of the basics. An HTML file is a plaintext file with the .html file extension. The text in these files is surrounded by *tags*, which are words enclosed in angle brackets. The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an *element*. The *text* (or *inner HTML*) is the content between the starting and closing tags. For example, the following HTML will display *Hello, world!* in the browser, with *Hello* in bold:

```
<strong>Hello</strong>, world!
```

This HTML will look like [Figure 12-1](#) in a browser.

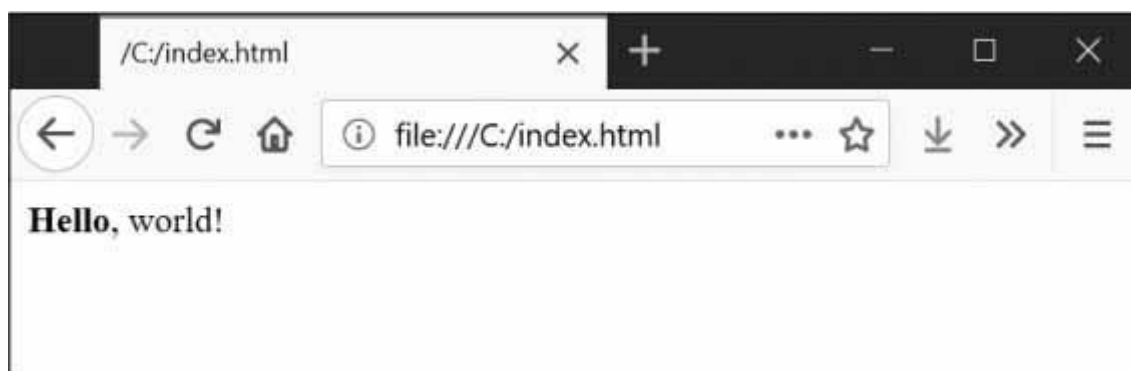


Figure 12-1: Hello, world! rendered in the browser

The opening `` tag says that the enclosed text will appear in bold. The closing `` tag tells the browser where the end of the bold text is.

There are many different tags in HTML. Some of these tags have extra properties in the form of *attributes* within the angle brackets. For example, the `<a>` tag encloses text that should be a link. The URL that the text links to is determined by the `href` attribute. Here's an example:

Al's free ``Python books``.

This HTML will look like [Figure 12-2](#) in a browser.

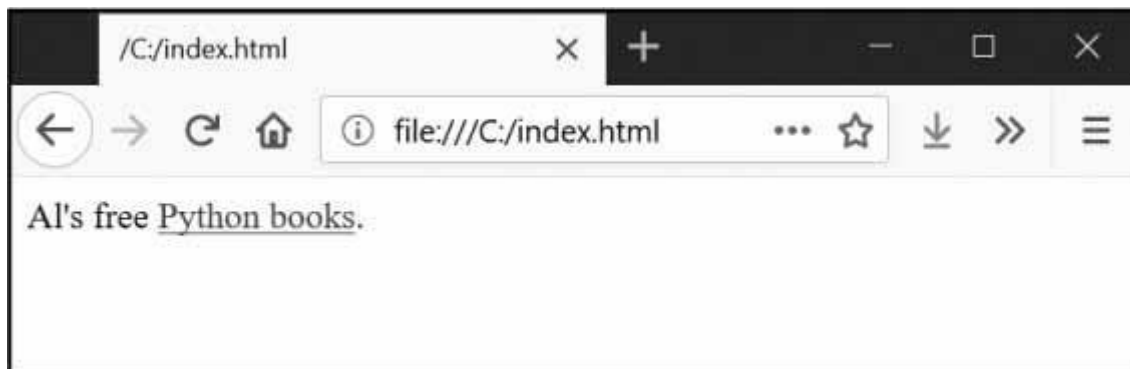


Figure 12-2: The link rendered in the browser

Some elements have an `id` attribute that is used to uniquely identify the element in the page. You will often instruct your programs to seek out an element by its `id` attribute, so figuring out an element's `id` attribute using the browser's developer tools is a common task in writing web scraping programs.

Viewing the Source HTML of a Web Page

You'll need to look at the HTML source of the web pages that your programs will work with. To do this, right-click (or CTRL-click on macOS) any web page in your web browser, and select **View Source** or **View page source** to see the HTML text of the page (see [Figure 12-3](#)). This is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.

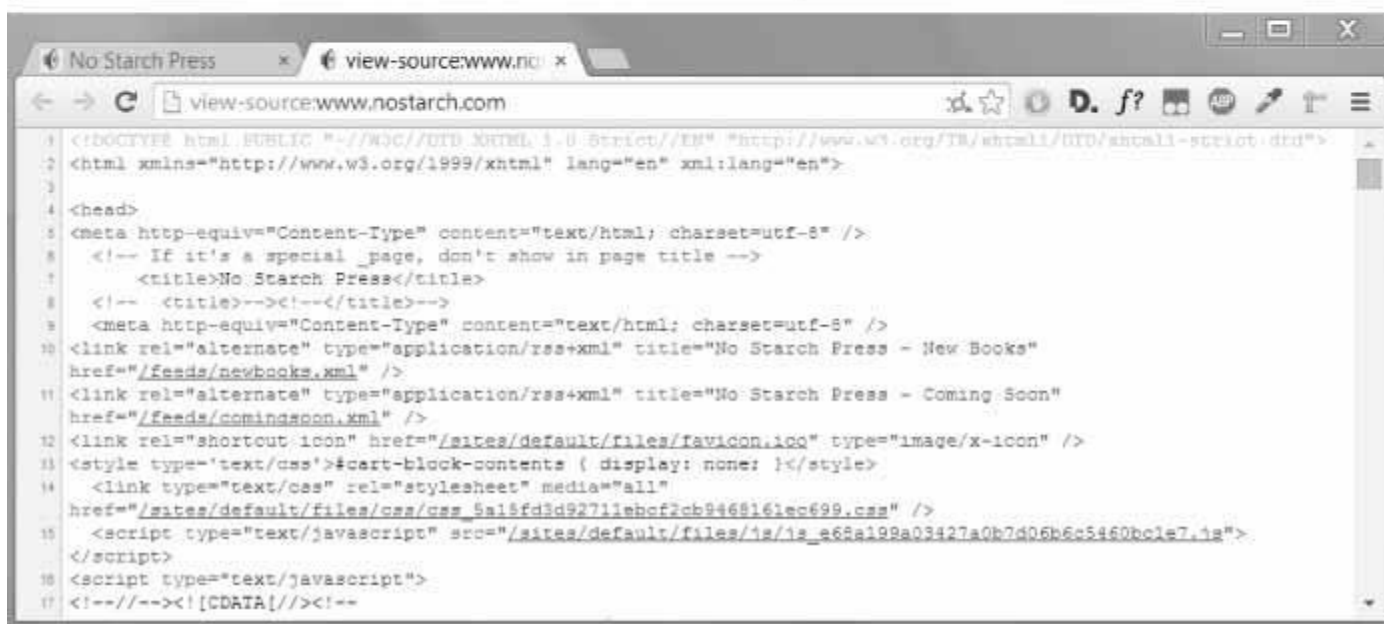


Figure 12-3: Viewing the source of a web page

I highly recommend viewing the source HTML of some of your favorite sites. It's fine if you don't fully understand what you are seeing when you look at the source. You won't need HTML mastery to write simple web scraping programs—after all, you won't be writing your own websites. You just need enough knowledge to pick out data from an existing site.

Opening Your Browser's Developer Tools

In addition to viewing a web page's source, you can look through a page's HTML using your browser's developer tools. In Chrome and Internet Explorer for Windows, the developer tools are already installed, and you can press F12 to make them appear (see [Figure 12-4](#)). Pressing

F12 again will make the developer tools disappear. In Chrome, you can also bring up the developer tools by selecting **View ▶ Developer ▶ Developer Tools**. In macOS, pressing **⌘-OPTION-I** will open Chrome's Developer Tools.



Figure 12-4: The Developer Tools window in the Chrome browser

In Firefox, you can bring up the Web Developer Tools Inspector by pressing **CTRL-SHIFT-C** on Windows and Linux or by pressing **⌘-OPTION-C** on macOS. The layout is almost identical to Chrome's developer tools.

In Safari, open the Preferences window, and on the Advanced pane check the **Show Develop menu in the menu bar** option. After it has been enabled, you can bring up the developer tools by pressing **⌘-OPTION-I**.

After enabling or installing the developer tools in your browser, you can right-click any part of the web page and select **Inspect Element** from the context menu to bring up the HTML responsible for that part of the page. This will be helpful when you begin to parse HTML for your web scraping programs.

DON'T USE REGULAR EXPRESSIONS TO PARSE HTML

Locating a specific piece of HTML in a string seems like a perfect case for regular expressions. However, I advise you against it. There are many different ways that HTML can be formatted and still be considered valid HTML, but trying to capture all these possible variations in a regular expression can be tedious and error prone. A module developed specifically for parsing HTML, such as bs4, will be less likely to result in bugs.

You can find an extended argument for why you shouldn't parse HTML with regular expressions at <https://stackoverflow.com/a/1732454/1893164/>.

Using the Developer Tools to Find HTML Elements

Once your program has downloaded a web page using the requests module, you will have the page's HTML content as a single string value. Now you need to figure out which part of the HTML corresponds to the information on the web page you're interested in.

This is where the browser's developer tools can help. Say you want to write a program to pull weather forecast data from <https://weather.gov/>. Before writing any code, do a little research. If you visit the site and search for the 94105 ZIP code, the site will take you to a page showing the forecast for that area.

What if you're interested in scraping the weather information for that ZIP code? Right-click where it is on the page (or CONTROL-click on macOS) and select **Inspect Element** from the context menu that appears. This will bring up the Developer Tools window, which shows you the HTML that produces this particular part of the web page. [Figure 12-5](#) shows the developer tools open to the HTML of the nearest forecast. Note that if the <https://weather.gov/> site changes the design of its web pages, you'll need to repeat this process to inspect the new elements.

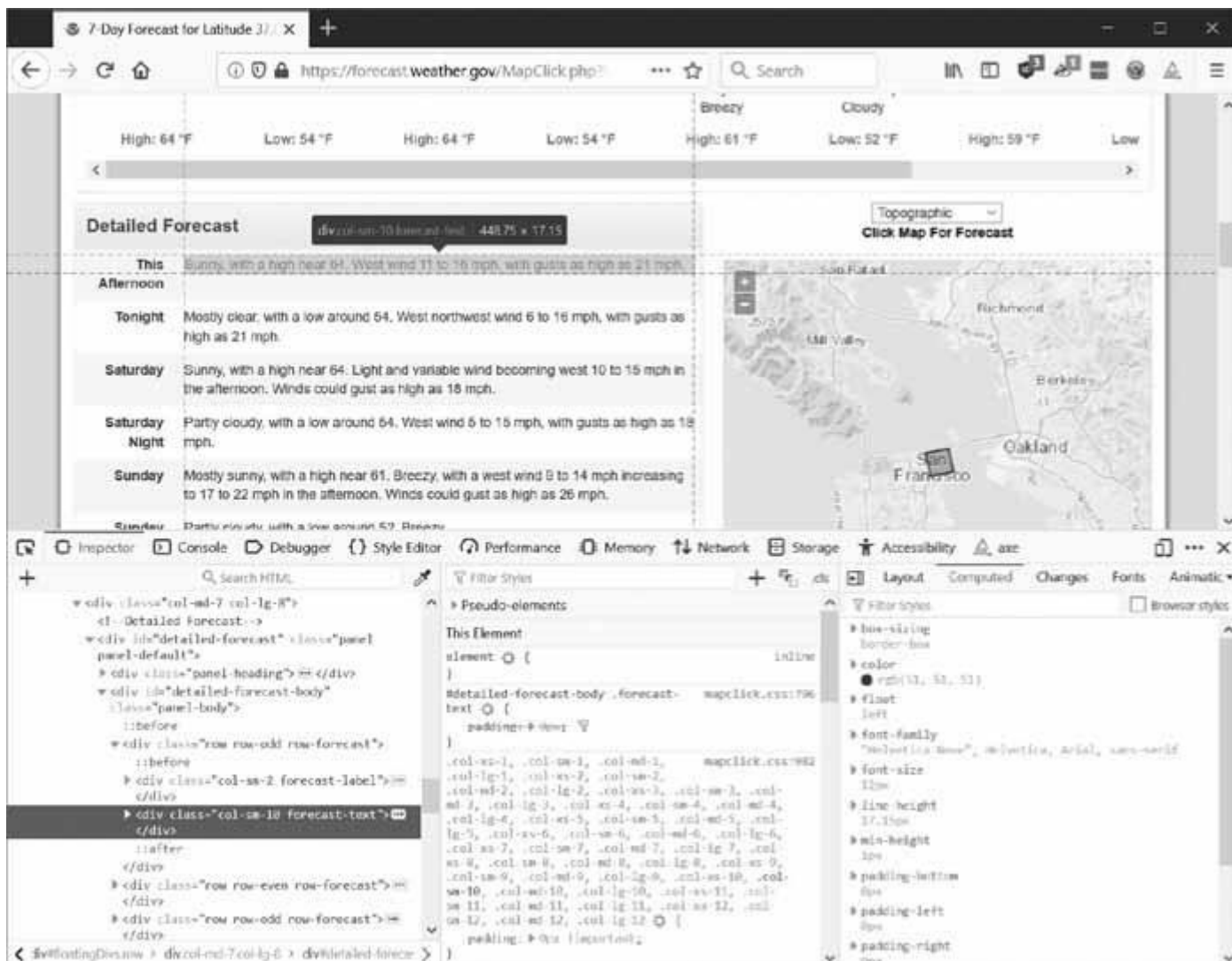


Figure 12-5: Inspecting the element that holds forecast text with the developer tools

From the developer tools, you can see that the HTML responsible for the forecast part of the web page is `<div class="col-sm-10 forecast-text">Sunny, with a high near 64. West wind 11 to 16 mph, with gusts as high as 21 mph.</div>`. This is exactly what you were looking for! It seems that the forecast information is contained inside a `<div>` element with the `forecast-text` CSS class. Right-click on this element in the browser's developer console, and from the context menu that appears, select **Copy ▶ CSS Selector**. This will copy a string such as `'div.row-odd:nth-child(1) > div:nth-child(2)'` to the clipboard. You can use this string for BeautifulSoup's `select()` or Selenium's `find_element_by_css_selector()` methods, as explained later in this chapter. Now that you know what you're looking for, the BeautifulSoup module will help you find it in the string.

PARSING HTML WITH THE BS4 MODULE

Beautiful Soup is a module for extracting information from an HTML page (and is much better for this purpose than regular expressions). The BeautifulSoup module's name is `bs4` (for BeautifulSoup, version 4). To install it, you will need to run `pip install --user beautifulsoup4` from the command line. (Check out [Appendix A](#) for instructions on installing third-party modules.) While `beautifulsoup4` is the name used for installation, to import BeautifulSoup you run `import bs4`.

For this chapter, the BeautifulSoup examples will *parse* (that is, analyze and identify the parts of) an HTML file on the hard drive. Open a new file editor tab in Mu, enter the following, and save it as `example.html`. Alternatively, download it from <https://nostarch.com/automatestuff2/>.

```
<!-- This is the example.html example file. -->

<html><head><title>The Website Title</title></head>
<body>
<p>Download my <strong>Python</strong> book from <a href="https://
inventwithpython.com">my website</a>.</p>
<p class="slogan">Learn Python the easy way!</p>
<p>By <span id="author">Al Sweigart</span></p>
</body></html>
```

As you can see, even a simple HTML file involves many different tags and attributes, and matters quickly get confusing with complex websites. Thankfully, BeautifulSoup makes working with HTML much easier.

Creating a BeautifulSoup Object from HTML

The `bs4.BeautifulSoup()` function needs to be called with a string containing the HTML it will parse. The `bs4.BeautifulSoup()` function returns a BeautifulSoup object. Enter the following into the interactive shell while your computer is connected to the internet:

```
>>> import requests, bs4
>>> res = requests.get('https://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text, 'html.parser')
>>> type(noStarchSoup)
<class 'bs4.BeautifulSoup'>
```

This code uses `requests.get()` to download the main page from the No Starch Press website and then passes the `text` attribute of the response to `bs4.BeautifulSoup()`. The `BeautifulSoup` object that it returns is stored in a variable named `noStarchSoup`.

You can also load an HTML file from your hard drive by passing a `File` object to `bs4.BeautifulSoup()` along with a second argument that tells Beautiful Soup which parser to use to analyze the HTML.

Enter the following into the interactive shell (after making sure the *example.html* file is in the working directory):

```
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile, 'html.parser')
>>> type(exampleSoup)
<class 'bs4.BeautifulSoup'>
```

The `'html.parser'` parser used here comes with Python. However, you can use the faster `'lxml'` parser if you install the third-party `lxml` module. Follow the instructions in [Appendix A](#) to install this module by running `pip install --user lxml`. Forgetting to include this second argument will result in a `UserWarning`: No parser was explicitly specified warning.

Once you have a `BeautifulSoup` object, you can use its methods to locate specific parts of an HTML document.

Finding an Element with the `select()` Method

You can retrieve a web page element from a `BeautifulSoup` object by calling the `select()` method and passing a string of a CSS *selector* for the element you are looking for. Selectors are like regular expressions: they specify a pattern to look for—in this case, in HTML pages instead of general text strings.

A full discussion of CSS selector syntax is beyond the scope of this book (there's a good selector tutorial in the resources at <https://nostarch.com/automatestuff2/>), but here's a short introduction to selectors. [Table 12-2](#) shows examples of the most common CSS selector patterns.

Table 12-2: Examples of CSS Selectors

Selector passed to the <code>select()</code> method	Will match . . .
--	------------------

Selector passed to the select() method	Will match . . .
soup.select('div')	All elements named <div>
soup.select('#author')	The element with an id attribute of author
soup.select('.notice')	All elements that use a CSS class attribute named notice
soup.select('div span')	All elements named that are within an element named <div>
soup.select('div > span')	All elements named that are <i>directly</i> within an element named <div>, with no other element in between
soup.select('input[name]')	All elements named <input> that have a name attribute with any value
soup.select('input[type="button"]')	All elements named <input> that have an attribute named type with value button

The various selector patterns can be combined to make sophisticated matches. For example, `soup.select('p #author')` will match any element that has an id attribute of author, as long as it is also inside a `<p>` element. Instead of writing the selector yourself, you can also right-click on the element in your browser and select **Inspect Element**. When the browser's developer console opens, right-click on the element's HTML and select **Copy ▶ CSS Selector** to copy the selector string to the clipboard and paste it into your source code.

The `select()` method will return a list of Tag objects, which is how BeautifulSoup represents an HTML element. The list will contain one Tag object for every match in the BeautifulSoup object's HTML. Tag values can be passed to the `str()` function to show the HTML tags they represent. Tag values also have an `attrs` attribute that shows all the HTML attributes of the tag as a dictionary. Using the `example.html` file from earlier, enter the following into the interactive shell:

```
>>> import bs4
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read(), 'html.parser')
>>> elems = exampleSoup.select('#author')
>>> type(elems) # elems is a list of Tag objects.
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> str(elems[0]) # The Tag object as a string.
'<span id="author">Al Sweigart</span>'
>>> elems[0].getText()
'Al Sweigart'
```

```
>>> elems[0].attrs  
{'id': 'author'}
```

This code will pull the element with `id="author"` out of our example HTML. We use `select('#author')` to return a list of all the elements with `id="author"`. We store this list of **Tag** objects in the variable `elems`, and `len(elems)` tells us there is one **Tag** object in the list; there was one match. Calling `getText()` on the element returns the element's text, or inner HTML. The text of an element is the content between the opening and closing tags: in this case, `'Al Sweigart'`.

Passing the element to `str()` returns a string with the starting and closing tags and the element's text. Finally, `attrs` gives us a dictionary with the element's attribute, `'id'`, and the value of the `id` attribute, `'author'`.

You can also pull all the `<p>` elements from the BeautifulSoup object. Enter this into the interactive shell:

```
>>> pElems = exampleSoup.select('p')  
>>> str(pElems[0])  
'<p>Download my <strong>Python</strong> book from <a href="https://  
inventwithpython.com">my website</a>.</p>'  
>>> pElems[0].getText()  
'Download my Python book from my website.'  
>>> str(pElems[1])  
'<p class="slogan">Learn Python the easy way!</p>'  
>>> pElems[1].getText()  
'Learn Python the easy way!'  
>>> str(pElems[2])  
'<p>By <span id="author">Al Sweigart</span></p>'  
>>> pElems[2].getText()  
'By Al Sweigart'
```

This time, `select()` gives us a list of three matches, which we store in `pElems`. Using `str()` on `pElems[0]`, `pElems[1]`, and `pElems[2]` shows you each element as a string, and using `getText()` on each element shows you its text.

Getting Data from an Element's Attributes

The `get()` method for Tag objects makes it simple to access attribute values from an element. The method is passed a string of an attribute name and returns that attribute's value. Using *example.html*, enter the following into the interactive shell:

```
>>> import bs4  
>>> soup = bs4.BeautifulSoup(open('example.html'), 'html.parser')  
>>> spanElem = soup.select('span')[0]
```

```
>>> str(spanElem)
'<span id="author">Al Sweigart</span>'
>>> spanElem.get('id')
'author'
>>> spanElem.get('some_nonexistent_addr') == None
True
>>> spanElem.attrs
{'id': 'author'}
```

Here we use `select()` to find any `` elements and then store the first matched element in `spanElem`. Passing the attribute name `'id'` to `get()` returns the attribute's value, `'author'`.

PROJECT: OPENING ALL SEARCH RESULTS

Whenever I search a topic on Google, I don't look at just one search result at a time. By middle-clicking a search result link (or clicking while holding CTRL), I open the first several links in a bunch of new tabs to read later. I search Google often enough that this workflow—opening my browser, searching for a topic, and middle-clicking several links one by one—is tedious. It would be nice if I could simply type a search term on the command line and have my computer automatically open a browser with all the top search results in new tabs. Let's write a script to do this with the search results page for the Python Package Index at <https://pypi.org/>. A program like this can be adapted to many other websites, although the Google and DuckDuckGo often employ measures that make scraping their search results pages difficult.

This is what your program does:

1. Gets search keywords from the command line arguments
2. Retrieves the search results page
3. Opens a browser tab for each result

This means your code will need to do the following:

1. Read the command line arguments from `sys.argv`.
2. Fetch the search result page with the `requests` module.
3. Find the links to each search result.
4. Call the `webbrowser.open()` function to open the web browser.

Open a new file editor tab and save it as *searchpypi.py*.

Step 1: Get the Command Line Arguments and Request the Search Page

Before coding anything, you first need to know the URL of the search result page. By looking at the browser's address bar after doing a search, you can see that the result page has a URL like `https://pypi.org/search/?q=<SEARCH_TERM_HERE>`. The `requests` module can download this page and then you can use Beautiful Soup to find the search result links in the HTML. Finally, you'll use the `webbrowser` module to open those links in browser tabs.

Make your code look like the following:

```
#!/ python3
# searchpypi.py - Opens several search results.

import requests, sys, webbrowser, bs4
print('Searching...') # display text while downloading the search result page
res = requests.get('https://google.com/search?q=' + 'https://pypi.org/search/?q='
+ ''.join(sys.argv[1:]))
res.raise_for_status()

# TODO: Retrieve top search result links.

# TODO: Open a browser tab for each result.
```

The user will specify the search terms using command line arguments when they launch the program. These arguments will be stored as strings in a list in `sys.argv`.

Step 2: Find All the Results

Now you need to use Beautiful Soup to extract the top search result links from your downloaded HTML. But how do you figure out the right selector for the job? For example, you can't just search for all `<a>` tags, because there are lots of links you don't care about in the HTML. Instead, you must inspect the search result page with the browser's developer tools to try to find a selector that will pick out only the links you want.

