Programming
in
Python

Eric S. Roberts
Charles Simonyi Professor of CS, *emeritus,* Stanford University
University Professor, Willamette University

CS 151
Willamette University
Spring 2023

Contents

1 Introducing Python 1

1.1 Data and types 2
1.2 Numeric data 3
1.3 Variables and assignment 8
1.4 Functions 12
1.5 Nonnumeric data 17
1.6 Writing Python programs 23
Summary 27
Review questions 29
Exercises 32

2 Control Statements 35

2.1 Boolean data 37
2.2 The `if` statement 43
2.3 The `while` statement 44
2.4 The `for` statement 50
2.5 The `assert` statement 58
Summary 60
Review questions 62
Exercises 63

3 Algorithmic Thinking 65

3.1 Algorithms in history 67
3.2 Devising your own algorithms 74
3.3 Testing and debugging 81
Summary 90
Review questions 92
Exercises 93

4 Simple Graphics 97

4.1 Your first graphics program 98
4.2 Classes, objects, and methods 99
4.3 Graphical objects 103
4.4 The graphics window 113
4.5 Creating graphical applications 115
4.6 Decomposition 117
4.7 Control structures and graphics 122
4.8 Functions that return graphical objects 124
Summary 125
Review questions 127
Exercises 128

5 Functions 137

5.1 A quick review of functions 138
5.2 The mechanics of function calls 141
5.3 Libraries and interfaces 146
5.4 The random library 148
5.5 Creating your own libraries 155
5.6 Inner functions 157
5.7 Introduction to recursion 159
Summary 169
Review questions 171
Exercises 172

6 Writing Interactive Programs 177

6.1 First-class functions 178
6.2 A simple interactive example 181
6.3 Controlling properties of objects 183
6.4 Responding to mouse events 189
6.5 Timer-based animation 192
6.6 Expanding the graphics library 196
Summary 204
Review questions 206
Exercises 207

7 Strings 217

7.1 Binary representation 218

7.2 String functions and operators 225

7.3 Common string patterns 229

7.4 String methods 231

7.5 Building string applications 234

7.6 Formatting strings 242

Summary 246

Review questions 247

Exercises 249

8 Lists 257

8.1 Introduction to arrays and lists 258

8.2 List methods 262

8.3 List comprehensions 266

8.4 Using lists for tabulation 268

8.5 Using files 272

8.6 Multidimensional arrays 280

8.7 Image processing 281

Summary 288

Review questions 289

Exercises 290

9 Searching and Sorting 297

9.1 Searching 298

9.2 Simple strategies for sorting 302

9.3 Computational complexity 306

9.4 Divide-and-conquer strategies 312

9.5 Standard complexity classes 317

9.6 The Quicksort algorithm 320

9.7 A formal definition of big-O 326

Summary 328

Review questions 329

Exercises 331

10 Classes and Objects 337
10.1 Records and tuples 338
10.2 Representing points 347
10.3 Rational numbers 352
10.4 Operator overloading 357
10.5 Implementing a token scanner 361
Summary 358
Review questions 369
Exercises 370

11 Dictionaries and Sets 375
11.1 Dictionaries 376
11.2 Using dictionaries as records 385
11.3 Designing an efficient dictionary 386
11.4 Sets 398
Summary 404
Review questions 406
Exercises 407

12 Designing Data Structures 415
12.1 Abstract data types 416
12.2 Representing real-world data 417
12.3 Data-driven programs 425
Summary 438
Review questions 439
Exercises 439

13 Inheritance 445
13.1 Class hierarchies 446
13.2 Defining an employee hierarchy 449
13.3 Extending the graphics classes 453
13.4 Decomposition and inheritance 456
13.5 Unit testing 461
13.6 Deciding when to use inheritance 464
Summary 466
Review questions 467
Exercises 469

CHAPTER 1

Introducing Python

It’s my belief that Python is a lot easier [to teach than] C or C++ or Java . . . because all the details of the languages are so much harder.

—Guido van Rossum

Guido van Rossum (1956–)

Guido van Rossum is a Dutch programmer best known as the inventor of the Python programming language. Van Rossum has overseen the development of Python from its genesis in 1989 as a “hobby project” through almost three decades of growing popularity. Van Rossum holds a Master’s degree in mathematics and computer science from the University of Amsterdam. His career includes positions at prestigious research labs, including the Centrum Wiskunde and Informatica in Amsterdam and the National Institute of Standards and Technology in Washington, as well as at leading industrial corporations such as Google and Dropbox.

Before you can appreciate the power of computing, you need to learn at least the basics of a programming language. The programs in this text use a programming language called ***Python,*** which was designed and implemented by Guido van Rossum. Van Rossum explains that the name is a result of "being in a slightly irreverent mood (and [being] a big fan of *Monty Python's Flying Circus*)."

Since its initial release in 1991, Python has since become one of the most popular programming languages in use today, both in industry and academia. In particular, Python is now the most common programming language in introductory computer science courses.

Van Rossum explicitly designed Python to be easy to teach, as demonstrated by the following design goals, which he included in a 1999 proposal entitled "Computer Programming for Everybody":

- An easy and intuitive language just as powerful as major competitors
- Open source, so anyone can contribute to its development
- Code that is as understandable as plain English
- Suitability for everyday tasks, allowing for short development times

Although this text uses Python as its programming language, its focus is not on the language itself but on the programs that you write using that language. It does not try to cover all of Python and deliberately avoids some of its more exotic features. Even so, the subset of Python it describes will give you the tools you need to write exciting applications that use the best features of the Python language.

1.1 Data and types

For much of their history, computing machines—even before the age of modern computing—have worked primarily with numeric data. The computers built in the mid 1960s were so closely tied to processing numeric data that they earned the nickname ***number crunchers*** as a result. Information, however, comes in many forms, and computers are increasingly good at working with data of many different types. When you write programs that count or add things up, you are working with ***numeric data.*** When you write programs that manipulate characters—typically assembled into larger units such as words, sentences, and paragraphs—you are working with ***string data.*** You will learn about these and many other data types as you progress through this book.

In computer science, a data type is defined by two properties: a domain and a set of operations. The ***domain*** is simply the set of values that are elements of that type. For numeric data, the domain consists of numbers like 0, 42, –273, and 3.14159265.

For string data, the domain comprises sequences of characters that appear on the keyboard or that can be displayed on the screen. The ***set of operations*** is the toolbox that allows you to manipulate values of that type. For numeric data, the set of operations includes addition, subtraction, multiplication, and division, along with a variety of more sophisticated functions. For string data, however, it is hard to imagine what an operation like subtraction might mean. Using string data requires a different set of operations, such as combining two strings to form a longer one or comparing two strings to see if they are in alphabetic order. The general rule is that the set of operations must be appropriate to the elements of the domain. The two components together—the domain and the operations—define a ***data type.***

1.2 Numeric data

Computers today store data in so many exciting forms that numbers may seem a bit boring. Even so, numbers are a good starting point for talking about data, mostly because they are both simple and familiar. You've been using numbers, after all, ever since you learned to count. Moreover, as you'll discover in Chapter 7, all information is represented inside the computer in numeric form.

Representing numbers in Python

One of the important design principles of modern programming languages is that concepts that are familiar to human readers should be expressed in an easily recognizable form. Like most languages, Python adopts that principle for numeric representation, which means that you can write numbers in a Python program in much the same way you would write them anywhere else.

In their most common form, numbers consist of a sequence of digits, optionally containing a decimal point. Negative numbers are preceded by a minus sign. For example, the following are all legal numbers in Python:

```
0    42    -273    3.14159265    -0.5    1000000
```

Note that large numbers, such as the value of one million shown in the last example, are written without using commas to separate the digits into groups of three.

Numbers can also be written in a variant of scientific notation, in which the value is represented as a number multiplied by a power of 10. To express a value in scientific notation, you write a number in standard decimal notation, followed immediately by the letter `E` and an integer exponent, optionally preceded by a `+` or `–` sign. For example, the speed of light is approximately 2.9979 × 10^8 meters per second, which can be written in Python as

```
2.9979E+8
```

In Python's scientific notation, the letter `E` is shorthand for *times 10 to the power.*

Like most languages, Python separates numbers into two classes: ***integers,*** which represent whole numbers, and ***floating-point*** numbers, which contain a decimal point. Integers have the advantage of being exact. Floating-point numbers, by contrast, are approximations whose accuracy is determined by hardware limitations. Fortunately, Python also defines its mathematical operators in a way that makes it less important than it is in most languages to pay attention to the distinction between these two types of numbers.

In addition to integers and floating-point numbers, Python defines a third type of numeric data used to represent ***complex numbers,*** which combine a real component and an imaginary component corresponding to the square root of –1. Although complex numbers are beyond the scope of this text, the fact that Python includes complex numbers as a fully supported, built-in type makes Python especially attractive for scientific and mathematical applications in which complex numbers play an important role.

Arithmetic expressions

The real power of numeric data comes from the fact that Python allows you to perform computation by applying mathematical operations, ranging in complexity from addition and subtraction up to highly sophisticated mathematical functions. As in mathematics, Python allows you to express those calculations through the use of operators, such as + and – for addition and subtraction.

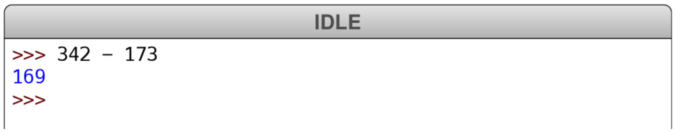
If you want to understand how Python works, the best approach is to use the Python interpreter, which is called IDLE. (Van Rossum claims that the name is an acronym of Integrated DeveLopment Environment, but the common assumption is that the name honors Monty Python's Eric Idle.) IDLE allows you to enter Python expressions and see what values they produce

To get a sense of how interactions with IDLE work, suppose that you want to solve the following problem, which the singer-songwriter, political satirist, and mathematician Tom Lehrer proposed in his song "New Math" in 1965:

Tom Lehrer

$$\begin{array}{r} 3\,4\,2 \\ -\,1\,7\,3 \\ \hline \end{array}$$

To find the answer, all you have to do is enter the subtraction into IDLE, as follows:



```
>>> 342 – 173
169
>>>
```

This computation is an example of an ***arithmetic expression,*** which consists of a sequence of values called ***terms*** combined using symbols called ***operators,*** most of which are familiar from elementary-school arithmetic. The arithmetic operators in Python include the following:

$-a$	Negation (multiply a by –1 to reverse its sign)
$a + b$	Addition (add a and b)
$a - b$	Subtraction (subtract b from a)
$a * b$	Multiplication (multiply a and b)
$a \,/\, b$	True division (divide a by b)
$a \,//\, b$	Floor division ($a \,/\, b$ rounded down to the next integer)
$a \,\%\, b$	Remainder (compute the mathematical result of a mod b)
$a ** b$	Exponentiation (raise a to the b power)

Although most of these operators should be familiar from basic arithmetic, the // and % operators require additional explanation. Intuitively, these operators compute the quotient and remainder, respectively, when one value divided by another. For example, 7 // 3 has the value 2, because 7 divided by 3 leaves a whole number quotient of 2. Similarly, 7 % 3 has the value 1, because 7 divided by 3 leaves a remainder of 1. If one number is evenly divisible by another, there is no remainder, so that, for example, 12 % 4 has the value 0.

Unlike almost every other programming language, Python defines // and % for negative operands so that the result is consistent with mathematical convention. The // operator computes the result by performing an exact division and then rounding the result down to the next smaller integer. In mathematics, rounding a number down to the closest integer is called computing its ***floor.*** For example, the expression -9 // 5 has the value –2, because exact division produces –1.8, and the floor of –1.8 is –2. In computing the remainder, the % operator applies what mathematicians call the ***mod*** operator, which always has the same sign as the divisor. The // and % operators are related by the following equivalence:

$$x \equiv (x \,//\, y) \times y + x \,\%\, y$$

Even though Python's definition of these operators makes mathematicians happy, the programs in this text use the // and % operators only with positive integers, where the result corresponds to the notions of quotient and remainder that you learned in elementary school. In part, the reason for this design decision is to avoid making programming seem more mathematical than it in fact is. In addition, it is dangerous to rely on how these operators behave with negative numbers because Python's definition—although it is clearly correct in mathematical terms—differs from how remainders are defined in other languages. If you write a Python program that relies on this behavior, it will be hard to translate that program into a language that uses a different interpretation.

Mixing types in an expression

Python allows you to mix integers and floating-point numbers freely in an expression. If you do so, the type of the result depends both on the operator and the types of the values to which it applies, which are called its ***operands.*** For almost all of Python's operators, the result is an integer if both operands are integers and a floating-point number if either or both of its operands is floating-point. Thus, evaluating the expression

```
17 + 25
```

produces the integer 42. By contrast, the expression

```
7.5 - 4.5
```

produces the floating-point value 3.0, even though the result is a whole number.

There are two exceptions to Python's standard rule for combining types. The / operator, which performs exact division, always returns a floating-point result, even if both operands are integers. The ** operator is a bit more complicated. The result is an integer if the left operand is an integer and the right operand is a nonnegative integer. In any other case, the result is a floating-point value. For example, the expression

```
2 ** 10
```

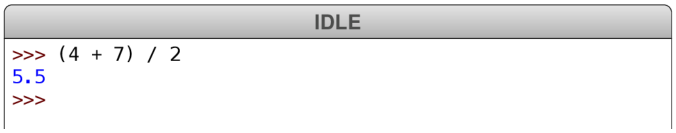
calculates 2^{10} and therefore produces the integer 1024. The expression

```
2 ** -1
```

calculates 2^{-1}, which is the floating-point number 0.5.

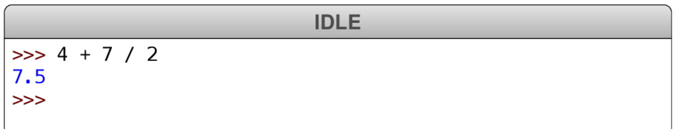
Precedence

Following the conventions of standard mathematics, multiplication, division, and remainder are performed before addition and subtraction, although you can use parentheses to change the evaluation order. For example, if you want to average the numbers 4 and 7, you can enter the following expression into IDLE:



```
>>> (4 + 7) / 2
5.5
>>>
```

If you leave out the parentheses, Python first divides 7 by 2 and then adds 4 and 3.5 to produce the value 7.5, as follows:



```
>>> 4 + 7 / 2
7.5
>>>
```

The order in which Python evaluates the operators in an expression is governed by their ***precedence,*** which is a measure of how tightly each operator binds to its operands. If two operators compete for the same operand, the one with higher precedence is applied first. If two operators have the same precedence, they are applied from left to right. The only exception is the exponentiation operator **, which is applied from right to left. Computer scientists use the term ***associativity*** to indicate whether an operator groups to the left or to the right. Most operators in Python are ***left-associative,*** which means that the leftmost operator is evaluated first. In Python, the only exception to this rule is the ** operator, which is ***right-associative*** and groups from right to left.

Figure 1-1 shows a complete precedence table for the Python operators, many of which you will have little or no occasion to use. As additional operators are introduced in this book, you can look them up in this table to see where they fit in the precedence hierarchy. Since the purpose of the precedence rules is to ensure that Python expressions obey the same rules as their mathematical counterparts, you can usually rely on your intuition. Moreover, if you are ever in any doubt, you can always include parentheses to make the order of operations explicit.

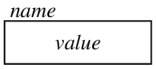
FIGURE 1-1 **Complete precedence table for the Python operators**

Operators in decreasing order of precedence
**
unary operators: + − ~
* / // %
+ −
<< >>
&
^
|
== != > >= < <= is is not in not in
not
and
or

1.3 Variables and assignment

When you write a program that works with data values, it is often convenient to use names to refer to a value that can change as the program runs. In programming, names that refer to values are called ***variables.***

Every variable in Python has two attributes: a *name* and a *value*. To understand the relationship of these attributes, it is best to think of a variable as a box with a label attached to the outside, like this:



The name of the variable appears on the label and is used to tell different boxes apart. If you have three variables in a program, each variable will have a different name. The value corresponds to the contents of the box. The name of the box is fixed, but you can change the value as often as you like.

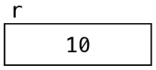
You create a new variable in Python by assigning it a value in the context of an ***assignment statement,*** which has the following form:

```
name = value
```

For example, if you execute the assignment statement

```
r = 10
```

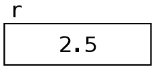
Python will create a new variable named `r` and assign it the value 10, as follows:



As the word *variable* implies, the value of a variable is not fixed but can change over the course of a program. For example, if you at some later point in a program execute the assignment statement

```
r = 2.5
```

the value in the box will change as follows:



The value that appears to the right of the equal sign in an assignment statement can be any Python expression. For example, you can compute the average of the numbers 3, 4, and 5 using the following statement:

```
average = (3 + 4 + 5) / 3
```

Shorthand assignment

Assignment statements are often used to modify the current value of a variable. For example, you can add the value of `deposit` to `balance` using the statement

```
balance = balance + deposit
```

This statement takes the current value of `balance`, adds the value of `deposit`, and then stores the result back in `balance`. Assignment statements of this form are so common that Python allows you to use the following shorthand:

```
balance += deposit
```

Similarly, you can subtract the value of `surcharge` from `balance` by writing

```
balance -= surcharge
```

More generally, the Python statement

```
variable op= expression
```

is equivalent to

```
variable = variable op (expression)
```

The parentheses are included in this pattern to emphasize that the expression is evaluated before *op* is applied. Such statements are called ***shorthand assignments.***

Multiple assignment

In Python, both the left and right sides of an assignment statement can be lists separated by commas. An assignment statement involving more than one value is called a ***multiple assignment statement.*** The left side of a multiple assignment statement is ordinarily a list of variables, and the right side is a list of expressions with the same number of elements.

When Python encounters a multiple assignment statement, it assigns the value of the first expression to the first variable, the value of the second expression to the second variable, and so on. For example, the following is a legal assignment statement in Python and has the effect of setting the variables `a`, `b`, and `c` to the values 3, 4, and 5, respectively:

```
a, b, c = 3, 4, 5
```

Python evaluates all the expressions on the right side of a multiple assignment statement before it assigns any of the values. For example, the statement

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

has the effect of exchanging the values of the variables `x` and `y`. Without multiple assignment, exchanging the value of two variables requires storing one of the values in a temporary variable so that its value is not lost when you perform the first assignment. To achieve the same effect using standard assignment would therefore require the following code:

```
tmp = x
x = y
y = tmp
```

Naming conventions

The names used for variables, functions, and so forth are collectively known as ***identifiers.*** In Python, the rules for identifier formation are

1. The identifier must start with a letter or an underscore (_).
2. All other characters must be letters, digits, or underscores.
3. The identifier must not be one of the reserved keywords listed in 1-2.

Uppercase and lowercase letters appearing in an identifier are considered to be different. Thus, the identifier `ABC` is not the same as the identifier `abc`.

You can make your programs more readable by using variable names that immediately suggest the meaning of that variable. If `r`, for example, refers to the radius of a circle, that name makes sense because it follows standard mathematical convention. In most cases, however, it is better to use longer names that make it clear to anyone reading your program exactly what value a variable contains. For example, if you need a variable to keep track of the number of pages in a document, it is better to use a name like `number_of_pages` than an abbreviated form like `np`.

When you use a name that consists of several English words, it is useful to adopt some convention for marking the word divisions. The official Python style guide recommends names like `number_of_pages` in which the individual words are

FIGURE 1-2 Reserved words in Python

```
False
None
True
and
as
assert
async
await
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield
```

separated by underscores. This style of separating words is called ***snake case,*** presumably because the underscores lie flat on the baseline of the text. In accordance with Python's style guidelines, this text uses snake case for the names of functions, variables, and methods.

For the names of classes, which are introduced beginning in Chapter 4, Python uses a different strategy for marking word boundaries. This strategy is called ***camel case,*** which marks word boundaries by using an uppercase letter at the beginning of each embedded word. The name *camel case* comes from the fact that this style creates humps in the middle of an identifier name. For example, one of the classes introduced in Chapter 12 is called `TokenScanner`, in which the division between `Token` and `Scanner` is marked using capitalization. This book also uses camel case for the names of files, such as the `AddTwoIntegers.py` program file introduced later in this chapter. Camel case is widely used in languages other than Python, so it is important to become familiar with this style.

Constants

In addition to choosing meaningful variable names, you can make your programs more readable by giving names to values that do not change as a program runs. Such values are called ***constants.*** By convention, the names used to designate constant values are written entirely in uppercase using underscores to indicate word boundaries. For example, you can use the statement

```
SPEED_OF_LIGHT = 2.9979E+8
```

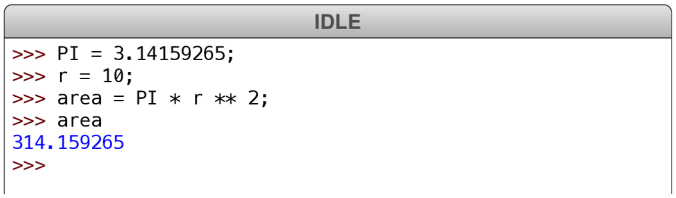
to assign a name to the specified value, which otherwise might be difficult for people reading your program to recognize as the speed of light.

Unlike most modern languages, Python does not allow you to specify that the value of a variable like `SPEED_OF_LIGHT` cannot be changed. The idea that this value is a constant is instead a matter of convention. Constant names in Python are written entirely in uppercase, adding underscores to indicate word boundaries.

Although it violates the spirit of constants to change their values while a program is running, it is perfectly appropriate to use constants for values that you might want to change over the development cycle of an application. The value of using constants for this purpose is discussed in more detail in section 1.8.

Sequential calculations

The ability to define variables and constants makes arithmetic calculations easier to follow, even in the IDLE interpreter. The following sequence of statements, for example, compute an approximation of the area of a circle of radius 10, which is accurate to the number of digits specified for the constant `PI`:



```
>>> PI = 3.14159265;
>>> r = 10;
>>> area = PI * r ** 2;
>>> area
314.159265
>>>
```

1.4 Functions

One of the most powerful features of any programming language is the ability to define a ***function,*** which is a sequence of statements collected together under a single name. Defining a function frees you from having to repeat the individual statements each time you want to perform that computation. You instead specify the name of the function, which invokes the entire sequence.

In computer science, the act of invoking a function by its name is referred to as ***calling the function.*** As part of that operation, the caller can supply information in the form of ***arguments,*** which are expressions computed at the time of the call. The arguments to a function are enclosed in parentheses and appear after the function name. Inside the function, each of the arguments is assigned to a variable called a ***parameter.*** The function uses these parameters to compute a result, which is delivered back to the caller. This process is called ***returning a result.***

The term *function* is intended to evoke the similar concept in mathematics. As in a programming language, functions in mathematics take data values enclosed in parentheses and compute a result, which is typically written in the form of a mathematical expression. As an example, the mathematical function

$$f(x) = x^2 - 5$$

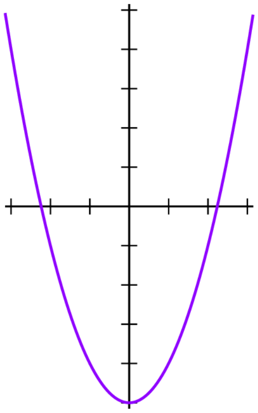
expresses a relationship between the value of x and the value of the function. To find the value of the function for a particular value of x, all you need to do is substitute that value in for the variable x in the function definition. For example, you can determine that the value of $f(0)$ is –5 by substituting 0 for x in the function definition, as follows:

$$f(0) = 0^2 - 5 = -5$$

Similarly, $f(3)$ has the value 4:

$$f(3) = 3^2 - 5 = 4$$

In mathematics, functions are often represented using a graph that shows how the value of a function changes with respect to the value of x. The graph for the function f appears in the left margin.



$f(x) = x^2 - 5$

Defining functions

```
def name(parameters):
    statements
```

In Python, a function definition takes the form shown inside the shaded pattern in the right margin, which is called a ***syntax box.*** Boldface text in a syntax box represents the fixed portion of the pattern, which will always appear in precisely that form. Italic text in a syntax box indicates the parts you can change for a particular instance of that pattern. In this case, for example, a function must always begin with the keyword `def` and include the parentheses and colon shown in the example. When you define a new function, you are free to choose the name, the list of parameters, and the statements that appear on subsequent lines. The first line of a syntactic pattern is called the ***header line.*** The statements that appear after the header line are called the ***body.***

Python determines the extent of the body of a function using the indentation of the code. Each statement in the body is indented four spaces with respect to the header line. If statements within the body have their own contents, as you will discover in Chapter 2, those statements use additional indentation to reflect the hierarchical structure of the code.

```
return exp
```

Most functions will also include one or more `return` statements that specify the value returned to the caller. This statement has the form shown on the right, where *exp* can be any Python expression. When Python executes a `return` statement, it evaluates the expression and then returns to the point at which the function was called, substituting the computed value into the calling function.

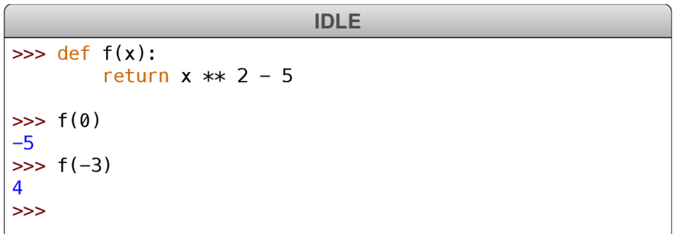
Simple function examples

The details of defining a function are best introduced through examples. The mathematical function $f(x) = x^2 - 5$ from page 12 has the following form in Python:

```
def f(x):
    return x ** 2 - 5
```

In this definition, `x` is the parameter variable, which is set by the argument passed by the caller. For example, if you were to call `f(2)`, the variable `x` would be set to the value 2. The `return` statement specifies the computation needed to calculate the result. Squaring `x` gives the value 4; subtracting 5 gives the final result of –1, which is passed back to the caller.

You can define a function directly in the IDLE interpreter by typing its definition and then entering a blank line to show that the function definition is complete. Once you have defined the function, you can call it from IDLE by writing its name and supplying the desired arguments, as shown in the following console session:



```
>>> def f(x):
        return x ** 2 - 5

>>> f(0)
-5
>>> f(-3)
4
>>>
```

Parameter variables and variables introduced in an assignment statement are accessible only inside the function in which they appear. For this reason, those variables are called ***local variables.*** By contrast, variables defined outside of a function are ***global variables,*** which can be used anywhere in the program. As programs get larger, using global variables makes those programs more difficult to read and maintain. The programs in this book therefore avoid using any global variables except for constants. Thus, it is reasonable to define `SPEED_OF_LIGHT` as a global constant, but variables whose values might change should always be local.

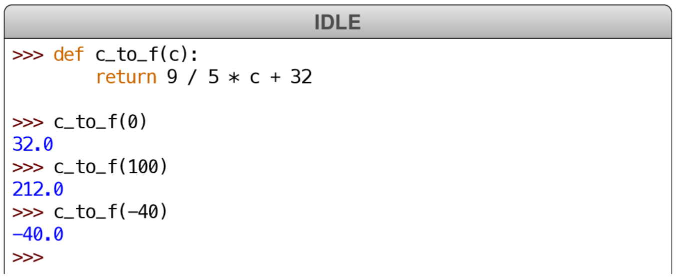
You can use functions to compute values that come up in practical situations that are largely outside of traditional mathematics. For example, if you travel outside the United States, you will discover that the rest of the world measures temperatures in Celsius rather than Fahrenheit. The formula to convert a Celsius temperature to its Fahrenheit equivalent is

$$F = \frac{9}{5}C + 32$$

which you can easily translate into the following Python function:

```
def c_to_f (c):
   return 9 / 5 * c + 32
```

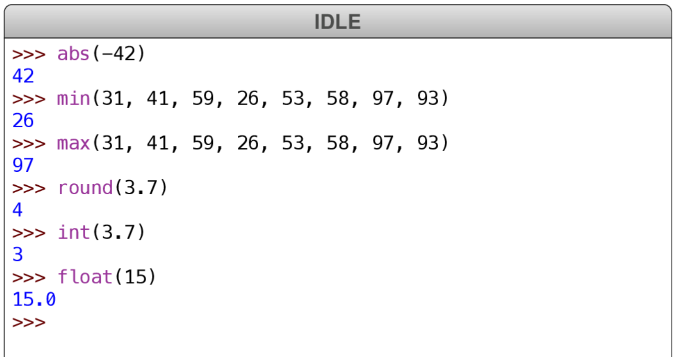
The use of the `c_to_f` function is illustrated in the following interaction with the IDLE console:



```
>>> def c_to_f(c):
        return 9 / 5 * c + 32

>>> c_to_f(0)
32.0
>>> c_to_f(100)
212.0
>>> c_to_f(-40)
-40.0
>>>
```

Built-in functions

Python defines more than 70 built-in functions that are always available for your programs to call. Most of those functions are beyond the scope of this chapter—and many are beyond the scope of this text—but the numeric functions in Figure 1-3 are likely to come in handy. The following IDLE session gives you a sense of how these functions work:



```
>>> abs(-42)
42
>>> min(31, 41, 59, 26, 53, 58, 97, 93)
26
>>> max(31, 41, 59, 26, 53, 58, 97, 93)
97
>>> round(3.7)
4
>>> int(3.7)
3
>>> float(15)
15.0
>>>
```

Library functions

Beyond the built-in functions, Python defines several collections of functions and other useful definitions and makes those collections available as ***libraries.*** One of the most useful libraries in Python is the `math` library, which includes several mathematical definitions that come up often when you are writing programs, even when those programs don't seem particularly mathematical. Figure 1 -4at the top of the next page lists several constants and functions available in the `math` library.

In Python, you gain access to the facilities of a library by ***importing*** that library. To do so, you have two options. The first is to use an `import` statement to indicate that your code would like to use the functions in that library using their ***fully qualified names,*** which consist of the name of the library, a period (which programmers usually

FIGURE 1-3 Built-in numeric functions

`abs(`*x*`)`	Returns the absolute value of *x*.
`max(`*x*`,` *y*`,` ...`)`	Returns the largest of the arguments.
`min(`*x*`,` *y*`,` ...`)`	Returns the smallest of the arguments.
`round(`*x*`)`	Returns the closest integer to *x*.
`int(`*x*`)`	Converts *x* to an integer, discarding any fraction.
`float(`*x*`)`	Converts *x* to a floating-point number.

FIGURE 1-4 Selected constants and functions from the Python `math` library

Mathematical constants

`pi`	The mathematical constant π.
`e`	The mathematical constant *e,* which is the base for natural logarithms.

General mathematical functions

`sqrt(`*x*`)`	Returns the square root of *x*.
`floor(`*x*`)`	Returns the largest integer less than or equal to *x*.
`ceil(`*x*`)`	Returns the smallest integer greater than or equal to *x*.
`copysign(`*x*`,` *y*`)`	Returns *x* with the sign of *y*.

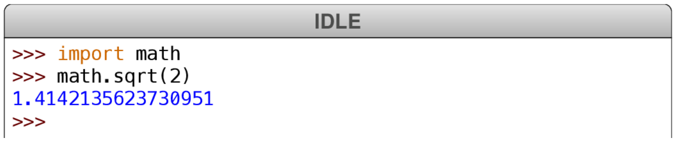
Logarithmic and exponential functions

`exp(`*x*`)`	Returns the exponential function of *x* (e^x).
`log(`*x*`)`	Returns the natural logarithm (base *e*) of *x*.
`log10(`*x*`)`	Returns the common logarithm (base 10) of *x*.

Trigonometric functions

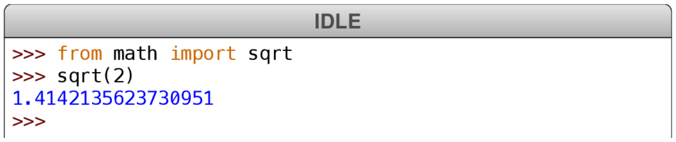
`cos(`*theta*`)`	Returns the cosine of the radian angle *theta.*
`sin(`*theta*`)`	Returns the sine of the radian angle *theta.*
`tan(`*theta*`)`	Returns the tangent of the radian angle *theta.*
`atan(`*x*`)`	Returns the principal arctangent of *x,* which lies between –π/2 and +π/2.
`atan2(`*y*`,` *x*`)`	Returns the angle between the *x*-axis and the line from the origin to (*x*, *y*).
`radians(`*angle*`)`	Converts an angle measured in degrees to radians.
`degrees(`*angle*`)`	Converts an angle measured in radians to degrees.

call a *dot*), and the name of the function. The following IDLE session illustrates the use of this form of the `import` statement to calculate the square root of 2:



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

The second option is to use a `from-import` statement to incorporate one or more functions directly into the current Python environment. This style of import allows you to use the name of the library function without qualification, as follows:



```
>>> from math import sqrt
>>> sqrt(2)
1.4142135623730951
>>>
```

The choice of which of these options to use depends to some extent on what you expect human readers of the program to understand. The advantage of the fully qualified name is that readers know precisely where the function is defined. On the other hand, if you are using mathematical functions extensively and expect any readers of the program to be familiar with them, the second version is more concise. The programs in this text use both styles.

1.5 Nonnumeric data

So far, the programming examples in this chapter have worked only with numeric data. These days, computers work less with numbers than with other forms of data. Although you will have a chance to learn about many other data types as you go through this text, learning a little bit about two common data types—strings and lists—will expand the range of programs that you can write. You will learn more about each of these types in a subsequent chapter, but this presentation will be enough to get you started.

Strings

In today's world, the most widely used type of data is ***string data,*** which is a generic term for information composed of individual characters. The ability of modern computers to process string data has led to the development of text messaging, electronic mail, word processing systems, social networking, and a wide variety of other useful applications.

Conceptually, a ***string*** is a sequence of characters taken together as a unit. As in most modern languages, Python includes strings as a built-in type, indicated in a program by enclosing the sequence of characters in quotation marks. For example, the string `"Python"` is a sequence of six characters consisting of an uppercase letter followed by five lowercase letters. The string `"To be, or not to be"` from Hamlet's soliloquy is a sequence of 19 characters including 13 letters, five spaces, and a comma.

