VISUAL QUICKSTART GUIDE

Python

Third Edition

TOBY DONALDSON

LEARN THE QUICK AND EASY WAY!

From the Library of Bill Wiecking

VISUAL QUICKSTART GUIDE

Python

TOBY DONALDSON

Peachpit Press

From the Library of Bill Wiecking

Visual QuickStart Guide
Python, Third Edition
Toby Donaldson

Peachpit Press
www.peachpit.com

To report errors, please send a note to errata@peachpit.com
Peachpit Press is a division of Pearson Education

Copyright © 2014 by Toby Donaldson

Editor: Scout Festa
Production Editor: Katerina Malone
Compositor: David Van Ness
Indexer: Valerie Haynes Perry
Cover Design: RHDG / Riezebos Holzbaur Design Group, Peachpit Press
Interior Design: Peachpit Press
Logo Design: MINE™ www.minesf.com

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Visual QuickStart Guide is a registered trademark of Peachpit Press, a division of Pearson Education.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN-13: 978-0-321-92955-6
ISBN-10: 0-321-92955-1

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

From the Library of Bill Wiecking

Acknowledgments

Thanks to Clifford Colby and Scout Festa for their expertise and patience in bringing this edition of the book to life; to the many students at SFU who continue to teach me how best to learn Python; to John Edgar and the other computer science teachers at SFU with whom I've had the pleasure to work; and to Bonnie, Thomas, and Emily for recommending I avoid using the word *blithering* more than once in these acknowledgments. And a special thank you to Guido van Rossum and the rest of the Python community for creating a programming language that is so much fun to use.

From the Library of Bill Wiecking

Contents at a Glance

Chapter 1	Introduction to Programming	1
Chapter 2	Arithmetic, Strings, and Variables	9
Chapter 3	Writing Programs	31
Chapter 4	Flow of Control	43
Chapter 5	Functions	67
Chapter 6	Strings	83
Chapter 7	Data Structures	101
Chapter 8	Input and Output	123
Chapter 9	Exception Handling	143
Chapter 10	Object-Oriented Programming	153
Chapter 11	Case Study: Text Statistics	177
Appendix A	Popular Python Packages	195
Appendix B	Comparing Python 2 and Python 3	199
	Index	203

From the Library of Bill Wiecking

Table of Contents

Chapter 1 Introduction to Programming 1

The Python Language 2
What Is Python Useful For? 3
How Programmers Work 4
Installing Python 6

Chapter 2 Arithmetic, Strings, and Variables 9

The Interactive Command Shell 10
Integer Arithmetic 11
Floating Point Arithmetic 13
Other Math Functions 16
Strings 17
String Concatenation 19
Getting Help 20
Converting Between Types 22
Variables and Values 24
Assignment Statements 26
How Variables Refer to Values 28
Multiple Assignment 29

Chapter 3 Writing Programs 31

Using IDLE's Editor 32
Compiling Source Code 35
Reading Strings from the Keyboard 36
Printing Strings on the Screen 39
Source Code Comments 41
Structuring a Program 42

Chapter 4 Flow of Control 43

Boolean Logic 44
If-Statements 49
Code Blocks and Indentation 51
Loops 54

From the Library of Bill Wiecking

Comparing For-Loops and While-Loops 59
Breaking Out of Loops and Blocks 64
Loops Within Loops 66

Chapter 5 Functions 67

Calling Functions 68
Defining Functions 70
Variable Scope 73
Using a main Function 75
Function Parameters 76
Modules 80

Chapter 6 Strings 83

String Indexing 84
Characters 87
Slicing Strings 89
Standard String Functions 92
Regular Expressions 98

Chapter 7 Data Structures 101

The type Command 102
Sequences 103
Tuples 104
Lists 108
List Functions 110
Sorting Lists 113
List Comprehensions 115
Dictionaries 118
Sets 122

Chapter 8 Input and Output 123

Formatting Strings 124
String Formatting 126
Reading and Writing Files 128
Examining Files and Folders 131
Processing Text Files 134
Processing Binary Files 138
Reading Webpages 141

From the Library of Bill Wiecking

Chapter 9 Exception Handling 143

Exceptions 144
Catching Exceptions 146
Clean-Up Actions 150

Chapter 10 Object-Oriented Programming 153

Writing a Class 154
Displaying Objects 156
Flexible Initialization 160
Setters and Getters 162
Inheritance 168
Polymorphism 171
Learning More 175

Chapter 11 Case Study: Text Statistics 177

Problem Description 178
Keeping the Letters We Want 180
Testing the Code on a Large Data File 182
Finding the Most Frequent Words 184
Converting a String to a Frequency Dictionary 187
Putting It All Together 188
Exercises 190
The Final Program 192

Appendix A Popular Python Packages 195

Some Popular Packages 196

Appendix B Comparing Python 2 and Python 3 199

What's New in Python 3 200

Index 203

From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

1

Introduction to Programming

Before we dive into the details of Python programming, it helps to learn a bit about what Python is and what kinds of programs it is used for. We will also outline exactly what it is that programmers do. Finally, we'll learn how to install Python and run the IDLE editor that comes with it.

If you are new to programming, this short introduction should help you get your footing in preparation for learning the Python programming language.

If you already have a grasp of the basic concepts, feel free to jump ahead to the sections on how to install Python and run the editor.

In This Chapter

The Python Language	2
What Is Python Useful For?	3
How Programmers Work	4
Installing Python	6

From the Library of Bill Wiecking

The Python Language

So what is Python? Briefly, it is a computer *programming language* and a corresponding set of software tools and libraries. It was originally developed in the early 1990s by Guido van Rossum, and it is now actively maintained by dozens of programmers around the world (including van Rossum).

Python was designed to be easy to read and learn. Compared with programs written in most other programming languages, Python programs look neat and clean: Python has few unnecessary symbols, and it uses straightforward English names.

Python is a very productive language: Once you're proficient with Python, you can get more done with it in less time than you can in most other programming languages. Python supports—but doesn't force you to use—*object-oriented programming* (OOP).

Python comes with a wide range of ready-made libraries that can be used in your own programs; as some Python programmers like to say, Python comes with "batteries included."

A very practical feature of Python is its *maintainability.* Since Python programs are relatively easy to read and modify, they are easy for programmers to keep up to date. Program maintenance can easily account for 50 percent or more of the work a programmer does, and so Python's support for maintenance is a big win in the eyes of many professionals.

Finally, a word about the name. According to Python's originator, Guido van Rossum, Python was named after the Monty Python comedy troupe. Despite this mirthful origin, Python now uses a pair of iconic blue and yellow snakes—presumably pythons—as its standard symbol.

From the Library of Bill Wiecking

What Is Python Useful For?

While Python is a general-purpose language that can be used to write any kind of program, it is especially popular for the following applications:

- **Scripts.** These short programs automate common administrative tasks, such as adding new users to a system, uploading files to a website, downloading webpages without using a browser, and so on.
- **Website development.** A number of Python projects—such as Django (www.djangoproject.com), Bottle (www.bottlepy.org), and Zope (www.zope.org)—are popular among developers as tools for quickly creating dynamic websites. For instance, the popular news site www.reddit.com was written using Python.
- **Text processing.** Python has excellent support for handling strings and text files, including regular expressions and Unicode.
- **Scientific computing.** Many superb scientific Python libraries are available on the web, providing functions for statistics, mathematics, and graphing.
- **Education.** Thanks to its relative simplicity and utility, Python is becoming more and more popular as a first programming language in schools.

Of course, Python isn't the best choice for all projects. It is often slower than languages such as Java, C#, or C++. So, for example, you wouldn't use Python to create a new operating system.

But when you need to minimize the amount of time a *programmer* spends on a project, Python is often an excellent choice.

From the Library of Bill Wiecking

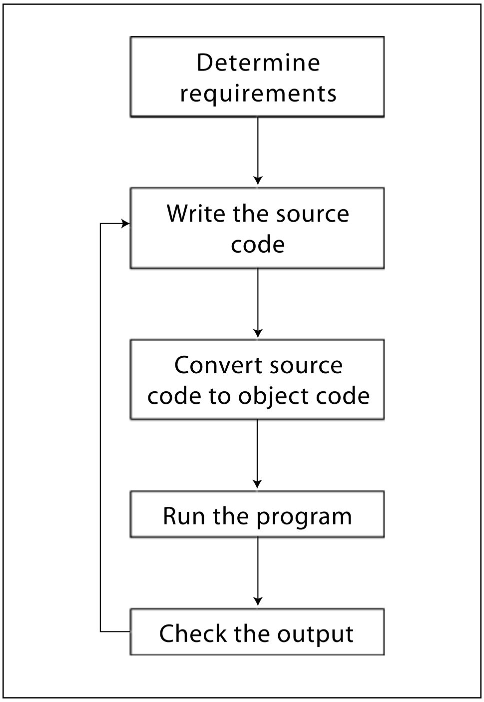
How Programmers Work

While there is no strict recipe for writing programs, most programmers follow a similar process.

The programming process

1. Determine what your program is supposed to do—that is, figure out its *requirements*.
2. Write the source code (in our case, the Python code) in IDLE (Python's integrated development environment) or any other text editor. This is often the most interesting and challenging step, and it often involves creative problem solving. Python source code files end with **`.py`**: **`web.py`**, **`urlexpand.py`**, **`clean.py`**, and so on.
3. Convert the source code to *object code* using the Python *interpreter*. Python puts object code in **`.pyc`** files. For example, if your source code is in **`urlexpand.py`**, its object code will be put in **`urlexpand.pyc`**.
4. *Run*, or *execute,* the program. With Python, this step is usually done immediately and automatically after step 2 is finished. In practice, Python programmers rarely work directly with object code or **`.pyc`** files.

From the Library of Bill Wiecking



A The basic steps of writing any computer program. Typically, after you check your program output, you find errors and so must go back to the code-writing step to fix them.

Lingo Alert

We typically call the contents of a `.py` file a *program*, *source code*, or just *code*.

Object code is sometimes referred to as *executable code*, the *executable*, or even just *software*.

5. Finally, *check* the program's output. If errors are discovered, go back to step 2 to try to fix them. The process of fixing errors is called *debugging*. For large or complex programs, debugging can sometimes take up most of the program development time, so experienced programmers try to design their programs in ways that will minimize debugging time.

As **A** shows, this is an *iterative* process: You write your program, test it, fix errors, test it again, and so on until the program behaves correctly.

From the Library of Bill Wiecking

Installing Python

Python is a hands-on language, so now we will see how to install it on your computer.

To install Python on Windows:

1. Go to the Python download page at www.python.org/download.
2. Choose the most recent version of Python 3 (it should have a name like Python 3.*x*, where *x* is some small number). This will take you to the appropriate download page with instructions for downloading Python on different computer systems.
3. Click the appropriate installer link for your computer. For instance, if you are running Windows, click Windows x86 MSI Installer (3.x).
4. Once the installer has finished downloading, run it by double-clicking it.
5. After the installation has finished (which could take a few minutes), test to see that Python is installed properly. Open the Windows Start menu and choose All Programs. You should see an entry for Python 3.0 (often highlighted in yellow). Select IDLE (Python GUI), and wait a moment for the IDLE program to launch **B**.
6. Try typing in **24 * 7** and pressing Return. The number 168 should appear.

B The starting screen of the IDLE editor. The first line tells you which version of Python you are using—in this case, it is version 3.0b1.

From the Library of Bill Wiecking

Installing Python on the Mac

OS X comes with a version of Python already installed, although it lacks the IDLE editor and is typically not the most up-to-date version. To install a more recent version of Python, follow the instructions given at www.python.org/download/mac/. Or, just download and run an installer from www.pythonmac.org/packages/. Be careful to ensure that you have the right version of Python (3.0 or better) and that the Mac OS version number matches yours.

Installing Python on Linux

If you are using Linux, chances are you already have Python installed. To find out, open a command-line window and type **python**. If you get something similar to the text shown in B, then Python is working.

Be sure to check the version number: This book covers Python 3. If you have Python 2.x or earlier, then you should install Python 3.

The exact details for doing so will depend upon your Linux system. For example, on Ubuntu Linux, you would search for Python in the Synaptic Package Manager. You can also get Linux installation help from www.python.org/download.

From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

2

Arithmetic, Strings, and Variables

The first step in learning how to program is to understand the basic Python *data types*: integers (whole numbers), floating point numbers (numbers with a decimal point), and strings. All programs use these (and other) data types, so it is important to have a good grasp of their basic uses.

Strings, in particular, are used in so many different kinds of programs that Python provides a tremendous amount of support for them. In this chapter, we'll introduce the basics of strings, and then we'll return to them in a later chapter.

We'll also introduce the important concept of a programming *variable*. Variables are used to store and manipulate data, and it's hard to write a useful program without employing at least a few of them.

Just like learning how to play the piano or speak a foreign language, the best way to learn how to program is to *practice*. Thus, we'll introduce all of this using the interactive command shell IDLE, and ideally you should follow along on your own computer by typing the examples as we go.

In This Chapter

The Interactive Command Shell	10
Integer Arithmetic	11
Floating Point Arithmetic	13
Other Math Functions	16
Strings	17
String Concatenation	19
Getting Help	20
Converting Between Types	22
Variables and Values	24
Assignment Statements	26
How Variables Refer to Values	28
Multiple Assignment	29

From the Library of Bill Wiecking

The Interactive Command Shell

Let's see how to interact with the Python shell. Start IDLE; you should find it listed as a program in your Start menu on Windows. On Mac or Linux you should be able to run it directly from a command line by typing **python**. The window that pops up is the Python *interactive command shell*, and it looks something like what's shown in **A**.

The shell prompt

In a Python transcript, **>>>** is the Python *shell prompt*. A **>>>** marks a line of input from you, the user, while lines without a **>>>** are generated by Python. Thus it is easy to distinguish at a glance what is from Python and what is from you.

Transcripts

A *shell transcript* is a snapshot of the command shell showing a series of user inputs and Python replies. We'll be using them frequently; they're a great way to learn Python by seeing real examples in action.

Lingo Alert

The interactive command shell is often abbreviated as *interactive shell*, *command shell*, *shell*, or even *command line*.

Shell transcripts are sometimes called *transcripts*, *interactive sessions*, or just *sessions*.

A What you should see when you first launch the Python interactive command shell. The top two lines tell you what version of Python you are running. The version you see here is Python 3.30, and it was created a little before 11 a.m. on September 29, 2012.

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```


From the Library of Bill Wiecking

Integer Arithmetic

An integer is a whole number, such as 25, –86, or 0. Python supports the four basic arithmetic operations: **+** (addition), **-** (subtraction), ***** (multiplication), and **/** (division). Python also uses ****** for exponentiation and **%** to calculate remainders (for example, **25 % 7** is 4 because 7 goes into 25 three times, with 4 left over). For example:

```
>>> 5 + 9
14
>>> 22 - 6
16
>>> 12 * 14
168
>>> 22 / 7
3.1428571428571428
>>> 2 ** 4
16
>>> 25 % 7
4
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

Integer division

Python also has an *integer division* operator, **//**. It works like **/**, except that it always returns an integer. For example, **7 // 3** evaluates to the integer 2; the digits after the decimal are simply chopped off (**//** doesn't round!).

From the Library of Bill Wiecking

Order of evaluation

Table 2.1 summarizes Python's basic arithmetic operators. They are grouped from lowest *precedence* to highest precedence. For example, when Python evaluates the expression **`1 + 2 * 3`**, it evaluates **`*`** before **`+`** because **`*`** has higher precedence (so the expression evaluates to 7—not 9!). Operators at the same level of precedence are evaluated in the order they are written. You can use *round brackets*, **`()`**, to change the order of evaluation—so, for example, **`(1 + 2) * 3`** evaluates to 9. In other words, Python arithmetic follows the same evaluation rules as regular arithmetic.

Unlimited size

Unlike most other programming languages, Python puts no limit on the size of an integer. You can do calculations involving numbers with dozens (or hundreds, or thousands) of digits:

```
>>> 27 ** 100
136891479058588375991326027382088315
→ 966463695625337436471480190078368
→ 997177499076593800206155688941388
→ 250484440597994042813512732765695
→ 774566001
```

TABLE 2.1 Basic Arithmetic Operators

Name	Operator	Example
addition	+	>>>3 + 4 7
subtraction	–	>>> 5 – 3 2
multiplication	*	>>> 2 * 3 6
division	/	>>> 3 / 2 1.5
integer division	//	>>> 3 // 2 1
remainder	%	>>> 25 % 3 1
exponentiation	**	>>> 3 ** 3 27

From the Library of Bill Wiecking

Floating Point Arithmetic

Floating point arithmetic is done with *floating point numbers,* which in Python are numbers that contain a decimal point. For instance, –3.1, 2.999, and –4.0 are floating point numbers. We'll call them *floats* for short.

All the basic arithmetic operations that work with integers also work with floats, even **%** (remainder) and **//** (integer division). See **B** for some examples.

B Examples of basic floating point arithmetic using the Python command shell. Notice that approximation errors are quite common, so exact values are often not printed.

```
>>> 3.8 + -43.2
-39.400000000000006
>>> 12.6 * 0.5
6.3
>>> 12.6 + 0.01
12.61
>>> 365.0 / 12
30.416666666666668
>>> 8.8 ** -5.4
7.939507629591553e-06
>>> 5.6 // 2
2.0
>>> 5.6 % 3.2
2.3999999999999995
```

Float literals

Very large or small floats are often written in *scientific notation*:

```
>>> 8.8 ** -5.4
7.939507629591553e-06
```

The **e-06** means to multiply the preceding number by 10^{-6}. You can use scientific notation directly if you like:

```
>>> 2.3e02
230.0
```

Python is quite forgiving about the use of decimal points:

```
>>> 3.
3.0
>>> 3.0
3.0
```

You can write numbers like 0.5 with or without the leading 0:

```
>>> .5
0.5
>>> 0.5
0.5
```

TIP It's usually clearer to write 5.0 instead of 5., as the latter notation can be quite confusing—it looks like the end of a sentence.

TIP The difference between 5 and 5.0 matters: 5 is an integer, while 5.0 is a floating point number. Their internal representations are significantly different.

From the Library of Bill Wiecking

Overflow

Unlike integers, floating point numbers have minimum and maximum values that, if exceeded, will cause *overflow errors*. An overflow error means you've tried to calculate a number that Python cannot represent as a float, because it is either too big or too small **C**. Overflow errors can be *silent errors*, meaning that Python does the calculation incorrectly without telling you that anything bad has happened. Generally speaking, it is up to you, the programmer, to avoid overflow errors.

Limited precision

Precision (or *accuracy*) is a fundamental difficulty with floats on all computers. Numbers are represented in binary (base 2) in a computer, and it turns out that not all floating point numbers can be represented precisely in binary. Even the simplest examples can have problems:

```
>>> 1 - 2 / 3
0.33333333333333337
```

This should have an infinite number of 3s after the decimal, but there are only (!) 17 digits here. Plus, the last digit is wrong (the 7 should be a 3).

C Floating point overflow: **`500.0 ** 10000`** is too big to store as a float.

```
>>> 500.0 ** 10000
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    500.0 ** 10000
OverflowError: (34, 'Result too large')
```


From the Library of Bill Wiecking

These small errors are not usually a problem: 17 digits after the decimal is enough for most programs. However, little errors like this have a nasty habit of becoming big errors when you are doing lots of calculations. If you are, say, computing the stresses on a newly designed bridge, it is necessary to take small floating point errors into account to ensure that they don't balloon into significant errors.

TIP **In general, you should prefer integers to floating point numbers. They are always accurate and never suffer overflow.**

Complex numbers

Python has built-in support for complex numbers—that is, numbers that involve the square root of –1. In Python, **`1j`** denotes the square root of –1:

```
>>> 1j
1j
>>> 1j * 1j
(-1+0j)
```

Complex numbers are useful in certain engineering and scientific calculations; we won't be using them again in this book.

From the Library of Bill Wiecking

Other Math Functions

Python comes with many different *modules* of prewritten code, including the **`math`** module. **Table 2.2** lists some of the most commonly used **`math`** module functions.

Using return values

We say that these functions *return* a value. That means they *evaluate* to either an integer or a floating point number, depending on the function.

You can use these functions anywhere that you can use a number. Python automatically evaluates the function and replaces it with its return value.

Importing a module

To use the **`math`** module, or any existing Python module, you must first *import* it:

```
>>> import math
```

You can now access any math function by putting **`math.`** in front of it:

```
>>> math.sqrt(5)
2.23606797749979
>>> math.sqrt(2) * math.tan(22)
0.012518132023611912
```

An alternative way of importing a module is this:

```
>>> from math import *
```

Now you can call all the math module functions *without* first appending **`math.`**:

```
>>> log(25 + 5)
3.4011973816621555
>>> sqrt(4) * sqrt(10 * 10)
20.0
```

TABLE 2.2 Some `math` Module Functions

Name	Description
ceil(x)	Ceiling of x
cos(x)	Cosine of x
degrees(x)	Converts x from radians to degrees
exp(x)	e to the power of x
factorial(n)	Calculates n! = 1*2*3*...*n n must be an integer
log(x)	Base e logarithm of x
log(x, b)	Base b logarithm of x
pow(x, y)	x to the power of y
radians(x	Converts x from degrees to radians
sin(x)	Sine of x
sqrt(x)	Square root of x
tan(x)	Tangent of x

TIP When using the `from math import *` style of importing, if you have functions with the same name as any of the functions in the `math` module, the `math` functions will overwrite them!

TIP Thus, it's generally *safer* to use the `import math` style of importing. This will never overwrite existing functions.

TIP You can also import specific functions from the `math` module. For example, `from math import sqrt, tan` imports just the `sqrt` and `tan` functions.

From the Library of Bill Wiecking

Strings

A string is a sequence of one or more characters, such as **`"cat!"`**, **`"567-45442"`**, and **`"Up and Down"`**. Characters include letters, numbers, punctuation, plus hundreds of other special symbols and unprintable characters.

Indicating a string

Python lets you write *string literals* in three main ways:

- Single quotes, such as **`'http'`**, **`'open house'`**, or **`'cat'`**
- Double quotes, such as **`"http"`**, **`"open house"`**, or **`"cat"`**
- Triple quotes, such as **`"""http"""`**, or multiline strings, such as

  ```
  """
  Me and my monkey
  Have something to hide
  """
  ```

TIP Many Python programmers prefer using single quotes to indicate strings, simply because they involve less typing than double quotes (which require pressing the Shift key).

TIP One of the main uses of single and double quotes is to conveniently handle `"` and `'` characters inside strings:

```
"It's great"
'She said "Yes!"'
```

TIP You'll get an error if you use the wrong kind of quote within a string.

TIP Triple quotes are useful when you need to create long, multiline strings. They can also contain `"` and `'` characters at the same time.

From the Library of Bill Wiecking

String length

To determine the number of characters in a string, use the **`len(s)`** function:

```
>>> len('pear')

4

>>> len('up, up, and away')

16

>>> len("moose")

5

>>> len("")

0
```

The last example uses the *empty string,* usually denoted by **`''`** or **`""`**. The empty string has zero characters in it.

Since **`len`** evaluates to (that is, returns) an integer, we can use **`len`** anywhere that an integer is allowed—for example:

```
>>> 5 + len('cat') * len('dog')

14
```


From the Library of Bill Wiecking

String Concatenation

You can create new strings by "adding" together old strings:

```
>>> 'hot ' + 'dog'
'hot dog'
>>> 'Once' + " " + 'Upon' + ' ' +
→ "a Time"
'Once Upon a Time'
```

This operation is known as *concatenation*.

There's a neat shortcut for concatenating the same string many times:

```
>>> 10 * 'ha'
'hahahahahahahahahaha'
>>> 'hee' * 3
'heeheehee'
>>> 3 * 'hee' + 2 * "!"
'heeheehee!!'
```

The result of string concatenation is always another string, so you can use concatenation anywhere that requires a string:

```
>>> len(12 * 'pizza pie!')
120
>>> len("house" + 'boat') * '12'
'121212121212121212'
```


From the Library of Bill Wiecking

Getting Help

Python is a largely *self-documenting* language. Most functions and modules come with brief explanations to help you figure out how to use them without resorting to a book or website.

Listing functions in a module

Once you've imported a module, you can list all of its functions using the **dir(m)** function:

```
>>> import math
>>> dir(math)
['__doc__', '__name__',
→'__package__', 'acos', 'acosh',
→'asin', 'asinh', 'atan', 'atan2',
→'atanh', 'ceil', 'copysign',
→'cos', 'cosh', 'degrees', 'e',
→'exp', 'fabs', 'factorial',
→'floor', 'fmod', 'frexp', 'hypot',
→'isinf', 'isnan', 'ldexp', 'log',
→'log10', 'log1p', 'modf', 'pi',
→'pow', 'radians', 'sin', 'sinh',
→'sqrt', 'sum', 'tan', 'tanh',
→'trunc']
```

This gives you a quick overview of the functions in a module, and many Python programmers use **dir(m)** all the time.

For now, you can ignore the names beginning with a double underscore __; they are used only in more advanced Python programming.

From the Library of Bill Wiecking

TIP To see a list of all the built-in functions in Python, type `dir(__builtins__)` at the command prompt.

TIP An alternative way to see the doc string for a function `f` is to use the `help(f)` function.

TIP You can run the ***Python help utility*** by typing `help()` at a prompt. This will provide you with all kinds of useful information, such as a list of all available modules, help with individual functions and keywords, and more.

TIP You can also get help from the Python documentation (www.python.org/doc/). There you'll find helpful tutorials, plus complete details of all the Python language and standard modules.

Printing documentation strings

Another useful trick is to print a function's *documentation string* (*doc string* for short):

```
>>> print(math.tanh.__doc__)
tanh(x)
Return the hyperbolic tangent of x.
```

Most built-in Python functions, along with most functions in Python's standard modules (such as **math**), have short doc strings you can access in this way.

As another example, here's the doc string for the built-in function **bin**:

```
>>> print(bin.__doc__)
bin(number) -> string
Return the binary representation of
→ an integer or long integer.
>>> bin(25)
'0b11001'
```


From the Library of Bill Wiecking

Converting Between Types

Converting from one type of data to another is a common task, and Python provides a number of built-in functions to make this easy.

Converting integers and strings to floats:

To convert the integer 3 to a float, use the **`float(x)`** function:

```
>>> float(3)
3.0
```

Converting a string to a float is similar:

```
>>> float('3.2')
3.2000000000000002
>>> float('3')
3.0
```

Converting integers and floats to strings:

The **`str(n)`** function converts any number to a corresponding string:

```
>>> str(85)
'85'
>>> str(-9.78)
'-9.78'
```

Implicit Conversions

Sometimes Python will convert between numeric types without requiring an explicit conversion function. For example:

```
>>> 25 * 8.5
212.5
```

Here, 25 is automatically converted to 25.0, and then multiplied by 8.5. In general, when you mix integers and floats in the same expression, Python *automatically* converts the integers to floats.

From the Library of Bill Wiecking

Rounding

Many people are surprised that **`round(8.5)`** is 8 in Python, and not 9. In elementary school, you were probably taught that numbers ending with .5 should always be rounded up.

But always rounding up leads to a bias that can cause inaccurate calculations. So Python uses a different strategy for rounding, called "round half to even," or, sometimes, "bankers rounding." The idea is that numbers ending in .5 are rounded to the nearest even integer. Thus, sometimes numbers ending in .5 are rounded down, and sometimes they are rounded up.

This strategy might seem strange at first, and it is different from how rounding works in Python 2. But it is generally accepted as the standard way to round numbers on a computer. If you are curious about the details, take a look at the Wikipedia entry on rounding: http://en.wikipedia.org/wiki/Rounding.

Converting a float to an integer:

This is a little tricky because you must decide how to handle any digits after the decimal in your float. The **`int(x)`** function simply chops off extra digits, while **`round(x)`** does the usual kind of rounding off:

```
>>> int(8.64)
8
>>> round(8.64)
9
>>> round(8.5)
8
```

Converting strings to numbers:

This is easily done with the **`int(s)`** and **`float(s)`** functions:

```
>>> int('5')
5
>>> float('5.1')
5.1
```

TIP For most applications, you should be able to handle numeric conversions using `int(x)`, `float(x)`, and `round(x)`. However, for more specific conversions, the Python `math` module has a number of functions for removing digits after decimals: `math.trunc(x)`, `math.ceil(x)`, and `math.floor(x)`.

TIP The `int(s)` and `float(s)` conversions from strings to floats/integers assume that the string `s` "looks" like a Python float/integer. If not, you'll get an error message saying the conversion could not be done.

From the Library of Bill Wiecking

Variables and Values

Variables are one of the most important concepts in all of programming. In Python, variables *label*, or *point to*, a value.

For example:

```
>>> fruit = "cherry"
>>> fruit
'cherry'
```

Here, `fruit` is a variable name, and it points to the string value `"cherry"`. Notice that variables are *not* surrounded by quotation marks.

The line `fruit = "cherry"` is called an *assignment statement*. The `=` (equals sign) is called the *assignment operator* and is used to make a variable point to a value.

When Python encounters a variable, it replaces it with the value it points to. Thus:

```
>>> cost = 2.99
>>> 0.1 * cost
0.29900000000000004
>>> 1.06 * cost + 5.99
9.1594000000000015
```

Lingo Alert

Just like variables, functions, modules, and classes all have names. We refer to these names collectively as *identifiers*.

From the Library of Bill Wiecking

TABLE 2.3 Legal and Illegal Variable Names

Legal	Illegal
`M`	`"m"`
`x1`	`1x`
`tax_rate`	`tax rate`
`taxRate`	`taxRate!`
`Else`	`else`

D **`else`** is a Python keyword, so it cannot be used as a variable.