After doing a search for *Beautiful Soup*, you can open the browser's developer tools and inspect some of the link elements on the page. They can look complicated, something like pages of this: `project/xml-parser/">`.

It doesn't matter that the element looks incredibly complicated. You just need to find the pattern that all the search result links have.

Make your code look like the following:

```
#!/ python3
# searchpypi.py - Opens several google results.
import requests, sys, webbrowser, bs4
--snip--
# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text, 'html.parser')
# Open a browser tab for each result.
linkElems = soup.select('.package-snippet')
```

If you look at the `<a>` elements, though, the search result links all have `class="package-snippet"`. Looking through the rest of the HTML source, it looks like the `package-snippet` class is used only for search result links. You don't have to know what the CSS

class package-snippet is or what it does. You're just going to use it as a marker for the <a> element you are looking for. You can create a BeautifulSoup object from the downloaded page's HTML text and then use the selector '.package-snippet' to find all <a> elements that are within an element that has the package-snippet CSS class. Note that if the PyPI website changes its layout, you may need to update this program with a new CSS selector string to pass to soup.select(). The rest of the program will still be up to date.

Step 3: Open Web Browsers for Each Result

Finally, we'll tell the program to open web browser tabs for our results. Add the following to the end of your program:

```
#!/python3
# searchpypi.py - Opens several search results.
import requests, sys, webbrowser, bs4
--snip--
# Open a browser tab for each result.
linkElems = soup.select('.package-snippet')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    urlToOpen = 'https://pypi.org' + linkElems[i].get('href')
    print('Opening', urlToOpen)
    webbrowser.open(urlToOpen)
```

By default, you open the first five search results in new tabs using the webbrowser module. However, the user may have searched for something that turned up fewer than five results. The soup.select() call returns a list of all the elements that matched your '.package-snippet' selector, so the number of tabs you want to open is either 5 or the length of this list (whichever is smaller).

The built-in Python function min() returns the smallest of the integer or float arguments it is passed. (There is also a built-in max() function that returns the largest argument it is passed.) You can use min() to find out whether there are fewer than five links in the list and store the number of links to open in a variable named numOpen. Then you can run through a for loop by calling range(numOpen).

On each iteration of the loop, you use webbrowser.open() to open a new tab in the web browser. Note that the href attribute's value in the returned <a> elements do not have the initial https://pypi.org part, so you have to concatenate that to the href attribute's string value.

Now you can instantly open the first five PyPI search results for, say, *boring stuff* by running searchpypi boring stuff on the command line! (See [Appendix B](#) for how to easily run programs on your operating system.)

Ideas for Similar Programs

The benefit of tabbed browsing is that you can easily open links in new tabs to peruse later. A program that automatically opens several links at once can be a nice shortcut to do the following:

- Open all the product pages after searching a shopping site such as Amazon.
- Open all the links to reviews for a single product.
- Open the result links to photos after performing a search on a photo site such as Flickr or Imgur.

PROJECT: DOWNLOADING ALL XKCD COMICS

Blogs and other regularly updating websites usually have a front page with the most recent post as well as a Previous button on the page that takes you to the previous post. Then that post will also have a Previous button, and so on, creating a trail from the most recent page to the first post on the site. If you wanted a copy of the site's content to read when you're not online, you could manually navigate over every page and save each one. But this is pretty boring work, so let's write a program to do it instead.

XKCD is a popular geek webcomic with a website that fits this structure (see Figure 12-6). The front page at <https://xkcd.com/> has a Prev button that guides the user back through prior comics. Downloading each comic by hand would take forever, but you can write a script to do this in a couple of minutes.

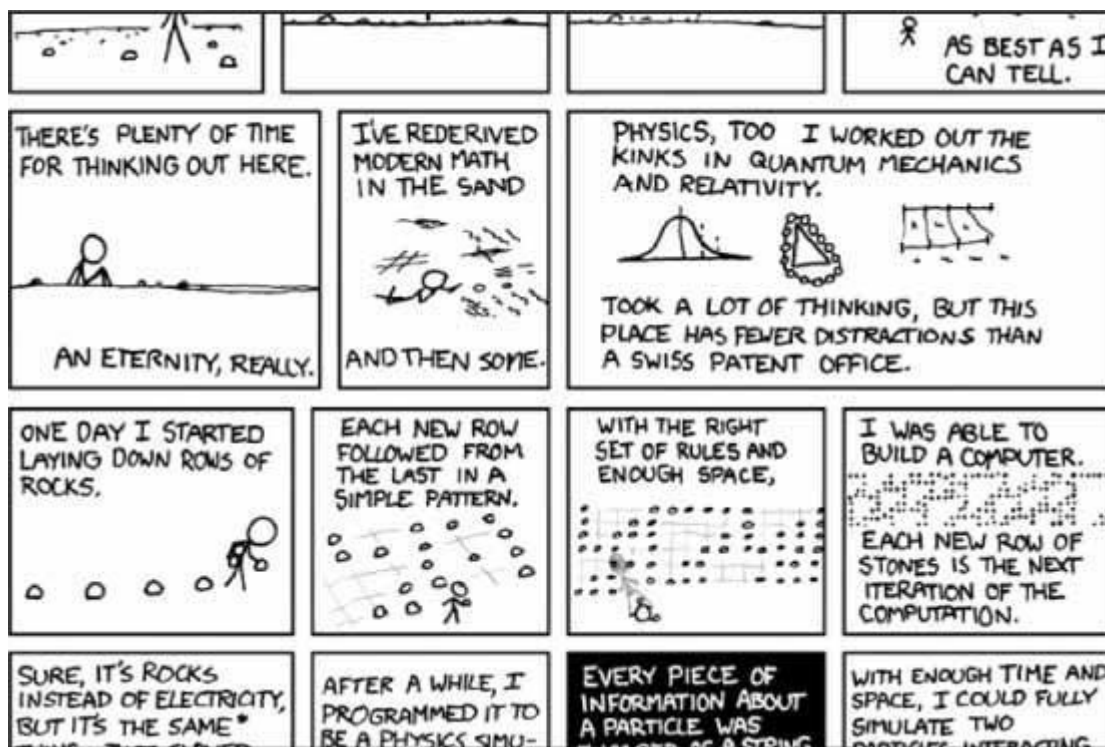


Figure 12-6: XKCD, “a webcomic of romance, sarcasm, math, and language”

Here's what your program does:

1. Loads the XKCD home page
2. Saves the comic image on that page

3. Follows the Previous Comic link
4. Repeats until it reaches the first comic

This means your code will need to do the following:

1. Download pages with the requests module.
2. Find the URL of the comic image for a page using BeautifulSoup.
3. Download and save the comic image to the hard drive with `iter_content()`.
4. Find the URL of the Previous Comic link, and repeat.

Open a new file editor tab and save it as *downloadXkcd.py*.

Step 1: Design the Program

If you open the browser's developer tools and inspect the elements on the page, you'll find the following:

- The URL of the comic's image file is given by the href attribute of an `` element.
- The `` element is inside a `<div id="comic">` element.
- The Prev button has a rel HTML attribute with the value prev.
- The first comic's Prev button links to the <https://xkcd.com/#> URL, indicating that there are no more previous pages.

Make your code look like the following:

```
#!/python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'https://xkcd.com'          # starting url
os.makedirs('xkcd', exist_ok=True) # store comics in ./xkcd
while not url.endswith('#'):
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

You'll have a url variable that starts with the value 'https://xkcd.com' and repeatedly update it (in a for loop) with the URL of the current page's Prev link. At every step in the loop, you'll download the comic at url. You'll know to end the loop when url ends with '#'.

You will download the image files to a folder in the current working directory named *xkcd*. The call `os.makedirs()` ensures that this folder exists, and the `exist_ok=True` keyword argument prevents the function from throwing an exception if this folder already exists. The remaining code is just comments that outline the rest of your program.

Step 2: Download the Web Page

Let's implement the code for downloading the page. Make your code look like the following:

```
#!/ python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'https://xkcd.com'          # starting url
os.makedirs('xkcd', exist_ok=True) # store comics in ./xkcd
while not url.endswith('#'):
    # Download the page.
    print('Downloading page %s...' % url)
    res = requests.get(url)
    res.raise_for_status()

    soup = bs4.BeautifulSoup(res.text, 'html.parser')

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

First, print `url` so that the user knows which URL the program is about to download; then use the `requests` module's `request.get()` function to download it. As always, you immediately call the `Response` object's `raise_for_status()` method to throw an exception and end the program if something went wrong with the download. Otherwise, you create a `BeautifulSoup` object from the text of the downloaded page.

Step 3: Find and Download the Comic Image

Make your code look like the following:

```
#!/python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

# Find the URL of the comic image.
comicElem = soup.select('#comic img')
if comicElem == []:
    print('Could not find comic image.')
else:
    comicUrl = 'https:' + comicElem[0].get('src')
    # Download the image.
    print('Downloading image %s...' % (comicUrl))
    res = requests.get(comicUrl)
    res.raise_for_status()

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

From inspecting the XKCD home page with your developer tools, you know that the `` element for the comic image is inside a `<div>` element with the `id` attribute set to `comic`, so the selector `'#comic img'` will get you the correct `` element from the BeautifulSoup object.

A few XKCD pages have special content that isn't a simple image file. That's fine; you'll just skip those. If your selector doesn't find any elements, then `soup.select('#comic img')` will return a blank list. When that happens, the program can just print an error message and move on without downloading the image.

Otherwise, the selector will return a list containing one `` element. You can get the `src` attribute from this `` element and pass it to `requests.get()` to download the comic's image file.

Step 4: Save the Image and Find the Previous Comic

Make your code look like the following:

```
#!/python3
# downloadXkcd.py - Downloads every single XKCD comic.
```

```
import requests, os, bs4
```

```
--snip--
```

```
    # Save the image to ./xkcd.
    imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)),
'wb')
    for chunk in res.iter_content(100000):
        imageFile.write(chunk)
    imageFile.close()

# Get the Prev button's url.
prevLink = soup.select('a[rel="prev"]')[0]
url = 'https://xkcd.com' + prevLink.get('href')

print('Done.')
```

At this point, the image file of the comic is stored in the `res` variable. You need to write this image data to a file on the hard drive.

You'll need a filename for the local image file to pass to `open()`. The `comicUrl` will have a value like `'https://imgs.xkcd.com/comics/heartbleed_explanation.png'`—which you might have noticed looks a lot like a file path. And in fact, you can call `os.path.basename()` with `comicUrl`, and it will return just the last part of the URL, `'heartbleed_explanation.png'`. You can use this as the filename when saving the image to your hard drive. You join this name with the name of your xkcd folder using `os.path.join()` so that your program uses backslashes (`\`) on Windows and forward slashes (`/`) on macOS and Linux. Now that you finally have the filename, you can call `open()` to open a new file in `'wb'` “write binary” mode.

Remember from earlier in this chapter that to save files you've downloaded using `requests`, you need to loop over the return value of the `iter_content()` method. The code in the `for` loop writes out chunks of the image data (at most 100,000 bytes each) to the file and then you close the file. The image is now saved to your hard drive.

Afterward, the selector `'a[rel="prev"]'` identifies the `<a>` element with the `rel` attribute set to `prev`, and you can use this `<a>` element's `href` attribute to get the previous comic's URL, which gets stored in `url`. Then the `while` loop begins the entire download process again for this comic.

The output of this program will look like this:

```
Downloading page https://xkcd.com...
Downloading image https://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page https://xkcd.com/1358/...
Downloading image https://imgs.xkcd.com/comics/nro.png...
```

Downloading page <https://xkcd.com/1357/...>
Downloading image https://imgs.xkcd.com/comics/free_speech.png...
Downloading page <https://xkcd.com/1356/...>
Downloading image https://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page <https://xkcd.com/1355/...>
Downloading image https://imgs.xkcd.com/comics/airplane_message.png...
Downloading page <https://xkcd.com/1354/...>
Downloading image https://imgs.xkcd.com/comics/heartbleed_explanation.png...
--snip--

This project is a good example of a program that can automatically follow links in order to scrape large amounts of data from the web. You can learn about BeautifulSoup's other features from its documentation at <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

Ideas for Similar Programs

Downloading pages and following links are the basis of many web crawling programs. Similar programs could also do the following:

- Back up an entire site by following all of its links.
- Copy all the messages off a web forum.
- Duplicate the catalog of items for sale on an online store.

The requests and bs4 modules are great as long as you can figure out the URL you need to pass to requests.get(). However, sometimes this isn't so easy to find. Or perhaps the website you want your program to navigate requires you to log in first. The selenium module will give your programs the power to perform such sophisticated tasks.

CONTROLLING THE BROWSER WITH THE SELENIUM MODULE

The selenium module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there were a human user interacting with the page. Using selenium, you can interact with web pages in a much more advanced way than with requests and bs4; but because it launches a web browser, it is a bit slower and hard to run in the background if, say, you just need to download some files from the web.

Still, if you need to interact with a web page in a way that, say, depends on the JavaScript code that updates the page, you'll need to use selenium instead of requests. That's because major ecommerce websites such as Amazon almost certainly have software systems to recognize traffic that they suspect is a script harvesting their info or signing up for multiple free accounts. These sites may refuse to serve pages to you after a while, breaking any scripts you've made. The selenium module is much more likely to function on these sites long-term than requests.

A major “tell” to websites that you’re using a script is the *user-agent* string, which identifies the web browser and is included in all HTTP requests. For example, the user-agent string for the requests module is something like 'python-requests/2.21.0'. You can visit a site such as <https://www.whatsmyua.info/> to see your user-agent string. Using selenium, you’re much more likely to “pass for human” because not only is Selenium’s user-agent is the same as a regular browser (for instance, 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:65.0) Gecko/20100101 Firefox/65.0'), but it has the same traffic patterns: a selenium-controlled browser will download images, advertisements, cookies, and privacy-invading trackers just like a regular browser. However, selenium can still be detected by websites, and major ticketing and ecommerce websites often block browsers controlled by selenium to prevent web scraping of their pages.

Starting a selenium-Controlled Browser

The following examples will show you how to control Firefox’s web browser. If you don’t already have Firefox, you can download it for free from <https://getfirefox.com/>. You can install selenium by running pip install --user selenium from a command line terminal. More information is available in [Appendix A](#).

Importing the modules for selenium is slightly tricky. Instead of import selenium, you need to run from selenium import webdriver. (The exact reason why the selenium module is set up this way is beyond the scope of this book.) After that, you can launch the Firefox browser with selenium. Enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> type(browser)
<class 'selenium.webdriver.firefox.webdriver.WebDriver'>
>>> browser.get('https://inventwithpython.com')
```

You’ll notice when webdriver.Firefox() is called, the Firefox web browser starts up. Calling type() on the value webdriver.Firefox() reveals it’s of the WebDriver data type. And calling browser.get('https://inventwithpython.com') directs the browser to <https://inventwithpython.com/>. Your browser should look something like [Figure 12-7](#).

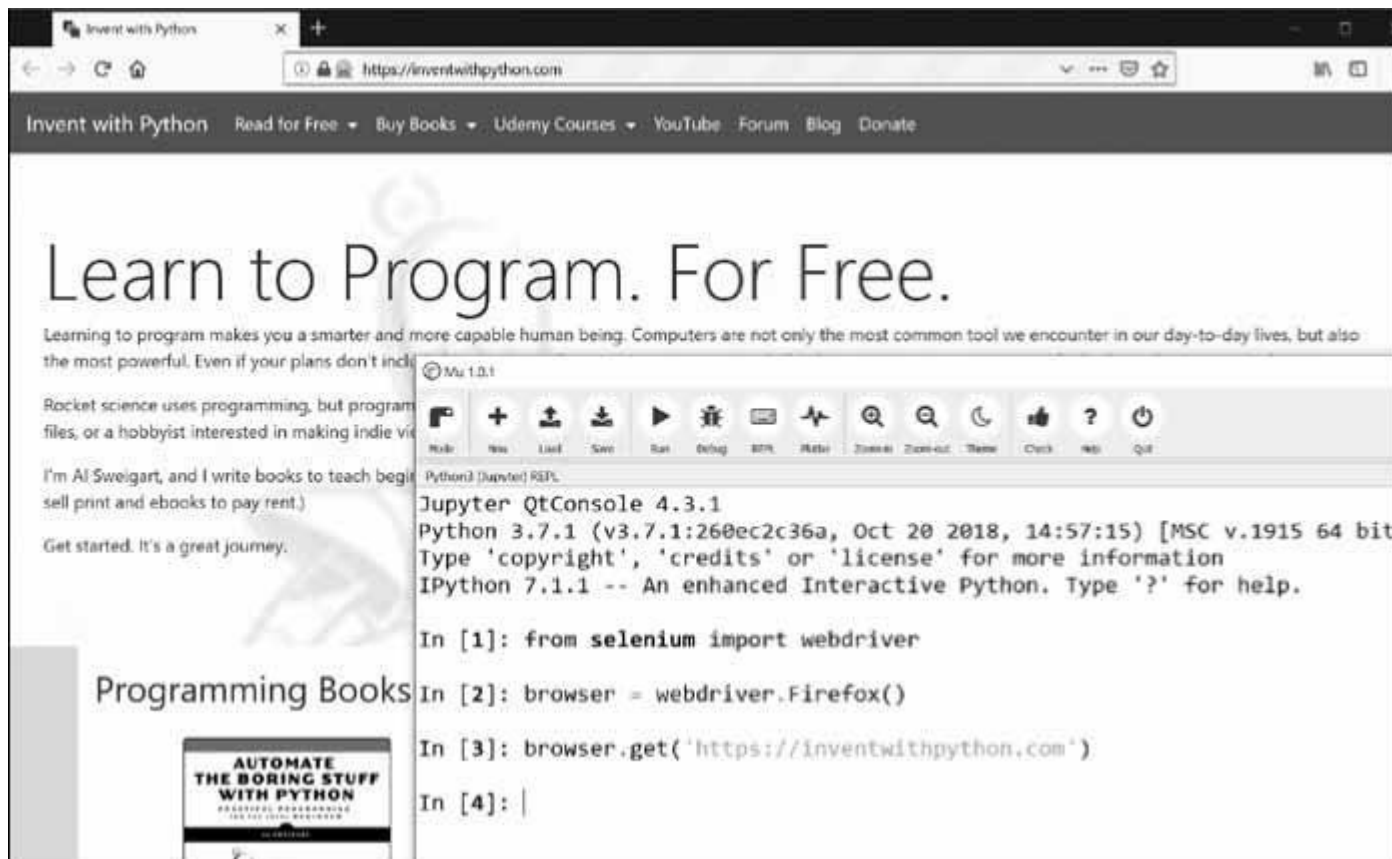


Figure 12-7: After we call `webdriver.Firefox()` and `get()` in Mu, the Firefox browser appears.

If you encounter the error message “‘geckodriver’ executable needs to be in PATH.”, then you need to manually download the webdriver for Firefox before you can use selenium to control it. You can also control browsers other than Firefox if you install the webdriver for them.

For Firefox, go to <https://github.com/mozilla/geckodriver/releases> and download the geckodriver for your operating system. (“Gecko” is the name of the browser engine used in Firefox.) For example, on Windows you’ll want to download the `geckodriver-v0.24.0-win64.zip` link, and on macOS, you’ll want the `geckodriver-v0.24.0-macos.tar.gz` link. Newer versions will have slightly different links. The downloaded ZIP file will contain a `geckodriver.exe` (on Windows) or `geckodriver` (on macOS and Linux) file that you can put on your system PATH. [Appendix B](#) has information about the system PATH, or you can learn more at <https://stackoverflow.com/q/40208051/1893164>.

For Chrome, go to <https://sites.google.com/a/chromium.org/chromedriver/downloads> and download the ZIP file for your operating system. This ZIP file will contain a `chromedriver.exe` (on Windows) or `chromedriver` (on macOS or Linux) file that you can put on your system PATH.

Other major web browsers also have webdrivers available, and you can often find these by performing an internet search for “<browser name> webdriver”.

If you still have problems opening up a new browser under the control of selenium, it may be because the current version of the browser is incompatible with the selenium module. One

workaround is to install an older version of the web browser—or, more simply, an older version of the selenium module. You can find the list of selenium version numbers at <https://pypi.org/project/selenium/#history>. Unfortunately, the compatibility between versions of selenium and a browser sometimes breaks, and you may need to search the web for possible solutions. [Appendix A](#) has more information about running pip to install a specific version of selenium. (For example, you might run `pip install --user -U selenium==3.14.1`.)

Finding Elements on the Page

WebDriver objects have quite a few methods for finding elements on a page. They are divided into the `find_element_*` and `find_elements_*` methods. The `find_element_*` methods return a single `WebElement` object, representing the first element on the page that matches your query. The `find_elements_*` methods return a list of `WebElement_*` objects for *every* matching element on the page.

[Table 12-3](#) shows several examples of `find_element_*` and `find_elements_*` methods being called on a `WebDriver` object that's stored in the variable `browser`.

Table 12-3: Selenium's WebDriver Methods for Finding Elements

Method name	WebElement object/list returned
<code>browser.find_element_by_class_name(name)</code>	Elements that use the CSS
<code>browser.find_elements_by_class_name(name)</code>	class <i>name</i>
<code>browser.find_element_by_css_selector(selector)</code>	Elements that match the CSS
<code>browser.find_elements_by_css_selector(selector)</code>	<i>selector</i>
<code>browser.find_element_by_id(id)</code>	Elements with a matching <i>id</i>
<code>browser.find_elements_by_id(id)</code>	attribute value
<code>browser.find_element_by_link_text(text)</code>	<a> elements that completely
<code>browser.find_elements_by_link_text(text)</code>	match the <i>text</i> provided
<code>browser.find_element_by_partial_link_text(text)</code>	<a> elements that contain the
<code>browser.find_elements_by_partial_link_text(text)</code>	<i>text</i> provided
<code>browser.find_element_by_name(name)</code>	Elements with a matching <i>name</i>
<code>browser.find_elements_by_name(name)</code>	attribute value
<code>browser.find_element_by_tag_name(name)</code>	Elements with a matching tag <i>name</i>
<code>browser.find_elements_by_tag_name(name)</code>	(case-insensitive; an <a> element is matched by 'a' and 'A')

Except for the `*_by_tag_name()` methods, the arguments to all the methods are case sensitive. If no elements exist on the page that match what the method is looking for, the selenium module raises a `NoSuchElementException`. If you do not want this exception to crash your program, add try and except statements to your code.

Once you have the WebElement object, you can find out more about it by reading the attributes or calling the methods in [Table 12-4](#).

Table 12-4: WebElement Attributes and Methods

Attribute or method	Description
tag_name	The tag name, such as 'a' for an <a> element
get_attribute(name)	The value for the element's name attribute
text	The text within the element, such as 'hello' in hello
clear()	For text field or text area elements, clears the text typed into it
is_displayed()	Returns True if the element is visible; otherwise returns False
is_enabled()	For input elements, returns True if the element is enabled; otherwise returns False
is_selected()	For checkbox or radio button elements, returns True if the element is selected; otherwise returns False
location	A dictionary with keys 'x' and 'y' for the position of the element in the page

For example, open a new file editor tab and enter the following program:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('https://inventwithpython.com')
try:
    elem = browser.find_element_by_class_name('cover-thumb')
    print('Found <%s> element with that class name!' % (elem.tag_name))
except:
    print('Was not able to find an element with that name.')
```

Here we open Firefox and direct it to a URL. On this page, we try to find elements with the class name 'bookcover', and if such an element is found, we print its tag name using the tag_name attribute. If no such element was found, we print a different message.

This program will output the following:

```
Found <img> element with that class name!
```

We found an element with the class name 'bookcover' and the tag name 'img'.

Clicking the Page

WebElement objects returned from the find_element_* and find_elements_* methods have a click() method that simulates a mouse click on that element. This method can be used to

follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('https://inventwithpython.com')
>>> linkElem = browser.find_element_by_link_text('Read Online for Free')
>>> type(linkElem)
<class 'selenium.webdriver.remote.webelement.FirefoxWebElement'>
>>> linkElem.click() # follows the "Read Online for Free" link
```

This opens Firefox to <https://inventwithpython.com/>, gets the WebElement object for the <a> element with the text *Read It Online*, and then simulates clicking that <a> element. It's just like if you clicked the link yourself; the browser then follows that link.

