TAKE YOUR
CODE TO
THE NEXT LEVEL

SUPERCHARGED PYTHON

BRIAN OVERLAND | JOHN BENNETT

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Supercharged Python

This page intentionally left blank

Supercharged Python

Brian Overland
John Bennett

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2019936408

Cover illustration: Open Studio/Shutterstock

ISBN-13: 978-0-13-515994-1
ISBN-10: 0-13-515994-6

1 19

To my beautiful and brilliant mother, Betty P. M. Overland. . . .
All the world is mad except for me and thee. Stay a little.
—Brian

To my parents, who did so much to shape who I am.
—John

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Books on Python aimed for the absolute beginner have become a cottage industry these days. Everyone and their dog, it seems, wants to chase the Python.

We're a little biased, but one book we especially recommend is *Python Without Fear*. It takes you by the hand and explains the major features one at a time. But what do you do after you know a little of the language but not enough to call yourself an "expert"? How do you learn enough to get a job or to write major applications?

That's what this book is for: to be the *second* book you ever buy on Python and possibly the last.

What Makes Python Special?

It's safe to say that many people are attracted to Python because it looks easier than C++. That may be (at least in the beginning), but underneath this so-called easy language is a tool of great power, with many shortcuts and software libraries called "packages" that—in some cases—do most of the work for you. These let you create some really impressive software, outputting beautiful graphs and manipulating large amounts of data.

For most people, it may take years to learn all the shortcuts and advanced features. This book is written for people who want to get that knowledge now, to get closer to being a Python expert much faster.

Paths to Learning: Where Do I Start?

This book offers different learning paths for different people.

- **You're rusty**: If you've dabbled in Python but you're a little rusty, you may want to take a look at Chapter 1, "Review of the Fundamentals." Otherwise, you may want to skip Chapter 1 or only take a brief look at it.
- **You know the basics but are still learning**: Start with Chapters 2 and 3, which survey the abilities of strings and lists. This survey includes some advanced abilities of these data structures that people often miss the first time they learn Python.
- **Your understanding of Python is strong, but you don't know everything yet**: Start with Chapter 4, which lists 22 programming shortcuts unique to Python, that most people take a long time to fully learn.
- **You want to master special features**: You can start in an area of specialty. For example, Chapters 5, 6, and 7 deal with text formatting and regular expressions. The two chapters on regular expression syntax, Chapters 6 and 7, start with the basics but then cover the finer points of this pattern-matching technology. Other chapters deal with other specialties. For example, Chapter 8 describes the different ways of handling text and binary files.
- **You want to learn advanced math and plotting software**: If you want to do plotting, financial, or scientific applications, start with Chapter 12, "The 'numpy' (Numeric Python) Package." This is the basic package that provides an underlying basis for many higher-level capabilities described in Chapters 13 through 15.

Clarity and Examples Are Everything

Even with advanced technology, our emphasis is on clarity, short examples, more clarity, and more examples. We emphasize an interactive approach, especially with the use of the IDLE environment, encouraging you to type in statements and see what they do. Text in bold represents lines for you to type in, or to be added or changed.

```
>>> print('Hello', 'my', 'world!')
Hello my world!
```

Several of the applications in this book are advanced pieces of software, including a Deck object, a fully functional "RPN" language interpreter, and a multifaceted stock-market program that presents the user with many choices. With these applications, we start with simple examples in the beginning, finally showing all the pieces in context. This approach differs from many

books, which give you dozens of functions all out of order, with no sense of architecture. In this book, architecture is everything.

You can download examples from **brianoverland.com/books**.

Learning Aids: Icons

This book makes generous use of tables for ease of reference, as well as conceptual art (figures). Our experience is that while poorly conceived figures can be a distraction, the best figures can be invaluable. A picture is worth a thousand words. Sometimes, more.

We also believe that in discussing plotting and graphics software, there's no substitute for showing all the relevant screen shots.

The book itself uses a few important, typographical devices. There are three special icons used in the text.

Note ▶ We sometimes use Notes to point out facts you'll eventually want to know but that diverge from the main discussion. You might want to skip over Notes the first time you read a section, but it's a good idea to go back later and read them.

◀ Note

The Key Syntax Icon introduces general syntax displays, into which you supply some or all of the elements. These elements are called "placeholders," and they appear in italics. Some of the syntax—especially keywords and punctuation—are in bold and intended to be typed in as shown. Finally, square brackets, when not in bold, indicate an optional item. For example:

```
set([iterable])
```

This syntax display implies that *iterable* is an iterable object (such as a list or a generator object) that you supply. And it's optional.

Square brackets, when in bold, are intended literally, to be typed in as shown. For example:

```
list_name = [obj1, obj2, obj3, …]
```

Ellipses (…) indicate a language element that can be repeated any number of times.

Performance Tip ▶ Performance tips are like Notes in that they constitute a short digression from the rest of the chapter. These tips address the question of how you can improve software performance. If you're interested in that topic, you'll want to pay special attention to these notes.

◀ Performance Tip

What You'll Learn

The list of topics in this book that are not in *Python Without Fear* or other "beginner" texts is a long one, but here is a partial list of some of the major areas:

- List, set, and dictionary comprehension.
- Regular expressions and advanced formatting techniques; how to use them in lexical analysis.
- Packages: the use of Python's advanced numeric and plotting software. Also, special types such as **`Decimal`** and **`Fraction`**.
- Mastering all the ways of using binary file operations in Python, as well as text operations.
- How to use multiple modules in Python while avoiding the "gotchas."
- Fine points of object-oriented programming, especially all the "magic methods," their quirks, their special features, and their uses.

Have Fun

When you master some or all of the techniques of this book, you should make a delightful discovery: Python often enables you to do a great deal with a relatively small amount of code. That's why it's dramatically increasing in popularity every day. Because Python is not just a time-saving device, it's fun to be able to program this way . . . to see a few lines of code do so much.

We wish you the joy of that discovery.

Register your copy of *Supercharged Python* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135159941) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

From Brian

I want to thank my coauthor, John Bennett. This book is the result of close collaboration between the two of us over half a year, in which John was there every step of the way to contribute ideas, content, and sample code, so his presence is there throughout the book. I also want to thank Greg Doench, acquisitions editor, who was a driving force behind the concept, purpose, and marketing of this book.

This book also had a wonderful supporting editorial team, including Rachel Paul and Julie Nahil. But I want to especially thank copy editor Betsy Hardinger, who showed exceptional competence, cooperation, and professionalism in getting the book ready for publication.

From John

I want to thank my coauthor, Brian Overland, for inviting me to join him on this book. This allows me to pass on many of the things I had to work hard to find documentation for or figure out by brute-force experimentation. Hopefully this will save readers a lot of work dealing with the problems I ran into.

This page intentionally left blank

About the Authors

Brian Overland started as a professional programmer back in his twenties, but also worked as a computer science, English, and math tutor. He enjoys picking up new languages, but his specialty is explaining them to others, as well as using programming to do games, puzzles, simulations, and math problems. Now he's the author of over a dozen books on programming.

In his ten years at Microsoft he was a software tester, programmer/writer, and manager, but his greatest achievement was in presenting Visual Basic 1.0, as lead writer and overall documentation project lead. He believes that project changed the world by getting people to develop for Windows, and one of the keys to its success was showing it could be fun and easy.

He's also a playwright and actor, which has come in handy as an instructor in online classes. As a novelist, he's twice been a finalist in the Pacific Northwest Literary Contest but is still looking for a publisher.

John Bennett was a senior software engineer at Proximity Technology, Franklin Electronic Publishing, and Microsoft Corporation. More recently, he's developed new programming languages using Python as a prototyping tool. He holds nine U.S. patents, and his projects include a handheld spell checker and East Asian handwriting recognition software.

This page intentionally left blank

Shortcuts, Command Line, and Packages

Master crafters need many things, but, above all, they need to master the tools of the profession. This chapter introduces tools that, even if you're a fairly experienced Python programmer, you may not have yet learned. These tools will make you more productive as well as increase the efficiency of your programs.

So get ready to learn some new tips and tricks.

4.1 Overview

Python is unusually gifted with shortcuts and time-saving programming techniques. This chapter begins with a discussion of twenty-two of these techniques.

Another thing you can do to speed up certain programs is to take advantage of the many packages that are available with Python. Some of these—such as **`re`** (regular expressions), **`system`**, **`random`**, and **`math`**—come with the standard Python download, and all you have to do is to include an **`import`** statement. Other packages can be downloaded quite easily with the right tools.

4.2 Twenty-Two Programming Shortcuts

This section lists the most common techniques for shortening and tightening your Python code. Most of these are new in the book, although a few of them have been introduced before and are presented in greater depth here.

- Use Python line continuation as needed.
- Use **`for`** loops intelligently.

- Understand combined operator assignment (+= etc.).
- Use multiple assignment.
- Use tuple assignment.
- Use advanced tuple assignment.
- Use list and string "multiplication."
- Return multiple values.
- Use loops and the **`else`** keyword.
- Take advantage of Booleans and **`not`**.
- Treat strings as lists of characters.
- Eliminate characters by using **`replace`**.
- Don't write unnecessary loops.
- Use chained comparisons.
- Simulate "switch" with a table of functions.
- Use the **`is`** operator correctly.
- Use one-line **`for`** loops.
- Squeeze multiple statements onto a line.
- Write one-line if/then/else statements.
- Create Enum values with **`range`**.
- Reduce the inefficiency of the **`print`** function within IDLE.
- Place underscores inside large numbers.

Let's look at these ideas in detail.

4.2.1 Use Python Line Continuation as Needed

In Python, the normal statement terminator is just the end of a physical line (although note the exceptions in Section 3.18). This makes programming easier, because you can naturally assume that statements are one per line.