```
>>> else = 25
SyntaxError: invalid syntax
```

Rules for making variable names

Variable names must follow a few basic rules (see **Table 2.3** for some examples):

- A variable name can be of any length, although the characters in it must be either letters, numbers, or the underscore character (_). Spaces, dashes, punctuation, quotation marks, and other such characters are *not* allowed.
- The first character of a variable name *cannot* be a number; it must be a letter or an underscore character.
- Python is *case sensitive*—it distinguishes between uppercase and lowercase letters. Thus **`TAX`**, **`Tax`**, and **`tax`** are three completely different variable names.
- You cannot use Python *keywords* as variable names. For example, **`if`**, **`else`**, **`while`**, **`def`**, **`or`**, **`and`**, **`not`**, **`in`**, and **`is`** are some of Python's keywords (we'll learn what these are used for later in the book). If you try to use one as a variable, you'll get an error **D**.

From the Library of Bill Wiecking

Assignment Statements

Assignment statements have three main parts: a *left-hand side*, the *assignment operator*, and a *right-hand side* E. Assignment statements have two purposes: They *define* new variable names, and they make already-defined variables point to values. For instance:

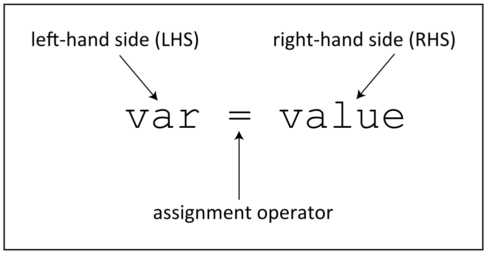
```
>>> x = 5
>>> 2 * x + 1
11
>>> x = 99
```

The first assignment statement, **x = 5**, does double duty: It is an *initialization statement*. It tells Python to create a new variable named **x** and that it should be assigned the value 5. We can now use **x** anywhere an integer can be used.

The second assignment statement, **x = 99**, reassigns **x** to point to a different value. It does *not* create **x**, because **x** already exists thanks to the previous assignment statement.

If you don't initialize a variable, Python complains with an error:

```
>>> 2 * y + 1
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in
  → <module>
    2 * y + 1
NameError: name 'y' is not defined
```



E Anatomy of an assignment statement. This makes **var** point to **value**. The left-hand side must always be a variable, while the right-hand side can be a variable, a value, or *any* expression that evaluates to a value.

From the Library of Bill Wiecking

Lingo Alert

A number of terms are commonly used to describe variables and values. We sometimes say a variable is *assigned* a value, or *given* a value.

A variable with an assigned value is said to *point* to its value, or *label* it, or simply *have* it.

Sometimes programmers say a variable *contains* its value, as if the variable were a bucket and the value was inside of it. The problem with this is that Python variables don't quite follow the rules you would expect a "containment" model to follow. For instance, an object can be in only one bucket at a time, but multiple values are allowed to point to the same value in Python.

This error message tells you that the variable **`y`** has not been defined, and so Python does not know what value to replace it with in the expression **`2 * y + 1`**.

A variable can be assigned any value, even if it comes from other variables. Consider this sequence of assignments:

```
>>> x = 5
>>> x
5
>>> y = 'cat'
>>> y
'cat'
>>> x = y
>>> x
'cat'
>>> y
'cat'
```


From the Library of Bill Wiecking

How Variables Refer to Values

A Python assignment statement of the form **`x = expr`** can be summarized in English like this: *Make **x** point to the value that **expr** evaluates to.*

Keep in mind that **`expr`** can be *any* Python expression that evaluates to a value.

There's a nice way of drawing diagrams to help understand sequences of assignments. For example, after the assignment **`rate = 0.04`**, you can imagine that your computer's memory looks like F. Then, after **`rate_2008 = 0.06`**, we get G. Finally, **`rate = rate_2008`** gives us H.

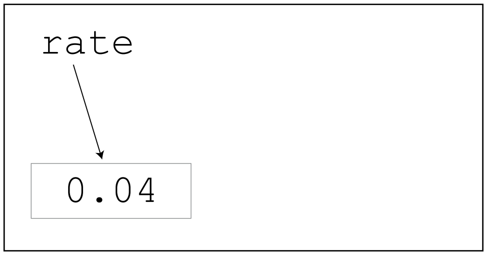
When a value no longer has any variable pointing to it (for example, 0.04 in H), Python automatically deletes it. In general, Python keeps track of all values and automatically deletes them when they are no longer referenced by a variable. This is called *garbage collection*, and so Python programmers rarely need to worry about deleting values themselves.

Assignments don't copy

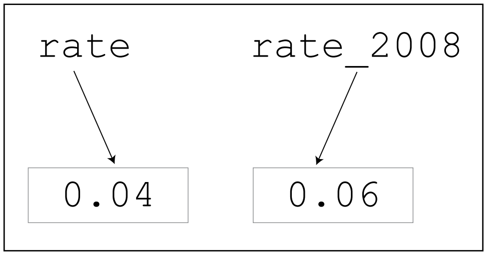
It's essential to understand that assignment statements *don't make a copy* of the value they point to. All they do is label, and relabel, existing values. Thus, no matter how big or complex the object a variable points to, assignment statements are always exceedingly efficient.

Numbers and strings are immutable

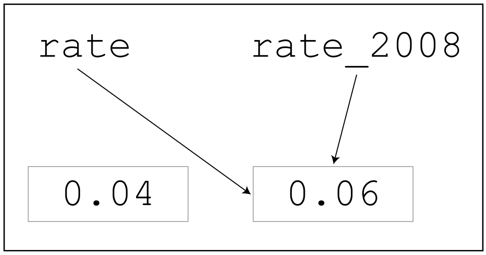
An important feature of Python numbers and strings is that they are *immutable*—that is, they cannot be changed in *any* way, ever. Whenever it seems that you are modifying a number or string, Python is in fact making a modified copy I.



F After running the statement **`rate = 0.04`**.



G After **`rate_2008 = 0.06`**.



H After **`rate = rate_2008`**. Notice that the value 0.04 no longer has any variable pointing to it. Thus Python automatically deletes it, a process known as *garbage collection*.

I Whenever it appears that you are modifying a string, Python is in fact making a copy. There is no way to modify numbers or strings in Python.

```
>>> s = 'apple'
>>> s + 's'
'apples'
>>> s
'apple'
>>> 5 = 1
SyntaxError: can't assign to literal
```


From the Library of Bill Wiecking

Multiple Assignment

Python has a convenient trick that lets you assign more than one variable at a time:

```
>>> x, y, z = 1, 'two', 3.0
>>> x
1
>>> y
'two'
>>> z
3.0
>>> x, y, z
(1, 'two', 3.0)
```

As the last statement shows, you can also display multiple values on one line by writing them as a *tuple*. Tuples always begin with an open round bracket (**(**) and end with a closed round bracket (**)**).

From the Library of Bill Wiecking

Swapping variable values

A useful trick you can do with multiple assignment is to *swap* the values of two variables:

```
>>> a, b = 5, 9
>>> a, b
(5, 9)
>>> a, b = b, a
>>> a, b
(9, 5)
```

The statement **a, b = b, a** is said to assign values to **a** and **b** *in parallel*. Without using multiple assignment, the standard way to swap variables is like this:

```
>>> a, b = 5, 9
>>> temp = a
>>> a = b
>>> b = temp
>>> a, b
(9, 5)
```

Multiple assignment doesn't do anything you can't already do with regular assignment. It is just a convenient shorthand that we will sometimes be using.

From the Library of Bill Wiecking

3

Writing Programs

Up to now, we've been writing single Python statements and running them at the interactive command line. While that's useful for learning about Python functions, it quickly becomes tiresome when you need to write many lines of Python code.

Thus we turn to writing *programs* (also known as *scripts*). Programs are just text files containing a collection of Python commands. When you *run* (or *execute*) a program, Python performs each statement in the file one after the other.

In this chapter, we'll learn how to write and run programs in IDLE and from the command line. We'll see how to get keyboard input from the user and print strings to the screen.

You should make an effort to type the code yourself, since it is an excellent way to get used to the various rules of writing Python. For larger programs, you can download the code from this book's website: http://pythonintro.googlecode.com.

In This Chapter

Using IDLE's Editor	32
Compiling Source Code	35
Reading Strings from the Keyboard	36
Printing Strings on the Screen	39
Source Code Comments	41
Structuring a Program	42

From the Library of Bill Wiecking

Using IDLE's Editor

IDLE comes with a Python-aware text editor. The best way to learn about it is to write a simple program.

To write a new program in IDLE:

1. Launch IDLE.
2. Choose File > New Window.

   A blank editor window should pop up.
3. To test it, enter the following into it:

   ```
   print('Welcome to Python!')
   ```
4. Save your program by choosing File > Save. Save it in your Python programs folder with the name **`welcome.py`**; the **`.py`** at the end indicates that this is a Python file.
5. Run your program by choosing Run > Run Module.

   A Python shell should appear, and you should see *Welcome to Python!* within it.

When you start to get more familiar with the IDLE editor, you may want to start using some of the key commands listed in **Table 3.1**. They can really speed up your editing.

TIP Create a special folder called, say, *python* on your computer's Desktop to store all your Python programs. *Never* save them in the Python directory; otherwise, you run the risk of accidentally overwriting one of Python's core files.

TIP You must type in Python programs *exactly* as you see them, character for character. A single wrong character—an extra space, an `l` instead of a 1—can cause errors.

TIP If you do see an error when you run your program, go back to the editor window and carefully check that your program was typed correctly, character for character.

TABLE 3.1 Some Useful IDLE Shortcuts

Command	What It Does
Ctrl-N	Open a new editor window.
Ctrl-O	Open a new file for editing.
Ctrl-S	Save the current program.
F5	Run the current program.
Ctrl-Z	Undo the last action.
Shift-Ctrl-Z	Redo the last undo.

From the Library of Bill Wiecking

Other Editors

IDLE is an excellent editor for beginners, and even some professionals use it all the time. But if IDLE is not to your liking, a quick web search for "programming editors" will give you many other suggestions. For instance, on Windows, Notepad++ is a popular free programming editor. Another popular, although not free, choice is Sublime Text, which works on Windows, Mac, and Linux systems.

Take a look at http://wiki.python.org/moin/PythonEditors for many more suggestions.

Running programs from the command line

Another common way to run a Python program is from the command line. For example, to run **welcome.py**, you can open a command-line window and run it by typing this:

```
C:\> python welcome.py
Welcome to Python!
```

You can also just call Python without a program and get a bare-bones (but still quite useful) version of the interactive interpreter.

To call Python from the command line:

Type the following:

```
C:\> python
Python 3.0b2 (r30b2:65106, Jul 18
→ 2008, 18:44:17) [MSC v.1500 32 bit
→ (Intel)] on win32
Type "help", "copyright", "credits"
→ or "license" for more information.
>>>
```

Calling Python from the command line is most commonly used when you run Python scripts as parts of other programs.

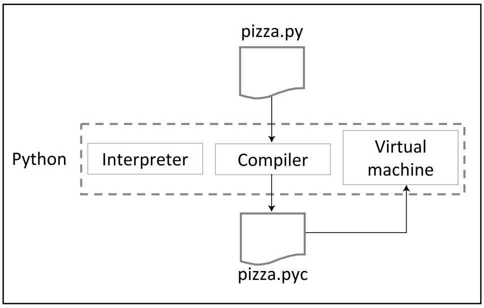
From the Library of Bill Wiecking

TIP The easiest way to open a command window in Windows is to click the Start menu; then type cmd in the search box and press Return. This should give you a command-line window.

TIP Running Python from the command line is similar on Mac and Linux systems: run a command shell (the exact details for doing this differ from system to system, but try browsing programs available through menus on your Desktop), and then type `python` followed by the name of the program you want to run.

TIP One annoyance with running Python from the command line is that it is often necessary to configure *environment variables*, in particular your system's *path* variable, so that your system knows where to find Python on your computer. The details are finicky and system specific, and are beyond the scope of this book. However, it is not hard to find detailed instructions online if you want to set this up. For instance, just type set windows path into your favorite search engine. Take care when you are modifying environment variables: If you are not sure exactly what you are doing, it is quite possible to "break" your system so that programs no longer run correctly. In that case, your best option is usually to start over and reinstall Python.

From the Library of Bill Wiecking



A Python consists of three major components: an interpreter for running single statements; a compiler for converting `.py` files to `.pyc` files; and a virtual machine for running `.pyc` files. Note that IDLE is *not* strictly part of Python; it is a separate application that sits on top of Python to make it easier to use.

Compiling Source Code

We often refer to the statements inside a Python program as *source code*, and so a program file is sometimes called a *source code file*, or *source file*. By convention, all Python source code files end with the extension `.py`. This makes it easy for people and programs to see at a glance that the file contains Python source code.

Object code

When you run a `.py` file, Python automatically creates a corresponding `.pyc` file **A**. A `.pyc` file contains *object code*, or *compiled code*. Object code is essentially a Python-specific language that represents your Python source code in a way that is easier for the computer to run efficiently. It is not meant for humans to read, and so most of the time you should just ignore the `.pyc` files that start to appear.

A Python program runs using a special piece of software called a *virtual machine*. This is essentially a software simulation of a computer designed just to run Python, and it is part of the reason why many `.pyc` files can run on different computer systems without any change.

TIP You will rarely, if ever, have to worry about `.pyc` files. Python automatically creates them when needed, and also automatically updates them when you change the corresponding `.py` files. Don't delete, rename, or modify the `.pyc` files!

TIP Since they are meant to be read only by the computer, `.pyc` files are *not* stored as text files. If you try to view a `.pyc` file in a text editor, you'll see nothing but junk characters.

From the Library of Bill Wiecking

Reading Strings from the Keyboard

Reading a string from the keyboard is one of the most basic ways of getting information from a user. For example, consider this simple program:

```
# name.py

name = input('What is your first
→ name? ')

print('Hello ' + name.capitalize()
→ + '!')
```

To run this in IDLE, open **name.py** in an IDLE window, and then to run it press F5 (or, equivalently, choose Run > Run Module).

You should see this in the window that appears:

```
What is your first name? jack

Hello Jack!
```

You, the user, must type in the name (in this case, the string **'jack'**).

Tracing the program

Let's look carefully at each line of the program. The first line is a *source code comment*, or *comment* for short. A comment is just a note to the programmer, and Python ignores it. Python comments always start with a **#** symbol and continue to the end of the line. This particular comment tells you that the program is stored in a file called **name.py**.

The second line calls the **input** function, which is the standard built-in function for reading strings from the keyboard. When it runs, the prompt **'What is your name?'** appears in the output window, followed by a blinking cursor. The program waits until the user enters a string *and* presses Enter.

From the Library of Bill Wiecking

The **`input`** function evaluates to whatever string the user enters, and so the variable **`name`** ends up labeling the string that the user types in.

The third and final line of the program displays a greeting. The function **`name.capitalize()`** ensures that the first character of the string is uppercase and the remaining characters are lowercase. This way, if the user happens to enter a name that isn't correctly capitalized, Python will correct it.

TIP To see what functions are available for strings, type `dir('')` at IDLE's interactive command line.

TIP If you run `name.py` with a number of sample strings, you'll soon discover that entering a name like `'Jack Aubrey'` will actually *un*capitalize the last name: `'Hello Jack aubrey!'`. That's because the `capitalize` function is very simpleminded—it knows nothing about words or spaces.

TIP Another common and useful trick when reading strings from the keyboard is to use the `strip()` function to remove any leading/trailing whitespace characters. For instance:

```
>>> ' oven '.strip()
'oven'
```

Stripping a string of unwanted spaces is so common that we often write calls to input like this:

```
name = input('Enter age: ').strip()
```


From the Library of Bill Wiecking

Reading numbers from the keyboard

The **input** function only returns strings, so if you need a number data type (for example, to do arithmetic), you must use one of Python's numeric conversion functions. For example, consider this program:

```
# age.py
age = input('How old are you
→ today? ')
age10 = int(age) + 10
print('In 10 years you will be ' +
→ str(age10) + ' years old.')
```

Suppose the user types in **22** in response to this program. Then the variable **age** labels the *string* **'22'**—Python does *not* automatically convert strings that look like numbers to integer or float values. If you want to do arithmetic with a string, you must first convert it to a number using either **int(s)** (if you want an integer) or **float(s)** (if you want a float).

The one final trick to notice is that in the **print** statement, it's necessary to convert the variable **age10** (which labels an integer) back into a string so that it can be printed. If you forget this conversion, Python issues an error saying it can't add numbers and strings.

Different Types of Numbers

All the different types of numbers can be confusing at first. Consider these four different values: **5**, **5.0**, **'5'**, and **'5.0'**. While they look similar, they have very different internal representations.

5 is an integer and can be used directly for arithmetic.

5.0 is a floating point number that can also be used for arithmetic, but it allows for digits after the decimal place.

'5' and **'5.0'** are strings consisting of one and three characters, respectively. Strings are meant for being displayed on the screen or for doing character-based operations (such as removing whitespace or counting characters). Strings can't be used to do numeric arithmetic. Of course, strings can be used with concatenation, although the results might be a bit jarring at first—for example:

```
>>> 3 * '5'
'555'
>>> 3 * '5.0'
'5.05.05.0'
```


From the Library of Bill Wiecking

Lingo Alert

Programmers often use the terminology *standard output*, abbreviated *stdout*, to refer to the window where text goes when printed. Typically, stdout is a simple text window that does little more than display strings: No graphics of any kind are allowed.

Similarly, *standard input*, abbreviated *stdin*, is the location from where the input function reads strings. Usually this is the same window as stdout, but it is possible to change one or both of stdout and stdin if necessary.

You will sometimes also see the term *standard error*, abbreviated *stderr*, to refer to where error messages are displayed. By default, error messages are usually displayed on stdout.

Printing Strings on the Screen

The **`print`** statement is the standard built-in function for printing strings to the screen. As we will see, it is extremely flexible and has many useful features for formatting strings and numbers in just the right way.

You can pass any number of strings to **`print`**:

```
>>> print('jack', 'ate', 'no', 'fat')
jack ate no fat
```

By default, it prints out each string in the *standard output* window, separating the strings with a space. You can easily change the string separator like this:

```
>>> print('jack', 'ate', 'no', 'fat',
→ sep = '.')
jack.ate.no.fat
```

By default, a printed string ends with a *newline* character: **`\n`**. A newline character causes the cursor to move to the next line when the string is printed, and so, by default, you can't print anything on the same line after calling **`print`**:

```
# jack1.py
print('jack ate ')
print('no fat')
```

This prints two lines of text:

```
jack ate
no fat
```

To put all the text on a single line, you can specify the end character of the first line to be the empty string:

```
# jack2.py
print('jack ate ', end = '')
print('no fat')
```


From the Library of Bill Wiecking

TIP The `print` function is one of the major differences between Python 2 and Python 3. Before Python 3, `print` was not technically a function, but instead was a built-in part of the language. The one advantage of this was that you didn't have to type the brackets—for example, you would type `print('jack ate no fat')`. However, despite that small convenience, `print`'s not being a function made it very difficult to change the default separator and ending strings, which is often necessary in more advanced programs.

TIP Another difference between Python 2 and 3 is that Python 3's `input` function was called `raw_input` in Python 2. Python 2 also had a function called `input`, but it evaluated the string that the user entered, which was occasionally handy. There is no equivalent of the Python 2 `input` function in Python 3, although you can easily simulate it by typing `eval(input(prompt))`. For example:

```
>>> eval(input('? '))
? 4 + 5 * 6
34
```


From the Library of Bill Wiecking

Source Code Comments

We've already seen source code comments used to specify the name of a file. But comments are useful for any kind of note that you might want to put into a program, such as documentation, reminders, explanations, or warnings. Python ignores all comments, and they are only there to be read by you and other programmers who might read the source code.

Here's a sample program that shows some more uses of comments:

```
# coins_short.py
# This program asks the user how many
# coins of various types they have,
# and then prints the total amount
# of money in pennies.
# get the number of nickels, dimes,
# and quarters from the user
n = int(input('Nickels? '))
d = int(input('Dimes? '))
q = int(input('Quarters? '))
# calculate the total amount of money
total = 5 * n + 10 * d + 25 * q
# print the results
print() # prints a blank line
print(str(total) + ' cents')
```

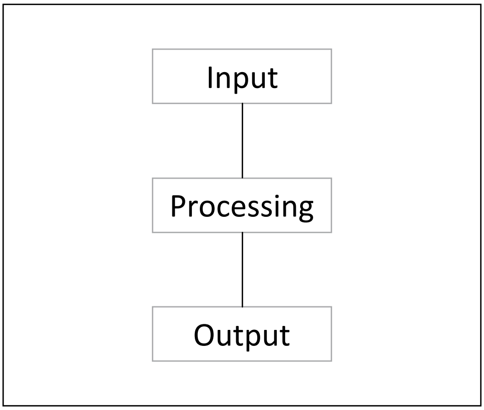

From the Library of Bill Wiecking

Structuring a Program

As you start to write more programs, you will soon notice that they tend to follow a common structure. Typically, programs are organized as in B: They have an input part, a processing part, and an output part.

For the small programs that we are starting out with, this structure is usually obvious and does not require much thought. But as your programs get bigger and more complex, it is easy to lose sight of this overall structure, which often results in messy code that is hard to understand.

Thus, indicating in comments what parts are for input, processing, and output is a good habit to get into. It helps clarify the different tasks your program performs; and, when we start writing functions, it provides a natural way of dividing up your programs into sensible functions.



B Most programs have the structure shown here: First you get input (for example, from the user using the **`input`** function), then you process it, and then you display the results for the user to see.

From the Library of Bill Wiecking

4

Flow of Control

The programs we've written so far are *straight-line* programs that consist of a sequence of Python statements executed one after the other. The flow of execution is simply a straight sequence of statements, with no branching or looping back to previous statements.

In this chapter, we look at how to change the order in which statements are executed by using if-statements and loops. Both are essential in almost any nontrivial program.

Both if-statements and loops are controlled by logical expressions, and so the first part of this chapter will introduce the idea of Boolean logic.

Read the sample programs in this chapter carefully. Take the time to try them out and make your own modifications.

In This Chapter

Boolean Logic	44
If-Statements	49
Code Blocks and Indentation	51
Loops	54
Comparing For-Loops and While-Loops	59
Breaking Out of Loops and Blocks	64
Loops Within Loops	66

From the Library of Bill Wiecking

Boolean Logic

In Python, as in most programming languages, decisions are made using *Boolean logic*. Boolean logic is all about manipulating so-called truth values, which in Python are written **`True`** and **`False`**. Boolean logic is simpler than numeric arithmetic, and is a formalization of logical rules you already know.

We combine Boolean values using four main *logical operators* (or *logical connectives*): **`not`**, **`and`**, **`or`**, and **`==`**. All decisions that can be made by Python—or any computer language, for that matter—can be made using these logical operators.

Suppose that **`p`** and **`q`** are two Python variables each labeling Boolean values. Since each has two possible values (**`True`** or **`False`**), altogether there are four different sets of values for **`p`** and **`q`** (see the first two columns of **Table 4.1**). We can now define the logical operators by specifying exactly what value they return for the different truth values of **`p`** and **`q`**. These kinds of definitions are known as *truth tables*, and Python uses an internal version of them to evaluate Boolean expressions.

TABLE 4.1 Truth Table for Basic Logical Operators

`p`	`q`	`p == q`	`p != q`	`p and q`	`p or q`	`not p`
False	False	True	False	False	False	True
False	True	False	True	False	True	True
True	False	False	True	False	True	False
True	True	True	False	True	True	False

From the Library of Bill Wiecking

Logical equivalence

Let's start with `==`. The expression `p == q` is `True` only when `p` and `q` both have the same truth value—that is, when `p` and `q` are either both `True` or both `False`. The expression `p != q` tests if `p` and `q` are not the same, and returns `True` only when they have different values.

```
>>> False == False
True
>>> True == False
False
>>> True == True
True
>>> False != False
False
>>> True != False
True
>>> True != True
False
```

Logical "and"

The Boolean expression `p and q` is `True` only when both `p` is `True` and `q` is `True`. In every other case it is `False`. The fifth column of Table 4.1 summarizes each case.

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

Logical "or"

The Boolean expression `p or q` is `True` exactly when `p` is `True` or `q` is `True`, or when both are `True`. This is summarized in the sixth column of Table 4.1. The only slightly tricky case is when both `p` and `q` are `True`. In this case, the expression `p or q` is `True`.

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

Logical negation

Finally, the Boolean expression `not p` is `True` when `p` is `False`, and `False` when `p` is `True`. It essentially *flips* the value of the variable.

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


From the Library of Bill Wiecking

Evaluating larger Boolean expressions

Since Boolean expressions are used to control both if-statements and loops, it is important to understand how they are evaluated. Just as with arithmetic expressions, Boolean expressions use both brackets and operator precedence to specify the order in which their sub-parts are evaluated.

To evaluate a Boolean expression with brackets:

Suppose we want to evaluate the expression **`not (True and (False or True))`**. We can do it by following these steps:

- **`not (True and (False or True))`**

  Expressions in brackets are always evaluated first, and so we first evaluate **`False or True`**, which is **`True`**. This makes the original expression equivalent to this simpler one: **`not (True and True)`**.

- **`not (True and True)`**

  To evaluate this expression, we again evaluate the expression in brackets first: **`True and True`** evaluates to **`True`**, which gives us the equivalent expression: **`not True`**.

- **`not True`**

  Finally, to evaluate this expression, we simply look up the answer in the last column of Table 4.1: **`not True`** evaluates to **`False`**. Thus, the entire expression **`not (True and (False or True))`** evaluates to **`False`**. You can easily check that this is the correct answer in Python itself:

  ```
  >>> not (True and (False or
  → True))
  False
  ```


From the Library of Bill Wiecking

TABLE 4.2 Boolean Operator Priority (Highest to Lowest)

`p == q`
`p != q`
`not p`
`p and q`
`p or q`

To evaluate a Boolean expression without brackets:

Suppose we want to evaluate the expression **`not True and False or True`**. This is the same as the previous one, but this time there are no brackets.

- **`not True and False or True`**

  We first evaluate the operator with the highest precedence, as listed in **Table 4.2**. In this case, **`not`** has the highest precedence, and so **`not True`** is evaluated first (the fact that it happens to be at the start of the expression is a coincidence). This simplifies the expression to **`False and False or True`**.

- **`False and False or True`**

  We again evaluate the operator with the highest precedence. According to Table 4.2, **`and`** has higher precedence than **`or`**, and so **`False and True`** is evaluated first. The expression simplifies to **`False or True`**.

- **`False or True`**

  This final expression evaluates to **`True`**, which is found by looking up the answer in Table 4.1. Thus the original expression, **`False and not False or True`**, evaluates to **`True`**.

TIP Writing complicated Boolean expressions without brackets is usually a bad idea because they are hard to read and evaluate—not all programmers remember the order of precedence of Boolean operators!

TIP One exception is when you use the *same* logical operator many times in a row. Then it is usually easier to read without the brackets. For example:

```
>>> (True or (False or (True or
→ False)))
True
>>> True or False or True or False
True
```


From the Library of Bill Wiecking

Short-circuit evaluation

The definition of the logical operators given in Table 4.1 is the standard definition you would find in any logic textbook. However, like most modern programming languages, Python uses a simple trick called *short-circuit evaluation* to speed up the evaluation of some Boolean expressions.

Consider the Boolean expression **`False and X`**, where **`X`** is *any* Boolean expression. It turns out that no matter whether **`X`** is **`True`** or **`X`** is **`False`**, the entire expression is **`False`**. The reason is that the initial **`False`** makes the whole **`and`**-expression **`False`**. The value of **`False and X`** does not depend on **`X`**—it is *always* **`False`**. In such cases, Python does not evaluate **`X`** at all—it simply stops and returns the value **`False`**. This can speed up the evaluation of Boolean expressions.

Similarly, Boolean expressions of the form **`True or X`** are *always* **`True`**, no matter the value of **`X`**. The precise rules for how Python does short-circuiting are given in Table 4.3.

Most of the time you can ignore short-circuiting and just reap its performance benefits. However, it is useful to remember that Python does this, since every once in a while it could be the source of a subtle bug.

TIP It's possible to use the definitions of and and `or` from Table 4.3 to write short and tricky code that simulates if-statements (which we will see in the next section). However, such expressions are usually quite difficult to read, so if you ever run across such expressions in other people's Python code (*you* should never put anything so ugly in *your* programs!), you may need to refer to Table 4.3 to figure out exactly what they are doing.

TABLE 4.3 Definition of Boolean Operators in Python

Operation	Result
`p or q`	if **`p`** is **`False`**, then **`q`**, else **`p`**
`p and q`	if **`p`** is **`False`**, then **`p`**, else **`q`**

From the Library of Bill Wiecking

If-Statements

If-statements let you change the flow of control in a Python program. Essentially, they let you write programs that can decide, while the programming is running, whether or not to run one block of code or another. Almost all nontrivial programs use one or more if-statements, so they are important to understand.

If/else-statements

Suppose you are writing a password-checking program. The user enters their password, and if it is correct, you log them in to their account. If it is not correct, then you tell them they've entered the wrong password:

```
# password1.py
pwd = input('What is the password? ')
if pwd == 'apple': # note use of == #
→ instead of =
  print('Logging on ...')
else:
  print('Incorrect password.')
print('All done!')
```

It's pretty easy to read this program: If the string that **pwd** labels is **'apple'**, then a login message is printed. But if **pwd** is anything other than **'apple'**, the message *incorrect password* is printed.

An if-statement always begins with the keyword **if**. It is then (always) followed by a Boolean expression called the *if*-condition, or just *condition* for short. After the if-condition comes a colon (**:**). As we will see, Python uses the **:** token to mark the end of conditions in if-statements, loops, and functions.

From the Library of Bill Wiecking

Everything from the **`if`** to the **`:`** is referred to as the *if-statement* header. If the condition in the header evaluates to **`True`**, then the statement **`print('Logging on ...')`** is immediately executed, and **`print('Incorrect password.')`** is skipped and never executed.

If the condition in the header evaluates to **`False`**, then **`print('Logging on ...')`** is skipped, and only the statement **`print('Incorrect password.')`** is executed.

In all cases, the final **`print('All done!')`** statement is executed.

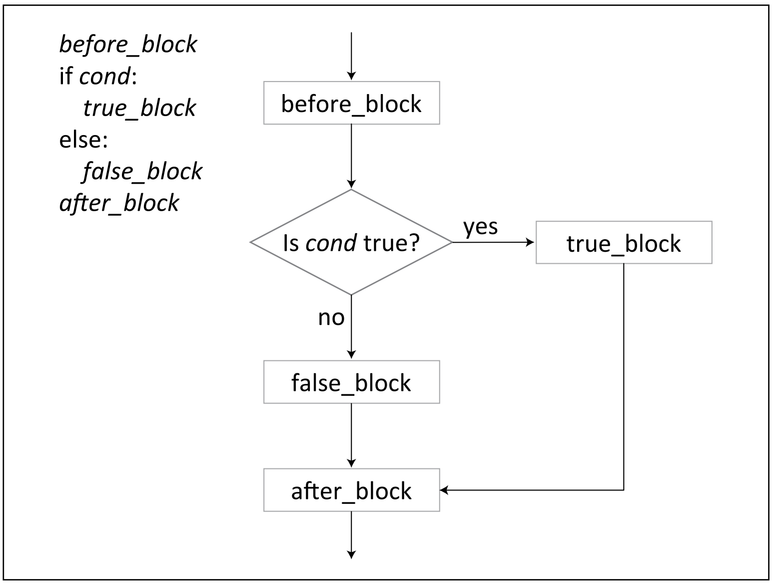
The general structure of an if/else-statement is shown in **A**.

TIP **We will often refer to the entire multiline if structure as a single if-statement.**

TIP **You must put at least one space after the `if` keyword.**

TIP **The `if` keyword, the condition, and the terminating `:` must appear all on one line without breaks.**

TIP **The else-block of an if-statement is optional. Depending on the problem you are solving, you may or may not need one.**



A This flow chart shows the general format and behavior of an if/else-statement. The code blocks can consist of any number of Python statements (even other if-statements!).

From the Library of Bill Wiecking

Code Blocks and Indentation

One of the most distinctive features of Python is its use of indentation to mark blocks of code. Consider the if-statement from our password-checking program:

```
if pwd == 'apple':
    print('Logging on ...')
else:
    print('Incorrect password.')
print('All done!')
```

The lines **`print('Logging on ...')`** and **`print('Incorrect password.')`** are two separate *code blocks*. These ones happen to be only a single line long, but Python lets you write code blocks consisting of any number of statements.

To indicate a block of code in Python, you must indent each line of the block by the same amount. The two blocks of code in our example if-statement are both indented four spaces, which is a typical amount of indentation for Python.

In most other programming languages, indentation is used only to help make the code look pretty. But in Python, it is required for indicating what block of code a statement belongs to. For instance, the final **`print('All done!')`** is *not* indented, and so is *not* part of the else-block.

Programmers familiar with other languages often bristle at the thought that indentation matters: Many programmers like the freedom to format their code how they please. However, Python's indentation rules follow a style that many programmers already use to make their code readable. Python simply takes this idea one step further and gives meaning to the indentation.

TIP IDLE is designed to automatically indent code for you. For instance, pressing Return after typing the : in an if-header automatically indents the cursor on the next line.

TIP The amount of indentation matters: A missing or extra space in a Python block could cause an error or unexpected behavior. Statements within the same block of code need to be indented at the same level.

From the Library of Bill Wiecking

If/elif-statements

An if/elif-statement is a generalized if-statement with more than one condition. It is used for making complex decisions. For example, suppose an airline has the following "child" ticket rates: Kids 2 years old or younger fly for free, kids older than 2 but younger than 13 pay a discounted child fare, and anyone 13 years or older pays a regular adult fare. This program determines how much a passenger should pay:

```
# airfare.py
age = int(input('How old are you? '))
if age <= 2:
  print(' free')
elif 2 < age < 13:
  print(' child fare)
else:
  print('adult fare')
```

After Python gets **age** from the user, it enters the if/elif-statement and checks each condition one after the other in the order they are given. So first it checks if **age** is less than 2, and if so, it indicates that the flying is free and jumps out of the elif-condition. If **age** is not less than 2, then it checks the next elif-condition to see if **age** is between 2 and 13. If so, it prints the appropriate message and jumps out of the if/elif-statement. If neither the if-condition nor the elif-condition is **True**, then it executes the code in the else-block.

TIP **elif is short for *else if*, and you can use as many elif-blocks as needed.**

TIP **Each of the code blocks in an if/elif-statement must be consistently indented the same amount.**

TIP **As with a regular if-statement, the else-block is optional. In an if/elif-statement *with* an else-block, *exactly one* of the if/elif-blocks will be executed. If there is no else-block, then it is possible that none of the conditions are True, in which case none of the if/elif-blocks are executed.**

From the Library of Bill Wiecking

Conditional expressions

Python has one more logical operator that some programmers like (and some don't!). It's essentially a shorthand notation for if-statements that can be used directly within expressions. Consider this code:

```
food = input("What's your favorite
→ food? ")
reply = 'yuck' if food == 'lamb'
→ else 'yum'
```

The expression on the right-hand side of `=` in the second line is called a *conditional expression*, and it evaluates to either `'yuck'` or `'yum'`. It's equivalent to the following:

```
food = input("What's your favorite
→ food? ")
if food == 'lamb':
  reply = 'yuck'
else:
  reply = 'yum'
```

Conditional expressions are usually shorter than the corresponding if/else-statements, although not always as flexible or easy to read. In general, you should use them when they make your code simpler.

From the Library of Bill Wiecking

Loops

Now we turn to loops, which are used to repeatedly execute blocks of code. Python has two main kinds of loops: *for-loops* and *while-loops*. For-loops are generally easier to use and less error prone than while-loops, although not quite as flexible.

Lingo Alert

Programmers often use the variable **`i`** because it is short for *index*, and is also commonly used in mathematics. When we start using loops within loops, it is common to use **`j`** and **`k`** as other loop variable names.

For-loops

The basic for-loop repeats a given block of code some specified number of times. For example, this snippet of code prints the numbers 0 to 9 on the screen:

```
# count10.py
for i in range(10):
  print(i)
```

The first line of a for-loop is called the *for-loop header*. A for-loop always begins with the keyword **`for`**. After that comes the *loop variable*, in this case **`i`**. Next is the keyword **`in`**, typically (but not always) followed by **`range(n)`** and a terminating **`:`** token. A for-loop repeats its *body*, the code block underneath it, exactly **`n`** times.

Each time the loop executes, the loop variable **`i`** is set to be the next value. By default, the initial value of **`i`** is 0, and it goes up to **`n - 1`** (not **`n`**!) by ones. Starting numbering at 0 instead of 1 might seem unusual, but it is common in programming.

If you want to change the starting value of the loop, add a starting value to **`range`**:

```
for i in range(5, 10):
  print(i)
```

This prints the numbers from 5 to 9.

From the Library of Bill Wiecking

TIP **If you want to print the numbers from 1 to 10 (instead of 0 to 9), there are two common ways of doing so. One is to change the start and end of the range:**

```
for i in range(1, 11):
  print(i)
```

Or, you can add 1 to `i` inside the loop body:

```
for i in range(10):
  print(i + 1)
```

TIP **If you would like to print numbers in *reverse* order, there are again two standard ways of doing so. The first is to set the `range` parameters like this:**

```
for i in range(10, 0, -1):
  print(i)
```

Notice that the first value of `range` is 10, the second value is 0, and the third value, called the *step*, is −1. Alternatively, you can use a simpler range and modify `i` in the loop body:

```
for i in range(10):
  print(10 - i)
```

TIP **For-loops are actually more general than described in this section: They can be used with any kind of *iterator*, which is a special kind of programming object that returns values. For instance, we will see later that for-loops are the easiest way to read the lines of a text file.**

From the Library of Bill Wiecking

While-loops

The second kind of Python loop is a *while-loop*. Consider this program:

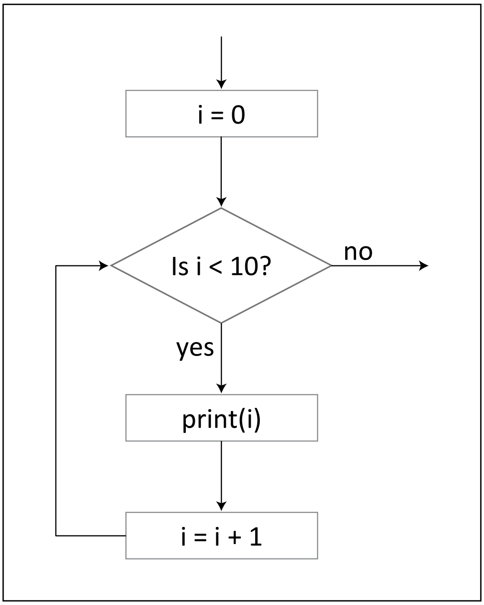
```
# while10.py
i = 0
while i < 10:
  print(i)
  i = i + 1 # add 1 to i
```

This prints out the numbers from 0 to 9 on the screen. It is noticeably more complicated than a for-loop, but it is also more flexible.

The while-loop itself begins on the line beginning with the keyword **`while`**; this line is called the *while-loop header*, and the indented code underneath it is called the *while-loop body*. The header always starts with **`while`** and is followed by the *while-loop condition*. The condition is a Boolean expression that returns **`True`** or **`False`**.

The flow of control through a while-loop goes like this: First, Python checks if the loop condition is **`True`** or **`False`**. If it's **`True`**, it executes the body; if it's **`False`**, it skips over the body (that is, it jumps out of the loop) and runs whatever statements appear afterward. When the condition is **`True`**, the body is executed, and then Python checks the condition again. As long as the loop condition is **`True`**, Python keeps executing the loop. B shows a flow chart for this program.

The very first line of the sample program is **`i = 0`**, and in the context of a loop it is known as an *initialization statement*, or an *initializer*. Unlike with for-loops, which automatically initialize their loop variable, it is the programmer's responsibility to give initial values to any variables used by a while-loop.



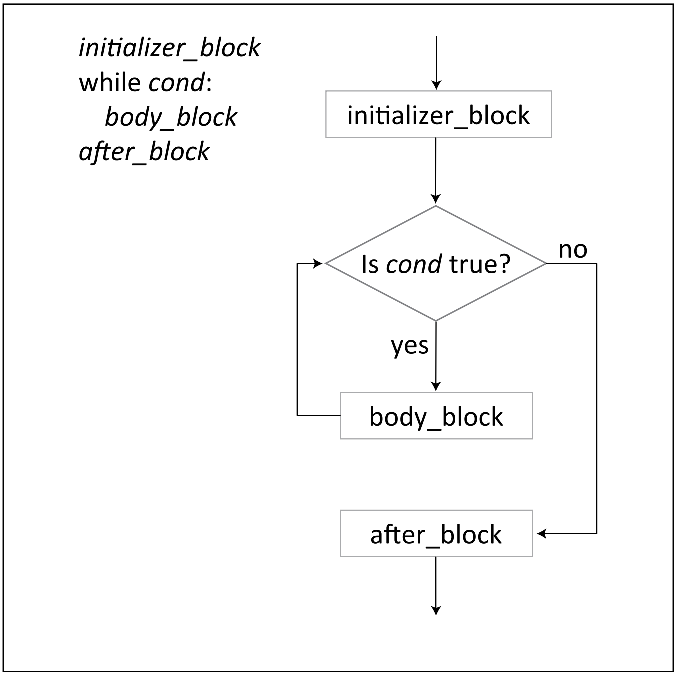
B This is a flow chart for code that counts from 0 to 9. Notice that when the loop condition is **`False`** (that is, the *no* branch is taken in the decision box), the arrow does not go into a box. That's because in our sample code there is nothing after the while-loop.

From the Library of Bill Wiecking

The last line of the loop body is `i = i + 1`. As it says in the source code comment, this line causes `i` to be incremented by 1. Thus, `i` increases as the loop executes, which guarantees that the loop will eventually stop. In the context of a while-loop, this line is called an *increment,* or *incrementer*, since its job is to increment the loop variable.

The general form of a while-loop is shown in the flow chart of C.

Even though almost all while-loops need an initializer and an incrementer, Python does not require that you include them. It is entirely up to you, the programmer, to remember these lines. Even experienced programmers find that while-loop initializers and incrementers are a common source of errors.



C A flow chart for the general form of a while-loop. Note that the incrementer is not shown explicitly: It is embedded somewhere in **`body_block`**, often (but not always) at the end of that block.

From the Library of Bill Wiecking

TIP **While-loops are *extremely* flexible. You can put any code whatsoever before a while-loop to do whatever kind of initialization is necessary. The loop condition can be *any* Boolean expression, and the incrementer can be put *anywhere* within the while-loop body, and it can do whatever you like.**

TIP **A loop that never ends is called an *infinite loop*. For instance, this runs forever:**

```
while True:
  print('spam')
```

TIP **Some programmers like to use infinite loops as a quick way to write a loop. However, this is generally considered to be poor style because such loops often become complex and hard to understand.**

TIP **Many Python programmers try to use for-loops whenever possible and use while-loops only when absolutely necessary.**

TIP **While-loops can be written with an else-block. However, this unusual feature is rarely used in practice, so we haven't discussed it. If you are curious, you can read about it in the online Python documentation—for example, http://docs.python.org/3/reference/compound_stmts.html.**

From the Library of Bill Wiecking

Comparing For-Loops and While-Loops

Let's take a look at a few examples of how for-loops and while-loops can be used to solve the same problems. Plus we'll see a simple program that can't be written using a for-loop.

Calculating factorials

Factorials are numbers of the form *1 × 2 × 3 × ... × n*, and they tell you how many ways *n* objects can be arranged in a line. For example, the letters ABCD can be arranged in 1 × 2 × 3 × 4 = 24 different ways. Here's one way to calculate factorials using a for-loop:

```
# forfact.py
n = int(input('Enter an integer
→>= 0: '))
fact = 1
for i in range(2, n + 1):
  fact = fact * i
print(str(n) + ' factorial is ' +
→str(fact))
```

Here's another way to do it using a while-loop:

```
# whilefact.py
n = int(input('Enter an integer
→>= 0: '))
fact = 1
i = 2
while i <= n:
  fact = fact * i
  i = i + 1
print(str(n) + ' factorial is ' +
→str(fact))
```

continues on next page

From the Library of Bill Wiecking

Both of these programs behave the same from the user's perspective, but the internals are quite different. As is usually the case, the while-loop version is a little more complicated than the for-loop version.

TIP In mathematics, the notation *n!* is used to indicate factorials. For example, 4! = 1 × 2 × 3 × 4 = 24. By definition, 0! = 1. Interestingly, there is *no* simple formula for calculating factorials.

TIP Python has no maximum integer, so you can use these programs to calculate very large factorials. For example, a deck of cards can be arranged in exactly 52! ways:

```
Enter an integer >= 0: 52
52 factorial is 80658175170943878571
→ 66063685640376697528950544088327782
→ 24000000000000
```


From the Library of Bill Wiecking

Summing numbers from the user

The following programs ask the user to enter some numbers, and then prints their sum. Here is a version using a for-loop:

```
# forsum.py
n = int(input('How many numbers to
→sum? '))
total = 0
for i in range(n):
  s = input('Enter number ' +
  →str(i + 1) + ': ')
  total = total + int(s)
print('The sum is ' + str(total))
```

Here's a program that does that same thing using a while-loop:

```
# whilesum.py
n = int(input('How many numbers to
→sum? '))
total = 0
i = 1
while i <= n:
  s = input('Enter number ' +
  →str(i) + ': ')
  total = total + int(s)
  i = i + 1
print('The sum is ' + str(total))
```

Again, the while-loop version is a little more complex than the for-loop version.

TIP These programs assume that the user is entering integers. Floating point numbers will be truncated when `int(s)` is called. Of course, you can easily change this to `float(s)` if you want to allow floating point numbers.

From the Library of Bill Wiecking

Summing an unknown number of numbers

Now here's something that can't be done with the for-loops we've seen so far. Suppose we want to let users enter a list of numbers to be summed without asking them ahead of time how many numbers they have. Instead, they just type **'done'** when they have no more numbers to add. Here's how to do it using a while-loop:

```
# donesum.py
total = 0
s = input('Enter a number (or
→ "done"): ')
while s != 'done':
  num = int(s)
  total = total + num
  s = input('Enter a number (or
→ "done"): ')
print('The sum is ' + str(total))
```

The idea here is to keep asking users to enter a number, quitting only when they enter **'done'**. The program doesn't know ahead of time how many times the loop body will be executed.

From the Library of Bill Wiecking

Notice a few more details:

- We must call **`input`** in two different places: before the loop and inside the loop body. This is necessary because the loop condition decides whether or not the input is a number or **`'done'`**.
- The ordering of the statements in the loop body is very important. If the loop condition is **`True`**, then we know **`s`** is not **`'done'`**, and so we assume it is an integer. Thus we can convert it to an integer, add it to the running total, and then ask the user for more input.
- We convert the input string **`s`** to an integer only *after* we know **`s`** is not the string **`'done'`**. If we had written

  ```
  s = int(input('Enter a number
  → (or "done"): '))
  ```

  as we had previously, the program would crash when the user typed **`'done'`**.
- There is no need for the **`i`** counter variable anymore. In the previous summing programs, **`i`** tracked how many numbers had been entered so far. As a general rule, a program with fewer variables is easier to read, debug, and extend.

From the Library of Bill Wiecking

Breaking Out of Loops and Blocks

The **break** statement is a handy way for exiting a loop from anywhere within the loop's body. For example, here is an alternative way to sum an unknown number of numbers:

```
# donesum_break.py
total = 0
while True:
  s = input('Enter a number (or
  → "done"): ')
  if s == 'done':
    break # jump out of the loop
  num = int(s)
  total = total + num
print('The sum is ' + str(total))
```

The while-loop condition is simply **True**, which means it will loop forever unless **break** is executed. The only way for **break** to be executed is if **s** equals **'done'**.

An advantage of this program over **donesum.py** is that the **input** statement is not repeated. But a disadvantage is that the reason for why the loop ends is buried in the loop body. It's not so hard to see it in this small example, but in larger programs **break** statements can be tricky to see. Furthermore, you can have as many **break**s as you want, which adds to the complexity of understanding the loop.

From the Library of Bill Wiecking

Generally, it is wise to avoid the `break` statement, and to use it only when it makes your code simpler or clearer.

A relative of `break` is the `continue` statement: When `continue` is called inside a loop body, it immediately jumps up to the loop condition—thus continuing with the next iteration of the loop. It is a little less common than `break`, and generally it should be avoided altogether.

TIP Both `break` and `continue` also work with for-loops.

From the Library of Bill Wiecking

Loops Within Loops

Loops within loops, also known as *nested loops,* occur frequently in programming. For instance, here's a program that prints the times tables up to 10:

```
# timestable.py
for row in range(1, 10):
  for col in range(1, 10):
    prod = row * col
    if prod < 10:
      print(' ', end = '')
    print(row * col, ' ', end = '')
  print()
```

Look carefully at the indentation of the code in this program: It's how you tell what statements belong to what blocks. The final `print()` statement lines up with the second `for`, meaning it is part of the outer for-loop (but not the inner).

Note that the statement `if prod < 10` is used to make the output look neatly formatted. Without it, the numbers won't line up nicely.

TIP When using nested loops, be careful with loop index variables: Do not accidentally reuse the same variable for a different loop. Most of the time, every individual loop needs its own control variables.

TIP You can nest as many loops within loops as you need, although the complexity increases greatly as you do so.

TIP As mentioned previously, if you use `break` or `continue` with nested loops, `break` only breaks out of the innermost loop, and `continue` only "continues" the innermost loop.

From the Library of Bill Wiecking

5

Functions

A *function* is a reusable chunk of code. It is a block of code with a name that takes input, provides output, and can be stored in files for later use. Pretty much any useful piece of Python code is stored in a function.

Python has excellent support for functions. For instance, it provides many ways to pass data into a function. It also lets you include documentation strings within the function itself so that you—or other programmers—can read how the function works.

You need to learn a number of details in order to completely understand functions. With practice, they will soon become second nature, so be sure to try out the examples from this chapter.

In This Chapter

Calling Functions	68
Defining Functions	70
Variable Scope	73
Using a main Function	75
Function Parameters	76
Modules	80

From the Library of Bill Wiecking

Calling Functions

We've been calling functions quite a bit so far, so let's take a moment to look a little more carefully at a *function call*.

Consider the built-in function **`pow(x, y)`**, which calculates **`x ** y`**—that is to say, **`x`** raised to the power **`y`**:

```
>>> pow(2, 5)
32
```

Here, we say that **`pow`** is the *function name*, and that the values 2 and 5 are *arguments* that are *passed into* **`pow`**. The value 32 is the *return value*, so we say that **`pow(2, 5)`** *returns* 32. Figure A gives a high-level overview of a function call.

When you call a function within an expression, Python essentially replaces the function call with its return value. For example, the expression **`pow(2, 5) + 8`** is the same as **`32 + 8`**, which evaluates to **`40`**.

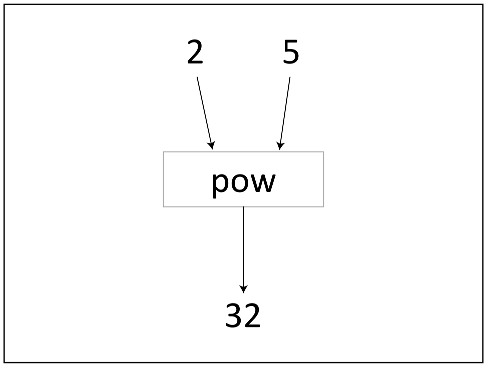
When a function takes no input (that is to say, it has zero arguments), you must still include the round brackets **`()`** after the function name:

```
>>> dir()
['__builtins__', '__doc__',
→ '__name__', '__package__']
```

The **`()`** tells Python to execute the function. If you leave off the **`()`**, then you get this:

```
>>> dir
<built-in function dir>
```

Without the **`()`**, Python does not execute the **`dir`** function and instead tells you that **`dir`** labels a function.



A It's often useful to think of functions as being black boxes that accept an input (2 and 5 in this case) and return an output (32). From the point of view of a programmer calling the **`pow`** function, there is no (easy) way to see inside of **`pow`**. All we know is what the documentation tells us, and what the function does when we call it.

Calculating Powers

Calling **`pow(x, y)`** is the same as calling **`x ** y`**. You may notice that **`pow(0, 0)`** (and also **`0 ** 0`**) is 1 in Python, and this has been a matter of some debate. According to some mathematicians, **`pow(0, 0)`** ought to be *indeterminate*, or undefined. But others say that it is more sensible to define **`pow(0, 0)`** as 1. Python has obviously sided with the latter group.

From the Library of Bill Wiecking

Functions that don't return a value

Some functions, such as **print**, are not meant to return values. Consider:

```
>>> print('hello')
hello
>>> x = print('hello')
hello
>>> x
>>> print(x)
None
```

Here the variable **x** has been assigned a special value called **None**. **None** indicates "no return value": It is not a string or a number, so you can't do any useful calculations with it.

Reassigning function names

You need to take care not to accidentally make a built-in function name refer to some other function or value. Unfortunately, Python doesn't stop you from writing code like this:

```
>>> dir = 3
>>> dir
3
>>> dir()
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in
  → <module>
  dir()
TypeError: 'int' object is not
→ callable
```

Here we've made **dir** label the number 3, so the function that **dir** used to refer to is no longer accessible! You will need to restart Python to get it back.

From the Library of Bill Wiecking

Defining Functions

Now we turn to creating our own functions. As an example, let's write a function to calculate the area of a circle. Recall that a circle's area is pi times its radius squared. Here is a Python function that does this calculation:

```
# area.py
import math
def area(radius):
  """ Returns the area of a circle
  with the given radius.
  For example:
  >>> area(5.5)
  95.033177771091246
  """
  return math.pi * radius ** 2
```

Save this function inside a Python file (**area.py** would be a good name); then load it into the IDLE editor and run it by pressing F5. If everything is typed correctly, a prompt should appear and nothing else; a function is not executed until you call it. To call it, just type the name of the function, with the radius in brackets:

```
>>> area(1)
3.1415926535897931
>>> area(5.5)
95.033177771091246
>>> 2 * (area(3) + area(4))
157.07963267948966
```

The **area** function can be called just like any other function, the difference being that *you* have written the function and so *you* have control over what it does and how it works.

Naming Functions

As with variable names, a function name must be one or more characters long and consist of letters, numbers, or the underscore (_) character. The first character of the name *cannot* be a number.

In general, give functions simple English names that hint at their *purpose*. Don't make them too long or too short. Other programmers reading your code or using your functions (including you, a few months down the road!) will thank you for choosing helpful names.

From the Library of Bill Wiecking

A Formatting Convention

Python doc strings tend to follow a standard formatting convention. Triple quotes are used to mark the beginning and end of the doc string. The first line is a succinct one-line description of the function useful to a programmer. After the first line come more details and examples.

Extra Benefits of Doc Strings

Just as with built-in functions, you can easily access the doc strings for your own functions, like this:

```
>>> print(area.__doc__)
Returns the area of a circle
→ with the given radius.
For example:
  >>> area(5.5)
  95.033177771091246
```

As you will see when you call **area** in the IDLE editor, IDLE automatically reads the function doc string and pops it up as an automatic tool tip.

Python also has a useful tool called *doctest* that can be used to automatically run example Python code found in doc strings. This is a good way to test your code, and to help ensure that the documentation accurately describes the function. We won't go into the details of doctest in this book, but it is easy to use and quite helpful, and you can read more about it here: http://docs.python.org/3/library/doctest.html.

Parts of a function

Let's look at each part of the **area** function. The first line, the one that begins with **def**, is called the *function header*; all the code indented beneath the header is called the *function body*.

Function headers always begin with the keyword **def** (short for *definition*), followed by a space, and then the *name* of the function (in this case, **area**). Function names follow essentially the same rules as names for variables.

After the function name comes the function *parameter list*. This is a list of variable names that label the input to the function. The **area** function has a single input, **radius**, although a function can have any number of inputs. If a function has 0 inputs, then only the round brackets are written, **()**.

Finally, like loops and if-statements, a function header ends with a colon (**:**).

After the function header comes an optional *documentation string*, or *doc string* for short. A doc string briefly explains what the function will do, and it may include examples or other helpful information. While doc strings are optional, they are almost always a good idea: When you start writing a lot of functions, it is easy to forget exactly what they do and how they work, and a well-written doc string can be a good reminder.

After the doc string comes the main *body* of the function. This is simply an indented block of code that does whatever you need it to do. The code in this block is allowed to use the variables from the function header.

Finally, the function should *return* a value using the **return** keyword. When a **return** statement is executed, Python jumps out of the function and back to the point in the program where it was called.

From the Library of Bill Wiecking

In the case of the **area** function, the **return** statement is the last line of the function, and it simply returns the value of the area of a circle using the standard formula. Note that it uses the **radius** parameter in its calculation; the value for **radius** is set when the **area** function is called.

A **return** is usually the last line of a function to be executed (the only time it isn't is when the function ends unexpectedly due to an *exception* being thrown, which we will talk about in a later chapter). You can put a **return** anywhere inside a function body, although it is typically the last physical line of the function.

A function is not required to have an explicit **return** statement. For example:

```
# hello.py

def say_hello_to(name):

  """ Prints a hello message.
  """

  cap_name = name.capitalize()

  print('Hello ' + cap_name + ', how
  → are you?')
```

If you don't put a **return** anywhere in a function, Python treats the function as if it ended with this line:

```
return None
```

The special value **None** is used to indicate that the function is not meant to be returning a useful value. This is fairly common: Functions are often used to perform tasks where the return values don't matter, such as printing output to the screen.

Lingo Alert

When a function makes a change in any way other than returning a value, we call that change a *side effect*. Printing to the screen, writing to a file, and downloading a webpage are all examples of side effects.

A style of programming known as *functional programming* is characterized by its near-complete banishment of side effects. In functional programming, the only changes you can make are via return values. Python has a lot of support for functional programming, including the ability to define functions within functions and to pass functions as values to other functions. When used correctly, functional programming can be a very elegant and powerful way of writing programs.

Although we won't be covering functional programming in detail in this book, it is nonetheless wise to avoid function side effects whenever possible.

From the Library of Bill Wiecking

Variable Scope

An important detail that functions bring up is the issue of *scope*. The scope of a variable (or function) is where in a program it is accessible, or visible. Consider these two functions:

```
# local.py
import math
def dist(x, y, a, b):
  s = (x - a) ** 2 + (y - b) ** 2
  return math.sqrt(s)
def rect_area(x, y, a, b):
  width = abs(x - a)
  height = abs(y - b)
  return width * height
```

Any variable assigned for the first time within a function is called a *local variable*. The function **dist** has one local variable, **s**, while **rect_area** has two local variables, **width** and **height**.

The parameters of a function are also considered local. Thus **dist** has a total of five local variables—**x**, **y**, **a**, **b**, and **s**; **rect_area** has a total of six. Notice that variables **x**, **y**, **a**, and **b** appear in both functions, but they generally label different values.

Importantly, local variables are usable only within the function they are local to. No code outside of the function can access its local variables.

When a function ends, its local variables are automatically deleted.

From the Library of Bill Wiecking

Global variables

Variables declared outside of any function are called *global variables*, and they are readable anywhere by any function or code within the program. However, there is a wrinkle in reassigning global variables within functions you need to be aware of. Consider the following:

```
# global_error.py
name = 'Jack'
def say_hello():
  print('Hello ' + name + '!')
def change_name(new_name):
  name = new_name
```

The variable **name** is a global variable because it is not declared inside any function. The **say_hello()** function reads the value of **name** and prints it to the screen as you would expect:

```
>>> say_hello()
Hello Jack!
```

However, things don't work as expected when you call **change_name**:

```
>>> change_name('Piper')
>>> say_hello()
Hello Jack!
```

Nothing changed—**name** still labels the value **'Jack'**. The problem is that Python treated **name** inside the **change_name** function as being a local variable, and so ignored the global **name** variable.

To access the global variable, you must use the **global** statement:

```
# global_correct.py
name = 'Jack'
def say_hello():
  print('Hello ' + name + '!')
def change_name(new_name):
  global name
  name = new_name
```

This makes all the difference. Both functions now work as expected:

```
>>> say_hello()
Hello Jack!
>>> change_name('Piper')
>>> say_hello()
Hello Piper!
```


From the Library of Bill Wiecking

Main in Other Languages

The idea of using a **`main`** function is quite common, and some other programming languages, notably C, C++, and Java, actually define the use of **`main`** as part of the language. In Python, however, **`main`** is entirely optional, and used only as a helpful convention.

Using a main Function

It is usually a good idea to use at least one function in any Python program you write: **`main()`**. A **`main()`** function is, by convention, assumed to be the starting point of your program. For instance, you could write the password program from the previous chapter using a **`main`** function:

```
# password2.py
def main():
  pwd = input('What is the
  → password? ')
  if pwd == 'apple':
    print('Logging on ...')
  else:
    print('Incorrect password.')
  print('All done!')
```

Notice that all the code is indented underneath the **`main`** function header.

When you run **`password2.py`** in IDLE, nothing happens—only the prompt appears. You must type **`main()`** to execute the code within in it.

The advantage of using a **`main`** function is that it is now easier to rerun programs and pass in input values.

From the Library of Bill Wiecking

Function Parameters

Parameters pass data into a function, and Python has several kinds of parameters.

Pass by reference

Python passes parameters to a function using a technique known as *pass by reference*. This means that when you pass parameters, the function refers to the *original* passed values using new names. For example, consider this simple program:

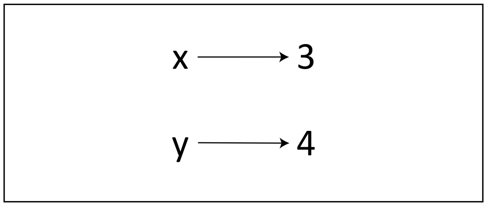
```
# reference.py
def add(a, b):
  return a + b
```

Run IDLE's interactive command line and type this:

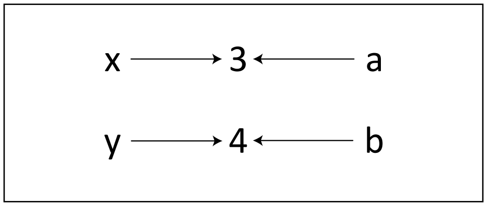
```
>>> x, y = 3, 4
>>> add(x, y)
7
```

After you set **x** and **y** in the first line, Python's memory looks like **B**. Now when **add(x, y)** is called, Python creates two new variables, **a** and **b**, that refer to the values of **x** and **y** **C**. The values are assigned in the order they occur—thus **a** refers to **x** because **x** is the first argument. Notice that the values are *not* copied: They are simply given new names that the function uses to refer to them.

After **a** and **b** are summed and the function returns, references **a** and **b** are automatically deleted. The original **x** and **y** are untouched throughout the entire function call.



B The state of memory after setting **x** to 3 and **y** to 4.

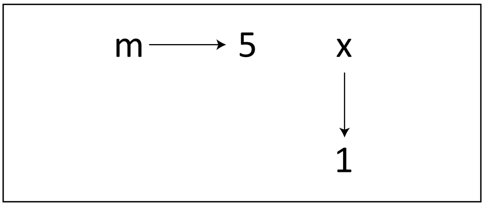


C The state of memory just after **add(x, y)** is called, and **a** and **b** have been set to refer to the values of **x** and **y**, respectively.

Pass by Value

Some programming languages, such as C++, can pass parameters using *pass by value*. When a parameter is passed by value, a *copy* of it is made and passed to the function. If the value being passed is large, the copying can take up a lot of time and memory. Python does not support pass by value.

From the Library of Bill Wiecking



D After **x** is assigned 1 in the function call **set1(m)**, **m** is unchanged and still refers to its original value of 5. However, the local variable **x** has indeed been set to 1.

An important example

Passing by reference is simple and efficient, but there are some things it cannot do. For example, consider this plausibly named function:

```
# reference.py
def set1(x):
  x = 1
```

The purpose of **set1** is to set the value of the passed-in variable to 1. But when you try it, it does not work as expected:

```
>>> m = 5
>>> set1(m)
>>> m
5
```

Surprisingly, the value of **m** has *not* changed. The reason why is a consequence of pass by reference. It's helpful to break the example down into steps:

1. Assign 5 to **m**.
2. Call **set1(m)**: Assign the value of **x** to the value of **m** (so now both **m** and **x** point to 5).
3. Assign 1 to **m**. Now the situation is as shown in D.
4. When the **set1** function ends, **x** is deleted.

The variable **m** is simply not accessible within **set1**, so there is no way to change what it points to.

From the Library of Bill Wiecking

Default values

It's often useful to include a *default value* with a parameter. For example, here we have given the `greeting` parameter a default value of `'Hello'`:

```
# greetings.py
def greet(name, greeting = 'Hello'):
  print(greeting, name + '!')
```

You can now call `greet` in two distinct ways:

```
>>> greet('Bob')
Hello Bob!
>>> greet('Bob', 'Good morning')
Good morning Bob!
```

TIP **Default parameters are quite handy and are used all the time in Python.**

TIP **A function can use as many default parameters as it needs, although no parameter without a default value can appear before a parameter with one.**

TIP **An important detail about default parameters is that they are evaluated only once, the first time they are called. In complicated programs, this can sometimes be the source of subtle bugs, so it is useful to keep this fact in mind.**

From the Library of Bill Wiecking

Keyword parameters

Another useful way to specify parameters in Python is by using *keywords*. For example:

```
# shopping.py
def shop(where = 'store',
         what = 'pasta',
         howmuch = '10 pounds'):
  print('I want you to go to the',
  → where)
  print('and buy', howmuch, 'of',
  → what + '.')
```

To call a function that uses keyword parameters, pass data in the form *param* = *value*. For example:

```
>>> shop()
I want you to go to the store
and buy 10 pounds of pasta.
>>> shop(what = 'towels')
I want you to go to the store
and buy 10 pounds of towels.
>>> shop(howmuch = 'a ton', what =
→ 'towels')
I want you to go to the store
and buy a ton of towels.
>>> shop(howmuch = 'a ton', what =
→ 'towels', where = 'bakery')
I want you to go to the bakery
and buy a ton of towels.
```

Keyword parameters have two big benefits. First, they make the parameter values clear, and thus help to make your programs easier to read. Second, the order in which you call keyword parameters does not matter. Both of these are quite helpful in functions with many parameters; for such functions it can be difficult to remember the exact order in which to put the parameters, and what they mean.

From the Library of Bill Wiecking

Modules

A *module* is collection of related functions and variables.

To create a Python module:

- Create a **.py** file containing your functions and assignments. For example, here is a simple module for printing shapes to the screen:

```
# shapes.py
"""A collection of functions
  for printing basic shapes.
"""
CHAR = '*'
def rectangle(height, width):
  """ Prints a rectangle. """
  for row in range(height):
    for col in range(width):
      print(CHAR, end = '')
    print()
def square(side):
  """ Prints a square. """
  rectangle(side, side)
def triangle(height):
  """ Prints a right triangle. """
  for row in range(height):
    for col in range(1, row + 2):
      print(CHAR, end = '')
    print()
```

  The only difference between this and a regular Python program is the intended use: A module is a toolbox of helpful functions that you can use when writing

From the Library of Bill Wiecking

other programs. Thus a module usually does not have a **`main()`** function.

- To use a module, you simply import it. For example:

```
>>> import shapes
>>> dir(shapes)
['CHAR', '__builtins__',
→'__doc__', '__file__',
→'__name__', '__package__',
→'rectangle', 'square',
→'triangle']
>>> print(shapes.__doc__)
A collection of functions
for printing basic shapes.
>>> shapes.CHAR
'*'
>>> shapes.square(5)
*****
*****
*****
*****
*****
>>> shapes.triangle(3)
*
**
***
```

- You can also import everything at once:

```
>>> from shapes import *
>>> rectangle(3, 8)
********
********
********
```


From the Library of Bill Wiecking

Namespaces

A very useful fact about modules is that they form *namespaces*. A namespace is essentially a set of unique variable and function names. The names within a module are visible outside the module only when you use an **`import`** statement.

To see why this is important, suppose Jack and Sophie are working together on a large programming project. Jack is on the West Coast, and Sophie is on the East. They agree that Jack will put all his code in the module **`jack.py`**, and Sophie will put all her code into **`sophie.py`**. They work independently, and it turns out that they both wrote a function called **`save_file(fname)`**. However, only the headers of their functions are the same; they do radically different things. Having two functions with the same name is fine because the functions are in different modules, so the names are in different namespaces. The full name of Jack's function is **`jack.save_file(fname)`**, and the full name of Sophie's is **`sophie.save_file(fname)`**.

Thus modules support independent development, by preventing *name clashes*. Even if you are not working with other programmers, name clashes can be an annoying problem in your own programs, especially as they get larger and more complex.

Of course, you can still run into name clashes as follows:

```
>>> from jack import *
>>> from sophie import *
```

These kinds of import statements essentially dump everything from each module into the current namespace, overwriting anything with the same name as they go. Thus, it is generally wise to avoid **`from ... import *`** statements in larger programs.

The Zen of Python

To see an interesting Python Easter egg, try importing the module **`this`** at the interactive command line:

```
>>> import this
```


From the Library of Bill Wiecking

6

Strings

After numbers, strings are the most important data type in Python. Strings are ubiquitous: You print them to the screen, you read them from the user, and as we will see in Chapter 8, files are often treated as big strings. The World Wide Web can be thought of as a collection of webpages, most of which consist of text.

Strings are an example of an *aggregate data structure*, and they provide our first look at indexing and slicing—techniques that are used to extract substrings from strings.

The chapter also contains a brief introduction to Python's regular expression library, which is a supercharged mini-language designed for processing strings.

In This Chapter

String Indexing	84
Characters	87
Slicing Strings	89
Standard String Functions	92
Regular Expressions	98

From the Library of Bill Wiecking

String Indexing

We introduced strings in Chapter 2, so you may want to go back to it if you need a refresher on string basics.

When working with strings, we often want to access their individual characters. For example, suppose you know that **s** is a string, and you want to print its individual characters. *String indexing* is how you do it:

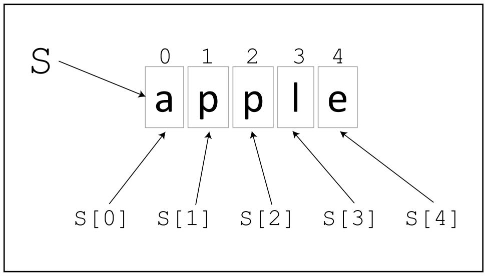
```
>>> s = 'apple'
>>> s[0]
'a'
>>> s[1]
'p'
>>> s[2]
'p'
>>> s[3]
'l'
>>> s[4]
'e'
```

Python uses *square brackets* to index strings: The number in the brackets indicates which character to get **A**. Python's index values always start at 0 and always end at one less than the length of the string.

So if **s** labels a string of length **n**, **s[0]** is the first character, **s[1]** is the second character, **s[2]** is the third character, and so on up to **s[n-1]**, which is the last character.

If you try to index past the right end of the string, you will get an "out of range" error:

```
>>> s[5]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in
→ <module>
  s[5]
IndexError: string index out of range
```

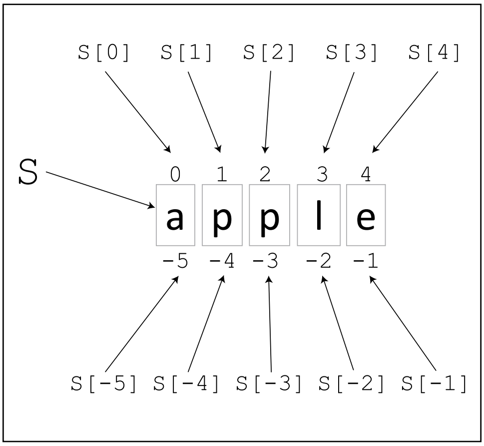


A This diagram shows the index values for the string '**apple**'. Square-bracket indexing notation is used to access individual characters within the string.

Why Start at 0?

Beginning programmers often find it odd that Python indexes begin at 0 instead of 1. It does take some getting used to and can be the source of *off-by-one errors* that plague many programmers. It can be helpful to think of an index value as *measuring the distance* from the first character of the string, just like a ruler (which also starts at 0). This makes some calculations with indexes a little simpler, and it also fits nicely with the **%** (mod) function, which is often used with index calculations and naturally returns 0s.

From the Library of Bill Wiecking



B Python strings have both positive and negative indexes. In practice, programmers usually use whatever index is most convenient.

Negative indexing

Suppose instead of the first character of **s**, you want to access the last character of **s**. The ungainly expression **s[len(s) - 1]** works, but it's rather complicated.

Fortunately, Python has a more convenient way of accessing characters near the right end of a string: *negative indexing*. The idea is that the characters of a string are indexed with negative numbers going from right to left:

```
>>> s = 'apple'
>>> s[-1]
'e'
>>> s[-2]
'l'
>>> s[-3]
'p'
>>> s[-4]
'p'
>>> s[-5]
'a'
```

Thus the last character of a string is simply **s[-1]**. Figure **B** shows how negative index values work.

From the Library of Bill Wiecking

Accessing characters with a for-loop

If you need to access every character of a string in sequence, a for-loop can be helpful. For example, this program calculates the sum of the character codes for a given string:

```
# codesum.py
def codesum1(s):
  """ Returns the sums of the
  character codes of s.
  """
  total = 0
  for c in s:
    total = total + ord(c)
  return total
```

Here is a sample call:

```
>>> codesum1('Hi there!')
778
```

When you use a for-loop like this, at the beginning of each iteration the loop variable **c** is set to be the next character in **s**. The indexing into **s** is handled automatically by the for-loop.

Compare **codesum1** with this alternative implementation, which uses regular string indexing:

```
def codesum2(s):
  """ Returns the sums of the
  character codes of s.
  """
  total = 0
  for i in range(len(s)):
    total = total + ord(s[i])
  return total
```

This gives the same results as **codesum1**, but the implementation is a little more complex and harder to read.

From the Library of Bill Wiecking

The Rise of Unicode

In the 1960s, ’70s, and ’80s, the most popular character encoding scheme was ASCII (American Standard Code for Information Interchange). ASCII is far simpler than Unicode, but its fatal flaw is that it can represent only 256 different characters—enough for English and French and a few other similar languages, but nowhere near enough to represent the huge variety of characters and symbols found in other languages. For instance, Chinese alone has *thousands* of ideograms that could appear in text documents.

Essentially, Unicode provides a far larger set of character codes. Conveniently, Unicode mimics the ASCII code for the first 256 characters, so if you are only dealing with English characters (as we are in this book), you’ll rarely need to worry about the details of Unicode. For more information, see the Unicode home page (www.unicode.org).

Characters

Strings consist of characters, and characters themselves turn out to be a surprisingly complex issue. As mentioned in Chapter 2, all characters have a corresponding character code that you can find using the **ord** function:

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('c')
99
```

Given a character code number, you can retrieve its corresponding character using the **chr** function:

```
>>> chr(97)
'a'
>>> chr(98)
'b'
>>> chr(99)
'c'
```

Character codes are assigned using *Unicode*, which is a large and complex standard for encoding all the symbols and characters that occur in all the world’s languages.

From the Library of Bill Wiecking

Escape characters

Not all characters have a standard visible symbol. For example, you can't see a newline character, a return character, or a tab (although you can certainly see their *effects*). They are *whitespace characters*, characters that appear as blanks on the printed page.

To handle whitespace and other unprintable characters, Python uses a special notation called *escape sequences*, or *escape characters*. **Table 6.1** shows the most commonly used escape characters.

The backslash, single-quote, and double-quote escape characters are often needed for putting those characters into a string. For instance:

```
>>> print('\' and \" are quotes')
' and " are quotes
>>> print('\\ must be written \\\\')
\ must be written \\
```

The standard way in Python for ending a line is to use the **\n** character:

```
>>> print('one\ntwo\nthree')
one
two
three
```

It's important to realize that an escape character is only a single character. The leading **** is needed to tell Python that this is a special character, but that **** does not count as an extra character when determining a string's length. For example:

```
>>> len('\\')
1
>>> len('a\nb\nc')
5
```

TABLE 6.1 Some Common Escape Characters

Character	Meaning
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\n`	Newline (linefeed)
`\r`	Return (carriage return)
`\t`	Tab (horizontal tab)

Ending Lines

Different operating systems follow different standards for ending a line of text. For instance, Windows uses **\r\n** to mark the end of a line, whereas OS X and Linux use just **\n**; Mac operating systems before OS X used **\r**.

Most good editors handle at least the **\r\n** and **\n** styles. Occasionally you will run into programs (such as Notepad on Windows) that do not recognize one of these line-end formats, so text might appear all on the same line, contain extra line breaks, or have a *^M* character at the end of each line. The easiest way to deal with this problem is to use a text editor that handles line endings correctly.

From the Library of Bill Wiecking

Slicing Strings

Slicing is how Python lets you extract a substring from a string. To slice a string, you indicate both the first character you want and one past the last character you want. For example:

```
>>> food = 'apple pie'
>>> food[0:5]
'apple'
>>> food[6:9]
'pie'
```

The indexing for slicing is the same as for accessing individual characters: The first index location is always 0, and the last is always one less than the length of the string. In general, **`s[begin:end]`** returns the substring starting at index **`begin`** and ending at index **`end - 1`**.

Note that if **`s`** is a string, then you can access the character at location **`i`** using either **`s[i]`** or **`s[i:i+1]`**.

From the Library of Bill Wiecking

Slicing shortcuts

If you leave out the begin index of a slice, then Python assumes you mean 0; and if you leave off the end index, Python assumes you want everything to the end of the string. For instance:

```
>>> food = 'apple pie'
>>> food[:5]
'apple'
>>> food[6:]
'pie'
>>> food[:]
'apple pie'
```

Here's a useful example of slicing in practice. This function returns the extension of a filename:

```
# extension.py
def get_ext(fname):
  """ Returns the extension of file
  fname.
  """
  dot = fname.rfind('.')
  if dot == -1: # no . in fname
    return ''
  else:
    return fname[dot + 1:]
```

Here's what **get_ext** does:

```
>>> get_ext('hello.text')
'text'
>>> get_ext('pizza.py')
'py'
>>> get_ext('pizza.old.py')
'py'
>>> get_ext('pizza')
''
```


From the Library of Bill Wiecking

The **get_ext** function works by determining the index position of the rightmost **'.'** (hence the use of **rfind** to search for it from right to left). If there is no **'.'** in **fname**, the empty string is returned; otherwise, all the characters from the **'.'** onward are returned.

Slicing with negative indexes

You can also use negative index values with slicing, although it can be more confusing. For example:

```
>>> food = 'apple pie'
>>> food[-9:-4]
'apple'
>>> food[:-4]
'apple'
>>> food[-3:0]
''
>>> food[-3:]
'pie'
```

When working with negative slicing, or negative indexes in general, it is often useful to write the string you are working with on a piece of paper, and then write the positive and negative index values over the corresponding characters (as in Figure 6.2). While this does take an extra minute or two, it's a good way to prevent common indexing errors.

From the Library of Bill Wiecking

Standard String Functions

Python strings come prepackaged with a number of useful functions; use **`dir`** on any string (for example, **`dir('')`**) to see them all. While it's not necessary to memorize precisely what all these functions do, it is a good idea to have a general idea of their abilities so that you can use them when you need them. Thus, in this section we present a list of all the functions that come with a string, grouped together by type.

This is not meant to be a complete reference: A few infrequently used parameters are left out, and not every detail of every function is explained. For more complete details, read a function's doc string or the online Python documentation (http://docs.python.org/3/).

Testing functions

The first and largest group of functions is composed of ones that test if a string has a certain form. The testing functions in **Table 6.2** all return either **`True`** or **`False`**. Testing functions are sometimes called *Boolean functions*, or *predicates*.

TABLE 6.2 String-Testing Functions

Name	Returns true just when ...
`s.endswith(t)`	**`s`** ends with string **`t`**
`s.startsswith(t)`	**`s`** starts with string **`t`**
`s.isalnum()`	**`s`** contains only letters or numbers
`s.isalpha()`	**`s`** contains only letters
`s.isdecimal()`	**`s`** contains only decimal characters
`s.isdigit()`	**`s`** contains only digits
`s.isidentifier()`	**`s`** is a valid Python identifier (that is, name)
`s.islower()`	**`s`** contains only lowercase letters
`s.isnumeric()`	**`s`** contains only numeric characters
`s.isprintable()`	**`s`** contains only printable characters
`s.isspace()`	**`s`** contains only whitespace characters
`s.istitle()`	**`s`** is a title-case string
`s.isupper()`	**`s`** contains only uppercase letters
`t in s`	**`s`** contains **`t`** as a substring

From the Library of Bill Wiecking

TABLE 6.3 String-Searching Functions

Name	Return Value
`s.find(t)`	–1, or index where **t** starts in **s**
`s.rfind(t)`	Same as **find**, but searches right to left
`s.index(t)`	Same as **find**, but raises **ValueError** if **t** is not in **s**
`s.rindex(t)`	Same as **index**, but searches right to left

Searching functions

As shown in **Table 6.3**, there are several ways to find substrings within a string. The difference between **index** and **find** functions is what happens when they don't find what they are looking for. For instance:

```
>>> s = 'cheese'
>>> s.index('eee')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in
  ➝ <module>
  s.index('eee')
ValueError: substring not found
>>> s.find('eee')
-1
```

The **find** function *raises* a **ValueError**; this is an example of an *exception*, which we will talk about in more detail in Chapter 9. The **index** function returns **-1** if the string being searched for is not found.

Normally, string-searching functions search the string from left to right, beginning to end. However, functions beginning with an **r** search from right to left. For example:

```
>>> s = 'cheese'
>>> s.find('e')
2
>>> s.rfind('e')
5
```

In general, **find** and **index** return the smallest index where the passed-in string starts, and **rfind** and **rindex** return the largest index where it starts.

From the Library of Bill Wiecking

Case-changing functions

Python gives you a variety of functions for changing the case of letters (**Table 6.4**). Keep in mind that Python *never* modifies a string: For all these functions, Python creates and returns a new string. We often talk as if the string were being modified, but this is only a convenient phrasing and does not mean the string is really being changed.

Formatting functions

The string-formatting functions listed in **Table 6.5** help you to make strings look nicer for presenting to the user or printing to a file.

The string **`format`** function is especially powerful, and it includes its own mini-language for formatting strings. To use **`format`**, you supply it variables or values—for example:

```
>>> '{0} likes {1}'.format('Jack',
→ 'ice cream')
'Jack likes ice cream'
```

The **`{0}`** and **`{1}`** in the string refer to the arguments in **`format`**: They are replaced by the values of the corresponding strings or variables. You can also refer to the names of keyword parameters:

```
>>> '{who} {pet} has fleas'.format
→ (pet = 'dog', who = 'my')
'my dog has fleas'
```

These examples show the most basic use of **`format`**; there are many other options for spacing strings, converting numbers to strings, and so on. All the details are provided in Python's online documentation (http://docs.python.org/3/library/string.html#format-string-syntax).

TABLE 6.4 String-Searching Functions

Name	Returned String
`s.capitalize()`	`s[0]` is made uppercase
`s.lower()`	All letters of **`s`** are made lowercase
`s.upper()`	All letters of **`s`** are made uppercase
`s.swapcase()`	Lowercase letters are made uppercase, and uppercase letters are made lowercase
`s.title()`	Title-case version of **`s`**

TABLE 6.5 String-Formatting Functions

Name	Returned String
`s.center(n, ch)`	Centers **`s`** within a string of **`n ch`** characters
`s.ljust(n, ch)`	Left-justifies **`s`** within a string of **`n ch`** characters
`s.rjust(n, ch)`	Right-justifies **`s`** within a string of **`n ch`** characters
`s.format(`*`vars`*`)`	See text

From the Library of Bill Wiecking

TABLE 6.6 String-Stripping Functions

Name	Returned String
`s.strip(ch)`	Removes all **ch** characters in **t** occurring at the beginning or end of **s**
`s.lstrip(ch)`	Removes all **ch** characters in **t** occurring at the beginning (that is, the left side) of **s**
`s.rstrip(ch)`	Removes all **ch** characters in **t** occurring at the end (that is, the right side) of **s**

TABLE 6.7 String-Splitting Functions

Name	Returned String
`s.partition(t)`	Chops **s** into three strings **(head, t, tail)**, where **head** is the substring before **t** and **tail** is the substring after **t**
`s.rpartition(t)`	Same as **partition** but searches for **t** starting at the right end of **s**
`s.split(t)`	Returns a list of substrings of **s** that are separated by **t**
`s.rsplit(t)`	Same as **split**, but starts searching for **t** at the right end of **s**
`s.splitlines()`	Returns a list of lines in **s**

Stripping functions

The stripping functions shown in **Table 6.6** are used for removing unwanted characters from the beginning or end of a string. By default, whitespace characters are stripped, and if a string argument is given, the characters in that string are stripped. For example:

```
>>> name = ' Gill Bates '
>>> name.lstrip()
'Gill Bates '
>>> name.rstrip()
' Gill Bates'
>>> name.strip()
'Gill Bates'
>>> title = '_-_- Happy Days!! _-_-'
>>> title.strip()
'_-_- Happy Days!! _-_-'
>>> title.strip('_-')
' Happy Days!! '
>>> title.strip('_ -')
'Happy Days!!'
```

Splitting functions

The splitting functions listed in **Table 6.7** chop a string into substrings.

The **partition** and **rpartition** functions divide a string into three parts:

```
>>> url = 'www.google.com'
>>> url.partition('.')
('www', '.', 'google.com')
>>> url.rpartition('.')
('www.google', '.', 'com')
```

These partitioning functions always return a value consisting of three strings in the form (head, sep, tail). This kind of return value is an example of a *tuple*, which we will learn about in more detail in Chapter 7.

From the Library of Bill Wiecking

The **split** function divides a string into substrings based on a given separator string. For example:

```
>>> url = 'www.google.com'
>>> url.split('.')
['www', 'google', 'com']
>>> story = 'A long time ago, a
➝princess ate an apple.'
>>> story.split()
['A', 'long', 'time', 'ago,', 'a',
➝'princess', 'ate', 'an', 'apple.']
```

The **split** function always returns a *list* of strings; a Python list always begins with a **[** and ends with a **]**, and uses commas to separate elements. As we'll see in Chapter 7, lists and tuples are very similar, the main difference being that lists can be modified, but tuples are constant.

Replacement functions

Python strings come with two replacing functions, as shown in **Table 6.8**. Note that the **replace** function can easily be used to delete substrings within a string:

```
>>> s = 'up, up and away'
>>> s.replace('up', 'down')
'down, down and away'
>>> s.replace('up', '')
', and away'
```

TABLE 6.8 String-Replacement Functions

Name	Returned String
`s.replace(old, new)`	Replaces every occurrence of **old** within **s** with **new**
`s.expandtabs(n)`	Replaces each tab character in **s** with **n** spaces

From the Library of Bill Wiecking

Other functions

Finally, **Table 6.9** lists the remaining string functions.

The **`translate`** and **`maketrans`** functions are useful when you need to convert one set of characters into another. For instance, here's one way to convert strings to "leet-speak":

```
>>> leet_table = ''.maketrans
→ ('EIOBT', '31087')
>>> 'BE COOL. SPEAK LEET!'.translate
→ (leet_table)
'83 COOL. SP3AK L337!'
```

The online documentation (http://docs.python.org/3/library/stdtypes.html#str.maketrans) also explains how to replace more than single characters.

The **`zfill`** function is used for formatting numeric strings:

```
>>> '23'.zfill(4)
'0023'
>>> '-85'.zfill(5)
'-0085'
```

However, it's not a very flexible function, so most programmers prefer using one of Python's other string-formatting techniques.

The **`join`** function can be quite useful. It concatenates a sequence of strings, including a separator string. For example:

```
>>> ' '.join(['once', 'upon', 'a',
→ 'time'])
'once upon a time'
>>> '-'.join(['once', 'upon', 'a',
→ 'time'])
'once-upon-a-time'
>>> ''.join(['once', 'upon', 'a',
→ 'time'])
'onceuponatime'
```

TABLE 6.9 Other String Functions

Name	Returned Value
`s.count(t)`	Number of times **`t`** occurs within **`s`**
`s.encode()`	Sets the encoding of **`s`**; see the online documentation (http://docs.python.org/3/library/stdtypes.html#str.encode) for more details
`s.join(seq)`	Concatenates the strings in **`seq`**, using **`s`** as a separator
`s.maketrans(old, new)`	Creates a translation table used to change the characters in **`old`** with the corresponding characters in **`new`**; note that **`s`** can be any string—it has no influence on the returned table
`s.translate(table)`	Makes the replacements in **`s`** using the given translation table (created with **`maketrans`**)
`s.zfill(width)`	Adds enough 0s to the left of **`s`** to make a string of length **`width`**

From the Library of Bill Wiecking

Regular Expressions

While Python strings provide many useful functions, real-world string processing often calls for more powerful tools.

Thus, programmers have developed a mini-language for advanced string processing known as *regular expressions*. Essentially, a regular expression is a way to *compactly* describe a *set* of strings. They can be used to efficiently perform common string-processing tasks such as matching, splitting, and replacing text. In this section, we'll introduce the basic ideas of regular expressions, as well as a few commonly used operators (**Table 6.10**).

TABLE 6.10 Some Regular Expression Operators

Operator	Set of Strings Described
`xy?`	`x`, `xy`
`x|y`	`x`, `y`
`x*`	`''`, `x`, `xx`, `xxx`, `xxxx`, ...
`x+`	`x`, `xx`, `xxx`, `xxxx`, ...

Simple regular expressions

Consider the string `'cat'`. It represents a single string consisting of the letters *c*, *a*, and *t*. Now consider the regular expression `'cats?'`. Here, the `?` does *not* mean an English question mark but instead represents a regular expression operator, meaning that the character to its immediate left is optional. Thus the regular expression `'cats?'` describes a set of two strings: `'cat'` and `'cats'`.

Another regular expression operator is `|`, which means "or." For example, the regular expression `'a|b|c'` describes the set of three strings `'a'`, `'b'`, and `'c'`.

The regular expression `'a*'` describes an infinite set of strings: `''`, `'a'`, `'aa'`, `'aaa'`, `'aaaa'`, `'aaaaa'`, and so on. In other words, `'a*'` describes the set of all strings consisting of a sequence of 0 or more `'a'`s. The regular expression `'a+'` is the same as `'a*'` but excludes the empty string `''`.

From the Library of Bill Wiecking

Finally, within a regular expression you can use round brackets to indicate what substring an operator ought to apply to. For example, the regular expression **'(ha)+!'** describes these strings: **'ha!'**, **'haha!'**, **'hahaha!'**, and so on. In contrast, **'ha+!'** describes a very different set: **'ha!'**, **'haa!'**, **'haaa!'**, and so on.

You can mix and match these (and many other) regular expression operators in any way you want. This turns out to be a very useful way to describe many commonly occurring types of strings, such as phone numbers or email addresses.

Matching with regular expressions

A common application of regular expressions is string matching. For example, suppose you are writing a program where the user must enter a string such as **done** or **quit** to end the program. To help recognize these strings, you could write a function like this:

```
# allover.py
def is_done1(s):
  return s == 'done' or s == 'quit'
```

Using regular expressions, an identically behaving function might look like this:

```
# allover.py
import re # use regular expressions
def is_done2(s):
  return re.match('done|quit', s)
  → != None
```

The first line of this new version imports Python's standard regular expression library. To match a regular expression, we use the **re.match(regex, s)** function, which returns **None** if **regex** does not match **s**, and a special regular expression *match object* otherwise. We don't care about the details of the match object in this example, so we only check to see if the result is **None** or not.

In such a simple example, the regular expression version is not much shorter or better than the first version; indeed, **is_done1** is probably preferable! However, regular expressions really start to shine as your programs become larger and more complex. For instance, suppose we decide to add a few more possible stopping strings. For the regular expression version, we just rewrite the regular expression string to be, say, **'done|quit|over|finished|end|stop'**. In contrast, to make the same change to the first version, we'd need to include **or s ==** for each string we added, which would make for a very long line of code that would be hard to read.

Here's a more complex example. Suppose you want to recognize *funny strings*, which consist of one or more **'ha'** strings followed immediately by one or more **'!'**s. For example, **'haha!'**, **'ha!!!!!'**, and **'hahaha!!'** are all funny strings. It's easy to match these using regular expressions:

```
# funny.py
import re
def is_funny(s):
  return re.match('(ha)+!+', s)
  → != None
```

Notice that the only essential difference between this **is_funny** and **is_done2** is that a different regular expression is used inside **match.** If you try writing this same function without using regular expressions, you will quickly see how much work **'(ha)+!+'** is doing for us.

continues on next page

From the Library of Bill Wiecking

More regular expressions

We have barely scratched the surface of regular expressions: Python's **`re`** library is large and has many regular expression functions that can perform string-processing tasks such as matching, splitting, and replacing. There are also tricks for speeding up the processing of commonly used regular expressions, and numerous shortcuts for matching commonly used characters. The documentation for the **`re`** module contains more examples (http://docs.python.org/3/library/re.html).

From the Library of Bill Wiecking

7

Data Structures

In this chapter, we introduce the important idea of *data structures*: collections of values along with commonly performed functions. Python's programmer-friendly philosophy is to provide a few powerful and efficient data structures—*tuples*, *lists*, *dictionaries*, and *sets*—that can be combined as needed to make more complex ones.

In the previous chapter we discussed strings, which can be thought of as data structures restricted to storing sequences of characters. The data structures in this chapter can contain not just characters but almost any kind of data.

Python's two workhorse data structures are lists and dictionaries. Lists store data in sequential order, and dictionaries are like little databases that efficiently store and retrieve data using keys.

In This Chapter

The type Command	102
Sequences	103
Tuples	104
Lists	108
List Functions	110
Sorting Lists	113
List Comprehensions	115
Dictionaries	118
Sets	122

From the Library of Bill Wiecking

The type Command

It's occasionally useful to check the data type of a value or a variable. This is easily done with the built-in **type** command:

```
>>> type(5)
<class 'int'>
>>> type(5.0)
<class 'float'>
>>> type('5')
<class 'str'>
>>> type(None)
<class 'NoneType'>
>>> type(print)
<class 'builtin_function_or_method'>
```

Notice that the output of the **type** command uses the term *class*. Roughly speaking, classes and types are synonymous.

The **type** command can be quite useful for debugging. For instance, it is not unusual in Python to work with data collections where you don't know the exact type of the data items, or where you don't even know the exact type of the container holding them. Using **type**, you can always determine the exact type of a Python object.

From the Library of Bill Wiecking

Sequences

In Python, a *sequence* is an ordered collection of values. Python has three built-in sequence types: *strings*, *tuples*, and *lists*.

One very nice feature of sequences is that they can be indexed and sliced, just as we saw for strings in the previous chapter. Thus, all sequences have the following characteristics:

- Their first positive index is 0, and it is at the left end.
- Their first negative index is –1, and it starts at the right end.
- Slice notation can be used to make copies of sub-sequences. For example, **`seq[begin:end]`** returns a copy of the elements of **`seq`** starting at index location **`begin`** and ending at location **`end - 1`**.
- They can be concatenated (i.e., combined) using **`+`** and **`*`**. The sequences must be of the same type for this to work—that is to say, you cannot concatenate a tuple and a list.
- Their length is calculated by the **`len`** function. For example, **`len(s)`** is the number of items in sequence **`s`**.
- The expression **`x in s`** tests if the sequence **`s`** contains the element **`x`**. That is, **`x`** in **`s`** returns **`True`** if **`x`** is somewhere in **`s`**, and **`False`** otherwise.

In practice, strings and lists are the most common kinds of sequences. Tuples have their uses but appear much less often.

Order Matters

When we say that sequences are *ordered*, we mean that the order of the elements in the sequence matters. Strings are ordered because '**`abc`**' is different from '**`acb`**'. Later we will see that dictionaries and sets are not ordered: They only care if an item is inside of them, and in fact they can make no promises about their relative order.

How Big Can a Sequence Be?

Theoretically, there is no limit to the length of a sequence: It can contain as many items as needed. Practically, however, you are restricted by the amount of RAM available in your computer when Python is running.

From the Library of Bill Wiecking

Tuples

A *tuple* is an *immutable* sequence of 0 or more values. It can contain any Python value—even other tuples. For example:

```
>>> items = (-6, 'cat', (1, 2))
>>> items
(-6, 'cat', (1, 2))
>>> len(items)
3
>>> items[-1]
(1, 2)
>>> items[-1][0]
1
```

As you can see, the items of a tuple are enclosed in round brackets and separated by commas. The empty tuple is represented by `()`, but tuples with a single item (*singleton tuples*) have the unusual notation `(x,)`. For instance:

```
>>> type(())
<class 'tuple'>
>>> type((5,))
<class 'tuple'>
>>> type((5))
<class 'int'>
```

If you forget the comma at the end of a singleton tuple, you have not created a tuple—all you've done is put brackets around an expression.

Trailing Commas

Whereas singleton tuples *require* a trailing comma, a trailing comma is allowed, but not required, in longer tuples (and lists). For example, `(1, 2, 3,)` is the same as `(1, 2, 3)`. Some programmers prefer to always include the trailing comma so that they never accidentally leave it out for singleton tuples.