Filling Out and Submitting Forms

Sending keystrokes to text fields on a web page is a matter of finding the <input> or <textarea> element for that text field and then calling the send_keys() method. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('https://login.metafilter.com')
>>> userElem = browser.find_element_by_id('user_name')
>>> userElem.send_keys('your_real_username_here')

>>> passwordElem = browser.find_element_by_id('user_pass')
>>> passwordElem.send_keys('your_real_password_here')
>>> passwordElem.submit()
```

As long as login page for MetaFilter hasn't changed the id of the Username and Password text fields since this book was published, the previous code will fill in those text fields with the provided text. (You can always use the browser's inspector to verify the id.) Calling the submit() method on any element will have the same result as clicking the Submit button for the form that element is in. (You could have just as easily called emailElem.submit(), and the code would have done the same thing.)

WARNING

Avoid putting your passwords in source code whenever possible. It's easy to accidentally leak your passwords to others when they are left unencrypted on your hard drive. If possible, have your program prompt users to enter their passwords from the keyboard using the pyinputplus.inputPassword() function described in [Chapter 8](#).

Sending Special Keys

The selenium module has a module for keyboard keys that are impossible to type into a string value, which function much like escape characters. These values are stored in attributes in the selenium.webdriver.common.keys module. Since that is such a long module name, it's much easier to run from selenium.webdriver.common.keys import Keys at the top of your program; if you do, then you can simply write Keys anywhere you'd normally have to write selenium.webdriver.common.keys. [Table 12-5](#) lists the commonly used Keys variables.

Table 12-5: Commonly Used Variables in the selenium.webdriver.common.keys Module

Attributes	Meanings
Keys.DOWN, Keys.UP, Keys.LEFT, Keys.RIGHT	The keyboard arrow keys
Keys.ENTER, Keys.RETURN	The ENTER and RETURN keys
Keys.HOME, Keys.END, Keys.PAGE_DOWN, Keys.PAGE_UP	The HOME, END, PAGEDOWN, and PAGEUP keys
Keys.ESCAPE, Keys.BACK_SPACE, Keys.DELETE	The ESC, BACKSPACE, and DELETE keys
Keys.F1, Keys.F2, . . . , Keys.F12	The F1 to F12 keys at the top of the keyboard
Keys.TAB	The TAB key

For example, if the cursor is not currently in a text field, pressing the HOME and END keys will scroll the browser to the top and bottom of the page, respectively. Enter the following into the interactive shell, and notice how the send_keys() calls scroll the page:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.keys import Keys
>>> browser = webdriver.Firefox()
>>> browser.get('https://nostarch.com')
>>> htmlElem = browser.find_element_by_tag_name('html')
>>> htmlElem.send_keys(Keys.END)    # scrolls to bottom
>>> htmlElem.send_keys(Keys.HOME)   # scrolls to top
```

The <html> tag is the base tag in HTML files: the full content of the HTML file is enclosed within the <html> and </html> tags. Calling browser.find_element_by_tag_name('html') is a good place to send keys to the general web page. This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Clicking Browser Buttons

The selenium module can simulate clicks on various browser buttons as well through the following methods:

browser.back() Clicks the Back button.

browser.forward() Clicks the Forward button.

browser.refresh() Clicks the Refresh/Reload button.

browser.quit() Clicks the Close Window button.

More Information on Selenium

Selenium can do much more beyond the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript. To learn more about these features, you can visit the selenium documentation at <https://selenium-python.readthedocs.org/>.

SUMMARY

Most boring tasks aren't limited to the files on your computer. Being able to programmatically download web pages will extend your programs to the internet. The requests module makes downloading straightforward, and with some basic knowledge of HTML concepts and selectors, you can utilize the BeautifulSoup module to parse the pages you download.

But to fully automate any web-based tasks, you need direct control of your web browser through the selenium module. The selenium module will allow you to log in to websites and fill out forms automatically. Since a web browser is the most common way to send and receive information over the internet, this is a great ability to have in your programmer toolkit.

PRACTICE QUESTIONS

1. Briefly describe the differences between the webbrowser, requests, bs4, and selenium modules.
2. What type of object is returned by requests.get()? How can you access the downloaded content as a string value?
3. What requests method checks that the download worked?
4. How can you get the HTTP status code of a requests response?
5. How do you save a requests response to a file?
6. What is the keyboard shortcut for opening a browser's developer tools?
7. How can you view (in the developer tools) the HTML of a specific element on a web page?
8. What is the CSS selector string that would find the element with an id attribute of main?

9. What is the CSS selector string that would find the elements with a CSS class of highlight?
10. What is the CSS selector string that would find all the <div> elements inside another <div> element?
11. What is the CSS selector string that would find the <button> element with a value attribute set to favorite?
12. Say you have a BeautifulSoup Tag object stored in the variable spam for the element <div>Hello, world!</div>. How could you get a string 'Hello, world!' from the Tag object?
13. How would you store all the attributes of a BeautifulSoup Tag object in a variable named linkElem?
14. Running import selenium doesn't work. How do you properly import the selenium module?
15. What's the difference between the find_element_* and find_elements_* methods?
16. What methods do Selenium's WebElement objects have for simulating mouse clicks and keyboard keys?
17. You could call send_keys(Keys.ENTER) on the Submit button's WebElement object, but what is an easier way to submit a form with selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with selenium?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Command Line Emailer

Write a program that takes an email address and string of text on the command line and then, using selenium, logs in to your email account and sends an email of the string to the provided address. (You might want to set up a separate email account for this program.)

This would be a nice way to add a notification feature to your programs. You could also write a similar program to send messages from a Facebook or Twitter account.

Image Site Downloader

Write a program that goes to a photo-sharing site like Flickr or Imgur, searches for a category of photos, and then downloads all the resulting images. You could write a program that works with any photo site that has a search feature.

2048

2048 is a simple game where you combine tiles by sliding them up, down, left, or right with the arrow keys. You can actually get a fairly high score by repeatedly sliding in an up, right, down, and left pattern over and over again. Write a program that will open the game

at <https://gabrielecirulli.github.io/2048/> and keep sending up, right, down, and left keystrokes to automatically play the game.

Link Verification

Write a program that, given the URL of a web page, will attempt to download every linked page on the page. The program should flag any pages that have a 404 “Not Found” status code and print them out as broken links.

WORKING WITH EXCEL SPREADSHEETS

Although we don’t often think of spreadsheets as programming tools, almost everyone uses them to organize information into two-dimensional data structures, perform calculations with formulas, and produce output as charts. In the next two chapters, we’ll integrate Python into two popular spreadsheet applications: Microsoft Excel and Google Sheets.

Excel is a popular and powerful spreadsheet application for Windows. The `openpyxl` module allows your Python programs to read and modify Excel spreadsheet files. For example, you might have the boring task of copying certain data from one spreadsheet and pasting it into another one. Or you might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria. Or you might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red. These are exactly the sort of boring, mindless spreadsheet tasks that Python can do for you.

Although Excel is proprietary software from Microsoft, there are free alternatives that run on Windows, macOS, and Linux. Both LibreOffice Calc and OpenOffice Calc work with Excel’s `.xlsx` file format for spreadsheets, which means the `openpyxl` module can work on spreadsheets from these applications as well. You can download the software from <https://www.libreoffice.org/> and <https://www.openoffice.org/>, respectively. Even if you already have Excel installed on your computer, you may find these programs easier to use. The screenshots in this chapter, however, are all from Excel 2010 on Windows 10.

EXCEL DOCUMENTS

First, let’s go over some basic definitions: an Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the `.xlsx` extension. Each workbook can contain multiple *sheets* (also called *worksheets*). The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*.

Each sheet has *columns* (addressed by letters starting at A) and *rows* (addressed by numbers starting at 1). A box at a particular column and row is called a *cell*. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.

INSTALLING THE OPENPYXL MODULE

Python does not come with OpenPyXL, so you'll have to install it. Follow the instructions for installing third-party modules in [Appendix A](#); the name of the module is `openpyxl`.

This book uses version 2.6.2 of OpenPyXL. It's important that you install this version by running `pip install --user -U openpyxl==2.6.2` because newer versions of OpenPyXL are incompatible with the information in this book. To test whether it is installed correctly, enter the following into the interactive shell:

```
>>> import openpyxl
```

If the module was correctly installed, this should produce no error messages. Remember to import the `openpyxl` module before running the interactive shell examples in this chapter, or you'll get a `NameError: name 'openpyxl' is not defined` error.

You can find the full documentation for OpenPyXL at <https://openpyxl.readthedocs.org/>.

READING EXCEL DOCUMENTS

The examples in this chapter will use a spreadsheet named *example.xlsx* stored in the root folder. You can either create the spreadsheet yourself or download it from <https://nostarch.com/automatestuff2/>. [Figure 13-1](#) shows the tabs for the three default sheets named *Sheet1*, *Sheet2*, and *Sheet3* that Excel automatically provides for new workbooks. (The number of default sheets created may vary between operating systems and spreadsheet programs.)

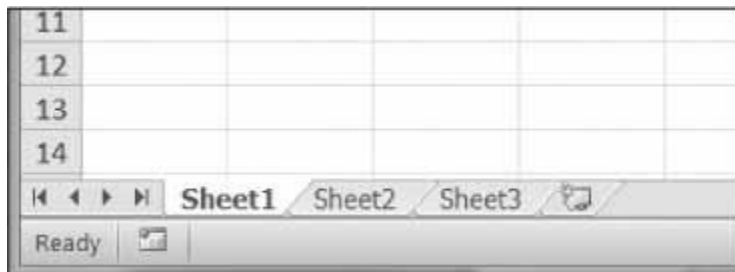


Figure 13-1: The tabs for a workbook's sheets are in the lower-left corner of Excel.

Sheet 1 in the example file should look like [Table 13-1](#). (If you didn't download *example.xlsx* from the website, you should enter this data into the sheet yourself.)

Table 13-1: The *example.xlsx* Spreadsheet

A	B	C
14/5/2015 1:34:02 PM	Apples	73
24/5/2015 3:41:23 AM	Cherries	85
34/6/2015 12:46:51 PM	Pears	14
44/8/2015 8:59:43 AM	Oranges	52

54/10/2015 2:07:00 AM Apples 152

64/10/2015 6:10:37 PM Bananas 23

74/10/2015 2:40:46 AM Strawberries 98

Now that we have our example spreadsheet, let's see how we can manipulate it with the openpyxl module.

Opening Excel Documents with OpenPyXL

Once you've imported the openpyxl module, you'll be able to use the openpyxl.load_workbook() function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

The openpyxl.load_workbook() function takes in the filename and returns a value of the workbook data type. This Workbook object represents the Excel file, a bit like how a File object represents an opened text file.

Remember that *example.xlsx* needs to be in the current working directory in order for you to work with it. You can find out what the current working directory is by importing os and using os.getcwd(), and you can change the current working directory using os.chdir().

Getting Sheets from the Workbook

You can get a list of all the sheet names in the workbook by accessing the sheetnames attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> wb.sheetnames # The workbook's sheets' names.
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb['Sheet3'] # Get a sheet from the workbook.
>>> sheet
<Worksheet "Sheet3">
>>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title # Get the sheet's title as a string.
'Sheet3'
>>> anotherSheet = wb.active # Get the active sheet.
>>> anotherSheet
<Worksheet "Sheet1">
```

Each sheet is represented by a Worksheet object, which you can obtain by using the square brackets with the sheet name string like a dictionary key. Finally, you can use the active attribute of a Workbook object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the Worksheet object, you can get its name from the title attribute.

Getting Cells from the Sheets

Once you have a Worksheet object, you can access a Cell object by its name. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb['Sheet1'] # Get a sheet from the workbook.
>>> sheet['A1'] # Get a cell from the sheet.
<Cell 'Sheet1'.A1>
>>> sheet['A1'].value # Get the value from the cell.
datetime.datetime(2015, 4, 5, 13, 34, 2)
>>> c = sheet['B1'] # Get another cell from the sheet.
>>> c.value
'Apples'
>>> # Get the row, column, and value from the cell.
>>> 'Row %, Column %s is %s' % (c.row, c.column, c.value)
'Row 1, Column B is Apples'
>>> 'Cell %s is %s' % (c.coordinate, c.value)
'Cell B1 is Apples'
>>> sheet['C1'].value
73
```

The Cell object has a value attribute that contains, unsurprisingly, the value stored in that cell. Cell objects also have row, column, and coordinate attributes that provide location information for the cell.

Here, accessing the value attribute of our Cell object for cell B1 gives us the string 'Apples'. The row attribute gives us the integer 1, the column attribute gives us 'B', and the coordinate attribute gives us 'B1'.

OpenPyXL will automatically interpret the dates in column A and return them as datetime values rather than strings. The datetime data type is explained further in [Chapter 17](#).

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start by using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the sheet's cell() method and passing integers for its row and column keyword arguments. The first row or column integer is 1, not 0. Continue the interactive shell example by entering the following:

```
>>> sheet.cell(row=1, column=2)
<Cell 'Sheet1'.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2): # Go through every other row:
...     print(i, sheet.cell(row=i, column=2).value)
...
1 Apples
3 Pears
5 Apples
7 Strawberries
```

As you can see, using the sheet's `cell()` method and passing it `row=1` and `column=2` gets you a Cell object for cell B1, just like specifying `sheet['B1']` did. Then, using the `cell()` method and its keyword arguments, you can write a for loop to print the values of a series of cells.

Say you want to go down column B and print the value in every cell with an odd row number. By passing 2 for the `range()` function's "step" parameter, you can get cells from every second row (in this case, all the odd-numbered rows). The for loop's `i` variable is passed for the row keyword argument to the `cell()` method, while 2 is always passed for the column keyword argument. Note that the integer 2, not the string 'B', is passed.

You can determine the size of the sheet with the Worksheet object's `max_row` and `max_column` attributes. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb['Sheet1']
>>> sheet.max_row # Get the highest row number.
7
>>> sheet.max_column # Get the highest column number.
3
```

Note that the `max_column` attribute is an integer rather than the letter that appears in Excel.

Converting Between Column Letters and Numbers

To convert from letters to numbers, call the `openpyxl.utils.column_index_from_string()` function. To convert from numbers to letters, call the `openpyxl.utils.get_column_letter()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.utils import get_column_letter, column_index_from_string
>>> get_column_letter(1) # Translate column 1 to a letter.
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb['Sheet1']
>>> get_column_letter(sheet.max_column)
'C'
>>> column_index_from_string('A') # Get A's number.
1
>>> column_index_from_string('AA')
27
```

After you import these two functions from the `openpyxl.utils` module, you can call `get_column_letter()` and pass it an integer like 27 to figure out what the letter name of the 27th column is. The function `column_index_string()` does the reverse: you pass it the letter name of a column, and it tells you what number that column is. You don't need to have a workbook loaded to use these functions. If you want, you can load a workbook, get a Worksheet object, and use a Worksheet attribute like `max_column` to get an integer. Then, you can pass that integer to `get_column_letter()`.

Getting Rows and Columns from the Sheets

You can slice Worksheet objects to get all the Cell objects in a row, column, or rectangular area of the spreadsheet. Then you can loop over all the cells in the slice. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb['Sheet1']
>>> tuple(sheet['A1':'C3']) # Get all cells from A1 to C3.
((<Cell 'Sheet1'.A1>, <Cell 'Sheet1'.B1>, <Cell 'Sheet1'.C1>), (<Cell
'Sheet1'.A2>, <Cell 'Sheet1'.B2>, <Cell 'Sheet1'.C2>), (<Cell 'Sheet1'.A3>,
<Cell 'Sheet1'.B3>, <Cell 'Sheet1'.C3>))
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:
❷ ...   for cellObj in rowOfCellObjects:
...       print(cellObj.coordinate, cellObj.value)
```

```
... print('--- END OF ROW ---')
```

```
A1 2015-04-05 13:34:02
```

```
B1 Apples
```

```
C1 73
```

```
--- END OF ROW ---
```

```
A2 2015-04-05 03:41:23
```

```
B2 Cherries
```

```
C2 85
```

```
--- END OF ROW ---
```

```
A3 2015-04-06 12:46:51
```

```
B3 Pears
```

```
C3 14
```

```
--- END OF ROW ---
```

Here, we specify that we want the Cell objects in the rectangular area from A1 to C3, and we get a Generator object containing the Cell objects in that area. To help us visualize this Generator object, we can use tuple() on it to display its Cell objects in a tuple.

This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the Cell objects in one row of our desired area, from the leftmost cell to the right. So overall, our slice of the sheet contains all the Cell objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.

To print the values of each cell in the area, we use two for loops. The outer for loop goes over each row in the slice ❶. Then, for each row, the nested for loop goes through each cell in that row ❷.

To access the values of cells in a particular row or column, you can also use a Worksheet object's rows and columns attribute. These attributes must be converted to lists with the list() function before you can use the square brackets and an index with them. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> list(sheet.columns)[1] # Get second column's cells.
(<Cell 'Sheet1'.B1>, <Cell 'Sheet1'.B2>, <Cell 'Sheet1'.B3>, <Cell 'Sheet1'.
B4>, <Cell 'Sheet1'.B5>, <Cell 'Sheet1'.B6>, <Cell 'Sheet1'.B7>)
>>> for cellObj in list(sheet.columns)[1]:
    print(cellObj.value)
```

```
Apples
```

```
Cherries
```

Pears
Oranges
Apples
Bananas
Strawberries

Using the rows attribute on a Worksheet object will give you a tuple of tuples. Each of these inner tuples represents a row, and contains the Cell objects in that row. The columns attribute also gives you a tuple of tuples, with each of the inner tuples containing the Cell objects in a particular column. For *example.xlsx*, since there are 7 rows and 3 columns, rows gives us a tuple of 7 tuples (each containing 3 Cell objects), and columns gives us a tuple of 3 tuples (each containing 7 Cell objects).

To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you use `list(sheet.columns)[1]`. To get the tuple containing the Cell objects in column A, you'd use `list(sheet.columns)[0]`. Once you have a tuple representing one row or column, you can loop through its Cell objects and print their values.

Workbooks, Sheets, Cells

As a quick review, here's a rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

1. Import the `openpyxl` module.
2. Call the `openpyxl.load_workbook()` function.
3. Get a Workbook object.
4. Use the active or sheetnames attributes.
5. Get a Worksheet object.
6. Use indexing or the `cell()` sheet method with row and column keyword arguments.
7. Get a Cell object.
8. Read the Cell object's value attribute.

PROJECT: READING DATA FROM A SPREADSHEET

Say you have a spreadsheet of data from the 2010 US Census and you have the boring task of going through its thousands of rows to count both the total population and the number of census tracts for each county. (A census tract is simply a geographic area defined for the purposes of the census.) Each row represents a single census tract. We'll name the spreadsheet file *censuspopdata.xlsx*, and you can download it from <https://nostarch.com/automatestuff2/>. Its contents look like [Figure 13-2](#).

	A	B	C	D	E
1	CensusTract	State	County	POP2010	
9841	06075010500	CA	San Francisco	2685	
9842	06075010600	CA	San Francisco	3894	
9843	06075010700	CA	San Francisco	5592	
9844	06075010800	CA	San Francisco	4578	
9845	06075010900	CA	San Francisco	4320	
9846	06075011000	CA	San Francisco	4827	
9847	06075011100	CA	San Francisco	5164	

Population by Census Tract

Ready

Figure 13-2: The censuspopdata.xlsx spreadsheet

Even though Excel can calculate the sum of multiple selected cells, you'd still have to select the cells for each of the 3,000-plus counties. Even if it takes just a few seconds to calculate a county's population by hand, this would take hours to do for the whole spreadsheet.

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds.

This is what your program does:

1. Reads the data from the Excel spreadsheet
2. Counts the number of census tracts in each county
3. Counts the total population of each county
4. Prints the results

This means your code will need to do the following:

1. Open and read the cells of an Excel document with the `openpyxl` module.
2. Calculate all the tract and population data and store it in a data structure.
3. Write the data structure to a text file with the `.py` extension using the `pprint` module.

Step 1: Read the Spreadsheet Data

There is just one sheet in the *censuspopdata.xlsx* spreadsheet, named 'Population by Census Tract', and each row holds the data for a single census tract. The columns are the tract number (A), the state abbreviation (B), the county name (C), and the population of the tract (D).

Open a new file editor tab and enter the following code. Save the file as *readCensusExcel.py*.

```
#!/ python3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.
```

```
❶ import openpyxl, pprint
    print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb['Population by Census Tract']
    countyData = {}

    # TODO: Fill in countyData with each county's population and tracts.
    print('Reading rows...')
❹ for row in range(2, sheet.max_row + 1):
    # Each row in the spreadsheet has data for one census tract.
    state = sheet['B' + str(row)].value
    county = sheet['C' + str(row)].value
    pop = sheet['D' + str(row)].value

# TODO: Open a new text file and write the contents of countyData to it.
```

This code imports the `openpyxl` module, as well as the `pprint` module that you'll use to print the final county data ❶. Then it opens the `censuspopdata.xlsx` file ❷, gets the sheet with the census data ❸, and begins iterating over its rows ❹.

Note that you've also created a variable named `countyData`, which will contain the populations and number of tracts you calculate for each county. Before you can store anything in it, though, you should determine exactly how you'll structure the data inside it.

Step 2: Populate the Data Structure

The data structure stored in `countyData` will be a dictionary with state abbreviations as its keys. Each state abbreviation will map to another dictionary, whose keys are strings of the county names in that state. Each county name will in turn map to a dictionary with just two keys, 'tracts' and 'pop'. These keys map to the number of census tracts and population for the county. For example, the dictionary will look similar to this:

```
{ 'AK': { 'Aleutians East': { 'pop': 3141, 'tracts': 1 },
        'Aleutians West': { 'pop': 5561, 'tracts': 2 },
        'Anchorage': { 'pop': 291826, 'tracts': 55 },
        'Bethel': { 'pop': 17013, 'tracts': 3 },
        'Bristol Bay': { 'pop': 997, 'tracts': 1 },
        --snip--
```

If the previous dictionary were stored in `countyData`, the following expressions would evaluate like this:

```
>>> countyData['AK']['Anchorage']['pop']
291826
```

```
>>> countyData['AK']['Anchorage']['tracts']
```

```
55
```

More generally, the countyData dictionary's keys will look like this:

```
countyData[state abbrev][county]['tracts']
```

```
countyData[state abbrev][county]['pop']
```

Now that you know how countyData will be structured, you can write the code that will fill it with the county data. Add the following code to the bottom of your program:

```
#!/ python 3
```

```
# readCensusExcel.py - Tabulates population and number of census tracts for
```

```
# each county.
```

```
--snip--
```

```
for row in range(2, sheet.max_row + 1):
```

```
    # Each row in the spreadsheet has data for one census tract.
```

```
    state = sheet['B' + str(row)].value
```

```
    county = sheet['C' + str(row)].value
```

```
    pop = sheet['D' + str(row)].value
```

```
    # Make sure the key for this state exists.
```

```
    ❶ countyData.setdefault(state, {})
```

```
    # Make sure the key for this county in this state exists.
```

```
    ❷ countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})
```

```
    # Each row represents one census tract, so increment by one.
```

```
    ❸ countyData[state][county]['tracts'] += 1
```

```
    # Increase the county pop by the pop in this census tract.
```

```
    ❹ countyData[state][county]['pop'] += int(pop)
```

```
# TODO: Open a new text file and write the contents of countyData to it.
```

The last two lines of code perform the actual calculation work, incrementing the value for tracts ❸ and increasing the value for pop ❹ for the current county on each iteration of the for loop.