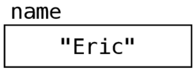
Python allows you to use either single or double quotation marks to specify a string, but it is good practice to pick a style and then use it consistently. The programs in this book use double quotation marks, mostly because that convention is common across a wide range of programming languages. The only exception is when the string itself contains a double quotation mark, as in `'"'`, which specifies a one-character string consisting of a double quotation mark. You can also include a quotation mark in a string by preceding it with a backslash character (\). Thus, you can also write the one-character string containing a double quotation mark as "\"".

For the most part, you can use strings as a Python data type in much the same way that you use numbers. You can, for example, assign string values to variables, just as you do with numeric values. For example, the assignment

```
name = "Eric"
```

creates a variable called name and initializes it to the four-character string "Eric".

As with the numeric variables introduced earlier in the chapter, the easiest way to represent a string-valued variable is to draw a box with the name on the outside and the value on the inside, like this:

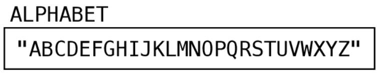


The quotation marks are not part of the string but are nonetheless included in box diagrams to make it easier to see where the string begins and ends.

Similarly, you can create string constants, as in the following example:

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

This statement defines the constant ALPHABET to be a string consisting of the 26 uppercase letters, as illustrated by the following box diagram:



Lists

The strings described in the preceding section consist of ordered sequences of individual characters. The string "Eric", for example, is a sequence of the four characters "E", "r", "i", and "c". Generically, programmers refer to sequences of data values that appear in a fixed order as ***lists.***

AFI Top-Ten Films

1. *Citizen Kane*
2. *Casablanca*
3. *The Godfather*
4. *Gone with the Wind*
5. *Lawrence of Arabia*
6. *The Wizard of Oz*
7. *The Graduate*
8. *On the Waterfront*
9. *Schindler's List*
10. *Singin' in the Rain*

As human beings, we seem to delight in making lists. In 1998, the American Film Institute polled 1500 leaders of the film industry and created a list of the top 100 American films of all time, of which the top ten entries appear in the sidebar in the left margin. Lists are an example of a ***compound data type.*** The list itself is a value in its own right, but it is also composed of individual components, which in this case are strings.

Python makes it extremely easy to create a list. All you have to do is enclose the elements of the list inside square brackets and separate the elements with commas.

For example, you can define the constant `AFI_TOP_TEN_FILMS` using the following declaration:

```
AFI_TOP_TEN_FILMS = [
    "Citizen Kane",
    "Casablanca",
    "The Godfather",
    "Gone with the Wind",
    "Lawrence of Arabia",
    "The Wizard of Oz",
    "The Graduate",
    "On the Waterfront",
    "Schindler's List",
    "Singing' in the Rain"
]
```

This declaration defines a list containing ten elements, each of which is a string. It also illustrates an important convention about formatting long lines in Python. In Python, the end of a statement is indicated by the end-of-line character, which is typically labeled as RETURN or ENTER on the keyboard. If you want to break a long statement across multiple lines to make it more readable, you ordinarily need to precede each line break with a ***backslash character*** (`/`). Python, however, makes an exception to this rule in the case of code enclosed within parentheses, square brackets, or curly braces, in which case the backslash character is not required. In the definition of `AFI_TOP_TEN_FILMS`, for example, all the strings are enclosed inside the square brackets that create the list, so the backslash characters do not appear.

The elements of a list can be of any type. The following definition creates a list containing the five integers 4, 6, 8, 12, and 20:

```
PLATONIC_SOLIDS = [ 4, 6, 8, 12, 20 ]
```

If you are a fan of classical mathematics (or, alternatively, role-playing games that use a variety of oddly shaped dice), you may recognize these values as the number of sides of the only three-dimensional shapes in which the faces are identical polygons with equal angles and side lengths. These shapes are the ***regular polyhedra,*** which are also known as the ***Platonic solids*** in honor of Plato's identification of them as the only shapes whose perfect symmetry revealed the beauty of the universe.

tetrahedron

cube

octahedron

dodecahedron

icosahedron

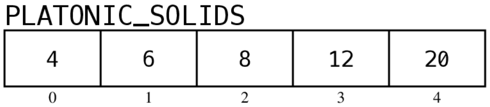
Operations on sequences

In Section 1.1, you learned that data types are defined by two properties: a *domain* and a *set of operations.* For strings, the domain is the set of all sequences of characters. For lists, the domain is the set of all sequences of Python values. Since both strings and lists are sequences, it is not surprising that the operations they support are similar. In fact, Python does a better job than most languages to ensure that the operations on these types adopt the same conventions, making those operations easier to learn and use.

In Python, most operations on sequences are defined using an object-oriented model of programming, which you will learn more about in later chapters. For the moment, it is sufficient to learn just three operations:

1. Selecting an element from a sequence**.**
2. Joining two sequences together end to end, which is called ***concatenation.***
3. Determining the length of a sequence.

In Python, the elements of a sequence—whether those elements are characters in a string or values in a list—are numbered starting with 0. For example, the elements of the list `PLATONIC_SOLIDS` are numbered like this:



PLATONIC_SOLIDS

4	6	8	12	20
0	1	2	3	4

In computer science, each element number in a list is called the ***index*** of the corresponding element. You select an individual element from a sequence by writing the name of the sequence, followed by the index number enclosed in square brackets. Thus, `PLATONIC_SOLIDS[3]` selects the value 12.

Python specifies concatenation using the `+` operator, which is also used to indicate addition for numbers. When Python evaluates the `+` operator, it first checks the types of the operands. If both operands are numeric, Python interprets the `+` operator as addition. If both operands are sequences, Python interprets the `+` operator as concatenation. For example, the expression

```
2 + 2
```

has the value 4, because both of the operands to `+` are numbers. Conversely,

```
"abc" + "def"
```

produces the six-character string `"abcdef"`.

In this example, it is important to observe that using the concatenation operator with strings does not introduce a space character or any other separator between the words. If you want to combine two strings into a single string that represents two distinct words, you have to include the space explicitly. For example, assuming that the variable `greeting` contains the string `"Hello"` and the variable `name` contains the string `"Eric"`, the expression

```
greeting + " " + name
```

produces the ten-character string "Hello Eric".

Some languages—most notably Java and JavaScript—define the + operator so that it allows you to combine a string with other types. If at least one of the operands to + is a string, those languages automatically convert the other to its string form. In Python, you must specify this conversion explicitly using the built-in function `str`, which converts values of any type to their string form as a sequence of characters. For example, calling `str(1729)` returns the four-character string "1729". Similarly, calling `str(12.50)` returns the string "12.5". The string value does not include the trailing zero that appears in the call, because Python interprets `12.50` as a number rather than as a string of digits. As a number, `12.50` has the value 12.5.

The `str` function therefore makes it possible to concatenate values of any type into a string expression. For example, the expression

```
"Fahrenheit " + str(451)
```

produces the string "Fahrenheit 451" because the built-in `str` function converts the numeric value 451 to the string "451" before combining the strings together.

Modern versions of Python, however, make it easy to assemble a string from values of different types by using a new feature called an ***f-string,*** which stands for formatted string. Formatted strings are so marvelously useful that it makes sense to introduce them in a simple form here, deferring the details of their operation to Chapter 7.

When Python encounters an *f*-string indicated by writing the letter `f` before the opening quotation mark, it looks inside the string for any expressions enclosed in curly braces and replaces each of those expressions with its value, performing any necessary string conversion automatically. For example, you can insert the value of the variable `temperature` after the word "Fahrenheit" using the following *f*-string:

```
f"The answer is {temperature}."
```

If `temperature` contains the integer 451, Python would interpret this string as "Fahrenheit 451", which is the title of a Ray Bradbury novel. If `temperature` instead contains the integer 911, the value would be "Fahrenheit 911", the title of a film by Michael Moore. Formatted strings are particularly useful in conjunction with calls to the `print` function, which you will learn about in Chapter 2.

Python also includes built-in functions that make it possible to convert strings that represent numbers into their numeric counterparts. If you give it a string argument, the `int` function tries to convert that string to an integer by checking to see that the value is a string of digits, possibly preceded by a minus sign. Thus, calling `int("1729")` produces the integer value 1729. Similarly, the `float` function converts a string representing a floating-point number into its numeric counterpart, so that `float("12.50")` produces that value 12.5. If you pass `int` or `float` a string that cannot be interpreted as a number of the appropriate type, Python signals that an error has occurred.

In Python, you can determine the length of any sequence by calling the built-in function `len`. For example, if you have initialized `ALPHABET` as shown on page 18, calling `len(ALPHABET)` returns the value 26. Similarly, calling `len("")` returns the value 0, because the string `""` contains no characters between the quotation marks. The string containing no characters at all is called the ***empty string.*** As you will discover in Chapter 7, the empty string comes up frequently when you are writing programs to manipulate string data.

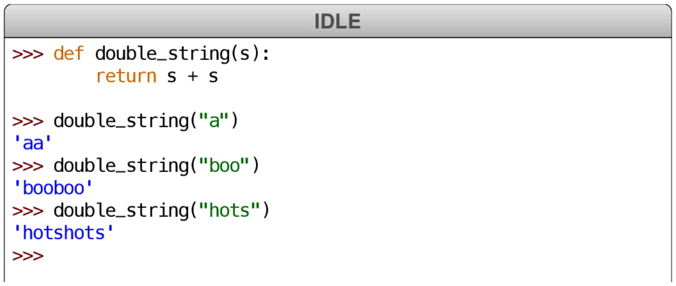
Writing simple string functions

Although you will need the additional operations from Chapter 7 to write anything more than the simplest string functions, it is worth looking at a few examples that use only the concatenation operator.

The following function

```
def double_string(s):
   return s + s
```

returns two copies of the supplied string joined together. This function enables the following IDLE session:



```
>>> def double_string(s):
        return s + s

>>> double_string("a")
'aa'
>>> double_string("boo")
'booboo'
>>> double_string("hots")
'hotshots'
>>>
```

This screen transcript also includes two features that illustrate how the IDLE interpreter works with strings. First, the IDLE interpreter displays string values using single quotation marks rather than the double-quoted strings used in the text. Second, the IDLE sessions—along with code in figures or indented text blocks but not in text paragraphs—uses ***syntax coloring,*** in which different components of a program appear in different colors. Keywords appear in orange, built-in functions appear in purple, strings appear in green, and console output appears in blue.

1.6 Writing Python programs

The program examples you have seen so far define functions by typing them directly into the IDLE interpreter. In practice, experienced programmers rarely type more than a few lines into an interactive interpretive environment like IDLE. What they do instead is use an ***editor,*** which is an application that allows you to create and modify files on your computer, to create files that define simple programs in their entirety or logically connected parts of a larger program. Those files are called ***modules.*** In Python, the names of files containing modules end with the suffix `.py`.

As an example, Figure 1-5 combines the definition of the function `c_to_f` from page 14 with the corresponding definition of a function `f_to_c` into a single module stored in a file called `temperature.py`. The code in Figure 1-5 illustrates two new features of Python, both of which are important for writing programs that are easy to understand and maintain. The first line of `temperature.py` is an example of a ***comment,*** which consists of text designed to explain the operation of the program to human readers. In Python, comments begin with the hashtag character (#) and extend through the end of the line. Here—as in the first line of all modules used in this text—the comment specifies the name of the file in which the code appears.

The other new feature is the text that appears in green in Figure 1-5, which is used in three different places in the code. These text strings are also comments for the reader, but are written in a different way. Each one begins with three quotation marks on a line, continues with any number of text lines, and then ends with another line of three quotation marks. In Python, these bits of commentary are called ***docstrings.*** It

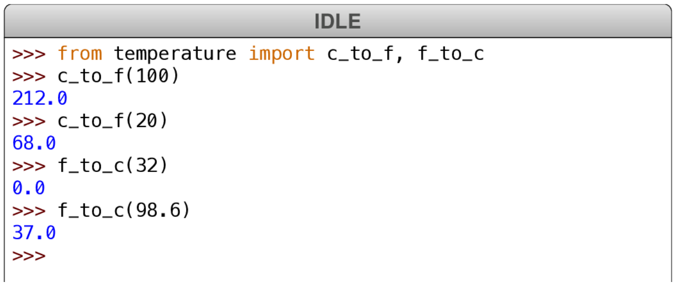
FIGURE 1-5 Implementation of the `temperature.py` module

```
# File: temperature.py

"""
This module defines functions for converting back and forth
between Celsius and Fahrenheit temperatures.
"""

def c_to_f(c):
    """
    Converts a temperature from Celsius to Fahrenheit.
    """
    return 9 / 5 * c + 32

def f_to_c(f):
    """
    Converts a temperature from Fahrenheit to Celsius.
    """
    return 5 / 9 * (f - 32)
```

is good programming practice to include a docstring at the beginning of each module or function to describe its operation.

Once you have created a module, you can use it with the IDLE interpreter in exactly the same way that you use one of the standard libraries. You use an `import` statement—which may appear in either the `import` or `from-import` form—to acquire the definitions you need, which you can then use in your own code.

The following IDLE session shows how to load the `c_to_f` and `f_to_c` functions from the `temperature` module and then use them in expressions:



```
>>> from temperature import c_to_f, f_to_c
>>> c_to_f(100)
212.0
>>> c_to_f(20)
68.0
>>> f_to_c(32)
0.0
>>> f_to_c(98.6)
37.0
>>>
```

Specifying a start-up function

The ability to store programs in files also makes it possible to run Python programs without using the IDLE interpreter. The details for doing so differ slightly depending on what type of machine you're using, but the idea is the same. You need to open up a command window—which is called the shell on Unix systems, the Terminal application on the Macintosh, and Command Prompt on Windows—and then use the command window to invoke the Python interpreter on your program file. Again, the precise syntax for the command line may differ from machine to machine, but the following command is typical:

```
    python3 name.py
```

where *name* is the name of the Python module you want to run.

Once you have entered this command, the Python interpreter takes over and reads in the contents of the specified module, just as if you typed those lines into IDLE. A typical module includes assignment statements and function definitions but must also include some code after those definitions to get the program running.

Although you could start the program by including an explicit function call at the end of the program file, doing so makes it difficult to use the program module as part of another application. By convention, what Python programmers do instead is add the following lines to the end of the program file, where *function* is the name of the function that starts the program:

```
if __name__ == "__main__":
    function()
```

The rather cryptic line at the beginning of this code fragment tests to see if this module is being invoked directly as a command or if it has been loaded as a library. To do so, it relies on the fact that Python initializes the variable `__name__` to the name of the module. If a module is invoked from the command line, this variable contains the string `"__main__"`. In this case, the Python interpreter calls the specified function. If the module has been loaded as a library, that call is skipped.

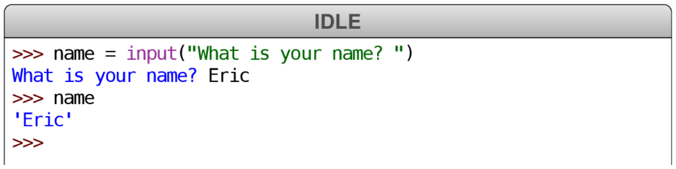
At this point, you shouldn't worry if you don't understand exactly what the startup code at the end of a program file does. Code patterns that always appear in a specific form are often called ***boilerplate.*** In general, it is more important simply to memorize the boilerplate than to learn precisely how it works.

Communicating with the user

Most application programs need to interact with the user in some way. Modern applications typically communicate with the user through a ***graphical user interface*** or ***GUI.*** Although you will have the opportunity to design simple graphical user interfaces beginning in Chapter 6, it is much easier to begin with an older style of interaction in which users communicate with applications through a console. The application asks the user to enter values by typing on the keyboard, after which the application performs some calculations and then displays the results on the screen. Generically, the information the user enters is called ***input,*** and the information displayed back to the user is called ***output.***

When you are first learning to program, it is essential to appreciate the distinction between the way that functions communicate within a program and the way in which an application communicates with a user. Functions communicates with one another by passing arguments and returning results. Applications communicate with the user by calling input and output functions, which for the console are the `input` and `print` functions described later in this section. If you are asked on an assignment or an exam to write a function that takes in some number of values and returns a result, that function will not ordinarily include calls to the `input` and `print` functions.

Console applications often require the user to enter information on the keyboard and then use the input data to compute some result. Python uses the built-in function `input` for this purpose. The `input` function takes a string argument, which is then displayed on the console so that the user knows what information the program needs. This string is called a ***prompt.*** After printing the prompt, the `input` function waits for the user to type in a line of text and then pressing the RETURN key. The `input` function than returns that string to the caller, as illustrated in the following IDLE session:



```
>>> name = input("What is your name? ")
What is your name? Eric
>>> name
'Eric'
>>>
```

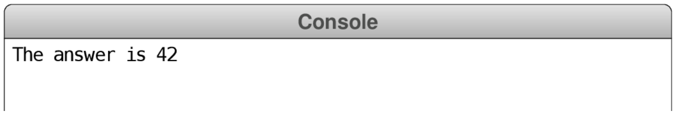
Even though the `input` function always returns a string, you can use it in conjunction with the built-in functions `int` and `float` to read in numeric values. If, for example, you want to read an integer from the user and store that value in the variable `n`, you can use the following code:

```
n = int(input("Enter an integer? "))
```

Console output uses the built-in function `print`, which takes one or more arguments and then prints each one on the console, separated by spaces. For example, if the variable `answer` contains the number 42, executing the statement

```
print("The answer is", answer)
```

generates the output



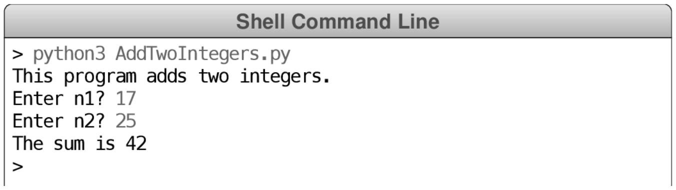
```
The answer is 42
```

where the space between the string and the number 42 is supplied by the comma in the `print` call.

In most cases, however, it makes more sense to use Python's *f*-string feature to generate the output string. As an example, you could rewrite this statement to use an *f*-string like this:

```
print(f"The answer is {answer}")
```

Console input and output are illustrated in more detail in the `AddTwoIntegers.py` program in Figure 1-6 at the top of the next page, which reads two integers from the user and then displays the result. If you run `AddTwoIntegers.py` from the command line, Python will call the `add_two_integers` function, which generates a console transcript that looks something like this:



```
> python3 AddTwoIntegers.py
This program adds two integers.
Enter n1? 17
Enter n2? 25
The sum is 42
>
```

FIGURE 1-6 A program to add two integers

```
# File: AddTwoIntegers.py

"""
This program adds two integers entered by the user.
"""

def add_two_integers():
    print("This program adds two integers.")
    n1 = int(input("Enter n1? "))
    n2 = int(input("Enter n2? "))
    total = n1 + n2
    print(f"The sum is {total}")

# Startup code

if __name__ == "__main__":
    add_two_integers()
```

Summary

In this chapter, you have started your journey toward programming in Python by learning how to write simple expressions involving a few important data types including integers, floating-point numbers, and strings. Important points introduced in the chapter include:

- The primary focus of this book is not the Python language itself but rather the principles you need to understand the fundamentals of programming.
- Data values come in many different types, each of which is defined by a *domain* and a *set of operations.*
- Integers in Python are written as a string of decimal digits optionally preceded by a minus sign.
- Floating-point numbers in Python are written in conventional decimal notation. Python also allows you to write numbers in scientific notation by adding the letter `E` and an exponent indicating the power of 10 by which the number is multiplied.
- Expressions consist of individual *terms* connected by *operators*. The subexpressions to which an operator applies are called its *operands*.
- The order of operations is determined by rules of *precedence*. The complete table of operators and their precedence appears in Figure 1-1 on page 7.
- *Variables* in Python have two attributes: a name and a value. Variables used in a Python program are created using an *assignment statement* in the form

    variable = *expression*

which establishes the name and value of the variable. You can subsequently use assignment statements to change the value of an existing variable. When you assign a new value to a variable, any previous value is lost.

- Python includes an abbreviated form of assignment in which the statement

  variable op= *expression*

  acts as a shorthand for the longer expression

  variable = *variable op* (*expression*)

- Python allows multiple assignments within the same statement, as in

  ```
  a, b, c = 3, 4, 5
  ```

  which assigns `a`, `b`, and `c` the values 3, 4, and 5, respectively. The values of the expressions on the right hand side are evaluated before any assignments are made, so that

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

  exchanges the values of the variables `x` and `y`.

- Variable names that include more than one word can be hard to read unless you mark the word divisions in some way. The programs in this text follow the standard Python convention of using *snake case* in the names of variables, functions, and methods, in which the individual words are separated by underscores, as in the variable name `number_of_pages`. Class names, which will be introduced in subsequent chapters, use a different style convention called *camel case,* in which each word embedded in the name starts with an uppercase letter, as in the class name `GFillableObject`.
- *Constants* are used to specify values that do not change within a program. By convention, you write the names of constants entirely in upper case, using the underscore to mark word boundaries.
- A *function* is a block of code that has been organized into a separate unit and given a name. Other parts of the program can then *call* that function, possibly passing it *arguments* and receiving a result *returned* by that function.
- Variables that are assigned values inside the body of a function are called *local variables* and are visible only inside that function. Variables defined outside of a function are *global variables,* which can be used anywhere in the program. This book avoids global variables, because they make programs harder to maintain.
- A function that returns a value must have a `return` statement that specifies the result. Functions may return values of any type.
- Python predefines a set of built-in functions that are always available for use in programs. Figure 1-3 lists several of the more important ones.

- Python's `math` library defines a variety of functions that implement such standard mathematical functions as `sqrt`, `sin`, and `cos`. A list of the more common mathematical functions appears in Figure 1-4 on page 16.
- Before you use a function defined in a library, you must *import* that library. Python supports two strategies for importing functions from libraries. The `import` statement allows you to use a library function by specifying its fully qualified name. The `import-from` statement allows you to import specific functions from a library, which you can then use without including the library name.
- A *string* is a sequence of characters taken together as a unit. In Python, you write a string by enclosing its characters in quotation marks. Python accepts either single or double quotation marks for this purpose.
- Although strings support many additional operations that will be presented in Chapter 7, the examples in this chapter and the next few chapters use only the `+` operator to *concatenate* string values along with the built-in functions `len`, `str`, `int`, and `float`.
- Although you can use concatenation to combine strings with values of other types, doing so is not as convenient as using *f-strings.* An *f*-string begins with the letter `f` before the opening quotation mark and signifies that Python should substitute the corresponding value for any expressions enclosed in curly braces.
- A *list* is a sequence of Python values taken together as a unit. In Python, you write a list by enclosing its elements in square brackets.
- Strings and lists are both examples of *sequences* in Python. Each element of a sequence is assigned an index number starting with 0. You can select an element from a sequence by enclosing the index in square brackets after the sequence.
- Python programs are often stored in files called *modules.* As with functions defined in a library, you can use the facilities defined in a module by importing those definitions into your own computation.
- Python applications typically define a *startup function* using the following pattern:

```
if __name__ == "__main__":
    function()
```

- Functions in Python communicate by passing arguments and returning results. Console applications communicate with the user through the built-in functions `input` and `print`.

Review questions

1. What are the two attributes that define a data type?

2. What is the difference between an *integer* and a *floating-point number?*

3. Identify which of the following are legal numbers in Python:

   a) `42`
   b) `-17`
   c) `2+3`
   d) `-2.3`
   e) `20`
   f) `2.0`
   g) `1,000,000`
   h) `3.1415926`
   i) `123456789`
   j) `0.000001`
   k) `1.1E+11`
   l) `1.1X+11`

4. Rewrite the following numbers using Python's form for scientific notation:

   a) 6.02252×10^{23}
   b) 29979250000.0
   c) 0.00000000529167
   d) 3.1415926535

   By the way, each of these values is an approximation of an important scientific or mathematical constant: (a) Avogadro's number, which is the number of molecules in one mole of a chemical substance; (b) the speed of light in centimeters per second; (c) the Bohr radius in centimeters, which is the average radius of an electron's orbit around a hydrogen atom in its lowest-energy state; and (d) the mathematical constant π.

5. Indicate which of the following are legal variable names in Python:

   a) `x`
   b) `formula1`
   c) `average_rainfall`
   d) `%correct`
   e) `pass`
   f) `pass2`
   g) `total output`
   h) `a_very_long_variable_name`
   i) `12_month_total`
   j) `marginal-cost`
   k) `b4hand`
   l) `_stk_depth`

6. In your own words, describe the effect of the `/`, `//`, and `%` operators in Python?

7. True or false: The – operator has the same precedence when it is used before an operand to indicate negation as it does when it is used to indicate subtraction.

8. By applying the appropriate precedence rules, calculate the result of each of the following expressions:

   a) `6 + 5 / 4 - 3`
   b) `2 + 2 * (2 * 2 - 2) % 2 / 2`
   c) `10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1`
   d) `1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10`

9. What shorthand assignment statement would you use to multiply the value of the variable `salary` by 2?

10. How can you use multiple assignment to exchange the values of the variables `x` and `y`? How would you achieve the same effect using traditional assignment statements?

11. What is the value of each of the following expressions:

    a) `round(5.99)`
    b) `math.floor(5.99)`
    c) `math.ceil(5.99)`
    d) `math.floor(-5.99)`
    e) `math.sqrt(3 ** 2 + 4 ** 2)`

12. What is meant by the terms *snake case* and *camel case?*

13. How do you specify a string value in Python?

14. What is an *f*-string, and how do you specify one in Python?

15. If a string is stored in the variable `s,` how would you determine its length?

16. How would you create a constant `DWARVES` containing the names of the seven Disney dwarves (Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, and Sneezy)?

17. What index expression would you use to select the name Happy from the `DWARVES` list defined in the previous question.

18. What is meant by the term *concatenation?*

19. How does Python decide whether to interpret the `+` operator as addition or concatenation?

20. What is the format of the startup boilerplate used to specify the starting function in a Python application?

21. In your own words, describe the difference between communicating information using arguments and results as opposed to communicating information using the `input` and `print` functions.

22. What is a *prompt?*

22. How would you ask the user to enter a floating-point number signifying the radius of a circle?

23. How would you rewrite the following *f*-string using concatenation, assuming that the values of the variables `n1` and n2 are integers:

```
print(f"The sum of {n1} and {n2} is {n1 + n2}.")
```

Exercises

1. How would you implement the following mathematical function in Python:

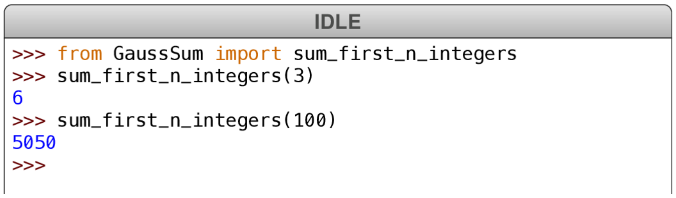
$$f(x) = x^2 - 5x + 6$$

Carl Friedrich Gauss

2. According to mathematical historians, the German mathematician Carl Friedrich Gauss (1777–1855) began to show his mathematical talent at a very early age. When he was in primary school, Gauss was asked by his teacher to compute the sum of the first 100 integers. Gauss is said to have produced the answer instantly by working out that the sum of the first N integers is given by the formula

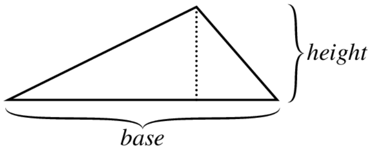
$$\frac{N \times (N+1)}{2}$$

Write a function `sum_first_n_integers` that takes an integer n and returns the sum of the first n integers, as illustrated in the following sample run:



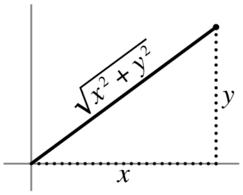
```
>>> from GaussSum import sum_first_n_integers
>>> sum_first_n_integers(3)
6
>>> sum_first_n_integers(100)
5050
>>>
```

3. Write a function `triangle_area` that computes the area of a triangle given values for its base and its height, as ahown in the following diagram:



Given any triangle, the area is always one half of the base times the height.

4. Write a function `distance` that calculates the distance from the origin to the point whose coordinates are given by the parameters `x` and `y`. The formula for calculating this distance, traditionally attributed to the Greek philosopher Pythagoras in the 6th century BCE, is illustrated in the following diagram:



5. Write a function `plural(word)` that adds the string `"s"` to the end of the word to create a simple plural form. This strategy, of course, does not work for all English words, many of which require adding `"es"` instead of `"s"` depending on the final consonant. You will have a chance to solve that more sophisticated problem in Chapter 7.

6. Write a function `quote(s)` that uses concatenation to add a double quotation mark to each end of the string `s`.

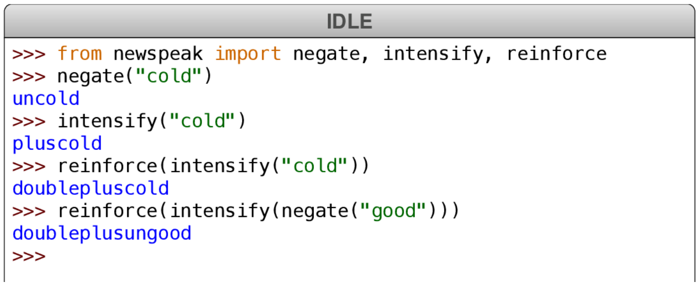
7. *It is a beautiful thing, the destruction of words.*

   —Syme in George Orwell's *1984*

   In Orwell's novel, Syme and his colleagues at the Ministry of Truth are engaged in simplifying English into a more regular language called *Newspeak*. As Orwell describes in his appendix entitled "The Principles of Newspeak," words can take a variety of prefixes to eliminate the need for the massive number of words we have in English. For example, Orwell writes,

   > Any word—this again applied in principle to every word in the language—could be negatived by adding the affix *un*-, or could be strengthened by the affix *plus*-, or, for still greater emphasis, *doubleplus*-. Thus, for example, *uncold* meant "warm," while *pluscold* and *doublepluscold* meant, respectively, "very cold" and "superlatively cold."

   Define three functions—`negate`, `intensify`, and `reinforce`—that take a string and add the prefixes `"un"`, `"plus"`, and `"double"` to that string, respectively. Your function definitions should allow you to generate the following console session:



```
>>> from newspeak import negate, intensify, reinforce
>>> negate("cold")
uncold
>>> intensify("cold")
pluscold
>>> reinforce(intensify("cold"))
doublepluscold
>>> reinforce(intensify(negate("good")))
doubleplusungood
>>>
```

8. Use the `temperature.py` module as a model to design and implement a new module called `metric.py` that defines the following functions:

   - A function `miles_to_kilometers` that takes a value representing a distance in miles and returns the corresponding distance in kilometers.
   - A function `feet_and_inches_to_centimeters` that takes two arguments (`feet` and `inches`) and returns the equivalent distance in centimeters.

Your module should define the following constants and use them to perform the conversions:

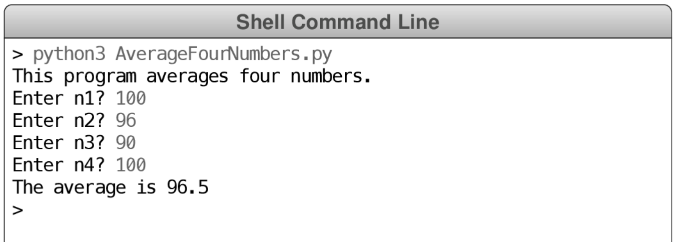
```
FEET_PER_MILE = 5280
CENTIMETERS_PER_INCH = 2.54
INCHES_PER_FOOT = 12
CENTIMETERS_PER_METER = 100
METERS_PER_KILOMETER = 1000
```

Test your implementation by importing this module into IDLE and verifying the following conversions:

- The standard distance for a marathon is 26.2 miles, which is approximately 42.165 kilometers.
- Eight feet and four inches represents a total length of 100 inches, which corresponds to 254 centimeters.

9. Rewrite the `temperature.py` module so that it includes a main program that asks the user for a Fahrenheit temperature and then displays the Celsius equivalent.

10. Write a program called `AverageThreeNumbers.py` that asks the user to enter three floating-point numbers and then computes their average. A sample run of this program from the command line might look like this:



```
> python3 AverageFourNumbers.py
This program averages four numbers.
Enter n1? 100
Enter n2? 96
Enter n3? 90
Enter n4? 100
The average is 96.5
>
```

11. Write a function `digit_name` that takes an integer between 0 and 9 and returns the English word for that digit as a string. For example, calling `digit_name(7)` should return the string `"seven"`. The easiest way to implement this function given the facilities of Python you know is to put the strings for the ten digits into a constant list and then have the function select the appropriate element from the list.

CHAPTER 2

Control Statements

> I had a running compiler and nobody would touch it. . . . They carefully told me, computers could only do arithmetic; they could not do programs.
>
> — Grace Murray Hopper, as quoted in Charlene Billings, *Grace Hopper: Navy Admiral and Computing Pioneer,* 1989

Grace Murray Hopper (1906–1992)

Grace Murray Hopper studied mathematics and physics at Vassar College and went on to earn her Ph.D. in mathematics at Yale. During the Second World War, Hopper joined the United States Navy and was posted to the Bureau of Ordinance Computation at Harvard University, where she worked with computing pioneer Howard Aiken. Hopper became one of the first programmers of the Mark I digital computer, which was one of the first machines capable of performing complex calculations. Hopper made several contributions to computing in its early years and was one of the major contributors to the development of the language COBOL, which continues to have widespread use in business programming applications. In 1985, Hopper became the first woman promoted to the rank of admiral. During her life, Grace Murray Hopper served as the most visible example of a successful woman in computer science. In recognition of that contribution, there is now an annual Celebration of Women in Computing, named in her honor.

In Chapter 1, you saw several examples of simple Python functions. In each of those examples, execution of the function started with the first statement in its body and then continued through the remaining statements in order, possibly calling other functions along the way. Before you can write more interesting applications, you need to learn how to control the operation of your program in more sophisticated ways.