From the Library of Bill Wiecking

Tuple immutability

As mentioned, tuples are immutable, meaning that once you've created a tuple, you cannot change it. This is not so unusual: Strings, integers, and floats are also immutable. If you do need to change a tuple, then you must create a new tuple that embodies the changes. For example, here's how you can chop off the first element of a tuple:

```
>>> lucky = (6, 7, 21, 77)
>>> lucky
(6, 7, 21, 77)
>>> lucky2 = lucky[1:]
>>> lucky2
(7, 21, 77)
>>> lucky
(6, 7, 21, 77)
```

On the plus side, immutability makes it impossible to accidentally modify a tuple, which helps prevent errors. On the minus side, making even the smallest change to a tuple requires copying essentially the whole thing, and so modifying large tuples can takes extra time and memory. If you find yourself needing to make frequent modifications to a tuple, then you should be using a list instead.

From the Library of Bill Wiecking

Tuple functions

Table 7.1 lists the most commonly used tuple functions. Compared with strings and lists, tuples have a relatively small number of functions. Here are some examples of how they are used:

```
>>> pets = ('dog', 'cat', 'bird',
→ 'dog')
>>> pets
('dog', 'cat', 'bird', 'dog')
>>> 'bird' in pets
True
>>> 'cow' in pets
False
>>> len(pets)
4
>>> pets.count('dog')
2
>>> pets.count('fish')
0
>>> pets.index('dog')
0
>>> pets.index('bird')
2
>>> pets.index('mouse')
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in
  → <module>
  pets.index('mouse')
ValueError: tuple.index(x): x not in
→ list
```

TABLE 7.1 Tuple Functions

Name	Return Value
`x in tup`	`True` if `x` is an element of `tup`, `False` otherwise
`len(tup)`	Number of elements in `tup`
`tup.count(x)`	Number of times element `x` occurs in `tup`
`tup.index(x)`	Index location of the first (leftmost) occurrence of `x` in `tup`; if `x` is not in `tup`, raises a `ValueError` exception

From the Library of Bill Wiecking

As with strings, you can use **+** and ***** to concatenate tuples:

```
>>> tup1 = (1, 2, 3)
>>> tup2 = (4, 5, 6)
>>> tup1 + tup2
(1, 2, 3, 4, 5, 6)
>>> tup1 * 2
(1, 2, 3, 1, 2, 3)
```


From the Library of Bill Wiecking

Lists

Lists are essentially the same as tuples but with one key difference: Lists are *mutable*. That is, you can add, remove, or modify elements to a list without making a copy. In practice, lists are used far more frequently than tuples (indeed, some Python programmers are only faintly aware that tuples exist!).

The elements of a list are separated by commas and enclosed in square brackets. As with strings and tuples, you can easily get the length of a list (using **len**), and concatenate lists (using **+** and *****):

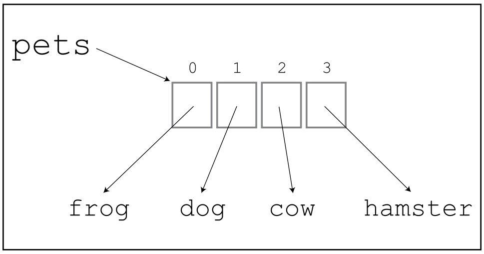
```
>>> numbers = [7, -7, 2, 3, 2]
>>> numbers
[7, -7, 2, 3, 2]
>>> len(numbers)
5
>>> numbers + numbers
[7, -7, 2, 3, 2, 7, -7, 2, 3, 2]
>>> numbers * 2
[7, -7, 2, 3, 2, 7, -7, 2, 3, 2]
```

And just as with strings and tuples, you can use indexing and slicing to access individual elements and sublists:

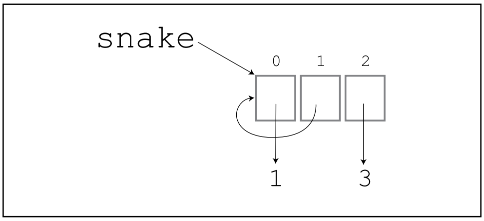
```
>>> lst = [3, (1,), 'dog', 'cat']
>>> lst[0]
3
>>> lst[1]
(1,)
>>> lst[2]
'dog'
>>> lst[1:3]
[(1,), 'dog']
>>> lst[2:]
['dog', 'cat']
>>> lst[-3:]
[(1,), 'dog', 'cat']
>>> lst[:-3]
[3]
```

Notice that lists can contain any kinds of values: numbers, strings, or even other sequences. The *empty list* is denoted by **[]**, and a singleton list containing just one element **x** is written **[x]** (in contrast to tuples, no trailing comma is necessary for a singleton list).

From the Library of Bill Wiecking



A A Python list points to its values.



B A self-referential list. Note that the second element is not pointing to the first element of the list, but to the entire list itself.

Lingo Alert

Many Python programmers speak as if a list *contains* its elements. Although that is not technically accurate, it is a common and convenient phrasing. Much of the time it does not cause any confusion. But when it comes to finding errors in programs that process lists, it is often essential to understand that lists actually point to their values and don't contain them.

Mutability

As mentioned earlier, mutability is the key feature that distinguishes lists from tuples. For example:

```
>>> pets = ['frog', 'dog', 'cow',
→ 'hamster']
>>> pets
['frog', 'dog', 'cow', 'hamster']
>>> pets[2] = 'cat'
>>> pets
['frog', 'dog', 'cat', 'hamster']
```

As you can see, this sets the second element of the list **pets** to point to **'cat'**. The string **'cow'** gets replaced and is automatically deleted by Python.

Figure **A** shows a helpful diagrammatic representation of a list. Just as with variables, it is important to understand that the elements of a list only *point* to their values and do not actually contain them.

The fact that lists point to their values can be the source of some surprising behavior. Consider this nasty example:

```
>>> snake = [1, 2, 3]
>>> snake[1] = snake
>>> snake
[1, [...], 3]
```

Here, we've made an element of a list point to the list itself: We've created a *self-referential data structure*. The **[...]** in the printout indicates that Python recognizes the self-reference and does not stupidly print the list forever(!). Figure **B** shows diagrammatically what **snake** looks like.

From the Library of Bill Wiecking

List Functions

Lists come with many useful functions (**Table 7.2**). All of these functions, except for **`count`** (which just returns a number), modify the list you call them with. Thus, these are *mutating functions*, so you need to use them with care. It is distressingly easy, for instance, to accidentally delete a wrong element or insert a new value at the wrong place.

The **`append`** function adds an element to the end of a list. One common programming pattern is to create an empty list at the beginning of a function and then add values to it in the rest of the function. For example, here is a function that creates a string of messages based on a list of input numbers:

```
# numnote.py
def numnote(lst):
  msg = []
  for num in lst:
    if num < 0:
      s = str(num) + ' is negative'
    elif 0 <= num <= 9:
      s = str(num) + ' is a digit'
    msg.append(s)
  return msg
```

For example:

```
>>> numnote([1, 5, -6, 22])
['1 is a digit', '5 is a digit',
'-6 is negative']
```

TABLE 7.2 List Functions

Name	Return Value
`s.append(x)`	Appends **`x`** to the end of **`s`**
`s.count(x)`	Returns the number of times **`x`** appears in **`s`**
`s.extend(lst)`	Appends each item of **`lst`** to **`s`**
`s.index(x)`	Returns the index value of the leftmost occurrence of **`x`**
`s.insert(i, x)`	Inserts **`x`** before index location **`i`** (so that **`s[i] == x`**)
`s.pop(i)`	Removes and returns the item at index **`i`** in **`s`**
`s.remove(x)`	Removes the leftmost occurrence of **`x`** in **`s`**
`s.reverse()`	Reverses the order of the elements of **`s`**
`s.sort()`	Sorts the elements of **`s`** into increasing order

From the Library of Bill Wiecking

Lingo Alert

In computer programming, the term *pop* usually refers to the act of removing the last element of a list. The related term, *push*, refers to adding an element to the same end (that is, exactly what Python's **append** does). When **push** and **pop** are used on the same list, we often refer to it as a *stack*: We say that items are pushed onto the *top* of the stack and then popped from the top of the stack. Despite their simplicity, stacks form the basis of a number of more advanced programming behaviors, such as recursion and undo.

To print the messages on their own individual lines, you could do this:

```
>>> for msg in numnote([1, 5, -6,
→ 22]): print(msg)
1 is a digit
5 is a digit
-6 is negative
```

Or even this:

```
>>> print('\n'.join(numnote([1, 5,
→ -6, 22])))
1 is a digit
5 is a digit
-6 is negative
```

The **extend** function is similar to **append**, but it adds an entire sequence:

```
>>> lst = []
>>> lst.extend('cat')
>>> lst
['c', 'a', 't']
>>> lst.extend([1, 5, -3])
>>> lst
['c', 'a', 't', 1, 5, -3]
```


From the Library of Bill Wiecking

The **pop** function removes an element at a given index position and then returns it. For example:

```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst.pop(2)
'c'
>>> lst
['a', 'b', 'd']
>>> lst.pop()
'd'
>>> lst
['a', 'b']
```

Notice that if you don't give **pop** an index, it removes and returns the element at the end of the list.

The **remove(x)** function removes the first occurrence of **x** from a list. However, it does not return **x**:

```
>>> lst = ['a', 'b', 'c', 'a']
>>> lst.remove('a')
>>> lst
['b', 'c', 'a']
```

As the name suggests, **reverse** reverses the order of the elements of a list:

```
>>> lst = ['a', 'b', 'c', 'a']
>>> lst
['a', 'b', 'c', 'a']
>>> lst.reverse()
>>> lst
['a', 'c', 'b', 'a']
```

It's important to realize that **reverse** does not make a copy of the list: It moves the items within the list itself, so we say the reversal is done *in place*.

From the Library of Bill Wiecking

Lingo Alert

The order in which Python sorts a list of sequences is called *lexicographical ordering*. This is just a general term meaning "alphabetical order," except that it applies to any sequence of orderable values, not just letters. The idea is that elements are ordered by their initial element, then their second element, then their third element, and so on.

Sorting Lists

Sorting data is one of the most common things that computers do. Sorted data is usually easier to work with than unsorted data, for both humans and computers. For instance, finding the smallest element of a sorted list requires no searching at all: It is simply the first element of the list. Humans often prefer to see data in sorted order—just imagine a phone book that was not printed alphabetically!

In Python, sorting is most easily done using the list **`sort()`** function. In practice, it can be used to quickly sort lists with tens of thousands of elements. Like **`reverse()`**, **`sort()`** modifies the list in place:

```
>>> lst = [6, 0, 4, 3, 2, 6]
>>> lst
[6, 0, 4, 3, 2, 6]
>>> lst.sort()
>>> lst
[0, 2, 3, 4, 6, 6]
```


From the Library of Bill Wiecking

The **sort** function always sorts elements into ascending order, from smallest to largest. If you want the elements sorted in reverse order, from largest to smallest, the simple trick of calling **reverse** after **sort** works well:

```
>>> lst = ['up', 'down', 'cat',
→ 'dog']
>>> lst
['up', 'down', 'cat', 'dog']
>>> lst.sort()
>>> lst
['cat', 'dog', 'down', 'up']
>>> lst.reverse()
>>> lst
['up', 'down', 'dog', 'cat']
```

Python also knows how to sort tuples and lists. For example:

```
>>> pts = [(1, 2), (1, -1), (3, 5),
→ (2, 1)]
>>> pts
[(1, 2), (1, -1), (3, 5), (2, 1)]
>>> pts.sort()
>>> pts
[(1, -1), (1, 2), (2, 1), (3, 5)]
```

Tuples (and lists) are sorted by their first element, then by their second element, and so on.

From the Library of Bill Wiecking

List Comprehensions

Lists are used so frequently that Python provides a special notation for creating them called *list comprehensions*. For example, here's how you can use a list comprehension to create a list of the squares of the numbers from 1 to 10:

```
>>> [n * n for n in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
→ 100]
```

The main advantage of this notation is that it is compact *and* readable. Compare this with equivalent code without a comprehension:

```
result = []
for n in range(1, 11):
  result.append(n * n)
```

Once you get the hang of them, list comprehensions are quick and easy to write, and you will find many uses for them.

From the Library of Bill Wiecking

Examples of list comprehensions

Let's see a few more examples of comprehensions. If you want to double each number on the list and 7, you can do this:

```
>>> [2 * n + 7 for n in
→ range(1, 11)]
[9, 11, 13, 15, 17, 19, 21, 23,
→ 25, 27]
```

Or if you want the first ten cubes:

```
>>> [n ** 3 for n in range(1, 11)]
[1, 8, 27, 64, 125, 216, 343, 512,
→ 729, 1000]
```

You can also use strings in comprehensions. For example:

```
>>> [c for c in 'pizza']
['p', 'i', 'z', 'z', 'a']
>>> [c.upper() for c in 'pizza']
['P', 'I', 'Z', 'Z', 'A']
```

A common application of comprehensions is to modify an existing list in some way. For instance:

```
>>> names = ['al', 'mei', 'jo',
→ 'del']
>>> names
['al', 'mei', 'jo', 'del']
>>> cap_names = [n.capitalize() for
→ n in names]
>>> cap_names
['Al', 'Mei', 'Jo', 'Del']
>>> names
['al', 'mei', 'jo', 'del']
```


From the Library of Bill Wiecking

Filtered comprehensions

List comprehensions can also filter out elements you don't want. For example, the following comprehension returns a list containing just the positive elements of **nums**:

```
>>> nums = [-1, 0, 6, -4, -2, 3]
>>> result = [n for n in nums if
→n > 0]
>>> result
[6, 3]
```

Here's equivalent code without a comprehension:

```
result = []
nums = [-1, 0, 6, -4, -2, 3]
for n in nums:
  if n > 0:
    result.append(n)
```

Again, we see that list comprehensions are more compact and readable than a regular loop.

Here's a comprehension that removes all the vowels from a word written inside a function:

```
# eatvowels.py
def eat_vowels(s):
  """ Removes the vowels from s.
  """
  return ''.join([c for c in s
  →if c.lower() not in 'aeiou'])
```

It works like this:

```
>>> eat_vowels('Apple Sauce')
'ppl Sc'
```

The body of **eat_vowels** looks rather cryptic at first, and the trick to understanding it is to read it a piece at a time. First, look at the comprehension:

```
[c for c in s if c.lower() not in
→'aeiou']
```

This is a filtered comprehension that scans through the characters of **s** one at a time. It converts each character to lowercase and then checks to see if it is a vowel. If it is a vowel, it is skipped and not added to the resulting list; otherwise, it is added.

The result of this comprehension is a list of strings, so we use **join** to concatenate all the strings into a single string that is then immediately returned.

Generator Expressions

There's one more simplification we could make to **eat_vowels**: The square brackets in the comprehension can be removed:

```
' '.join(c for c in s if c.lower() not in 'aeiou')
```

The expression inside **join** is an example of a *generator expression*. In more advanced Python programming, generator expressions can be used to efficiently generate only the needed part of a list or sequence, with the elements being generated *on demand* instead of all at once as with a list comprehension.

From the Library of Bill Wiecking

Dictionaries

A Python *dictionary* is an extremely efficient data structure for storing *pairs* of values in the form *key:value*. For example:

```
>>> color = {'red' : 1, 'blue' : 2,
→ 'green' : 3}
>>> color
{'blue': 2, 'green': 3, 'red': 1}
```

The dictionary **color** has three members. One of them is **'blue':2**, where **'blue'** is the *key* and **2** is its associated *value*.

You access values in a dictionary by their keys:

```
>>> color['green']
3
>>> color['red']
1
```

Accessing dictionary values by their keys is extremely efficient, even if the dictionary has many thousands of pairs.

Like lists, dictionaries are mutable: You can add or remove key:value pairs. For example:

```
>>> color = {'red' : 1, 'blue' : 2,
→ 'green' : 3}
>>> color
{'blue': 2, 'green': 3, 'red': 1}
>>> color['red'] = 0
>>> color
{'blue': 2, 'green': 3, 'red': 0}
```

Lingo Alert

Dictionaries are also referred to as *associative arrays*, *maps*, or *hash tables*.

Hashing

Python's dictionaries use a clever programming trick known as *hashing*. Essentially, each key in a dictionary is converted to a number called its *hash value* using a specially designed *hash function*. The associated values are stored in an underlying list at the index location of their hash value. Accessing a value involves converting the supplied key to a hash value and then jumping to that index location in the list. The exact details of hashing are tricky, but thankfully Python takes care of everything for us.

From the Library of Bill Wiecking

Key restrictions

Dictionary keys have a couple of restrictions you need to be aware of. First, keys are *unique* within the dictionary: You can't have two key:value pairs in the same dictionary with the same key. For example:

```
>>> color = {'red' : 1, 'blue' : 2,
→ 'green' : 3, 'red' : 4}
>>> color
{'blue': 2, 'green': 3, 'red': 4}
```

Even though we've written the key **'red'** twice, Python only stores the second pair, **'red':4**. There's simply no way to have duplicate keys: *Dictionary keys must always be unique*.

The second restriction on keys is that they must be immutable. So, for example, a dictionary key *cannot* be a list or another dictionary. The reason for this requirement is that the location in a dictionary where a key:value pair is stored is calculated from the key. If the key changes even slightly, the location of the key:value pair in the dictionary can also change. If that happens, then pairs in the dictionary can become lost and inaccessible.

Neither of these restrictions holds for values. Values can be mutable and can appear as many times as you like within the same dictionary.

From the Library of Bill Wiecking

Dictionary functions

Table 7.3 lists the functions that come with all dictionaries.

As we've seen, the standard way to retrieve a value from a dictionary is to use square-bracket notation: **`d[key]`** returns the value associated with **`key`**. Calling **`d.get(key)`** will do the same thing. If you call either function when **`key`** is not in **`d`**, you'll get a **`KeyError`**.

If you are not sure whether a key is in a dictionary ahead of time, you can check by calling **`key in d`**. This expression returns **`True`** if **`key`** is in the **`d`**, and **`False`** otherwise. It is an extremely efficient check (especially as compared with using **`in`** with sequences!).

You can also retrieve dictionary values using the **`pop(key)`** and **`popitem()`** functions. The difference between **`pop(key)`** and **`get(key)`** is that **`pop(key)`** returns the value associated with **`key`** and also

TABLE 7.3 Dictionary Functions

Name	Return Value
`d.items()`	Returns a view of the (key, value) pairs in **`d`**
`d.keys()`	Returns a view of the keys of **`d`**
`d.values()`	Returns a view of the values in **`d`**
`d.get(key)`	Returns the value associated with **`key`**
`d.pop(key)`	Removes **`key`** and returns its corresponding value
`d.popitem()`	Returns some (key, value) pair from **`d`**
`d.clear()`	Removes all items from **`d`**
`d.copy()`	A copy of **`d`**
`d.fromkeys(s, t)`	Creates a new dictionary with keys taken from **`s`** and values taken from **`t`**
`d.setdefault(key, v)`	If **`key`** is in **`d`**, returns its value; if **`key`** is not in **`d`**, returns **`v`** and adds (**`key`**, **`v`**) to **`d`**
`d.update(e)`	Adds the (key, value) pairs in **`e`** to **`d`**; **`e`** may be another dictionary or a sequence of pairs

From the Library of Bill Wiecking

removes its pair from the dictionary (**`get`** only returns the value). The **`popitem()`** function returns and removes *some* (key, value) pair from the dictionary. You don't know ahead of time which pair will be popped, so it's useful only when you don't care about the order in which you access the dictionary elements.

The **`items()`**, **`keys()`**, and **`values()`** functions all return a special object known as a *view*. A view is linked to the original dictionary, so that if the dictionary changes, so does the view. For example:

```
>>> color
{'blue': 2, 'orange': 4, 'green': 3,
→ 'red': 0}
>>> k = color.keys()
>>> for i in k: print(i)
blue
orange
green
red
>>> color.pop('red')
0
>>> color
{'blue': 2, 'orange': 4, 'green': 3}
>>> for i in k: print(i)
blue
orange
green
```


From the Library of Bill Wiecking

Sets

In Python, *sets* are collections of 0 or more items with *no duplicates*. A set is similar to a dictionary that only has keys and no associated values.

Sets come in two categories: *mutable sets* and *immutable frozensets*. You can add and remove elements from a regular set, whereas a frozenset can never change once it is created.

Perhaps the most common use of sets is to remove duplicates from a sequence. For example:

```
>>> lst = [1, 1, 6, 8, 1, 5, 5, 6,
→ 8, 1, 5]
>>> s = set(lst)
>>> s
{8, 1, 5, 6}
```

Just as with dictionaries, the order of the elements in the set cannot be guaranteed.

Calling **`dir(set)`** in the interactive command line will list the functions that all sets come with—there are quite a few! Since sets are not as frequently used as lists and dictionaries, we won't list them all here. But keep sets in mind, and when you need them, refer to their online documentation at http://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset.

Sets and Dictionaries

Sets are a relatively new addition to Python. Before sets, programmers used dictionaries to simulate sets, and indeed the first implementations of sets in Python did the same. If you find yourself using dictionaries and not caring about the values, then changing your code to use sets might make it more readable.

From the Library of Bill Wiecking

8

Input and Output

To be useful, a program needs to communicate with the world around it. It needs to interact with the user, or read and write files, or access webpages, and so on. In general, we refer to this as *input and output*, or *I/O* for short.

We've already seen basic *console I/O*, which involves printing messages and using the **`input`** function to read strings from the user. Now we'll see some string formatting that lets you make fancy output strings for console I/O and anywhere you need a formatted string.

Then we'll turn to *file I/O*, which is all about reading and writing files. Python provides a lot of support for basic file I/O, making it as easy as possible for programmers. In particular, we'll see how to use text files, binary files, and the powerful **`pickle`** module.

In This Chapter

Formatting Strings	124
String Formatting	126
Reading and Writing Files	128
Examining Files and Folders	131
Processing Text Files	134
Processing Binary Files	138
Reading Webpages	141

From the Library of Bill Wiecking

Formatting Strings

Python provides a number of different ways to create formatted strings. We will discuss the older string interpolation and the newer *format strings*.

String interpolation

String interpolation is a simple approach to string formatting that Python borrows from the C programming language. For instance, here's how you can control the number of decimal places in a float:

```
>>> x = 1/81
>>> print(x)
0.0123456790123
>>> print('value: %.2f' % x)
value: 0.01
>>> print('value: %.5f' % x)
value: 0.01235
```

String interpolation expressions always have the form **`format % values`**, where **`format`** is a string containing one or more occurrences of the **`%`** character. In the example **`'x = %.2f' % x`**, the substring **`%.2f`** is a formatting command that tells Python to take the first supplied value (**`x`**) and to display it as a floating point value with two decimal places.

From the Library of Bill Wiecking

TABLE 8.1 Some Conversion Specifiers

Specifier	Meaning
`d`	Integer
`o`	Octal (base 8) value
`x`	Lowercase hexadecimal (base 16)
`X`	Uppercase hexadecimal (base 16)
`e`	Lowercase float exponential
`E`	Uppercase float exponential
`F`	Float
`s`	String
`%`	% character

Octal and Hexadecimal

The **o** and **x** conversion specifiers, which convert values to base 8 (*octal*) and base 16 (*hexadecimal*), respectively, might seem to be of questionable value. However, in many computer-oriented applications it is convenient to represent values in base 16 or, less frequently, base 8. As we will see later in this chapter, hexadecimal is commonly used when dealing with binary files.

Conversion specifiers

The character `f` in the format string is a *conversion specifier*, and it tells Python how to render the corresponding value. **Table 8.1** lists the most commonly used conversion specifiers.

The `e`, `E`, and `f` specifiers give you different ways of representing floats. For example:

```
>>> x
0.012345679012345678
>>> print('x = %f' % x)
x = 0.012346
>>> print('x = %e' % x)
x = 1.234568e-02
>>> print('x = %E' % x)
x = 1.234568E-02
```

You can put as many specifiers as you need in a format string, although you must supply exactly one value for each specifier. For example:

```
>>> a, b, c = 'cat', 3.14, 6
>>> s = 'There\'s %d %ss older than
→%.2f years' % (c, a, b)
>>> s
"There's 6 cats older than 3.14
→years"
```

As this example shows, the format string acts as a simple template that gets filled in by the values. The values are given in a tuple, and they must be in the order in which you want them replaced.

TIP The `d`, `f`, and `s` conversion specifiers are the most frequently used, so they are the ones worth remembering. In particular, `f` is the easiest way to control the format of floats.

TIP If you need the % character to appear as % itself, then you must type `'%%'`.

From the Library of Bill Wiecking

String Formatting

A second way to create fancy strings in Python is to use *format strings* with the string function **`format(value, format_spec)`**. For example:

```
>>> 'My {pet} has {prob}'.format
→(pet = 'dog', prob='fleas')
'My dog has fleas'
```

In a format string, anything within curly braces is replaced. This is called *named replacement*, and it is especially readable in this example.

You can also replace values by *position*:

```
>>> 'My {0} has {1}'.format
→('dog', 'fleas')
'My dog has fleas'
```

Or apply formatting codes similar to interpolated strings:

```
>>> '1/81 = {x}'.format(x=1/81)
'1/81 = 0.0123456790123'
>>> '1/81 = {x:f}'.format(x=1/81)
'1/81 = 0.012346'
>>> '1/81 = {x:.3f}'.format(x=1/81)
'1/81 = 0.012'
```

Templating Packages

When neither string interpolation nor format strings are powerful or flexible enough, you may want to use a templating package, such as Cheetah (www.cheetahtemplate.org) or the one that comes with Django (www.djangoproject.com). Both allow you to do some very sophisticated replacements and are good choices if you are making, say, a lot of dynamically generated webpages.

From the Library of Bill Wiecking

You can specify formatting parameters *within* braces, like this:

```
>>> 'num = {x:.{d}f}'.format
→ (x=1/81, d=3)
'num = 0.012'
>>> 'num = {x:.{d}f}'.format
→ (x=1/81, d=4)
'num = 0.0123'
```

This is something you can't do with regular string interpolation.

TIP If you need the { or }characters to appear as themselves in a format string, type them as {{ and }}.

TIP Format strings are more flexible and powerful than string interpolation, but also more complicated. If you are creating only a few simple formatted strings, string interpolation is probably the best choice. Otherwise, format strings are more useful for larger and more complex formatting jobs, such as creating webpages or form letters for email.

From the Library of Bill Wiecking

Reading and Writing Files

A *file* is a named collection of bits stored on a *secondary storage device*, such as a hard disk, USB drive, flash memory stick, and so on. We distinguish between two categories of files: *text files*, which are essentially strings stored on disk, and *binary files*, which are everything else.

Text files have the following characteristics:

- They are essentially "strings on disk." Python source code files and HTML files are examples of text files.
- They can be edited with any text editor. Thus, they are relatively easy for humans to read and modify.
- They tend to be difficult for *programs* to read. Typically, programs called *parsers* are needed to read each different kind of text file. For instance, Python uses a special-purpose parser to help read `.py` files, and an HTML-specific parser is needed to read HTML files.
- They are usually larger than equivalent binary files. This can be a major problem when, for instance, you need to send a large text file over the Internet. Thus, text files are often compressed (for example, into *zip* format) to speed up transmission and to save disk space.

From the Library of Bill Wiecking

Binary files have the following characteristics:

- They are not usually human-readable, at least within a regular text editor. A binary file is displayed in a text editor as a random-looking series of characters. Some kinds of binary files, such as JPEG image files, have special viewers for displaying their content.
- They usually take up less space than equivalent text files. For instance, a binary file might group the information within it in chunks of 32 bits without using separator characters, such as commas or spaces.
- They are often easier for programs to read and write than text files. Although each binary file is different, it's often *not* necessary to write complex parsers to read them.
- They are often tied to a specific program and are often unusable if you lack that program. Some popular binary files may have their file formats published so that you can, if so motivated, write your programs to read and write them. However, this usually requires substantial effort.

From the Library of Bill Wiecking

Folders

In addition to files, *folders* (or *directories*) are used to store files and other folders. The folder structure of most file systems is quite large and complex, forming a *hierarchical folder structure*.

A *pathname* is the name used to identify a file or a folder. The *full pathname* can be quite long. For example, the Python folder on my Windows computer has this full pathname: C:\Documents and Settings\tjd\Desktop\python.

Windows pathnames use a *backward slash* (\) character to separate names in a path, and they start with the letter of the disk drive (in this example, *C:*).

On Mac and Linux systems, a *forward slash* (/) is used to separate names. Plus, there is no drive letter at the start. For example, here is the full pathname for my Python folder on Linux: /home/tjd/Desktop/python.

TIP Recall that if you want to write a \ character in a Python string, it must be doubled:

```
'C:\\home\\tjd\\Desktop\\python'
```

To avoid the double backslashes, you can use a *raw string*:

```
r'C:\home\tjd\Desktop\python'
```

TIP Getting Python programs to work with both styles of pathnames is a bit tricky, and you should read the documentation for Python's `os.path` module for (much!) more information.

The current working directory

Many programs use the idea of a *current working directory*, or *cwd*. This is simply one directory that has been designated as the *default directory*: Whenever you do something to a file or a folder without providing a full pathname, Python assumes you mean a file or a folder in the current working directory.

From the Library of Bill Wiecking

Examining Files and Folders

Python provides many functions that return information about your computer's files and folders (its *file system*). **Table 8.2** lists a few of the most useful ones.

Let's write a couple of useful functions to see how these work. For instance, a common task is retrieving the files and folders in the current working directory. Writing **`os.listdir(os.getcwd())`** is unwieldy, so we can write this function:

```
# list.py
def list_cwd():
  return os.listdir(os.getcwd())
```

The following two related helper functions use list comprehensions to return just the files and folders in the current working directory:

```
# list.py
def files_cwd():
  return [p for p in list_cwd()
          if os.path.isfile(p)]
def folders_cwd():
  return [p for p in list_cwd()
          if os.path.isdir(p)]
```

TABLE 8.2 Useful File and Folder Functions

Name	Action
`os.getcwd()`	Returns the name of the current working directory
`os.listdir(p)`	Returns a list of strings of the names of all the files and folders in the folder specified by path **`p`**
`os.chdir(p)`	Sets the current working directory to be path **`p`**
`os.path.isfile(p)`	Returns **`True`** just when path **`p`** specifies the name of a file, and **`False`** otherwise
`os.path.isdir(p)`	Returns **`True`** just when path **`p`** specifies the name of a folder, and **`False`** otherwise
`os.stat(fname)`	Returns information about **`fname`**, such as its size in bytes and the last modification time

From the Library of Bill Wiecking

If you just want a list of, say, the **.py** files in the current working directory, then this will work:

```
# list.py
def list_py(path = None):
  if path == None:
    path = os.getcwd()
    return [fname for fname in
    → os.listdir(path)
      if os.path.isfile(fname)
      if fname.endswith('.py')]
```

This function plays a useful trick with its input parameter: If you call **list_py()** without a parameter, it runs on the current working directory. Otherwise, it runs on the directory you pass in.

Finally, here's a function that returns the sum of the sizes of the files in the current working directory:

```
# list.py
def size_in_bytes(fname):
  return os.stat(fname).st_size
def cwd_size_in_bytes():
  total = 0
  for name in files_cwd():
    total = total +
    → size_in_bytes(name)
  return total
```

A Neat Trick

The **cwd_size_in_bytes** function can be written as a single-line function:

```
def cwd_size_in_bytes2():
  return sum(size_in_bytes(f)
             for f in files_cwd())
```

The details of how **cwd_size_in_bytes2** works is beyond the scope of an introductory book, but if you are curious about this more compact form, search the web for *python generator expressions*.