The other code is there because you cannot add a county dictionary as the value for a state abbreviation key until the key itself exists in countyData. (That is, countyData['AK']['Anchorage']['tracts'] += 1 will cause an error if the 'AK' key doesn't

exist yet.) To make sure the state abbreviation key exists in your data structure, you need to call the `setdefault()` method to set a value if one does not already exist for state ❶.

Just as the `countyData` dictionary needs a dictionary as the value for each state abbreviation key, each of *those* dictionaries will need its own dictionary as the value for each county key ❷. And each of *those* dictionaries in turn will need keys 'tracts' and 'pop' that start with the integer value 0. (If you ever lose track of the dictionary structure, look back at the example dictionary at the start of this section.)

Since `setdefault()` will do nothing if the key already exists, you can call it on every iteration of the for loop without a problem.

Step 3: Write the Results to a File

After the for loop has finished, the `countyData` dictionary will contain all of the population and tract information keyed by county and state. At this point, you could program more code to write this to a text file or another Excel spreadsheet. For now, let's just use the `pprint.pformat()` function to write the `countyData` dictionary value as a massive string to a file named *census2010.py*. Add the following code to the bottom of your program (making sure to keep it unindented so that it stays outside the for loop):

```
#!/ python 3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.

--snip--

for row in range(2, sheet.max_row + 1):
    --snip--

# Open a new text file and write the contents of countyData to it.
print('Writing results...')
resultFile = open('census2010.py', 'w')
resultFile.write('allData = ' + pprint.pformat(countyData))
resultFile.close()
print('Done.')
```

The `pprint.pformat()` function produces a string that itself is formatted as valid Python code. By outputting it to a text file named *census2010.py*, you've generated a Python program from your Python program! This may seem complicated, but the advantage is that you can now import *census2010.py* just like any other Python module. In the interactive shell, change the current working directory to the folder with your newly created *census2010.py* file and then import it:

```
>>> import os

>>> import census2010
>>> census2010.allData['AK']['Anchorage']
{'pop': 291826, 'tracts': 55}
>>> anchoragePop = census2010.allData['AK']['Anchorage']['pop']
>>> print('The 2010 population of Anchorage was ' + str(anchoragePop))
The 2010 population of Anchorage was 291826
```

The *readCensusExcel.py* program was throwaway code: once you have its results saved to *census2010.py*, you won't need to run the program again. Whenever you need the county data, you can just run `import census2010`.

Calculating this data by hand would have taken hours; this program did it in a few seconds. Using OpenPyXL, you will have no trouble extracting information that is saved to an Excel spreadsheet and performing calculations on it. You can download the complete program from <https://nostarch.com/automatestuff2/>.

Ideas for Similar Programs

Many businesses and offices use Excel to store various types of data, and it's not uncommon for spreadsheets to become large and unwieldy. Any program that parses an Excel spreadsheet has a similar structure: it loads the spreadsheet file, preps some variables or data structures, and then loops through each of the rows in the spreadsheet. Such a program could do the following:

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.
- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

WRITING EXCEL DOCUMENTS

OpenPyXL also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, it's simple to create spreadsheets with thousands of rows of data.

Creating and Saving Excel Documents

Call the `openpyxl.Workbook()` function to create a new, blank Workbook object. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook() # Create a blank workbook.
>>> wb.sheetnames # It starts with one sheet.
```

```
['Sheet']
>>> sheet = wb.active
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet' # Change title.
>>> wb.sheetnames
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named *Sheet*. You can change the name of the sheet by storing a new string in its title attribute.

Any time you modify the Workbook object or its sheets and cells, the spreadsheet file will not be saved until you call the `save()` workbook method. Enter the following into the interactive shell (with *example.xlsx* in the current working directory):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx') # Save the workbook.
```

Here, we change the name of our sheet. To save our changes, we pass a filename as a string to the `save()` method. Passing a different filename than the original, such as *example_copy.xlsx*, saves the changes to a copy of the spreadsheet.

Whenever you edit a spreadsheet you've loaded from a file, you should always save the new, edited spreadsheet to a different filename than the original. That way, you'll still have the original spreadsheet file to work with in case a bug in your code caused the new, saved file to have incorrect or corrupt data.

Creating and Removing Sheets

Sheets can be added to and removed from a workbook with the `create_sheet()` method and `del` operator. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.sheetnames
['Sheet']
>>> wb.create_sheet() # Add a new sheet.
<Worksheet "Sheet1">
>>> wb.sheetnames
['Sheet', 'Sheet1']
>>> # Create a new sheet at index 0.
>>> wb.create_sheet(index=0, title='First Sheet')
```

```
<Worksheet "First Sheet">
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<Worksheet "Middle Sheet">
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

The `create_sheet()` method returns a new `Worksheet` object named `SheetX`, which by default is set to be the last sheet in the workbook. Optionally, the index and name of the new sheet can be specified with the index and title keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> del wb['Middle Sheet']
>>> del wb['Sheet1']
>>> wb.sheetnames
['First Sheet', 'Sheet']
```

You can use the `del` operator to delete a sheet from a workbook, just like you can use it to delete a key-value pair from a dictionary.

Remember to call the `save()` method to save the changes after adding sheets to or removing sheets from the workbook.

Writing Values to Cells

Writing values to cells is much like writing values to keys in a dictionary. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 'Hello, world!' # Edit the cell's value.
>>> sheet['A1'].value
'Hello, world!'
```

If you have the cell's coordinate as a string, you can use it just like a dictionary key on the `Worksheet` object to specify which cell to write to.

PROJECT: UPDATING A SPREADSHEET

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their

prices. Download this spreadsheet from <https://nostarch.com/automatestuff2/>. Figure 13-3 shows what the spreadsheet looks like.

	A	B	C	D	E
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL	
2	Potatoes	0.86	21.6	18.58	
3	Okra	2.26	38.6	87.24	
4	Fava beans	2.69	32.8	88.23	
5	Watermelon	0.66	27.3	18.02	
6	Garlic	1.19	4.9	5.83	
7	Parsnips	2.27	1.1	2.5	
8	Asparagus	2.49	37.9	94.37	
9	Avocados	3.23	9.2	29.72	
10	Celery	3.07	28.9	88.72	
11	Okra	2.26	40	90.4	

Figure 13-3: A spreadsheet of produce sales

Each row represents an individual sale. The columns are the type of produce sold (A), the cost per pound of that produce (B), the number of pounds sold (C), and the total revenue from the sale (D). The TOTAL column is set to the Excel formula `=ROUND(B3*C3, 2)`, which multiplies the cost per pound by the number of pounds sold and rounds the result to the nearest cent. With this formula, the cells in the TOTAL column will automatically update themselves if there is a change in column B or C.

Now imagine that the prices of garlic, celery, and lemons were entered incorrectly, leaving you with the boring task of going through thousands of rows in this spreadsheet to update the cost per pound for any garlic, celery, and lemon rows. You can't do a simple find-and-replace for the price, because there might be other items with the same price that you don't want to mistakenly "correct." For thousands of rows, this would take hours to do by hand. But you can write a program that can accomplish this in seconds.

Your program does the following:

1. Loops over all the rows
2. If the row is for garlic, celery, or lemons, changes the price

This means your code will need to do the following:

1. Open the spreadsheet file.
2. For each row, check whether the value in column A is Celery, Garlic, or Lemon.
3. If it is, update the price in column B.
4. Save the spreadsheet to a new file (so that you don't lose the old spreadsheet, just in case).

Step 1: Set Up a Data Structure with the Update Information

The prices that you need to update are as follows:

Celery	1.19
Garlic	3.07
Lemon	1.27

You could write code like this:

```
if produceName == 'Celery':  
    cellObj = 1.19  
if produceName == 'Garlic':  
    cellObj = 3.07  
if produceName == 'Lemon':  
    cellObj = 1.27
```

Having the produce and updated price data hardcoded like this is a bit inelegant. If you needed to update the spreadsheet again with different prices or different produce, you would have to change a lot of the code. Every time you change code, you risk introducing bugs.

A more flexible solution is to store the corrected price information in a dictionary and write your code to use this data structure. In a new file editor tab, enter the following code:

```
#!/ python3  
# updateProduce.py - Corrects costs in produce sales spreadsheet.  
  
import openpyxl  
  
wb = openpyxl.load_workbook('produceSales.xlsx')  
sheet = wb['Sheet']  
  
# The produce types and their updated prices  
PRICE_UPDATES = {'Garlic': 3.07,  
                  'Celery': 1.19,  
                  'Lemon': 1.27}  
  
# TODO: Loop through the rows and update the prices.
```

Save this as *updateProduce.py*. If you need to update the spreadsheet again, you'll need to update only the PRICE_UPDATES dictionary, not any other code.

Step 2: Check All Rows and Update Incorrect Prices

The next part of the program will loop through all the rows in the spreadsheet. Add the following code to the bottom of *updateProduce.py*:

```
#!/python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

--snip--

# Loop through the rows and update the prices.
❶ for rowNum in range(2, sheet.max_row): # skip the first row
    ❷ produceName = sheet.cell(row=rowNum, column=1).value
    ❸ if produceName in PRICE_UPDATES:
        sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceName]

❹ wb.save('updatedProduceSales.xlsx')
```

We loop through the rows starting at row 2, since row 1 is just the header ❶. The cell in column 1 (that is, column A) will be stored in the variable produceName ❷. If produceName exists as a key in the PRICE_UPDATES dictionary ❸, then you know this is a row that must have its price corrected. The correct price will be in PRICE_UPDATES[produceName].

Notice how clean using PRICE_UPDATES makes the code. Only one if statement, rather than code like if produceName == 'Garlic':, is necessary for every type of produce to update. And since the code uses the PRICE_UPDATES dictionary instead of hardcoding the produce names and updated costs into the for loop, you modify only the PRICE_UPDATES dictionary and not the code if the produce sales spreadsheet needs additional changes.

After going through the entire spreadsheet and making changes, the code saves the Workbook object to *updatedProduceSales.xlsx* ❹. It doesn't overwrite the old spreadsheet just in case there's a bug in your program and the updated spreadsheet is wrong. After checking that the updated spreadsheet looks right, you can delete the old spreadsheet.

You can download the complete source code for this program from <https://nostarch.com/automatestuff2/>.

Ideas for Similar Programs

Since many office workers use Excel spreadsheets all the time, a program that can automatically edit and write Excel files could be really useful. Such a program could do the following:

- Read data from one spreadsheet and write it to parts of other spreadsheets.
- Read data from websites, text files, or the clipboard and write it to a spreadsheet.
- Automatically “clean up” data in spreadsheets. For example, it could use regular expressions to read multiple formats of phone numbers and edit them to a single, standard format.

SETTING THE FONT STYLE OF CELLS

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. In the produce spreadsheet, for example, your program could apply bold text to the potato, garlic, and parsnip rows. Or perhaps you want to italicize every row with a cost per pound greater than \$5. Styling parts of a large spreadsheet by hand would be tedious, but your programs can do it instantly.

To customize font styles in cells, important, import the `Font()` function from the `openpyxl.styles` module.

```
from openpyxl.styles import Font
```

This allows you to type `Font()` instead of `openpyxl.styles.Font()`. (See “[Importing Modules](#)” on [page 47](#) to review this style of import statement.)

Here’s an example that creates a new workbook and sets cell A1 to have a 24-point, italicized font. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
❶ >>> italic24Font = Font(size=24, italic=True) # Create a font.
❷ >>> sheet['A1'].font = italic24Font # Apply the font to A1.
>>> sheet['A1'] = 'Hello, world!'
>>> wb.save('styles.xlsx')
```

In this example, `Font(size=24, italic=True)` returns a `Font` object, which is stored in `italic24Font` ❶. The keyword arguments to `Font()`, `size` and `italic`, configure the `Font` object’s styling information. And when `sheet['A1'].font` is assigned the `italic24Font` object ❷, all that font styling information gets applied to cell A1.

FONT OBJECTS

To set font attributes, you pass keyword arguments to `Font()`. [Table 13-2](#) shows the possible keyword arguments for the `Font()` function.

Table 13-2: Keyword Arguments for Font Objects

Keyword argument	Data type	Description
name	String	The font name, such as 'Calibri' or 'Times New Roman'
size	Integer	The point size
bold	Boolean	True, for bold font
italic	Boolean	True, for italic font

You can call `Font()` to create a `Font` object and store that `Font` object in a variable. You then assign that variable to a `Cell` object's `font` attribute. For example, this code creates various font styles:

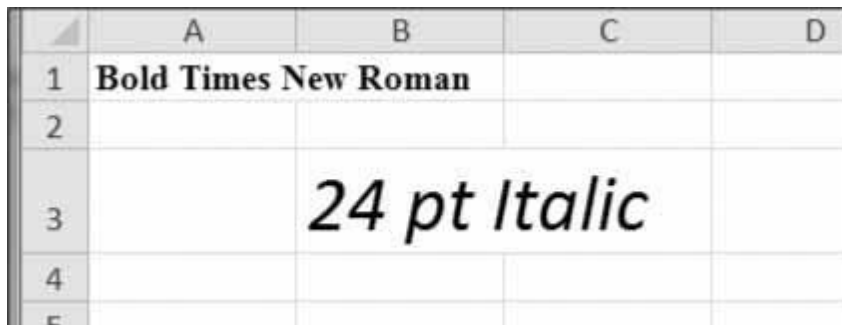
```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']

>>> fontObj1 = Font(name='Times New Roman', bold=True)
>>> sheet['A1'].font = fontObj1
>>> sheet['A1'] = 'Bold Times New Roman'

>>> fontObj2 = Font(size=24, italic=True)
>>> sheet['B3'].font = fontObj2
>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles.xlsx')
```

Here, we store a `Font` object in `fontObj1` and then set the `A1` Cell object's `font` attribute to `fontObj1`. We repeat the process with another `Font` object to set the font of a second cell. After you run this code, the styles of the `A1` and `B3` cells in the spreadsheet will be set to custom font styles, as shown in [Figure 13-4](#).



	A	B	C	D
1	Bold Times New Roman			
2				
3		<i>24 pt Italic</i>		
4				
5				

Figure 13-4: A spreadsheet with custom font styles

For cell `A1`, we set the font name to `'Times New Roman'` and set `bold` to `true`, so our text appears in bold Times New Roman. We didn't specify a size, so the `openpyxl` default, 11, is used. In cell `B3`, our text is italic, with a size of 24; we didn't specify a font name, so the `openpyxl` default, Calibri, is used.

FORMULAS

Excel formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells. In this section, you'll use the `openpyxl` module to programmatically add formulas to cells, just like any normal value. For example:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

This will store `=SUM(B1:B8)` as the value in cell B9. This sets the B9 cell to a formula that calculates the sum of values in cells B1 to B8. You can see this in action in [Figure 13-5](#).

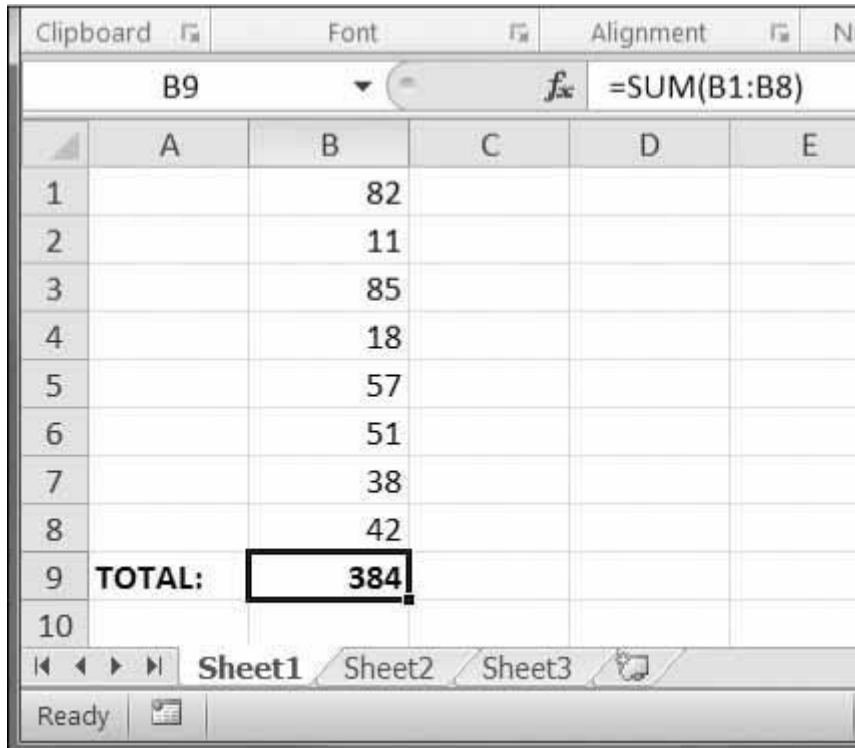


Figure 13-5: Cell B9 contains the formula `=SUM(B1:B8)`, which adds the cells B1 to B8.

An Excel formula is set just like any other text value in a cell. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet['A1'] = 200
>>> sheet['A2'] = 300
>>> sheet['A3'] = '=SUM(A1:A2)' # Set the formula.
>>> wb.save('writeFormula.xlsx')
```

The cells in A1 and A2 are set to 200 and 300, respectively. The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spreadsheet is opened in Excel, A3 will display its value as 500.

Excel formulas offer a level of programmability for spreadsheets but can quickly become unmanageable for complicated tasks. For example, even if you're deeply familiar with Excel formulas, it's a headache to try to decipher what `=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!A1:B10000, 2, FALSE))>0, SUBSTITUTE(VLOOKUP(F7,`

Sheet2!\$A\$1:\$B\$10000, 2, FALSE), " ", ""),""), "")) actually does. Python code is much more readable.

ADJUSTING ROWS AND COLUMNS

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header. But if you need to set a row or column's size based on its cells' contents or if you want to set sizes in a large number of spreadsheet files, it will be much quicker to write a Python program to do it.

Rows and columns can also be hidden entirely from view. Or they can be “frozen” in place so that they are always visible on the screen and appear on every page when the spreadsheet is printed (which is handy for headers).

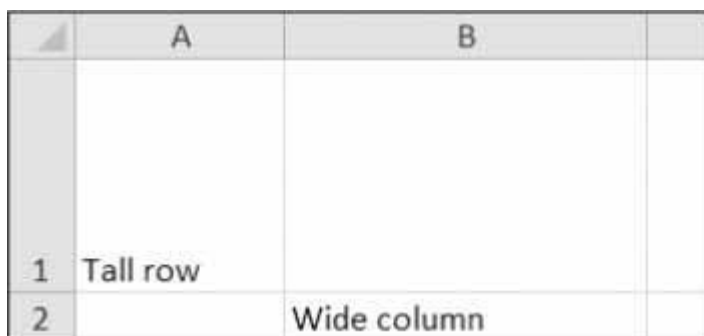
Setting Row Height and Column Width

Worksheet objects have `row_dimensions` and `column_dimensions` attributes that control row heights and column widths. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet['A1'] = 'Tall row'
>>> sheet['B2'] = 'Wide column'
>>> # Set the height and width:
>>> sheet.row_dimensions[1].height = 70
>>> sheet.column_dimensions['B'].width = 20
>>> wb.save('dimensions.xlsx')
```

A sheet's `row_dimensions` and `column_dimensions` are dictionary-like values; `row_dimensions` contains `RowDimension` objects and `column_dimensions` contains `ColumnDimension` objects. In `row_dimensions`, you can access one of the objects using the number of the row (in this case, 1 or 2). In `column_dimensions`, you can access one of the objects using the letter of the column (in this case, A or B).

The *dimensions.xlsx* spreadsheet looks like [Figure 13-6](#).



	A	B
1	Tall row	
2		Wide column

Figure 13-6: Row 1 and column B set to larger heights and widths

Once you have the RowDimension object, you can set its height. Once you have the ColumnDimension object, you can set its width. The row height can be set to an integer or float value between 0 and 409. This value represents the height measured in *points*, where one point equals 1/72 of an inch. The default row height is 12.75. The column width can be set to an integer or float value between 0 and 255. This value represents the number of characters at the default font size (11 point) that can be displayed in the cell. The default column width is 8.43 characters. Columns with widths of 0 or rows with heights of 0 are hidden from the user.

Merging and Unmerging Cells

A rectangular area of cells can be merged into a single cell with the `merge_cells()` sheet method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet.merge_cells('A1:D3') # Merge all these cells.
>>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5') # Merge these two cells.
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged.xlsx')
```

The argument to `merge_cells()` is a single string of the top-left and bottom-right cells of the rectangular area to be merged: 'A1:D3' merges 12 cells into a single cell. To set the value of these merged cells, simply set the value of the top-left cell of the merged group.

When you run this code, *merged.xlsx* will look like [Figure 13-7](#).

	A	B	C	D	E
1	Twelve cells merged together.				
2					
3					
4					
5			Two merged cells.		
6					
7					

Figure 13-7: Merged cells in a spreadsheet

To unmerge cells, call the `unmerge_cells()` sheet method. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
```

```
>>> sheet = wb.active
>>> sheet.unmerge_cells('A1:D3') # Split these cells up.
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged.xlsx')
```

If you save your changes and then take a look at the spreadsheet, you'll see that the merged cells have gone back to being individual cells.

Freezing Panes

For spreadsheets too large to be displayed all at once, it's helpful to “freeze” a few of the top rows or leftmost columns onscreen. Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as *freeze panes*. In OpenPyXL, each Worksheet object has a `freeze_panes` attribute that can be set to a Cell object or a string of a cell's coordinates. Note that all rows above and all columns to the left of this cell will be frozen, but the row and column of the cell itself will not be frozen.

To unfreeze all panes, set `freeze_panes` to `None` or `'A1'`. Table 13-3 shows which rows and columns will be frozen for some example settings of `freeze_panes`.