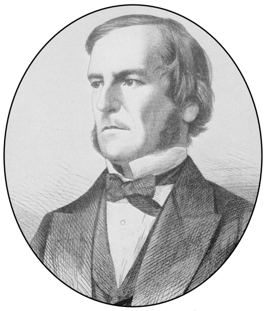
This chapter introduces new statement forms that enable you to change the way in which Python executes your programs. Collectively, these statements are called ***control statements.*** Control statements fall into the following two classes:

1. ***Conditional statements.*** Conditional statements specify that certain statements in a program should be executed only if a particular condition holds. In Python, you specify conditional execution using an `if` statement, which exists in several forms. The last section in this chapter also introduces the `assert` statement, which makes it easy to write simple test programs.
2. ***Iterative statements.*** Iterative statements specify that certain statements in a program should be executed repeatedly, forming what programmers call a ***loop.*** Python supports two iterative statements: a `while` statement that allows you to repeat a set of statements as long as some condition holds, and a `for` statement that allows you to repeat a set of statements a certain number of times or for each item in some collection.

Students often believe that there must be some rule that determines when they need to use each of the various control statements a programming language provides. That's not how programming works. Control statements are tools for solving problems. Before you can determine what control statement makes sense in a particular context, you have to give serious thought to the problem you are trying to solve and the strategy you should choose to solve it. You write the code for a program *after* you have decided how to solve the underlying problem. There is nothing automatic about the programming process.

The fact that there are no magic rules that turn a problem statement into a working program is what makes programming such a valuable skill. If it *were* possible to create programs by following some well-defined procedure, it would be easy to automate the process and eliminate the need for programmers entirely. Programming consists of solving problems, many of which are extremely complex and require considerable ingenuity and creativity to solve. Solving such problems is what makes computer programming hard; it is also what makes programming interesting and fun.

2.1 Boolean data



George Boole

In Python, you express conditions by constructing expressions whose values are either true or false. Such expressions are called ***Boolean expressions,*** after the English mathematician George Boole, who developed an algebraic approach for working with data of this type. Boolean values are represented in Python using a built-in type whose domain consists of exactly two values: `True` and `False`.

Python defines several operators that work with Boolean values. These operators fall into two classes—relational operators and logical operators—as described in the next two sections.

Relational operators

The simplest questions you can ask in Python are those that compare two data values. You might want, for example, to determine whether two values are equal or whether one is greater than or smaller than another. Traditional mathematics uses the operators =, ≠, <, >, ≤, and ≥ to signify the relationships *equal to, not equal to, less than, greater than, less than or equal to,* and *greater than or equal to,* respectively. Because several of these symbols don't appear on a standard keyboard, Python represents these operators in a slightly different form, which uses the following character combinations in place of the usual mathematical symbols:

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

Collectively, these operators are called ***relational operators*** because they test the relationship between two values. Like the arithmetic operators introduced in Chapter 1, relational operators appear between the two values to which they apply. For example, if you need to check whether the value of `x` is less than 0, you can use the expression `x < 0`.

Logical operators

In addition to the relational operators, which take values of any type and produce Boolean results, Python defines three operators that take Boolean operands and combine them to form other Boolean values:

`not`	Logical not (`True` if the following operand is `False`)
`and`	Logical and (`True` if both operands are `True`)
`or`	Logical or (`True` if either or both operands are `True`)

These operators are called ***logical operators*** and are listed in decreasing order of precedence.

Although the operators `and`, `or`, and `not` correspond to the English words *and*, *or*, and *not,* it is important to remember that English is somewhat imprecise when it comes to logic. To avoid that imprecision, it helps to think of these operators in a more formal, mathematical way. Logicians define these operators using ***truth tables,*** which show how the value of a Boolean expression changes as the values of its operands change. For example, the truth table for the `and` operator, given Boolean values `p` and `q`, is

p	q	p and q
False	False	False
False	True	False
True	False	False
True	True	True

The last column of the table indicates the value of the Boolean expression `p and q`, given the individual values of the Boolean variables `p` and `q` shown in the first two columns. Thus, the first line in the truth table shows that when `p` is `False` and `q` is `False`, the value of the expression `p and q` is also `False`.

The truth table for `or` is

p	q	p or q
False	False	False
False	True	True
True	False	True
True	True	True

Even though the `or` operator corresponds to the English word *or,* it does not indicate *one or the other*, as it often does in English, but instead indicates *either or both*, which is its mathematical meaning.

The `not` operator has the following simple truth table:

p	not p
False	True
True	False

If you need to determine how a more complex logical expression operates, you can break it down into these primitive operations and build up a truth table for the individual pieces of the expression.

In most cases, logical expressions are not so complicated that you need a truth table to figure them out. The only case that often causes confusion is when the `not` operator comes up in conjunction with `and` or `or`. When English speakers talk about situations that are not true (as is the case when you work with the `not` operator), a statement whose meaning is clear to human listeners is often at odds with mathematical logic. Whenever you find that you need to express a condition involving the word *not,* you should use extra care to avoid errors.

As an example, suppose you wanted to express the idea "x is not equal to either 2 or 3" as part of a program. Just reading from the English version of this conditional test, new programmers are likely to code this expression as follows:

```
x != 2 or x != 3
```

As noted in Chapter 1, this book uses the bug symbol to mark sections of code that contain deliberate errors. In this case, the problem is that an informal English translation of the code does not correspond to its interpretation in Python. If you look at this conditional test from a mathematical point of view, you can see that the expression is `True` if either (a) `x` is not equal to 2 or (b) `x` is not equal to 3. No matter what value `x` has, one of the statements must be `True`, since, if `x` is 2, it cannot also be equal to 3, and vice versa.

To fix this problem, you need to refine your understanding of the English expression so that it states the condition more precisely. That is, you want the condition to be `True` whenever "it is not the case that either `x` is 2 or `x` is 3." You could translate this expression directly to Python by writing

```
not (x == 2 or x == 3)
```

but the resulting expression would be a bit ungainly. The question you really want to ask is whether *both* of the following conditions are `True`:

- `x` is not equal to 2, *and*
- `x` is not equal to 3.

If you think about the question in this form, you can write the test as

```
x != 2 and x != 3
```

This simplification is a specific illustration of the following more general relationship from mathematical logic:

`not (p or q)`   *is equivalent to*   `not p and not q` for any logical expressions p and q. This transformation rule and its symmetric counterpart

```
not (p and q)      is equivalent to      not p or not q
```

Augustus De Morgan

are called ***De Morgan's laws*** after the British mathematician Augustus De Morgan. Forgetting to apply these rules and relying instead on the English style of logic can lead to programming errors that are difficult to find.

Short-circuit evaluation

Python interprets the `and` and `or` operators in a way that differs from the interpretation used in many other programming languages. In the programming language Pascal, for example, evaluating these operators requires evaluating both halves of the condition, even when the result can be determined partway through the process.

The designers of Python (or, more accurately, the designers of earlier languages that influenced Python's design) took a different approach that is usually more convenient for programmers. Whenever Python evaluates an expression of the form

exp_1 `and` exp_2

or

exp_1 `or` exp_2

the individual subexpressions are always evaluated from left to right, and evaluation ends as soon as the answer can be determined. For example, if exp_1 is `False` in the expression involving `and`, there is no need to evaluate exp_2 since the final answer will always be `False`. Similarly, in the example using `or`, there is no need to evaluate the second operand if the first operand is `True`. This style of evaluation, which stops as soon as the answer is known, is called ***short-circuit evaluation.***

A primary advantage of short-circuit evaluation is that it allows one condition to control the execution of a second one. In many situations, the second part of a compound condition is meaningful only if the first part comes out a certain way. For example, suppose you want to express the combined condition that (1) the value of the integer `x` is nonzero and (2) `x` divides evenly into `y`. You can express this conditional test in Python as

```
(x != 0) and (y % x == 0)
```

because the expression `y % x` is evaluated only if x is nonzero. The corresponding expression in Pascal fails to generate the desired result, because both parts of the Pascal condition will always be evaluated. Thus, if `x` is 0, a Pascal program containing this expression will end up dividing by 0 even though it appears to have a conditional test to check for that case. Conditions that protect against evaluation errors in subsequent parts of a compound condition, such as the conditional test

```
(x != 0)
```

in the preceding example, are called ***guards.***

Avoiding fuzzy standards of truth

In the programs included in this book, every conditional test produces a Boolean value, which means that it will always be either `True` or `False`. Unfortunately, the Python language is rather less disciplined on this point. Python defines the following values (a couple of which you have not yet seen) to be ***falsy,*** presumably to imply that they are like the legitimate Boolean value `False`:

> `False`, `0`, `None`, `math.nan`, and any sequence of length 0 including `""`

Conversely, Python defines any other value to be ***truthy.*** In any conditional context, any "falsy" value is treated as if it were the value `False`; any "truthy" value is treated as if it were the value `True`.

The complexity of this situation is increased further by the fact that the `and` and `or` operators are implemented so that they allow operands to be of any type. When Python evaluates a sequence of expressions joined together by the `and` operator, it returns the first falsy value in the sequence, so that the expression

```
0 and True
```

returns the integer 0, because 0 is falsy and thus determines the value of the entire expression. Conversely, a sequence of expressions joined together by the `or` operator returns the first truthy value in the sequence.

Overly clever programmers will find uses for Python's rather convoluted interpretation of Boolean values. If, however, you want to write programs that are easy to read and maintain, you should avoid relying on these fuzzy definitions of truth and falsity and make sure—as this book does—that every test produces a legitimate Boolean value. In his book, *JavaScript: The Good Parts,* Douglas Crockford lists the "surprisingly large number of falsy values" in his appendix on the "awful parts" of JavaScript. That feature is no less awful in Python. But you might also take the following advice from a somewhat older source:

> Let what you say be simply "Yes" or "No"; anything more than this comes from evil.
>
> —Matthew 5:37, *The New English Bible*

Predicate functions

Although functions in Python can return values of any type, functions that return Boolean values deserve special attention because they play such an important role in programming. Functions that return Boolean values are called ***predicate functions.***

As you know from earlier in this chapter, there are only two Boolean values: True and False. Thus a predicate function—no matter how many arguments it takes or how complicated its internal processing may be—must eventually return one of these two values. The process of calling a predicate function is therefore analogous to asking a yes/no question and getting an answer. For example, the following function definition answers the question "is n an even number?" for a particular integer n supplied by the caller as an argument:

```
def is_even(n):
    return n % 2 == 0
```

A number is even if there is no remainder when you divide that number by two. If n is even, the expression n % 2 == 0 has the value True, which is returned as the value of is_even. If n is odd, the function returns False.

As a second example, it is an interesting exercise to implement the predicate function is_leap_year, which determines whether a given year qualifies as a leap year. Although one tends to think of leap years as occurring once every four years, astronomical realities are not quite so tidy. Because it takes about a quarter of a day more than 365 days for the earth to complete its orbit, adding an extra day once every four years helps keep the calendar in sync with the sun, but it is still off by a slight amount. To ensure that the beginning of the year does not slowly drift through the seasons, the rule used for leap years is in fact more complicated. Leap years come every four years, except for years ending in 00, which are leap years only if they are divisible by 400. Thus, 1900 was not a leap year even though 1900 is divisible by 4. The year 2000, on the other hand, was a leap year because 2000 is divisible by 400. For any leap year, one of the following conditions must hold:

- The year is divisible by 4 but not divisible by 100, *or*
- The year is divisible by 400.

It is easy to code the correct rule in Python as a predicate function, as follows:

```
def is_leap_year(year):
   return ((year % 4 == 0) and (year % 100 != 0)) or \
           (year % 400 == 0)
```

The return statement in the is_leap_year function illustrates an important feature of Python. In contrast to most modern languages that ignore spaces and line breaks, Python uses that spacing to define the hierarchical structure of a program. In Python, a line break ordinarily signals the end of a statement. Because the `return` statement includes a Boolean expression that doesn't fit comfortably on a single line, you need to find some way to let the expression extend across more than one line. This example solves the problem by preceding the line break in the middle of the expression by a backward slash (\), which causes Python to treat the following line as part of this one. Python also ignores any line breaks that occur within parentheses, square brackets, or curly braces, but that rule doesn't apply in this example as it appears. You will have many opportunities to see applications of this second rule, which removes the need for the line-continuation character.

2.2 The `if` statement

The simplest way to express conditional execution in Python is by using the `if` statement, which comes in three forms, as shown in the syntax boxes on the left. The first form of the `if` statement is useful when you want to perform an action only under certain conditions. The second is appropriate when you need to choose between two alternative courses of action. The third, which can contain any number of `elif` clauses, makes sense if you need to choose among several different courses of action.

```
if condition:
    statements
```

```
if condition:
    statements
else:
    statements
```

```
if condition_1:
    statements
elif condition_2:
    statements
elif condition_3:
    statements
else:
    statements
```

The *condition* component of these templates is a Boolean expression, as defined in the preceding section. This Boolean expression can be a simple comparison, a logical expression involving the `and`, `or`, and `not` operators, or a call to a predicate function. For example, if you want to test whether the number stored in `year` corresponds to a leap year, you can use the following `if` statement, which calls the `is_leap_year` function defined on the preceding page:

```
if is_leap_year(year):
```

In the first form of the `if` statement, Python executes the block of statements only if the conditional test evaluates to `True`. If the conditional test is `False`, Python skips the body of the `if` statement entirely. In the second form, Python executes the first block of statements if the condition is `True` and the second if the condition is `False`. In the third form, Python evaluates each of the conditions in turn and executes the statements associated with the first condition that evaluates to `True`. If none of the conditions apply, Python executes the statements associated with the `else` keyword.

You can use the `if` statement to implement your own versions of Python's built-in functions. For example, you can implement `abs`—at least for integers and floating-point numbers—as follows:

```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Similarly, you can implement `max` for two arguments like this:

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

As a third example, you can use the following definition to implement `sign(x)`, which returns –1, 0, or 1, depending on the sign of `x`:

```
def sign(x):
    if x < 0:
        return -1
    elif x == 0:
        return 0
    else:
        return 1
```

Choosing which form of the `if` statement to use requires you to think about the structure of the problem. You use the simple `if` statement when the problem requires code to be executed only if a particular condition applies. You use the `if-else` form for situations in which the program must choose between two independent sets of actions. You can often make this decision based on how you would describe the problem in English. If that description contains the word *otherwise* or some similar expression, there is a good chance that you'll need the `if-else` form. If the English description conveys no such notion, the simple form of the `if` statement is probably sufficient. Finally, you use the `if-elif-else` form to express a choice among several different options.

2.3 The `while` statement

The simplest iterative construct is the `while` statement, which repeatedly executes a simple statement or block until the conditional expression becomes `False`. The template for the `while` statement appears in the syntax box on the right. The entire statement, including both the `while` control line itself and the statements enclosed within the body, constitutes a ***while loop.*** When the program executes a `while` statement, it first evaluates the conditional expression to see if it is `True` or `False`. If the condition is `False`, the loop ***terminates*** and the program continues with the next

```
while condition:
    statements
```

statement after the entire loop. If the condition is True, the entire body is executed, after which the program goes back to the top to check the condition again. A single pass through the statements in the body constitutes a ***cycle*** of the loop.

There are two important principles to observe about the operation of a while loop:

1. The conditional test is performed before every cycle of the loop, including the first. If the test is False initially, the body of the loop is not executed at all.
2. The conditional test is performed only at the *beginning* of a loop cycle. If that condition happens to become False at some point during the loop, the program doesn't notice that fact until it has executed a complete cycle. At that point, the program reevaluates the test condition. If it is still False, the loop terminates.

Summing the digits in a number

As an illustration of the use of while, suppose that you have been asked to write a function called digit_sum that adds up the digits in an integer without converting it to a string. Calling digit_sum(1729) should therefore produce the result 19, which is 1 + 7 + 2 + 9. How would you go about implementing such a function?

The first thing that your function needs to do is keep track of a running total. The usual strategy for doing so is to declare a variable called total, set it to 0, add each digit to total one at a time, and finally return the value of total. That much of the structure, with the rest of the problem written in English, appears below:

```
def digit_sum(n):
    total = 0
    For each digit in the number, add that digit to total.
    return total
```

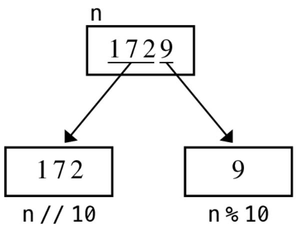
Programs that are written partly in a programming language and partly in English are called ***pseudocode.***

The sentence

For each digit in the number, add that digit to total.

clearly specifies a loop structure of some sort, since there is an operation that needs to be repeated for each digit in the number. If it were easy to determine how many digits a number contained, you might choose to use the for loop described later in this chapter to run through precisely that many cycles. As it happens, finding out how many digits there are in a number is just as hard as adding them up in the first place. The best way to write this program is just to keep adding in digits until you discover that you have added the last one. Loops that run until some condition occurs are most often coded using the while statement.

The essence of this problem lies in determining how to break up a number into its component digits. The last digit of an integer `n` is simply the remainder left over when `n` is divided by 10, which is the result of the expression `n % 10`. The rest of the number—the integer that consists of all digits *except* the last one—is given by `n // 10`. For example, if `n` has the value 1729, you can use these two expressions to break that number into two parts, 172 and 9, as shown in the following diagram:



Thus, in order to add up the digits in the number, all you need to do is add the value `n % 10` to the variable `total` on each cycle of the loop and then replace the value of `n` by `n // 10`. The next cycle will add in the second-to-last digit from the original number, and so on, until all the digits have been processed.

But how do you know when to stop? As you compute `n // 10` in each cycle, you will eventually reach the point at which `n` becomes 0. At that point, you've processed all the digits in the number. Thus, the `while` loop needed for the problem is

```
while n > 0:
    total += n % 10
    n = n // 10
```

The implementation of the `digit_sum` function appears in Figure 2-1.

FIGURE 2-1 **Function to add up the digits in a number**

```
# File: DigitSum.py

"""
This module defines the function digit_sum, which sums the digits
in a number.
"""

def digit_sum(n):
    """Returns the sum of the digits in the nonnegative integer n."""
    total = 0
    while n > 0:
        total += n % 10
        n = n // 10
    return total
```

Aligning output fields

You can also use the `while` statement to add spaces to a string in order to ensure that strings of different lengths line up correctly when displayed on the Python console. For example, columns of numbers are conventionally aligned on the right by adding spaces at the beginning of the number. Although you will discover in Chapter 7 that there is a library function that has the same effect, you can also use the following function, which takes a value and a field width:

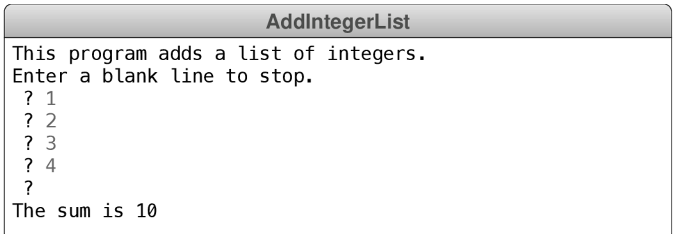
```
def align_right(value, width):
    result = str(value)
    while len(result) < width:
        result = " " + result
    return result
```

The function returns a string in which `value` appears at the right edge of a field that is `width` characters wide. The first line of the function uses the `str` function to convert `value` to a string and then assigns that value to the variable `result`. From here, the function uses concatenation to add spaces to the beginning of `result` until it has attained the desired length. The last statement then returns the padded string to the caller. You will have a chance to see `align_right` in action in section 2.4.

Reading console input until a sentinel appears

Another context in which the `while` loop comes in handy is in programs that read input lines until the user enters a special value that indicates that the data entry process is complete. That special value is called a ***sentinel.*** This style of input has the character required for a `while` loop in that it repeatedly reads lines from the user as long as those lines don't match the sentinel value.

To get a better sense of when this style of operation might come up, imagine that you have been asked to write a program that reads in integers from the user, adding them up as the process goes along. When the input list is complete, the program should display the overall total. The program needs some sentinel to stop in order to ensure that the program doesn't keep asking for numbers forever. The following execution trace shows the operation of an as-yet-unwritten `add_list` function that uses a blank line as the sentinel to mark the end of the input:



```
This program adds a list of integers.
Enter a blank line to stop.
 ? 1
 ? 2
 ? 3
 ? 4
 ?
The sum is 10
```

The structure of this program is closely related to that of the `digit_sum` function shown in Figure 2-1. In pseudocode form, the `add_list` function looks like this:

```
def add_list():
    print("This program adds a list of integers.")
    print("Enter a blank line to stop.")
    total = 0
    For each input value until the sentinel appears, add that number to total.
    print("The sum is", total)
```

What's left is finding a way to express the remaining pseudocode line in English.

Writing the necessary code to implement this operation is not as simple as it might at first appear. As the pseudocode makes clear, the loop should terminate when the input value is equal to the sentinel. In order to check this condition, however, the program must have first read in some value. If the program has not yet read in a value, the termination condition doesn't make sense.

The problem that arises in implementing the read-until-sentinel pattern is that the check for the termination condition appears in the middle of the loop instead of at the beginning. A loop that requires that some operation be performed before testing for completion represents an instance of what programmers call the ***loop-and-a-half problem.***

Python offers at several strategies for solving the loop-and-a-half problem, each of which uses a `while` loop in some form. Unfortunately, none of these strategies is perfect. One strategy for is to read in the first number outside the loop and then execute the loop until the sentinel appears. In this form, the missing statements look like this:

```
line = input(" ? ")
while line != "":
    total += int(line)
    line = input(" ? ")
```

Although this strategy works, there are two aspects of this code that violate one's intuition about the problem. First, the statements in the body of the `while` loop begin by adding a value to `total` and then reading in a line, even though the conceptual order of operations in the pseudocode formulation is to read a line and then add a value. Second, the same exact statement

```
line = input(" ? ")
```

appears twice in the code, even though there is only one conceptual operation. Such instances of repeated code make programs more difficult to maintain, because it is easy to change one instance without changing the other.

A second approach to solving the loop-and-a-half problem is to make use of the `break` statement, which Python has inherited from the programming language C. The `break` statement makes it possible to express the necessary control structure in a form that mirrors the order of operations from the pseudocode, which consists of repeating the following steps:

1. Read in a value.
2. If the value is equal to the sentinel, exit from the loop.
3. Perform whatever processing is required for that value.

Using `break` allows you to code the loop-until-sentinel pattern like this:

```
while True:
    line = input(" ? ")
    if line == "":
        break
    total += int(line)
```

The order of operations now matches the intuitive conception of the process, and the code includes no duplicated lines.

Although several early studies have demonstrated that students are more likely to write correct code if they use this coding model, many computer science experts reject the idea of using the `break` statement because it buries the exit condition inside the body of the loop. This style of programming makes it impossible for readers of the program to tell from the header line of the `while` loop—which on its own makes it seem as if the loop will run forever—without searching through the entire loop body looking for the point at which the loop is complete.

The approach used in this text is to introduce a Boolean variable, which programmers often refer to as a ***flag,*** to keep track of whether the process has finished. Using the flag-based strategy, the implementation has the form shown in Figure 2-2. As long as the `finished` flag is `False`, as it is at the beginning because of the explicit assignment statement that precedes the loop, the code reads in a line from the user and then checks to see whether that line is empty. If so, the first clause of the `if` statement sets `finished` to `True` so the loop will terminate. If not, the `else` clause adds the numeric value of the line to the variable `total`.

Although the flag-based code is slightly longer, this strategy avoids the problems from the earlier approaches and is well worth memorizing for your own programs.

FIGURE 2-2 **Program to add a list of integers**

```
# File: AddIntegerList.py

"""This program adds a list of integers entered by the user."""

def add_integer_list():
    print("This program adds a list of integers.")
    print("Enter a blank line to stop.")
    total = 0
    finished = False
    while not finished:
        line = input(" ? ")
        if line == "":
            finished = True
        else:
            total += int(line)
    print(f"The sum is {total}")

# Startup code

if __name__ == "__main__":
    add_integer_list()
```

2.4 The `for` statement

```
for var in iterable:
    statements
```

The most important control statement in Python is the `for` statement, which is typically used in situations in which you know how many cycles the loop will run before it begins. The general form of the `for` statement appears in the syntax box on the left. When Python encounters a loop of this sort, it executes the statement in the body with the variable indicated by the placeholder *var* set to each element in the collection of values specified by *iterable*. Python uses the term ***iterable*** to specify a data value that supports ***iteration,*** which is the formal term computer scientists use for the process of looping through a collection one value at a time

Iterating over a range of integers

One of the most common uses of the `for` statement is to cycle through a range of integers. In this case, the *iterable* placeholder in the `for` loop paradigm consists of a call to the built-in function `range`, which returns an iterable value whose elements are the desired integers. The `for` loop then executes one cycle for each value.

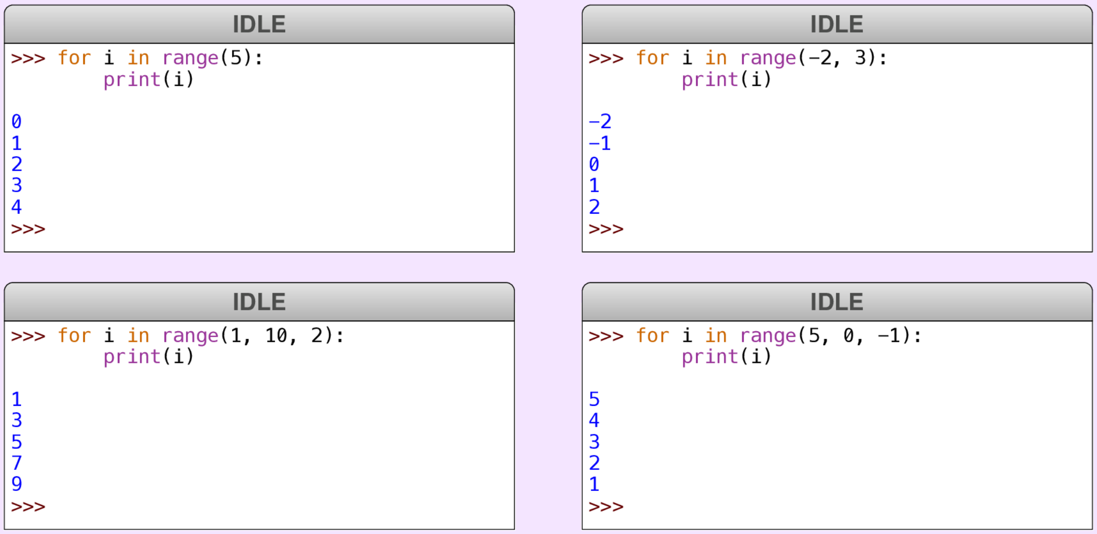
The `range` function offers several different patterns that give you considerable control over the order in which the `for` loop processes the elements. These patterns are determined by the number of arguments, as follows:

- If you call `range` with one argument, as in `range(`*n*`)`, the result generates a sequence of *n* values beginning with 0 and extending up to the value *n* – 1.
- If you call `range` with two arguments in the form `range(`*start*`,` *limit*`)`, the result generates a sequence beginning with *start* and continuing up to but not including the value of *limit.*
- If you call `range` with three arguments in the form `range(`*start*`,` *limit*`,` *step*`)`, the result generates a sequence beginning with *start* and then counts in increments of *step* up to but not including *limit.* If the value of *step* is negative, the sequence begins with *start* and then counts backwards down to but not including *limit.*

Figure 2-3 illustrates each of these argument patterns in the context of a `for` loop that displays each of the values in the range.

The variable that appears in the `for` loop pattern is called an ***index variable.*** In each of the examples in Figure 2-3, the index variable is named `i`. Although using single-letter names can sometimes make programs more difficult to understand, using `i` as an index variable follows a well-established programming convention. Just as the single-letter variables names `x` and `y` are perfectly appropriate if they refer to coordinate values, programmers immediately recognize the variable name `i` as a loop index that cycles through a sequence of integers.

FIGURE 2-3 **Examples of `for` loops using the `range` function**



```
>>> for i in range(5):
        print(i)

0
1
2
3
4
>>>
```

```
>>> for i in range(-2, 3):
        print(i)

-2
-1
0
1
2
>>>
```

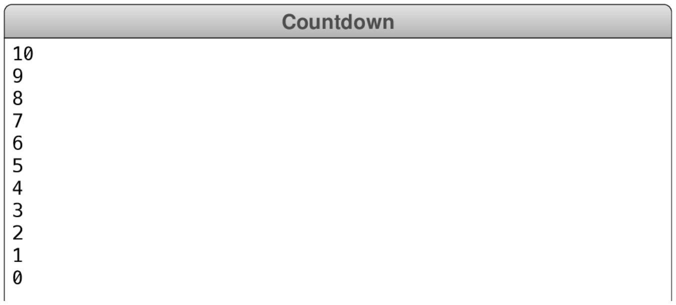
```
>>> for i in range(1, 10, 2):
        print(i)

1
3
5
7
9
>>>
```

```
>>> for i in range(5, 0, -1):
        print(i)

5
4
3
2
1
>>>
```

The last example in Figure 2-3 shows that you can use the `range` function to count backwards. You could use this feature to write a function that simulates a countdown from the early days of the space program:

```
def countdown(n):
    for t in range(n, -1, -1):
        print(t)
```

Calling `countdown(10)` produces the following output on the console:



```
10
9
8
7
6
5
4
3
2
1
0
```

The `countdown` function also demonstrates that any variable can be used as an index variable. In this case, the variable is called `t`, presumably because that is traditional for a rocket countdown, as in the phrase “T minus 10 seconds and counting.”

The factorial function

The ***factorial*** of a nonnegative integer *n,* which is traditionally written as *n*! in mathematics, is defined to be the product of the integers between 1 and `n`. The first ten factorials are shown in the following table:

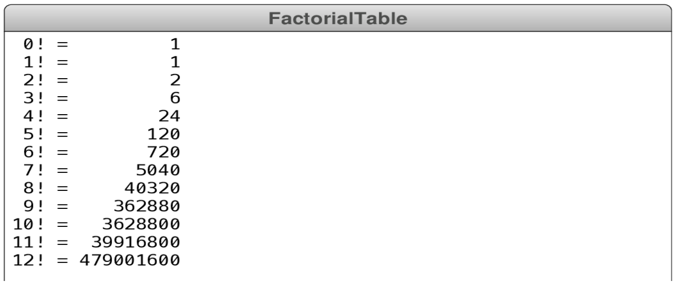
0!	=	1		(by definition)
1!	=	1	=	1
2!	=	2	=	1 × 2
3!	=	6	=	1 × 2 × 3
4!	=	24	=	1 × 2 × 3 × 4
5!	=	120	=	1 × 2 × 3 × 4 × 5
6!	=	720	=	1 × 2 × 3 × 4 × 5 × 6
7!	=	5040	=	1 × 2 × 3 × 4 × 5 × 6 × 7
8!	=	40320	=	1 × 2 × 3 × 4 × 5 × 6 × 7 × 8
9!	=	362880	=	1 × 2 × 3 × 4 × 5 × 6 × 7 × 8 × 9

Factorials have extensive applications in statistics, combinatorial mathematics, and computer science. A function to compute factorials is therefore a useful tool for solving problems in those domains. You can implement a function `fact(n)` by initializing a variable called `result` to 1 and then multiplying `result` by each of the integers between 1 and `n`, inclusive. The resulting code looks like this:

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Note that the `for` loop specifies the upper limit of the range as `n + 1` to ensure that the value `n` is included in the product.

The `FactorialTable.py` program in Figure 2-4 on the next page displays a list of the factorials starting at `LOWER_LIMIT` and extending up to but not including `UPPER_LIMIT`, as illustrated by the following sample run:



```
 0! =         1
 1! =         1
 2! =         2
 3! =         6
 4! =        24
 5! =       120
 6! =       720
 7! =      5040
 8! =     40320
 9! =    362880
10! =   3628800
11! =  39916800
12! = 479001600
```

The code for this program is divided into multiple modules. The main program is stored in the `FactorialTable.py` module shown at the top of Figure 2-4. The code in `FactorialTable.py` produces the table shown in the sample run but relies on a separate `factorial.py` model for the `fact` function and an `alignment.py` module (which you will have a chance to write in exercise 9) for the `align_right` function defined on page 47.

The module that together define the `FactorialTable` application play slightly different roles. The `FactorialTable.py` module defines a Python ***program*** that delivers output to the user through the use of `print` statements. The `factorial.py` and the as-yet-unwritten `alignment.py` modules each represent a Python ***library*** that performs a service for the main program without communicating directly with the user. The functions in the library modules communicate with their callers by taking arguments as input and returning results.

The names of these modules reflect a convention that applies throughout this text. Modules that are intended to be run as programs use camel-case names beginning with an uppercase letter, as illustrated by the module name `FactorialTable.py`. Modules intended to be used as libraries, such as `factorial.py` and `alignment.py` in this chapter or like the `temperature.py` module from Chapter 1 have names written entirely in lowercase letters.

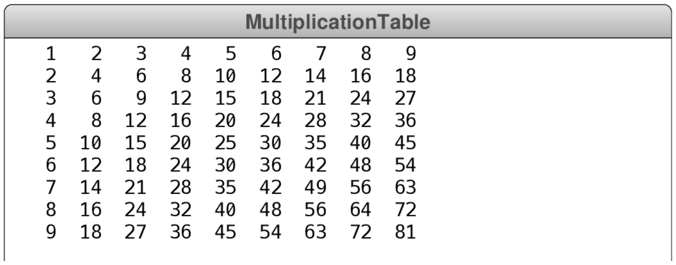
FIGURE 2-4 Program to display a list of factorials on the console

```
# File: FactorialTable.py

"""This program prints a table of factorials."""

from alignment import align_right
from factorial import fact

# Constants

LOWER_LIMIT = 0
UPPER_LIMIT = 13
INDEX_DIGITS = 2
FACTORIAL_DIGITS = 9

# Implementation notes: print_factorial_table
# -------------------------------------------
# Prints a table of factorials in the Pythen range(LOWER_LIMIT, UPPER_LIMIT),
# which starts at LOWER_LIMIT but stops just before UPPER_LIMIT.  The
# fact function itself is imported from the factorial.py module.  The
# values of both the index and the corresponding factorial are aligned on
# the right using the align_right function from the alignment module.

def print_factorial_table():
    """Displays a table of factorials."""
    for i in range(LOWER_LIMIT, UPPER_LIMIT):
        print(align_right(i, INDEX_DIGITS) + "! = " +
              align_right(fact(i), FACTORIAL_DIGITS))

# Startup code

if __name__ == "__main__":
    print_factorial_table()
```

```
# File: factorial.py

"""
This module defines the fact function, which calculates the factorial
of an integer n (traditionally written in mathematics as n!).  The
result is the product of the numbers between 1 and n, inclusive.
"""

def fact(n):
    """Returns the factorial of n."""
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Nested for statements

In many applications, you will discover that you need to write one for loop inside another so that the statements in the innermost loop are executed for every possible combination of values of the for loop indices. Suppose, for example, that you want to display a multiplication table showing the product of every pair of numbers in the range 1 to 10. You would like the output of the program to look like this:



```
   1   2   3   4   5   6   7   8   9
   2   4   6   8  10  12  14  16  18
   3   6   9  12  15  18  21  24  27
   4   8  12  16  20  24  28  32  36
   5  10  15  20  25  30  35  40  45
   6  12  18  24  30  36  42  48  54
   7  14  21  28  35  42  49  56  63
   8  16  24  32  40  48  56  64  72
   9  18  27  36  45  54  63  72  81
```

The code to draw this multiplication table appears in Figure 2-5. To create the individual entries, you need a pair of nested for loops: an outer loop that runs through each of the rows and an inner loop that runs through each of the entries in each row. The code inside the inner for loop will be executed once for every row and column, for a total of 100 individual entries in the table.