From the Library of Bill Wiecking

TIP **To save space, we've removed the doc strings for these functions. However, the supplementary code files on Google's "pythonintro" website (http://pythonintro.googlecode.com) all include doc strings.**

TIP **You can tell from the name `cwd_size_in_bytes` that the return value will be in bytes. Putting the unit of the return value in the function name prevents the need to check the documentation for the units.**

TIP **In general, it's a good idea to use lots of functions. Even single-line functions such as `list_dir()` are useful because they make your programs easier to read and maintain.**

TIP **The `os.stat()` function is fairly complex and provides much more information about files than we've shown here. Check Python's online documentation for more information (http://docs.python.org/3/library/os.html).**

From the Library of Bill Wiecking

Processing Text Files

Python makes it relatively easy to process text files. In general, file processing follows the three steps shown in A.

Reading a text file, line by line

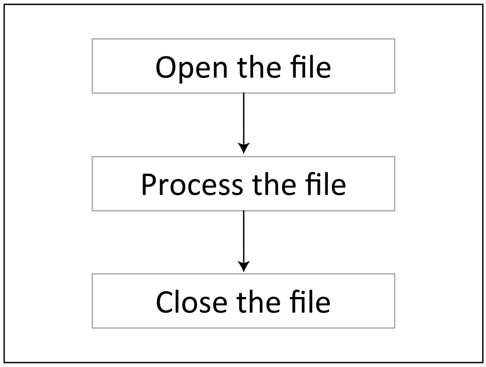
Perhaps the most common way of reading a text file is to read it one line at a time. For example, this prints the contents of a file to the screen:

```
# printfile.py
def print_file1(fname):
  f = open(fname, 'r')
  for line in f:
    print(line, end = '')
  f.close() # optional
```

The first line of the function opens the file: **`open`** requires the name of the file you want to process, and also the *mode* you want it opened in. We are only reading the file, so we open the file in read mode **`'r'`**. **Table 8.3** lists Python's main file modes.

The **`open`** function returns a special *file object*, which represents the file on disk. Importantly, **`open`** does *not* read the file into RAM. Instead, in this example, the file is read a line at a time using a for-loop.

The last line of **`print_file1`** closes the file. As the comment notes, this is optional: Python *almost always* automatically closes files for you. In this case, variable **`f`** is local to **`print_file1`**, so when **`print_file1`** ends, Python automatically closes and then deletes the file object (not the *file* itself, of course!) that **`f`** points to.



A The three main steps for processing a text file. A file must be opened before you can use it, and then it should be closed when you are done with it to ensure that all changes are committed to the file.

TABLE 8.3 Python File Modes

Character	Meaning
`'r'`	Open for reading (default)
`'w'`	Open for writing
`'a'`	Open for appending to the end of the file
`'b'`	Binary mode
`'t'`	Text mode (default)
`'+'`	Open a file for reading and writing

From the Library of Bill Wiecking

Reading by Default

When reading a text file, you can use **open** with just the filename. For example:

```
f = open(fname)
```

When no mode parameters are supplied, Python assumes you are opening a text file for reading.

TIP The `print` statement in `print_file1` sets `end = ''` because the lines of a file always end with a `\n` character. Thus if we had written just `print(line)`, the file would be displayed with extra blank lines (try it and see!).

TIP If errors occur while a file is open, it is possible that the program could end without the file being properly closed. In the next chapter, we will see how to handle such errors and ensure that a file is always correctly closed.

Reading a text file as a string

Another common way of reading a text file is to read it as one big string. For example:

```
# printfile.py
def print_file2(fname):
    f = open(fname, 'r')
    print(f.read())
    f.close()
```

This is shorter and simpler than **print_file1**, so many programmers prefer it. However, if the file you are reading is *very* large, it will take up a lot of RAM, which could slow down, or even crash, your computer.

Finally, we note that many programmers would write this as a single line:

```
print(open(fname, 'r').read())
```

While this more compact form might take some getting used to, many programmers like this style because it is both quick to type and still relatively readable.

From the Library of Bill Wiecking

Writing to a text file

Writing text files is only a little more involved than reading them. For example, this function creates a new text file named **`story.txt`**:

```
# write.py
def make_story1():
  f = open('story.txt', 'w')
  f.write('Mary had a little
  → lamb,\n')
  f.write('and then she had some
  → more.\n')
```

The **`'w'`** tells Python to open the file in *write mode*. To put text into the file, you call **`f.write`** with the string you want to put into the file. Strings are written to the file in the order in which they are given.

Important: If **`story.txt`** already exists, then calling **`open('story.txt', 'w')`** will *delete it!* If you want to avoid overwriting **`story.txt`**, you need to check to see if it exists:

```
# write.py
import os
def make_story2():
  if os.path.isfile('story.txt'):
    print('story.txt already exists)
  else:
    f = open('story.txt', 'w')
    f.write('Mary had a little
    → lamb,\n')
    f.write('and then she had some
    → more.\n')
```

Appending to a text file

One common way of adding strings to a text file is to append them to the end of the file. Unlike **`'w'`** mode, this does *not* delete anything that might already be in the file. For example:

```
def add_to_story(line, fname =
→ 'story.txt'):
  f = open(fname, 'a')
  f.write(line)
```

The important thing to note here is that the file is opened in append mode **`'a'`**.

From the Library of Bill Wiecking

Inserting a string at the start of a file

Writing a string to the *beginning* of a file is not as easy as appending one to the end because the Windows, Linux, and Macintosh operating systems don't directly support inserting text at the beginning of a text file. Perhaps the simplest way to insert text at the beginning of a file is to read the file into a string, insert the new text into the string, and then write the string back to the original file. For example:

```
def insert_title(title, fname =
→ 'story.txt'):
  f = open(fname, 'r+')
  temp = f.read()
  temp = title + '\n\n' + temp
  f.seek(0) # reset file pointer
            # to beginning
  f.write(temp)
```

First, notice that we open the file using the special mode **'r+'**, which means the file can be both read from and written to. Then we read the entire file into the string **temp** and insert the title using string concatenation.

Before writing the newly created string back into the file, we first have to tell the file object **f** to reset its internal *file pointer*. All text file objects keep track of where they are in the file, and after **f.read()** is called, the file pointer is at the very end. Calling **f.seek(0)** puts it back at the start of the file, so that when we write to **f**, it begins at the start of the file.

From the Library of Bill Wiecking

Processing Binary Files

If a file is not a text file, then it is considered to be a binary file. Binary files are opened in **'b'** mode, and you access the individual bytes of the file. For example:

```
def is_gif(fname):
  f = open(fname, 'br')
  first4 = tuple(f.read(4))
  return first4 == (0x47, 0x49,
  → 0x46, 0x38)
```

This function tests if **fname** is a GIF image file by checking to see if its first four bytes are **(0x47, 0x49, 0x46, 0x38)** (all GIFs must start with those four bytes).

In Python, numbers like **0x47** are base-16 *hexadecimal numbers*, or *hex* for short. They are very convenient for dealing with bytes, since each hexadecimal digit corresponds to a pattern of four bits, and so one byte can be described using two hex digits (such as **0x47**).

Notice that the file is opened in **'br'** mode, which means *binary reading* mode. When reading a binary file, you call **f.read(n)**, which reads the next **n** bytes. As with text files, binary file objects use a file pointer to keep track of which byte should be read next in the file.

From the Library of Bill Wiecking

Lingo Alert

The Python **pickle** module performs what is generally known as *object serialization*, or just *serialization*. The idea is to take a complex data structure and convert it to a stream of bytes—that is, create a serial representation of the data structure.

Pickling

Accessing the individual bytes of binary files is a very low-level operation that, while useful in *systems programming*, is less useful in higher-level *applications programming*.

Pickling is often a much more convenient way to deal with binary files. Python's **pickle** module lets you easily read and write almost any data structure. For example:

```
# picklefile.py
import pickle
def make_pickled_file():
  grades = {'alan' : [4, 8, 10, 10],
            'tom' : [7, 7, 7, 8],
            'dan' : [5, None, 7, 7],
            'may' : [10, 8, 10, 10]}
  outfile = open('grades.dat', 'wb')
  pickle.dump(grades, outfile)

def get_pickled_data():
  infile = open('grades.dat', 'rb')
  grades = pickle.load(infile)
  return grades
```

Essentially, pickling lets you store a data structure on disk using **pickle.dump** and then retrieve it later with **pickle.load**. This is an extremely useful feature in many application programs, so you should keep it in mind whenever you need to store binary data.

From the Library of Bill Wiecking

TIP **In addition to data structures, pickling can store functions.**

TIP **You can't use pickling to read or write binary files that have a specific format, such as GIF files. For such files, you must work byte by byte.**

TIP **Python has a module called `shelve` that provides an even higher-level way to store and retrieve data. The `shelve` module essentially lets you treat a file as if it were a dictionary. For more details, see the Python documentation (http://docs.python.org/3/library/shelve.html).**

TIP **Python also has a module named `sqlite3`, which provides an interface to the SQLite database. This lets you write SQL commands to store and retrieve data very much like using a larger database product such as Postgres or MySQL. For more details, see the Python documentation (http://docs.python.org/3/library/sqlite3.html).**

From the Library of Bill Wiecking

Reading Webpages

Python has good support for accessing the web. One common task is to have a program automatically read a webpage. This is easily done using the **urllib** module:

```
>>> import urllib.request
>>> page = urllib.request.
→ urlopen('http://www.python.org')
>>> html = resp.read()
>>> html[:25]
b'<!DOCTYPE html PUBLIC "-/'
```

Now **html** contains the complete text of the webpage at www.python.org. It is in HTML, of course, so it looks just like what you would see if you were to use the View Source option on your web browser. Since the webpage is now a string on your computer, you can use Python's string-manipulation functions to extract information from it.

TIP The urllib module also lets you programmatically post information to web forms. For details of how to do this and more, see the Python documentation (http://docs.python.org/3/howto/urllib2.html).

TIP Reading a webpage into a string is the first step in creating a web browser. The next major step is to parse the string—to identify and extract titles, paragraphs, tables, and so on. Python provides a basic HTML parsing library in the html.parser module. See the Python documentation (http://docs.python.org/3/library/html.parser.html) for details.

TIP Another nifty module is webbrowser, which lets you programmatically display a webpage in a browser. For example, when you type this into Python, the Yahoo home page should pop up in your default web browser:

```
>>> import webbrowser
>>> webbrowser.open
→ ('http://www.yahoo.com')
True
>>>
```


From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

9

Exception Handling

Exceptions are a solution to a difficult problem: How can programs deal with *unexpected errors*? For instance, what happens if a file disappears in the middle of being read because some other program on your computer has deleted it? Or what if the website your program is downloading pages from suddenly crashes?

In these and many other situations, what Python does is *raise an exception*. An exception is a special kind of error object that you can *catch* and then examine in order to determine how to handle the *error*.

Exceptions can change the flow of control of your program. Depending on when it occurs, an exception can cause the flow of control to jump out of the middle of a function or loop into another block of code that does error handling.

Often, you cannot be sure exactly which line might raise an exception, and this creates some tricky problems. Thus Python provides special exception-handling constructs for both catching exceptions and executing *clean-up code* whether or not an exception is raised.

In This Chapter

Exceptions	144
Catching Exceptions	146
Clean-Up Actions	150

From the Library of Bill Wiecking

Exceptions

An example of an exception is **IOError**, which is raised when you try to open a file that doesn't exist:

```
>>> open('unicorn.dat')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in
  → <module>
  open('unicorn.dat')
  File "C:\Python30\lib\io.py", line
  → 284, in __new__
  return open(*args, **kwargs)
  File "C:\Python30\lib\io.py", line
  → 223, in open
  closefd)
IOError: [Errno 2] No such file or
→ directory: 'unicorn.dat'
```

When an exception is raised and is not caught or handled in any way, Python immediately halts the program and outputs a *traceback*, which is a list of the functions that were called before the exception occurred. This can be quite useful in pinning down exactly what line causes an error.

The last line of the traceback indicates that an **IOError** exception has been raised, and, specifically, it means that **unicorn.dat** could not be found in the current working directory. The error message given by an **IOError** differs depending on the exact reason for the exception.

Lingo Alert

In Python, when an exception occurs, we say that it has been *raised*, or has been *thrown*. If we do nothing with a raised exception, the program usually halts immediately with a *traceback*, or a *stack trace*. However, especially in programs meant to be used by other people, we usually *catch* and *handle* exceptions, as we will see shortly.

From the Library of Bill Wiecking

Raising an exception

As we saw with the **open** function, Python's built-in functions and library functions usually raise exceptions when something unexpected happens.

For instance, dividing by zero throws an exception:

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

Syntax errors can also cause exceptions in Python:

```
>>> x := 5
SyntaxError: invalid syntax
→ (<pyshell#2>, line 1)
>>> print('hello world)
SyntaxError: EOL while scanning
→ string literal (<pyshell#3>,
→ line 1)
```

You can also intentionally raise an exception anywhere in your code using the **raise** statement. For example:

```
>>> raise IOError('This is a test!')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in
  → <module>
  raise IOError('This is a test!')
IOError: This is a test!
```

Python has numerous built-in exceptions organized into a hierarchy. See the Python documentation (http://docs.python.org/3/library/exceptions.html#bltin-exceptions) for more details.

From the Library of Bill Wiecking

Catching Exceptions

You have essentially two options for dealing with a raised exception:

1. Ignore the exception and let your program crash with a traceback. This is usually what you want when you are developing a program, since the traceback provides helpful debugging information.
2. Catch the exception and print a friendly error message, or possibly even try to fix the problem. This is almost always what you want to do with a program meant to be used by non-programmers. Regular users don't want to deal with tracebacks!

Here's an example of how to catch an exception. Suppose you want to read an integer from the user, prompting repeatedly until a valid integer is entered:

```
def get_age():
  while True:
    try:
      n = int(input('How old are
      → you? '))
      return n
    except ValueError:
      print('Please enter an integer
      → value.')
```

Inside this function's while-loop is a *try/except block*. You put whatever code you like in the **try** part of the block, with the understanding that one or more lines of that code *might* raise an exception.

What Exceptions Do Functions Raise?

How do we know to check for an exception named **ValueError** in **get_age()**? The answer depends on the function's documentation. A well-documented function will tell you what exceptions it might raise. For instance, the documentation for the **open** function (http://docs.python.org/3/library/functions.html?#open) tells you that it might raise an **IOError**. However, not all of Python's built-in functions are so forthcoming: The documentation for the **int** function says nothing about what exceptions it might raise. In this case, you have to figure out the possible exceptions by reading samples of other Python code, or by doing command-line experiments.

From the Library of Bill Wiecking

If any line of the **`try`** block does raise an exception, then the flow of control immediately jumps to the **`except`** block, skipping over any statements that have not been executed yet. In this example, the **`return`** statement will be skipped when an exception is raised.

If the **`try`** block raises no exceptions, then the **`except ValueError`** block is ignored and not executed.

So in this example, the **`int()`** function raises a **`ValueError`** if the user enters a string that is not a valid integer. When that happens, the flow of control jumps to the **`except ValueError`** block and prints the error message. When a **`ValueError`** is raised, the **`return`** statement is skipped—the flow of control jumps immediately to the **`except`** block.

If the user enters a valid integer, then no exception is raised, and Python proceeds to the following **`return`** statement, thus ending the function.

From the Library of Bill Wiecking

Try/except blocks

Try/except blocks work a little bit like if-statements. However, they are different in an important way: If-statements decide what to do based on the evaluation of Boolean expressions, whereas try/except blocks decide what to do based on whether or not an exception is raised.

A function can raise more than one kind of exception, and it can even raise the same type of exception for different reasons. Look at these three different **`int()`** exceptions (the tracebacks have been trimmed for readability):

```
>>> int('two')
ValueError: invalid literal for
➝int() with base 10: 'two'
>>> int(2, 10)
TypeError: int() can't convert non-
➝string with explicit base
>>> int('2', 1)
ValueError: int() arg 2 must be >= 2
➝and <= 36
```

So **`int()`** raises **`ValueError`** for at least two different reasons, and it raises **`TypeError`** in at least one other case.

From the Library of Bill Wiecking

Catching multiple exceptions

You can write try/except blocks to handle multiple exceptions. For example, you can group together multiple exceptions in the **except** clause:

```
def convert_to_int1(s, base):
  try:
    return int(s, base)
  except (ValueError, TypeError):
    return 'error'
```

Or, if you care about the specific exception that is thrown, you can add extra **except** clauses:

```
def convert_to_int2(s, base):
  try:
    return int(s, base)
  except ValueError:
    return 'value error'
  except TypeError:
    return 'type error'
```

Catching any exception

If you write an **except** clause without any exception name, it will catch any and all exceptions:

```
def convert_to_int3(s, base):
  try:
    return int(s, base)
  except:
    return 'error'
```

This form of **except** clause will catch any exception—it doesn't care about what kind of error has occurred, just that one has occurred. In many situations, this is all you need.

From the Library of Bill Wiecking

Clean-Up Actions

A **`finally`** code block can be added to any try/except block to perform clean-up actions. For example:

```
def invert(x):
  try:
    return 1 / x
  except ZeroDivisionError:
    return 'error'
  finally:
    print('invert(%s) done' % x)
```

The code block underneath **`finally`** will always be executed after the **`try`** block or the **`except`** block. This is quite useful when you have code that you want to perform regardless of whether an exception is raised. For instance, file close statements are often put in **`finally`** clauses so that files are guaranteed to be closed, even if an unexpected **`IOError`** occurs.

From the Library of Bill Wiecking

Alternative Formatting

The **print** statements in these two snippets of code use string interpolation to print a right-justified and zero-padded number before each line of the printed file. If you prefer string formatting, you could replace the **print** statements with this one:

```
print('{0:04} {1}'.format(num,
→line), end = '')
```

The with statement

Python's **with** statement is another way to ensure that clean-up actions (such as closing a file) are done as soon as possible, even if there is an exception. For example, consider this code, which prints a file to the screen with numbers for each line:

```
num = 1
f = open(fname)
for line in f:
    print('%04d %s' % (num, line),
    →end = '')
    num = num + 1
    # following code
```

What's unknown here is *when* the file object **f** is closed. At some point *after* the for-loop, **f** will usually be closed. But we don't know when precisely that will happen; it will remain unclosed but unused for an indeterminate amount of time, which might be a problem if other programs try to access the file.

To ensure that the file is closed as soon as it is no longer needed, use a **with** statement:

```
num = 1
with open(fname, 'r') as f:
    for line in f:
        print('%04d %s' % (num, line),
        →end = '')
        num = num + 1
```

The onscreen results are the same as the previous code, but when you use a **with** statement, the file objects' clean-up action (that is to say, closing the file) is automatically called as soon as the for-loop ends. Thus **f** does not sit around unclosed.

From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

10

Object-Oriented Programming

In this chapter, we will briefly look at *object-oriented programming*, or *OOP* for short. OOP is a methodology for organizing programs that encourages careful design and code reuse. Most modern programming languages support it, and it is has proved to be a practical way to structure and create large programs.

Essentially, an object is a collection of data, and functions that operate on that data. We've already been using objects in Python; numbers, strings, lists, dictionaries, and functions are all examples of objects.

To create new kinds of objects, you must first create a *class*. A class is essentially a blueprint for creating an object of a particular kind. The class specifies what data and functions the objects will contain, and how they relate to other classes. An object *encapsulates* both the data and functions that operate on that data.

An important OOP feature is *inheritance*: You can create new classes that inherit their data and functions from an existing class. When used properly, inheritance can save you from rewriting code, and it can also make your programs easier to understand.

In This Chapter

Writing a Class	154
Displaying Objects	156
Flexible Initialization	160
Setters and Getters	162
Inheritance	168
Polymorphism	171
Learning More	175

From the Library of Bill Wiecking

Writing a Class

Let's jump right into OOP by creating a simple class to represent a person:

```
# person.py
class Person:
  """ Class to represent a person
  """
  def __init__(self):
    self.name = ''
    self.age = 0
```

This defines a *class* named **Person**. It defines the data and functions a **Person** object will contain. We've started simple and given **Person** a **name** and an **age**. The only function so far is **__init__**, which is the standard function for initializing an object's values. As we will see, Python automatically calls **__init__** when you create a **Person** object.

A function defined inside a class is called a *method*. Just like **__init__**, methods must have **self** as their first parameter (**self** will be discussed in more detail shortly).

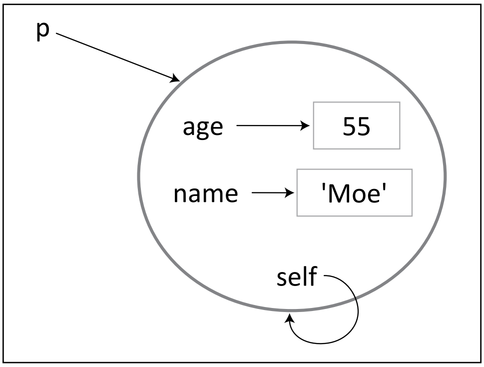
We can use **Person** objects like this:

```
>>> p = Person()
>>> p
<__main__.Person object at
→ 0x00AC3370>
>>> p.age
0
>>> p.name
''
>>> p.age = 55
>>> p.age
55
>>> p.name = 'Moe'
>>> p.name
'Moe'
```

Lingo Alert

In some OOP languages, **__init__** is called a *constructor*, because it constructs the object. A constructor is called every time a new object is created. In languages such as Java and C++, an explicit **new** keyword is used to indicate when an object is being constructed.

From the Library of Bill Wiecking



A In this example, the variable **`p`** points to a **`Person`** object (represented by the circle). As we know from looking at the **`Person`** class, a **`Person`** object contains an **`age`** and a **`name`**. These can be used just like regular variables, with the stipulation that they be accessed using dot notation—that is, **`p.age`** and **`p.name`**. The special variable **`self`** is automatically added by Python to all objects; it points to the object itself and lets functions within the class unambiguously refer to the data and functions within the object.

To create a **`Person`** object, we simply call **`Person()`**. This causes Python to run the **`__init__`** function in the **`Person`** class and to return a new object of type **`Person`**.

The **`age`** and **`name`** variables are inside an object, and every newly created **`Person`** object has its own personal copy of **`age`** and **`name`**. To access **`age`** or **`name`**, you must specify what object holds them using *dot notation*.

The self parameter

You'll notice that we don't provide any parameters for **`Person()`**, but the **`__init__(self)`** function expects an input named **`self`**. That's because in OOP, **`self`** is a variable that *refers to the object itself* **A**. This is a simple idea, but one that trips up many beginners.

TIP All classes should have an `__init__(self)` method whose job is to initialize the object—for example, initializing an object's variables. The `__init__` method is only called once when the object is created. As we will see, you can provide extra parameters to `__init__` if needed.

TIP We have followed standard Python terminology and given the first parameter of `__init__` the name `self`. This name is not required: You can use any variable name you like instead of `self`. However, the use of `self` is a universal convention in Python, and using any other name would likely just cause confusion for any programmer trying to read your code. Some other languages, such as Java and C++, use—and require—the name `this`.

TIP Objects can be used like any other data type in Python: You can pass them to functions, store them in lists and dictionaries, pickle them in files, and so on.

From the Library of Bill Wiecking

Displaying Objects

As mentioned, a method is a function defined within an object. Let's add a method to the **Person** class that prints the contents of a **Person** object:

```
# person.py
class Person:
  """ Class to represent a person
  """
  def __init__(self):
    self.name = ''
    self.age = 0
  def display(self):
    print("Person('%s', age)" %
    ➝(self.name, self.age))
```

The **display** method prints the contents of a **Person** object to the screen in a format useful to a programmer:

```
>>> p = Person()
>>> p.display()
Person('', 0)
>>> p.name = 'Bob'
>>> p.age = 25
>>> p.display()
Person('Bob', 25)
```


From the Library of Bill Wiecking

The **`display`** method works fine, but we can do better: Python provides some special methods that let you customize objects for seamless printing. For instance, the special **`__str__`** method is used to generate a string representation of an object:

```
# person.py
class Person:
  # __init__ removed for space
  def display(self):
    print("Person('%s', age)" %
    → (self.name, self.age))
  def __str__(self):
    return "Person('%s', age)" %
    → (self.name, self.age)
```

Now we can write code like this:

```
>>> p = Person()
>>> str(p)
"Person('', 0)"
```

We can use **`str`** to simplify the **`display`** method:

```
# person.py
class Person:
  # __init__ removed for space
  def display(self):
    print(str(self))
  def __str__(self):
    return "Person('%s', age)" %
    → (self.name, self.age)
```


From the Library of Bill Wiecking

You can also define a special method named **__repr__** that returns the "official" representation of an object. For example, the default representation of a **Person** is not very helpful:

```
>>> p = Person()
>>> p
<__main__.Person object at
→ 0x012C3170>
```

By adding a **__repr__** method, we can control the string that is printed here. In most objects, it is the same as the **__str__** method:

```
# person.py
class Person:
  # __init__ removed for space
  def display(self):
    print(str(self))
  def __str__(self):
    return "Person('%s', age)" %
    → (self.name, self.age)
  def __repr__(self):
    return str(self)
```

Now **Person** objects are easier to work with:

```
>>> p = Person()
>>> p
Person('', 0)
>>> str(p)
"Person('', 0)"
```


From the Library of Bill Wiecking

TIP When creating your own classes and objects, it is almost always worthwhile to write `__str__` and `__repr__` functions. They are extremely useful for displaying the contents of your objects, which is helpful when debugging your programs.

TIP If you define a `__repr__` method but not a `__str__` method, then when you call `str()` on the object, it will run `__repr__`.

TIP Once you've added the `__repr__` method, the display method for `Person` can be further simplified:

```
def display(self):
  print(self)
```

In practice, it's often not necessary to write a `display` method.

TIP The Python documentation recommends that the string representation of an object be the same as the code you would write to create that object. This is a very useful convention: It lets you easily re-create objects by cutting and pasting the string representation into the command line.

From the Library of Bill Wiecking

Flexible Initialization

If you want to create a **Person** object with a particular **name** and **age**, you must currently do this:

```
>>> p = Person()
>>> p.name = 'Moe'
>>> p.age = 55
>>> p
Person('Moe', 55)
```

A more convenient approach is to pass the name and age to **__init__** when the object is constructed. So let's rewrite **__init__** to allow for this:

```
# person.py
class Person:
  def __init__(self, name = '',
               age = 0):
    self.name = name
    self.age = age
```

Now initializing a **Person** is much simpler:

```
>>> p = Person('Moe', 55)
>>> p
Person('Moe', 55)
```


From the Library of Bill Wiecking

Since the parameters to `__init__` have default values, you can even create an "empty" `Person`:

```
>>> p = Person()
>>> p
Person('', 0)
```

Notice that inside the `__init__` method we use `self.name` and `name` (and also `self.age` and `age`). The variable `name` refers to the value passed into `__init__`, and `self.name` refers to the value stored in the object. The use of `self` helps make clear which is which.

TIP **Although it is easy to create default values for `__init__` parameters and thus allow the creation of empty `Person` objects, it is not so clear if this is a good idea from a design point of view. An empty `Person` does not have a real name or age, so you will need to check for that in code that processes `Person` objects. Constantly checking for special cases can soon become a real burden that's easy to forget about. Thus, many programmers prefer not to give the `__init__` parameters default values in cases like this.**

From the Library of Bill Wiecking

Setters and Getters

As it stands now, we can both read and write the **name** and **age** values of a **Person** object using dot notation:

```
>>> p = Person('Moe', 55)
>>> p.age
55
>>> p.name
'Moe'
>>> p.name = 'Joe'
>>> p.name
'Joe'
>>> p
Person('Joe', 55)
```

A problem with this is that we could, accidentally, set the age to be a nonsensical value, such as –45 or 509. With regular Python variables, there is no way to restrict what values they can be assigned. But within an object, we can write special *setter* and *getter* methods that give us control over how values are accessed.

First, let's add a setter method that changes **age** only if a sensible value is given:

```
  def set_age(self, age):
    if 0 < age <= 150:
      self.age = age
```


From the Library of Bill Wiecking

Decorators

Decorators are a general-purpose construct in Python used to systematically modify existing functions. They are usually placed at the beginning of a function, and start with the **@** character. We will use them in this book for this one example of creating setters and getters.

Now we can write code like this:

```
>>> p = Person('Jen', 25)
>>> p
Person('Jen', 25)
>>> p.set_age(30)
>>> p
Person('Jen', 30)
>>> p.set_age(-6)
>>> p
Person('Jen', 30)
```

A common complaint about this kind of setter is that typing **p.set_age(30)** is more cumbersome than **p.age = 30**. Property decorators solve this problem.

Property decorators

Property *decorators* combine the brevity of variables with the flexibility of functions. Decorators indicate that a function or method is special in some way, and here we use them to indicate setters and getters.

A getter returns the value of a variable, and we indicate this using the **@property** decorator:

```
@property
def age(self):
  """ Returns this person's age.
  """
  return self._age
```

This **age** method takes no parameters (other than the required **self**). We've put **@property** before it, which indicates that it's a getter function. The name of the method, **age**, will be used to set the variable.

From the Library of Bill Wiecking

We have also renamed the underlying **self.age** variable to **self._age**. Putting an underscore in front of an object variable is a common convention, and we use it here to distinguish it from the **age** method. You need to replace every occurrence of **self.age** in **Person** with **self._age**. For consistency, it is also a good idea to everywhere replace **self.name** with **self._name**. The modified **Person** class should look like this:

```
# person.py
class Person:
  def __init__(self, name = '',
               age = 0):
    self._name = name
    self._age = age

  @property
  def age(self):
    return self._age

  def set_age(self, age):
    if 0 < age <= 150:
      self._age = age

  def display(self):
    print(self)

  def __str__(self):
    return "Person('%s', %s)" %
    → (self._name, self._age)

  def __repr__(self):
    return str(self)
```


From the Library of Bill Wiecking

To create an **age** setter, we rename the **set_age** method to **age** and decorate it with **@age.setter**:

```
@age.setter
def age(self, age):
    if 0 < age <= 150:
        self._age = age
```

With these changes, we can now write code like this:

```
>>> p = Person('Lia', 33)
>>> p
Person('Lia', 33)
>>> p.age = 55
>>> p.age
55
>>> p.age = -4
>>> p.age
55
```

The setter and getters for **age** work just as if we were using the variable **age** directly. The difference is that now when you call, say, **p.age = -4**, Python is really calling the **age(self, age)** method. Similarly, when you write **p.age**, the **age(self)** method is called. Thus we get the advantage of the simple assignment syntax combined with the flexibility of controlling how variables are set and get.

From the Library of Bill Wiecking

Private variables

It's still possible to access **`self._age`** directly:

```
>>> p._age = -44
>>> p
Person('Lia', -44)
```

The problem is that **`_age`** might be modified in some way that makes the object inconsistent, and so we don't usually want to allow it.

One way to decrease the chance of this kind of problem is to rename **`self._age`** to **`self.__age`**—that is to say, to put two underscores in front of the variable name. The two underscores declare that **`age`** is a *private* variable that is not meant to be accessed by any code outside of **`Person`**. To access **`self.__age`** directly, you now have to put **`_Person`** on the front, like this:

```
>>> p._Person__age = -44
>>> p
Person('Lia', -44)
```

While this does not prevent you from modifying internal variables, it does make it almost impossible to do so accidentally.

Lingo Alert

Variables that don't begin with an underscore are called *public* variables, and any code can access them.