But what if you need to write a statement longer than one physical line? This dilemma can crop up in a number of ways. For example, you might have a string to print that you can't fit on one line. You could use literal quotations, but line wraps, in that case, are translated as newlines—something you might

not want. The solution, first of all, is to recognize that literal strings positioned next to other literal strings are automatically concatenated.

```
>>> my_str = 'I am Hen-er-y the Eighth,' ' I am!'
>>> print(my_str)
I am Hen-er-y the Eighth, I am!
```

If these substrings are too long to put on a single physical line, you have a couple of choices. One is to use the line-continuation character, which is a backslash (\).

```
my_str = 'I am Hen-er-y the Eighth,' \
' I am!'
```

Another technique is to observe that any open—and so far unmatched—parenthesis, square bracket, or brace automatically causes continuation onto the next physical line. Consequently, you can enter as long a statement as you want—and you can enter a string of any length you want—without necessarily inserting newlines.

```
my_str = ('I am Hen-er-y the Eighth, '
'I am! I am not just any Henry VIII, '
'I really am!')
```

This statement places all this text in one string. You can likewise use open parentheses with other kinds of statements.

```
length_of_hypotenuse = ( (side1 * side1 + side2 * side2)
                          ** 0.5 )
```

A statement is not considered complete until all open parentheses [**(**] have been matched by closing parentheses [**)**]. The same is true for braces and square brackets. As a result, this statement will automatically continue to the next physical line.

4.2.2 Use "for" Loops Intelligently

If you come from the C/C++ world, you may tend to overuse the **range** function to print members of a list. Here's an example of the C way of writing a **for** loop, using **range** and an indexing operation.

```
beat_list = ['John', 'Paul', 'George', 'Ringo']
for i in range(len(beat_list)):
    print(beat_list[i])
```

If you ever write code like this, you should try to break the habit as soon as you can. It's better to print the contents of a list or iterator directly.

```
beat_list = ['John', 'Paul', 'George', 'Ringo']
for guy in beat_list:
    print(guy)
```

Even if you need access to a loop variable, it's better to use the **enumerate** function to generate such numbers. Here's an example:

```
beat_list = ['John', 'Paul', 'George', 'Ringo']
for i, name in enumerate(beat_list, 1):
    print(i, '. ', name, sep='')
```

This prints

```
1. John
2. Paul
3. George
4. Ringo
```

There are, of course, some cases in which it's necessary to use indexing. That happens most often when you are trying to change the contents of a list in place.

4.2.3 Understand Combined Operator Assignment (+= etc.)

The combined operator-assignment operators are introduced in Chapter 1 and so are reviewed only briefly here. Remember that assignment (**=**) can be combined with any of the following operators: **+**, **-**, **/**, **//**, **%**, ******, **&**, **^**, **|**, **<<**, **>>**.

The operators **&**, **|**, and **^** are bitwise "and," "or," and "exclusive or," respectively. The operators **<<** and **>>** perform bit shifts to the left and to the right.

This section covers some finer points of operator-assignment usage. First, any assignment operator has low precedence and is carried out last.

Second, an assignment operator may or may not be in place, depending on whether the type operated on is mutable. *In place* refers to operations that work on existing data in memory rather than creating a completely new object. Such operations are faster and more efficient.

Integers, floating-point numbers, and strings are immutable. Assignment operators, used with these types, do not cause in-place assignment; they instead must produce a completely new object, which is reassigned to the variable. Here's an example:

```
s1 = s2 = 'A string.'
s1 += '...with more stuff!'
print('s1:', s1)
print('s2:', s2)
```

The **`print`** function, in this case, produces the following output:

```
s1: A string...with more stuff!
s2: A string.
```

When `s1` was assigned a new value, it did not change the string data in place; it assigned a whole new string to `s1`. But `s2` is a name that still refers to the original string data. This is why `s1` and `s2` now contain different strings.

But lists are mutable, and therefore changes to lists can occur in place.

```
a_list = b_list = [10, 20]
a_list += [30, 40]
print('a_list:', a_list)
print('b_list:', b_list)
```

This code prints

```
a_list: [10, 20, 30, 40]
b_list: [10, 20, 30, 40]
```

In this case, the change was made to the list in place, so there was no need to create a new list and reassign that list to the variable. Therefore, `a_list` was not assigned to a new list, and `b_list`, a variable that refers to the same data in memory, reflects the change as well.

In-place operations are almost always more efficient. In the case of lists, Python reserves some extra space to grow when allocating a list in memory, and that in turns permits **`append`** operations, as well as **`+=`**, to efficiently grow lists. However, occasionally lists exceed the reserved space and must be moved. Such memory management is seamless and has little or no impact on program behavior.

Non-in-place operations are less efficient, because a new object must be created. That's why it's advisable to use the **`join`** method to grow large strings rather than use the **`+=`** operator, especially if performance is important. Here's an example using the **`join`** method to create a list and join 26 characters together.

```
str_list = []
n = ord('a')
for i in range(n, n + 26):
     str_list += chr(i)
alphabet_str = ''.join(str_list)
```

Figures 4.1 and 4.2 illustrate the difference between in-place operations and non-in-place operations. In Figure 4.1, string data seems to be appended onto an existing string, but what the operation really does is to create a new string and then assign it to the variable—which now refers to a different place in memory.

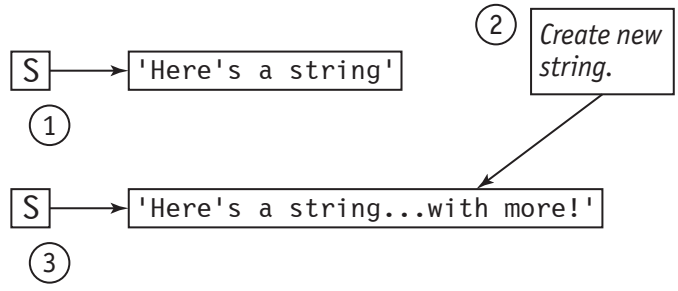

Figure 4.1. Appending to a string (not in-place)

But in Figure 4.2, list data is appended onto an existing list without the need to create a new list and reassign the variable.

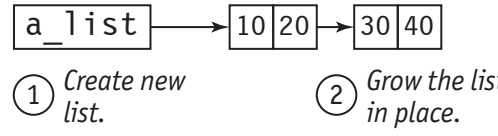

Figure 4.2. Appending to a list (in-place)

Here's a summary:

- Combined assignment operators such as **+=** cause in-place changes to data if the object is mutable (such as a list); otherwise, a whole new object is assigned to the variable on the left.
- In-place operations are faster and use space more efficiently, because they do not force creation of a new object. In the case of lists, Python usually allocates extra space so that the list can be grown more efficiently at run time.

4.2.4 Use Multiple Assignment

Multiple assignment is one of the most commonly used coding shortcuts in Python. You can, for example, create five different variables at once, assigning them all the same value—in this case, 0:

```
a = b = c = d = e = 0
```

Consequently, the following returns **`True`**:

```
a is b
```

This statement would no longer return **`True`** if either of these variables was later assigned to a different object.

Even though this coding technique may look like it is borrowed from C and C++, you should not assume that Python follows C syntax in most respects. Assignment in Python is a statement and not an expression, as it is in C.

4.2.5 Use Tuple Assignment

Multiple assignment is useful when you want to assign a group of variables the same initial value.

But what if you want to assign different values to different variables? For example, suppose you want to assign 1 to a, and 0 to b. The obvious way to do that is to use the following statements:

```
a = 1
b = 0
```

But through *tuple assignment*, you can combine these into a single statement.

```
a, b = 1, 0
```

In this form of assignment, you have a series of values on one side of the equals sign (=) and another on the right. They must match in number, with one exception: You can assign a tuple of any size to a single variable (which itself now represents a tuple as a result of this operation).

```
a = 4, 8, 12   # a is now a tuple containing three values.
```

Tuple assignment can be used to write some passages of code more compactly. Consider how compact a Fibonacci-generating function can be in Python.

```
def fibo(n):
    a, b = 1, 0
    while a <= n:
        print(a, end=' ')
        a, b = a + b, a
```

In the last statement, the variable `a` gets a new value: `a + b`; the variable `b` gets a new value—namely, the old value of `a`.

Most programming languages have no way to set `a` and `b` simultaneously. Setting the value of `a` changes what gets put into `b`, and vice versa. So normally, a temporary variable would be required. You could do that in Python, if you wanted to:

```
temp = a      # Preserve old value of a
a = a + b     # Set new value of a
b = temp      # Set b to old value of a
```

But with tuple assignment, there's no need for a temporary variable.

```
a, b = a + b, a
```

Here's an even simpler example of tuple assignment. Sometimes, it's useful to swap two values.

```
x, y = 1, 25
print(x, y)   # prints 1 25
x, y = y, x
print(x, y)   # prints 25 1
```

The interesting part of this example is the statement that performs the swap:

```
x, y = y, x
```

In another language, such an action would require three separate statements. But Python does not require this, because—as just shown—it can do the swap all at once. Here is what another language would require you to do:

```
temp = x
x = y
y = temp
```

4.2.6 Use Advanced Tuple Assignment

Tuple assignment has some refined features. For example, you can unpack a tuple to assign to multiple variables, as in the following example.

```
tup = 10, 20, 30
a, b, c = tup
print(a, b, c)   # Produces 10, 20, 30
```

It's important that the number of input variables on the left matches the size of the tuple on the right. The following statement would produce a runtime error.

```
tup = 10, 20, 30
a, b = tup   # Error: too many values to unpack
```

Another technique that's occasionally useful is creating a tuple that has one element. That would be easy to do with lists.

```
my_list = [3]
```

This is a list with one element, 3. But the same approach won't work with tuples.

```
my_tup = (3)
print(type(my_tup))
```

This **`print`** statement shows that `my_tup`, in this case, produced a simple integer.

```
<class 'int'>
```

This is not what was wanted in this case. The parentheses were treated as a no-op, as would any number of enclosing parentheses. But the following statement produces a tuple with one element, although, to be fair, a tuple with just one element isn't used very often.

```
my_tup = (3,)     # Assign tuple with one member, 3.
```

The use of an asterisk (`*`) provides a good deal of additional flexibility with tuple assignment. You can use it to split off parts of a tuple and have one (and only one) variable that becomes the default target for the remaining elements, which are then put into a list. Some examples should make this clear.

```
a, *b = 2, 4, 6, 8
```

In this example, `a` gets the value 2, and `b` is assigned to a list:

```
2
[4, 6, 8]
```

You can place the asterisk next to any variable on the left, but in no case more than one. The variable modified with the asterisk is assigned a list of whatever elements are left over. Here's an example:

```
a, *b, c = 10, 20, 30, 40, 50
```

In this case, `a` and `c` refer to 10 and 50, respectively, after this statement is executed, and `b` is assigned the list `[20, 30, 40]`.