FIGURE 2-5 Program to display a multiplication table

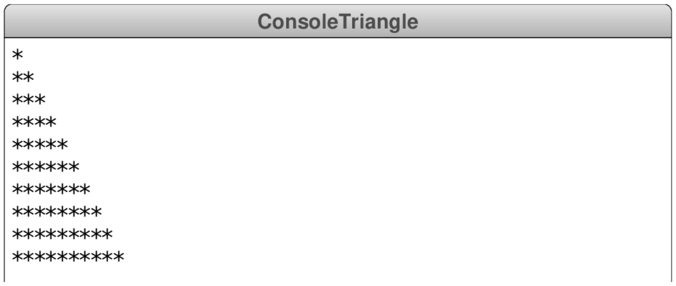
```python
# File: MultiplicationTable.py

from alignment import align_right

# Constants

UPPER_LIMIT = 10
FIELD_WIDTH = 4

# Main program

def print_multiplication_table():
    """Prints a multiplication table on the console."""
    for i in range(1, UPPER_LIMIT):
        line = ""
        for j in range(1, UPPER_LIMIT):
            line += align_right(i * j, FIELD_WIDTH)
        print(line)

# Startup code

if __name__ == "__main__":
    print_multiplication_table()
```

The outer loop runs through each value of i from 1 to 10 and is responsible for displaying one row of the table on each cycle. To do so, the code first declares the variable line and initializes it to be the empty string. The inner loop then runs through the values of j from 1 to 10 and concatenates the product of i and j to the end of line after calling the align_right function to ensure that the columns have the same width. When the inner loop is complete, the program calls print to display the completed line of the multiplication table.

A useful way to get some practice using nested for loops is to write programs that draw patterns on the console by displaying lines of characters. As a simple example, the following function draws a triangle in which the number of stars increases by one in each row:

```
def draw_console_triangle(size):
    for i in range(size):
        line = ""
        for j in range(i + 1):
            line += "*"
        print(line)
```

Calling draw_console_triangle(10), for example, produces the following output on the console:



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

You will have a chance to create several similar displays in the exercises.

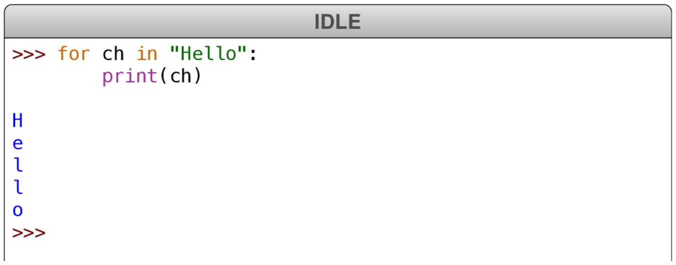
Iterating over sequences

The examples of the for statement you have seen so far all use the range function to specify the sequence of values. If you look back at the original description of the for loop pattern, however, you will see that header line has the following more general format:

```
for var in iterable:
```

Python includes several data types that support iteration, any of which can be used in place of the *iterable* component of this pattern. In many cases, those data types represent sequences of individual values. Given any sequence, you can use the `for` statement to step through the elements of that sequence one value at a time. The first cycle of the `for` loop sets the index variable to the first element in the sequence, the second cycle sets the variable to the second element, and the process continues in this fashion through the entire sequence.

Even though you won't encounter most of Python's iterable types until later in this book, you have already seen two iterable types, which are the string and list type introduced in Chapter 1. Conceptually, a string is a sequence of characters. As with any sequence, you can use a `for` loop to step through each of these characters in turn. The following IDLE session, for example, uses a `for` loop to display the characters in the string `"Hello"`, one character per line:



```
>>> for ch in "Hello":
        print(ch)

H
e
l
l
o
>>>
```

When Python executes this `for` loop, it interprets `"Hello"` as a sequence of one-character strings and then assigns each of those strings to the index variable `ch` on successive cycles of the loop, starting with `"H"` and continuing through `"o"`.

You can use this strategy of iterating through each character in a string to implement a function `count_char(c, s)`, which returns the number of times the character `c` appears in the string `s`:

```
def count_char(c, s):
    count = 0
    for ch in s:
        if c == ch:
            count += 1
    return count
```

For example, calling `count_char("u", "unusual")` returns the value 3 because the character `"u"` appears three times in the string `"unusual"`.

You will have more opportunities to use the `for` loop with strings in Chapter 7 and with more general sequences beginning in Chapter 8.

The `reversed` and `sorted` functions

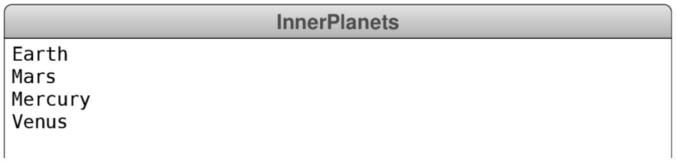
Python’s library of built-in functions includes two that take one iterable value as an argument and return another that cycles through the same values but does so in a different order. The `reversed` function returns an iterable value that runs through its elements backwards. For example, you can use `reversed` to rewrite the `countdown` function from page 52 as follows:

```
def countdown(n):
    for t in reversed(range(0, n + 1)):
        print(t)
```

The built-in `sorted` function takes any iterable object and return a list that contains the elements from that object in sorted order. For example, the code sequence

```
INNER_PLANETS = [ "Mercury", "Venus", "Earth", "Mars" ]
for planet in sorted(INNER_PLANETS):
    print(planet)
```

produces the following output on the console:



```
Earth
Mars
Mercury
Venus
```

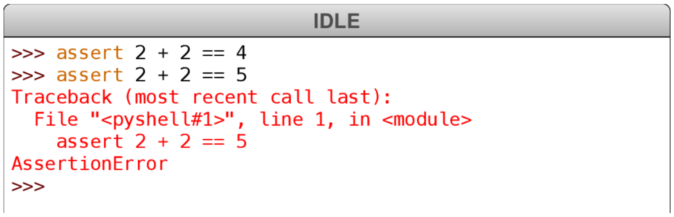
Similarly, calling `sorted` on a string (which is, after all, an iterable object whose elements are the individual characters) returns a sorted list of those characters. For example, calling `sorted("word")` returns the list `[ "d", "o", "r", "w" ]`. You will have occasion to use this function in Chapter 3.

2.5 The `assert` statement

`assert` *test*

The last control statement covered in this chapter is the `assert` statement, which typically has the form shown in the syntax box on the left. Unlike the other control statements, the `assert` statement doesn’t include a body; all you have is a conditional tesr, which should be a Boolean expression of the form used in the `if` and `while` statements. The `assert` statement represents a declaration of something that you, as the programmer, believe to be true at that point in the execution of the program and asks Python to verify that fact for you. If the test is indeed true, Python moves on to the next statement. If not, Python displays the failed assertion and terminates the program execution with a failure condition called an ***assertion error.***

The following sample run illustrates the operation of the `assert` statement in the context of the IDLE interpreter:



```
>>> assert 2 + 2 == 4
>>> assert 2 + 2 == 5
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert 2 + 2 == 5
AssertionError
>>>
```

The first line, which makes the generally uncontroversial statement that two plus two is four, executes silently, producing no output. The second line, which claims (as O'Brien does in the George Orwell novel *Nineteen Eighty-Four*) that two plus two is five, generates an error message showing the offending assertion.

The most useful application of the `assert` statement comes in writing functions that test the operation of a module. For example, in the library `factorial.py` module shown in Figure 2-4 on page 54, it would be good programming practice to add a `test_fact` function composed of several `assert` statements checking that the `fact` function computed the correct result. That function might look like this:

```
def test_fact():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(2) == 2
    assert fact(5) == 120
    assert fact(10) == 3628800
    assert fact(20) == 2432902008176640000
```

You can't possibly test all possible values for the arguments, but you can select specific values that give you confidence in the correctness of the implementation. Here, for example, the code tests the value of `fact(0)`, which is defined to be 1, and the value for a reasonably large argument like 20.

You can also use `assert` statements to verify that the arguments that the caller passes to a library function meet the conditions required for a correct result. For example, the `fact` function is defined only for nonnegative integers. Adding the following `assert` statement to check that requirement can simplify later debugging:

```
    assert isinstance(n, int) and n >= 0
```

This statement uses the built-in `isinstance` function, which checks whether the first argument has the type specified by the second. A version of the `factorial.py` module that incorporates both these changes appears in Figure 2-6.

FIGURE 2-6 The `factorial.py` module including a test program

```python
# File: factorial.py

"""
This module defines the fact function, which calculates the factorial of
a nonnegative integer n (traditionally written in mathematics as n!).
The result is the product of the numbers between 1 and n, inclusive.
"""

def fact(n):
    """Returns the factorial of n."""
    assert isinstance(n, int) and n >= 0
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Test program

def test_fact():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(2) == 2
    assert fact(5) == 120
    assert fact(10) == 3628800
    assert fact(20) == 2432902008176640000

if __name__ == "__main__":
    test_fact()
```

Summary

The purpose of this chapter is to introduce the most common control statements in Python and explore examples of their use. The important points include:

- One of the most useful types in any modern programming language is *Boolean data,* for which the domain consists of just two values: `True` and `False`.
- The *relational operators* (<, <=, >, >=, ==, and `!=`) perform comparisons to create Boolean values. The *logical operators* (`and`, `or`, and `not`) combine Boolean values to express more complex conditions.
- The logical operators `and` and `or` are evaluated in left-to-right order in such a way that the evaluation stops as soon as the program can determine the result. This behavior is called *short-circuit evaluation*.
- Functions that return Boolean values play an important role in computer science and are called *predicate functions*.
- Control statements fall into two classes: *conditional* and *iterative*.

- The `if` statement specifies conditional execution when a section of code should be executed only in certain cases or when the program needs to choose between two alternate paths.
- The `while` statement specifies repetition as long as some condition is met.
- In some applications, it is necessary to perform part of a `while` loop before checking whether the termination condition applies. Programmers refer to this situation as the *loop-and-a-half problem*. Several strategies exist for coding such loops, none of which is perfect. This book recommends using a flag to keep track of whether the loop has finished.
- A particular common example of the loop-and-a-half problem arises when a program needs to read input lines until a blank line appears. This book uses the following code pattern to implement this structure:

```
finished = False
while not finished:
    line = input(" ? ")
    if line == "":
        finished = True
    else:
        perform some operation using the input line
```

- The `for` statement is used to cycle through every value in an iterable object.
- Most of the `for` loops used in this chapter specify the limits of the iteration using the built-in function `range`, which can take one, two, or three arguments. Calling `range(`*n*`)` generates a sequence of *n* values beginning with 0 and extending up to the value $n - 1$. The two-argument form `range(`*start*`,` *limit*`)` iterates through a sequence beginning with *start* and continuing up to but not including the value of *limit.* The three-argument form `range(`*start*`,` *limit*`,` *step*`)` iterates through a sequence beginning with *start* and then counts in increments of *step* up to but not including the value of *limit.*
- You can use the `for` statement to step through every value in a sequence, such as a string or a list.
- The built-in functions `reversed` and `sorted` each convert an iterable object into a new one that cycles through the elements in a different order. The `reversed` function produces an iterator that runs backwards; the `sorted` function produces a list in which the elements appear in sorted order.
- The `assert` statement asks Python to check whether some Boolean condition holds. You can use the `assert` statement to write test functions for your modules or to check that functions in a library are called with valid arguments.
- The built-in function `isinstance(`*value*`,` *type*`)` checks to see whether *value* has the specified type.

Review questions

1. What are the Python keywords for the two Boolean values?

2. Describe in English what the following conditional expression means:

   ```
   (x != 4) or (x != 17)
   ```

   For what values of `x` is this condition equal to `True`?

3. What is meant by the term *short-circuit evaluation?*

4. What is a *predicate function?*

5. What are the two classes of control statements?

6. What does it mean to say that two control statements are *nested?*

7. Suppose the body of a `while` loop contains a statement that, when executed, causes the condition for that `while` loop to become `False`. Does the loop terminate immediately at that point or does it complete the current cycle?

8. What is the *loop-and-a-half problem?*

9. What programming pattern does these notes recommend for reading input lines until a blank line appears?

10. What term do computer scientists use to refer to an incomplete program written partly in a programming language and partly in English?

11. Describe the sequence of values generated by each of the following calls to the built-in function `range`:

    a) `range(7)`
    b) `range(1, 10)`
    c) `range(5, 25, 5)`
    d) `range(1, -2, -2)`

12. What `for` loop header line would you use in each of the following situations:

    a) Counting from 1 to 100.
    b) Counting by sevens starting at 0 until the number has more than two digits.
    c) Counting backward by twos from 100 to 0.

13. How would you write a `for` loop to cycle through the characters in a string `s`?

14. Describe briefly the built-in functions `reversed`, `sorted`, and `isinstance`.

15. What two applications does the chapter describe for the `assert` statement?

Exercises

1. As a way to pass the time on long bus trips, young people growing up in the United States have been known to sing the following rather repetitive song:

   99 bottles of beer on the wall.
   99 bottles of beer.
   You take one down, pass it around.
   98 bottles of beer on the wall.

   98 bottles of beer on the wall. . . .

   Anyway, you get the idea. Write a Python program to display the lyrics of this song using `print`. In testing your program, it would make sense to use some constant other than 99 as the initial number of bottles.

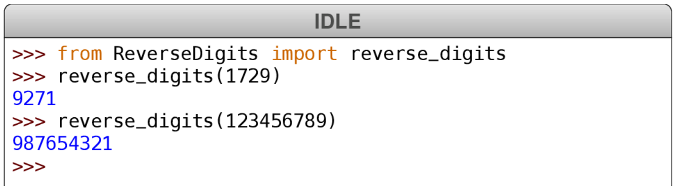
2. Write a function that takes a positive integer *N* and then calculates and displays the sum of the first *N* odd integers. For example, if *N* is 4, your function should display the value 16, which is 1 + 3 + 5 + 7.

3. *Why is everything either at sixes or at sevens?*

   —Gilbert and Sullivan, *H.M.S. Pinafore,* 1878

   Write a program that displays the integers between 1 and 100 that are divisible by either 6 or 7 but not both.

4. Use the `digit_sum` function as a model to define a function that takes a number and returns a number that contains the same digits in the reverse order, as illustrated by the following IDLE transcript:



```
>>> from ReverseDigits import reverse_digits
>>> reverse_digits(1729)
9271
>>> reverse_digits(123456789)
987654321
>>>
```

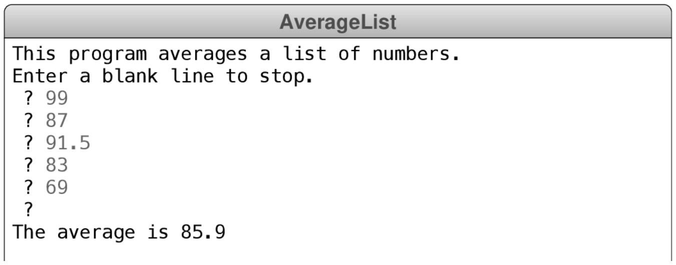
   The idea in this exercise is not to take the integer apart character by character, which you will not learn how to do until Chapter 7. Instead, you need to use arithmetic to compute the reversed integer as you go.

5. The ***digital root*** of an integer *n* is defined as the result of summing the digits repeatedly until only a single digit remains. For example, the digital root of 1729 can be calculated using the following steps:

Step 1:	1 + 7 + 2 + 9	→	19
Step 2:	1 + 9	→	10
Step 3:	1 + 0	→	1

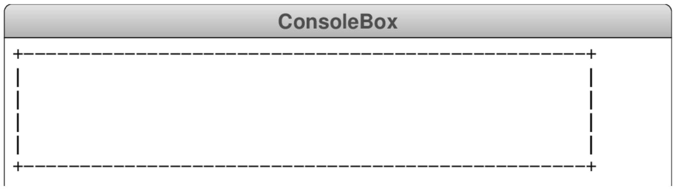
Because the total at the end of step 3 is the single digit 1, that value is the digital root. Write a function `digital_root` that returns this value.

6. Write a Python program that reads in numbers until the user enters a blank line and then prints their average. A sample run of your program might look like this:



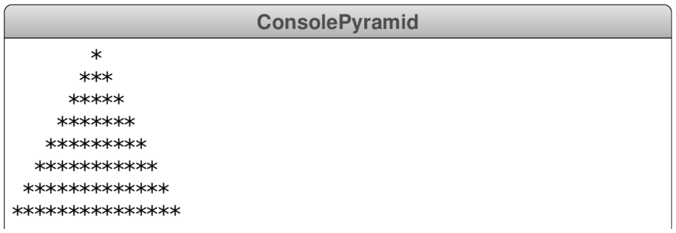
```
This program averages a list of numbers.
Enter a blank line to stop.
 ? 99
 ? 87
 ? 91.5
 ? 83
 ? 69
 ?
The average is 85.9
```

7. Write a function `draw_console_box(width, height)` that draws a box on the console with the specified dimensions. The corners of the box should be represented using a plus sign (+), the top and bottom borders using a minus sign (–), and the left and right borders using a vertical bar (|). For example, calling `draw_console_box(52, 6)` should produce the following diagram:



```
+--------------------------------------------------+
|                                                  |
|                                                  |
|                                                  |
|                                                  |
+--------------------------------------------------+
```

8. Write a function `draw_console_pyramid(height)` that draws a pyramid of the specified height in which the width of each row increases by two as you move downward on the console. Each of the rows should be centered with respect to the others, and the bottom line should begin at the left margin. Thus, calling `draw_console_pyramid(8)` should produce the following figure:



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

9. Implement a library module called `alignment.py` that defines the function `align_right` as given on page 47 along with the corresponding functions `align_left` and `align_center`. Make sure that your module includes a test function.

CHAPTER 3

Algorithmic Thinking

Computational thinking is a fundamental skill for everyone, not just for computer scientists. To reading, writing, and arithmetic, we should add computational thinking to every child's analytical ability.

— Jeannette Wing, "Computational Thinking,"
Communications of the ACM, 2006

Jeannette Wing (1956–)

Jeannette Wing received her Ph.D. in Computer Science from MIT in 1983 and has subsequently held faculty positions at the University of Southern California, Carnegie Mellon, and Columbia University. In addition to her academic work, Wing has served as Corporate Vice President of Microsoft Research and as a research director at the National Science Foundation. In 2006, during her time as head of the Computer Science Department at Carnegie Mellon, Wing published an influential article entitled "Computational Thinking" in *Communications of the ACM,* the flagship journal of the leading professional society for computing. In that article, Wing argued that every person growing up today needs to understand not only what computation can do but also how to unlock that power using the patterns of thought that studying computer science fosters. As Wing notes in her article, "computational thinking will have become ingrained in everyone's lives when words like *algorithm* . . . are part of everyone's vocabulary."

The concept of an algorithm is fundamental to computer science. The word *algorithm* comes from the name of the 9th-century Persian mathematician Muhammad ibn Mūsā al-Khwārizmī, whose work had significant impact on modern mathematics. Figure 3-6 shows a photograph of a statue of al-Khwārizmī near his birthplace in what is now Uzbekistan.

Although it is usually sufficient to think of an algorithm as a strategy for solving a problem, modern computer science formalizes that definition so that ***algorithm*** refers to a solution strategy that is

- *Clear and unambiguous,* in the sense that the description is understandable.
- *Effective,* in the sense that it is possible to carry out the steps in the strategy.
- *Finite,* in the sense that the strategy terminates after some number of steps.

This chapter begins by looking at a few historically important algorithms and then shifts its focus toward the process of designing your own solution strategies. The final section then explores topics in testing, debugging, and software maintenance.

FIGURE 3-1 **Statue of al-Khwārizmī outside the gates of Khiva, Uzbekistan**

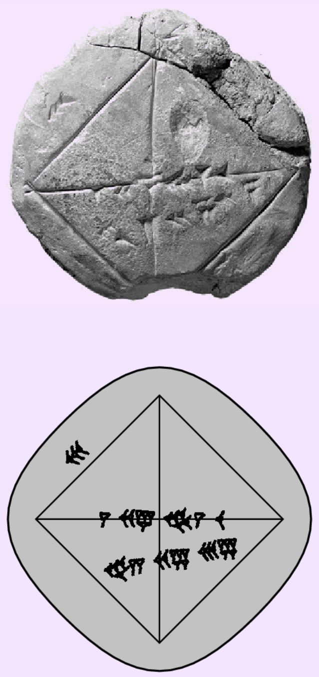
3.1 Algorithms in history

Although the concept of an algorithm has taken on new significance in the computing age, algorithmic techniques for solving problems have existed even before the time of al-Khwārizmī. The next few sections offer a few examples of historically important algorithms. These algorithms are interesting in their own right, but they also offer useful illustrations of how control statements can be used in practice.

An early square-root algorithm

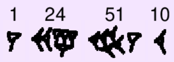
One of the earliest known algorithms dates back almost 4000 years to when Babylonian mathematicians discovered a procedure for calculating square roots. The primary evidence of the existence of an algorithmic process comes from cuneiform tablets such as the one shown in Figure 3-2, which shows an approximation of the square root of 2 that is far more accurate than anyone could possibly derive through measurement alone. And although the precise details of how Babylonian mathematicians performed the necessary calculations have been lost, historians believe that their technique was similar to the algorithm described by the 1st-century Greek mathematician Hero of Alexandria, who noted its Babylonian origin.

FIGURE 3-2 Babylonian cuneiform tablet showing an approximation of the square root of 2



The cuneiform fragment on the left dates from the First Babylonian Dynasty, which ran from the 19th century to the 16th century BCE. The diagram consists of a square and its diagonals together with three numbers written using Babylonian numerals. Those numerals are barely visible in the photograph but are reproduced in the stylized diagram at the bottom of the figure.

The numerals across the horizontal diagonal have the following values:

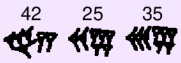


Babylonian arithmetic uses a base-60 system, which means that each digit counts for one-sixtieth of the digit position to its left. This sequence of digits therefore corresponds to the following calculation:

$$1 + \frac{24}{60} + \frac{51}{60 \times 60} + \frac{10}{60 \times 60 \times 60} \approx 1.414213$$

This value is the closest possible representation of the square root of 2 in four base-60 digits.

The two other numbers on the tablet illustrate the relationship between the length of the side and the diagonal. If the length of the side is 30 (the 𒌍 in the upper left), the length of the diagonal is approximately



$$= 42 + \frac{25}{60} + \frac{35}{60 \times 60} \approx 42.4264$$

The Babylonian method for calculating square roots is an example of a general technique called ***successive approximation,*** in which you begin by making a rough guess at the answer and then improve that guess through a series of refinements that get closer and closer to the exact answer. For example, if you want to find the square root of some number *n,* you start by choosing some smaller number *g* as your first guess. At every point in the process, your guess *g* will be smaller or larger than the actual square root. In either case, if you divide *n* by *g,* the result will inevitably lie on the opposite side of the desired value. For example, if *g* is too small, *n* divided by *g* will be too large, and vice versa. Averaging the two values will always give a better approximation. At each step, you simply replace your previous guess *g* with the result of the following formula, which averages *g* and *n* divided by *g:*

$$\frac{g + \frac{n}{g}}{2}$$

You then continue to apply this formula to each new guess until the answer is as close to the actual value as you need it to be.

To get more of a sense of how the Babylonian method works, it helps to consider a simple example. Suppose that you want to calculate, as the scribes who incised the cuneiform tablet did, the square root of 2. One possible first guess for *g* is 1, which is half the value of *n.* The first approximation step therefore computes the following average:

$$\frac{1 + \frac{2}{1}}{2} = \frac{3}{2} = 1.5$$

The value 1.5 is closer to the actual square root of 2—which is approximately 1.4142136—so the process is on the right track.

To calculate the next approximation, all you need to do is plug $\frac{3}{2}$ into the formula as the next value of *g,* and calculate the new average, as follows:

$$\frac{\frac{3}{2} + \frac{2}{\frac{3}{2}}}{2} = \frac{17}{12} \approx 1.4166667$$

From this point, you simply repeat the calculation with $\frac{17}{12}$ as the new value of *g:*

$$\frac{\frac{17}{12} + \frac{2}{\frac{17}{12}}}{2} = \frac{577}{408} \approx 1.4142157$$

Applying successive approximation one more time gives you

$$\frac{\frac{577}{408} + \frac{2}{\frac{577}{408}}}{2} = \frac{665857}{470832} \approx 1.4142136$$

After just four cycles, the Babylonian method has produced an approximation to the square root of 2 that is correct to eight decimal digits. Moreover, because each step generates an approximation that is closer to the exact value, you can repeat the process to produce an approximation with any desired level of accuracy.

Figure 3-3 shows the definition of a `sqrt` function that uses the Babylonian method to approximate the square root of its argument. The function uses a `while` loop to continue the process until the approximation reaches the desired level of precision. In this implementation, the `while` loop continues until the difference between the square of the current approximation and the original number is no larger than the value of the constant `TOLERANCE`.

Finding the greatest common divisor

Although you have seen a few simple algorithms implemented in the context of the programming examples, you have had little chance to focus on the nature of the algorithmic process itself. Most of the programming problems you have seen so far are simple enough that the appropriate solution strategy springs immediately to mind.

FIGURE 3-3 **Function to compute square roots using the Babylonian algorithm**

```
# File: BabylonianSquareRoot.py

"""
This module implements a function sqrt that calculates square roots
using the Babylonian method, which operates as follows:

1. Choose a guess g (any value will do; this code uses n / 2).
2. Compute a new guess by averaging g and n / g.
3. Repeat step 2 until the error is less than the desired tolerance.
"""

# Constants

TOLERANCE = 0.000000000000001

# Function: sqrt

def sqrt(n):
    """Calculates the square root of n using the Babylonian method."""
    g = n / 2
    while abs(n - g * g) > TOLERANCE:
        g = (g + n / g) / 2
    return g
```

As problems become more complex, however, their solutions require more thought, and you will need to consider more than one strategy before writing the final program.

As an illustration of how algorithmic strategies take shape, the sections that follow consider two solutions to another problem from classical mathematics, which is to find the greatest common divisor of two integers. Given two integers *x* and *y,* the ***greatest common divisor*** (or ***gcd*** for short) is the largest integer that divides evenly into both. For example, the gcd of 49 and 35 is 7, the gcd of 6 and 18 is 6, and the gcd of 32 and 33 is 1.

Suppose that you have been asked to write a function that accepts two positive integers *x* and *y* as input and returns their greatest common divisor. From the caller's point of view, what you want is a function `gcd(x, y)` that takes the two integers as arguments and returns another integer that is their greatest common divisor. The header line for this function is therefore

```
def gcd(x, y):
```

In many ways, the most obvious approach is simply to try every possibility. To start, you simply "guess" that `gcd(x, y)` is the smaller of `x` and `y`, because any larger value could not possibly divide evenly into a smaller number. You then proceed by dividing `x` and `y` by your guess and seeing if it divides evenly into both. If it does, you have the answer; if not, you subtract 1 from your guess and try again. A strategy that tries every possibility is often called a ***brute-force approach.***

The brute-force approach to calculating the `gcd` function looks like this in Python:

```
def gcd(x, y):
    guess = min(x, y)
    while x % guess != 0 or y % guess != 0:
        guess -= 1
    return guess
```

Before you decide that this implementation is in fact a valid algorithm for computing the `gcd` function, you need to ask yourself several questions about the code. Will the brute-force implementation of `gcd` always give the correct answer? Will it always terminate, or might the function continue forever?

To determine whether the program gives the correct answer, you need to look at the condition in the `while` loop, which looks like this:

```
x % guess != 0 or y % guess != 0
```

As always, the `while` condition indicates under what circumstances the loop will continue. To find out what condition causes the loop to terminate, you have to negate the `while` condition. Negating a condition involving logical operators is tricky unless you remember De Morgan's laws, which were introduced in Chapter 2. De Morgan's laws indicate that the following condition must hold when the `while` loop exits:

```
x % guess == 0 and y % guess == 0
```

From this condition, you can see immediately that the final value of `guess` is certainly a common divisor. To recognize that it is in fact the greatest common divisor, you have to think about the strategy embodied in the `while` loop. The critical factor to notice in the strategy is that the program counts *backward* through all the possibilities. The greatest common divisor can never be larger than `x` or `y`, and the brute-force search therefore begins with the smaller of these two values. If the program ever gets out of the `while` loop, it must have already tried each value between the starting point and the current value of `guess`. Thus, if there were a larger value that divided evenly into both `x` and `y`, the program would already have found it in an earlier iteration of the `while` loop.

In recognizing that the function terminates, the key insight is that the value of `guess` must eventually reach 1, unless a larger common divisor is found. At this point, the `while` loop will surely terminate, because 1 will divide evenly into both `x` and `y`, no matter what values those variables have.

Brute force is not, however, the only effective strategy. Although brute-force algorithms have their place in other contexts, they are a poor choice for the `gcd` function if you are concerned about efficiency. For example, if you call

```
gcd(1000005, 1000000)
```

the brute-force algorithm will run through the body of the `while` loop almost a million times before it comes up with the answer 5, even though you can instantly arrive at that result just by thinking about the two numbers.

What you need to find is an algorithm that is guaranteed to terminate with the correct answer but that requires fewer steps than the brute-force approach. This is where cleverness and a clear understanding of the problem pay off. Fortunately, the necessary creative insight was described sometime around 300 BCE by the Greek mathematician Euclid, whose *Elements* (book 7, proposition II) contains an elegant solution to this problem. In modern English, Euclid's algorithm can be described as follows:

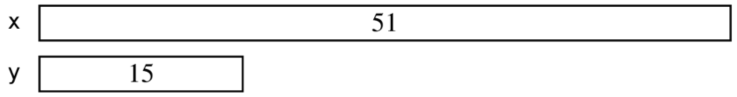
1. Divide `x` by `y` and compute the remainder; call that remainder `r`.
2. If `r` is zero, the procedure is complete, and the answer is `y`.
3. If `r` is not zero, set `x` equal to the old value of `y`, set `y` equal to `r`, and repeat the entire process.

You can easily translate this algorithmic description into the following code:

```
def gcd(x, y):
    r = x % y
    while r != 0:
        x = y
        y = r
        r = x % y
    return y
```

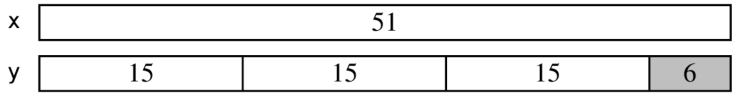
This implementation of the gcd function also correctly finds the greatest common divisor of two integers. It differs from the brute-force implementation in two respects. On the one hand, it computes the result much more quickly. On the other, it is more difficult to prove correct.

Although a formal proof of correctness for Euclid's algorithm is beyond the scope of this book, you can easily get a feel for how the algorithm works by adopting the mental model of mathematics the Greeks used. In Greek mathematics, geometry held center stage, and numbers were thought of as distances. For example, when Euclid set out to find the greatest common divisor of two whole numbers, such as 51 and 15, he framed the problem as one of finding the longest measuring stick that could be used to mark off each of the two distances involved. Thus, you can visualize the specific problem by starting out with two sticks, one 51 units long and one 15 units long, as follows:



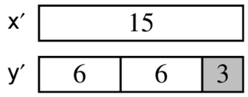
The problem is to find a new measuring stick that you can lay end to end on top of each of these sticks so that it precisely covers each of the distances x and y.

Euclid's algorithm begins by marking off the large stick in units of the shorter one, like this:

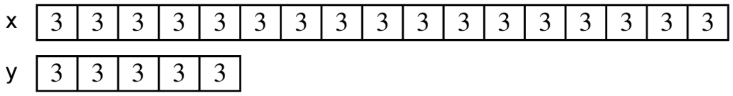


Unless the smaller number is an exact divisor of the larger one, there is some remainder, as indicated by the shaded section of the lower stick. In this case, 15 goes into 51 three times with 6 left over, which means that the shaded region is 6 units long. The fundamental insight that Euclid had is that the greatest common divisor for the original two distances must also be the greatest common divisor of the length of the shorter stick and the length of the shaded region in the diagram.

Given this observation, you can solve the original problem by reducing it to a simpler problem involving smaller numbers. Here, the new numbers are 15 and 6, and you can find their greatest common divisor by reapplying Euclid's algorithm. You start by representing the new values, `x′` and `y′`, as measuring sticks of the appropriate length. You then mark off the larger stick in units of the smaller one.



Once again, this process results in a leftover region, which this time has length 3. If you then repeat the process one more time, you discover that the shaded region of length 3 is itself the common divisor of `x′` and `y′` and, therefore, by Euclid's proposition, of the original numbers `x` and `y`. That 3 is indeed a common divisor of the original numbers is demonstrated by the following diagram:



Euclid supplies a complete proof of his proposition in the *Elements*. If you are intrigued by how mathematicians thought about such problems more than 2000 years ago, you may find it interesting to look up translations of the original Greek text.