Table 13-3: Frozen Pane Examples

freeze_panes setting	Rows and columns frozen
<code>sheet.freeze_panes = 'A2'</code>	Row 1
<code>sheet.freeze_panes = 'B1'</code>	Column A
<code>sheet.freeze_panes = 'C1'</code>	Columns A and B
<code>sheet.freeze_panes = 'C2'</code>	Row 1 and columns A and B
<code>sheet.freeze_panes = 'A1'</code> or <code>sheet.freeze_panes = None</code>	No frozen panes

Make sure you have the produce sales spreadsheet from <https://nostarch.com/automatestuff2/>. Then enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales.xlsx')
>>> sheet = wb.active
>>> sheet.freeze_panes = 'A2' # Freeze the rows above A2.
>>> wb.save('freezeExample.xlsx')
```

If you set the `freeze_panes` attribute to `'A2'`, row 1 will always be viewable, no matter where the user scrolls in the spreadsheet. You can see this in [Figure 13-8](#).

The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F
1	FRUIT	COST PER POUND	POUNDS SOLD	TOTAL		
1591	Fava beans	2.69	0.7	1.88		
1592	Grapefruit	0.76	28.5	21.66		
1593	Green peppers	1.89	37	69.93		
1594	Watermelon	0.66	30.4	20.06		
1595	Celery	3.07	36.6	112.36		
1596	Strawberries	4.4	5.5	24.2		
1597	Green beans	2.52	40	100.8		

Figure 13-8: With `freeze_panes` set to 'A2', row 1 is always visible, even as the user scrolls down.

CHARTS

OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

1. Create a Reference object from a rectangular selection of cells.
2. Create a Series object by passing in the Reference object.
3. Create a Chart object.
4. Append the Series object to the Chart object.
5. Add the Chart object to the Worksheet object, optionally specifying which cell should be the top-left corner of the chart.

The Reference object requires some explaining. You create Reference objects by calling the `openpyxl.chart.Reference()` function and passing three arguments:

1. The Worksheet object containing your chart data.
2. A tuple of two integers, representing the top-left cell of the rectangular selection of cells containing your chart data: the first integer in the tuple is the row, and the second is the column. Note that 1 is the first row, not 0.
3. A tuple of two integers, representing the bottom-right cell of the rectangular selection of cells containing your chart data: the first integer in the tuple is the row, and the second is the column.

Figure 13-9 shows some sample coordinate arguments.

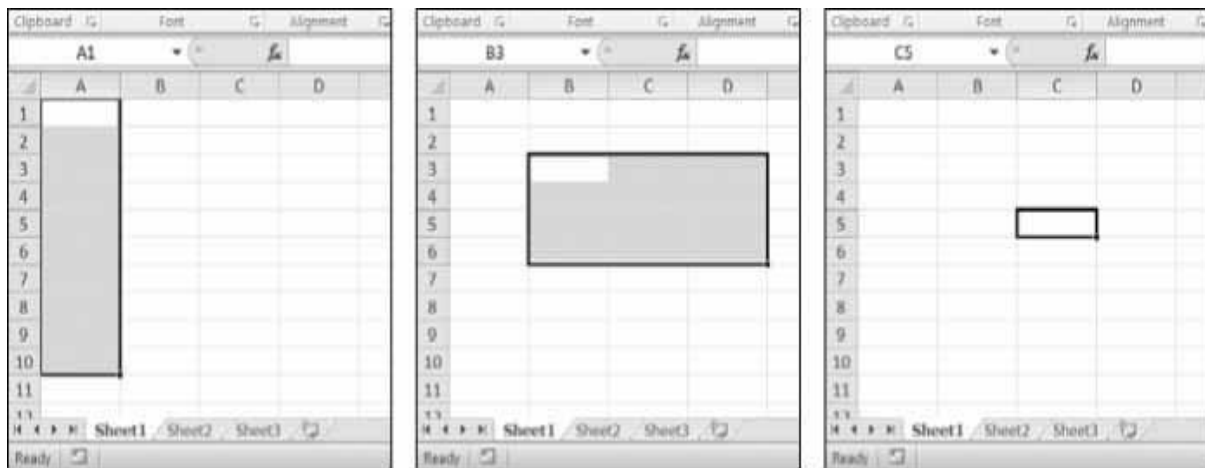


Figure 13-9: From left to right: (1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3)

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> for i in range(1, 11): # create some data in column A
...     sheet['A' + str(i)] = i
...
>>> refObj = openpyxl.chart.Reference(sheet, min_col=1, min_row=1, max_col=1,
max_row=10)
>>> seriesObj = openpyxl.chart.Series(refObj, title='First series')

>>> chartObj = openpyxl.chart.BarChart()
>>> chartObj.title = 'My Chart'
>>> chartObj.append(seriesObj)

>>> sheet.add_chart(chartObj, 'C5')
>>> wb.save('sampleChart.xlsx')
```

This produces a spreadsheet that looks like [Figure 13-10](#).

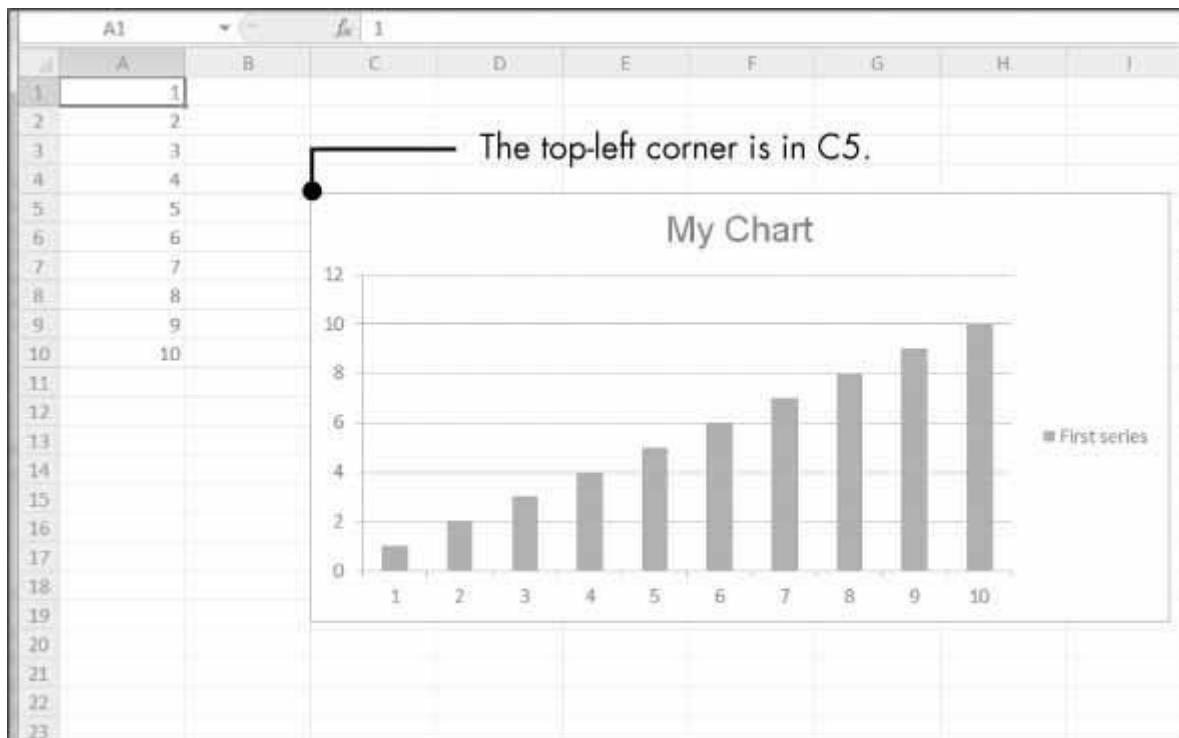


Figure 13-10: A spreadsheet with a chart added

We've created a bar chart by calling `openpyxl.chart.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.charts.LineChart()`, `openpyxl.chart.ScatterChart()`, and `openpyxl.chart.PieChart()`.

SUMMARY

Often the hard part of processing information isn't the processing itself but simply getting the data in the right format for your program. But once you have your spreadsheet loaded into Python, you can extract and manipulate its data much faster than you could by hand.

You can also generate spreadsheets as output from your programs. So if colleagues need your text file or PDF of thousands of sales contacts transferred to a spreadsheet file, you won't have to tediously copy and paste it all into Excel.

Equipped with the `openpyxl` module and some programming knowledge, you'll find processing even the biggest spreadsheets a piece of cake.

In the next chapter, we'll take a look at using Python to interact with another spreadsheet program: the popular online Google Sheets application.

PRACTICE QUESTIONS

For the following questions, imagine you have a `Workbook` object in the variable `wb`, a `Worksheet` object in `sheet`, a `Cell` object in `cell`, a `Comment` object in `comm`, and an `Image` object in `img`.

1. What does the `openpyxl.load_workbook()` function return?

2. What does the `wb.sheetnames` workbook attribute contain?
3. How would you retrieve the `Worksheet` object for a sheet named 'Sheet1'?
4. How would you retrieve the `Worksheet` object for the workbook's active sheet?
5. How would you retrieve the value in the cell C5?
6. How would you set the value in the cell C5 to "Hello"?
7. How would you retrieve the cell's row and column as integers?
8. What do the `sheet.max_column` and `sheet.max_row` sheet attributes hold, and what is the data type of these attributes?
9. If you needed to get the integer index for column 'M', what function would you need to call?
10. If you needed to get the string name for column 14, what function would you need to call?
11. How can you retrieve a tuple of all the `Cell` objects from A1 to F1?
12. How would you save the workbook to the filename *example.xlsx*?
13. How do you set a formula in a cell?
14. If you want to retrieve the result of a cell's formula instead of the cell's formula itself, what must you do first?
15. How would you set the height of row 5 to 100?
16. How would you hide column C?
17. What is a freeze pane?
18. What five functions and methods do you have to call to create a bar chart?

PRACTICE PROJECTS

For practice, write programs that perform the following tasks.

Multiplication Table Maker

Create a program *multiplicationTable.py* that takes a number N from the command line and creates an $N \times N$ multiplication table in an Excel spreadsheet. For example, when the program is run like this:

```
py multiplicationTable.py 6
```

... it should create a spreadsheet that looks like [Figure 13-11](#).

	A	B	C	D	E	F	G	H
1		1	2	3	4	5	6	
2	1	1	2	3	4	5	6	
3	2	2	4	6	8	10	12	
4	3	3	6	9	12	15	18	
5	4	4	8	12	16	20	24	
6	5	5	10	15	20	25	30	
7	6	6	12	18	24	30	36	
8								
9								

Figure 13-11: A multiplication table generated in a spreadsheet

Row 1 and column A should be used for labels and should be in bold.

Blank Row Inserter

Create a program *blankRowInserter.py* that takes two integers and a filename string as command line arguments. Let's call the first integer *N* and the second integer *M*. Starting at row *N*, the program should insert *M* blank rows into the spreadsheet. For example, when the program is run like this:

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

... the “before” and “after” spreadsheets should look like [Figure 13-12](#).

A1		Potatoes				
	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pep	Green bea	Fava b
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3	Fava bean	Spinach	Grapefruit	Grapes	Apricots	Papaya
4	Watermel	Cucumber	Ginger	Watermel	Red onion	Butter
5	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
6	Parsnips	Okra	Cucumber	Apples	Grapes	Avocad
7	Asparagus	Fava bean	Green cab	Grapefruit	Ginger	Butter
8	Avocados	Watermel	Eggplant	Grapes	Strawberri	Celery

A1		Potatoes				
	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pep	Green bea	Fava b
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3						
4						
5	Fava bean	Spinach	Grapefruit	Grapes	Apricots	Papaya
6	Watermel	Cucumber	Ginger	Watermel	Red onion	Butter
7	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
8	Parsnips	Okra	Cucumber	Apples	Grapes	Avocad

Figure 13-12: Before (left) and after (right) the two blank rows are inserted at row 3

You can write this program by reading in the contents of the spreadsheet. Then, when writing out the new spreadsheet, use a for loop to copy the first *N* lines. For the remaining lines, add *M* to the row number in the output spreadsheet.

Spreadsheet Cell Inverter

Write a program to invert the row and column of the cells in the spreadsheet. For example, the value at row 5, column 3 will be at row 3, column 5 (and vice versa). This should be done for all cells in the spreadsheet. For example, the “before” and “after” spreadsheets would look something like [Figure 13-13](#).

	A1									
	A	B	C	D	E	F	G	H	I	J
1	ITEM	SOLD								
2	Eggplant	334								
3	Cucumber	252								
4	Green cabi	238								
5	Eggplant	516								
6	Garlic	98								
7	Parsnips	16								
8	Asparagus	335								
9	Avocados	84								
10										

	A1									
	A	B	C	D	E	F	G	H	I	J
1	ITEM	Eggplant	Cucumber	Green cabi	Eggplant	Garlic	Parsnips	Asparagus	Avocados	
2	SOLD	334	252	238	516	98	16	335	84	
3										
4										
5										
6										
7										
8										
9										
10										

Figure 13-13: The spreadsheet before (top) and after (bottom) inversion

You can write this program by using nested for loops to read the spreadsheet's data into a list of lists data structure. This data structure could have `sheetData[x][y]` for the cell at column `x` and row `y`. Then, when writing out the new spreadsheet, use `sheetData[y][x]` for the cell at column `x` and row `y`.

Text Files to Spreadsheet

Write a program to read in the contents of several text files (you can make the text files yourself) and insert those contents into a spreadsheet, with one line of text per row. The lines of the first text file will be in the cells of column A, the lines of the second text file will be in the cells of column B, and so on.

Use the `readlines()` File object method to return a list of strings, one string per line in the file. For the first file, output the first line to column 1, row 1. The second line should be written to column 1, row 2, and so on. The next file that is read with `readlines()` will be written to column 2, the next file to column 3, and so on.

Spreadsheet to Text Files

Write a program that performs the tasks of the previous program in reverse order: the program should open a spreadsheet and write the cells of column A into one text file, the cells of column B into another text file, and so on.

WORKING WITH PDF AND WORD DOCUMENTS

PDF and Word documents are binary files, which makes them much more complex than plaintext files. In addition to text, they store lots of font, color, and layout information. If you want your programs to read or write to PDFs or Word documents, you'll need to do more than simply pass their filenames to `open()`.

Fortunately, there are Python modules that make it easy for you to interact with PDFs and Word documents. This chapter will cover two such modules: PyPDF2 and Python-Docx.

PDF DOCUMENTS

PDF stands for *Portable Document Format* and uses the *.pdf* file extension. Although PDFs support many features, this chapter will focus on the two things you'll be doing most often with them: reading text content from PDFs and crafting new PDFs from existing documents.

The module you'll use to work with PDFs is PyPDF2 version 1.26.0. It's important that you install this version because future versions of PyPDF2 may be incompatible with the code. To install it, run `pip install --user PyPDF2==1.26.0` from the command line. This module name is case sensitive, so make sure the `y` is lowercase and everything else is uppercase. (Check out [Appendix A](#) for full details about installing third-party modules.) If the module was installed correctly, running `import PyPDF2` in the interactive shell shouldn't display any errors.

THE PROBLEMATIC PDF FORMAT

While PDF files are great for laying out text in a way that's easy for people to print and read, they're not straightforward for software to parse into plaintext. As a result, PyPDF2 might make mistakes when extracting text from a PDF and may even be unable to open some PDFs at all. There isn't much you can do about this, unfortunately. PyPDF2 may simply be unable to work with some of your particular PDF files. That said, I haven't found any PDF files so far that can't be opened with PyPDF2.

Extracting Text from PDFs

PyPDF2 does not have a way to extract images, charts, or other media from PDF documents, but it can extract text and return it as a Python string. To start learning how PyPDF2 works, we'll use it on the example PDF shown in [Figure 15-1](#).

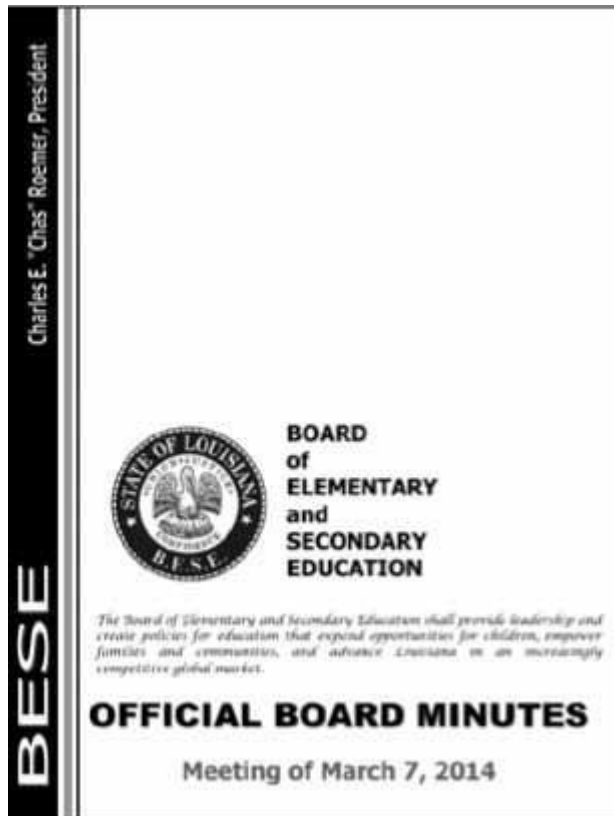


Figure 15-1: The PDF page that we will be extracting text from

Download this PDF from <https://nostarch.com/automatestuff2/> and enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
'OFFFFIICCIIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of March 7,
2015  \n  The Board of Elementary and Secondary Education shall
provide leadership and create policies for education that expand opportunities
for children, empower families and communities, and advance Louisiana in an
increasingly competitive global market. BOARD of ELEMENTARY and SECONDARY
EDUCATION '
>>> pdfFileObj.close()
```

First, import the PyPDF2 module. Then open *meetingminutes.pdf* in read binary mode and store it in pdfFileObj. To get a PdfFileReader object that represents this PDF,

call `PyPDF2.PdfFileReader()` and pass it `pdfFileObj`. Store this `PdfFileReader` object in `pdfReader`.

The total number of pages in the document is stored in the `numPages` attribute of a `PdfFileReader` object ❶. The example PDF has 19 pages, but let's extract text from only the first page.

To extract text from a page, you need to get a `Page` object, which represents a single page of a PDF, from a `PdfFileReader` object. You can get a `Page` object by calling the `getPage()` method ❷ on a `PdfFileReader` object and passing it the page number of the page you're interested in—in our case, 0.

`PyPDF2` uses a *zero-based index* for getting pages: The first page is page 0, the second is page 1, and so on. This is always the case, even if pages are numbered differently within the document. For example, say your PDF is a three-page excerpt from a longer report, and its pages are numbered 42, 43, and 44. To get the first page of this document, you would want to call `pdfReader.getPage(0)`, not `getPage(42)` or `getPage(1)`.

Once you have your `Page` object, call its `extractText()` method to return a string of the page's text ❸. The text extraction isn't perfect: The text *Charles E. "Chas" Roemer, President* from the PDF is absent from the string returned by `extractText()`, and the spacing is sometimes off. Still, this approximation of the PDF text content may be good enough for your program.

Decrypting PDFs

Some PDF documents have an encryption feature that will keep them from being read until whoever is opening the document provides a password. Enter the following into the interactive shell with the PDF you downloaded, which has been encrypted with the password *rosebud*:

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❶ >>> pdfReader.isEncrypted
True
>>> pdfReader.getPage(0)
❷ Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <module>
    pdfReader.getPage()
  --snip--
  File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in getObject
    raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❸ >>> pdfReader.decrypt('rosebud')
1
>>> pageObj = pdfReader.getPage(0)
```

All PdfFileReader objects have an isEncrypted attribute that is True if the PDF is encrypted and False if it isn't ❶. Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error ❷.

NOTE

Due to a bug in PyPDF2 version 1.26.0, calling getPage() on an encrypted PDF before calling decrypt() on it causes future getPage() calls to fail with the following error: IndexError: list index out of range. This is why our example reopened the file with a new PdfFileReader object.

To read an encrypted PDF, call the decrypt() function and pass the password as a string ❸. After you call decrypt() with the correct password, you'll see that calling getPage() no longer causes an error. If given the wrong password, the decrypt() function will return 0 and getPage() will continue to fail. Note that the decrypt() method decrypts only the PdfFileReader object, not the actual PDF file. After your program terminates, the file on your hard drive remains encrypted. Your program will have to call decrypt() again the next time it is run.

Creating PDFs

PyPDF2's counterpart to PdfFileReader is PdfFileWriter, which can create new PDF files. But PyPDF2 cannot write arbitrary text to a PDF like Python can do with plaintext files. Instead, PyPDF2's PDF-writing capabilities are limited to copying pages from other PDFs, rotating pages, overlaying pages, and encrypting files.

PyPDF2 doesn't allow you to directly edit a PDF. Instead, you have to create a new PDF and then copy content over from an existing document. The examples in this section will follow this general approach:

1. Open one or more existing PDFs (the source PDFs) into PdfFileReader objects.
2. Create a new PdfFileWriter object.
3. Copy pages from the PdfFileReader objects into the PdfFileWriter object.
4. Finally, use the PdfFileWriter object to write the output PDF.

Creating a PdfFileWriter object creates only a value that represents a PDF document in Python. It doesn't create the actual PDF file. For that, you must call the PdfFileWriter's write() method.

The write() method takes a regular File object that has been opened in *write-binary* mode. You can get such a File object by calling Python's open() function with two arguments: the string of what you want the PDF's filename to be and 'wb' to indicate the file should be opened in write-binary mode.

If this sounds a little confusing, don't worry—you'll see how this works in the following code examples.

Copying Pages

You can use PyPDF2 to copy pages from one PDF document to another. This allows you to combine multiple PDF files, cut unwanted pages, or reorder pages.

Download *meetingminutes.pdf* and *meetingminutes2.pdf* from <https://nostarch.com/automatestuff2/> and place the PDFs in the current working directory. Enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdf1File = open('meetingminutes.pdf', 'rb')
>>> pdf2File = open('meetingminutes2.pdf', 'rb')
❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

>>> for pageNum in range(pdf1Reader.numPages):
    ❹ pageObj = pdf1Reader.getPage(pageNum)
    ❺ pdfWriter.addPage(pageObj)

>>> for pageNum in range(pdf2Reader.numPages):
    ❹ pageObj = pdf2Reader.getPage(pageNum)
    ❺ pdfWriter.addPage(pageObj)

❻ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
>>> pdfWriter.write(pdfOutputFile)
>>> pdfOutputFile.close()
>>> pdf1File.close()
>>> pdf2File.close()
```

Open both PDF files in read binary mode and store the two resulting File objects in pdf1File and pdf2File. Call PyPDF2.PdfFileReader() and pass it pdf1File to get a PdfFileReader object for *meetingminutes.pdf* ❶. Call it again and pass it pdf2File to get a PdfFileReader object for *meetingminutes2.pdf* ❷. Then create a new PdfFileWriter object, which represents a blank PDF document ❸.

Next, copy all the pages from the two source PDFs and add them to the PdfFileWriter object. Get the Page object by calling getPage() on a PdfFileReader object ❹. Then pass that Page object to your PdfFileWriter's addPage() method ❺. These steps are done first for pdf1Reader and then again for pdf2Reader. When you're done copying pages, write a new PDF called *combinedminutes.pdf* by passing a File object to the PdfFileWriter's write() method ❻.

NOTE

PyPDF2 cannot insert pages in the middle of a PdfFileWriter object; the addPage() method will only add pages to the end.

You have now created a new PDF file that combines the pages from *meetingminutes.pdf* and *meetingminutes2.pdf* into a single document. Remember that

the File object passed to PyPDF2.PdfFileReader() needs to be opened in read-binary mode by passing 'rb' as the second argument to open(). Likewise, the File object passed to PyPDF2.PdfFileWriter() needs to be opened in write-binary mode with 'wb'.

Rotating Pages

The pages of a PDF can also be rotated in 90-degree increments with the rotateClockwise() and rotateCounterClockwise() methods. Pass one of the integers 90, 180, or 270 to these methods. Enter the following into the interactive shell, with the *meetingminutes.pdf* file in the current working directory:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❶ >>> page = pdfReader.getPage(0)
❷ >>> page.rotateClockwise(90)
{'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
--snip--
]}
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(page)
❸ >>> resultPdfFile = open('rotatedPage.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> resultPdfFile.close()
>>> minutesFile.close()
```

Here we use getPage(0) to select the first page of the PDF ❶, and then we call rotateClockwise(90) on that page ❷. We write a new PDF with the rotated page and save it as *rotatedPage.pdf* ❸.

The resulting PDF will have one page, rotated 90 degrees clockwise, as shown in [Figure 15-2](#). The return values from rotateClockwise() and rotateCounterClockwise() contain a lot of information that you can ignore.