You can, of course, place the asterisk next to a variable at the end.

```
big, bigger, *many = 100, 200, 300, 400, 500, 600
```

Printing these variables produces the following:

```
>>> print(big, bigger, many, sep='\n')
100
200
[300, 400, 500, 600]
```

4.2.7 Use List and String "Multiplication"

Serious programs often deal with large data sets—for example, a collection of 10,000 integers all initialized to 0. In languages such as C and Java, the way to do this is to first declare an array with a large dimension.

Because there are no data declarations in Python, the only way to create a large list is to construct it on the right side of an assignment. But constructing a super-long list by hand is impractical. Imagine trying to construct a super-long list this way:

```
my_list = [0, 0, 0, 0, 0, 0, 0, 0...]
```

As you can imagine, entering 10,000 zeros into program code would be very time-consuming! And it would make your hands ache.

Applying the multiplication operator provides a more practical solution:

```
my_list = [0] * 10000
```

This example creates a list of 10,000 integers, all initialized to 0.

Such operations are well optimized in Python, so that even in the interactive development environment (IDLE), such interactions are handled quickly.

```
>>> my_list = [0] * 10000
>>> len(my_list)
10000
```

Note that the integer may be either the left or the right operand in such an expression.

```
>>> my_list = 1999 * [12]
>>> len(my_list)
1999
```

You can also "multiply" longer lists. For example, the following list is 300 elements long. It consists of the numbers 1, 2, 3, repeated over and over.

```
>>> trip_list = [1, 2, 3] * 100
>>> len(trip_list)
300
```

The multiplication sign (*) does not work with dictionaries and sets, which require unique keys. But it does work with the string class (**str**); for example, you can create a string consisting of 40 underscores, which you might use for display purposes:

```
divider_str = '_' * 40
```

Printing out this string produces the following:

```
________________________________________
```

4.2.8 Return Multiple Values

You can't pass a simple variable to a Python function, change the value inside the function, and expect the original variable to reflect the change. Here's an example:

```
def double_me(n):
    n *= 2

a = 10
double_me(a)
print(a)        # Value of a did not get doubled!!
```

When `n` is assigned a new value, the association is broken between that variable and the value that was passed. In effect, `n` is a local variable that is now associated with a different place in memory. The variable passed to the function is unaffected.

But you can always use a return value this way:

```
def double_me(n):
    return n * 2

a = 10
a = double_me(a)
print(a)
```

Therefore, to get an *out* parameter, just return a value. But what if you want more than one out parameter?

In Python, you can return as many values as you want. For example, the following function performs the quadratic equation by returning two values.

```
def quad(a, b, c):
    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
```

```
    x2 = (-b - determin) / (2 * a)
    return x1, x2
```

This function has three input arguments and two output variables. In calling the function, it's important to receive both arguments:

```
x1, x2 = quad(1, -1, -1)
```

If you return multiple values to a single variable in this case, that variable will store the values as a tuple. Here's an example:

```
>>> x = quad(1, -1, -1)
>>> x
(1.618033988749895, -0.6180339887498949)
```

Note that this feature—returning multiple values—is actually an application of the use of tuples in Python.

4.2.9 Use Loops and the "else" Keyword

The **`else`** keyword is most frequently used in combination with the **`if`** keyword. But in Python, it can also be used with **`try-except`** syntax and with loops.

With loops, the **`else`** clause is executed if the loop has completed without an early exit, such as **`break`**. This feature applies to both **`while`** loops and **`for`** loops.

The following example tries to find an even divisor of `n`, up to and including the limit, `max`. If no such divisor is found, it reports that fact.

```
def find_divisor(n, max):
    for i in range(2, max + 1):
        if n % i == 0:
            print(i, 'divides evenly into', n)
            break
    else:
        print('No divisor found')
```

Here's an example:

```
>>> find_divisor(49, 6)
No divisor found
>>> find_divisor(49, 7)
7 divides evenly into 49
```

4.2.10 Take Advantage of Boolean Values and "not"

Every object in Python evaluates to **True** or **False**. For example, every empty collection in Python evaluates to **False** if tested as a Boolean value; so does the special value **None**. Here's one way of testing a string for being length zero:

```
if len(my_str) == 0:
    break
```

However, you can instead test for an input string this way:

```
if not s:
     break
```

Here are the general guidelines for Boolean conversions.

- Nonempty collections and nonempty strings evaluate as **True**; so do nonzero numeric values.
- Zero-length collections and zero-length strings evaluate to **False**; so does any number equal to 0, as well as the special value **None**.

4.2.11 Treat Strings as Lists of Characters

When you're doing complicated operations on individual characters and building a string, it's sometimes more efficient to build a list of characters (each being a string of length 1) and use list comprehension plus **join** to put it all together.

For example, to test whether a string is a palindrome, it's useful to omit all punctuation and space characters and convert the rest of the string to either all-uppercase or all-lowercase. List comprehension does this efficiently.

```
test_str = input('Enter test string: ')
a_list = [c.upper() for c in test_str if c.isalnum()]
print(a_list == a_list[::-1])
```

The second line in this example uses list comprehension, which was introduced in Section 3.15, "List Comprehension."

The third line in this example uses slicing to get the reverse of the list. Now we can test whether `test_str` is a palindrome by comparing it to its own reverse. These three lines of code have to be the shortest possible program for testing whether a string is a palindrome. Talk about compaction!

```
Enter test string: A man, a plan, a canal, Panama!
True
```

4.2.12 Eliminate Characters by Using "replace"

To quickly remove all instances of a particular character from a string, use **`replace`** and specify the empty string as the replacement.

For example, a code sample in Chapter 10 asks users to enter strings that represent fractions, such as "1/2". But if the user puts extra spaces in, as in "1 / 2", this could cause a problem. Here's some code that takes an input string, `s`, and quickly rids it of all spaces wherever they are found (so it goes beyond stripping):

```
s = s.replace(' ', '')
```

Using similar code, you can quickly get rid of all offending characters or substrings in the same way—but only one at a time. Suppose, however, that you want to get rid of all vowels in one pass. List comprehension, in that case, comes to your aid.

```
a_list = [c for c in s if c not in 'aeiou']
s = ''.join(a_list)
```

4.2.13 Don't Write Unnecessary Loops

Make sure that you don't overlook all of Python's built-in abilities, especially when you're working with lists and strings. With most computer languages, you'd probably have to write a loop to get the sum of all the numbers in a list. But Python performs summation directly. For example, the following function calculates 1 + 2 + 3 … + N:

```
def calc_triangle_num(n):
     return sum(range(n+1))
```

Another way to use the **`sum`** function is to quickly get the average (the mean) of any list of numbers.

```
def get_avg(a_list):
    return sum(a_list) / len(a_list)
```

4.2.14 Use Chained Comparisons (n < x < m)

This is a slick little shortcut that can save you a bit of work now and then, as well as making your code more readable.

It's common to write **`if`** conditions such as the following:

```
if 0 < x and x < 100:
    print('x is in range.')
```

But in this case, you can save a few keystrokes by instead using this:

```
if 0 < x < 100:        # Use 'chained' comparisons.
    print('x is in range.')
```

This ability potentially goes further. You can chain together any number of comparisons, and you can include any of the standard comparison operators, including **==**, **<**, **<=**, **>**, and **>=**. The arrows don't even have to point in the same direction or even be combined in any order! So you can do things like this:

```
a, b, c = 5, 10, 15
if 0 < a <= c > b > 1:
    print('All these comparisons are true!')
    print('c is equal or greater than all the rest!')
```

You can even use this technique to test a series of variables for equality. Here's an example:

```
a = b = c = d = e = 100
if a == b == c == d == e:
    print('All the variables are equal to each other.')
```

For larger data sets, there are ways to achieve these results more efficiently. Any list, no matter how large, can be tested to see whether all the elements are equal this way:

```
if min(a_list) == max(a_list):
    print('All the elements are equal to each other.')
```

However, when you just want to test a few variables for equality or perform a combination of comparisons on a single line, the techniques shown in this section are a nice convenience with Python. Yay, Python!

4.2.15 Simulate "switch" with a Table of Functions

This next technique is nice because it can potentially save a number of lines of code.

Section 15.12 offers the user a menu of choices, prompts for an integer, and then uses that integer to decide which of several functions to call. The obvious way to implement this logic is with a series of **if**/**elif** statements, because Python has no "switch" statement.

```
if n == 1:
    do_plot(stockdf)
elif n == 2:
    do_highlow_plot(stockdf)
```

```
elif n == 3:
    do_volume_subplot(stockdf)
elif n == 4:
    do_movingavg_plot(stockdf)
```

Code like this is verbose. It will work, but it's longer than it needs to be. But Python functions are objects, and they can be placed in a list just like any other kind of objects. You can therefore get a reference to one of the functions and call it.

```
fn = [do_plot, do_highlow_plot, do_volume_subplot,
      do_movingavg_plot][n-1]
fn(stockdf)                  # Call the function
```

For example, `n-1` is evaluated, and if that value is 0 (that is, `n` is equal to 1), the first function listed, `do_plot`, is executed.