Although Euclid's algorithm and the brute-force algorithm correctly compute the greatest common divisor of two integers, there is an enormous difference in the efficiency between the two algorithmic strategies. Suppose once again that you call

```
gcd(1000005, 1000000)
```

The brute-force algorithm requires on the order of a million steps to find the answer; Euclid's algorithm requires only two. At the beginning of Euclid's algorithm, `x` is `1000005`, `y` is `1000000`, and `r` is set to `5` during the first cycle of the loop. Since the value of `r` is not 0, the program sets `x` to `1000000`, sets `y` to `5`, and starts again. On the second cycle, the new value of `r` is `0`, so the program exits from the `while` loop and reports that the answer is 5.

The two strategies for computing greatest common divisors presented in this section offer a clear demonstration that the choice of algorithm can have a profound effect on the efficiency of the solution. In Chapter 9, you will learn how to quantify such differences in performance along with several general approaches for improving algorithmic efficiency.

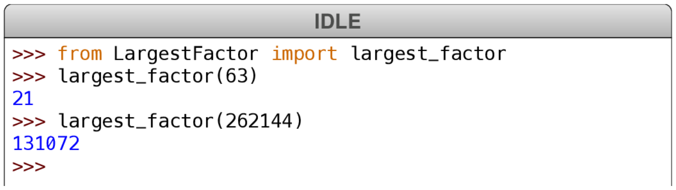
The first program in modern computing history

One problem that has particular relevance to the history of modern computing is that of finding the largest factor of an integer, which is believed to be the first programs executed on the Small-Scale Experimental Machine at Manchester University—the first computer to implement the stored-program architecture that is used in essentially all computers today. The author of the program was Tom Kilburn, the lead engineer on the team that built the machine, which its inventors nicknamed the “Baby.”

Because the Baby had extremely limited capabilities, Kilburn’s solution strategy had to be almost absurdly simple. Given a number N, the program used the brute-force strategy of counting down from $N - 1$ until it found a divisor is found. Taking some advantage of Python’s extended set of operations, Kilburn’s algorithm might look like this:

```
def largest_factor(n):
    factor = n - 1
    while n % factor != 0:
        factor -= 1
    return factor
```

The following IDLE log shows the results from two calls to the `largest_factor` function:



```
>>> from LargestFactor import largest_factor
>>> largest_factor(63)
21
>>> largest_factor(262144)
131072
>>>
```

When the second of these calculations was run on the Manchester Baby on June 21, 1948, the program took 52 minutes to compute the answer. In the process, it demonstrated both the efficacy and the reliability of the Baby’s architecture.

3.2 Devising your own algorithms

In my many years of experience teaching programming, I am convinced that the students who experience the most trouble are those who believe that the computer is a magical device for which they have not yet learned the proper incantations. When faced with a programming problem, those students believe that they don’t know what to do and look instead for some essentially mechanical procedure—an algorithm, if you will—for turning a programming problem into working code. No such algorithm exists. Each new programming problem requires creativity to apply the tools you already know to come up with a solution.

When you start working on a programming problem, the most important question to ask is how *you* would solve the problem if you didn't have a computer. If you can figure that out, the actual process of writing the code becomes much easier. You need to design an algorithm specifically tailored for the problem at hand.

Devising such an algorithm, of course, may not be easy. I certainly wouldn't expect most students to discover Euclid's algorithm for finding the greatest common divisor on their own. At the same time, anyone who has learned basic arithmetic should be able to design and implement the brute-force algorithm that counts down from the smaller number until it finds a number that divides evenly into both of the numbers in question. The code, which you have already seen on page 70, is worth repeating:

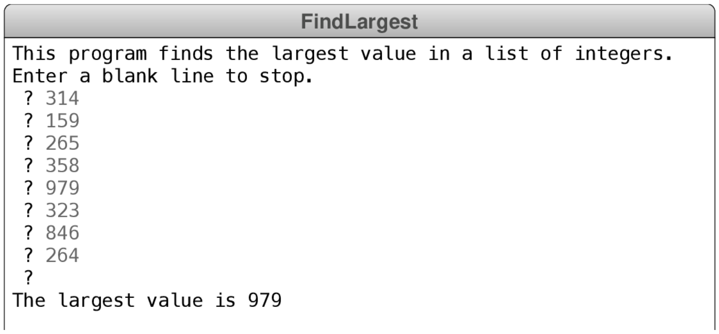
```
def gcd(x, y):
    guess = min(x, y)
    while x % guess != 0 or y % guess != 0:
        guess -= 1
    return guess
```

There is undoubtedly some complexity involved in coding the condition for the `while` loop, but the overall approach is straightforward. Similarly, it doesn't take a scientist of Tom Kilburn's cleverness to realize that you can find the largest factor of an integer *N* by counting downward from *N* – 1 until you find one that divides evenly into *N*.

The next few sections work through problems for which the solution should be at least as manageable. In each of these examples, you can easily work out a solution by hand. The computer may find the solution faster, but there aren't any operations or concepts that you don't already understand.

Finding the largest value in a list entered by the user

The first example of a problem you could easily solve on your own is that of finding the largest value in a sequence of numbers given to you, one at a time, by the user. A sample run of the program might look like this:



```
This program finds the largest value in a list of integers.
Enter a blank line to stop.
 ? 314
 ? 159
 ? 265
 ? 358
 ? 979
 ? 323
 ? 846
 ? 264
 ?
The largest value is 979
```

Before you try to write the program (or look at the solution in Figure 3-4 on the next page), imagine that you are trying to solve this problem just as the computer does. You need to ask the user for each number, check for the blank line marking the end of the input, and then print out the largest value entered. Much of the logic is therefore the same as that in the AddList.py program from Chapter 2. The only difference is that, instead of adding the numbers as you go, you need to find the largest.

What do you do when you get the first number, which is 314 in the sample run? You have to write it down somewhere because it might turn out to be the largest value since you haven't yet seen any of the others. But what happens when the user gives you the second number, which is 159? Do you need to write it down? The 159 can't be the largest number because it's smaller than the 314 you wrote down earlier. You can therefore ignore it entirely. You can ignore the third number (265) for exactly the same reason. When you get the 358 as the fourth number, however, you can't be so cavalier because 358 is greater than 314. You therefore need to remember the 358, although you can now forget about the 314.

The fundamental insight you need to design this algorithm is that you only have to keep track of the value that is the *largest value so far.* You are free to discard all of the other values. In the Python implementation, you can store this value in a variable called largest. If the current number is stored in the variable value, all you need to do as each new value comes in is execute the following code:

```
if value > largest:
    largest = value
```

In English, these lines tell Python to check whether the current value is greater than the largest value so far and, if so, record the current value as the new largest value.

There are still a few details to consider before writing the final program, one of which is how to initialize the value of largest. In the AddList.py program, it made since to initialize the value of total to 0 and then add each new input value to it. In this program, however, you can't simply set largest to 0 because the program would then fail if all the input values were negative (and no one said they couldn't be). What makes sense instead is to initialize largest to the special Python value None, which is included in the language to indicate a missing value. You need to check to see if largest is None when you check the current value and again when you print the final answer, but those changes are conceptually small.

Even small conceptual changes, however, can pose issues for programmers just starting out. For reasons that are beyond the scope of this text, Python's conventions dictate that programs should check for the value None using the is operator rather than the == operator. Although either would work in this case, it makes sense to start following the convention as early as possible.

FIGURE 3-4 **Program to find the largest integer**

```
# File: FindLargest.py

"""This program finds the largest integer in a list entered by the user."""

def find_largest():
    print("This program finds the largest value in a list of integers.")
    print("Enter a blank line to stop.")
    largest = None
    finished = False
    while not finished:
        line = input(" ? ")
        if line == "":
            finished = True
        else:
            value = int(line)
            if largest is None or value > largest:
                largest = value
    if largest is None:
        print("No values were entered")
    else:
        print(f"The largest value is {largest}")

# Startup code

if __name__ == "__main__":
    find_largest()
```

Finding all two-letter words

Given the popularity of word puzzles like Wordle and Spelling Bee in *The New York Times,* it is fun to look for algorithms that might be useful in playing word games. One example that illustrates how different algorithms exist for solving the same problem is making a list of the two-letter words that appear in a dictionary. This list is so important in Scrabble that most serious players take the time to memorize it.

Once again, the first step in solving this problem is to think about how you would approach it without the aid of a computer. Imagine that you have a physical dictionary and a notepad. How might you go about making the list of all two-letter words?

Although doing so will be time-consuming, you can adopt the brute-force strategy of opening the dictionary and going through all the words in order, making a list as you go of all the two-letter entries. In a dictionary of English words, the first entry is the word *a.* That word has only one-letter, so you ignore it and move on to the next. If the dictionary is reasonably complete, the second entry is the word *aa,* which is the Hawaiian name for a particular form of lava characterized by its rough texture. That word has two letters, so you write it down. The pseudocode version of this algorithm therefore looks like this:

```
for each word in the dictionary:
    if the length of that word is two:
        Write down the word.
```

To translate this pseudocode into an actual program, you need a dictionary that Python can read. The libraries associated with this text include an `english.py` module that defines a constant `ENGLISH_WORDS` whose value is an alphabetical list of the words in an at-least-reasonably-complete dictionary of English words. No magic is involved in creating this list. If you look at the contents of `english.py`, you will see a definition that begins with the lines

```
ENGLISH_WORDS = [
  "a", "aa", "aah", "aahed", "aahing", "aahs", "aal", "aalii",
```

and ends many thousands of lines later with the words

```
  "zymotic", "zymurgies", "zymurgy", "zyzzyva", "zyzzyvas"
]
```

A program that uses this definition to produce the two-letter word list appears in Figure 3-5.

The solution strategy in Figure 3-5, however, is not the only one you might choose, nor is it the most efficient one. Looking at every word in the dictionary takes time, particularly if you try to do so by hand. By contrast, looking up a collection of letters

FIGURE 3-5 **Program to list all two-letter English words (version 1)**

```
# File: TwoLetterWords.py (brute-force version)

"""This program prints a list of all two-letter words in English."""

from english import ENGLISH_WORDS

# Implementation notes: two_letter_words
# --------------------------------------
# This function prints a list of all two-letter words in English by going
# through the entire dictionary and checking the length of each word.

def two_letter_words():
    for word in ENGLISH_WORDS:
        if len(word) == 2:
            print(word)
```

to see whether it exists in the dictionary is comparatively fast because the alphabetical arrangement of the dictionary helps guide you to where the word—if indeed it is a word—must appear. This insight suggests the following pseudocode algorithm:

```
for every possible combination of two letters:
    if that combination exists in the dictionary:
        Write down that two-letter combination.
```

Implementing this strategy in Python requires you to use a second feature from the `english.py` module. In addition to the constant `ENGLISH_WORDS`, this library defines a predicate function `is_english_word(letters)` that checks whether the string `letters` appears in the list of English words. That check, moreover, is extremely efficient because it uses a fast search algorithm that you will learn about in Chapter 9. Figure 3-6 shows the Python implementation of this strategy, which runs almost three times faster than the brute-force version.

Finding anagrams

The `english.py` module allows you to solve other interesting problems that arise in word games, a few of which appear in the exercises. To emphasize how much you can accomplish with a small amount of code, this section implements a program that finds all the English words that contain a particular set of letters. For example, given the seven letters *a, e, i, m, n, r,* and *s,* you can form each of the following three words: *marines, remains,* and *seminar.* Words that contain the same letters in a different order are called ***anagrams.***

FIGURE 3-6 Program to list all two-letter English words (version 2)

```python
# File: TwoLetterWords.py (look only at two-letter combinations)

"""This program prints a list of all two-letter words in English."""

from english import is_english_word

# Implementation notes: two_letter_words
# --------------------------------------
# This function prints a list of all two-letter words in English by
# generating all two-letter sequences and checking which ones are words.

def two_letter_words():
    ALPHABET = "abcdefghijklmnopqrstuvwxyz"
    for ch1 in ALPHABET:
        for ch2 in ALPHABET:
            word = ch1 + ch2
            if is_english_word(word):
                print(word)
```

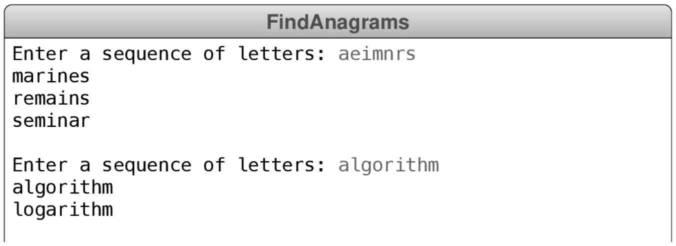
Having a program to find anagrams would be helpful in solving Wordle puzzles but even more valuable in Scrabble, where you get a 50-point bonus by playing all seven of your tiles in a single turn for what Scrabble players call a ***bingo.*** Although using such a program in an actual game would certainly be cheating, you could use it to help you learn various letter combinations that form seven-letter words.

Although a general technique for generating all rearrangements of a string requires concepts beyond the scope of this text, you can achieve the desired result by going through the list of English words and adding every word that contains a particular combination of letters to a list of anagrams. Moreover, all you need to do to test whether two combinations of letters are the same is to see if sorting those combinations produces the same list. As it happens, you already saw in Chapter 2 that you can use the built-in function sorted to transform a string into an alphabetized list of its characters. That insight is all you need to write the program to find all anagrams, which appears in Figure 3-7.

FIGURE 3-7 **Program to find all anagrams of a set of letters**

```
# File: FindAnagrams.py

"""This program prints all the anagrams for a sequence of letters."""

from english import ENGLISH_WORDS

def create_anagram_list(letters):
    """Returns a list of the anagrams of the specified set of letters."""
    anagrams = [ ]
    sorted_letters = sorted(letters)
    for word in ENGLISH_WORDS:
        if sorted_letters == sorted(word):
            anagrams += [ word ]
    return anagrams

def find_anagrams():
    finished = False
    while not finished:
        letters = input("Enter a sequence of letters: ")
        if letters == "":
            finished = True
        else:
            for word in create_anagram_list(letters):
                print(word)
            print()

# Startup code

if __name__ == "__main__":
    find_anagrams()
```

The program in Figure 3-7 defines both a `create_anagram_list` function that generates the list of anagrams and a main program called `find_anagrams` that makes it easy for the user to enter a set of letters and get back the list of anagrams. A sample run of this program might look like this:



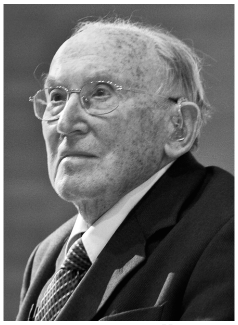
```
Enter a sequence of letters: aeimnrs
marines
remains
seminar

Enter a sequence of letters: algorithm
algorithm
logarithm
```

3.3 Testing and debugging

Although you may sometimes get lucky with extremely simple programs, one of the truths you'll soon have to accept as a programmer is that very few of your programs will run correctly the first time around. Most of the time, you will need to spend a considerable fraction of your time testing the program to see whether it works, discovering that it doesn't, and then settling into the process of ***debugging,*** in which you find and fix the errors in your code.

Perhaps the most compelling description of the centrality of debugging to the programming process comes from the British computing pioneer Maurice Wilkes (1913–2010), who in 1979 offered the following reflection from his early years in the field:

> As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.


Maurice Wilkes

Programming defensively

Even though it is impossible to avoid bugs altogether, you can reduce the number of bugs by being careful during the programming process. Just as it's important to drive defensively in your car, it makes sense to program defensively as you write your code. The most important aspect of defensive programming is looking over your programs to ensure that they do what you intend them to do. You will also find that taking the time to make your code as clear and readable as possible will help avoid problems down the road.

Becoming a good debugger

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. If you are just beginning your study of programming, it is likely that the frustrating aspects of debugging will loom much larger than the excitement of meeting an interesting intellectual challenge. That fact in itself is by no means surprising. Debugging, after all, is a skill that takes time to learn. Before you have developed the necessary experience and expertise, your forays into the world of debugging will often leave you facing a completely mysterious problem that you have no idea how to solve. And when your assignment is due the next day and you can make no progress until you somehow solve that mystery, frustration is probably the most natural reaction.

To a surprising extent, the challenges that people face while debugging are not so much technical as they are psychological. To become a successful debugger, the most important thing is to start thinking in new ways that get you beyond the psychological barriers that stand in your way. There is no magical, step-by-step approach to finding the problems, which are usually of your own making. What you need is logic, creativity, patience, and a considerable amount of practice.

The phases of the programming process

When you are developing a program, the actual process of writing the code is only one piece of a more complex intellectual activity. Before you sit down to write the code, it is always wise to spend some time thinking about the program design. As you will discover as you start to write more sophisticated applications, programs are often too large to write as a single function, which in turn forces you to decompose the problem into more manageable pieces. Putting some thought into the design of that decomposition before you start writing the individual functions is almost certain to reduce the total amount of time—and frustration—involved in the project as a whole. After you've written the code, you need to test whether it works and, in all probability, spend some time ferreting out the bugs that prevent the program from doing what you want.

These four activities—designing, coding, testing, and debugging—constitute the principal components of the programming process. And although there are certainly some constraints on order (you can't debug code that you haven't yet written, for example), it is a mistake to think of these phases as rigidly sequential. The biggest problem that students have comes from thinking that it makes sense to design and code the entire program and then try to get it working as a whole. Professional programmers never work that way. They develop a preliminary design, write some pieces of the code, test those pieces to see if they work as intended, and then fix the bugs that the testing uncovers. Only when that individual piece is working do professional programmers return to code, test, and debug the next section of the program. From time to time, they go back and revisit the design as they learn from the experience of seeing how well the original design works in practice. You must learn to work in much the same way.

It is equally important to recognize that each phase in the programming process requires a fundamentally different approach. As you move back and forth among the various phases, you need to adopt different ways of thinking. In my experience, the best way to illustrate how these approaches differ is to associate each phase with a profession that depends on much the same skills and modes of thought.

Phases and roles in the programming process

Design	=	Architect
Coding	=	Engineer
Testing	=	Vandal
Debugging	=	Detective

During the design phase, you have to think like an *architect.* You need to have a sense not only of the problem that must be solved but also an understanding of the underlying aesthetics of different solution strategies. Those aesthetic judgments are not entirely free from constraints. You know what's needed, you recognize what's possible, and you choose the best design that lies within those constraints.

When you move to the coding phase, your role shifts to that of the *engineer.* Now your job is to apply your understanding of programming to transform a theoretical design into an actual implementation. This phase is by no means mechanical and requires a significant amount of creativity, but your goal is to produce a program that you believe implements the design.

In many respects, the testing phase is the most difficult aspect of the process to understand. When you act as a tester, your role is not to establish that the program works, but just the opposite. Your job is to break it. A tester therefore needs to assume the role of a *vandal.* You need to search deliberately for anything that might go wrong and take real joy in finding any flaws. It is in this phase of the programming process that the most difficult psychological barriers arise. As the author of the program, you want it to work; as the tester, you want it to fail. Many people have trouble shifting focus in this way. After all, it's hard to be overjoyed at pointing out the stupid mistakes the programmer made when you also happen to be that programmer. Even so, you need to make this shift.

Finally, your job in the debugging phase is that of a *detective.* The testing process reveals the existence of errors but does not necessarily reveal why they occur. Your job during the debugging phase is to sort through all the available evidence, create a hypothesis about what is going wrong, check that hypothesis through additional testing, and then make the necessary corrections.

As with testing, the debugging phase is full of psychological pitfalls. When you were writing the code in your role as engineer, you believed that it worked correctly when you designed it in your role as architect. You now have to discover why it doesn't, which means that you have to discard any preconceptions you've retained from those earlier phases and approach the problem with a fresh perspective. Making that shift successfully is always a difficult challenge. Code that looked correct to you once is likely to look just as good when you come back to it a second time.

What you need to keep in mind is that the testing phase has determined that the program is not working correctly. There must be a problem somewhere. It's not the browser or Python that's misbehaving or some unfortunate conjunction of the planets. As Cassius reminds Brutus in Shakespeare's *Julius Caesar,* "the fault, dear Brutus, is not in our stars, but in ourselves." You introduced the error when you wrote the code, and it is your job to find it.

This book will offer additional suggestions about debugging as you learn how to write more complex programs, but the following principle will serve you better than any specific debugging strategy or technique:

> *When you are trying to find a bug, it is more important to understand what your program is doing than to understand what it isn't doing.*

Most students who come upon a problem in their code go back to the original problem and try to figure out why their program isn't doing what they wanted. Although such an approach can be helpful in some cases, it is far more likely that this kind of thinking will make you blind to the real problem. If you make an unwarranted assumption the first time around, you are likely to make it again, and therefore find it difficult to see any reason why your program isn't doing the right thing. You need instead to gather information about what your program is doing and then work out where it goes wrong.

Although many modern Python programming environments come equipped with sophisticated debuggers, you are likely to get the most mileage out of the built-in function `print`. If you discover that your program isn't working, you can add a few calls to `print` at places where you think your program might be going down the wrong path. In some cases, it's sufficient to include a line like

```
print("I got here")
```

to the program. If the message `"I got here"` appears on the console, you know that the program got to that point in the code. It is often even more helpful to use `print` to display the value of an important variable. If, for example, you expect the variable `n` to have the value 100 at some point in the code, you can add the line

```
print(f"n = {n}")
```

If running the program shows that `n` has the value 0 instead, you know that something has gone wrong prior to this point. Narrowing down the region of the program in which the problem might be located puts you in a much better position to find and correct the error.

FIGURE 3-8 **Debugging advice from detective fiction**

Regard with distrust all circumstances which seem to favor our secret desires.
—Émile Gaboriau, *Monsieur Lecoq,* 1868

There is nothing like first-hand evidence.
—Sir Arthur Conan Doyle, *A Study in Scarlet,* 1888

It is a capital mistake to theorise before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.
—Sir Arthur Conan Doyle, *A Scandal in Bohemia,* 1892

It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which vital. Otherwise your energy and attention must be dissipated instead of being concentrated.
—Sir Arthur Conan Doyle, *The Adventure of the Reigate Squires,* 1892

With method and logic one can accomplish anything.
—Agatha Christie, *Poirot Investigates,* 1924

Detection requires a patient persistence which amounts to obstinacy.
—P. D. James, *An Unsuitable Job for a Woman,* 1972

It was always more difficult than you thought it would be.
—Alexander McCall Smith, *The No. 1 Ladies' Detective Agency,* 1998

Since the process of debugging is similar to the art of detection, it seems appropriate to offer some of the more relevant bits of debugging wisdom I've encountered in detective fiction, which appear in Figure 3-8. I also strongly recommend Robert Pirsig's critically acclaimed novel *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values* (Bantam, 1974), which stands as the best exposition of the art and psychology of debugging ever written. The most relevant section is the discussion of "gumption traps" in Chapter 26.

An example of a psychological barrier

Although most testing and debugging challenges involve a level of programming sophistication beyond the scope of this chapter, there is a very simple program that illustrates just how easy it is to let your assumptions blind you not only to the cause of an error but even to its very existence. Throughout the many years I've taught computer science, one of my favorite problems to assign at the beginning of the term is to write a function that solves the quadratic equation

$$ax^2 + bx + c = 0$$

This equation has two solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

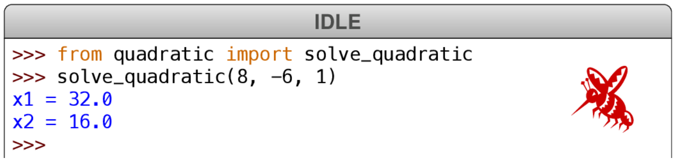
The first solution is obtained by using + in place of the ± symbol; the second is obtained by using – instead. The problem I give students is to write a function that takes *a, b,* and *c* as parameters and displays the two resulting solutions for *x.*

Although the majority can solve this problem correctly, there are always a number of students—as much as 20 percent of a large class—who turn in functions that look something like this:

```python
def solve_quadratic(a, b, c):
    r = math.sqrt(b*b - 4*a*c)
    x1 = (-b + r) / 2*a
    x2 = (-b - r) / 2*a
    print(f"x1 = {x1}")
    print(f"x2 = {x2}")
```

This text uses a red bug to mark code that is incorrect. This implementation of `solve_quadratic` is buggy, although the problem is subtle. It *looks* as if the expression `2*a` is in the denominator of the fraction, when in fact it isn't. In Python, operators in the same precedence class, such as the `/` and `*` in the lines defining `x1` and `x2`, are evaluated in left-to-right order. The parenthesized value in these expressions is therefore first divided by 2 and then multiplied by `a`. Python requires parentheses around the denominator `(2*a)`.

The real lesson in this example, however, lies in the fact that many students compound their mistake by failing to discover it. Most of the students who make this error fail to test their programs for any values of the coefficient *a* other than 1, since those are the easiest answers to compute by hand. If *a* is 1, it doesn't matter whether you multiply or divide by *a* because the answer will be the same. Worse still, students who test their program for other values of *a* often fail to notice that their programs give incorrect answers. I often get sample runs that look like this:



```
>>> from quadratic import solve_quadratic
>>> solve_quadratic(8, -6, 1)
x1 = 32.0
x2 = 16.0
>>>
```

This sample run asserts that $x = 32$ and $x= 16$ are solutions to the equation

$$8x^2 - 6x + 1 = 0$$

but it is easy to check that neither of these values satisfy the equation. Even so, students happily submit such programs without noticing that the answers are wrong.

Writing test programs

Whenever you write a function, it is a good idea to write a function to check that your implementation works for a reasonably large set of cases. In doing so, it is often useful to make use of Python's `assert` statement, which you have already seen in Chapter 2. The code in Figure 3-9 defines three functions: a main program called `quadratic` that lets the user enter the coefficients and see the results, a function called `find_quadratic_roots` that other code could use to determine the solve the quadratic equation, and a function called `find_quadratic` that uses `assert` statements to verify that the results are correct.

The fact that the `quadratic.py` module includes both a main program and a function that other programmers might want to use as a library makes it a little

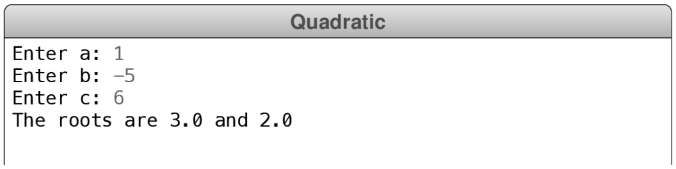
FIGURE 3-10 Implementation of the `quadratic` module, including a test function

```
# File: quadratic.py

import math

"""
This file solves the quadratic equation by asking the user to enter
values for the coefficients a, b, and c.
"""

def quadratic():
    """Asks the user for a, b, and c, and then prints the roots."""
    a = float(input("Enter a: "))
    b = float(input("Enter b: "))
    c = float(input("Enter c: "))
    roots = find_quadratic_roots(a, b, c)
    print(f"The roots are {roots[0]} and {roots[1]}")

def find_quadratic_roots(a, b, c):
    """Returns a list the roots of the quadratic equation."""
    r = math.sqrt(b**2 - 4*a*c)
    x1 = (-b + r) / (2 * a)
    x2 = (-b - r) / (2 * a)
    return [ x1, x2 ]

def test_quadratic():
    assert find_quadratic_roots(1, -5, 6) == [3.0, 2.0]
    assert find_quadratic_roots(1, -4, 4) == [2.0, 2.0]
    assert find_quadratic_roots(2, 0, -8) == [2.0, -2.0]
    assert find_quadratic_roots(8, -6, 1) == [0.5, 0.25]

if __name__ == "__main__":
    quadratic()
```

difficult to know exactly what the standard startup code should do. In this example, the boilerplate consists of the lines

```
if __name__ == "__main__":
    quadratic()
```

Thus, if you invoke the quadratic.py module from the command line or a Python development environment, it runs the main program for the user, which might produce the following sample run:



```
Enter a: 1
Enter b: -5
Enter c: 6
The roots are 3.0 and 2.0
```

If you want to run the test program instead, you need to invoke it explicitly from IDLE or whatever Python development environment you're using. A better strategy, however, is to install the pytest package, which implements an automated testing environment. If you run it from the command line using

```
pytest quadratic.py
```

the pytest application will search the quadratic.py module for any functions whose names begin with test_ to see if any of the assertions fail.

The decision to combine a main program and a test function in the same has another implication for the module design. It is difficult to write test functions for a main program that communicates directly with the user by making calls to input and print. What you need to instead is separate the functions—as the code in Figure 3-9 does—so that different functions take care of the user interaction and the underlying calculation. The test function can then make assertions about the calculations without requiring the user to take any explicit action.

Software maintenance

One of the more surprising aspects of software development is that programs require maintenance. In fact, studies of software development indicate that, for commercial applications, paying programmers to maintain the software after it has been released constitutes between 80 and 90 percent of the total cost. In the context of software, however, it is a little hard to imagine precisely what maintenance means. At first hearing, the idea sounds rather bizarre. If you think in terms of a car or a bridge, maintenance occurs when something has broken—some of the metal has rusted away, a piece of some mechanical linkage has worn out from overuse, or something has gotten smashed up in an accident. None of these situations apply to software. The code itself doesn't rust. Using the same program over and over again does not in any way diminish its functioning. Accidental misuse can certainly have dangerous consequences but does not usually damage the program itself; even if it does, the program can often be restored from a backup copy. What does maintenance mean in such an environment?

Software requires maintenance for two principal reasons. First, even after considerable testing and, in some cases, years of field use, bugs can still survive in the code. When some unanticipated situation arises, the bug, previously dormant, causes the program to fail. Thus, debugging is an essential part of program maintenance. It is not, however, the most important part. Far more consequential, especially in terms of the impact on the overall cost of program maintenance, is that programs need to change in response to changing requirements. Users often want new features in their applications, and software developers try to provide those features to maintain customer loyalty. In either case—whether one wants to repair a bug or add a feature—someone has to look at the program, figure out what's going on, make the necessary changes, verify that those changes work, and then release a new version. This process is difficult, time-consuming, expensive, and prone to error.

Program maintenance is especially difficult because many programmers do not write their programs for the long haul. To them it seems sufficient to get the program working and then move on to something else. The discipline of writing programs so that they can be understood and maintained by others is called ***software engineering.*** In this text, you are encouraged to write programs that demonstrate effective software engineering techniques.

Many novice programmers are disturbed to learn that there is no precise set of rules you can follow to ensure good programming style. Software engineering is not a cookbook sort of process. It is instead a skill blended with more than a little bit of artistry. Practice is critical. One learns to write well-structured programs by writing them, and by reading others, much as one learns to be a novelist. Becoming an effective programmer requires discipline—the discipline not to cut corners or to forget, in the rush to complete a project, about that future maintainer. Good programming practice also requires developing an aesthetic sense of what it means for a program to be readable and well presented.

Although there are no hard-and-fast rules for writing maintainable programs, there are certainly some important principles, including the following:

- Write both your code and your comments with future maintainers in mind.
- Choose names for variables, constants, and functions that convey their purpose.
- Use indentation to highlight the hierarchical structure of your programs.
- Design your programs so that they are easy to modify as requirements change.

The last point in this list deserves additional discussion. Given that programs will inevitably change over their lifetimes, it is good programming practice to help future maintainers make the necessary changes. A useful strategy to support ongoing maintenance is to use constant definitions for values that you expect might change at some point down the road.

The value of using constant definitions is perhaps easiest to illustrate in the context of a historical example. Imagine for the moment that you are a programmer in the late 1960s working on the initial design of the ***ARPANET,*** which is the forerunner of today's internet. Because resources were highly constrained at that time, the designers of the ARPANET placed a limit on the number of computers (which were called ***hosts*** in the ARPANET days) that could be connected to the network. In the early years of the ARPANET, that limit was 127 hosts. If Python had existed in 1969, you might have declared a constant like this:

```
MAXIMUM_NUMBER_OF_HOSTS = 127
```

At some later point, however, the explosive growth of networking would force you to raise this bound.

Making that change would be easy if you had defined a constant but hard if you had instead written the number 127. In that case, you would need to change all instances of 127 that refer to the number of hosts. Some instances of 127 might refer to things other than the limit on the number of hosts, and it would be important not to change any of those values. In the likely event that you had made a mistake in that process, you would have a very hard time tracking down the bug.

Summary

The focus of this chapter is on the concept of an algorithm and on how to approach designing, implementing, testing, and debugging algorithms of your own. The important points include:

- An algorithm is a strategy that is *clear and unambiguous, effective,* and *finite*.
- Algorithms have existed since long before computers. Important historical examples include the *Babylonian method* for approximating square roots from between the 19th and 16th centuries BCE and *Euclid's algorithm* for finding the greatest common divisors of two integers, which dates from around 300 BCE.
- There are usually several different algorithms for solving a particular problem. Algorithms for solving a problem often vary dramatically in their efficiency. Choosing the algorithm that best fits the application is an important part of your task as a programmer.

- Although they are rarely the most efficient, it is often easiest to find a *brute-force algorithm* that tries every possible solution looking for an answer. An early example from the dawn of the computing age is the first program run on the Small-Scale Experimental Machine at Manchester University in 1948, which calculated the largest factor of an integer *N* by trying each successively smaller number until it found one that divided evenly into *N.*
- When you are given a programming problem, it is often useful to think about how you would solve it without a computer. If you don't understand the solution strategy, you will be unable to write a program that carries out the necessary steps.
- Another useful programming strategy is figuring out what information you need to remember and what information you can safely forget. In the program shown in Figure 3-4 that finds the largest number in a list of values entered by the user, you don't need to remember *all* the values the user gives you; all you need to remember is the *largest value so far.*
- Python includes a keyword called `None` that you can use to indicate the absence of an actual value. By convention, Python programs test for this special value using the `is` operator, as in the following statement from the `FindLargest.py` program in Figure 3-4:

  ```
  if largest is None:
  ```

- The libraries associated with this text include a module called `english.py` that exports a constant `ENGLISH_WORDS` containing an alphabetical list of English words and a predicate function `is_english_word` that checks whether its argument is a valid English word.
- The four phases of the programming process are *design, coding, testing,* and *debugging,* although it is best to view these phases as interrelated rather than sequential. Professional programmers typically code one piece of a program, test it, debug it, and then go back and work on the next piece.
- Each phase in the programming process requires you to behave in a different way. During the design phase, you act as an *architect.* When you are coding, you function as an *engineer.* During testing, you must act like a *vandal,* striving to break the program, not to prove that it works. When debugging, you need to think like a *detective* employing all the cleverness and insight of a Sherlock Holmes.
- When you are trying to find a bug, it is more important to understand what your program *is* doing than to understand what it *isn't* doing.
- In seeking to understand what your program is doing, your most helpful resource is the built-in `print` function.
- The most serious problems programmers face during the testing and debugging phases are psychological rather than technical. It is extremely easy to let your assumptions and desires get in the way of understanding where the problems lie.