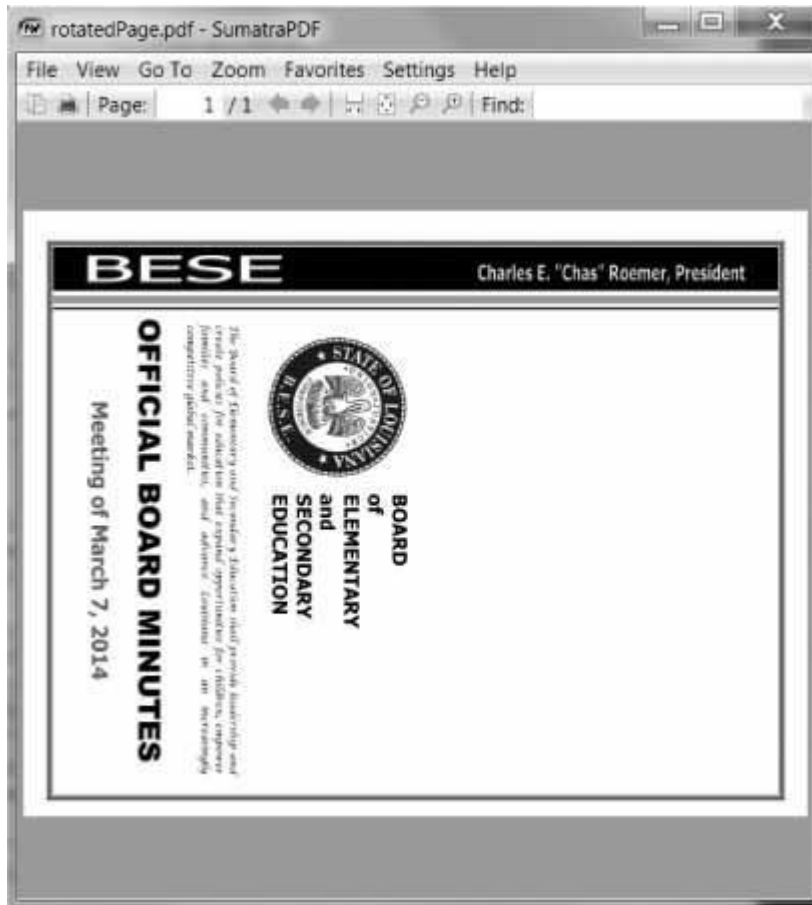


Figure 15-2: The rotatedPage.pdf file with the page rotated 90 degrees clockwise

Overlaying Pages

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With Python, it's easy to add watermarks to multiple files and only to pages your program specifies.

Download *watermark.pdf* from <https://nostarch.com/automatestuff2/> and place the PDF in the current working directory along with *meetingminutes.pdf*. Then enter the following into the interactive shell:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
❶ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸ >>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
❺ >>> pdfWriter = PyPDF2.PdfFileWriter()
❻ >>> pdfWriter.addPage(minutesFirstPage)

❷ >>> for pageNum in range(1, pdfReader.numPages):
```

```
pageObj = pdfReader.getPage(pageNum)
pdfWriter.addPage(pageObj)
```

```
>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> minutesFile.close()
>>> resultPdfFile.close()
```

Here we make a PdfFileReader object of *meetingminutes.pdf* ❶. We call `getPage(0)` to get a Page object for the first page and store this object in `minutesFirstPage` ❷. We then make a PdfFileReader object for *watermark.pdf* ❸ and call `mergePage()` on `minutesFirstPage` ❹. The argument we pass to `mergePage()` is a Page object for the first page of *watermark.pdf*.

Now that we've called `mergePage()` on `minutesFirstPage`, `minutesFirstPage` represents the watermarked first page. We make a PdfFileWriter object ❺ and add the watermarked first page ❻. Then we loop through the rest of the pages in *meetingminutes.pdf* and add them to the PdfFileWriter object ❼. Finally, we open a new PDF called *watermarkedCover.pdf* and write the contents of the PdfFileWriter to the new PDF.

Figure 15-3 shows the results. Our new PDF, *watermarkedCover.pdf*, has all the contents of the *meetingminutes.pdf*, and the first page is watermarked.



Figure 15-3: The original PDF (left), the watermark PDF (center), and the merged PDF (right)

Encrypting PDFs

A PdfFileWriter object can also add encryption to a PDF document. Enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdfReader.numPages):
>>>     pdfWriter.addPage(pdfReader.getPage(pageNum))
```

```
❶ >>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

Before calling the `write()` method to save to a file, call the `encrypt()` method and pass it a password string ❶. PDFs can have a *user password* (allowing you to view the PDF) and an *owner password* (allowing you to set permissions for printing, commenting, extracting text, and other features). The user password and owner password are the first and second arguments to `encrypt()`, respectively. If only one string argument is passed to `encrypt()`, it will be used for both passwords.

In this example, we copied the pages of *meetingminutes.pdf* to a `PdfFileWriter` object. We encrypted the `PdfFileWriter` with the password *swordfish*, opened a new PDF called *encryptedminutes.pdf*, and wrote the contents of the `PdfFileWriter` to the new PDF. Before anyone can view *encryptedminutes.pdf*, they'll have to enter this password. You may want to delete the original, unencrypted *meetingminutes.pdf* file after ensuring its copy was correctly encrypted.

PROJECT: COMBINING SELECT PAGES FROM MANY PDFs

Say you have the boring job of merging several dozen PDF documents into a single PDF file. Each of them has a cover sheet as the first page, but you don't want the cover sheet repeated in the final result. Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together. Let's write a Python program to customize which pages you want in the combined PDF.

At a high level, here's what the program will do:

1. Find all PDF files in the current working directory.
2. Sort the filenames so the PDFs are added in order.
3. Write each page, excluding the first page, of each PDF to the output file.

In terms of implementation, your code will need to do the following:

1. Call `os.listdir()` to find all the files in the working directory and remove any non-PDF files.
2. Call Python's `sort()` list method to alphabetize the filenames.
3. Create a `PdfFileWriter` object for the output PDF.
4. Loop over each PDF file, creating a `PdfFileReader` object for it.
5. Loop over each page (except the first) in each PDF file.

6. Add the pages to the output PDF.
7. Write the output PDF to a file named *allminutes.pdf*.

For this project, open a new file editor tab and save it as *combinePdfs.py*.

Step 1: Find All PDF Files

First, your program needs to get a list of all files with the *.pdf* extension in the current working directory and sort them. Make your code look like the following:

```
#!/ python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# into a single PDF.

❶ import PyPDF2, os
# Get all the PDF filenames.
pdfFiles = []
for filename in os.listdir('.'):
    if filename.endswith('.pdf'):
        ❷ pdfFiles.append(filename)
❸ pdfFiles.sort(key = str.lower)

❹ pdfWriter = PyPDF2.PdfFileWriter()

# TODO: Loop through all the PDF files.

# TODO: Loop through all the pages (except the first) and add them.

# TODO: Save the resulting PDF to a file.
```

After the shebang line and the descriptive comment about what the program does, this code imports the *os* and *PyPDF2* modules ❶. The *os.listdir('.')* call will return a list of every file in the current working directory. The code loops over this list and adds only those files with the *.pdf* extension to *pdfFiles* ❷. Afterward, this list is sorted in alphabetical order with the *key = str.lower* keyword argument to *sort()* ❸.

A *PdfFileWriter* object is created to hold the combined PDF pages ❹. Finally, a few comments outline the rest of the program.

Step 2: Open Each PDF

Now the program must read each PDF file in *pdfFiles*. Add the following to your program:

```
#!/ python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
```

a single PDF.

```
import PyPDF2, os
```

Get all the PDF filenames.

```
pdfFiles = []
```

--snip--

Loop through all the PDF files.

for filename in pdfFiles:

```
    pdfFileObj = open(filename, 'rb')
```

```
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
```

```
    # TODO: Loop through all the pages (except the first) and add them.
```

```
# TODO: Save the resulting PDF to a file.
```

For each PDF, the loop opens a filename in read-binary mode by calling open() with 'rb' as the second argument. The open() call returns a File object, which gets passed to PyPDF2.PdfFileReader() to create a PdfFileReader object for that PDF file.

Step 3: Add Each Page

For each PDF, you'll want to loop over every page except the first. Add this code to your program:

```
#!/ python3
```

```
# combinePdfs.py - Combines all the PDFs in the current working directory into
```

```
# a single PDF.
```

```
import PyPDF2, os
```

--snip--

Loop through all the PDF files.

```
for filename in pdfFiles:
```

--snip--

```
    # Loop through all the pages (except the first) and add them.
```

```
    ❶ for pageNum in range(1, pdfReader.numPages):
```

```
        pageObj = pdfReader.getPage(pageNum)
```

```
        pdfWriter.addPage(pageObj)
```

```
# TODO: Save the resulting PDF to a file.
```

The code inside the for loop copies each Page object individually to the PdfFileWriter object. Remember, you want to skip the first page. Since PyPDF2 considers 0 to be the first page, your loop should start at 1 **❶** and then go up to, but not include, the integer in pdfReader.numPages.

Step 4: Save the Results

After these nested for loops are done looping, the pdfWriter variable will contain a PdfFileWriter object with the pages for all the PDFs combined. The last step is to write this content to a file on the hard drive. Add this code to your program:

```
#!/ python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# a single PDF.

import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
    --snip--
    # Loop through all the pages (except the first) and add them.
    for pageNum in range(1, pdfReader.numPages):
        --snip--

# Save the resulting PDF to a file.
pdfOutput = open('allminutes.pdf', 'wb')
pdfWriter.write(pdfOutput)
pdfOutput.close()
```

Passing 'wb' to open() opens the output PDF file, *allminutes.pdf*, in write-binary mode. Then, passing the resulting File object to the write() method creates the actual PDF file. A call to the close() method finishes the program.

Ideas for Similar Programs

Being able to create PDFs from the pages of other PDFs will let you make programs that can do the following:

- Cut out specific pages from PDFs.
- Reorder pages in a PDF.
- Create a PDF from only those pages that have some specific text, identified by extractText().

WORD DOCUMENTS

Python can create and modify Word documents, which have the *.docx* file extension, with the `docx` module. You can install the module by running `pip install --user -U python-docx==0.8.10`. ([Appendix A](#) has full details on installing third-party modules.)

NOTE

When using `pip` to first install `Python-Docx`, be sure to install `python-docx`, not `docx`. The package name `docx` is for a different module that this book does not cover. However, when you are going to import the module from the `python-docx` package, you'll need to run `import docx`, not `import python-docx`.

If you don't have Word, LibreOffice Writer and OpenOffice Writer are free alternative applications for Windows, macOS, and Linux that can be used to open *.docx* files. You can download them from <https://www.libreoffice.org/> and <https://openoffice.org/>, respectively. The full documentation for Python-Docx is available at <https://python-docx.readthedocs.io/>. Although there is a version of Word for macOS, this chapter will focus on Word for Windows.

Compared to plaintext, *.docx* files have a lot of structure. This structure is represented by three different data types in Python-Docx. At the highest level, a Document object represents the entire document. The Document object contains a list of Paragraph objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.) Each of these Paragraph objects contains a list of one or more Run objects. The single-sentence paragraph in [Figure 15-4](#) has four runs.

A plain paragraph with some **bold** and some *italic*

Run Run Run Run

Figure 15-4: The Run objects identified in a Paragraph object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it. A *style* in Word is a collection of these attributes. A Run object is a contiguous run of text with the same style. A new Run object is needed whenever the text style changes.

Reading Word Documents

Let's experiment with the `docx` module. Download *demo.docx* from <https://nostarch.com/automatestuff2/> and save the document to the working directory. Then enter the following into the interactive shell:

```
>>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
7
❸ >>> doc.paragraphs[0].text
```


'Document Title'

```
❹ >>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
4
❻ >>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
'bold'
❽ >>> doc.paragraphs[1].runs[2].text
' and some '
❾ >>> doc.paragraphs[1].runs[3].text
'italic'
```

At ❶, we open a *.docx* file in Python, call `docx.Document()`, and pass the filename *demo.docx*. This will return a Document object, which has a `paragraphs` attribute that is a list of Paragraph objects. When we call `len()` on `doc.paragraphs`, it returns 7, which tells us that there are seven Paragraph objects in this document ❷. Each of these Paragraph objects has a `text` attribute that contains a string of the text in that paragraph (without the style information). Here, the first text attribute contains 'DocumentTitle' ❸, and the second contains 'A plain paragraph with some bold and some italic' ❹.

Each Paragraph object also has a `runs` attribute that is a list of Run objects. Run objects also have a `text` attribute, containing just the text in that particular run. Let's look at the text attributes in the second Paragraph object, 'A plain paragraph with some bold and some italic'. Calling `len()` on this Paragraph object tells us that there are four Run objects ❺. The first run object contains 'A plain paragraph with some ' ❻. Then, the text changes to a bold style, so 'bold' starts a new Run object ❼. The text returns to an unbolded style after that, which results in a third Run object, ' and some ' ❽. Finally, the fourth and last Run object contains 'italic' in an italic style ❾.

With Python-Docx, your Python programs will now be able to read the text from a *.docx* file and use it just like any other string value.

Getting the Full Text from a .docx File

If you care only about the text, not the styling information, in the Word document, you can use the `getText()` function. It accepts a filename of a *.docx* file and returns a single string value of its text. Open a new file editor tab and enter the following code, saving it as *readDocx.py*:

```
#!/ python3
```

```
import docx
```

```
def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The `getText()` function opens the Word document, loops over all the Paragraph objects in the paragraphs list, and then appends their text to the list in `fullText`. After the loop, the strings in `fullText` are joined together with newline characters.

The `readDocx.py` program can be imported like any other module. Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

You can also adjust `getText()` to modify the string before returning it. For example, to indent each paragraph, replace the `append()` call in `readDocx.py` with this:

```
fullText.append(' ' + para.text)
```

To add a double space between paragraphs, change the `join()` call code to this:

```
return '\n\n'.join(fullText)
```

As you can see, it takes only a few lines of code to write functions that will read a `.docx` file and return a string of its content to your liking.

Styling Paragraph and Run Objects

In Word for Windows, you can see the styles by pressing CTRL-ALT-SHIFT-S to display the Styles pane, which looks like [Figure 15-5](#). On macOS, you can view the Styles pane by clicking the **View ► Styles** menu item.

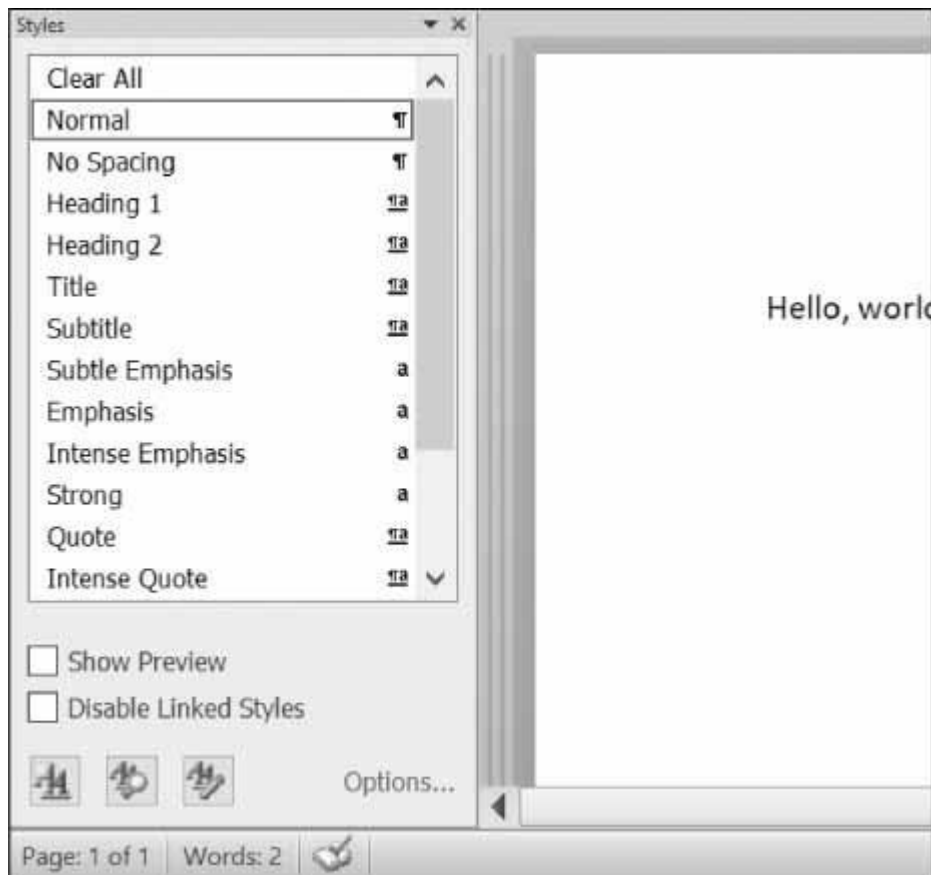


Figure 15-5: Display the Styles pane by pressing CTRL-ALT-SHIFT-S on Windows.

Word and other word processors use styles to keep the visual presentation of similar types of text consistent and easy to change. For example, perhaps you want to set body paragraphs in 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and assign it to all body paragraphs. Then, if you later want to change the presentation of all body paragraphs in the document, you can just change the style, and all those paragraphs will be automatically updated.

For Word documents, there are three types of styles: *paragraph styles* can be applied to Paragraph objects, *character styles* can be applied to Run objects, and *linked styles* can be applied to both kinds of objects. You can give both Paragraph and Run objects styles by setting their style attribute to a string. This string should be the name of a style. If style is set to None, then there will be no style associated with the Paragraph or Run object.

The string values for the default Word styles are as follows:

'Normal'	'Heading 5'	'List Bullet'	'List Paragraph'
'Body Text'	'Heading 6'	'List Bullet 2'	'MacroText'
'Body Text 2'	'Heading 7'	'List Bullet 3'	'No Spacing'
'Body Text 3'	'Heading 8'	'List Continue'	'Quote'
'Caption'	'Heading 9'	'List Continue 2'	'Subtitle'
'Heading 1'	'Intense Quote'	'List Continue 3'	'TOC Heading'
'Heading 2'	'List'	'List Number'	'Title'

'Heading 3'	'List 2'	'List Number 2'
'Heading 4'	'List 3'	'List Number 3'

When using a linked style for a Run object, you will need to add 'Char' to the end of its name. For example, to set the Quote linked style for a Paragraph object, you would use `paragraphObj.style = 'Quote'`, but for a Run object, you would use `runObj.style = 'Quote Char'`.

In the current version of Python-Docx (0.8.10), the only styles that can be used are the default Word styles and the styles in the opened *.docx*. New styles cannot be created—though this may change in future versions of Python-Docx.

Creating Word Documents with Nondefault Styles

If you want to create Word documents that use styles beyond the default ones, you will need to open Word to a blank Word document and create the styles yourself by clicking the **New Style** button at the bottom of the Styles pane (Figure 15-6 shows this on Windows).

This will open the Create New Style from Formatting dialog, where you can enter the new style. Then, go back into the interactive shell and open this blank Word document with `docx.Document()`, using it as the base for your Word document. The name you gave this style will now be available to use with Python-Docx.

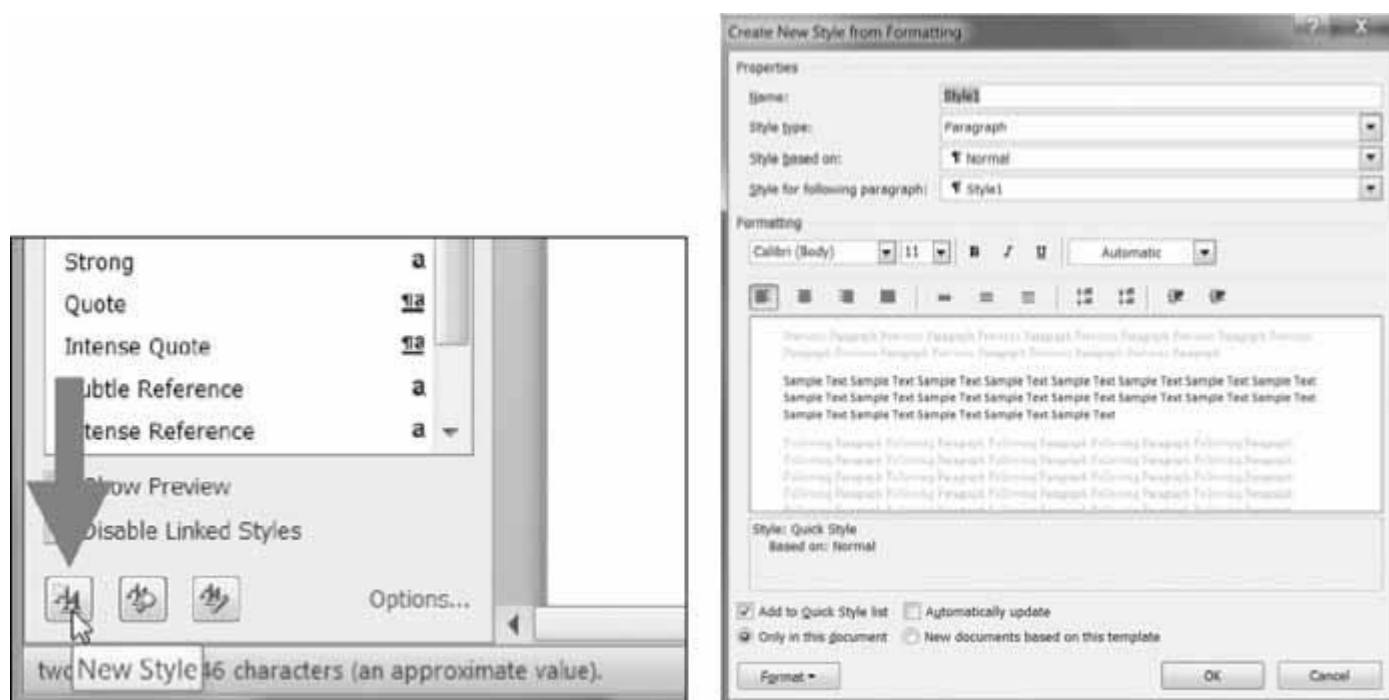


Figure 15-6: The New Style button (left) and the Create New Style from Formatting dialog (right)

Run Attributes

Runs can be further styled using text attributes. Each attribute can be set to one of three values: True (the attribute is always enabled, no matter what other styles are applied to the

run), False (the attribute is always disabled), or None (defaults to whatever the run's style is set to).

Table 15-1 lists the text attributes that can be set on Run objects.

Table 15-1: Run Object text Attributes

Attribute	Description
bold	The text appears in bold.
italic	The text appears in italic.
underline	The text is underlined.
strike	The text appears with strikethrough.
double_strike	The text appears with double strikethrough.
all_caps	The text appears in capital letters.
small_caps	The text appears in capital letters, with lowercase letters two points smaller.
shadow	The text appears with a shadow.
outline	The text appears outlined rather than solid.
rtl	The text is written right-to-left.
imprint	The text appears pressed into the page.
emboss	The text appears raised off the page in relief.

For example, to change the styles of *demo.docx*, enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style # The exact id may be different:
_ParagraphStyle('Title') id: 3095631007984
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Here, we use the text and style attributes to easily see what's in the paragraphs in our document. We can see that it's simple to divide a paragraph into runs and access each run individually. So we get the first, second, and fourth runs in the second paragraph; style each run; and save the results to a new document.

The words *Document Title* at the top of *restyled.docx* will have the Normal style instead of the Title style, the Run object for the text *A plain paragraph with some* will have the QuoteChar style, and the two Run objects for the words *bold* and *italic* will have their underline attributes set to True. [Figure 15-7](#) shows how the styles of paragraphs and runs look in *restyled.docx*.

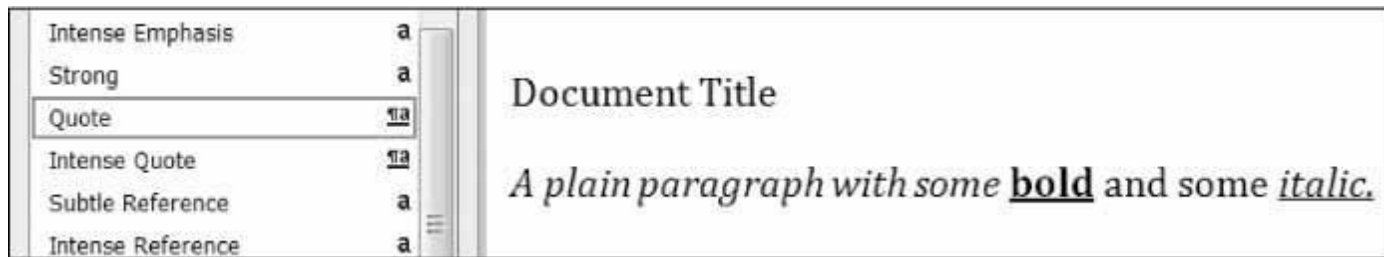


Figure 15-7: The *restyled.docx* file

You can find more complete documentation on Python-Docx's use of styles at <https://python-docx.readthedocs.io/en/latest/user/styles.html>.