This code creates a compact version of a C++ switch statement by calling a different function depending on the value of `n`. (By the way, the value 0 is excluded in this case, because that value is used to exit.)

You can create a more flexible control structure by using a dictionary combined with functions. For example, suppose that "load," "save," "update," and "exit" are all menu functions. We might implement the equivalent of a switch statement this way:

```
menu_dict = {'load':load_fn, 'save':save_fn,
             'exit':exit_fn, 'update':update_fn}
(menu_dict[selector])()       # Call the function
```

Now the appropriate function will be called, depending on the string contained in `selector`, which presumably contains `'load'`, `'save'`, `'update'`, or `'exit'`.

4.2.16 Use the "is" Operator Correctly

Python supports both a test-for-equality operator (**==**) and an **is** operator. These tests sometimes return the same result, and sometimes they don't. If two strings have the same value, a test for equality always produces **True**.

```
a = 'cat'
b = 'cat'
a == b    # This must produce True.
```

But the **is** operator isn't guaranteed to produce **True** in string comparisons, and it's risky to rely upon. A constructed string isn't guaranteed to

match another string if you use **is** rather than test-for-equality (**==**). For example:

```
>>> s1 = 'I am what I am and that is all that I am.'
>>> s2 = 'I am what I am' + ' and that is all that I am.'
>>> s1 == s2
True
>>> s1 is s2
False
```

What this example demonstrates is that just because two strings have identical contents does not mean that they correspond to the same object in memory, and therefore the **is** operator produces **False**.

If the **is** operator is unreliable in such cases, why is it in the language at all? The answer is that Python has some unique objects, such as **None**, **True**, and **False**. When you're certain that you're comparing a value to a unique object, then the **is** keyword works reliably; moreover, it's preferable in those situations because such a comparison is more efficient.

```
a_value = my_function()
if a_value is None:
    # Take special action if None is returned.
```


4.2.17 Use One-Line "for" Loops

If a **for** loop is short enough, with only one statement inside the loop (that is, the statement *body*), you can squeeze the entire **for** loop onto a single physical line.


```
for var in sequence: statement
```

Not all programmers favor this programming style. However, it's useful as a way of making your program more compact. For example, the following one-line statement prints all the numbers from 0 to 9:

```
>>> for i in range(10): print(i, end='  ')

0  1  2  3  4  5  6  7  8  9
```

Notice that when you're within IDLE, this **for** loop is like any other: You need to type an extra blank line in order to terminate it.

4.2.18 Squeeze Multiple Statements onto a Line

If you have a lot of statements you want to squeeze onto the same line, you can do it—if you're determined and the statements are short enough.

The technique is to use a semicolon (;) to separate one statement on a physical line from another. Here's an example:

```
>>> for i in range(5): n=i*2; m = 5; print(n+m, end=' ')

5 7 9 11 13
```

You can squeeze other kinds of loops onto a line in this way. Also, you don't have to use loops but can place any statements on a line that you can manage to fit there.

```
>>> a = 1; b = 2; c = a + b; print(c)
3
```

At this point, some people may object, "But with those semicolons, this looks like C code!" (Oh, no—anything but that!)

Maybe it does, but it saves space. Keep in mind that the semicolons are statement separators and not terminators, as in the old Pascal language.

4.2.19 Write One-Line if/then/else Statements

This feature is also called an *in line* **if** conditional. Consider the following if/else statement, which is not uncommon:

```
turn = 0
...
if turn % 2:
    cell = 'X'
else:
    cell = 'O'
```

The book *Python Without Fear* uses this program logic to help operate a tic-tac-toe game. On alternate turns, the cell to be added was either an "X" or an "O". The turn counter, advanced by 1 each time, caused a switch back and forth (a *toggle*) between the two players, "X" and "O."

That book replaced the if/else block just shown with the more compact version:

```
cell = 'X' if turn % 2 else 'O'
```


```
true_expr if conditional else false_expr
```

If the *conditional* is true, then the *true_expr* is evaluated and returned; otherwise the *false_expr* is evaluated and returned.

4.2.20 Create Enum Values with "range"

Many programmers like to use enumerated (or "enum") types in place of so-called magic numbers. For example, if you have a `color_indicator` variable, in which the values 1 through 5 represent the values red, green, blue, back, and white, the code becomes more readable if you can use the color names instead of using the literal numbers 1 through 5.

You could make this possible by assigning a number to each variable name.

```
red  = 0
blue = 1
green = 2
black = 3
white = 4
```

This works fine, but it would be nice to find a way to automate this code. There is a simple trick in Python that allows you to do that, creating an enumeration. You can take advantage of multiple assignment along with use of the **range** function:

```
red, blue, green, black, white = range(5)
```

The number passed to **range** in this case is the number of settings. Or, if you want to start the numbering at 1 instead of 0, you can use the following:

```
red, blue, green, black, white = range(1, 6)
```

Note ▶ For more sophisticated control over the creation and specification of enumerated types, you can import and examine the **enum** package.

```
import enum
help(enum)
```

You can find information on this feature at

https://docs.python.org/3/library/enum.html.

◀ Note

4.2.21 Reduce the Inefficiency of the "print" Function Within IDLE

Within IDLE, calls to the **print** statement are incredibly slow. If you run programs from within the environment, you can speed up performance dramatically by reducing the number of separate calls to **print**.

For example, suppose you want to print a 40 × 20 block of asterisks (*). The slowest way to do this, by far, is to print each character individually. Within IDLE, this code is painfully slowly.

```
for i in range(20):
    for j in range(40):
        print('*', end='')
    print()
```

You can get much better performance by printing a full row of asterisks at a time.

```
row_of_asterisks = '*' * 40
for i in range(20):
    print(row_of_asterisks)
```

But the best performance is achieved by revising the code so that it calls the **print** function only once, after having assembled a large, multiline output string.

```
row_of_asterisks = '*' * 40
s = ''
for i in range(20):
    s += row_of_asterisks + '\n'
print(s)
```

This example can be improved even further by utilizing the string class **join** method. The reason this code is better is that it uses in-place appending of a list rather than appending to a string, which must create a new string each time.

```
row_of_asterisks = '*' * 40
list_of_str = []
for i in range(20):
    list_of_str.append(row_of_asterisks)
print('\n'.join(list_of_str))
```

Better yet, here is a one-line version of the code!

```
print('\n'.join(['*' * 40] * 20))
```

4.2.22 Place Underscores Inside Large Numbers

In programming, you sometimes have to deal with large numeric literals. Here's an example:

```
CEO_salary = 1500000
```

Such numbers are difficult to read in programming code. You might like to use commas as separators, but commas are reserved for other purposes, such as creating lists. Fortunately, Python provides another technique: You can use underscores (_) inside a numeric literal.

```
CEO_salary = 1_500_000
```

Subject to the following rules, the underscores can be placed anywhere inside the number. The effect is for Python to read the number as if no underscores were present. This technique involves several rules.

- You can't use two underscores in a row.
- You can't use a leading or trailing underscore. If you use a leading underscore (as in _1), the figure is treated as a variable name.
- You *can* use underscores on either side of a decimal point.

This technique affects only how numbers appear in the code itself and not how anything is printed. To print a number with thousands-place separators, use the **`format`** function or method as described in Chapter 5, "Formatting Text Precisely."

4.3 Running Python from the Command Line

If you've been running Python programs from within IDLE—either as commands entered one at a time or as scripts—one way to improve execution speed is to run programs from a command line instead; in particular, doing so greatly speeds up the time it takes to execute calls to the **`print`** function.

Some of the quirks of command-line operation depend on which operating system you're using. This section covers the two most common operating systems: Windows and Macintosh.

4.3.1 Running on a Windows-Based System

Windows systems, unlike Macintosh, usually do not come with a version of Python 2.0 preloaded, a practice that actually saves you a good deal of fuss as long as you install Python 3 yourself.

To use Python from the command line, first start the DOS Box application, which is present as a major application on all Windows systems. Python should be easily available because it should be placed in a directory that is part of the PATH setting. Checking this setting is easy to do while you're running a Windows DOS Box.

In Windows, you can also check the PATH setting by opening the Control Panel, choose Systems, and select the Advanced tab. Then click Environment Variables.

You then should be able to run Python programs directly as long as they're in your PATH. To run a program from the command line, enter **`python`** and the name of the source file (the main module), including the **`.py`** extension.

```
python test.py
```

4.3.2 Running on a Macintosh System

Macintosh systems often come with a version of Python already installed; unfortunately, on recent systems, the version is Python 2.0 and not Python 3.0.

To determine which version has been installed for command-line use, first bring up the Terminal application on your Macintosh system. You may need to first click the Launchpad icon.

You should find yourself in your default directory, whatever it is. You can determine which command-line version of Python you have by using the following command:

```
python -V
```

If the version of Python is 2.0+, you'll get a message such as the following:

```
python 2.7.10
```

But if you've downloaded some version of Python 3.0, you should have that version of Python loaded as well. However, to run it, you'll have to use the command **`python3`** rather than **`python`**.

If you do have **`python3`** loaded, you can verify the exact version from the command line as follows:

```
python3 -V
python 3.7.0
```

For example, if the file `test.py` is in the current directory, and you want to compile it as a Python 3.0 program, then use the following command:

```
python3 test.py
```

The Python command (whether **`python`** or **`python3`**) has some useful variations. If you enter it with `-h`, the "help" flag, you get a printout on all the

possible flags that you can use with the command, as well as relevant environment variables.

```
python3 -h
```

4.3.3 Using pip or pip3 to Download Packages

Some of the packages in this book require that you download and install the packages from the Internet before you use those packages. The first chapter that requires that is Chapter 12, which introduces the **numpy** package.