From the Library of Bill Wiecking

TIP **When writing large programs, a useful rule of thumb is to always make object variables private (that is, starting with two underscores) by default, and then change them to be public if you have a good reason to do so. That way, you will prevent errors caused by unintended meddling with the internals of an object.**

TIP **The syntax for creating setters and getters is strange at first, but once you get used to it, it is fairly clear. Keep in mind that you don't always need to create special setters and getters; for simple objects, like the original `Person`, regular variables may be fine.**

TIP **Some programmers prefer to avoid setters whenever possible, thus making the object immutable (just like numbers, strings, and tuples). In an object with no setters, after you create the object, there is no "official" way to change anything within it. As with other immutable objects, this can prevent many subtle errors and allow different variables to share the same object (thus saving memory). The downside, of course, is that if you do need to modify the object, your only option is to create a new object that incorporates the change.**

TIP **If the programmer tries to set the `age` to be something out of range, then `age(self, age)` doesn't make any change. An alternative approach is to purposely raise an exception, thus requiring any code that calls it to handle the exception. The advantage of raising an exception is that it might help you find more errors. Trying to set the `age` to be a nonsensical value is likely a sign of a problem elsewhere in your program.**

From the Library of Bill Wiecking

Inheritance

Inheritance is a mechanism for reusing classes. Essentially, inheritance allows you to create a brand new class by adding extra variables and methods to a copy of an existing class.

Suppose we are creating a game that has human players and computer players. Let's create a **Player** class that contains things common to all players, such as the score and a name:

```
# players.py
class Player:
  def __init__(self, name):
    self._name = name
    self._score = 0
  def reset_score(self):
    self._score = 0
  def incr_score(self):
    self._score = self._score + 1
  def get_name(self):
    return self._name
  def __str__(self):
    return "name = '%s', score = %s"
    → % (self._name, self._score)
  def __repr__(self):
    return 'Player(%s)' % str(self)
```

We can use **Player** objects this way:

```
>>> p = Player('Moe')
>>> p
Player(name = 'Moe', score = 0)
>>> p.incr_score()
>>> p
Player(name = 'Moe', score = 1)
>>> p.reset_score()
>>> p
Player(name = 'Moe', score = 0)
```


From the Library of Bill Wiecking

Lingo Alert

Many different terms are used to describe inheritance. Given that class **`Human`** inherits from class **`Player`**, we can say the following:

- **`Human`** *extends* **`Player`**.
- **`Human`** is *derived* from **`Player`**.
- **`Human`** is a *subclass* of **`Player`**, and **`Player`** is a *superclass* of **`Human`**.
- **`Human`** *isa* **`Player`**.

The last term, *isa*, implies that all humans are players. Thinking about possible isa relationships between classes is one way to create class hierarchies.

Let's assume that there are two kinds of players: humans and computers. The main difference is that humans enter their moves from the keyboard, whereas computers generate their moves from functions. Otherwise they are the same, each having a name and a score.

So let's write a **`Human`** class that represents a **`Human`** player. One way to do that would be to cut and paste a new copy of the **`Player`** class, and then add a **`make_move(self)`** method that asks the player to make a move. While that approach certainly would work, a better way is to use *inheritance*. We can define the **`Human`** class to *inherit* all the variables and methods from the **`Player`** class so that we don't have to rewrite them:

```
class Human(Player):
  pass
```

In Python, the **`pass`** statement means "Do nothing." This is a complete—and useful!—definition for the **`Human`** class. It simply inherits the code from **`Player`**, which lets us do the following:

```
>>> h = Human('Jerry')
>>> h
Player(name = 'Jerry', score = 0)
>>> h.incr_score()
>>> h
Player(name = 'Jerry', score = 1)
>>> h.reset_score()
>>> h
Player(name = 'Jerry', score = 0)
```

This is pretty impressive given that we wrote only two lines of code for the **`Human`** class!

From the Library of Bill Wiecking

Overriding methods

One small wart is that the string representation of **h** says **Player** when it would be more accurate for it to say **Human**. We can fix that by giving **Human** its own **__repr__** method:

```
class Human(Player):
  def __repr__(self):
    return 'Human(%s)' % str(self)
```

Now we get this:

```
>>> h = Human('Jerry')
>>> h
Human(name = 'Jerry', score = 0)
```

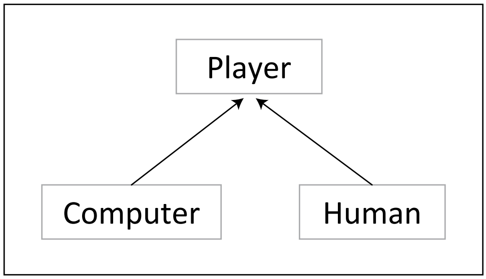
This is an example of *method overriding*: The **__repr__** method in **Human** overrides the **__repr__** method inherited from **Player**. This is a common way to customize inherited classes.

Now it's easy to write a similar **Computer** class to represent computer moves:

```
class Computer(Player):
  def __repr__(self):
    return Computer(%s)' % str(self)
```

These three classes form a small *class hierarchy*, as shown in the *class diagram* of B. The **Player** class is called the *base* class, and the other two classes are *derived*, or *extended*, classes.

Essentially, an extended class inherits the variables and methods from the base class. Any code you want to be shared by all the derived classes should be placed inside the base class.



B A class diagram showing how the **Player**, **Human**, and **Computer** classes relate. The arrows indicate inheritance, and the entire diagram is a hierarchy of classes. The more abstract (that is, general) classes appear near the top, and the more concrete (that is, specific) ones nearer the bottom.

Lingo Alert

It's also common to use the term *parent class* to refer to the base class, and *child class* to refer to the derived class.

From the Library of Bill Wiecking

Polymorphism

To demonstrate the power of OOP, let's implement a simple game called *Undercut*. In Undercut, two players simultaneously pick an integer from 1 to 10 (inclusive). If a player picks a number one less than the other player—if he undercuts the other player by 1—then he wins. Otherwise, the game is a draw. For example, if Thomas and Bonnie are playing Undercut, and they pick the numbers 9 and 10, respectively, then Thomas wins. If, instead, they choose 4 and 7, the game is a draw.

Here's a function for playing one game of Undercut:

```
def play_undercut(p1, p2):
  p1.reset_score()
  p2.reset_score()
  m1 = p1.get_move()
  m2 = p2.get_move()
  print("%s move: %s" % (p1.get_
  ➝ name(), m1))
  print("%s move: %s" % (p2.get_
  ➝ name(), m2))
  if m1 == m2 - 1:
    p1.incr_score()
    return p1, p2, '%s wins!' %
    ➝ p1.get_name()
  elif m2 == m1 - 1:
    p2.incr_score()
    return p1, p2, '%s wins!' %
    ➝ p2.get_name()
  else:
    return p1, p2, 'draw: no winner'
```

If you read this function carefully, you will note that **p1.get_move()** and **p2.get_move()** are called. We haven't yet implemented these functions because they are game-dependent. So let's do that now.

From the Library of Bill Wiecking

Implementing the move functions

Even though moves in Undercut are just numbers from 1 to 10, humans and computers *determine* their moves in very different ways. Human players enter a number from 1 to 10 at the keyboard, whereas computer players use a function to generate their moves. Thus the **`Human`** and **`Computer`** classes need their own special-purpose **`get_move(self)`** methods.

Here is a **`get_move`** method for the human (the error messages have been shortened to save space; fuller and more user-friendly messages are given in the accompanying source code on the website):

```
class Human(Player):
  def __repr__(self):
    return 'Human(%s)' % str(self)


  def get_move(self):
    while True:
      try:
        n = int(input('%s move (1 -
        ➝10): ' % self.get_name()))
        if 1 <= n <= 10:
          return n
        else:
          print('Oops!')
      except:
        print(Oops!')
```

This code asks the user to enter an integer from 1 to 10 and doesn't quit until the user does so. The **`try`**/**`except`** structure is used to catch the exception that the **`int`** function will throw if the user enters a non-integer (like "two").

From the Library of Bill Wiecking

For the computer's move, we will simply have it always return a random number from 1 to 10 (we can improve the computer strategy later if we want):

```
import random

class Computer(Player):
  def __repr__(self):
    return 'Computer(%s)' % str(self)

  def get_move(self):
    return random.randint(1, 10)
```

Playing Undercut

With all the pieces in place, we can now start playing Undercut. Let's try a game between a human and a computer:

```
>>> c = Computer('Hal Bot')
>>> h = Human('Lia')
>>> play_undercut(c, h)
Lia move (1 - 10): 7
Hal Bot move: 10
Lia move: 7
(Computer(name = 'Hal Bot',
→score = 0), Human(name = 'Lia',
→score = 0), 'draw: no winner')
```

It's important to realize that the player objects must be created *outside* of the **`play_undercut`** function. That's good design: The **`play_undercut`** function worries only about playing the game, and not about how to initialize the player objects.

The **`play_undercut`** function returns a 3-tuple of the form (*p1*, *p2*, *message*). The *p1* and *p2* values are the player objects that were initially passed in; if one player happens to win the game, then her score will have been incremented. The *message* is a string indicating who won the game or if it was a draw.

From the Library of Bill Wiecking

It's possible to pass two computer players to **play_undercut**:

```
>>> c1 = Computer('Hal Bot')
>>> c2 = Computer('MCP Bot')
>>> play_undercut(c1, c2)
Hal Bot move: 8
MCP Bot move: 7
(Computer(name = 'Hal Bot',
→score = 0), Computer(name = 'MCP
→Bot', score = 1), 'MCP Bot wins!')
```

There's no human player in this game, so the user is not asked to enter a number.

We can also pass in two human players:

```
>>> h1 = Human('Bea')
>>> h2 = Human('Dee')
>>> play_undercut(h1, h2)
Bea move (1 - 10): 5
Dee move (1 - 10): 4
Bea move: 5
Dee move: 4
(Human(name = 'Bea', score = 0),
→Human(name = 'Dee', score = 1),
→'Dee wins!')
```

Dumb Interface

While **play_undercut** works if you pass it two **Human** objects, it is not a very sensible interface: The second player will get to see the first player's move! For this to actually be fun for two humans, you would need to think of some way to keep the first player's move hidden from the second player.

These two examples, plus the earlier one of a human playing against a computer, all show the power of *polymorphism*: We've used *the same* **play_undercut** function to get very different behaviors. Instead of writing three different functions, we wrote only one and changed the objects we gave it.

In practice, this often turns out to be a big win. Although it takes experience and careful attention to design details to make polymorphism work out, it is often worth the extra time and effort.

From the Library of Bill Wiecking

Learning More

This chapter introduced a few of the essentials of OOP. Python has many more OOP features you can learn about by reading the online documentation.

Creating good object-oriented designs is a major topic. Using objects *well* is much harder than merely *using* them. One popular way of organizing object-oriented programs is to use *object-oriented design patterns*, which are proven recipes for using objects to solve common programming problems.

The most influential book on this topic is *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Once you've learned all the technical details of OOP, reading this book would be an excellent next step to learning about larger design issues.

From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

11

Case Study: Text Statistics

So far, most of the code we've seen consists of a few statements that demonstrate a feature of Python. New programmers quickly discover that it is a big step to go from these small snippets to entire programs. Bigger programs require more careful planning, and require some understanding of how best to combine individual Python features. When you first start writing larger programs, there can be a lot of trial and error.

In this chapter we will walk through the development of a larger Python program. We'll start with a description of a problem we want to solve, and then create and test a Python program that solves it.

It is difficult to show how messy writing a program can be. It will appear that we go straight from a clear problem description to a clean and simple solution. In reality, the process of writing a program is never so simple. There is a lot of trial and error, there are false starts, and you often have to backtrack to re-do things. By writing programs, you start to learn how best to combine techniques and what sorts of solutions tend to work with what sorts of problems.

In This Chapter

Problem Description	178
Keeping the Letters We Want	180
Testing the Code on a Large Data File	182
Finding the Most Frequent Words	184
Converting a String to a Frequency Dictionary	187
Putting It All Together	188
Exercises	190
The Final Program	192

From the Library of Bill Wiecking

Problem Description

When asked to write a program that solves some non-trivial problem, beginning programmers often don't know where to start. At a high level at least, the answer is simple: You start writing a big program by first *understanding* the problem you want to solve. This sounds simple, but misunderstanding what problem you are trying to solve is an extremely common programming error. Sometimes, writing a program is hard because you don't really understand what it is you want to do.

The problem we want to solve here is to calculate, and print, statistics about the contents of a text file. We want to know how many characters, lines, and words a given text file contains. In addition to the number of words, we also want to know the top ten most frequently occurring words in the file, sorted by frequency.

Let's look at an example using a short piece of text:

> A long time ago, in a galaxy far, far away ...

We can see that it contains:

- One line of text. We assume that the return-line character, **`\n`**, is used to indicate the end of a line, and that every text file (that is not empty!) is at least one line long.
- Forty-six characters, including spaces and punctuation.
- Ten words in total. However, there are only eight unique words, because *far* and *A* both occur twice.

A useful thing to do in Python is to play with examples in the interpreter. For example:

```
>>> s = 'A long time ago, in a
→galaxy far, far away ...'
>>> len(s)
46
>>> s.split()
['A', 'long', 'time', 'ago,', 'in',
→'a', 'galaxy', 'far,', 'far',
→'away', '...']
```

As you can see, the **`len`** function tells us there are 46 characters in the string. The **`split`** function divides a string into words; ignoring the **`'...'`** at the end, we can see there are ten total words in **`s`**.

Look carefully at the list of words that **`split`** returns. The word *far* occurs twice, but **`split`** treats them as the two different strings: "far," (with a comma at the end) and "far" (without a comma). Similarly, *A* and *a* are the same word, differing only in capitalization.

From the Library of Bill Wiecking

To handle these sorts of details, we will give a precise definition of what it means for a string to be a word. For us, a word will be a string that is one or more characters in length, and each character is one of the lowercase letters *a* to *z*. We will ignore non-letters (e.g., digits and punctuation), and convert uppercase letters to lowercase. So our sentence becomes this:

> Original: A long time ago, in a galaxy far, far away ...
>
> Modified: a long time ago in a galaxy far far away

Splitting the modified sentence into words now gives more accurate results:

```
>>> t = 'a long time ago in a galaxy
→ far far away'
>>> t.split()
['a', 'long', 'time', 'ago', 'in',
→ 'a', 'galaxy', 'far', 'far',
→ 'away']
>>> len(t.split())
10
```

We can count the number of unique words by converting the list to a set (recall that a set never stores duplicates):

```
>>> set(t.split())
{'a', 'ago', 'far', 'away', 'time',
→ 'long', 'in', 'galaxy'}
>>> len(set(t.split()))
8
```

There are some downsides to getting rid of non-lowercase letters. First, the number of characters will be wrong since some characters have been removed. But we can deal with this by counting the characters *before* modifying them. Second, there's no good way to remove punctuation symbols from some words. For instance, how should you handle the apostrophe in *I'd*? If you delete it (and convert the *I* to lowercase), you get *id*, which is a different word. If you replace the apostrophe with a space, then you get *I* and *d*—one word, and one non-word. To solve this problem we will treat apostrophes—and also hyphens—as "letters." Third, changing punctuation can change the meaning of words. For instance, uncapitalized versions of some names are words, such as *Polish* and *polish*, or *Bonnie* and *bonnie*. We will just ignore this particular problem, as it does not seem to be a very big one.

From the Library of Bill Wiecking

Keeping the Letters We Want

Next, let's think about how to automatically convert a string to the format we want. Converting a string to lowercase is easy:

```
>>> s = "I'd like a copy!"
>>> s.lower()
"i'd like a copy!"
```

Getting rid of characters we don't want is a bit trickier. One way to do it is to use the string **`replace`** function to replace individual characters with nothing; for example:

```
>>> s = "I'd like a copy!"
>>> s.replace('!', '')
"I'd like a copy"
```

The problem with this way of doing things is that **`replace`** needs to be called many times; that is, once for each character we *don't* want. There are many more characters that we don't want to keep than we do want to keep, so this turns out to be quite inefficient.

Another Normalize Function

A more compact way to write this function is this:

```
def normalize2(s):
  """Convert s to normalized
  → string.
  """
  return ''.join(c for c in
  → s.lower() if c in keep)
```

Many experienced programmers prefer this function because it is short and, at least for them, readable.

From the Library of Bill Wiecking

Regular Expressions

Another approach to solving this problem is to use regular expressions. For instance, you could create a regular expression defining a word, and then use the **`findall`** function to extract all the words from a given string. Since we want to illustrate basic Python programming, we won't use any regular expressions in the code that follows.

A better approach is to keep the letters we want. For example:

```
# Set of all characters to keep
keep = {'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j',
  'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't',
  'u', 'v', 'w', 'x', 'y',
  'z',
  ' ', '-', "'"}

  def normalize(s):
  """Convert s to a normalized
  → string.
  """
  result = ''
  for c in s.lower():
    if c in keep:
      result += c
  return result
```

This function loops through the string **`s`** one character at a time, appending it to the end of **`result`** only if it's in the set of characters we want to keep.

From the Library of Bill Wiecking

Testing the Code on a Large Data File

We've written only a small amount of code, but it is enough to do some useful experiments. In the examples that follow, we'll use a file called **bill.txt**. It is a 5.4 megabyte text file containing the complete works of Shakespeare (which are free on the Project Gutenberg site, www.gutenberg.org). This is a relatively large file, and so is a good test of the efficiency of our code.

One way to process a text file is to read the entire thing into memory as a string. Let's try this by hand in the interpreter:

```
>>> bill = open('bill.txt',
→ 'r').read()
>>> len(bill)
5465395
>>> bill.count('\n')
124796
>>> len(bill.split())
904087
>>> len(normalize(bill).split())
897610
```

We can see that the file has about 5.4 million characters, 125 thousand lines, and about 900 thousand words.

From the Library of Bill Wiecking

Now let's automate this by putting all the code in a function:

```
def file_stats(fname):
  """Print statistics for the given
  file.
  """
  s = open(fname, 'r').read()
  num_chars = len(s)
  num_lines = s.count('\n')
  num_words = len(normalize(s).
  → split())

  print("The file '%s' has: " %
  → fname)
  print("   %s characters" %
  → num_chars)
  print("   %s lines" % num_lines)
  print("   %s words" % num_words)
```

Calling **`file_stats`** prints this:

```
>>> file_stats('bill.txt')
The file 'bill.txt' has:
  5465395 characters
  124796 lines
  897610 words
```

On my computer, it takes about 1 second to run this program. That includes the time it takes to load the file into memory and to do all the processing. Not bad for a simple Python program!

From the Library of Bill Wiecking

Finding the Most Frequent Words

Let's consider the problem of finding the most frequently occurring words in a text file. The basic idea will be to use a dictionary whose keys are words and whose values are the counts of the words in the file.

For example, consider our original example text (in normalized form):

> *a long time ago in a galaxy far far away*

We can make a count of all the words like this:

a: 2
long: 1
time: 1
ago: 1
in: 1
galaxy: 1
far: 2
away: 1

If we convert this to a Python dictionary, it looks like this:

```
d = {
        'a': 2,
     'long': 1,
     'time': 1,
      'ago': 1,
       'in': 1,
   'galaxy': 1,
      'far': 2,
     'away': 1
   }
```


From the Library of Bill Wiecking

We can extract a lot of useful information from this dictionary:

- **`d.keys()`** is the list of all the *unique* words in the file.
- **`len(d.keys())`** is the number of *unique* words in the file.
- **`sum(d[k] for k in d)`** is the sum of all the values in **`d`**; that is, the total number of words (including duplicates) in the file. The **`sum`** function is a built-in Python function that returns the sum of a sequence.

Dictionaries do not store their data in sorted order, and so to get a list of all the words sorted from most frequent to least frequent, we'll need to convert it to a list of tuples, like this:

```
lst = []
for k in d:
  pair = (d[k], k)
  lst.append(pair)
#
# [(2, 'a'), (1, 'ago'),
#  (1, 'galaxy'), (1, 'time'),
#  (2, 'far'), ...]
lst.sort()
#
# [(1, 'ago'), (1, 'away'),
#  (1, 'galaxy'), (1, 'in'),
#  (1, 'long'), ...]
lst.reverse()
#
# [(2, 'far'), (2, 'a'),
#  (1, 'time'), (1, 'long'),
#  (1, 'in'), ...]
```


From the Library of Bill Wiecking

The for-loop converts the dictionary **d** to a list of (count, word) tuples. We do this conversion so that we can then use the list sort function to order the data by frequency. By default, the sort function orders data from smallest to biggest, and so we reverse the list to put the most frequently occurring words—the ones we are usually most interested in—at the start of the list.

With **lst** ordered from most frequent word to least frequent word, we can use slicing to access, say, the top three most frequent words on the list:

```
print(lst[:3])
#
# [(2, 'far'), (2, 'a'),
#  (1, 'time')]
```

Or, if we want neater formatting, we can do this:

```
for count, word in lst:
  print('%4s %s' % (count, word))
```

Which prints:

```
2 far
2 a
1 time
1 long
1 in
1 galaxy
1 away
1 ago
```

Notice that the word counts are preceded by three blanks each. That's because the format command **%4s** in the print statement puts the numbers right-justified in a field of length 4. As long as you have no word with 10,000 or more occurrences, this will keep the margins of the counts perfectly aligned.

From the Library of Bill Wiecking

Converting a String to a Frequency Dictionary

Now let's write a function that takes any string, **s**, and generates a dictionary whose keys are the words of **s**, and whose values are the frequency counts for the words:

```
def make_freq_dict(s):
  """Returns a dictionary whose keys
     are the words of s, and whose
     values are the counts of those
     words.
  """
  s = normalize(s)
  words = s.split()
  d = {}
  for w in words:
    if w in d:    # seen w before?
      d[w] += 1
    else:
      d[w] = 1
  return d
```

The idea of this function is to scan through each word of the string **s**, adding it to the dictionary **d** as we go. The if-statement, **if w in d**, is true if **w** is a key in **d**, and false otherwise. If **w** is a key in **d**, then that means we've seen **w** before, and so increment its frequency count by 1. But if **w** is not a key in **d**, then we add it as a new key with the statement **d[w] = 1**.

From the Library of Bill Wiecking

Putting It All Together

We now have all the pieces to make a function that automatically calculates and displays statistics for any given text file:

```
def print_file_stats(fname):
  """Print statistics for the given file.
  """
  s = open(fname, 'r').read()
  num_chars = len(s)                # count characters before normalizing s
  num_lines = s.count('\n')         # count lines before normalizing s
  d = make_freq_dict(s)
  num_words = sum(d[w] for w in d)  # count number of words in s
  # create list of (count, pair) words ordered from
  # most frequent to least frequent
  lst = [(d[w], w) for w in d]
  lst.sort()
  lst.reverse()
  # print the results to the screen
  print("The file '%s' has: " % fname)
  print("   %s characters" % num_chars)
  print("   %s lines" % num_lines)
  print("   %s words" % num_words)
  print("\nThe top 10 most frequent words are:")
  i = 1   # i is the number of the list item
  for count, word in lst[:10]:
    print('%2s. %4s %s' % (i, count, word))
    i += 1
```


From the Library of Bill Wiecking

Running this on the bill.txt file prints this:

```
The file 'bill.txt' has:
  5465395 characters
  124796 lines
  897610 words
The top 10 most frequent words are:
 1. 27568 the
 2. 26705 and
 3. 20115 i
 4. 19211 to
 5. 18263 of
 6. 14391 a
 7. 13606 you
 8. 12460 my
 9. 11107 that
10. 11001 in
```

This list of words is not unexpected, although perhaps unexciting. In English text, the most frequent words are almost always small function words, like *the* and *and*. You need to go farther down the list to find more interesting words.

This program takes less than 1.5 seconds to run on my computer (a typical desktop machine), which is pretty good for such a large file.

From the Library of Bill Wiecking

Exercises

1. Modify **`print_file_stats`** so that it also prints the total number of unique words in the file.
2. Modify **`print_file_stats`** so that it prints the average length of the words in the file.
3. A *hapax legomenon* is a word that occurs exactly once in a file. Modify **`print_file_stats`** so that it prints the total number of hapax legomena.
4. As mentioned, the ten most frequent words in bill.txt are function words, like *the* and *and*. Often, we are not interested in those words, and so we can create a set of *stop words* that contain all the words we want to ignore.

   Add a new variable called **`stop_words`** in the programming containing **`print_file_stats`**, like this:

   ```
   stop_words = {'the', 'and', 'i',
   → 'to', 'of', 'a', 'you', 'my',
   → 'that', 'in'}
   ```

   Of course, you can change the list of stop words to be anything you like. Now modify the code in your program so that the words on **`stop_list`** are not included in *any* of the statistics.

From the Library of Bill Wiecking

5. (Challenging) The **`print_file_stats`** function takes a file name as input, and then reads the *entire* file into a single string. The problem with this approach is that storing the entire file as a string uses a lot of memory if the file is big.

   An alternative approach that usually uses much less memory is to read the file a line at a time.

   Write a new function called **`print_file_stats_lines`** that does the same thing as **`print_file_stats`**, except it reads the input file line by line. The output of the two functions should be the same when they are run on the same file.

From the Library of Bill Wiecking

The Final Program

```
# wordstats.py
# Set of all allowable characters.
keep = {'a', 'b', 'c', 'd', 'e',
        'f', 'g', 'h', 'i', 'j',
        'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't',
        'u', 'v', 'w', 'x', 'y',
        'z',
        ' ', '-', "'"}

def normalize(s):
  """Convert s to a normalized string.
  """
  result = ''
  for c in s.lower():
    if c in keep:
      result += c
  return result

def make_freq_dict(s):
  """Returns a dictionary whose keys are the words of s, and whose values
  are the counts of those words.
  """
  s = normalize(s)
  words = s.split()
  d = {}
  for w in words:
    if w in d:    # add 1 to its count if w has been seen before
      d[w] += 1
    else:
      d[w] = 1    # initialize to 1 if this is the first time w has been seen
  return d
```


From the Library of Bill Wiecking

```
def print_file_stats(fname):
  """Print statistics for the given file.
  """
  s = open(fname, 'r').read()

  num_chars = len(s)                # count characters before normalizing s
  num_lines = s.count('\n')         # count lines before normalizing s

  d = make_freq_dict(s)
  num_words = sum(d[w] for w in d)  # count number of words in s

  # create list of (count, pair) words ordered from
  # most frequent to least frequent
  lst = [(d[w], w) for w in d]
  lst.sort()
  lst.reverse()

  # print the results to the screen
  print("The file '%s' has: " % fname)
  print("   %s characters" % num_chars)
  print("   %s lines" % num_lines)
  print("   %s words" % num_words)

  print("\nThe top 10 most frequent words are:")
  i = 1   # i is the number of the list item
  for count, word in lst[:10]:
    print('%2s. %4s %s' % (i, count, word))
    i += 1

def main():
  print_file_stats('bill.txt')

if __name__ == '__main__':
  main()
```


From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

A

Popular Python Packages

Part of the reason for Python's popularity is the availability of many high-quality libraries that help with various software tasks. In this appendix are descriptions of a few popular packages.

It is useful to keep in mind that many of these packages may work only with specific versions of Python (which you can always download for free from www.python.org). In particular, many packages do not yet support Python 3, so you may need to use Python 2.6 (or later) to run some of these. Fortunately, if you already know Python 3, it is not too hard to step back a version to use Python 2. Appendix B briefly discusses some of the major differences between Python 2 and Python 3.

In This Appendix

Some Popular Packages 196

From the Library of Bill Wiecking

Some Popular Packages

PIL: The Python Imaging Library

PIL (http://www.pythonware.com/products/pil/index.htm) is an image-processing library. It works with many different kinds of image formats, and can do things like crop, resize, rotate, and filter images.

Tkinter: Python GUIs

Tkinter comes with the Python library and is the standard means of accessing the popular Tk GUI tool kit. If you want to create a graphical user interface (GUI) in Python, this should be your first stop. See http://docs.python.org/3/library/tkinter.html for more information.

Django: Interactive websites

Django (www.djangoproject.com) is a framework for creating interactive websites. In this way, it is similar to Ruby on Rails, but it uses Python instead of Ruby as the underlying programming language.

Bottle: Interactive websites

Bottle (http://bottlepy.org/docs/dev/) is similar to Django in the sense that it is a framework for creating interactive websites. In contrast to Django, Bottle is a small and light framework that might be a better choice for smaller websites.

From the Library of Bill Wiecking

Pygame: 2D animation

Pygame (www.pygame.org) lets you create and control two-dimensional animations, especially for games. It provides tools for graphical animation and sound and for input devices such as joysticks. There are also introductory tutorials and sample programs at the Pygame website to help get you started.

SciPy: Scientific computing

SciPy (www.scipy.org) is a large and popular library of software tools for scientific computing (it even has its own associated conferences!). It provides mathematical software to do things such as solve optimization problems, perform numerical linear algebra calculations, process signals, and much more.

Twisted: Network programming

Twisted (http://twistedmatrix.com/trac) is a popular Python library for network programming. It supports numerous networking protocols, and includes things like web servers, mail servers, and chat clients/servers.

PyPI: The Python Package Index

The Python Package Index (http://pypi.python.org/pypi) is a frequently updated list of thousands of user-submitted Python packages. It's a good place to look for special-purpose Python libraries, or just to browse to see what uses Python has been put to.

You can easily find thousands of other Python packages by searching the web. For almost any programming task that someone has done before, you are likely to find a Python library!

From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

B

Comparing Python 2 and Python 3

Python 3 was released at the end of 2008 and marked a major update to Python. Some of the changes introduced in Python 3 are not backward-compatible with Python 2, and so development of Python 2 has continued in parallel with the newer Python 3.

In this chapter, we'll summarize some of the main changes to Python 3 and also explain how you can convert a Python 2 program to Python 3.

In This Appendix

What's New in Python 3 200

From the Library of Bill Wiecking

What's New in Python 3

Python 3 introduced many new features; the following are some of the most visible:

- Python 3's **`print`** function is indeed a function. In Python 2, **`print`** was a language construct, similar to **`if`** and **`for`**. The problem with Python 2's **`print`** was that it was difficult to modify—for example, changing **`print`** statements to print to a file instead of to the console is much easier in Python 3 because you can just reassign the **`print`** function.
- Dividing integers in Python 3 works as you would expect when fractions are involved:

  ```
  Python3>>> 1 / 2
  0.5
  ```

  However, Python 2 chops off all digits after the decimal when dividing integers:

  ```
  Python2>>> 1 / 2
  0
  ```

  While Python 2's way of dividing integers appears in other programming languages, many programmers find it counterintuitive and the cause of subtle errors.
- Python 2 has two kinds of classes: *old-style classes* and *new-style classes*. Python 3 drops old-style classes completely.
- Python 3 renames a couple of important functions: The **`input`** and **`range`** functions are called **`raw_input`** and **`xrange`** in Python 2.
- The format strings described in Chapter 9 exist only in Python 3, and not Python 2. Python 2 only has string interpolation with the **`%`** operator.

From the Library of Bill Wiecking

Many other technical changes were made in Python 3. For a complete list of differences, see "What's New in Python 3.0" (http://docs.python.org/3/whatsnew/3.0.html).

It is often not too difficult to convert a Python 2 program into Python 3. A useful tool that helps with this process is **2to3** (http://docs.python.org/3/library/2to3.html). It can automatically convert almost all Python 2 programs into equivalent Python 3 programs.

Which version of Python should you use?

When deciding what version of Python to use—2 or 3—there are a few things to take into consideration:

- If you must work with programs that are written in Python 2, then you should probably use Python 2. Otherwise, you would need to convert all the existing Python 2 programs into Python 3, which might be difficult.
- Some special-purpose libraries may only work with one version of Python, and so if you need to use one of those, your choice of Python may be constrained.
- If you are just starting out as a programmer and have no old Python programs to maintain or special-purpose libraries that you must use, then Python 3 is probably the best choice.