Writing Word Documents

Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello, world!')
<docx.text.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

To create your own *.docx* file, call `docx.Document()` to return a new, blank Word Document object. The `add_paragraph()` document method adds a new paragraph of text to the document and returns a reference to the Paragraph object that was added. When you're done adding text, pass a filename string to the `save()` document method to save the Document object to a file.

This will create a file named *helloworld.docx* in the current working directory that, when opened, looks like [Figure 15-8](#).

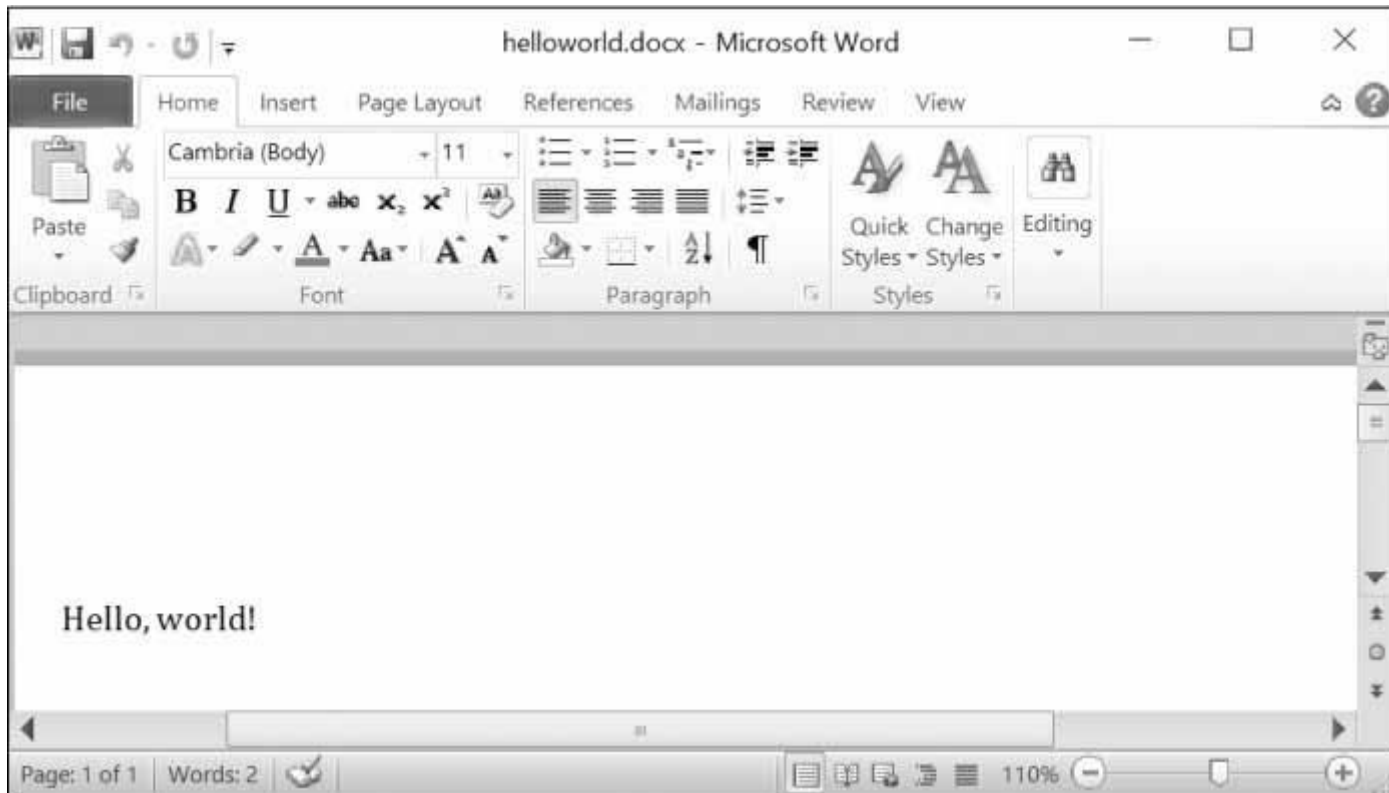


Figure 15-8: The Word document created using `add_paragraph('Hello, world!')`

You can add paragraphs by calling the `add_paragraph()` method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's `add_run()` method and pass it a string. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x000000000366AD30>
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
<docx.text.Run object at 0x0000000003A2C860>
>>> doc.save('multipleParagraphs.docx')
```

The resulting document will look like [Figure 15-9](#). Note that the text *This text is being added to the second paragraph.* was added to the Paragraph object in `paraObj1`, which was the second paragraph added to `doc`. The `add_paragraph()` and `add_run()` functions return paragraph and Run objects, respectively, to save you the trouble of extracting them as a separate step.

Keep in mind that as of Python-Docx version 0.8.10, new Paragraph objects can be added only to the end of the document, and new Run objects can be added only to the end of a Paragraph object.

The `save()` method can be called again to save the additional changes you've made.

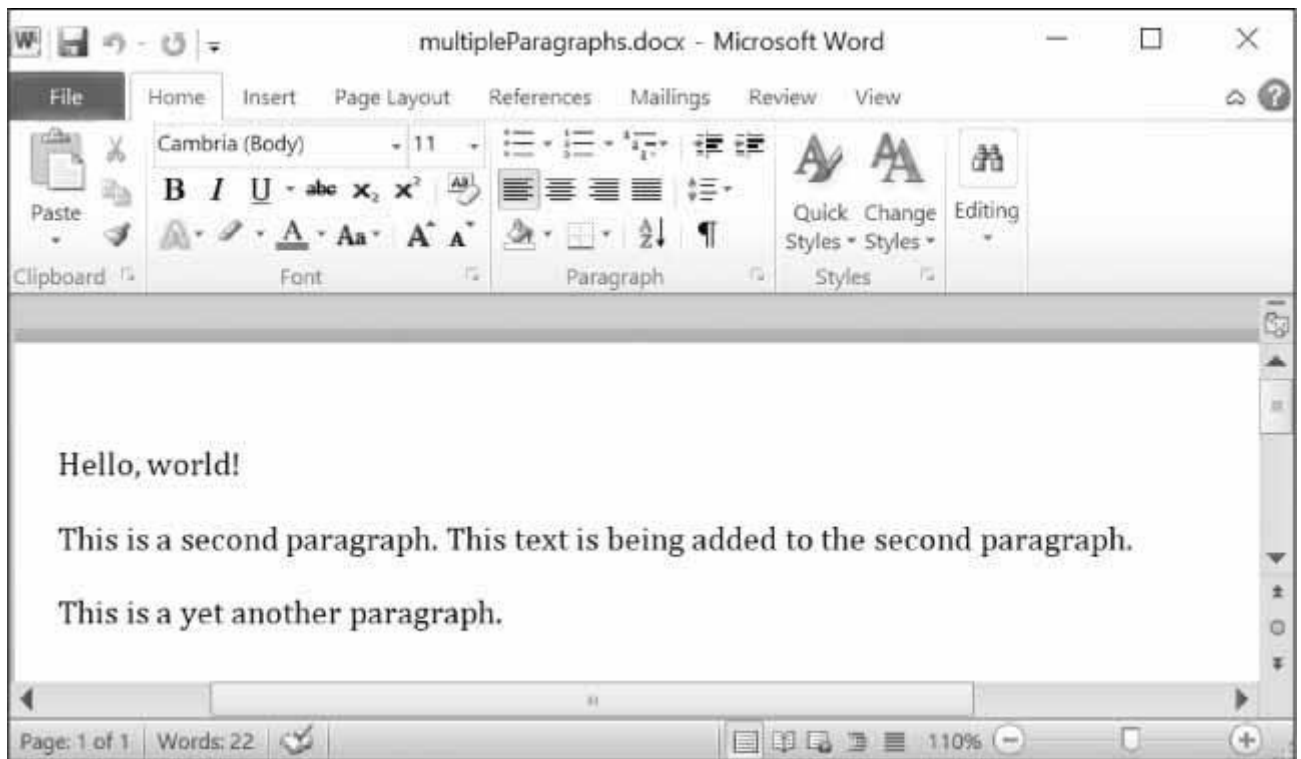


Figure 15-9: The document with multiple Paragraph and Run objects added

Both `add_paragraph()` and `add_run()` accept an optional second argument that is a string of the Paragraph or Run object's style. Here's an example:

```
>>> doc.add_paragraph('Hello, world!', 'Title')
```

This line adds a paragraph with the text *Hello, world!* in the Title style.

Adding Headings

Calling `add_heading()` adds a paragraph with one of the heading styles. Enter the following into the interactive shell:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
<docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.Paragraph object at 0x00000000036CB828>
```

```
>>> doc.add_heading('Header 3', 3)
<docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

The arguments to `add_heading()` are a string of the heading text and an integer from 0 to 4. The integer 0 makes the heading the Title style, which is used for the top of the document. Integers 1 to 4 are for various heading levels, with 1 being the main heading and 4 the lowest subheading. The `add_heading()` function returns a Paragraph object to save you the step of extracting it from the Document object as a separate step.

The resulting *headings.docx* file will look like [Figure 15-10](#).



Figure 15-10: The *headings.docx* document with headings 0 to 4

Adding Line and Page Breaks

To add a line break (rather than starting a whole new paragraph), you can call the `add_break()` method on the Run object you want to have the break appear after. If you want to add a page break instead, you need to pass the value `docx.enum.text.WD_BREAK.PAGE` as a lone argument to `add_break()`, as is done in the middle of the following example:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.enum.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
<docx.text.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```

This creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on

the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

Adding Pictures

Document objects have an `add_picture()` method that will let you add an image to the end of the document. Say you have a file *zophie.png* in the current working directory. You can add *zophie.png* to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),  
height=docx.shared.Cm(4))  
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional width and height keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.

You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you're specifying the width and height keyword arguments.

CREATING PDFs FROM WORD DOCUMENTS

The PyPDF2 module doesn't allow you to create PDF documents directly, but there's a way to generate PDF files with Python if you're on Windows and have Microsoft Word installed. You'll need to install the Pywin32 package by running `pip install --user -U pywin32==224`. With this and the `docx` module, you can create Word documents and then convert them to PDFs with the following script.

Open a new file editor tab, enter the following code, and save it as *convertWordToPDF.py*:

```
# This script runs on Windows only, and you must have Word installed.  
import win32com.client # install with "pip install pywin32==224"  
import docx  
wordFilename = 'your_word_document.docx'  
pdfFilename = 'your_pdf_filename.pdf'  
  
doc = docx.Document()  
# Code to create Word document goes here.  
doc.save(wordFilename)  
  
wdFormatPDF = 17 # Word's numeric code for PDFs.  
wordObj = win32com.client.Dispatch('Word.Application')
```

```
docObj = wordObj.Documents.Open(wordFilename)
docObj.SaveAs(pdfFilename, FileFormat=wdFormatPDF)
docObj.Close()
wordObj.Quit()
```

To write a program that produces PDFs with your own content, you must use the docx module to create a Word document, then use the Pywin32 package's win32com.client module to convert it to a PDF. Replace the # Code to create Word document goes here. comment with docx function calls to create your own content for the PDF in a Word document.

This may seem like a convoluted way to produce PDFs, but as it turns out, professional software solutions are often just as complicated.

SUMMARY

Text information isn't just for plaintext files; in fact, it's pretty likely that you deal with PDFs and Word documents much more often. You can use the PyPDF2 module to read and write PDF documents. Unfortunately, reading text from PDF documents might not always result in a perfect translation to a string because of the complicated PDF file format, and some PDFs might not be readable at all. In these cases, you're out of luck unless future updates to PyPDF2 support additional PDF features.

Word documents are more reliable, and you can read them with the python-docx package's docx module. You can manipulate text in Word documents via Paragraph and Run objects. These objects can also be given styles, though they must be from the default set of styles or styles already in the document. You can add new paragraphs, headings, breaks, and pictures to the document, though only to the end.

Many of the limitations that come with working with PDFs and Word documents are because these formats are meant to be nicely displayed for human readers, rather than easy to parse by software. The next chapter takes a look at two other common formats for storing information: JSON and CSV files. These formats are designed to be used by computers, and you'll see that Python can work with these formats much more easily.

PRACTICE QUESTIONS

1. A string value of the PDF filename is *not* passed to the PyPDF2.PdfFileReader() function. What do you pass to the function instead?
2. What modes do the File objects for PdfFileReader() and PdfFileWriter() need to be opened in?
3. How do you acquire a Page object for page 5 from a PdfFileReader object?
4. What PdfFileReader variable stores the number of pages in the PDF document?
5. If a PdfFileReader object's PDF is encrypted with the password swordfish, what must you do before you can obtain Page objects from it?

6. What methods do you use to rotate a page?
7. What method returns a Document object for a file named *demo.docx*?
8. What is the difference between a Paragraph object and a Run object?
9. How do you obtain a list of Paragraph objects for a Document object that's stored in a variable named doc?
10. What type of object has bold, underline, italic, strike, and outline variables?
11. What is the difference between setting the bold variable to True, False, or None?
12. How do you create a Document object for a new Word document?
13. How do you add a paragraph with the text 'Hello, there!' to a Document object stored in a variable named doc?
14. What integers represent the levels of headings available in Word documents?

PRACTICE PROJECTS

For practice, write programs that do the following.

PDF Paranoia

Using the `os.walk()` function from [Chapter 10](#), write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an *_encrypted.pdf* suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is incorrect, the program should print a message to the user and continue to the next PDF.

Custom Invitations as Word Documents

Say you have a text file of guest names. This *guests.txt* file has one name per line, as follows:

Prof. Plum
Miss Scarlet
Col. Mustard
Al Sweigart
RoboCop

Write a program that would generate a Word document with custom invitations that look like [Figure 15-11](#).

Since Python-Docx can use only those styles that already exist in the Word document, you will have to first add these styles to a blank Word file and then open that file with Python-Docx. There should be one invitation per page in the resulting Word document, so

call `add_break()` to add a page break after the last paragraph of each invitation. This way, you will need to open only one Word document to print all of the invitations at once.



Figure 15-11: The Word document generated by your custom invite script

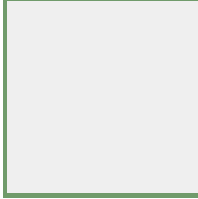
You can download a sample `guests.txt` file from <https://nostarch.com/automatestuff2/>.

Brute-Force PDF Password Breaker

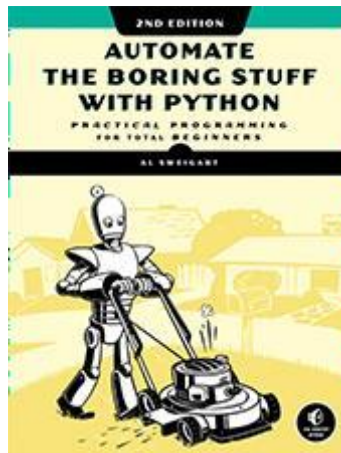
Say you have an encrypted PDF that you have forgotten the password to, but you remember it was a single English word. Trying to guess your forgotten password is quite a boring task. Instead you can write a program that will decrypt the PDF by trying every possible English word until it finds one that works. This is called a *brute-force password attack*. Download the text file `dictionary.txt` from <https://nostarch.com/automatestuff2/>. This dictionary file contains over 44,000 English words with one word per line.

Using the file-reading skills you learned in [Chapter 9](#), create a list of word strings by reading this file. Then loop over each word in this list, passing it to the `decrypt()` method. If this method returns the integer 0, the password was wrong and your program should continue to the next password. If `decrypt()` returns 1, then your program should break out of the loop and print the hacked password. You should try both the uppercase and lowercase form of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)

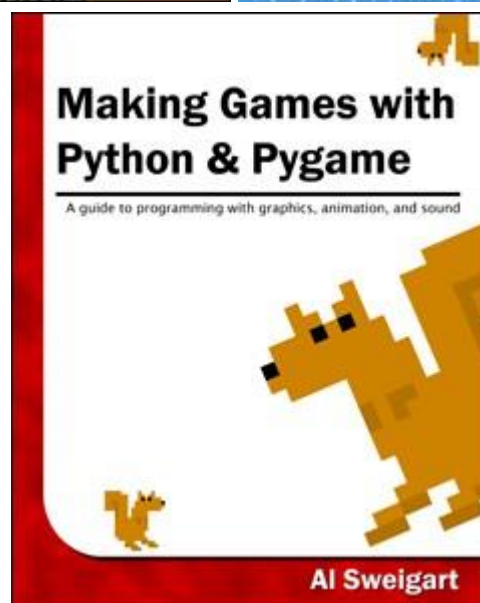
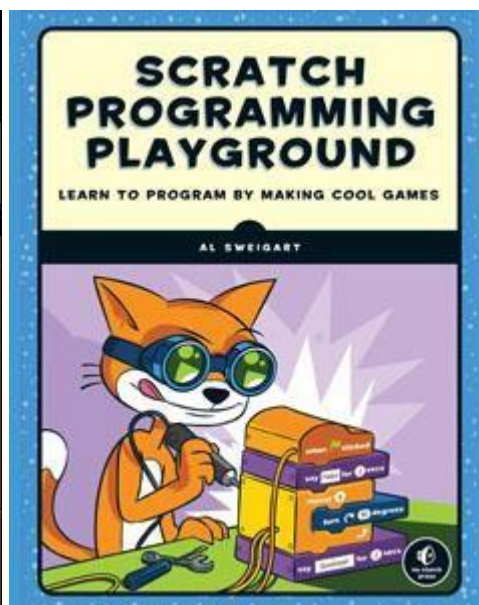
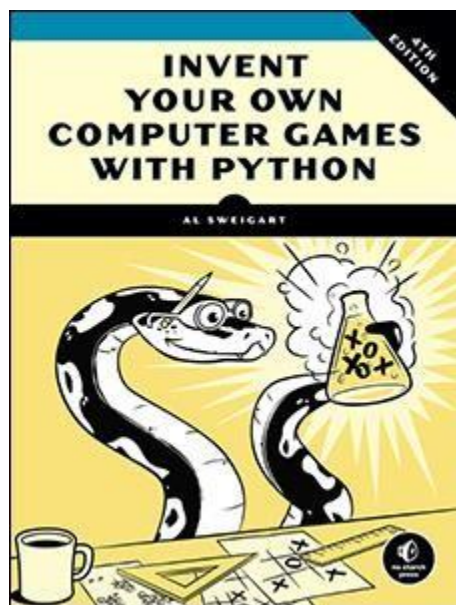
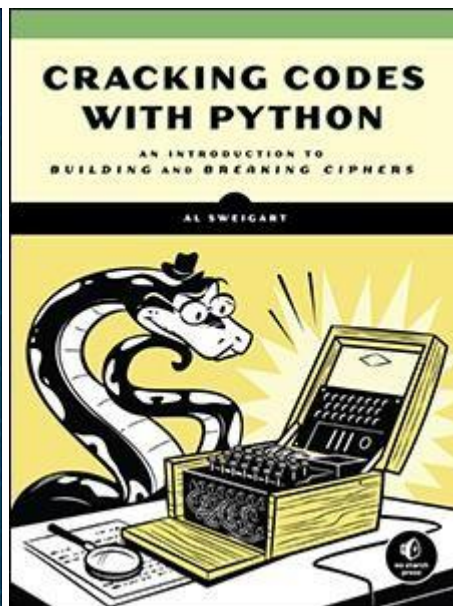
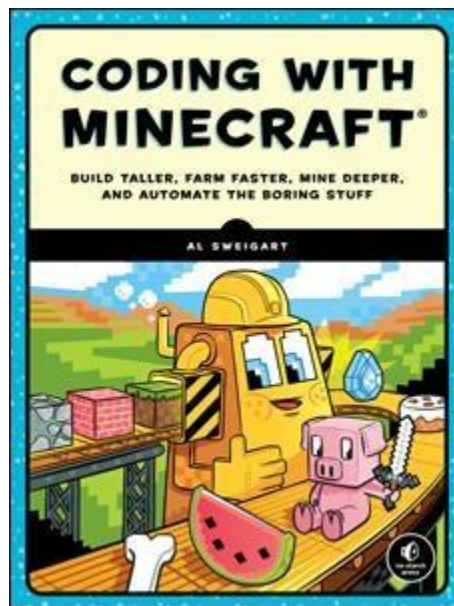
[Home](#) | [Buy on No Starch Press \(comes with free ebook\)](#) | [Buy on Amazon](#) | [@AlSweigart](#) |



Support the author by purchasing the print/ebook bundle from [No Starch Press](#) or separately on [Amazon](#).



Read the author's other Creative Commons licensed Python books.



**WORKING WITH CSV FILES AND JSON DATA**

In [Chapter 15](#), you learned how to extract text from PDF and Word documents. These files were in a binary format, which required special Python modules to access their data. CSV and JSON files, on the other hand, are just plaintext files. You can view them in a text editor, such as Mu. But Python also comes with the special `csv` and `json` modules, each providing functions to help you work with these file formats.

CSV stands for “comma-separated values,” and CSV files are simplified spreadsheets stored as plaintext files. Python’s `csv` module makes it easy to parse CSV files.

JSON (pronounced “JAY-sawn” or “Jason”—it doesn’t matter how because either way people will say you’re pronouncing it wrong) is a format that stores information as JavaScript source code in plaintext files. (JSON is short for JavaScript Object Notation.) You don’t need to know the JavaScript programming language to use JSON files, but the JSON format is useful to know because it’s used in many web applications.

THE CSV MODULE

Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. For example, the spreadsheet *example.xlsx* from <https://nostarch.com/automatestuff2/> would look like this in a CSV file:

4/5/2015	13:34	Apples	73
4/5/2015	3:41	Cherries	85
4/6/2015	12:46	Pears	14
4/8/2015	8:59	Oranges	52
4/10/2015	2:07	Apples	152
4/10/2015	18:10	Bananas	23
4/10/2015	2:40	Strawberries	98

I will use this file for this chapter's interactive shell examples. You can download *example.csv* from <https://nostarch.com/automatestuff2/> or enter the text into a text editor and save it as *example.csv*.

CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, CSV files:

- Don't have types for their values—everything is a string
- Don't have settings for font size or color
- Don't have multiple worksheets
- Can't specify cell widths and heights
- Can't have merged cells
- Can't have images or charts embedded in them

The advantage of CSV files is simplicity. CSV files are widely supported by many types of programs, can be viewed in text editors (including Mu), and are a straightforward way to represent spreadsheet data. The CSV format is exactly as advertised: it's just a text file of comma-separated values.

Since CSV files are just text files, you might be tempted to read them in as a string and then process that string using the techniques you learned in [Chapter 9](#). For example, since each cell in a CSV file is separated by a comma, maybe you could just call `split(',')` on each line of text to get the comma-separated values as a list of strings. But not every comma in a CSV file represents the boundary between two cells. CSV files also have their own set of escape characters to allow commas and other characters to be included *as part of the values*. The `split()` method doesn't handle these escape characters. Because of these potential pitfalls, you should always use the `csv` module for reading and writing CSV files.

reader Objects

To read data from a CSV file with the `csv` module, you need to create a reader object. A reader object lets you iterate over lines in the CSV file. Enter the following into the interactive shell, with *example.csv* in the current working directory:

```
❶ >>> import csv
❷ >>> exampleFile = open('example.csv')
❸ >>> exampleReader = csv.reader(exampleFile)
❹ >>> exampleData = list(exampleReader)
❺ >>> exampleData
[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
 ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
 ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
 ['4/10/2015 2:40', 'Strawberries', '98']]
```

The `csv` module comes with Python, so we can import it ❶ without having to install it first.

To read a CSV file with the csv module, first open it using the open() function ❷, just as you would any other text file. But instead of calling the read() or readlines() method on the File object that open() returns, pass it to the csv.reader() function ❸. This will return a reader object for you to use. Note that you don't pass a filename string directly to the csv.reader() function.

The most direct way to access the values in the reader object is to convert it to a plain Python list by passing it to list() ❹. Using list() on this reader object returns a list of lists, which you can store in a variable like exampleData. Entering exampleData in the shell displays the list of lists ❺.

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression exampleData[row][col], where row is the index of one of the lists in exampleData, and col is the index of the item you want from that list. Enter the following into the interactive shell:

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
>>> exampleData[0][2]
'73'
>>> exampleData[1][1]
'Cherries'
>>> exampleData[6][1]
'Strawberries'
```

As you can see from the output, exampleData[0][0] goes into the first list and gives us the first string, exampleData[0][2] goes into the first list and gives us the third string, and so on.