All the packages mentioned in this book are completely free of charge (as most packages for Python are). Even better, the **pip** utility—which is included with the Python 3 download—goes out and finds the package that you name; thus all you should need is an Internet connection!

On Windows-based systems, use the following command to download and install a desired package.

```
pip install package_name
```

The package name, incidentally, uses no file extension:

```
pip install numpy
```

On Macintosh systems, you may need to use the **pip3** utility, which is download with Python 3 when you install it on your computer. (You may also have inherited a version of pip, but it will likely be out-of-date and unusable.)

```
pip3 install package_name
```

4.4 Writing and Using Doc Strings

Python *doc strings* enable you to leverage the work you do writing comments to get free online help. That help is then available to you while running IDLE, as well as from the command line, when you use the **pydoc** utility.

You can write doc strings for both functions and classes. Although this book has not yet introduced how to write classes, the principles are the same. Here's an example with a function, showcasing a doc string.

```
def quad(a, b, c):
    '''Quadratic Formula function.

    This function applies the Quadratic Formula
    to determine the roots of x in a quadratic
    equation of the form ax^2 + bx + c = 0.
    '''
```

```
    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
    x2 = (-b - determin) / (2 * a)
    return x1, x2
```

When this doc string is entered in a function definition, you can get help from within IDLE:

```
>>> help(quad)
Help on function quad in module __main__:

quad(a, b, c)
    Quadratic Formula function.

    This function applies the Quadratic Formula
    to determine the roots of x in a quadratic
    equation of the form ax^2 + bx + c = 0.
```

The mechanics of writing a doc string follow a number of rules.

- The doc string itself must immediately follow the heading of the function.
- It must be a literal string utilizing the triple-quote feature. (You can actually use any style quote, but you need a literal quotation if you want to span multiple lines.)
- The doc string must also be aligned with the "level-1" indentation under the function heading: For example, if the statements immediately under the function heading are indented four spaces, then the beginning of the doc string must also be indented four spaces.
- Subsequent lines of the doc string may be indented as you choose, because the string is a literal string. You can place the subsequent lines flush left or continue the indentation you began with the doc string. In either case, Python online help will line up the text in a helpful way.

This last point needs some clarification. The doc string shown in the previous example could have been written this way:

```
def quad(a, b, c):
    '''Quadratic Formula function.

    This function applies the Quadratic Formula
    to determine the roots of x in a quadratic
    equation of the form ax^2 + bx + c = 0.
    '''
```

```
    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
    x2 = (-b - determin) / (2 * a)
    return x1, x2
```

You might expect this doc string to produce the desired behavior—to print help text that lines up—and you'd be right. But you can also put in extra spaces so that the lines also align *within program code*. It might seem this shouldn't work, but it does.

For stylistic reasons, programmers are encouraged to write the doc string this way, in which the subsequent lines in the quote line up with the beginning of the quoted string instead of starting flush left in column 1:

```
def quad(a, b, c):
    '''Quadratic Formula function.

    This function applies the Quadratic Formula
    to determine the roots of x in a quadratic
    equation of the form ax^2 + bx + c = 0.
    '''
```

As part of the stylistic guidelines, it's recommended that you put in a brief summary of the function, followed by a blank line, followed by more detailed description.

When running Python from the command line, you can use the **pydoc** utility to get this same online help shown earlier. For example, you could get help on the module named `queens.py`. The **pydoc** utility responds by printing a help summary for every function. Note that "py" is *not* entered as part of the module name in this case.

```
python -m pydoc queens
```

4.5 Importing Packages

Later sections in this chapter, as well as later chapters in the book, make use of packages to extend the capabilities of the Python language.

A *package* is essentially a software library of objects and functions that perform services. Packages come in two varieties:

- Packages included with the Python download itself. This includes **`math`**, **`random`**, **`sys`**, **`os`**, **`time`**, **`datetime`**, and **`os.path`**. These packages are especially convenient, because no additional downloading is necessary.
- Packages you can download from the Internet.

The syntax shown here is the recommended way to an import a package. There are a few variations on this syntax, as we'll show later.


```
import package_name
```

For example:

```
import math
```

Once a package is imported, you can, within IDLE, get help on its contents. Here's an example:

```
>>> import math
>>> help(math)
```

If you type these commands from within IDLE, you'll see that the math package supports a great many functions.

But with this approach, each of the functions needs to be qualified using the dot (.) syntax. For example, one of the functions supported is **sqrt** (square root), which takes an integer or floating-point input.

```
>>> math.sqrt(2)
1.4142135623730951
```

You can use the **math** package, if you choose, to calculate the value of **pi**. However, the math package also provides this number directly.

```
>>> math.atan(1) * 4
3.141592653589793
>>> math.pi
3.141592653589793
```

Let's look at one of the variations on the **import** statement.


```
import package_name [as new_name]
```

In this syntax, the brackets indicate that the **as** *new_name* clause is optional. You can use it, if you choose, to give the package another name, or *alias*, that is referred to in your source file.

This feature provides short names if the full package name is long. For example, Chapter 13 introduces the **matplotlib.pyplot** package.

```
import matplotlib.pyplot as plt
```

Now, do you want to use the prefix **matplotlib.pyplot**, or do you want to prefix a function name with **plt**? Good. We thought so.

Python supports other forms of syntax for the **import** statement. With both of these approaches, the need to use the package name and the dot syntax is removed.


```
from package_name import symbol_name
from package_name import *
```

In the first form of this syntax, only the *symbol_name* gets imported, and not the rest of the package. But the specified symbol (such as **pi** in this next example) can then be referred to without qualification.

```
>>> from math import pi
>>> print(pi)
3.141592653589793
```

This approach imports only one symbol—or a series of symbols separated by commas—but it enables the symbolic name to be used more directly. To import an entire package, while also gaining the ability to refer to all its objects and functions directly, use the last form of the syntax, which includes an asterisk (*).

```
>>> from math import *
>>> print(pi)
3.141592653589793
>>> print(sqrt(2))
1.4142135623730951
```

The drawback of using this version of **import** is that with very large and complex programs, it gets difficult to keep track of all the names you're using, and when you import packages without requiring a package-name qualifier, name conflicts can arise.

So, unless you know what you're doing or are importing a really small package, it's more advisable to import specific symbols than use the asterisk (*****).

4.6 A Guided Tour of Python Packages

Thousands of other packages are available if you go to **python.org**, and they are all free to use. The group of packages in Table 4.1 is among the most useful of all packages available for use with Python, so you should be sure to look them over.

The **re**, **math**, **random**, **array**, **decimal**, and **fractions** packages are all included with the standard Python 3 download, so you don't need to download them separately.

The **numpy**, **matplotlib**, and **pandas** packages need to be installed separately by using the **pip** or **pip3** utility. Later chapters, starting with Chapter 12, cover those utilities in depth.

Table 4.1. Python Packages Covered in This Book

NAME TO IMPORT	DESCRIPTION
`re`	Regular-expression package. This package lets you create text patterns that can match many different words, phrases, or sentences. This pattern-specification language can do sophisticated searches with high efficiency. This package is so important that it's explored in both Chapters 6 and 7.
`math`	Math package. Contains helpful and standard math functions so that you don't have to write them yourself. These include trigonometric, hyperbolic, exponential, and logarithmic functions, as well as the constants `e` and `pi`. This package is explored in Chapter 11.
`random`	A set of functions for producing pseudo-random values. Pseudo-random numbers behave as if random—meaning, among other things, it's a practical impossibility for a user to predict them. This random-number generation package includes the ability to produce random integers from a requested range, as well as floating-point numbers and normal distributions. The latter cluster around a mean value to form a "bell curve" of frequencies. This package is explored in Chapter 11.
`decimal`	This package supports the `Decimal` data type, which (unlike the `float` type) enables you to represent dollars-and-cents figures precisely without any possibility of rounding errors. `Decimal` is often preferred for use in accounting and financial applications. This package is explored in Chapter 10.
`fractions`	This package supports the `Fraction` data type, which stores any fractional number with absolute precision, provided it can be represented as the ratio of two integers. So, for example, this data type can represent the ratio 1/3 absolutely, something that neither the `float` nor `Decimal` type can do without rounding errors. This package is explored in Chapter 10.
`array`	This package supports the `array` class, which differs from lists in that it holds raw data in contiguous storage. This isn't always faster, but sometimes it's necessary to pack your data into contiguous storage so as to interact with other processes. However, the benefits of this package are far exceeded by the `numpy` package, which gives you the same ability, but much more. This package is briefly covered in Chapter 12.
`numpy`	This package supports the `numpy` (numeric Python) class, which in turn supports high-speed batch operations on one-, two-, and higher-dimensional arrays. The class is useful not only in itself, as a way of supercharging programs that handle large amounts of data, but also as the basis for work with other classes. This package is explored in Chapters 12 and 13. `numpy` needs to be installed with `pip` or `pip3`.

Table 4.1. Python Packages Covered in This Book (*continued*)

NAME TO IMPORT	DESCRIPTION
`numpy.random`	Similar to **`random`**, but designed especially for use with **`numpy`**, and ideally suited to situations in which you need to generate a large quantity of random numbers quickly. In head-to-head tests with the standard **`random`** class, the **`numpy`** random class is several times faster when you need to create an array of such numbers. This package is also explored in Chapter 12.
`matplotlib.pyplot`	This package supports sophisticated plotting routines for Python. Using these routines, you can create beautiful looking charts and figures—even three-dimensional ones. This package is explored in Chapter 13. It needs to be installed with **`pip`** or **`pip3`**.
`pandas`	This package supports *data frames*, which are tables that can hold a variety of information, as well as routines for going out and grabbing information from the Internet and loading it. Such information can then be combined with the **`numpy`** and plotting routines to create impressive-looking graphs. This package is explored in Chapter 15. It also needs to be downloaded.