- It is good programming practice to include test programs along with the definitions of any functions that you write.
- Programs require maintenance over their life cycles both to correct bugs and to add new features as user requirements change.

Review questions

1. What are the two words that Muhammad ibn Mūsā al-Khwārizmī and his work gave to English?

2. What conditions must a solution strategy meet in order to be an algorithm?

3. How would you define a *brute-force strategy?*

4. Use Euclid's algorithm to compute the greatest common divisor of 7735 and 4185. What values does the local variable `r` take on during the calculation?

5. In the examples that use Euclid's algorithm to calculate `gcd(x, y)`, the value of `x` has always been larger than `y`. What happens if `x` is smaller than `y`?

6. What was the nickname of the Small-Scale Experimental Machine developed at Manchester University that was in many respects the first modern digital computer?

7. True or false: A good way to approach many programming problems is to figure out how you would solve it yourself without using a computer.

8. How would you write a Python expression that tests whether the value of the variable `x` is equal to the special constant `None`?

9. The programs in this chapter use two definitions exported by the `english.py` module. What are those two definitions?

10. The two versions of the `TwoLetterWords.py` program both generate a list of the valid two-letter words in English. Will those versions always generate those words in the same order? Why or why not?

11. Explain the purpose of the calls to the function `sorted` in the `FindAnagrams.py` module shown in Figure 3-7.

12. What are the four phases of the programming process identified in this chapter? For each of those phases, what professional role does the chapter offer as a model for how to perform that phase?

13. True or false: Professional programmers work through the four phases of the programming process in order, finishing each one before moving on to the next.

14. True or false: When you are testing your program, your primary goal is to show that it works.

15. What piece of advice does the chapter offer to help you think effectively about debugging?

16. What built-in function does the text identify as the most useful debugging tool?

17. In your own words, explain what is meant by *program maintenance.*

Exercises

1. Modify the Babylonian algorithm as presented in the text so that it calculates cube roots instead of square roots. Express the algorithm in the form of a function `cuberoot(`*n*`)` that returns the cube root of the argument *n.* The creative part of this problem is figuring out what numbers you should average to obtain a new guess on each cycle of the loop. If *g,* for example, lies to one side of the cube root of *n,* what value can you compute using *n* and *g* that would be approximately as close to the root but on the opposite side? If you can find such a value, averaging the two will yield a result that is closer to the actual answer.

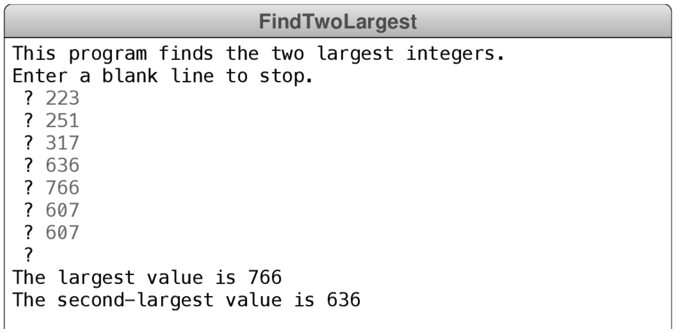
2. As noted in the description of the program for the Manchester Baby to find the largest factor of an integer, the solution that appears in the text takes "some advantage of Python's extended set of operations." The Manchester Baby could not perform multiplication and division and therefore had no immediate way to calculate a remainder. Rewrite the `largest_factor` function so that it uses only assignment, addition, subtraction, and comparing a number against 0.

3. One of the important strategic principles in Scrabble is to conserve your **S** tiles, because the rules for English plurals mean that many words take an **S**-hook at the end. Some words, of course, allow an **S** tile to be added at the beginning, but it turns out that there are 680 words—including, for example, the words *cold* and *hot*—that allow an **S**-hook on both ends. Write a program that uses the `english.py` module to display a list of all such words.

4. The `FindAnagrams.py` program checks to see if two words are anagrams by sorting the letters of each word and seeing whether those sorted lists match. Some English words already have their letters arranged in sorted order, such as *is, aft, cost, below,* and *almost.* Write a program that display a list of all words defined in the `english.py` module that have this property.

5. In many cases, as in the case of the two strategies presented for calculating the greatest common divisor, using a better algorithm can result in an enormous increases in efficiency. Even if you can't find improvements at that level, it is still useful to look for modifications that produce more modest performance gains. As presented in the text, the `FindAnagrams.py` program ends up sorting the letters in every dictionary word, but there is no point in doing so unless the word has the correct length. Rewrite `FindAnagrams.py` so that it checks that the length of the dictionary word matches that of the letter sequence before you call `sorted`.

6. Write a Python program that reads in integers up to a blank line and then prints both the largest and second-largest values in the user's input, as follows:



```
This program finds the two largest integers.
Enter a blank line to stop.
 ? 223
 ? 251
 ? 317
 ? 636
 ? 766
 ? 607
 ? 607
 ?
The largest value is 766
The second-largest value is 636
```

The values in this sample run are the number of pages in the British hardcover editions of J. K. Rowling's *Harry Potter* series. The output tells us that the longest book is the *Harry Potter and the Order of the Phoenix* at 766 pages and the second-longest book is *Harry Potter and the Goblet of Fire* at 636 pages.

7. The German mathematician Gottfried Wilhelm von Leibniz discovered the rather remarkable fact that the mathematical constant π can be computed using the following mathematical relationship:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

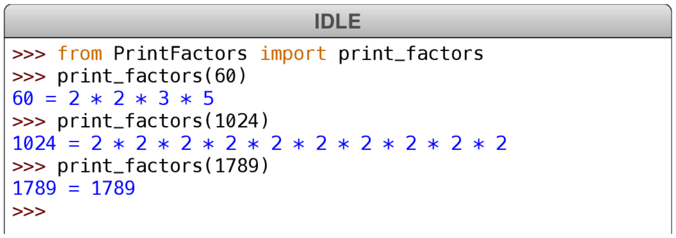
The formula to the right of the equal sign represents an infinite series; each fraction represents a term in that series. If you start with 1, subtract one-third, add one-fifth, and so on, for each of the odd integers, you get a number that gets closer and closer to the value of $\pi/4$ as you go along.

Write a program that calculates an approximation of π consisting of the first 10,000 terms in Leibniz's series.

8. An integer greater than 1 is said to be ***prime*** if it has no divisors other than itself and one. The number 17, for example, is prime because it has no factors other

than 1 and 17. The number 91, however, is not prime because it is divisible by 7 and 13. Write a predicate function `is_prime(n)` that returns `True` if the integer `n` is prime, and `False` otherwise. As an initial strategy, implement `is_prime` using a brute-force algorithm that simply tests every possible divisor. Once you have that version working, look for improvements that increase your algorithm's efficiency without affecting its correctness.

9. The first program written for the Manchester Baby found the largest factor of a number. A more interesting problem is to find the complete set of factors. Write a function `print_factors(n)` that lists all the factors in the form of a single line that includes the number `n`, an equal sign, and the individual factors separated by asterisks, as illustrated in the following IDLE transcript:



```
>>> from PrintFactors import print_factors
>>> print_factors(60)
60 = 2 * 2 * 3 * 5
>>> print_factors(1024)
1024 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
>>> print_factors(1789)
1789 = 1789
>>>
```

10. Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of *n* is any divisor less than *n* itself). They called such numbers ***perfect numbers.*** For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

    Write a predicate function `is_perfect(n)` that returns `True` if the integer `n` is perfect, and `False` otherwise. Test your implementation by writing a program that uses the `is_perfect` function to check for perfect numbers in the range 1 to 9999 by testing each number in turn. Whenever your program identifies a perfect number, it should display that number on the screen. The first two lines of output should be 6 and 28. Your program should find two other perfect numbers in that range as well.

11. Although Euclid's algorithm for calculating the greatest common divisor is one of the oldest to be dignified with that term, there are other algorithms that date back many centuries. In the Middle Ages, one of the problems that required sophisticated algorithmic thinking was determining the date of Easter, which falls on the first Sunday after the first full moon following the vernal equinox. Given this definition, the calculation involves interacting cycles of the day of the week, the orbit of the moon, and the passage of the sun through the zodiac. Early algorithms for solving this problem date back to the third century and are

FIGURE 3-11 Gauss's algorithm for computing the date of Easter

I. Divide the number of the year for which one wishes to calculate Easter by 19, by 4, and by 7, and call the remainders of these divisions a, b, and c, respectively. If the division is even, set the remainder to 0; the quotients are not taken into account. Precisely the same is true of the following divisions.

II. Divide the value $19a + 23$ by 30 and call the remainder d.

III. Finally, divide $2b + 4c + 6d + 3$, or $2b + 4c + 6d + 4$, choosing the former for years between 1700 and 1799 and the latter for years between 1800 and 1899, by 7 and call the remainder e.

Then Easter falls on March $22 + d + e$, or when $d + e$ is greater than 9, on April $d + e - 9$.

Translated from Karl Friedrich Gauss, "Berechnung des Osterfestes," August 1800
http://gdz.sub.uni-goettingen.de/no_cache/dms/load/img/?IDDOC=137484

described in the writings of the eighth-century scholar known as the Venerable Bede. In 1800, the German mathematician Carl Friedrich Gauss published an algorithm for determining the date of Easter that was purely computational in the sense that it relied on arithmetic rather than looking up values in tables. His algorithm—translated from the German—appears in Figure 3-11.

Write a Python function `find_easter_date(year)` that returns a string showing the date of Easter in the specified year. For example, calling `find_easter_date(1800)` returns the string `"April 13"` because that is the date of Easter in the year that Gauss published his algorithm.

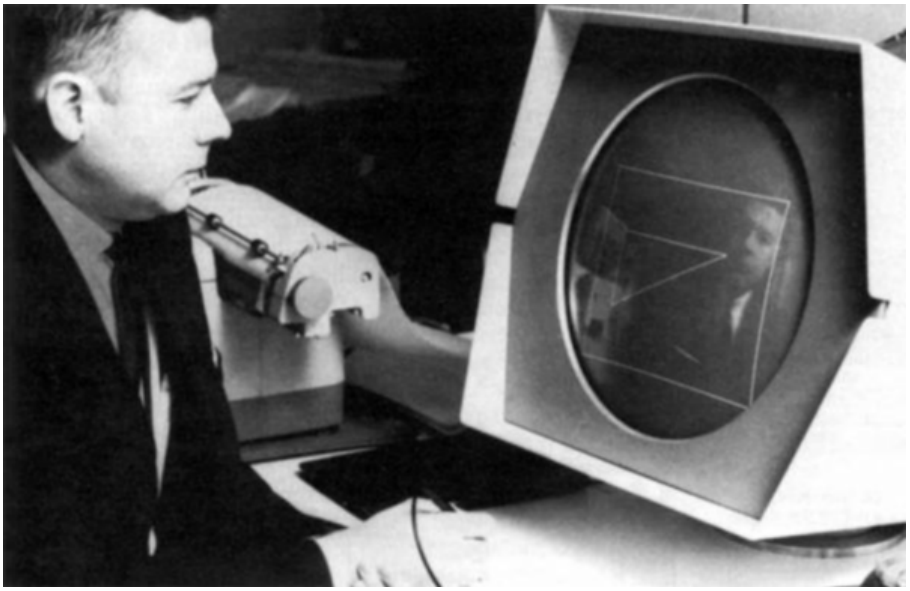
Unfortunately, the algorithm in Figure 3-11 works only for years in the 18th and 19th centuries. It is easy, however, to search the web for extensions that work for all years. Once you have completed your implementation of Gauss's algorithm, undertake the necessary research to implement a more general approach.

12. Working from the perspective of a designer, come up with an algorithm for tying your shoelaces and then write it down in English as carefully as you can. Once you have done so, shift your role to that of a tester and see if you can find any parts of the algorithm that are sufficient unclear or ambiguous that you can make the process fail even while obeying each of the instructions to the letter, at least under some interpretation. Finally, take on the role of debugger to fix the bugs you found during the testing phase.

CHAPTER 4

Simple Graphics

> A display connected to a digital computer gives us a chance to gain familiarity with concepts not realizable in the physical world. It is a looking glass into a mathematical wonderland.
>
> — Ivan Sutherland, "The Ultimate Display," 1965



Ivan Sutherland (1938–)

Ivan Sutherland was born in Nebraska and developed a passion for computers while still in high school, when a family friend gave him the opportunity to program a tiny relay-based machine called SIMON. Since computer science was not yet an academic discipline, Sutherland majored in electrical engineering at Pittsburgh's Carnegie Institute of Technology (now Carnegie Mellon University) and then went on to get a Master's degree at Caltech and a Ph.D. from MIT. His doctoral thesis, "Sketchpad: A man-machine graphical communications system," became one of the cornerstones of computer graphics and introduced the idea of the graphical user interface, which has become an essential feature of modern software. After completing his degree, Sutherland held faculty positions at Harvard, the University of Utah, and Caltech before leaving academia to found a computer-graphics company. Sutherland received the ACM Turing Award in 1988.

Although it is possible to learn the fundamentals of programming using only the numeric and string types you saw in Chapter 1, numbers and strings are not as exciting as they were in the early years of computing. For students who have grown up in the 21st century, much of the excitement surrounding computers comes from their ability to work with other more interesting types of data, including images and interactive graphical objects. Python is ideal for working with graphical data. Introducing just a few graphical types will enable you to create applications that are much more engaging and give you a greater incentive to master the material.

This chapter introduces you to the facilities in the ***Portable Graphics Library,*** a collection of tools for writing simple graphical applications. The discussion in this chapter provides enough information to get you started; more advanced features of the graphics library will be introduced as they are needed.

4.1 Your first graphics program

As is usually the case when you are studying programming, the best way to learn how graphical programs work is to look at an example. Although many examples might serve, the cultural history of computer science suggests using a particular programming problem as a starting point. That programming problem first appeared in *The C Programming Language* by Brian Kernighan and Dennis Ritchie, who offer the following advice on the first page of Chapter 1:

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
>
> *Print the words*
> `hello, world`
>
> This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where the output went. With these mechanical details mastered, everything else is comparatively easy.

That advice was followed by the four-line text of the “Hello World” program, which became part of the heritage shared by all C programmers.

Although the existence of a sophisticated interactive environment like IDLE makes it unnecessary—and entirely too easy—to use “Hello World” as your first Python program, it makes sense to use that problem as a starting point for graphical programs. The code for a graphically oriented `GraphicsHelloWorld.py` program appears in Figure 4-1. The new goal is not to print the words “hello, world” but instead to display those words in a graphics window.

FIGURE 4-1 **A graphical version of the “Hello World” program**

```
# File: GraphicsHelloWorld.py

"""
This program displays the string "hello, world" on the graphics window.
The inspiration for this program comes from Brian Kernighan and Dennis
Ritchie's book, The C Programming Language.
"""

from pgl import GWindow, GLabel

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 200

# Main program

def hello_world():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    msg = GLabel("hello, world", 50, 100)
    gw.add(msg)

# Startup code

if __name__ == "__main__":
    hello_world()
```

The main function for the `GraphicsHelloWorld.py` program looks like this:

```
def hello_world():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    msg = GLabel("hello, world", 50, 100)
    gw.add(msg)
```

The first two statements in this function define the variable `gw`, which stands for “graphics window,” and the variable `msg`, which refers to the message on the screen. The last statement then adds the message to the graphics window.

At one level, these statements are similar to the ones you have seen in the earlier chapters. Each of the two assignment statements introduces a new variable and then initializes it to a value produced by calling a function. The important difference lies in the types of those values.

4.2 Classes, objects, and methods

One of the most important things to notice about the `GraphicsHelloWorld.py` program in Figure 4-1 is that the values stored in the variables `gw` and `msg` are more complex than the values you've worked with so far, even though the underlying principles are the same. So far, the values you have stored in variables have been numbers and strings. In the `GraphicsHelloWorld.py` program, the value stored in each of the variables is an ***object,*** which is the term computer science uses to refer to a conceptually integrated entity that ties together the information that defines the state of the object and the operations that affect that state.

Each of these objects is a representative of a ***class,*** which is easiest to imagine as a template that defines the attributes and operations shared by all objects of a particular type. A single class can give rise to many different objects; each such object is said to be an ***instance*** of that class.

Creating objects

The `GraphicsHelloWorld.py` program uses the following assignment statements to create two objects:

```
gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
msg = GLabel("hello, world", 50, 100)
```

The names `GWindow` and `GLabel` are part of the `pgl` module, which implements the Portable Graphics Library. The first line after the introductory comments is

```
from pgl import GWindow, GLabel
```

which imports these two classes. The `GWindow` class represents a graphical window on the screen, and the `GLabel` class represents a string that can appear in that window. Functions that create new objects are called ***constructors*** and are written using camel case, starting with an uppercase letter.

The assignment statement

```
gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

uses the `GWindow` constructor to create an object whose class is `GWindow`. The parameters `GWINDOW_WIDTH` and `GWINDOW_HEIGHT` specify the window size in units called ***pixels,*** which are the tiny dots that cover the face of the display. The call to `GWindow` therefore creates a new `GWindow` object that is 500 pixels wide and 200 pixels high. That object is then assigned to a variable named `gw`, which makes it possible for the program to refer to the window in the rest of the code.

Even though the declarations of the variables `gw` and `msg` create the necessary objects, these lines alone do not cause the `GLabel` to appear in the `GWindow`. To get the message to appear, the program has to tell the `GWindow` object stored in `gw` to add the `GLabel` stored in `msg` to its internal list of graphical objects to display on the window. This step in the process is the responsibility of the last line in the `GraphicsHelloWorld.py` program, which looks like this:

```
gw.add(msg)
```

Understanding how this statement works requires you to learn a little more about the way that Python implements objects.

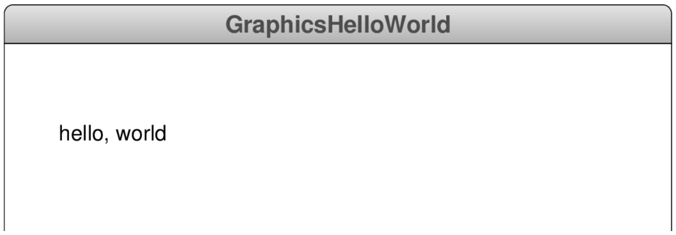
Sending messages to objects

When you are programming in a language that supports objects, it is useful to adopt at least some of the ideas and terminology of the ***object-oriented paradigm,*** a conceptual model of programming that focuses on objects and their interactions rather than on the more traditional model in which data and operations are seen as separate. In object-oriented programming, the generic term for anything that triggers a particular behavior in an object is called a ***message.*** In Python, sending a message to an object is implemented by calling a function associated with that object. Functions that are associated with an object are called ***methods,*** and the object on which the method is invoked is called the ***receiver.***

In Python, method calls use the following syntax:

receiver`.`*name*`(`*arguments*`)`

In the method call `gw.add(msg)`, the graphics window stored in `gw` is the receiver, and `add` is the name of the method that responds to the message. The argument `msg` lets the implementation of the `GWindow` class know what graphical object to add, which in this case is the `GLabel` stored in the `msg` variable. The `GWindow` responds by displaying the message at the specified coordinates on the screen, which creates the following image:



hello, world

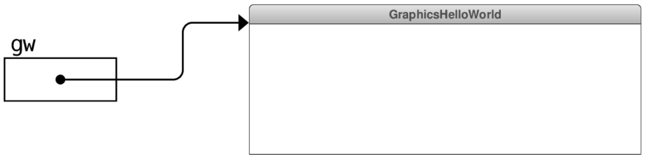
As you can see from the screen image, the desired message is there. The message is not very large or exciting, but you'll have a chance to spice it up later in the chapter.

References

In Python, the value stored in a variable like `gw` is not the entire object but instead a ***reference,*** which is a value internal to the computer that serves as a link to the data in the actual object. In the `GraphicsHelloWorld.py` program, the declaration

```
gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

initializes the variable gw to contain a reference to an on-screen window capable of displaying graphical objects, as illustrated by the following diagram:

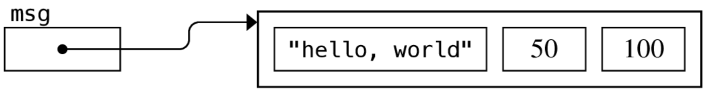


As the arrow suggests, the reference stored in gw points to a larger value that represents the graphics window on the screen.

The assignment statement

```
msg = GLabel("hello, world", 50, 100)
```

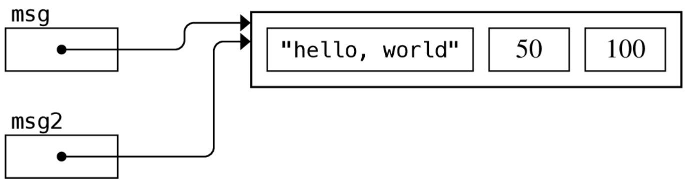
operates in a similar fashion. This line creates a GLabel object and assigns a reference to that object to the variable msg, as follows:



Although it is often possible to ignore the distinction between a reference and its associated object, it is important to understand that assigning an object value to a variable does not copy the entire object but instead copies only the reference. For example, if you were to write the statement

```
msg2 = msg
```

Python would not create a second object but would instead arrange it so that msg and msg2 both contained references to the *same* object, as follows:



If you need to create a second GLabel—even one that has the same contents—you need to call the GLabel constructor.

Understanding how Python uses references as links to larger data structures will be particularly important when you learn about arrays and objects in Chapters 8 through 12.

Encapsulation

The diagrams for the GLabel objects in the preceding section show only the data values that are stored inside those objects. In addition to these values, objects also contain the private data and associated code necessary to implement the class. That information, however, is not available to the main program but is instead securely packaged inside the GLabel object. This model of packaging data and code together is called ***encapsulation.***

4.3 Graphical objects

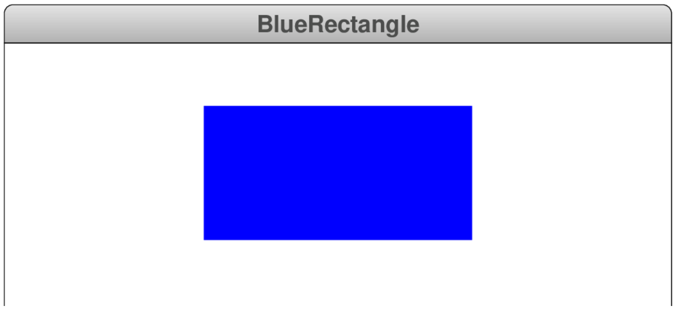
The GLabel class is only one of several classes in the Portable Graphics Library. This section introduces three other classes—GRect, GOval, and GLine—that, together with GLabel and GWindow, provide a useful "starter kit" for writing graphical programs.

The GRect class

The GRect class allows you to create rectangles and add them to the graphics window. For example, the program in Figure 4-2 creates a graphics window and then adds a rectangle to the window, solidly filled using the color blue, like this:

FIGURE 4-2 **Program to draw a blue rectangle on the graphics window**

```
# File: BlueRectangle.py

"""
This program draws a blue rectangle on the screen.
"""

from pgl import GWindow, GRect

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 200

# Main program

def blue_rectangle():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    rect = GRect(150, 50, 200, 100)
    rect.set_color("Blue")
    rect.set_filled(True)
    gw.add(rect)

# Startup code

if __name__ == "__main__":
    blue_rectangle()
```



The BlueRectangle.py program is similar to the GraphicsHelloWorld.py program from Figure 4-1. The blue_rectangle function begins—as the main functions for all graphics programs in this book do—by creating a GWindow of the desired size and assigning it to the variable gw.

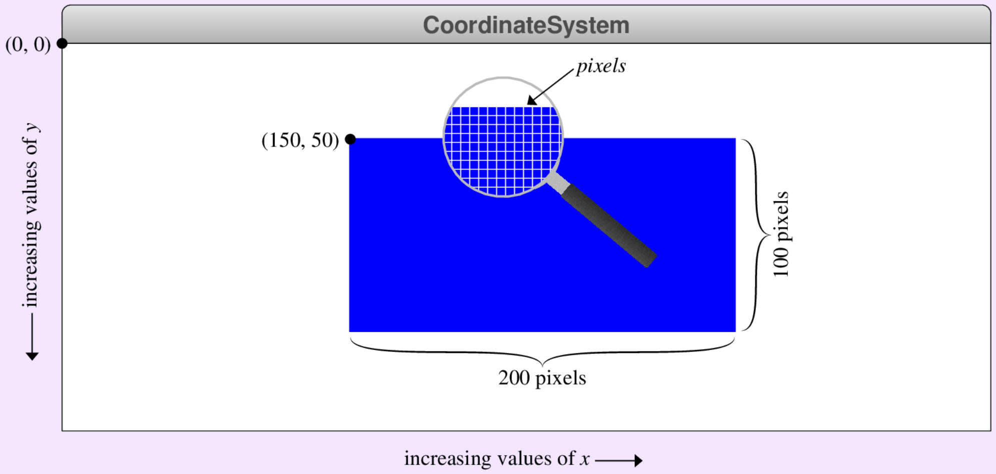
The next statement in the blue_rectangle function is

```
    rect = GRect(150, 50, 200, 100)
```

which creates a GRect object used to display the rectangle in the window. In this call, the first two arguments, 150 and 50, indicate the *x* and *y* coordinates at which the rectangle should be positioned; the second two arguments, 200 and 100, specify the width and height of the rectangle. As in the earlier call to GWindow, each of these values is measured in pixels. This geometry is illustrated in Figure 4-3.

When you work with the graphics library, it is important to keep in mind that the coordinate values in the *y* direction increase as you move down the screen, with the

FIGURE 4-3 The coordinate system used in the graphics library



(0, 0) origin in the upper left corner. To maintain consistency with this convention, the origin of a graphical object is usually defined to be its upper left corner. The `GRect` object stored in the variable `rect` is therefore positioned so that its upper left corner is at the point (150, 50) relative to the upper left corner of the window.

The remaining statements in the `blue_rectangle` function are all examples of method calls. For example, the statement

```
rect.set_color("Blue")
```

sends the rectangle object a `set_color` message asking it to change its color. The argument to `set_color` is a string, which is usually one of the color names from Figure 4-4. Here, the `set_color` call tells the rectangle to set its color to blue.

FIGURE 4-4 Predefined color names in the Portable Graphics Library

```
AliceBlue
AntiqueWhite
Aqua
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Blue
BlueViolet
Brown
BurlyWood
CadetBlue
Chartreuse
Chocolate
Coral
CornflowerBlue
Cornsilk
Crimson
Cyan
DarkBlue
DarkCyan
DarkGoldenrod
DarkGray
DarkGrey
DarkGreen
DarkKhaki
DarkMagenta
DarkOliveGreen
DarkOrange
DarkOrchid
DarkRed
DarkSalmon
DarkSeaGreen
DarkSlateBlue
DarkSlateGray
DarkSlateGrey
DarkTurquoise
DarkViolet
DeepPink
DeepSkyBlue
DimGray
DimGrey
DodgerBlue
FireBrick
FloralWhite
ForestGreen
Fuchsia
Gainsboro
GhostWhite
Gold
Goldenrod
Gray
Grey
Green
GreenYellow
Honeydew
HotPink
IndianRed
Indigo
Ivory
Khaki
Lavender
LavenderBlush
LawnGreen
LemonChiffon
LightBlue
LightCoral
LightCyan
LightGoldenrodYellow
LightGray
LightGrey
LightGreen
LightPink
LightSalmon
LightSeaGreen
LightSkyBlue
LightSlateGray
LightSlateGrey
LightSteelBlue
LightYellow
Lime
LimeGreen
Linen
Magenta
Maroon
MediumAquamarine
MediumBlue
MediumOrchid
MediumPurple
MediumSeaGreen
MediumSlateBlue
MediumSpringGreen
MediumTurquoise
MediumVioletRed
MidnightBlue
MintCream
MistyRose
Moccasin
NavajoWhite
Navy
OldLace
Olive
OliveDrab
Orange
OrangeRed
Orchid
PaleGoldenrod
PaleGreen
PaleTurquoise
PaleVioletRed
PapayaWhip
PeachPuff
Peru
Pink
Plum
PowderBlue
Purple
RebeccaPurple
Red
RosyBrown
RoyalBlue
SaddleBrown
Salmon
SandyBrown
SeaGreen
Seashell
Sienna
Silver
SkyBlue
SlateBlue
SlateGray
SlateGrey
Snow
SpringGreen
SteelBlue
Tan
Teal
Thistle
Tomato
Turquoise
Violet
Wheat
White
WhiteSmoke
Yellow
YellowGreen
```

If the 140 standard web colors listed in Figure 4-4 are not enough for you, the Portable Graphics Library allows you to specify 16,777,216 different colors by indicating the proportion of the three primary colors of light: red, green, and blue. To do so, all you need to do is specify the color as a string in the form "#*rrggbb*", where *rr* indicates the red value, *gg* indicates the green value, and *bb* indicates the blue value. Each of these values is expressed as a two-digit number written in ***hexadecimal,*** or base 16. You may already be familiar with this form of color specification from designing web pages. If not, you will have a chance to learn more about hexadecimal notation in Chapter 7.

The next line in the `blue_rectangle` function is the method call

```
rect.set_filled(True)
```

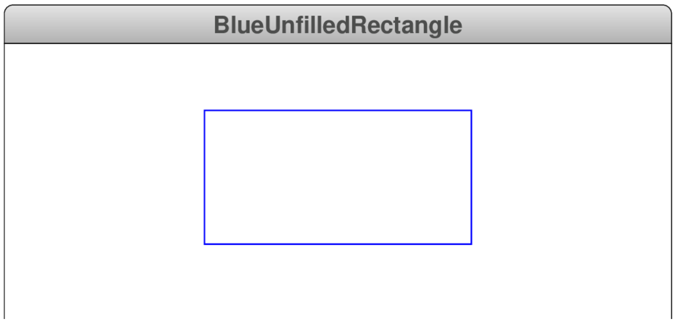
which sends a `set_filled` message to the rectangle. The `set_filled` method takes a Boolean argument, which specifies whether the rectangle is filled or outlined. Calling `rect.set_filled(True)` indicates that the interior of the rectangle should be filled. Conversely, calling `rect.set_filled(False)` indicates that it should not be, which leaves only the outline.

The final line in the `blue_rectangle` function is the method call

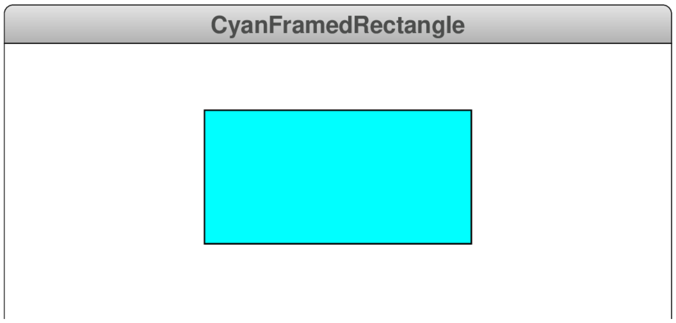
```
gw.add(rect)
```

which sends an `add` message to the graphics window, asking it to add the graphical object stored in `rect` to the contents of the window. Adding the rectangle produces the final contents of the display.

By default, the `GRect` function creates rectangles that are unfilled. Thus, if you left this statement out of `blue_rectangle`, the result would look like this:



For filled shapes, you can set the interior color by calling `set_fill_color` with any of the color names from Figure 4-4. For example, if you replace the call to `set_color("Blue")` in Figure 4-2 with a call to `set_fill_color("Cyan")`, the rectangle would be filled in cyan but outlined in black, like this:



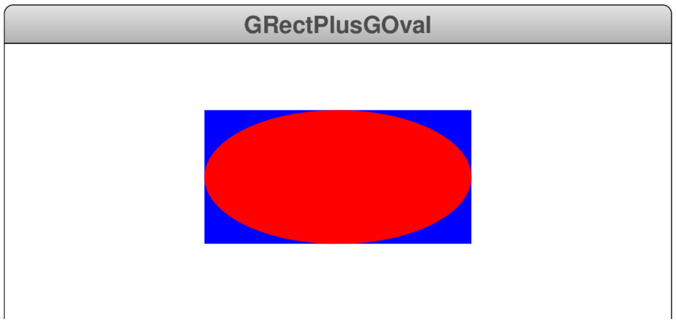
The GOval class

As its name suggests, the GOval class is used to display an oval-shaped figure in a graphics window. The GOval constructor takes the same arguments as GRect, but the two classes display different objects on the screen. The GRect class displays a rectangle whose location and size are determined by the arguments. The GOval class displays the oval that fits exactly inside the corresponding rectangle.

The relationship between the GRect and the GOval classes is most easily illustrated by example. The following function definition takes the code from the earlier BlueRectangle.py program and extends it by adding a GOval with the same coordinates and dimensions:

```
def grect_plus_goval():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    rect = GRect(150, 50, 200, 100)
    rect.set_filled(True)
    rect.set_color("Blue")
    gw.add(rect)
    oval = GOval(150, 50, 200, 100)
    oval.set_filled(True)
    oval.set_color("Red")
    gw.add(oval)
```

The resulting output looks like this:



There are two important things to notice in this example. First, the red GOval extends so that its edges touch the boundary of the rectangle. Second, the GOval, which was added after the GRect, hides the portions of the rectangle that lie underneath the oval. If you were to add these figures in the opposite order, all you would see is the blue GRect, because the entire GOval would be within the boundaries of the GRect.