Reading Data from reader Objects in a for Loop

For large CSV files, you'll want to use the reader object in a for loop. This avoids loading the entire file into memory at once. For example, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
    print('Row #' + str(exampleReader.line_num) + ' ' + str(row))
```

```
Row #1 ['4/5/2015 13:34', 'Apples', '73']
Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
Row #4 ['4/8/2015 8:59', 'Oranges', '52']
```

Row #5 ['4/10/2015 2:07', 'Apples', '152']

Row #6 ['4/10/2015 18:10', 'Bananas', '23']

Row #7 ['4/10/2015 2:40', 'Strawberries', '98']

After you import the csv module and make a reader object from the CSV file, you can loop through the rows in the reader object. Each row is a list of values, with each value representing a cell.

The print() function call prints the number of the current row and the contents of the row. To get the row number, use the reader object's line_num variable, which contains the number of the current line.

The reader object can be looped over only once. To reread the CSV file, you must call csv.reader to create a reader object.

writer Objects

A writer object lets you write data to a CSV file. To create a writer object, you use the csv.writer() function. Enter the following into the interactive shell:

```
>>> import csv
❶ >>> outputFile = open('output.csv', 'w', newline='')
❷ >>> outputWriter = csv.writer(outputFile)
>>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
21
>>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
32
>>> outputWriter.writerow([1, 2, 3.141592, 4])
16
>>> outputFile.close()
```

First, call open() and pass it 'w' to open a file in write mode ❶. This will create the object you can then pass to csv.writer() ❷ to create a writer object.

On Windows, you'll also need to pass a blank string for the open() function's newline keyword argument. For technical reasons beyond the scope of this book, if you forget to set the newline argument, the rows in *output.csv* will be double-spaced, as shown in [Figure 16-1](#).

	A1							
	A	B	C	D	E	F	G	
1	42	2	3	4	5	6	7	
2								
3	2	4	6	8	10	12	14	
4								
5	3	6	9	12	15	18	21	
6								
7	4	8	12	16	20	24	28	
8								
9	5	10	15	20	25	30	35	
10								

Figure 16-1: If you forget the `newline=""` keyword argument in `open()`, the CSV file will be double-spaced.

The `writerow()` method for writer objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file. The return value of `writerow()` is the number of characters written to the file for that row (including newline characters).

This code produces an *output.csv* file that looks like this:

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

Notice how the writer object automatically escapes the comma in the value 'Hello, world!' with double quotes in the CSV file. The csv module saves you from having to handle these special cases yourself.

The delimiter and lineterminator Keyword Arguments

Say you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced. You could enter something like the following into the interactive shell:

```
>>> import csv
>>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
```

32

```
>>> csvFile.close()
```

This changes the delimiter and line terminator characters in your file. The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the *delimiter* and *lineterminator* keyword arguments with `csv.writer()`.

Passing `delimiter='\t'` and `lineterminator='\n\n'` ❶ changes the character between cells to a tab and the character between rows to two newlines. We then call `writerow()` three times to give us three rows.

This produces a file named *example.tsv* with the following contents:

apples oranges grapes

eggs bacon ham

spam spam spam spam spam spam

Now that our cells are separated by tabs, we're using the file extension *.tsv*, for tab-separated values.

DictReader and DictWriter CSV Objects

For CSV files that contain header rows, it's often more convenient to work with the `DictReader` and `DictWriter` objects, rather than the `reader` and `writer` objects.

The `reader` and `writer` objects read and write to CSV file rows by using lists. The `DictReader` and `DictWriter` CSV objects perform the same functions but use dictionaries instead, and they use the first row of the CSV file as the keys of these dictionaries.

Go to <https://nostarch.com/automatestuff2/> and download the *exampleWithHeader.csv* file. This file is the same as *example.csv* except it has `Timestamp`, `Fruit`, and `Quantity` as the column headers in the first row.

To read the file, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('exampleWithHeader.csv')
>>> exampleDictReader = csv.DictReader(exampleFile)
>>> for row in exampleDictReader:
...     print(row['Timestamp'], row['Fruit'], row['Quantity'])
...
4/5/2015 13:34 Apples 73
4/5/2015 3:41 Cherries 85
4/6/2015 12:46 Pears 14
```

4/8/2015 8:59 Oranges 52
4/10/2015 2:07 Apples 152
4/10/2015 18:10 Bananas 23
4/10/2015 2:40 Strawberries 98

Inside the loop, DictReader object sets row to a dictionary object with keys derived from the headers in the first row. (Well, technically, it sets row to an OrderedDict object, which you can use in the same way as a dictionary; the difference between them is beyond the scope of this book.) Using a DictReader object means you don't need additional code to skip the first row's header information, since the DictReader object does this for you.

If you tried to use DictReader objects with *example.csv*, which doesn't have column headers in the first row, the DictReader object would use '4/5/2015 13:34', 'Apples', and '73' as the dictionary keys. To avoid this, you can supply the DictReader() function with a second argument containing made-up header names:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleDictReader = csv.DictReader(exampleFile, ['time', 'name',
'amount'])
>>> for row in exampleDictReader:
...     print(row['time'], row['name'], row['amount'])
...
4/5/2015 13:34 Apples 73
4/5/2015 3:41 Cherries 85
4/6/2015 12:46 Pears 14
4/8/2015 8:59 Oranges 52
4/10/2015 2:07 Apples 152
4/10/2015 18:10 Bananas 23
4/10/2015 2:40 Strawberries 98
```

Because *example.csv*'s first row doesn't have any text for the heading of each column, we created our own: 'time', 'name', and 'amount'.

DictWriter objects use dictionaries to create CSV files.

```
>>> import csv
>>> outputFile = open('output.csv', 'w', newline='')
>>> outputDictWriter = csv.DictWriter(outputFile, ['Name', 'Pet', 'Phone'])
>>> outputDictWriter.writeheader()
>>> outputDictWriter.writerow({'Name': 'Alice', 'Pet': 'cat', 'Phone': '555-
1234'})
20
>>> outputDictWriter.writerow({'Name': 'Bob', 'Phone': '555-9999'})
```

15

```
>>> outputDictWriter.writerow({'Phone': '555-5555', 'Name': 'Carol', 'Pet':  
'dog'})  
20  
>>> outputFile.close()
```

If you want your file to contain a header row, write that row by calling `writeheader()`. Otherwise, skip calling `writeheader()` to omit a header row from the file. You then write each row of the CSV file with a `writerow()` method call, passing a dictionary that uses the headers as keys and contains the data to write to the file.

The *output.csv* file this code creates looks like this:

Name	Pet	Phone
Alice	cat	555-1234
Bob		555-9999
Carol	dog	555-5555

Notice that the order of the key-value pairs in the dictionaries you passed to `writerow()` doesn't matter: they're written in the order of the keys given to `DictWriter()`. For example, even though you passed the `Phone` key and value before the `Name` and `Pet` keys and values in the fourth row, the phone number still appeared last in the output.

Notice also that any missing keys, such as `'Pet'` in `{'Name': 'Bob', 'Phone': '555-9999'}`, will simply be empty in the CSV file.

PROJECT: REMOVING THE HEADER FROM CSV FILES

Say you have the boring job of removing the first line from several hundred CSV files. Maybe you'll be feeding them into an automated process that requires just the data and not the headers at the top of the columns. You *could* open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the `.csv` extension in the current working directory, read in the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

WARNING

As always, whenever you write a program that modifies files, be sure to back up the files first, just in case your program does not work the way you expect it to. You don't want to accidentally erase your original files.

At a high level, the program must do the following:

1. Find all the CSV files in the current working directory.
2. Read in the full contents of each file.
3. Write out the contents, skipping the first line, to a new CSV file.

At the code level, this means the program will need to do the following:

1. Loop over a list of files from `os.listdir()`, skipping the non-CSV files.
2. Create a CSV reader object and read in the contents of the file, using the `line_num` attribute to figure out which line to skip.
3. Create a CSV writer object and write out the read-in data to the new file.

For this project, open a new file editor window and save it as *removeCsvHeader.py*.

Step 1: Loop Through Each CSV File

The first thing your program needs to do is loop over a list of all CSV filenames for the current working directory. Make your *removeCsvHeader.py* look like this:

```
#!/ python3
# removeCsvHeader.py - Removes the header from all CSV files in the current

# working directory.

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Loop through every file in the current working directory.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        ❶ continue # skip non-csv files

    print('Removing header from ' + csvFilename + '...')

    # TODO: Read the CSV file in (skipping first row).

    # TODO: Write out the CSV file.
```

The `os.makedirs()` call will create a `headerRemoved` folder where all the headless CSV files will be written. A for loop on `os.listdir('.')` gets you partway there, but it will loop over *all* files in the working directory, so you'll need to add some code at the start of the loop that skips filenames that don't end with `.csv`. The `continue` statement ❶ makes the for loop move on to the next filename when it comes across a non-CSV file.

Just so there's *some* output as the program runs, print out a message saying which CSV file the program is working on. Then, add some `TODO` comments for what the rest of the program should do.

Step 2: Read in the CSV File

The program doesn't remove the first line from the CSV file. Rather, it creates a new copy of the CSV file without the first line. Since the copy's filename is the same as the original filename, the copy will overwrite the original.

The program will need a way to track whether it is currently looping on the first row. Add the following to *removeCsvHeader.py*.

```
#!/ python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--

# Read the CSV file in (skipping first row).
csvRows = []
csvFileObj = open(csvFilename)
readerObj = csv.reader(csvFileObj)
for row in readerObj:
    if readerObj.line_num == 1:
        continue # skip first row
    csvRows.append(row)
csvFileObj.close()

# TODO: Write out the CSV file.
```

The reader object's `line_num` attribute can be used to determine which line in the CSV file it is currently reading. Another for loop will loop over the rows returned from the CSV reader object, and all rows but the first will be appended to `csvRows`.

As the for loop iterates over each row, the code checks whether `readerObj.line_num` is set to 1. If so, it executes a `continue` to move on to the next row without appending it to `csvRows`. For every row afterward, the condition will be always be False, and the row will be appended to `csvRows`.

Step 3: Write Out the CSV File Without the First Row

Now that `csvRows` contains all rows but the first row, the list needs to be written out to a CSV file in the *headerRemoved* folder. Add the following to *removeCsvHeader.py*:

```
#!/ python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--

# Loop through every file in the current working directory.
```

```
❶ for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue # skip non-CSV files

--snip--

# Write out the CSV file.
csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
                  newline='')
csvWriter = csv.writer(csvFileObj)
for row in csvRows:
    csvWriter.writerow(row)
csvFileObj.close()
```

The CSV writer object will write the list to a CSV file in headerRemoved using csvFilename (which we also used in the CSV reader). This will overwrite the original file.

Once we create the writer object, we loop over the sublists stored in csvRows and write each sublist to the file.

After the code is executed, the outer for loop ❶ will loop to the next filename from os.listdir('.'). When that loop is finished, the program will be complete.

To test your program, download *removeCsvHeader.zip* from <https://nostarch.com/automatestuff2/> and unzip it to a folder. Run the *removeCsvHeader.py* program in that folder. The output will look like this:

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
--snip--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

This program should print a filename each time it strips the first line from a CSV file.

Ideas for Similar Programs

The programs that you could write for CSV files are similar to the kinds you could write for Excel files, since they're both spreadsheet files. You could write programs to do the following:

- Compare data between different rows in a CSV file or between multiple CSV files.
- Copy specific data from a CSV file to an Excel file, or vice versa.
- Check for invalid data or formatting mistakes in CSV files and alert the user to these errors.
- Read data from a CSV file as input for your Python programs.

JSON AND APIS

JavaScript Object Notation is a popular way to format data as a single human-readable string. JSON is the native way that JavaScript programs write their data structures and usually resembles what Python's `pprint()` function would produce. You don't need to know JavaScript in order to work with JSON-formatted data.

Here's an example of data formatted as JSON:

```
{ "name": "Zophie", "isCat": true,  
  "miceCaught": 0, "napsTaken": 37.5,  
  "felineIQ": null }
```

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an *application programming interface (API)*. Accessing an API is the same as accessing any other web page via a URL. The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

Many websites make their data available in JSON format. Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs for programs to use. Some of these sites require registration, which is almost always free. You'll have to find documentation for what URLs your program needs to request in order to get the data you want, as well as the general format of the JSON data structures that are returned. This documentation should be provided by whatever site is offering the API; if they have a "Developers" page, look for the documentation there.

Using APIs, you could write programs that do the following:

- Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.)
- Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook.
- Create a "movie encyclopedia" for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

You can see some examples of JSON APIs in the resources at <https://nostarch.com/automatestuff2/>.

JSON isn't the only way to format data into a human-readable string. There are many others, including XML (eXtensible Markup Language), TOML (Tom's Obvious, Minimal Language), YML (Yet another Markup Language), INI (Initialization), or even the outdated ASN.1 (Abstract Syntax Notation One) formats, all of which provide a structure for representing data as human-readable text. This book won't cover these, because JSON has quickly become the most widely used alternate format, but there are third-party Python modules that readily handle them.

THE JSON MODULE

Python's json module handles all the details of translating between a string with JSON data and Python values for the json.loads() and json.dumps() functions. JSON can't store *every* kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and NoneType. JSON cannot represent Python-specific objects, such as File objects, CSV reader or writer objects, Regex objects, or Selenium WebElement objects.

Reading JSON with the loads() Function

To translate a string containing JSON data into a Python value, pass it to the json.loads() function. (The name means "load string," not "loads.") Enter the following into the interactive shell:

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,
"felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
>>> jsonDataAsPythonValue
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

After you import the json module, you can call loads() and pass it a string of JSON data. Note that JSON strings always use double quotes. It will return that data as a Python dictionary. Python dictionaries are not ordered, so the key-value pairs may appear in a different order when you print jsonDataAsPythonValue.

Writing JSON with the dumps() Function

The json.dumps() function (which means "dump string," not "dumps") will translate a Python value into a string of JSON-formatted data. Enter the following into the interactive shell:

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie',
'felineIQ': None}
>>> import json
>>> stringOfJsonData = json.dumps(pythonValue)
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

The value can only be one of the following basic Python data types: dictionary, list, integer, float, string, Boolean, or None.

PROJECT: FETCHING CURRENT WEATHER DATA

Checking the weather seems fairly trivial: Open your web browser, click the address bar, type the URL to a weather website (or search for one and then click the link), wait for the page to load, look past all the ads, and so on.

Actually, there are a lot of boring steps you could skip if you had a program that downloaded the weather forecast for the next few days and printed it as plaintext. This program uses the requests module from [Chapter 12](#) to download data from the web.

Overall, the program does the following:

1. Reads the requested location from the command line
2. Downloads JSON weather data from OpenWeatherMap.org
3. Converts the string of JSON data to a Python data structure
4. Prints the weather for today and the next two days

So the code will need to do the following:

1. Join strings in sys.argv to get the location.
2. Call requests.get() to download the weather data.
3. Call json.loads() to convert the JSON data to a Python data structure.
4. Print the weather forecast.

For this project, open a new file editor window and save it as *getOpenWeather.py*. Then visit <https://openweathermap.org/api/> in your browser and sign up for a free account to obtain an *API key*, also called an app ID, which for the OpenWeatherMap service is a string code that looks something like '30144aba38018987d84710d0e319281e'. You don't need to pay for this service unless you plan on making more than 60 API calls per minute. Keep the API key secret; anyone who knows it can write scripts that use your account's usage quota.

Step 1: Get Location from the Command Line Argument

The input for this program will come from the command line. Make *getOpenWeather.py* look like this:

```
#!/python3
# getOpenWeather.py - Prints the weather for a location from the command line.

APPID = 'YOUR_APPID_HERE'

import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: getOpenWeather.py city_name, 2-letter_country_code')
    sys.exit()
location = ' '.join(sys.argv[1:])

# TODO: Download the JSON data from OpenWeatherMap.org's API.

# TODO: Load JSON data into a Python variable.
```

In Python, command line arguments are stored in the `sys.argv` list. The `APPID` variable should be set to the API key for your account. Without this key, your requests to the weather service will fail. After the `#!/shebang` line and import statements, the program will check that there is more than one command line argument. (Recall that `sys.argv` will always have at least one element, `sys.argv[0]`, which contains the Python script's filename.) If there is only one element in the list, then the user didn't provide a location on the command line, and a "usage" message will be provided to the user before the program ends.

The OpenWeatherMap service requires that the query be formatted as the city name, a comma, and a two-letter country code (like "US" for the United States). You can find a list of these codes at https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2. Our script displays the weather for the first city listed in the retrieved JSON text. Unfortunately, cities that share a name, like Portland, Oregon, and Portland, Maine, will both be included, though the JSON text will include longitude and latitude information to differentiate between the cities.

Command line arguments are split on spaces. The command line argument San Francisco, US would make `sys.argv` hold `['getOpenWeather.py', 'San', 'Francisco,', 'US']`. Therefore, call the `join()` method to join all the strings except for the first in `sys.argv`. Store this joined string in a variable named `location`.

Step 2: Download the JSON Data

OpenWeatherMap.org provides real-time weather information in JSON format. First you must sign up for a free API key on the site. (This key is used to limit how frequently you make requests on their server, to keep their bandwidth costs down.) Your program simply has to download the page at `https://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3&APPID=<APIkey>`, where `<Location>` is the name of the city whose weather you want and `<APIkey>` is your personal API key. Add the following to *getOpenWeather.py*.

```
#!/python3
# getOpenWeather.py - Prints the weather for a location from the command line.

--snip--

# Download the JSON data from OpenWeatherMap.org's API.
url
='https://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3&APPID=%s '
% (location,
APPID)
response = requests.get(url)
response.raise_for_status()

# Uncomment to see the raw JSON text:
#print(response.text)
```

TODO: Load JSON data into a Python variable.

We have location from our command line arguments. To make the URL we want to access, we use the %s placeholder and insert whatever string is stored in location into that spot in the URL string. We store the result in url and pass url to requests.get(). The requests.get() call returns a Response object, which you can check for errors by calling raise_for_status(). If no exception is raised, the downloaded text will be in response.text.

Step 3: Load JSON Data and Print Weather

The response.text member variable holds a large string of JSON-formatted data. To convert this to a Python value, call the json.loads() function. The JSON data will look something like this:

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
          'country': 'United States of America',
          'id': '5391959',
          'name': 'San Francisco',
          'population': 0},
 'cnt': 3,
 'cod': '200',
 'list': [{'clouds': 0,
           'deg': 233,
           'dt': 1402344000,
           'humidity': 58,
           'pressure': 1012.23,
           'speed': 1.96,
           'temp': {'day': 302.29,
                    'eve': 296.46,
                    'max': 302.29,
                    'min': 289.77,
                    'morn': 294.59,
                    'night': 289.77},
           'weather': [{'description': 'sky is clear',
                        'icon': '01d'}]}],
 'sys': {'type': 1,
         'message': 1,
         'offset': 0,
         'sunrise': 1402338000,
         'sunset': 1402356000},
 'timezone': -420,
 'units': 'SI'}
```

--snip--

You can see this data by passing weatherData to pprint.pprint(). You may want to check <https://openweathermap.org/> for more documentation on what these fields mean. For example, the online documentation will tell you that the 302.29 after 'day' is the daytime temperature in Kelvin, not Celsius or Fahrenheit.

The weather descriptions you want are after 'main' and 'description'. To neatly print them out, add the following to *getOpenWeather.py*.

```
! python3
# getOpenWeather.py - Prints the weather for a location from the command line.

--snip--

# Load JSON data into a Python variable.
weatherData = json.loads(response.text)

# Print weather descriptions.
❶ w = weatherData['list']
print('Current weather in %s:' % (location))
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])
print()
print('Tomorrow:')
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])
print()
print('Day after tomorrow:')
print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

Notice how the code stores `weatherData['list']` in the variable `w` to save you some typing ❶. You use `w[0]`, `w[1]`, and `w[2]` to retrieve the dictionaries for today, tomorrow, and the day after tomorrow's weather, respectively. Each of these dictionaries has a 'weather' key, which contains a list value. You're interested in the first list item, a nested dictionary with several more keys, at index 0. Here, we print the values stored in the 'main' and 'description' keys, separated by a hyphen.

When this program is run with the command line argument `getOpenWeather.py San Francisco, CA`, the output looks something like this:

Current weather in San Francisco, CA:

Clear - sky is clear

Tomorrow:

Clouds - few clouds

Day after tomorrow:

Clear - sky is clear

(The weather is one of the reasons I like living in San Francisco!)

Ideas for Similar Programs

Accessing weather data can form the basis for many types of programs. You can create similar programs to do the following:

- Collect weather forecasts for several campsites or hiking trails to see which one will have the best weather.
- Schedule a program to regularly check the weather and send you a frost alert if you need to move your plants indoors. ([Chapter 17](#) covers scheduling, and [Chapter 18](#) explains how to send email.)
- Pull weather data from multiple sites to show all at once, or calculate and show the average of the multiple weather predictions.

SUMMARY

CSV and JSON are common plaintext formats for storing data. They are easy for programs to parse while still being human readable, so they are often used for simple spreadsheets or web app data. The `csv` and `json` modules greatly simplify the process of reading and writing to CSV and JSON files.

The last few chapters have taught you how to use Python to parse information from a wide variety of file formats. One common task is taking data from a variety of formats and parsing it for the particular information you need. These tasks are often specific to the point that commercial software is not optimally helpful. By writing your own scripts, you can make the computer handle large amounts of data presented in these formats.

In [Chapter 18](#), you'll break away from data formats and learn how to make your programs communicate with you by sending emails and text messages.

PRACTICE QUESTIONS

1. What are some features Excel spreadsheets have that CSV spread-sheets don't?
2. What do you pass to `csv.reader()` and `csv.writer()` to create reader and writer objects?
3. What modes do File objects for reader and writer objects need to be opened in?
4. What method takes a list argument and writes it to a CSV file?
5. What do the `delimiter` and `lineterminator` keyword arguments do?
6. What function takes a string of JSON data and returns a Python data structure?
7. What function takes a Python data structure and returns a string of JSON data?

PRACTICE PROJECT

For practice, write a program that does the following.

Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the `openpyxl` module

from Chapter 12, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

A single Excel file might contain multiple sheets; you'll have to create one CSV file per *sheet*. The filenames of the CSV files should be `<excel filename>_<sheet title>.csv`, where `<excel filename>` is the filename of the Excel file without the file extension (for example, 'spam_data', not 'spam_data.xlsx') and `<sheet title>` is the string from the Worksheet object's title variable.

This program will involve many nested for loops. The skeleton of the program will look something like this:

```
for excelFile in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheetName in wb.get_sheet_names():
        # Loop through every sheet in the workbook.
        sheet = wb.get_sheet_by_name(sheetName)

        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for rowNum in range(1, sheet.max_row + 1):
            rowData = [] # append each cell to this list
            # Loop through each cell in the row.
            for colNum in range(1, sheet.max_column + 1):
                # Append each cell's data to rowData.

            # Write the rowData list to the CSV file.

csvFile.close()
```

Download the ZIP

file `excelSpreadsheets.zip` from <https://nostarch.com/automatestuff2/> and unzip the spreadsheets into the same directory as your program. You can use these as the files to test the program on.