4.7 Functions as First-Class Objects

Another productivity tool—which may be useful in debugging, profiling, and related tasks—is to treat Python functions as *first-class* objects. That means taking advantage of how you can get information about a function at run time. For example, suppose you've defined a function called `avg`.

```
def avg(a_list):
    '''This function finds the average val in a list.'''
    x = (sum(a_list) / len(a_list))
    print('The average is:', x)
    return x
```

The name `avg` is a symbolic name that refers to a function, which in Python lingo is also a callable. There are a number of things you can do with `avg`, such as verify its type, which is **`function`**. Here's an example:

```
>>> type(avg)
<class 'function'>
```

We already know that `avg` names a function, so this is not new information. But one of the interesting things you can do with an object is assign it to a

new name. You can also assign a different function altogether to the symbolic name, avg.

```
def new_func(a_list):
    return (sum(a_list) / len(a_list))

old_avg = avg
avg = new_func
```

The symbolic name old_avg now refers to the older, and longer, function we defined before. The symbolic name avg now refers to the newer function just defined.

The name old_avg now refers to our first averaging function, and we can call it, just as we used to call avg.

```
>>> old_avg([4, 6])
The average is 5.0
5.0
```

The next function shown (which we might loosely term a "metafunction," although it's really quite ordinary) prints information about another function—specifically, the function argument passed to it.

```
def func_info(func):
    print('Function name:', func.__name__)
    print('Function documentation:')
    help(func)
```

If we run this function on old_avg, which has been assigned to our first averaging function at the beginning of this section, we get this result:

```
Function name: avg
Function documentation:
Help on function avg in module __main__:

avg(a_list)
    This function finds the average val in a list.
```

We're currently using the symbolic name old_avg to refer to the first function that was defined in this section. Notice that when we get the function's name, the information printed uses the name that *the function was originally defined with*.

All of these operations will become important when we get to the topic of "decorating" in Section 4.9, "Decorators and Function Profilers."

4.8 Variable-Length Argument Lists

One of the most versatile features of Python is the ability to access variable-length argument lists. With this capability, your functions can, if you choose, handle any number of arguments—much as the built-in **print** function does.

The variable-length argument ability extends to the use of named arguments, also called "keyword arguments."

4.8.1 The *args List

The ***args** syntax can be used to access argument lists of any length.

```
def func_name([ordinary_args,] *args):
    statements
```

The brackets are used in this case to show that ***args** may optionally be preceded by any number of ordinary positional arguments, represented here as *ordinary_args*. The use of such arguments is always optional.

In this syntax, the name **args** can actually be any symbolic name you want. By convention, Python programs use the name **args** for this purpose.

The symbolic name **args** is then interpreted as a Python list like any other; you expand it by indexing it or using it in a **for** loop. You can also take its length as needed. Here's an example:

```
def my_var_func(*args):
    print('The number of args is', len(args))
    for item in args:
        print(items)
```

This function, my_var_func, can be used with argument lists of any length.

```
>>> my_var_func(10, 20, 30, 40)
The number of args is 4
10
20
30
40
```

A more useful function would be one that took any number of numeric arguments and returned the average. Here's an easy way to write that function.

```
def avg(*args):
    return sum(args)/len(args)
```

Now we can call the function with a different number of arguments each time.

```
>>> avg(11, 22, 33)
22.0
>>> avg(1, 2)
1.5
```

The advantage of writing the function this way is that no brackets are needed when you call this function. The arguments are interpreted as if they were elements of a list, but you pass these arguments without list syntax.

What about the ordinary arguments we mentioned earlier? Additional arguments, not included in the list **`*args`**, must either precede **`*args`** in the argument list or be keyword arguments.

For example, let's revisit the `avg` example. Suppose we want a separate argument that specifies what units we're using. Because `units` is not a keyword argument, it must appear at the beginning of the list, in front of **`*args`**.

```
def avg(units, *args):
    print (sum(args)/len(args), units)
```

Here's a sample use:

```
>>> avg('inches', 11, 22, 33)
22.0 inches
```

This function is valid because the ordinary argument, `units`, precedes the argument list, **`*args`**.

Note ▶ The asterisk (**`*`**) has a number of uses in Python. In this context, it's called the *splat* or the *positional expansion* operator. Its basic use is to represent an "unpacked list"; more specifically, it replaces a list with a simple sequence of separate items.

The limitation on such an entity as **`*args`** is that there isn't much you can do with it. One thing you can do (which will be important in Section 4.9, "Decorators and Function Profilers") is pass it along to a function. Here's an example:

```
>>> ls = [1, 2, 3]  # Unpacked list.
>>> print(*ls)      # Print unpacked version
1 2 3
>>> print(ls)       # Print packed (ordinary list).
[1, 2, 3]
```

The other thing you can do with ***args** or `*ls` is to pack it (or rather, *repack* it) into a standard Python list; you do that by dropping the asterisk. At that point, it can be manipulated with all the standard list-handling abilities in Python.

Note

4.8.2 The "**kwargs" List

The more complete syntax supports keyword arguments, which are named arguments during a function call. For example, in the following call to the **print** function, the **end** and **sep** arguments are named.

```
print(10, 20, 30, end='.', sep=',')
```

The more complete function syntax recognizes both unnamed and named arguments.

Key Syntax

```
def func_name([ordinary_args,] *args, **kwargs):
    statements
```

As with the symbolic name **args**, the symbolic name **kwargs** can actually be any name, but by convention, Python programmers use **kwargs**.

Within the function definition, **kwargs** refers to a dictionary in which each key-value pair is a string containing a named argument (as the key) and a value, which is the argument value passed.

An example should clarify. Assume you define a function as follows:

```
def pr_named_vals(**kwargs):
    for k in kwargs:
        print(k, ':', kwargs[k])
```

This function cycles through the dictionary represented by **kwargs**, printing both the key values (corresponding to argument names) and the corresponding values, which have been passed to the arguments.

For example:

```
>>> pr_named_vals(a=10, b=20, c=30)
a : 10
b : 20
c : 30
```

A function definition may combine any number of named arguments, referred to by **kwargs**, with any number of arguments that are not named, referred to by **args**. Here is a function definition that does exactly that.

The following example defines such a function and then calls it.

```
def pr_vals_2(*args, **kwargs):
    for i in args:
        print(i)
    for k in kwargs:
        print(k, ':', kwargs[k])

pr_vals_2(1, 2, 3, -4, a=100, b=200)
```

This miniprogram, when run as a script, prints the following:

```
1
2
3
-4
a : 100
b : 200
```

Note ▶ Although **args** and **kwargs** are expanded into a list and a dictionary, respectively, these symbols can be passed along to another function, as shown in the next section.

◀ Note

4.9 Decorators and Function Profilers

When you start refining your Python programs, one of the most useful things to do is to time how fast individual functions run. You might want to know how many seconds and fractions of a second elapse while your program executes a function generating a thousand random numbers.

Decorated functions can profile the speed of your code, as well as provide other information, because functions are first-class objects. Central to the concept of decoration is a *wrapper* function, which does everything the original function does but also adds other statements to be executed.

Here's an example, illustrated by Figure 4.3. The decorator takes a function F1 as input and returns another function, F2, as output. This second function, F2, is produced by including a call to F1 but adding other statements as well. F2 is a wrapper function.

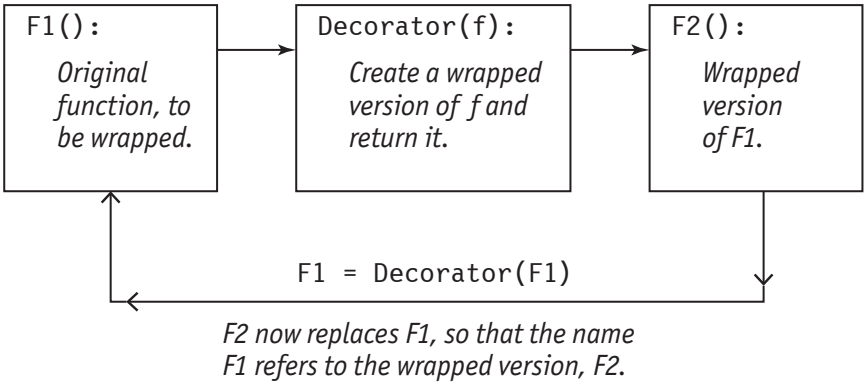

Figure 4.3. How decorators work (high-level view)

Here's an example of a decorator function that takes a function as argument and wraps it by adding calls to the **`time.time`** function. Note that **`time`** is a package, and it must be imported before **`time.time`** is called.

```
import time

def make_timer(func):
    def wrapper():
        t1 = time.time()
        ret_val = func()
        t2 = time.time()
        print('Time elapsed was', t2 - t1)
        return ret_val
    return wrapper
```

There are several functions involved with this simple example (which, by the way, is not yet complete!), so let's review.

- There is a function to be given as input; let's call this the *original* function (F1 in this case). We'd like to be able to input any function we want, and have it decorated—that is, acquire some additional statements.
- The *wrapper* function is the result of adding these additional statements to the original function. In this case, these added statements report the number of seconds the original function took to execute.
- The *decorator* is the function that performs the work of creating the wrapper function and returning it. The decorator is able to do this because it internally uses the **`def`** keyword to define a new function.

- Ultimately, the wrapped version is intended to replace the original version, as you'll see in this section. This is done by reassigning the function name.

If you look at this decorator function, you should notice it has an important omission: The arguments to the original function, *func*, are ignored. The wrapper function, as a result, will not correctly call *func* if arguments are involved.