The GLine class

The GLine class is used to display line segments on the graphics window. The GLine function takes four arguments, which are the *x* and *y* coordinates of the two endpoints. For example, the function call

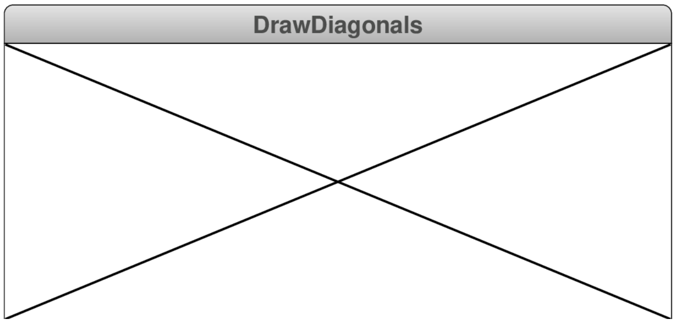
```
GLine(0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

creates a GLine object running from the point (0, 0) in the upper left corner of the graphics window to the point at the opposite corner in the lower right.

The following function uses the GLine class to draw the two diagonals across the graphics window:

```
def draw_diagonals():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    gw.add(GLine(0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT))
    gw.add(GLine(0, GWINDOW_HEIGHT, GWINDOW_WIDTH, 0))
```

Loading this program in the browser generates the following display:



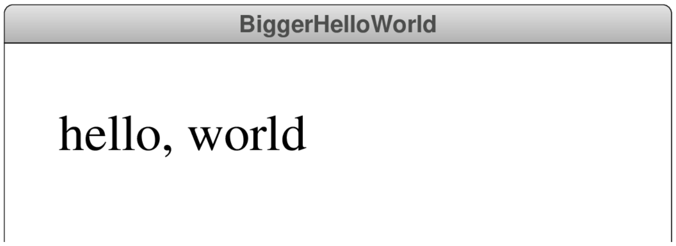
The GLabel class

When you last saw the GLabel class in the GraphicsHelloWorld.py program, the results were not entirely satisfying. The message appearing on the screen was too small to generate much excitement. To make the "hello, world" message bigger, you need to display the GLabel in a different font.

In all likelihood, you already know about fonts from working with other computer applications and have an intuitive sense that fonts determine the style in which characters appear. More formally, a ***font*** is an encoding that maps characters into images that appear on the screen. To change the font of the `GLabel`, you need to send it a `set_font` message, which might look like this:

```
msg.set_font("36px 'Times New Roman'")
```

This call to the `set_font` method tells the `GLabel` stored in `msg` to change its font to one in which the height of a text line is 36 pixels and the font family is Times New Roman, used by *The New York Times.* If you include the `set_font` call in the program, the graphics window will look like this:



BiggerHelloWorld

hello, world

The string passed as the argument to `set_font` is written so that it conforms to style specifications used on the web, which is called ***CSS*** for ***cascading style sheets.*** This string specifies several font properties, which appear in the following order:

- The ***font style,*** which can be used to indicate an alternative form of the font. This specification is ordinarily omitted from the font string to indicate a normal font but may appear as `italic` or `oblique` to indicate an italic variant or a slanted one.
- The ***font weight,*** which specifies how dark the font should be. This specification is omitted for normal fonts but may appear as `bold` to specify a boldface one.
- The ***font size,*** which specifies how tall the characters should be by indicating the distance between two successive lines of text. In CSS, the font size is usually specified in pixel units as a number followed by the suffix `px`, as in the `36px` specification used in the most recent call to `set_font`.
- The ***font family,*** which specifies the name associated with the font. If the name of the font contains spaces, it must be quoted, usually using single quotation marks because the font specification appears inside a double-quoted string. Setting the text in Times New Roman, for example, therefore requires the font string to include `'Times New Roman'`. Because different computers support different fonts, CSS allows a font specification to include several family names separated by commas. The browser will then use the first font family that is available.

FIGURE 4-5 **Generic font family names**

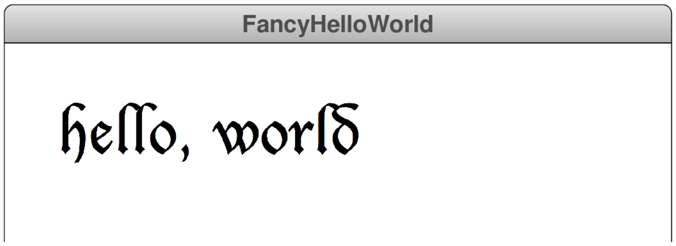
`Serif`	A traditional newspaper-style font in which the characters have short lines at the top and bottom, called ***serifs,*** that lead the eye to read words as single units. The most common example of a serif font is `'Times New Roman'`.
`Sans-Serif`	An unadorned font style lacking serifs. Common examples of sans-serif fonts include `'Arial'` and `'Helvetica Neue'`.
`Monospaced`	A typewriter-style font in which all characters have the same width. The most common monospaced fonts are `'Monaco'` and `'Courier New'`.

CSS defines several ***generic family names,*** which do not identify a specific font but instead describe a type of font that is always available in some form. The most common generic family names appear in Figure 4-5. It is good practice to end the list of preferred font families with one of these generic names to ensure that your program will run on the widest possible set of browsers.

As you probably know from using your word processor, it can be fun to experiment with different fonts. On most Macintosh systems, for example, there is a font called Lucida Blackletter that produces a script reminiscent of the style of illuminated manuscripts of medieval times. To set the message in this font, you could change the `set_font` call in this program to

```
msg.set_font("24px 'Lucida Blackletter',Serif")
```

Note that the font string includes the generic family name `Serif` as an alternative. If the browser displaying the page could not find a font called Lucida Blackletter, it could then substitute one of the standard serif fonts, such as Times New Roman. If, however, it were able to load the Lucida Blackletter font successfully, the output would look something like this:



hello, world

The `GLabel` class uses its own geometric model, which is similar to the ones that typesetters have used over the centuries since Gutenberg’s invention of the printing press. The notion of a font, of course, originally comes from printing. Printers would load different sizes and styles of type into their presses to control the way in which characters appeared on a page. The terminology that the graphics library uses to describe both fonts and labels also derives from the typesetting world. You will find it easier to understand the behavior of the `GLabel` class if you learn the following terms:

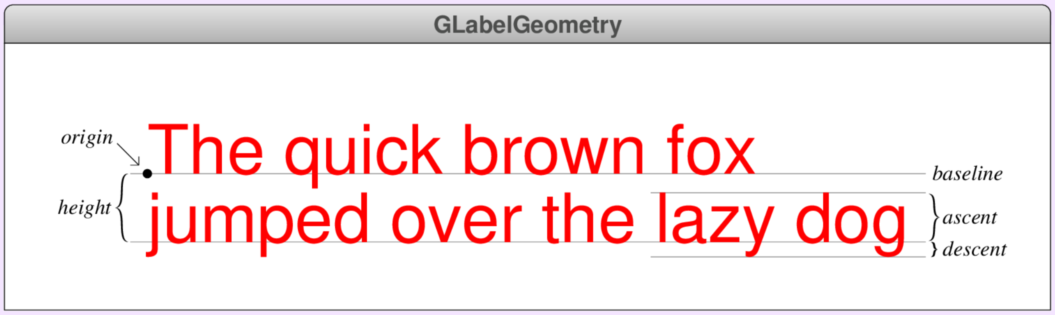
- The ***baseline*** is the imaginary line on which characters sit.
- The ***origin*** is the point at which the text of a label begins. In languages that read left to right, the origin is the point on the baseline at the left edge of the first character. In languages that read right to left, the origin is the point at the right edge of the first character, at the right end of the line.
- The ***height*** is the distance between successive baselines in multiline text.
- The ***ascent*** is the maximum distance that characters extend above the baseline.
- The ***descent*** is the maximum distance that characters extend below the baseline.

The interpretation of these terms in the context of the `GLabel` class is illustrated in Figure 4-6.

The `GLabel` class includes methods that allow you to determine these properties. For example, the `GLabel` class includes a method called `get_ascent` to determine the ascent of the font in which the label appears. In addition, it includes a method called `get_width` that determines the horizontal extent of the `GLabel`.

These methods make it possible to center a label in the window, although they raise an interesting question. The only function you've seen to create a `GLabel` takes its initial coordinates as parameters. If you want to center a label, you won't know those coordinates until after you have created the label. To solve this problem, the function that creates a `GLabel` comes in two forms. The first takes the string for the label along with the *x* and *y* coordinates of the origin. The second leaves out the origin point, setting the origin to the default value of (0, 0).

FIGURE 4-6 The geometry of the `GLabel` class



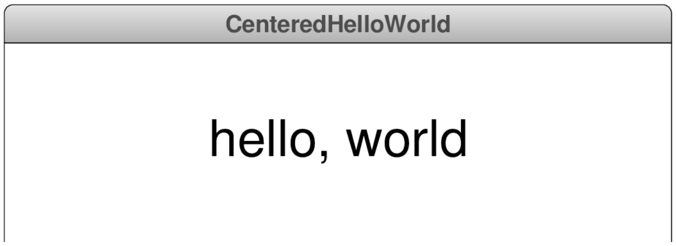
The quick brown fox
jumped over the lazy dog

Suppose, for example, that you want to center the string "hello, world" in the graphics window. To do so, you first need to create the GLabel, then change its font so the label has the appearance you want, and finally determine the dimensions of the label to calculate the correct initial position. You can then supply those coordinates in the add method, which takes optional *x* and *y* parameters to set the location of the object when you add it to the GWindow. The following function from the CenteredHelloWorld.py program implements this strategy:

```
def hello_world():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    msg = GLabel("hello, world")
    msg.set_font("36px 'Sans-Serif'")
    x = (gw.get_width() - msg.get_width()) / 2
    y = (gw.get_height() + msg.get_ascent()) / 2
    gw.add(msg, x, y)
```

The coordinate values necessary to center the GLabel appear in the assignment statements for x and y, which specify the origin point for the centered label. To compute the *x* coordinate of the label, you need to shift the origin left by half the width of the label from the center of the window. Centering the label in the vertical dimension is a bit trickier. You can get pretty close by defining the *y* coordinate to be half the font ascent below the centerline. These declarations also introduce the fact that the GWindow object also implements get_width and get_height, so you can use these methods to determine the width and height of the window.

Running the CenteredHelloWorld.py program creates the following image:



hello, world

If you're a stickler for aesthetic detail, you may find that using get_ascent to center a GLabel vertically doesn't produce the optimal result. Most labels that you display on the canvas will appear to be a few pixels too low. If you want things to look perfect, you may have to adjust the vertical centering by a pixel or two.

The most important methods in the GRect, GOval, GLine, and GLabel classes are summarized in Figure 4-7. Other classes and methods will be introduced in later chapters as they become relevant.

FIGURE 4-7 Summary of methods that apply to graphical objects

Constructors to create graphical objects

GRect(*x*, *y*, *width*, *height*)	Creates a GRect object with the specified dimensions.
GRect(*width*, *height*)	Creates a GRect object at (0, 0) with the specified size.
GOval(*x*, *y*, *width*, *height*)	Creates a GOval that fits inside the corresponding rectangle.
GOval(*width*, *height*)	Creates a GOval object in which the oval fits inside a rectangle of the specified size. The origin of the GOval is (0, 0).
GLine(x_1, y_1, x_2, y_2)	Creates a GLine object connecting (x_1, y_1) and (x_2, y_2).
GLabel(*str*, *x*, *y*)	Creates a GLabel object containing the specified string with its baseline origin at the point (*x*, *y*).
GLabel(*str*)	Creates a GLabel object containing the specified string with its baseline origin at the point (0, 0).

Methods common to all graphical objects

object.get_x()	Returns the *x* coordinate of the object.
object.get_y()	Returns the *y* coordinate of the object.
object.get_width()	Returns the width of the graphical object.
object.get_height()	Returns the height of the graphical object.
object.set_color(*color*)	Sets the color of the object to *color*.

Methods available only for the GRect and GOval classes

object.set_filled(*flag*)	Sets whether this object is filled.
object.set_fill_color(*color*)	Sets the color used to fill the interior of the object.

Methods available only for the GLabel class

object.set_font(*str*)	Sets the font for the label. The format of the font specification is a CSS string as described in the text.
object.get_ascent()	Gets the *font ascent* (maximum distance above the baseline).
object.get_descent()	Gets the *font descent* (maximum distance below the baseline).

4.4 The graphics window

Although it is essential for any program that uses the graphics library, the GWindow class is conceptually different from the other classes in the library. Classes like GRect and GLabel represent objects that you can display in a graphics window. The GWindow class represents the graphics window itself.

The GWindow object is conventionally initialized using the line

```
gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

which appears at the beginning of every program that uses the graphics library. This statement creates the graphics window and installs it in the web page so that it is visible to the user. It also serves to implement the conceptual framework for displaying graphical objects. The conceptual framework implemented by a library package is called its ***model.*** The model gives you a sense of how you should think about working with that package.

One of the most important roles of a model is to establish what analogies and metaphors are appropriate for the package. Many real-world metaphors are possible for computer graphics, just as there are many different ways to create visual art. One possible metaphor is that of painting, in which the artist selects a paintbrush and a color and then draws images by moving the brush across a screen that represents a virtual canvas.

For consistency with the principles of object-oriented design, the Portable Graphics Library uses the metaphor of a ***collage.*** A collage artist works by taking various objects and assembling them on a background canvas. In the real world, those objects might be, for example, geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. The graphics library offers counterparts for all these objects.

The fact that the graphics window uses the collage model has implications for the way you describe the process of creating a design. If you were using the metaphor of painting, you might talk about making a brush stroke in a particular position or filling an area with paint. With the collage model, the key operations are adding and removing objects, along with repositioning them on the background canvas.

Collages also have the property that some objects can be positioned on top of other objects, obscuring whatever is behind them. Removing those objects reveals whatever used to be underneath. In this book, the back-to-front ordering of objects in the collage is called the ***stacking order,*** although you will sometimes see it referred to as ***z-ordering*** in more formal writing. The name *z-ordering* comes from the fact that the stacking order occurs along the axis that comes out of the two-dimensional plane formed by the *x* and *y* axes. In mathematics, the axis coming out of the plane is called the ***z-axis.***

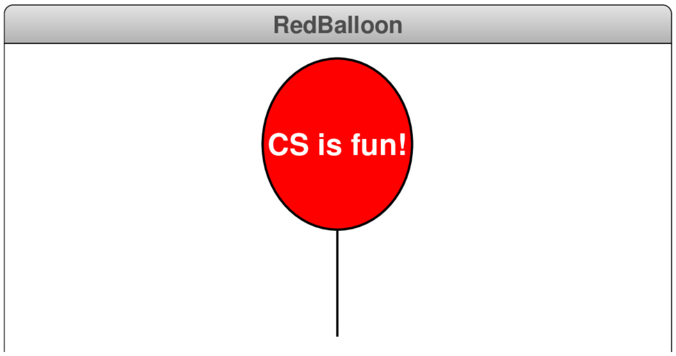
The methods exported by the GWindow class appear in Figure 4-8. For now, your most important methods are add, get_width, and get_height. The other methods will be described in more detail when they are needed for an application.

FIGURE 4-8 **Methods in the GWindow class**

GWindow(*width*, *height*)	Creates a new GWindow object of the specified size.
gw.get_width()	Returns the width of the graphics window.
gw.get_height()	Returns the height of the graphics window.
gw.add(*obj*)	Adds the object to the graphics window.
gw.add(*obj*, *x*, *y*)	Repositions the object at (*x*, *y*) and adds it to the window.
gw.remove(*obj*)	Removes the object from the graphics window.
gw.get_element_count()	Returns the number of objects displayed in the window.
gw.get_element(*k*)	Returns the object at index *k*, numbering from back to front.
gw.get_element_at(*x*, *y*)	Returns the topmost graphical object covering the point (*x*, *y*). If no such object exists get_element_at returns None.
gw.add_event_listener(*type*, *fn*)	Primes the window to respond to events of the specified type by calling *fn*. Event listeners are discussed in Chapter 6.

4.5 Creating graphical applications

You can use the Portable Graphics Library to create graphical displays composed of instances of the GRect, GOval, GLine, and GLabel classes. Suppose, for example, that you want to display a red balloon marked with an upbeat message, as follows:



This program, which appears in Figure 4-9 at the top of the next page, displays three graphical objects:

1. A GOval representing the balloon itself, outlined in black and filled in red
2. A GLine representing the cord attached to the balloon.
3. A GLabel displaying the string "CS is fun!" drawn in white.

FIGURE 4-9 **Program to draw a red balloon on a string**

```
# File: RedBalloon.py

from pgl import GWindow, GLabel, GOval, GLine

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
BALLOON_WIDTH = 140
BALLOON_HEIGHT = 160
BALLOON_LABEL = "CS is fun!"
CORD_LENGTH = 100

# Main program

def red_balloon():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    cx = gw.get_width() / 2
    cy = gw.get_height() / 2
    balloon_x = cx - BALLOON_WIDTH / 2
    balloon_y = cy - (BALLOON_HEIGHT + CORD_LENGTH) / 2
    balloon = GOval(balloon_x, balloon_y, BALLOON_WIDTH, BALLOON_HEIGHT)
    balloon.set_filled(True)
    balloon.set_fill_color("Red")
    cord_y = balloon_y + BALLOON_HEIGHT
    cord = GLine(cx, cord_y, cx, cord_y + CORD_LENGTH)
    label = GLabel(BALLOON_LABEL)
    label.set_font("bold 28px 'Helvetica Neue','Arial','Sans-Serif'")
    label.set_color("White")
    label_x = cx - label.get_width() / 2
    label_y = balloon_y + (BALLOON_HEIGHT + label.get_ascent()) / 2
    gw.add(balloon)
    gw.add(cord)
    gw.add(label, label_x, label_y)

# Startup code

if __name__ == "__main__":
    red_balloon()
```

The objects themselves are not hard to create. What typically takes the most time when you are creating this kind of display is figuring out how to specify the sizes of each object and how to position them in the window so that everything fits together in the way you want it to appear.

The simplest strategy for specifying the sizes and other properties of graphical objects is to define them as constants, as shown in the `RedBalloon.py` example. The constants indicate that the graphics window is 500 pixels wide and 300 pixels high, that the balloon itself is 140 pixels wide and 160 pixels tall, that the message it displays is the string "CS is fun!", and that the cord tied to the base of the balloon is 100 pixels long.

Your primary task in writing the program is to figure out exactly how to position the graphical objects given the values of these constants. The entire figure—the balloon together with its cord—is centered in the graphics window, which means that you have to figure out the coordinate locations for each of the objects relative to the center of the window. The coordinates of the center are easily computed using the following declarations, which will show up repeatedly in other examples:

```
cx = gw.get_width() / 2
cy = gw.get_height() / 2
```

The upper left corner of the oval representing the balloon is then shifted left from cx by half the width of the balloon and shifted upward from cy by half the total height, which is BALLOON_HEIGHT + CORD_LENGTH. The coordinates of the upper left corner of the oval can therefore be computed as follows:

```
balloon_x = cx - BALLOON_WIDTH / 2
balloon_y = cy - (BALLOON_HEIGHT + CORD_LENGTH) / 2
```

The remaining coordinates can be computed similarly. The *y*-coordinate of the top of the cord, for example, can be computed using the following expression:

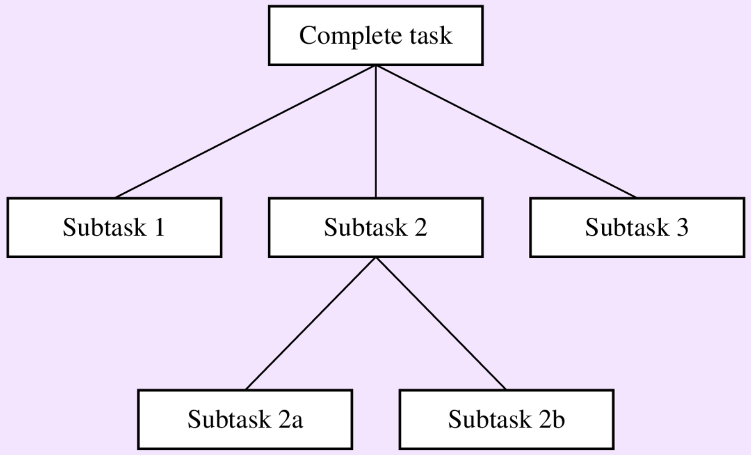
```
cord_y = balloon_y + BALLOON_HEIGHT
```

4.6 Decomposition

One of the most important challenges you will face as a programmer is finding ways to reduce the conceptual complexity of your programs. Large programs are typically difficult to understand as a whole. The only way to keep such programs within the limits of human comprehension is to break them up into simpler, more manageable pieces. In programming, this process is called ***decomposition.***

Decomposition is a fundamental strategy that applies at several levels of the programming process. At the function level, decomposition is the process of breaking a large task down into simpler subtasks that together complete the task as a whole. Those subtasks may themselves require further decomposition, which creates a hierarchy of subtasks of the sort illustrated in Figure 4-10. In that diagram—which presents only the general structure of a typical solution and offers no details about the problem itself—the complete task is decomposed into three primary subtasks. The second of those subtasks is then divided further into two subtasks at an even lower level of detail. Depending on the complexity of the actual problem, the subdivision may require more subtasks or more levels of decomposition.

FIGURE 4-10 Decomposition of a problem into subtasks



Learning how to find the most useful decomposition requires considerable practice. If you define the individual subtasks appropriately, each one will have conceptual integrity as a unit and make the program as a whole much simpler to understand. If you choose the subtasks inappropriately, your decomposition can end up getting in the way. Although this chapter offers some useful guidelines, there are no hard-and-fast rules for selecting a particular decomposition; you will learn how to apply this process through experience.

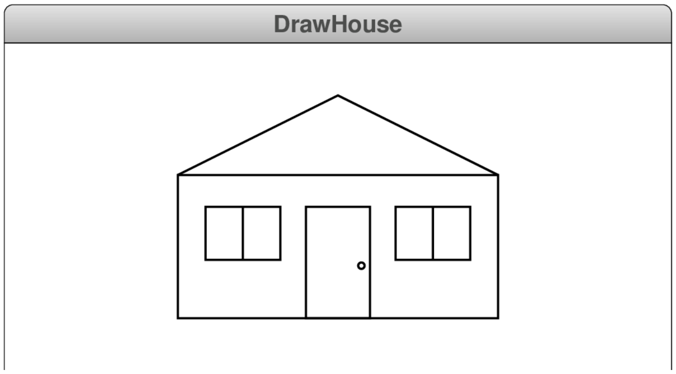
Stepwise refinement

When you are trying to find an effective decomposition, one of the best strategies is to start at the highest levels of abstraction and work your way downward to the details. You begin by thinking about the program as a whole. Assuming that the program is large enough to require decomposition, your next step is to divide the entire problem into its major components. Once you figure out what the major subtasks are, you can then repeat the process to decompose any of the subtasks that are themselves too large to solve in a few lines of code. At the end of this process, you will be left with a set of individual tasks, each of which is simple enough to be implemented as a single function. This process is called ***top-down design,*** or ***stepwise refinement.***

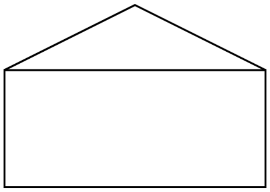
A simple example of decomposition

The best way to understand the process of stepwise refinement is to work through a simple example. The `RedBalloon.py` program in Figure 4-9 is written as a single function. In more sophisticated graphical applications, it makes sense to decompose the program into multiple functions, each of which is responsible for part of the drawing. As you do so, it is important to think carefully about how to decompose the problem so that each of the functions makes sense on its own.

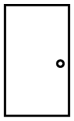
Suppose, for example, that you have decided to draw a picture of your dream house, using a level of detail that one might find in an elementary-school art class. In the end, you want the picture on the graphics window to look like this:



Although there are other reasonable choices, one strategy is to subdivide the problem into functions that draw the house frame, the door, and each of the windows. These functions then have responsibility for drawing the parts of the picture shown in the right margin. You can then draw the entire house by making one call to `draw_frame`, one call to `draw_door`, and two calls to `draw_window`. An implementation of `DrawHouse.py` using this strategy appears in Figure 4-11.


`draw_frame`

An essential part of the decomposition process is figuring out what parameters need to be passed to each of subsidiary function so that it knows precisely how to draw the component of the picture for which that function is responsible. As in the `RedBalloon.py` program, some of the values can be specified using constants. Some, however, have to be passed as parameters to these functions. At a minimum, the `draw_window` function needs to know the *x* and *y* coordinates of the window so that it can draw a window in two different places.


`draw_door`

Deciding which values to declare as constants and which to pass as parameters requires evaluating the tradeoffs between the two models. In general, declaring constants is simpler but limits the program's flexibility. At the same time, passing

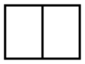

`draw_window`

FIGURE 4-11 Program to draw a simple frame house

```
# File: DrawHouse.py

"""
This program draws a simple frame house at the center of the window.
"""

from pgl import GWindow, GLine, GOval, GRect

# Constants

GWINDOW_WIDTH = 500        # The width of the graphics window
GWINDOW_HEIGHT = 300       # The height of the graphics window
HOUSE_WIDTH = 300          # The width of the house
HOUSE_HEIGHT = 210         # The height of the house including the roof
ROOF_HEIGHT = 75           # The height of the roof above the frame
DOOR_WIDTH = 60            # The width of the door
DOOR_HEIGHT = 105          # The height of the door
DOORKNOB_SIZE = 6          # The diameter of the doorknob
DOORKNOB_INSET_X = 5       # The distance from the knob to the door edge
WINDOW_WIDTH = 70          # The width of each window
WINDOW_HEIGHT = 50         # The height of each window
WINDOW_INSET_X = 26        # The distance from outer wall to the window
WINDOW_INSET_Y = 30        # The distance from the ceiling to the window

# Functions

def draw_house():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    house_x = (gw.get_width() - HOUSE_WIDTH) / 2
    house_y = (gw.get_height() - HOUSE_HEIGHT) / 2
    draw_house_at(gw, house_x, house_y)

def draw_house_at(gw, x, y):
    """
    Draws a simple frame house on the graphics window gw at the
    specified location.  The x and y parameters indicate the upper
    left corner of the bounding box that surrounds the entire house.
    """
    draw_frame(gw, x, y)
    door_x = x + (HOUSE_WIDTH - DOOR_WIDTH) / 2
    door_y = y + HOUSE_HEIGHT - DOOR_HEIGHT
    draw_door(gw, door_x, door_y)
    left_window_x = x + WINDOW_INSET_X
    right_window_x = x + HOUSE_WIDTH - WINDOW_INSET_X - WINDOW_WIDTH
    window_y = y + ROOF_HEIGHT + WINDOW_INSET_Y
    draw_window(gw, left_window_x, window_y)
    draw_window(gw, right_window_x, window_y)
```

FIGURE 4-11 **Program to draw a simple frame house (continued)**

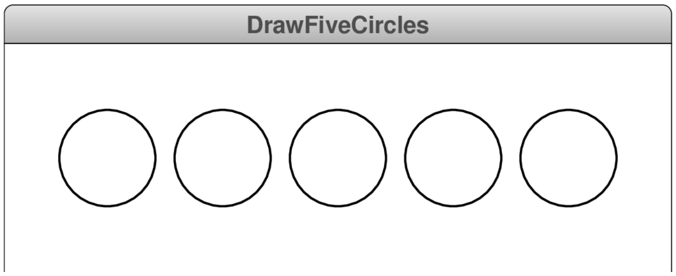
```python
def draw_frame(gw, x, y):
    """
    Draws the frame for the house on the graphics window gw.  The x and
    y parameters indicate the upper left corner of the bounding box.
    """
    roof_y = y + ROOF_HEIGHT
    gw.add(GRect(x, roof_y, HOUSE_WIDTH, HOUSE_HEIGHT - ROOF_HEIGHT))
    gw.add(GLine(x, roof_y, x + HOUSE_WIDTH / 2, y))
    gw.add(GLine(x + HOUSE_WIDTH / 2, y, x + HOUSE_WIDTH, roof_y))

def draw_door(gw, x, y):
    """
    Draws a door (with its doorknob) on the graphics window gw.  The
    x and y parameters indicate the upper left corner of the door.
    """
    gw.add(GRect(x, y, DOOR_WIDTH, DOOR_HEIGHT))
    doorknob_x = x + DOOR_WIDTH - DOORKNOB_INSET_X - DOORKNOB_SIZE
    doorknob_y = y + DOOR_HEIGHT / 2
    gw.add(GOval(doorknob_x, doorknob_y, DOORKNOB_SIZE, DOORKNOB_SIZE))

def draw_window(gw, x, y):
    """
    Draws a rectangular window divided vertically into two panes.  The
    x and y parameters indicate the upper left corner of the window.
    """
    gw.add(GRect(x, y, WINDOW_WIDTH, WINDOW_HEIGHT))
    gw.add(GLine(x + WINDOW_WIDTH / 2, y,
                 x + WINDOW_WIDTH / 2, y + WINDOW_HEIGHT))

# Startup code

if __name__ == "__main__":
    draw_house()
```

too many parameters makes functions harder to understand and use. In most applications, it makes sense to adopt a hybrid strategy in which you use constants to specify values that remain the same throughout the program and parameters to specify values that callers will want to change.

Each of the functions in Figure 4-11 takes three parameters: the graphics window gw and the coordinates x and y, which specify the location at which that part of the entire picture should appear. For consistency with the model used by the Portable Graphics Library, these coordinate values specify the upper left corner of that component of the picture. For graphical objects that don't have an upper left corner, the usual strategy is to have the coordinates refer—as they do for the GOval class—to the upper left corner of the rectangle that encloses the object, which is called its ***bounding box.***

4.7 Control structures and graphics

The control statements you learned about in Chapter 2 come up often in graphical programming, particularly when you need to draw many copies of the same figure in different positions on the graphics window. As an example, the program in Figure 4-12 draws five circles centered in the graphics window, like this:



It is worth taking a look at the code for the `DrawFiveCircles.py` program to make sure you understand how the expressions ensure that the circles are centered.

FIGURE 4-12 **Program to draw five circles centered in the graphics window**

```
# File: DrawFiveCircles.py
"""
This program draws a row of five circles centered in the graphics window.
"""

from pgl import GWindow, GOval

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 200
CIRCLE_SIZE = 75
CIRCLE_SEP = 15

def draw_five_circles():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    cx = gw.get_width() / 2
    cy = gw.get_height() / 2
    r = CIRCLE_SIZE / 2
    for i in range(5):
        x = cx + (i - 2) * (CIRCLE_SIZE + CIRCLE_SEP)
        gw.add(GOval(x - r, cy - r, CIRCLE_SIZE, CIRCLE_SIZE))

# Startup code

if __name__ == "__main__":
    draw_five_circles()
```

When you work with two-dimensional graphical designs, you often need nested loops to arrange graphical objects in both the horizontal and vertical directions. The `Checkerboard.py` program in Figure 4-13, for example, draws a checkerboard that looks like this:

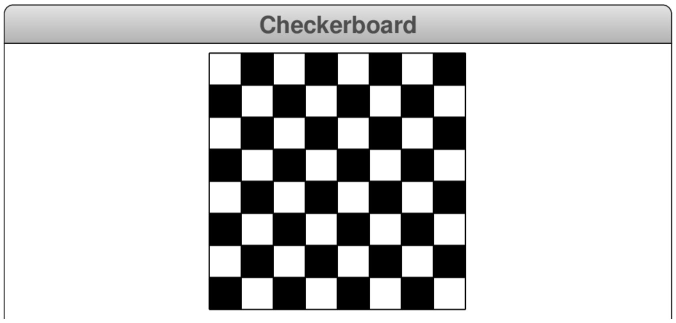


FIGURE 4-13 Program to draw a checkerboard

```
# File: Checkerboard.py

"""
This program draws a checkerboard centered in the graphics window.
"""

from pgl import GWindow, GRect

# Constants

GWINDOW_WIDTH = 500            # Width of the graphics window
GWINDOW_HEIGHT = 300           # Height of the graphics window
N_COLUMNS = 8                  # Number of columns
N_ROWS = 8                     # Number of rows
SQUARE_SIZE = 35               # Size of a square in pixels

def checkerboard():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    x0 = (gw.get_width() − N_COLUMNS ∗ SQUARE_SIZE) / 2
    y0 = (gw.get_height() − N_ROWS ∗ SQUARE_SIZE) / 2
    for row in range(N_ROWS):
        for col in range(N_COLUMNS):
            x = x0 + col ∗ SQUARE_SIZE
            y = y0 + row ∗ SQUARE_SIZE
            sq = GRect(x, y, SQUARE_SIZE, SQUARE_SIZE)
            sq.set_filled((row + col) % 2 != 0)
            gw.add(sq)

# Startup code

if __name__ == "__main__":
    checkerboard()
```

Once again, it is worth taking some time to go through the code in Figure 4-13, paying particular attention to the following details:

- The program is designed so that you can easily change the dimensions of the checkerboard by changing the values of the constants `N_ROWS` and `N_COLUMNS`.
- The checkerboard is arranged so that it is centered in the graphics window. The variables `x0` and `y0` are used to hold the coordinates of the upper left corner of the centered board.
- The decision to fill a square is made by checking whether the sum of its row number and column number is even or odd. For white squares, this sum is even; for black squares, this sum is odd. Note, however, that you don't need to include an `if` statement in the code to test this condition. All you need to do is call the `set_filled` method with the appropriate Boolean value.

4.8 Functions that return graphical objects

It is important to keep in mind that graphical objects are data values in Python in precisely the same way that numbers and strings are. You can therefore assign graphical objects to variables, pass them as arguments to function calls, or have functions return them as results. Functions that return values of one of the `GObject` subclasses can be extremely useful as tools in creating graphical applications that need to display a shape with certain preset features, such as size and color.