From the Library of Bill Wiecking

This page intentionally left blank

From the Library of Bill Wiecking

Index

Numbers

2to3 tool, using for Python conversions, 201
5 vs. 5.0, 13

Symbols

'+ file module, meaning of, 134, 137
== operator, 44
+ (addition) operator, 12
= (assignment) operator, example of, 24
**** (backward slash)
  using with pathnames, 130
  writing, 130
) (closed round bracket), using with tuples, 29
% conversion specifier, meaning of, 125
@ (decorators), using, 163
/ (division) operator, 12
// (division) operator, 11
" (double quote), using with strings, 17
__ (double underscore), use of, 20–21
'' and **""** (empty strings), using, 18
\" escape character, 88
**** escape character, 88
\' escape character, 88
****** (exponentiation) operator, 12
// (integer division) operator, 11–12
% (**mod**) function, using with strings, 84
***** (multiplication) operator, 12
# (number sign), using with comments, 36
((open round bracket), using with tuples, 29
% (remainder) operator, 12
() (round brackets)
  using with functions, 68
  using with regular expressions, 99
  using with tuples, 104
>>> (shell prompt), 10
' (single quote), using with strings, 17
[] (square brackets)
  using with lists, 108
  using with strings, 84
- (subtraction) operator, 12

A

'a file module, meaning of, 134
addition (**+**) operator, 12
aggregate data structures, strings as, 83
and operator, 44–45
append function, using with lists, 110–111
append mode, using with text files, 134, 136
area function
  calling, 70–71
  parts of, 71
  return statement, 72
arithmetic operators. *See also* floating point arithmetic; integer arithmetic; math functions
  addition (**+**), 12
  division (**/**), 12
  exponentiation (******), 12
  integer division (**//**), 11–12
  multiplication (*****), 12
  remainder (**%**), 12
  subtraction (**-**), 12
ASCII (American Standard Code for Information Interchange), 87
assignment (**=**) operator, example of, 24
assignment statements
  diagrams, 28
  example of, 24
  initialization statement, 26
  labeling values, 28
  left-hand side, 26
  multiple, 29
  operator, 26
  right-hand side, 26
associative arrays. *See* dictionaries

From the Library of Bill Wiecking

B

`'b` file module, meaning of, 134, 138
backward slash (`\`)
  using with pathnames, 130
  writing, 130
base class, explained, 169–170
bill.txt file, using, 182–183, 189
bin built-in function, printing doc string for, 21
binary files, processing, 138–140
binary mode, indicating, 134
binary vs. text files, 128–129
blocks of code
  breaking out of, 64–65
  indenting, 51–53
  indicating, 51
Boolean logic
  == operator, 44
  and operator, 44–45, 48
  with brackets (**()**), 46
  definition of operators, 48
  evaluating expressions, 46–47
  explained, 44
  False values, 44
  logical equivalence, 45
  logical negation, 44
  logical operators, 44
  not operator, 44–46
  operator priority, 47
  or operator, 44–45
  or operator, 48
  short-circuit evaluation, 48
  True values, 44
  truth table, 44
  truth values, 44
  without brackets (**()**), 47
Bottle framework, 196
brackets, preceding word counts with, 186
break statement, using, 64–65

C

calculating
  area of circle, 70
  factorials, 59–60
  powers, 68
case sensitivity, explained, 25
case study. *See* text statistics case study
case-changing functions
  s.capitalize(), 94
  s.lower(), 94
  s.swapcase(), 94
  s.title(), 94
  s.upper(), 94
catching exceptions, 146–149
ceil(x) function, 16
character codes, finding, 87
character length, determining, 178
characters
  accessing with for-loop, 86
  escape, 88
  getting rid of unwanted, 180–181
  whitespace, 88
Cheetah templating package, 126
child class, explained, 170
chr function, using, 87
circle, calculating area of, 70
class diagram, example of, 170
class hierarchy, example of, 170
classes
  defined, 153
  deriving, 169–170
  extending, 169–170
  and methods, 154
  and objects, 153
  reusing, 168–170
  self parameter, 155
  subclasses of, 169–170
  writing, 154–155
clean-up actions
  finally code block, 150
  with statement, 151
closed round bracket (**)**), using with tuples, 29
code blocks
  breaking out of, 64–65
  indenting, 51–53
  indicating, 51
command line
  calling Python from, 33–34
  environment variables, 34
  path variable, 34
  running programs from, 33
command shell
  interacting with, 10
  shell prompt, 10

From the Library of Bill Wiecking

command window, opening, 34
comments
  defined, 36
  using, 41–42
compiled code. *See* object code
compiling source code, 35
complex numbers, 15
concatenating
  strings, 19
  tuples, 107
conditional expressions, 53
constructors, explained, 154
`continue` statement, using, 64–65
conversion specifiers
  `%` character, 125
  base 8 value, 125
  base 16, 125
  float, 125
  integers, 125
  lowercase float exponential, 125
  lowercase hexadecimal, 125
  octal value, 125
  string, 125
  uppercase hexadecimal, 125
  uppercase float exponential, 125
converting
  floats to integers, 23
  floats to strings, 22
  integers to floats, 22
  integers to strings, 22
  strings to floats, 22
  strings to numbers, 23
`cost(x)` function, 16
`count` function, using with lists, 110
current working directory
  `cwd_size_in_bytes` function, 132
  explained, 130

D

`d` conversion specifier, meaning of, 125
data structures
  defined, 101
  dictionaries, 118–121
  list comprehensions, 115–117
  list functions, 110–112
  lists, 108–109
  reading, 139
  self-referential, 109
  sequences, 103
  sets, 122
  sorting lists, 113–114
  tuples, 104–107
  `type` command, 102
  writing, 139
data types
  checking with **`type`** command, 102
  converting between, 22–23
  converting numeric types, 22
  explained, 9
  floats to strings, 22
  implicit conversions, 22–23
  integers to floats, 22
  integers to strings, 22
  strings, 9
  strings to floats, 22
decorators (**`@`**), using 163
`degrees(x)` function, 16
derived class, explained, 169–170
dictionaries
  converting to, 184, 187
  converting to list of tuples, 185–186
  defined, 118
  extracting information from, 185
  key restrictions, 119
  and sets, 122
  unique keys, 119
dictionary functions
  `d.clear()`, 120
  `d.copy()`, 120
  `d.fromkeys()`, 120
  `d.get(key)`, 120
  `d.items()`, 120–121
  `d.keys()`, 120–121
  `d.popitem()`, 120–121
  `d.pop(key)`, 120
  `d.setdefault()`, 120
  `d.update()`, 120
  `d.values()`, 120–121
`dir ('')` command, entering, 37
`dir` function, using, 92
directory
  current working, 130–132
  default, 130
`dir(m)` function, using, 20

From the Library of Bill Wiecking

display method, using, 157, 159
division (**//** and **/**) operators, 11–12
Django framework, 196
documentation strings
  accessing for functions, 71
  benefits, 71
  formatting convention, 71
  printing, 21
documentation website, accessing, 133
dot notation, using with objects, 155
double quote ("), using with strings, 17
double underscore (__), use of, 20–21

E

e conversion specifier, meaning of, 125
E conversion specifier, meaning of, 125
Easter egg example, 82
eat_vowels example, 117
editor window, opening in IDLE, 32
elif (else if) statements, 52
else statements, 49–50
empty lists, denoting, 108
empty strings (**''** and **""**), using, 18, 39
ending lines of text, 88
environment variables, 34
errors, handling, 143
escape characters
  \", 88
  \', 88
  ****, 88
  \n, 88
  \r, 88
  \t, 88
exceptions
  built-in, 145
  catching, 146–149
  checking for, 146
  defined, 143
  IOError, 143
  outputting tracebacks, 144
  raising, 143–145
  syntax errors, 145
  throwing, 144
executable code. *See* object code
exponentiation (******) operator, 12
exp(x) function, 16

F

F conversion specifier, meaning of, 125
factorials, calculating, 59–60
factorial(x) function, 16
file modules, 134
file_stats, calling, 183
files
  examining, 131–133
  functions, 131
  reading, 128–130
  text vs. binary, 128–129
  writing, 128–130
finally code block, adding, 150
find function vs. **index**, 93
float, conversion specifier for, 125
float literals, 13
floating point arithmetic. *See also* arithmetic operators
  5 vs. 5.0, 13
  complex numbers, 15
  decimal points, 13
  errors, 15
  examples, 13
  limited precision, 14–15
  overflow, 14
  scientific notation, 13
  silent errors, 14
  truncating, 61
floats
  converting integers to, 22
  converting strings to, 22
  converting to integers, 23
  converting to strings, 22
float(s) conversions, making, 23
flow of control
  backing out of blocks, 64–65
  backing out of loops, 64–65
  Boolean logic, 44–48
  code blocks, 51–53
  explained, 43
  for-loops vs. while-loops, 59–63
  if-statements, 49–50
  indentation, 51–53
  loops, 54–58
  nested loops, 66

From the Library of Bill Wiecking

folders
  backward slash (\), 130
  functions, 131
  pathnames, 130
  structure, 130
for-loops
  accessing characters with, 86
  changing starting value of, 54
  headers, 54
  i (index) variable, 54, 63
  printing numbers, 55
  using iterators with, 55
  vs. while-loops, 58–63
format function
  using, 94
  using with strings, 124
format strings
  named replacement, 126
  using, 126–127
  using curly braces (**{}**), 127
formatting functions for strings. *See* string-formatting functions
formatting parameters, specifying, 127
f.read(), calling, 137–138
frequency dictionaries, converting strings to, 187
f.seek(), calling, 137
function names, reassigning, 69
function parameters
  default values, 78
  keyword parameters, 79
  pass by reference, 76–77
  pass by value, 76
  state of memory, 76
functional programming style, 72
functions. *See also* string functions; tuple functions
  accessing doc strings for, 71
  append, 110–111
  availability to strings, 37
  as black boxes, 68
  calculating powers, 68
  calling, 68–69
  chr, 87
  count, 110
  defined, 67
  defining, 70–72
  files and folders, 131
  listing built-in, 21
  listing in modules, 20
  main(), 75
  vs. methods, 154
  modules, 80–81
  naming, 70
  not returning values, 69
  ord, 87
  side effects, 72
  using, 133
  using round brackets (**()**) with, 68
  variable scope, 73–74

G

generator expressions
  explained, 117
  searching for, 132
getters and setters
  avoiding setters, 167
  decorators, 163
  name and **age** values, 162
  private variables, 166
  property decorators, 163–165
  syntax, 167
  using, 162–167
global variables, explained, 74

H

hapax legomenon, explained, 190
hash tables. *See* dictionaries
hashing, using with dictionaries, 118
help
  documentation, 21
  listing functions in modules, 20
  utility, 21
help(f) function, using, 21
hexadecimal numbers, explained, 138
Human class, writing, 169

I

i (index) variable, use of, 54, 63
identifiers, explained, 24
IDLE (integrated development environment), 4

From the Library of Bill Wiecking

IDLE editor
  alternatives, 33
  starting screen, 6
  using, 32–34
IDLE shortcuts
  opening editor window, 32
  opening files for editing, 32
  redoing last undo, 32
  running programs, 32
  saving programs, 32
  undoing actions in IDLE, 32
if/elif-statements, 52
if/else-statements, 49–50
if-statements
  explained, 49
  flow chart, 50
  headers, 50
  structure, 50
immutable objects, 167
importing
  modules, 16, 81
  this module at command line, 82
indenting code blocks, 51–53
index function vs. **find**, 93
indexing
  beginning at 0, 84
  negative, 85, 91
  strings, 84–86
  using **%** (**mod**) function for, 84
infinite loops, 58
inheritance
  defined, 153, 168–169
  Human class, 169
  isa terminology, 169
  overriding methods, 170
  Player class, 168–169
__init__ function, using, 155, 160–161
initialization, flexibility of, 160–161
initialization statement, explained, 26
input built-in function
  explained, 36–37
  using, 123
installing Python
  on Linux systems, 7
  on Macs, 7
  on Windows systems, 6
int function, documentation for, 146
integer arithmetic. *See also* arithmetic operators; math functions
  defined, 11
  division, 11
  operators, 12
  order of evaluation, 12
  unlimited size, 12
integer division (**//**) operator, 11–12
integers
  conversion specifier, 125
  converting floats to, 23
  converting to floats, 22
  converting to strings, 22
  lack of maximum, 60
interactive command shell, 10
interpreter, playing with examples in, 178
int(s) conversions, making, 23
I/O (input and output)
  console, 123
  examining files, 131–133
  examining folders, 131–133
  explained, 123
  formatting strings, 124–125
  processing binary files, 138–140
  processing text files, 134–137
  reading files, 128–130
  reading webpages, 141
  string formatting, 126–127
  writing files, 128–130
IOError, raising, 143
isa terminology, using with inheritance, 169
iterators, using with for-loops, 55

J

join function
  using, 97
  using with list comprehensions, 117

K

keywords
  restriction for variables, 25
  using, 79

L

len function, using with characters, 178
letters, keeping desired, 180–181
lexicographical ordering, 113

From the Library of Bill Wiecking

lines of text, ending, 88
Linux, installing Python on, 7
list comprehensions
  examples, 116
  explained, 115
  filtering, 117
  generator expressions, 117
list functions
  mutating, 110
  s.append(), 110–111
  s.count(), 110
  s.extend(), 110
  s.index(), 110
  s.insert(), 110
  s.pop(), 110
  s.remove(), 110, 112
  s.reverse(), 110, 112
  s.sort(), 110
lists. *See also* tuples
  [] (square brackets), 108
  containing elements vs. pointing, 109
  empty, 108
  lexicographical ordering, 113
  mutability, 109
  pointing to values, 109
  pop and push, 111–112
  self-referential data structure, 109
  sorting, 113–114
  using, 108
local variable, explained, 73
log(x) functions, 16
loops
  breaking out of, 64–65
  infinite loops, 58
  for-loops, 54–55
  nesting, 66
  while-loops, 56–58
lowercase float exponential, conversion specifier for, 125
lowercase hexadecimal, conversion specifier for, 125

M

^M character, handling, 88
Macs, installing Python on, 7
main() function, using, 75
maps. *See* dictionaries
math functions. *See also* arithmetic operators; integer arithmetic
  importing modules, 16
  return values, 16
math module
  ceil(x) function, 16
  cost(x) function, 16
  degrees(x) function, 16
  exp(x) function, 16
  factorial(x) function, 16
  log(x) functions, 16
  pow(x) function, 16
  radians(x) function, 16
  sin(x) function, 16
  sqrt(x) function, 16
  tan(x) function, 16
  using, 16
methods
  vs. functions, 154
  overriding, 170
mod (%) function, using with strings, 84
modules
  creating, 80
  importing, 16, 81
  listing functions in, 20
  namespaces, 82
  pickle, 140
  shelve, 140
  sqlite3, 140
  urllib, 141
  using, 81
  webbrowser, 141
move functions, implementing for Undercut game, 172–173
multiplication (*) operator, 12

N

\n (newline) character, explained, 39, 88
n! notation, using, 60
name clashes, preventing, 82
namespaces
  explained, 82
  preventing name clashes, 82
negative indexing, 85, 91
nested loops
  break statement, 66
  continue statement, 66
  using, 66

From the Library of Bill Wiecking

`new` keyword, using with constructors, 154
newline (`\n`) character, explained, 39
`None` value, using with functions, 72
`normalize()` function, using, 180
`not` operator, 44–46
number sign (`#`), using with comments, 41
numbers
  converting strings to, 23
  floating point, 38
  immutable quality, 28
  integers, 38
  reading from keyboard, 38
  as strings, 38
  summing, 62
  summing from users, 61
  types of, 38

O

`o` conversion specifier, meaning of, 125
object code
  converting source code to, 5
  explained, 35
object serialization, explained, 139
objects
  and classes, 153
  creating, 159
  defined, 153
  displaying, 156–159
  dot notation, 155
  immutable, 167
  string representation of, 159
  using, 155
octal values, conversion specifier for, 125
OOP (object-oriented programming), 2
  classes, 153–155
  constructors, 154
  explained, 153
  getters, 162–167
  inheritance, 168–170
  initialization, 160–161
  objects, 156–159
  polymorphism, 171–174
  setters, 162–167
`open` function
  documentation, 146
  using, 135
open round bracket (`(`), using with tuples, 29
operators. *See* arithmetic operators; assignment operator
`or` operator, 44–45
`ord` function, using with character codes, 87
order of evaluation, 12
ordered sequences, 103
`os.chdir()` function, 131
`os.getcwd()` function, 131
`os.listdir()` function, 131
`os.path.isdir()` function, 131
`os.path.isfile()` function, 131
`os.stat()` function, 131, 133
overflow errors, 14

P

packages
  Bottle, 196
  Django, 196
  PIL (Python Imaging Library), 196
  Pygame, 197
  PyPI (Python Package Index), 197
  SciPy, 197
  Tkinter, 196
  Twisted, 197
parent class, explained, 170
`partition` function, using, 95
pass by reference, explained, 76
pass by value, explained, 76
path variable, 34
pathnames, using with folders, 130
`Person` class
  adding `method` to, 156
  creating, 154
`Person` objects
  with `name` and `age`, 160–161
  working with, 158
pi calculation, doing, 70
`pickle` module
  restriction, 140
  using, 139
PIL (Python Imaging Library) package, 196
`play_undercut` function, analyzing, 174
`Player` class, creating, 168
polymorphism
  defined, 153
  power of, 174
  Undercut game, 171–174
`pop`, using on lists, 111–112

From the Library of Bill Wiecking

powers, calculating, 68
pow(x) function, 16
print statement
  using, 39–40, 135
  using string interpolation with, 151
printing
  documentation strings, 21
  numbers in for-loops, 55
  strings on screen, 39–40
private variables, 166–167
problems, understanding, 178
programming
  process, 4–5
  requirements, 4
  source code, 5
programming problems, understanding, 178
programs
  checking output, 5
  defined, 31
  flow of execution, 43
  managing variables, 167
  running, 5
  running from command line, 33
  running with IDLE, 32
  storing, 32
  straight-line, 43
  structuring, 42
  tracing, 36–37
  writing in IDLE, 32
property decorators, using, 163–165
public variables, 166
push, using on lists, 111
.py files
  versus **.pyc** files, 35
  contents of, 5
  listing, 132
  running, 35
.pyc files
  contents, 35
  explained, 4
Pygame 2D animation package, 197
PyPI (Python Package Index) package, 197
Python 2
  classes, 200
  converting into Python 3, 201
  dividing integers, 200
  vs. Python 3, 40, 200–201
  raw_input function, 200
  string interpolation, 200
  xrange function, 200
Python 3
  dividing integers, 200
  format strings, 200
  input function, 200
  print function, 200
  range function, 200
Python components
  compiler, 35
  interpreter, 35
  virtual machine, 35
Python language
  calling from command line, 33–34
  design, 2
  download page, 6
  education, 3
  installing on Linux, 7
  installing on Macs, 7
  installing on Windows, 6
  libraries, 2
  maintainability, 2
  origin of name, 2
  scientific computing, 3
  scripts, 3
  text processing, 3
  uses, 3
  website development, 3
Python packages
  Bottle, 196
  Django, 196
  PIL (Python Imaging Library), 196
  Pygame, 197
  PyPI (Python Package Index), 197
  SciPy, 197
  Tkinter, 196
  Twisted, 197
pythonintro website, accessing, 133

Q

quotes (**'** and **"**), using with strings, 17
quotes, triple, 17

R

'r' file module, meaning of, 134
\r escape character, 88
radians(x) function, 16

From the Library of Bill Wiecking

re module, accessing documentation for, 100
reading
    files, 128–130
    text files as strings, 135
    webpages, 141
regular expressions
    examples, 98–99
    matching with, 99
    operators, 98
    using, 181
    using round brackets (**()**) with, 99
    x* operator, 98
    x|y operator, 98
    x+ operator, 98
    xy? operator, 98
remainder (**%**) operator, 12
remove function, using with lists, 112
replace function, using with strings, 96, 180
__repr__ method, using, 158–159
return statement, using with **area** function, 72
return values, using, 16
reverse function, using with lists, 112
round brackets (**()**)
    using with functions, 68
    using with regular expressions, 99
    using with tuples, 104
rpartition function, using, 95

S

s conversion specifier, meaning of, 125
saving programs with IDLE, 32. *See also* IDLE (integrated development environment)
scientific notation, using, 13
SciPy scientific computing package, 197
scope. *See* variable scope
scripts. *See* programs
searching functions for strings. *See* string-searching functions
self parameter, using with classes, 155
sentences, splitting into words, 179
sequence types
    lists, 103
    strings, 103
    tuples, 103–107
sequences. *See also* values
    defined, 103
    ordered, 103
    size restriction, 103
serialization, explained, 139
sessions. *See* shell transcripts
sets
    calling **dir(set)**, 122
    and dictionaries, 122
    explained, 122
    immutable frozensets, 122
    mutable, 122
    online documentation, 122
setters and getters
    avoiding setters, 167
    decorators, 163
    name and **age** values, 162
    private variables, 166
    property decorators, 163–165
    syntax, 167
    using, 162–167
shell prompt (**>>>**), 10
shell transcript, explained, 10
shelve module, explained, 140
side effects, relationship to functions, 72
sin(x) function, 16
single quote (**'**), using with strings, 17
slicing strings
    explained, 89
    with negative indexes, 91
    shortcuts, 90–91
software. *See* object code
sort function, using with lists, 114
sorting
    lists, 113–114
    tuples, 114
source code
    comments, 36, 41–42
    compiling, 35
    converting to object code, 5
    writing, 5
split function, using, 96, 178–179
splitting functions for strings. *See* string-splitting functions
sqlite3 module, explained, 140
sqrt(x) function, 16
square brackets (**[]**)
    using with lists, 108
    using with strings, 84
standard error (stderr), explained, 39

From the Library of Bill Wiecking

standard input (stdin), explained, 39
standard output (stdout), explained, 39
stop words, creating set of, 190
string functions. *See also* functions
  case-changing, 94
  for contents of substrings, 92
  `s.count()`, 97
  for searching, 93
  `s.encode()`, 97
  `s.endswith()`, 92
  `s.find()`, 93
  `s.index()`, 93
  `s.isalnum()`, 92
  `s.isalpha()`, 92
  `s.isdecimal()`, 92
  `s.isdigit()`, 92
  `s.isidentifier()`, 92
  `s.islower()`, 92
  `s.isnumeric()`, 92
  `s.isprintable()`, 92
  `s.isspace()`, 92
  `s.istitle()`, 92
  `s.isupper()`, 92
  `s.join()`, 97
  `s.maketrans()`, 97
  `split`, 95–96
  `s.rfind()`, 93
  `s.rindex()`, 93
  `s.startswith()`, 92
  `s.translate()`, 97
  for stripping, 95–96
  `s.zfill()`, 97
  for testing, 92
string interpolation, 124, 151
string literals, writing, 17
string-formatting functions
  `s.center()`, 94
  `s.format()`, 94
  `s.ljust()`, 94
  `s.rjust()`, 94
string-replacement functions
  `s.expandtabs()`, 96
  `s.replace()`, 96
strings
  as aggregate data structures, 83
  characters, 86–88
  concatenating, 19
  conversion specifiers, 125
  converting floats to, 22
  converting integers to, 22
  converting to floats, 22
  converting to formats, 180–181
  converting to frequency dictionaries, 187
  converting to numbers, 23
  creating, 19
  defined, 17
  empty, 18
  escape characters, 88
  extracting substrings from, 89
  formatting, 124–127
  immutable quality, 28
  indexing, 84–86
  indicating, 17
  inserting at start of files, 137
  lengths, 18
  number of characters in, 18
  printing on screen, 39–40
  reading from keyboard, 36–38
  regular expressions, 98–100
  representations of objects, 159
  returning list of, 131
  slicing, 89–91
  splitting, 178–179
  square brackets (**`[]`**) for indexing, 84
  uses of, 9
  using quotes (**`'`** and **`"`**) with, 17
  using **`strip()`** function with, 37
  as words, 179
string-searching functions
  `s.find()`, 93
  `s.index()`, 93
  `s.rfind()`, 93
  `s.rindex()`, 93
string-splitting functions
  `s.partition()`, 95
  `s.rpartition()`, 95
  `s.rsplit()`, 95
  `s.split()`, 95
  `s.splitlines()`, 95
string-stripping functions
  `s.lstrip()`, 95
  `s.rstrip()`, 95
  `s.strip()`, 95

From the Library of Bill Wiecking

string-testing functions
for contents of substrings, 92
s.endswith(), 92
s.isalnum(), 92
s.isalpha(), 92
s.isdecimal(), 92
s.isdigit(), 92
s.isidentifier(), 92
s.islower(), 92
s.isnumeric(), 92
s.isprintable(), 92
s.isspace(), 92
s.istitle(), 92
s.isupper(), 92
s.startswith(), 92
strip() function, using with strings, 37
subclasses, using with classes, 169–170
substrings, extracting from strings, 89
subtraction (**-**) operator, 12
summing
numbers, 62
numbers from users, 61
syntax errors, causing, 145

T

't file module, meaning of, 134, 137
tan(x) function, 16
templating packages, using, 126
testing functions. *See* Boolean logic; string-testing functions
text files
appending to, 136
closing, 134
opening, 134
processing, 134–137
reading as strings, 135
reading line by line, 134–137
writing to, 136
text mode, indicating, 134
text statistics case study
completing, 188–189
converting strings to formats, 180–181
final program, 192–193
finding frequent words, 184–186
normalize() function, 180–181
problem description, 178–179
regular expressions, 181
strings to frequency dictionary, 187
testing code on data file, 182–183
text vs. binary files, 128–129
this module, importing at command line, 82
Tkinter package, 196
tracebacks, outputting, 144
tracing programs, 36–37
transcripts, explained, 10
True values, returning for paths, 131
try/except blocks
adding **finally** code block to, 150
examples of, 146–148
in Undercut game, 172
tuple functions. *See also* functions
len(), 106
tup.count(), 106
tup.index(), 106
x in tup, 106
tuples. *See also* lists
concatenating, 107
creating list of, 185–186
defined, 103
example of, 95
immutability, 105
round brackets (**()**), 104
singleton, 104
sorting, 114
trailing commas, 104
writing values as, 29
Twisted network programming package, 197
type command, using, 102
types. *See* data types

U

Undercut game
implementing, 171–174
move functions, 172–173
playing, 173–174
try/except blocks, 172
Unicode, rise of, 87
uppercase float exponential, conversion specifier for, 125
uppercase hexadecimal, conversion specifier for, 125
urllib module, using, 141

From the Library of Bill Wiecking

V

ValueError example, 146–147
values. *See also* sequences
  assigning in parallel, 30
  assigning to variables, 27
  displaying multiple, 29
  referring variables to, 28
  replacing by position, 126
  and variables, 24–25
  writing as tuples, 29
variable names
  case sensitivity, 25
  first character, 25
  keywords, 25
  lengths, 25
  rules for, 25
variable scope
  explained, 73
  global variables, 74
  local variables, 73
variable values, swapping, 30
variables
  adding multiple, 29
  assigned values, 27
  assigning values to, 27
  explained, 9
  pointing to values, 27
  private vs. public, 166–167
  referring to values, 28
  terminology, 27
  and values, 24–25
virtual machine, explained, 35
von Rossum, Guido, 2

W

'w file module, meaning of, 134
web browsers, creating, 141
webbrowser module, explained, 141
webpages, reading, 141
websites
  2to3 conversion for Python, 201
  Bottle, 196
  Django, 196
  online documentation, 133
  PIL (Python Imaging Library), 196
  Pygame, 197
  PyPI (Python Package Index), 197
  Python download page, 6
  pythonintro, 133
  re module documentation, 100
  SciPy, 197
  templating packages, 126
  Tkinter, 196
  Twisted, 197
  Unicode home page, 87
while-loops
  flexibility of, 58
  flow of control, 56
  vs. for-loops, 58–63
  form of, 57
  incrementers, 57
  initializers, 57
  sample program, 56
  try/except block in, 146
whitespace characters, handling, 88
Windows, installing Python on, 6
with statement, using, 151
word counts, preceding with brackets, 186
words
  creating set of stop words, 190
  finding frequent, 184–186
  getting sorted list of, 185–186
  splitting sentences into, 179
  strings as, 179
writing
  data structures, 139
  files, 128–130
  opening text files for, 134
  to text files, 136

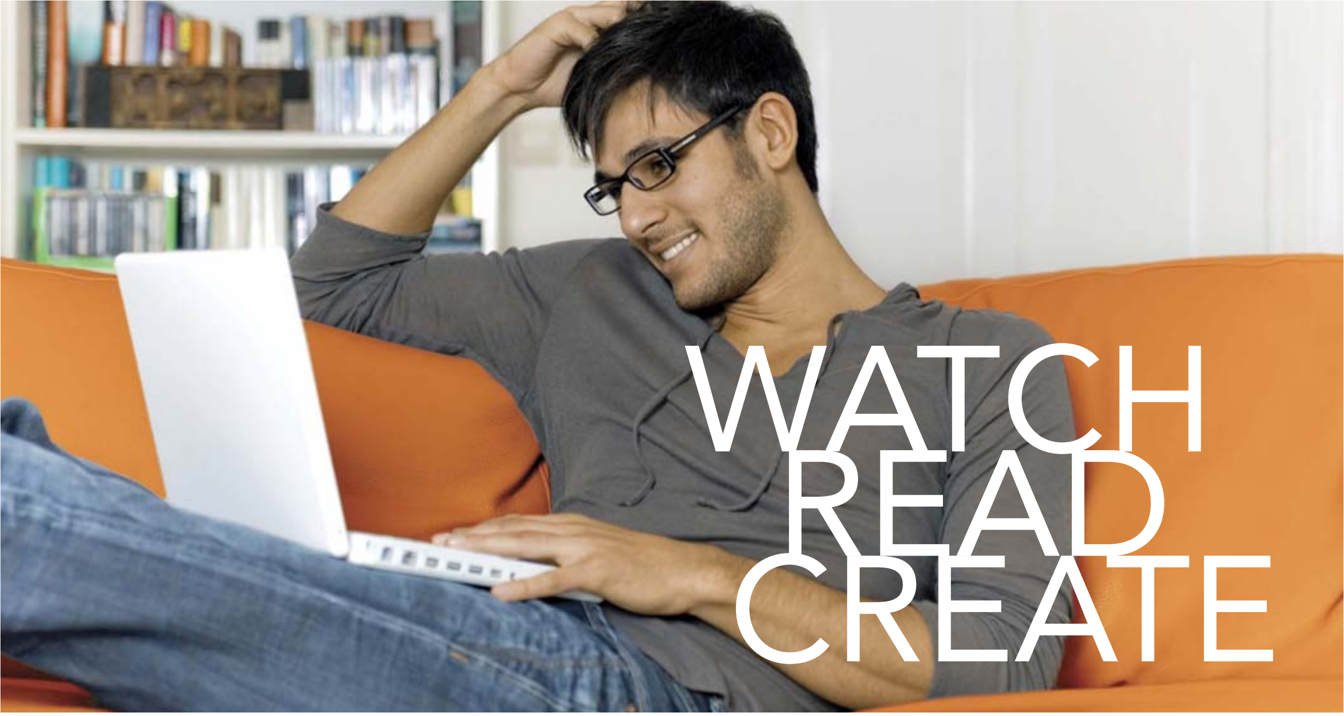
X

x = expr, 28
x conversion specifier, meaning of, 125
X conversion specifier, meaning of, 125

Z

zfill function, using, 97

From the Library of Bill Wiecking



Unlimited online access to all Peachpit, Adobe Press, Apple Training and New Riders videos and books, as well as content from other leading publishers including: O'Reilly Media, Focal Press, Sams, Que, Total Training, John Wiley & Sons, Course Technology PTR, Class on Demand, VTC and more.

No time commitment or contract required!
Sign up for one month or a year.
All for $19.99 a month

SIGN UP TODAY

peachpit.com/creativeedge

creative
edge

From the Library of Bill Wiecking

JOIN THE PEACHPIT AFFILIATE TEAM!

You love our books and you love to share them with your colleagues and friends...why not earn some $$ doing it!

If you have a website, blog or even a Facebook page, you can start earning money by putting a Peachpit link on your page.

If a visitor clicks on that link and purchases something on peachpit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post an ad and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.
To learn more go to:
http://www.peachpit.com/affiliates/

*Valid for all books, eBooks and video sales at www.Peachpit.com

Peachpit

From the Library of Bill Wiecking