The solution involves the ***args** and ****kwargs** language features, introduced in the previous section. Here's the full decorator:

```
import time

def make_timer(func):
    def wrapper(*args, **kwargs):
        t1 = time.time()
        ret_val = func(*args, **kwargs)
        t2 = time.time()
        print('Time elapsed was', t2 - t1)
        return ret_val
    return wrapper
```

The new function, remember, will be `wrapper`. It is `wrapper` (or rather, the function temporarily named `wrapper`) that will eventually be called in place of `func`; this wrapper function therefore must be able to take any number of arguments, including any number of keyword arguments. The correct action is to pass along all these arguments to the original function, `func`. Here's how:

```
        ret_val = func(*args, **kwargs)
```

Returning a value is also handled here; the wrapper returns the same value as `func`, as it should. What if `func` returns no value? That's not a problem, because Python functions return **None** by default. So the value **None**, in that case, is simply passed along. (You don't have to test for the existence of a return value; there always is one!)

Having defined this decorator, `make_timer`, we can take any function and produce a wrapped version of it. Then—and this is almost the final trick—we reassign the function name so that it refers to the wrapped version of the function.

```
def count_nums(n):
     for i in range(n):
        for j in range(1000):
            pass

count_nums = make_timer(count_nums)
```

The wrapper function produced by `make_timer` is defined as follows (except that the identifier `func` will be reassigned, as you'll see in a moment).

```
def wrapper(*args, **kwargs):
    t1 = time.time()
    ret_val = func(*args, **kwargs)
    t2 = time.time()
    print('Time elapsed was', t2 - t1)
    return ret_val
```

We now reassign the name `count_nums` so that it refers to *this* function—`wrapper`—which will call the original `count_nums` function but also does other things.

Confused yet? Admittedly, it's a brain twister at first. But all that's going on is that (1) a more elaborate version of the original function is being created at run time, and (2) this more elaborate version is what the name, `count_nums`, will hereafter refer to. Python symbols can refer to any object, including functions (callable objects). Therefore, we can reassign function names all we want.

```
count_nums = wrapper
```

Or, more accurately,

```
count_nums = make_timer(count_nums)
```

So now, when you run `count_nums` (which now refers to the wrapped version of the function), you'll get output like this, reporting execution time in seconds.

```
>>> count_nums(33000)
Time elapsed was 1.063697338104248
```

The original version of `count_nums` did nothing except do some counting; this wrapped version reports the passage of time in addition to calling the original version of `count_nums`.

As a final step, Python provides a small but convenient bit of syntax to automate the reassignment of the function name.


```
@decorator
def func(args):
    statements
```

This syntax is translated into the following:

```
def func(args):
    statements
func = decorator(func)
```

In either case, it's assumed that *decorator* is a function that has already been defined. This decorator must take a function as its argument and return a wrapped version of the function. Assuming all this has been done correctly, here's a complete example utilizing the **@** sign.

```
@make_timer
def count_nums(n):
     for i in range(n):
        for j in range(1000):
            pass
```

After this definition is executed by Python, `count_num` can then be called, and it will execute `count_num` as defined, but it will also add (as part of the wrapper) a **`print`** statement telling the number of elapsed seconds.

Remember that this part of the trick (the final trick, actually) is to get the name `count_nums` to refer to the *new* version of `count_nums`, after the new statements have been added through the process of decoration.

4.10 Generators

There's no subject in Python about which more confusion abounds than generators. It's not a difficult feature once you understand it. Explaining it's the hard part.

But first, what does a generator do? The answer: It enables you to deal with a sequence one element at a time.

Suppose you need to deal with a sequence of elements that would take a long time to produce if you had to store it all in memory at the same time. For example, you want to examine all the Fibonacci numbers up to 10 to the 50th power. It would take a lot of time and space to calculate the entire sequence. Or you may want to deal with an infinite sequence, such as all even numbers.

The advantage of a generator is that it enables you to deal with one member of a sequence at a time. This creates a kind of "virtual sequence."

4.10.1 What's an Iterator?

One of the central concepts in Python is that of *iterator* (sometimes confused with *iterable)*. An iterator is an object that produces a stream of values, one at a time.

All lists can be iterated, but not all iterators are lists. There are many functions, such as **reversed**, that produce iterators that are not lists. These cannot be indexed or printed in a useful way, at least not directly. Here's an example:

```
>>> iter1 = reversed([1, 2, 3, 4])
>>> print(iter1)
<list_reverseiterator object at 0x1111d7f28>
```

However, you can convert an iterator to a list and then print it, index it, or slice it:

```
>>> print(list(iter1))
[4, 3, 2, 1]
```

Iterators in Python work with **for** statements. For example, because `iter1` is an iterator, the following lines of code work perfectly well.

```
>>> iter1 = reversed([1, 2, 3, 4])
>>> for i in iter1:
      print(i, end='  ')

4  3  2  1
```

Iterators have *state information*; after reaching the end of its series, an iterator is exhausted. If we used `iter1` again without resetting it, it would produce no more values.

4.10.2 Introducing Generators

A generator is one of the easiest ways to produce an iterator. But the generator function is not itself an iterator. Here's the basic procedure.

- Write a generator function. You do this by using a **yield** statement anywhere in the definition.
- Call the function you completed in step 1 to get an iterator object.
- The iterator created in step 2 is what yields values in response to the **next** function. This object contains state information and can be reset as needed.

Figure 4.4 illustrates the process.

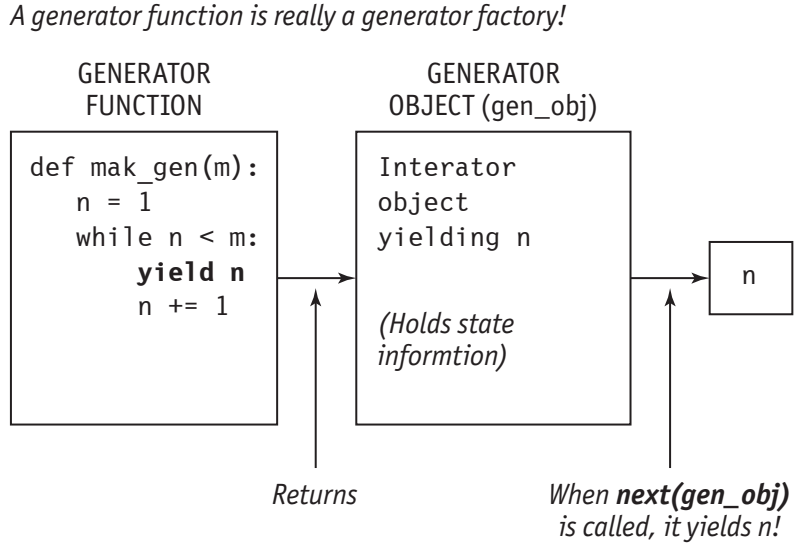

Figure 4.4. Returning a generator from a function

Here's what almost everybody gets wrong when trying to explain this process: It looks as if the **`yield`** statement, placed in the generator function (the thing on the left in Figure 4.4), is doing the yielding. That's "sort of" true, but it's not really what's going on.

The generator function defines the behavior of the iterator. But the iterator object, the thing to its right in Figure 4.4, is what actually executes this behavior.

When you include one or more **`yield`** statements in a function, the function is no longer an ordinary Python function; **`yield`** describes a behavior in which the function does not return a value but sends a value back to the caller of **`next`**. State information is saved, so when **`next`** is called again, the iterator advances to the next value in the series without starting over. This part, everyone seems to understand.

But—and this is where people get confused—it isn't the generator function that performs these actions, even though that's where the behavior is *defined*. Fortunately, you don't need to understand it; you just need to use it. Let's start with a function that prints even numbers from 2 to 10:

```
def print_evens():
    for n in range(2, 11, 2):
        print(n)
```

Now replace `print(n)` with the statement `yield n`. Doing so changes the nature of what the function does. While we're at it, let's change the name to `make_evens_gen` to have a more accurate description.

```
def make_evens_gen():
    for n in range(2, 11, 2):
        yield n
```

The first thing you might say is "This function no longer returns anything; instead, it yields the value `n`, suspending its execution and saving its internal state."

But this revised function, `make_evens_gen`, does indeed have a return value! As shown in Figure 4.4, the value returned is not `n`; the return value is an iterator object, also called a "generator object." Look what happens if you call `make_evens_gen` and examine the return value.

```
>>> make_evens_gen()
<generator object make_evens_gen at 0x1068bd410>
```

What did the function do? Yield a value for `n`? No! Instead, it returned an iterator object, and that's the object that yields a value. We can save the iterator object (or generator object) and then pass it to **`next`**.

```
>>> my_gen = make_evens_gen()
>>> next(my_gen)
2
>>> next(my_gen)
4
>>> next(my_gen)
6
```

Eventually, calling **`next`** exhausts the series, and a **`StopIteration`** exception is raised. But what if you want to reset the sequence of values to the beginning? Easy. You can do that by calling `make_evens_gen` again, producing a new instance of the iterator. This has the effect of starting over.

```
>>> my_gen = make_evens_gen()   # Start over
>>> next(my_gen)
2
>>> next(my_gen)
4
>>> next(my_gen)
6
>>> my_gen = make_evens_gen()   # Start over
>>> next(my_gen)
2
>>> next(my_gen)
4
>>> next(my_gen)
6
```

What happens if you call `make_evens_gen` every time? In that case, you keep starting over, because each time you're creating a new generator object. This is most certainly not what you want.

```
>>> next(make_evens_gen())
2
>>> next(make_evens_gen())
2
>>> next(make_evens_gen())
2
```

Generators can be used in **`for`** statements, and that's one of the most frequent uses. For example, we can call `make_evens_gen` as follows:

```
for i in make_evens_gen():
    print(i, end='  ')
```

This block of code produces the result you'd expect:

```
2  4  6  8  10
```

But let's take a look at what's really happening. The **`for`** block calls `make_evens_gen` one time. The result of the call is to get a generator object. That object then provides the values in the **`for`** loop. The same effect is achieved by the following code, which breaks the function call onto an earlier line.

```
>>> my_gen = make_evens_gen()
>>> for i in my_gen:
      print(i, end='  ')
```

Remember that `my_gen` is an iterator object. If you instead referred to `make_evens_gen` directly, Python would raise an exception.

```
for i in make_evens_gen:       # ERROR! Not an iterable!
    print(i, end='  ')
```

Once you understand that the object returned by the generator function is the generator object, also called the iterator, you can call it anywhere an *`iterable`* or *`iterator`* is accepted in the syntax. For example, you can convert a generator object to a list, as follows.

```
>>> my_gen = make_evens_gen()
>>> a_list = list(my_gen)
>>> a_list
[2, 4, 6, 8, 10]
```

```
>>> a_list = list(my_gen)       # Oops! No reset!
>>> a_list
[]
```

The problem with the last few statements in this example is that each time you iterate through a sequence using a generator object, the iteration is exhausted and needs to be reset.

```
>>> my_gen = make_evens_gen()   # Reset!
>>> a_list = list(my_gen)
>>> a_list
[2, 4, 6, 8, 10]
```

You can of course combine the function call and the **list** conversion. The list itself is stable and (unlike a generator object) will retain its values.

```
>>> a_list = list(make_evens_gen())
>>> a_list
[2, 4, 6, 8, 10]
```

One of the most practical uses of an iterator is with the **in** and **not in** keywords. We can, for example, generate an iterator that produces Fibonacci numbers up to and including N, but not larger than N.

```
def make_fibo_gen(n):
    a, b = 1, 1
    while a <= n:
        yield a
        a, b = a + b, a
```

The **yield** statement changes this function from an ordinary function to a generator function, so it returns a generator object (iterator). We can now determine whether a number is a Fibonacci by using the following test:

```
n = int(input('Enter number: '))
if n in make_fibo_gen(n):
    print('number is a Fibonacci. ')
else:
    print('number is not a Fibonacci. ')
```

This example works because the iterator produced does not yield an infinite sequence, something that would cause a problem. Instead, the iterator terminates if `n` is reached without being confirmed as a Fibonacci.

Remember—and we state this one last time—by putting **yield** into the function `make_fibo_gen`, it becomes a generator function and it returns the

generator object we need. The previous example could have been written as follows, so that the function call is made in a separate statement. The effect is the same.

```
n = int(input('Enter number: '))
my_fibo_gen = make_fibo_gen(n)
if n in my_fibo_gen:
    print('number is a Fibonacci. ')
else:
    print('number is not a Fibonacci. ')
```

As always, remember that a generator function (which contains the **yield** statement) is not a generator object at all, but rather a generator factory. This is confusing, but you just have to get used to it. In any case, Figure 4.4 shows what's really going on, and you should refer to it often.

4.11 Accessing Command-Line Arguments

Running a program from the command lets you provide the program an extra degree of flexibility. You can let the user specify *command-line arguments*; these are optional arguments that give information directly to the program on start-up. Alternatively, you can let the program prompt the user for the information needed. But use of command-line arguments is typically more efficient.

Command-line arguments are always stored in the form of strings. So—just as with data returned by the input function—you may need to convert this string data to numeric format.

To access command-line arguments from within a Python program, first import the **sys** package.

```
import sys
```

You can then refer to the full set of command-line arguments, including the function name itself, by referring to a list named **argv**.


```
argv       # If 'import sys.argv' used
sys.argv   # If sys imported as 'import sys'
```

In either case, **argv** refers to a list of command-line arguments, all stored as strings. The first element in the list is always the name of the program itself. That element is indexed as **argv[0]**, because Python uses zero-based indexing.

For example, suppose that you are running `quad` (a quadratic-equation evaluator) and input the following command line:

```
python quad.py -1 -1 1
```

In this case, **`argv`** will be realized as a list of four strings.

Figure 4.5 illustrates how these strings are stored, emphasizing that the first element, **`argv[0]`**, refers to a string containing the program name.

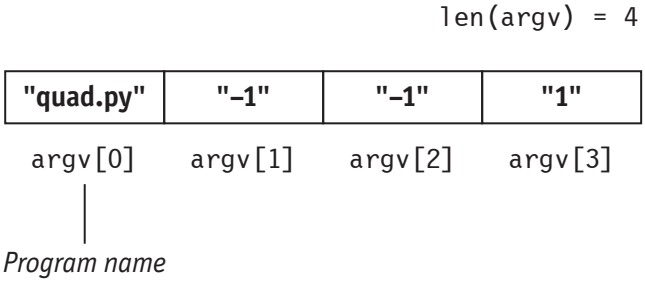

Figure 4.5. Command-line arguments and argv

In most cases, you'll probably ignore the program name and focus on the other arguments. For example, here is a program named `silly.py` that does nothing but print all the arguments given to it, including the program name.

```
import sys
for thing in sys.argv:
    print(thing, end='  ')
```

Now suppose we enter this command line:

```
python  silly.py  arg1  arg2  arg3
```

The Terminal program (in Mac) or the DOS Box prints the following:

```
silly.py  arg1  arg2  arg3
```

The following example gives a more sophisticated way to use these strings, by converting them to floating-point format and passing the numbers to the `quad` function.

```
import sys

def quad(a, b, c):
    '''Quadratic Formula function.'''

    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
```

```
    x2 = (-b - determin) / (2 * a)
    return x1, x2

def main():
    '''Get argument values, convert, call quad.'''

    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
    a, b, c = float(s1), float(s2), float(s3)
    x1, x2 = quad(a, b, c)
    print('x values: {}, {}.'.format(x1, x2))

main()
```

The interesting line here is this one:

```
    s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
```

Again, the **`sys.argv`** list is zero-based, like any other Python list, but the program name, referred to as **`sys.arg[0]`**, typically isn't used in the program code. Presumably you already know what the name of your program is, so you don't need to look it up.

Of course, from within the program you can't always be sure that argument values were specified on the command line. If they were not specified, you may want to provide an alternative, such as prompting the user for these same values.

Remember that the length of the argument list is always N+1, where N is the number of command-line arguments—beyond the program name, of course.

Therefore, we could revise the previous example as follows:

```
import sys

def quad(a, b, c):
    '''Quadratic Formula function.'''

    determin = (b * b - 4 * a * c) ** .5
    x1 = (-b + determin) / (2 * a)
    x2 = (-b - determin) / (2 * a)
    return x1, x2

def main():
    '''Get argument values, convert, call quad.'''
```

```
    if len(sys.argv) > 3:
        s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
    else:
        s1 = input('Enter a: ')
        s2 = input('Enter b: ')
        s3 = input('Enter c: ')
    a, b, c = float(s1), float(s2), float(s3)
    x1, x2 = quad(a, b, c)
    print('x values: {}, {}.'.format(x1, x2))

main()
```

The key lines in this version are in the following **`if`** statement:

```
    if len(sys.argv) > 3:
        s1, s2, s3 = sys.argv[1], sys.argv[2], sys.argv[3]
    else:
        s1 = input('Enter a: ')
        s2 = input('Enter b: ')
        s3 = input('Enter c: ')
    a, b, c = float(s1), float(s2), float(s3)
```

If there are at least four elements in **`sys.argv`** (and therefore three command-line arguments beyond the program name itself), the program uses those strings. Otherwise, the program prompts for the values.

So, from the command line, you'll be able to run the following:

```
python quad.py 1 -9 20
```

The program then prints these results:

```
x values: 4.0 5.0
```

Chapter 4 *Summary*

A large part of this chapter presented ways to improve your efficiency through writing better and more efficient Python code. Beyond that, you can make your Python programs run faster if you call the **`print`** function as rarely as possible from within IDLE—or else run programs from the command line only.

A technique helpful in making your code more efficient is to profile it by using the **`time`** and **`datetime`** packages to compute the relative speed of the code, given different algorithms. Writing decorators is helpful in this respect, because you can use them to profile function performance.

One of the best ways of supercharging your applications, in many cases, is to use one of the many free packages available for use with Python. Some of these are built in; others, like the **numpy** package, you'll need to download.

Chapter 4 *Questions for Review*

1. Is an assignment operator such as **+=** only a convenience? Can it actually result in faster performance at run time?
2. In most computer languages, what is the minimum number of statements you'd need to write instead of the Python statement `a, b = a + b, a`?
3. What's the most efficient way to initialize a list of 100 integers to 0 in Python?
4. What's the most efficient way of initializing a list of 99 integers with the pattern `1, 2, 3` repeated? Show precisely how to do that, if possible.
5. If you're running a Python program from within IDLE, describe how to most efficiently print a multidimensional list.
6. Can you use list comprehension on a string? If so, how?
7. How can you get help on a user-written Python program from the command line? From within IDLE?
8. Functions are said to be "first-class objects" in Python but not in most other languages, such as C++ or Java. What is something you can do with a Python function (callable object) that you cannot do in C or C++?
9. What's the difference between a wrapper, a wrapped function, and a decorator?
10. When a function is a generator function, what does it return, if anything?
11. From the standpoint of the Python language, what is the one change that needs to be made to a function to turn it into a generator function?
12. Name at least one advantage of generators.

Chapter 4 *Suggested Problems*

1. Print a matrix of 20 × 20 stars or asterisks (*****). From within IDLE, demonstrate the slowest possible means of doing this task and the fastest possible means. (Hint: Does the fastest way utilize string concatenation of the **join**

method?) Compare and contrast. Then use a decorator to profile the speeds of the two ways of printing the asterisks.

2 Write a generator to print all the perfect squares of integers, up to a specified limit. Then write a function to determine whether an integer argument is a perfect square if it falls into this sequence—that is, if `n` is an integer argument, the phrase `n in square_iter(n)` should yield **`True`** or **`False`**.

This page intentionally left blank

Index

Symbols

B

C

D

E

F

G

H

I

M

N

Q

R

S

T

U

V

W

X

Y

Z