The `Target.py` program in Figure 4-14 illustrates this feature by defining a `create_filled_circle` function that takes four arguments: the values *x* and *y* representing the coordinates of the center of the circle, a number *r* specifying the radius of the circle, and a string *color* indicating the Python color name. The `Target.py` program calls `create_filled_circle` three times to create three circles that alternate in color between red and white and progressively decrease in size. The radius of the outer circle is given by the constant `OUTER_RADIUS`. The two inner circles are two-thirds and one-third that size, respectively. Running the `Target.py` program produces the following output:

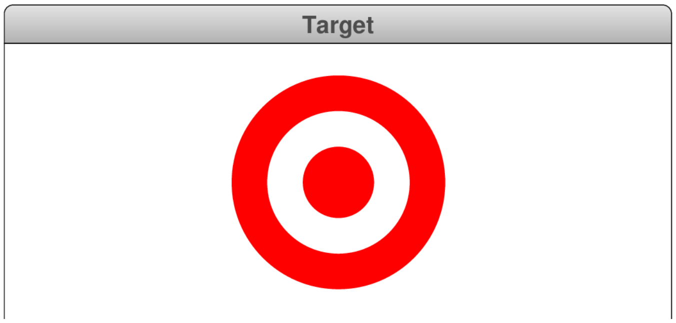


FIGURE 4-14 **Program to draw a red and white target on the graphics window**

```
# File: Target.py

"""
This program draws a target at the center of the graphics window composed
of three concentric circles alternately colored red and white.
"""

from pgl import GWindow, GRect, GOval

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 200
TARGET_RADIUS = 75

# Functions

def target():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    cx = gw.get_width() / 2
    cy = gw.get_height() / 2
    gw.add(create_filled_circle(cx, cy, TARGET_RADIUS, "Red"))
    gw.add(create_filled_circle(cx, cy, 2 * TARGET_RADIUS / 3, "White"))
    gw.add(create_filled_circle(cx, cy, TARGET_RADIUS / 3, "Red"))

def create_filled_circle(x, y, r, color):
    """
    Creates a circle of radius r centered at the point (x, y) filled
    with the specified color.
    """
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.set_color(color)
    circle.set_filled(True)
    return circle

# Startup code

if __name__ == "__main__":
    target()
```

Summary

This chapter introduced the Portable Graphics Library, which allows you to create simple pictures on the screen using lines, rectangles, ovals, and labels. Along the way, you had a chance to practice using objects in Python.

Important points introduced in the chapter include:

- The graphical programs in this book use the *Portable Graphics Library,* which is a collection of graphical tools designed for use in introductory courses.

- Python supports a modern style of programming called the *object-oriented paradigm,* which focuses attention on data objects and their interactions.
- In the object-oriented paradigm, an *object* is a conceptually integrated entity that combines the state of that object and the operations that affect its state. Each object is a representative of a *class,* which is a template that defines the attributes and operations shared by all objects of a particular type. A single class can give rise to many different objects; each such object is an *instance* of that class.
- Objects communicate by sending *messages.* In Python, those messages are implemented by calling *methods,* which are simply functions that belong to a particular class.
- Method calls in Python use the receiver syntax, which looks like this:

  ```
  receiver.name(arguments)
  ```

  The *receiver* is the object to which the message is sent, *name* indicates the name of the method that responds to the message, and *arguments* is a list of values that convey any additional information carried by the message.
- Functions that create new objects are called *constructors* and conventionally have names that begin with an uppercase letter.
- The first line in any Python program that uses the Portable Graphics Library creates a `GWindow` object using the following declaration:

  ```
  gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
  ```

  The constants `GWINDOW_WIDTH` and `GWINDOW_HEIGHT` specify the dimensions of the graphics window in *pixels,* which are the tiny dots that cover the face of the display. Once you have initialized the variable `gw`, you can then create graphical objects of various kinds and add them to the window.
- This chapter introduces four classes of graphical objects—`GRect`, `GOval`, `GLine`, and `GLabel`—that represent rectangles, ovals, line segments, and text strings, respectively. Other graphical objects are introduced in later chapters.
- All graphical objects support the method `set_color`, which takes the name of the color as a string. Python defines 140 standard colors whose names appear in Figure 4-4 on page 105.
- The `GRect` and `GOval` classes use `set_filled` and `set_fill_color` to control whether the shape is filled and what color is used for the interior.
- The `GLabel` class uses the `set_font` method to set the font in which the label appears. The argument to `set_font` is the CSS specification of a font, which is described on page 107.
- The `GLabel` class uses a geometric model that is different from the one used by the other graphical objects. That model is illustrated in Figure 4-6 on page 109.

- One of the most effective strategies for managing the complexity of programs is *decomposition,* which is the process of breaking a large task down into smaller, more manageable subtasks.
- In most cases, it makes sense to apply decomposition by starting at the level of the problem as a whole and then working your way downward to the details. This strategy is called *top-down design* or *stepwise refinement.*
- Graphical objects are data values in Python in the same way that numbers and strings are. You can therefore assign graphical objects to variables, pass them as arguments to function calls, or have functions return them as results.
- The `Target.py` program in Figure 4-14 defines a `create_filled_circle` function that illustrates the strategy of returning graphical objects from functions. This technique will be used in many programs throughout the remaining chapters.

Review questions

1. What is the name of the Python library used in this chapter to implement programs that produce graphical output?

2. In your own words, define the terms *class, object,* and *method.*

3. What is a *reference?*

4. The object-oriented paradigm uses the metaphor of sending messages to model communication between objects. How does Python implement this idea?

5. What is the *receiver syntax?*

6. What is a *constructor?*

7. What is the first line in every graphical program that appears in this book?

8. What are the four classes of graphical objects introduced in this chapter?

9. How do you change the color of a graphical object?

10. What is the purpose of the `set_filled` and `set_fill_color` methods in the `GRect` and `GOval` classes?

11. What is the format of the argument string passed to `set_font`?

12. Define the following terms in the context of the `GLabel` class: *baseline, origin, height, ascent,* and *descent.*

13. Explain the purpose of the following lines in the `CenteredHelloWorld.py` program:

```
x = (gw.get_width() - msg.get_width()) / 2
y = (gw.get_height() + msg.get_ascent()) / 2
```

Why is there a minus sign in the calculation of the *x* coordinate and a plus sign in the calculation of the *y* coordinate?

14. When you center a `GLabel` vertically using the `get_ascent` method, why does the resulting text often appear to be a few pixels too low?

15. What is the *collage model?*

16. What is meant by the term *stacking order?* What other term is often used for the same purpose?

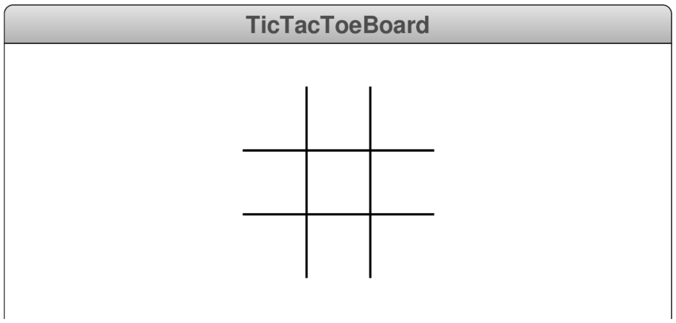
17. Explain in your own words the process of *stepwise refinement.*

18. What two strategies does this chapter propose for conveying information between a program and the individual functions that result from decomposing that program into smaller pieces? What are the advantages and disadvantages of each of these strategies?

Exercises

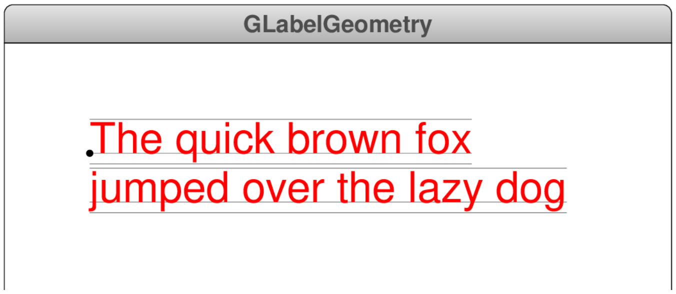
1. Use your program editor to create the file `GraphicsHelloWorld.py` exactly as it appears in Figure 4-1. Make a copy of the `pgl.py` library file and store it in the same folder. Invoke Python on `GraphicsHelloWorld.py` to show that you can get execute a graphical program.

2. Write a graphical program `TicTacToeBoard.py` that draws a Tic-Tac-Toe board centered in the graphics window, as shown in the following sample run:



The size of the board should be specified as a constant, and the diagram should be centered in the window, both horizontally and vertically.

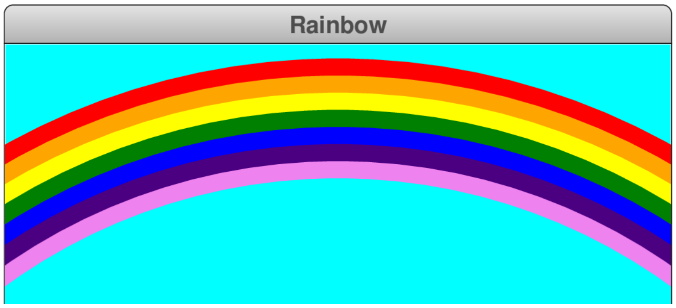
3. Draw a simplified version of Figure 4-6, which illustrates the geometry of the `GLabel` class. In your implementation, you should display the two strings (`"The quick brown fox"` and `"jumped over the lazy dog"`) in red using a sans-serif font that is large enough to make the guidelines easy to see. Then for each of the strings, you should draw a gray line along the baseline, the line that marks the font ascent, and the line that marks the font descent. Finally, you should draw a small filled circle indicating the baseline origin of the first string. The graphics window will then look like this:



```
The quick brown fox
jumped over the lazy dog
```

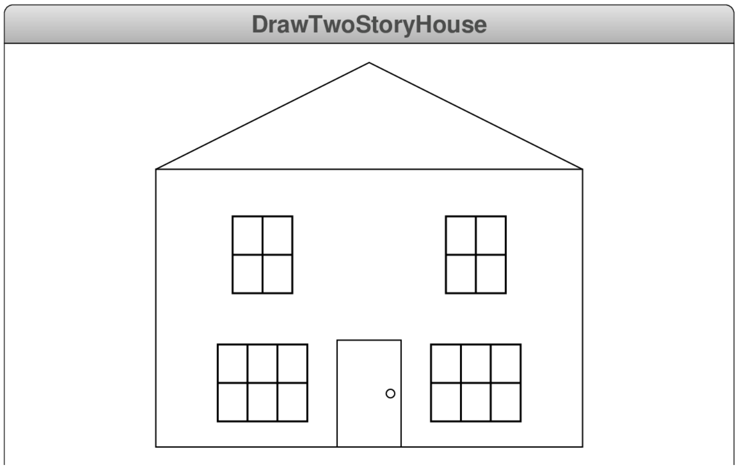
This output is a little more honest than Figure 4-6 about the font ascent, which appears slightly above the top of the uppercase characters.

4. Use the graphics library to draw a rainbow that looks something like this:



Starting at the top, the seven bands in the rainbow are red, orange, yellow, green, blue, indigo, and violet, respectively; cyan makes a lovely color for the sky. Remember that this chapter defines only the `GRect`, `GOval`, `GLine`, and `GLabel` classes and does not include a graphical object that represents an arc. It will help to think outside the box, in a more literal sense than usual.

5. Use top-down design to design a program that creates the following picture of a more complex house than the one presented in Figure 4-11:

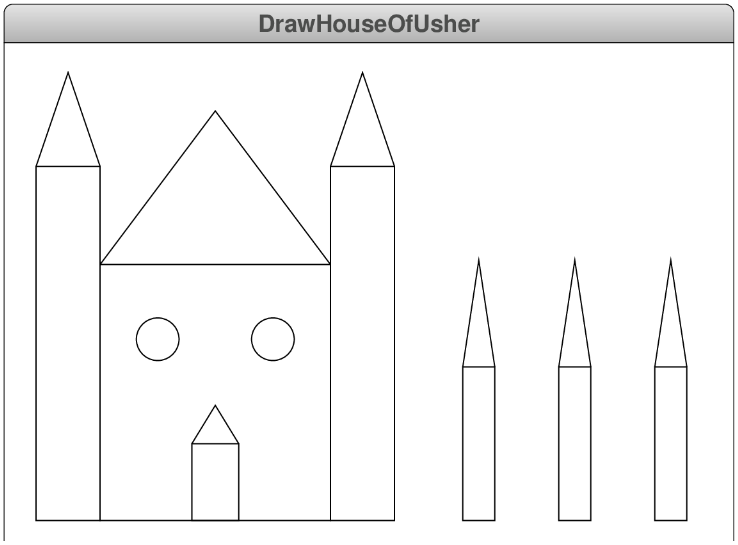


Think carefully about the decomposition to see whether it is possible to exploit common features of the design.

6. If the house diagrams in Figure 4-11 and the preceding exercise seem a bit mundane, you might instead want to draw a diagram of the House of Usher, which Edgar Allan Poe describes as follows:

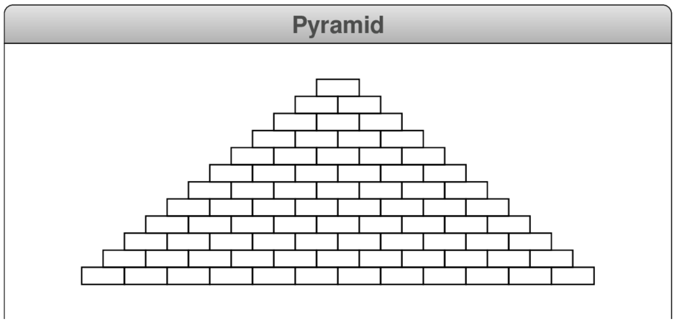
> With the first glimpse of the building, a sense of insufferable gloom pervaded my spirit. . . . I looked upon the scene before me—upon the mere house, and the simple landscape features of the domain—upon the bleak walls—upon the vacant eye-like windows . . . upon a few white trunks of decayed trees—with an utter depression of soul.

From Poe's description, you might draw a house that looks something like this:



The figure on the left is the house with its "vacant eye-like windows" and the three figures on the right are a stylized rendition of the "few white trunks of decayed trees."

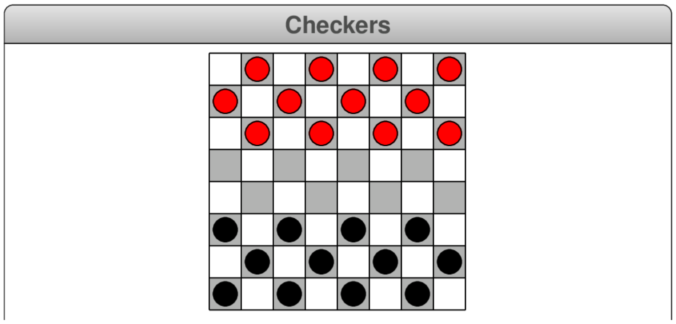
7. Write a program that displays a pyramid on the graphics window. The pyramid consists of bricks in horizontal rows, arranged so that the number of bricks in each row decreases by one as you move upward, as follows:



The pyramid should be centered in the window both horizontally and vertically and should use constants to define the dimensions of each brick and the height of the pyramid.

8. Rewrite (and suitably rename) the `DrawFiveCircles.py` program shown in Figure 4-12 so that the number of circles is given by the constant `N_CIRCLES`.

9. Enhance the `Checkerboard.py` program shown in Figure 4-13 so that the graphics window also displays the red and black checkers corresponding to the initial state of the game, which looks like this:

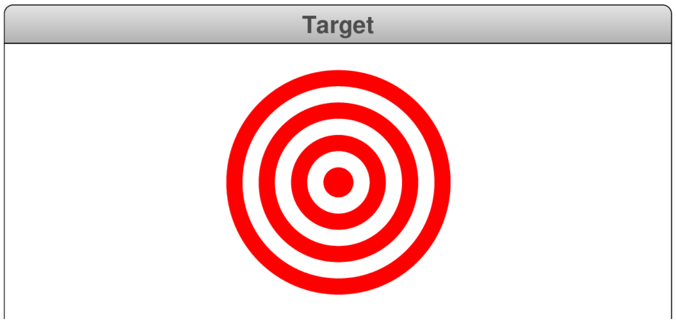


The other change in this program is that the color of the dark squares has been changed from black to gray so that the black checkers are not lost against the background.

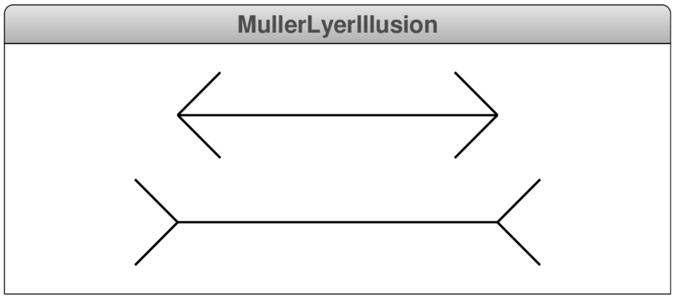
10. Rewrite the `Target.py` program from Figure 4-14 so that the number and radii of the circles are controlled by the following constants:

```
N_CIRCLES = 7
OUTER_RADIUS = 75
INNER_RADIUS = 10
```

Given those values, the program should generate the following display:



11. Classical optical illusions offer a rich source of interesting graphical exercises. One of the simplest examples is the ***Müller-Lyer illusion,*** named after the German sociologist Franz Karl Müller-Lyer, who first described the effect in 1889. In one of its more common forms, the Müller-Lyer illusion asks the viewer which of the two horizontal lines is longer in the following figure:

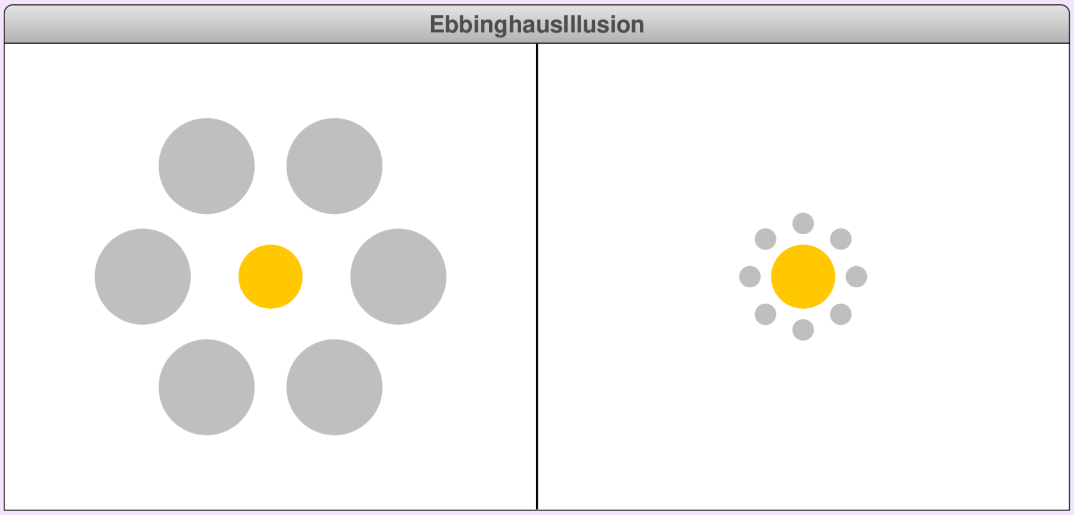


Most people are convinced that the bottom line is longer, but the two lines are in fact the same length.

Write a program to produce the Müller-Lyer illusion as it appears in this example. Make sure you use constants to define parameters like the lengths of the various lines.

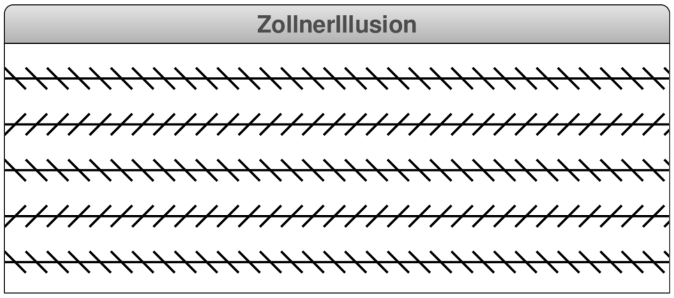
12. Another illusion that shows how context affects the perception of relative size is the ***Ebbinghaus illusion,*** which was discovered by the German psychologist Hermann Ebbinghaus and published in a 1901 book by the British psychologist

FIGURE 4-15 **The Ebbinghaus illusion: Are the inner circles the same size?**

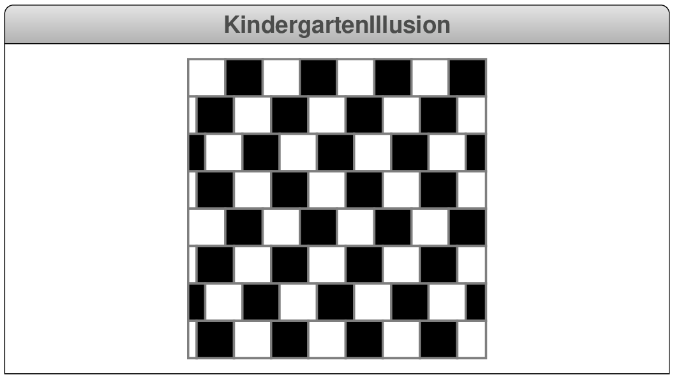


Edward Tichener. This illusion, which appears in Figure 4-15, makes it seem as if the central circle on the left is smaller than the circle on the right, even though the two are the same size. Write a program to produce this illusion.

13. Write a program to produce the ***Zöllner illusion,*** which was discovered by the German astrophysicist Johann Karl Friedrich Zöllner in 1860. In this illusion, the diagonal lines that run in opposite directions on every other line make it difficult to see that the horizontal lines are actually parallel:



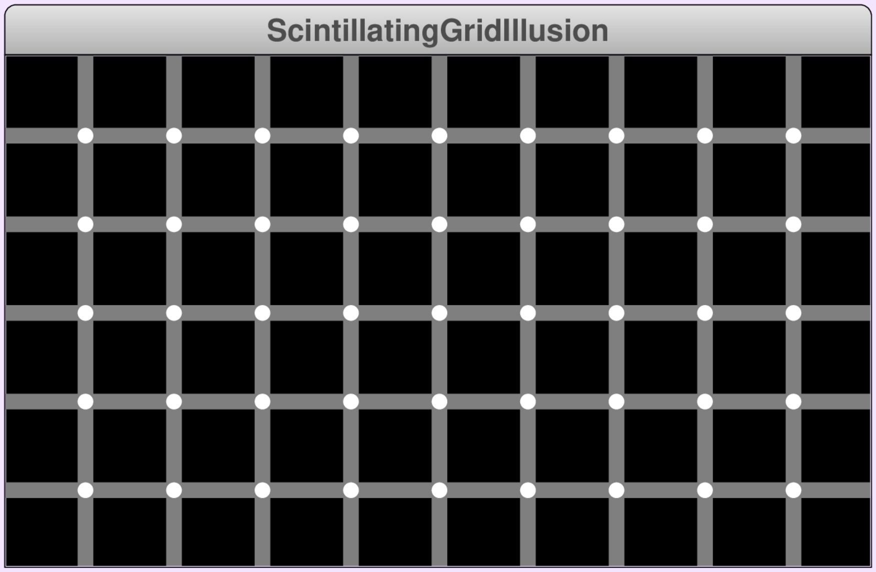
14. An even more exotic illusion is the ***kindergarten illusion*** (also called the ***café wall illusion***), which was first described by the American psychologist Arthur Henry Pierce in 1898. In this illusion, shifting the squares slightly on each row of a checkerboard pattern makes the horizontal lines of the checkerboard appear slanted instead of straight. Starting with the `Checkerboard` program from Figure 4-13, make the changes necessary to produce the following image:



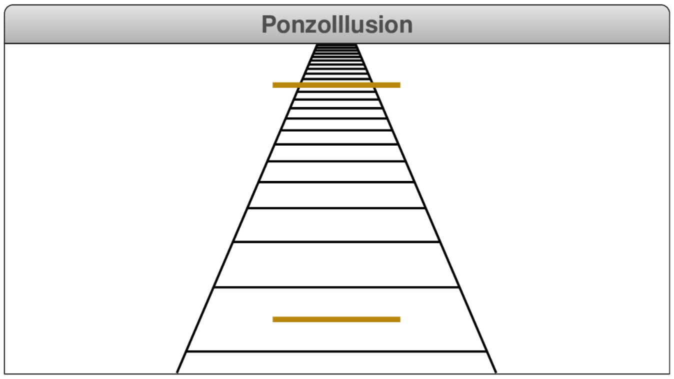
15. The ***scintillating grid illusion*** shown in Figure 4-16 was popularized by Elke Lingelbach in the 1990s and is based on an earlier illusion published by Ludimar Hermann in 1870. In this illusion, the viewer sees black dots inside the white circles at the intersections of the grid. Write a program that replicates this illusion.

16. Our visual sense is powerfully affected by our assumptions about an image. In 1911, the Italian psychologist Mario Ponzo showed that people expect objects viewed at a distance in a perspective drawing to appear smaller. If an object appears to violate the rules of perspective, our minds compensate by changing our perception of its size.

FIGURE 4-16 The scintillating grid illusion

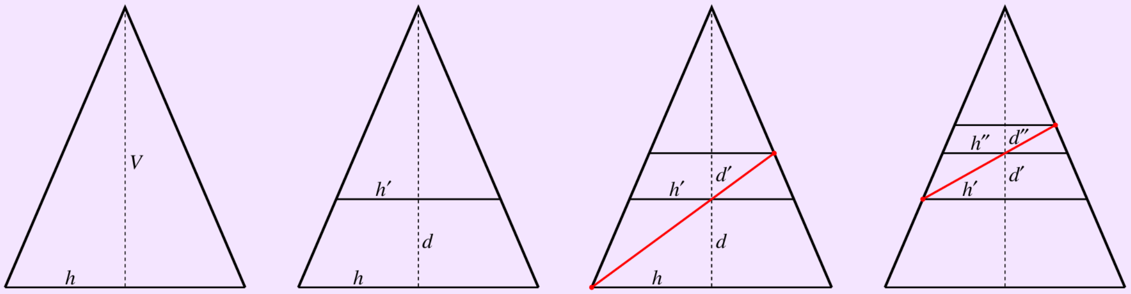


In one of its more popular forms, the ***Ponzo illusion*** illustrates this principle by superimposing two horizontal lines onto a stylized image of a railroad track receding into the distance. Since our experience assures us that the rails are equally far apart all the way down the track, a line that crosses it must be larger than one that falls entirely inside it, as illustrated in the following example:



Your mission is to reproduce this image using ***one-point perspective,*** which is a technique for representing a three-dimensional scene in a two-dimensional drawing. In a drawing that uses one-point perspective, objects move toward a single ***vanishing point*** as they move farther from the viewer. This technique was developed during the early Renaissance and was used by the Florentine artist and architect Filippo Brunelleschi in a 1415 painting. Your challenge in creating the Ponzo illusion is to figure out where each of the crossties should go in the railroad track as it vanishes into the distance. The mathematical formulae you need to perform these calculations appear in Figure 4-17.

FIGURE 4-17 **Mathematics of perspective-based foreshortening**



Step 1. Define parameters

V = distance to vanishing point

$$h = \frac{\textit{first crosstie length}}{2}$$

Define the slope $m = \frac{V}{h}$

Step 2. Draw second crosstie

d = distance to next crosstie

$$h' = h - \frac{d}{m}$$

Step 3. Find next crosstie

Red line marks next crosstie.

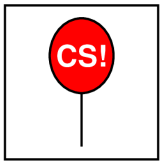
$$d' = \frac{d - \frac{d^2}{mh}}{1 + \frac{d}{mh}}$$

Step 4. Repeat the process

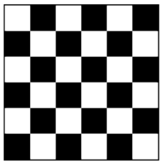
Use the same process to find each new crosstie.

17. Back in the early 1990s—long before Python existed—Julie Zelenski and Katie Capps Parlante developed a lovely graphics assignment that we used in Stanford's introductory course for many years. The goal of the assignment was to draw a ***sampler quilt,*** which is composed of several different block types that illustrate a variety of quilting styles.

    For this exercise, your job is to use the graphics library to create the sampler quilt shown in Figure 4-18. This quilt is composed of a repeating pattern of the following four blocks, three of which are examples of previous work:

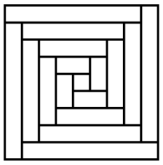


Balloon



Checkerboard

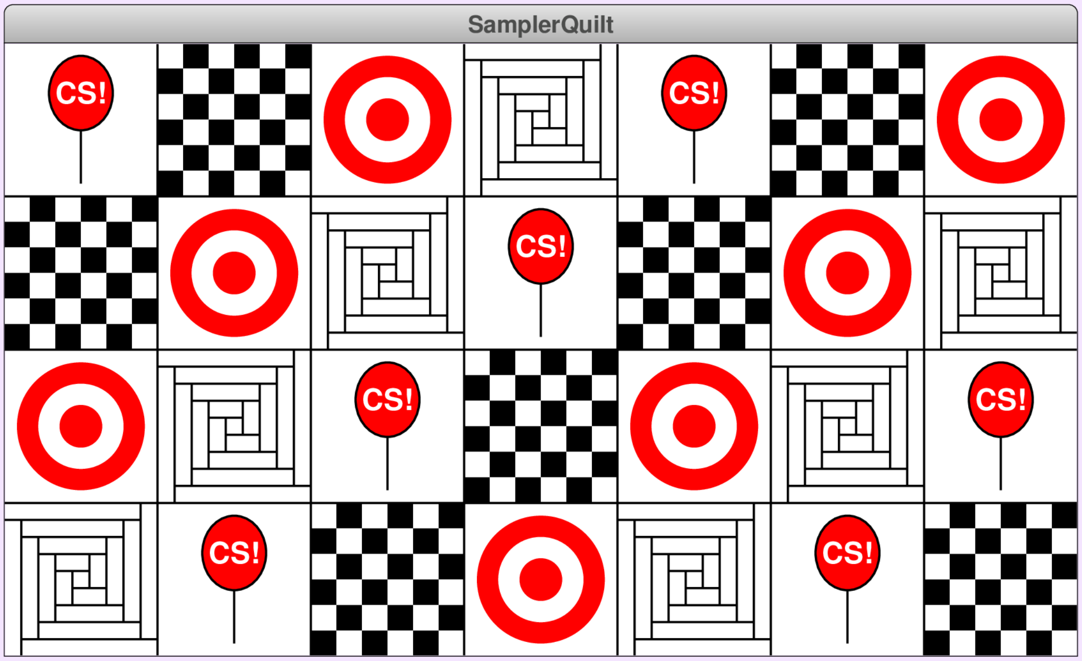
Target



Log Cabin

The only new block is the fourth one, which is a classic quilting pattern called a ***log cabin block.*** This block is composed of rectangles that spiral inward toward a square in the center. The width of each rectangle and the width of the central square are all the same, which means that the dimensions are determined by the block size and the number of frames in the spiral.

FIGURE 4-18 **Graphical display of a sampler quilt**

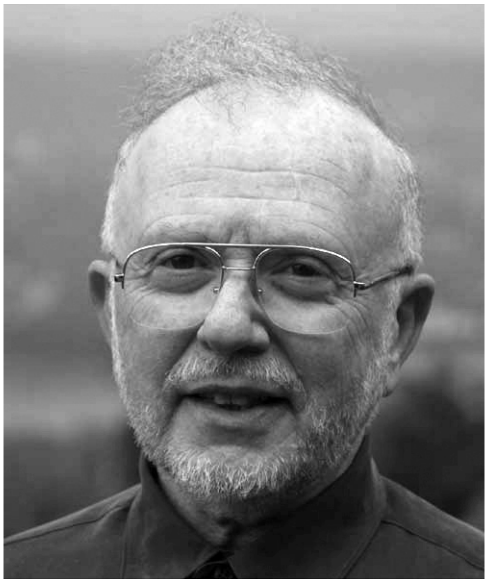


a

CHAPTER 5
Functions

Our module structure is based on the decomposition criteria known as information hiding. According to this principle, system details that are likely to change independently should be the secrets of separate modules.

— David Parnas, Paul Clements, and David Weiss,
"The modular structure of complex systems," 1984



David Parnas (1941–)

David Parnas is Professor of Software Engineering *emeritus* at the University of Limerick in Ireland, where he directed the Software Quality Research Laboratory, and has also taught at universities in Germany, Canada, and the United States. His most influential contribution to software engineering is his groundbreaking 1972 paper entitled "On the criteria to be used in decomposing systems into modules," which provided much of the foundation for the strategy of decomposition described in this chapter. Professor Parnas also attracted considerable public attention in 1985 when he resigned from a Department of Defense panel investigating the software requirements of the proposed Strategic Defense Initiative—more commonly known as "Star Wars"—on the grounds that the requirements of the system were impossible to achieve. For his courageous stand in bringing these problems to light, Parnas received the 1987 Norbert Wiener Award from Computer Professionals for Social Responsibility.

This chapter examines in more detail the concept of a function, which was initially presented in Chapter 1. A function is a set of statements that have been collected together and given a name. Because functions allow the programmer to invoke the entire set of operations using a single name, programs become much shorter and much simpler. Without functions, programs would become unmanageable as they increased in size and sophistication.

In order to appreciate how functions reduce the complexity of programs, it helps to examine the role of functions from two distinct philosophical perspectives, *reductionism* and *holism.* ***Reductionism*** is the philosophical principle that the whole of an object can best be understood by understanding the parts that make it up. Its antithesis is ***holism,*** which recognizes that the whole is often more than the sum of its parts. As you try to master the discipline of dividing large programs into functions, you must learn to see the process from each of these perspectives. If you concentrate only on the big picture, you will end up not understanding the tools you need for solving problems. However, if you focus exclusively on details, you will invariably miss the forest for the trees.

When you are first learning about programming, the best approach is usually to alternate between these two perspectives. Taking the holistic view helps sharpen your intuition about the programming process and enables you to stand back from a program and say, "I understand what this function does." Taking the reductionistic view allows you to say, "I understand how this function works." Both perspectives are essential. You need to understand how functions work so that you can code them correctly. At the same time, you must be able to take a step backward and look at functions holistically, so that you also understand why they are important and how to use them effectively.

5.1 A quick review of functions

Although you have been working with functions ever since you wrote your first programs in Chapter 1, you have so far seen only a part of the computational power that functions provide. Before delving more deeply into the details of how functions work, it helps to review some basic terminology. First of all, a ***function*** consists of a set of statements that have been collected together and given a name. The act of executing the set of statements associated with a function is known as ***calling*** that function. To indicate a function call in Python, you write the name of the function, followed by a list of expressions enclosed in parentheses. These expressions, which are called ***arguments,*** allow the caller to pass information to the function.

The syntax of a function definition

A typical function definition has the form shown in the syntax box on the right. The *name* component of this pattern indicates the function name, *parameters* is the list of parameter names that receive the values of the arguments, and *statements* represents the body of the function. Functions that return a value to the caller must contain at least one `return` statement that specifies the value of the function, as illustrated in the second syntax box.

```
def name(parameters):
    statements
```

```
return value
```

These syntactic patterns are illustrated in the definition of the `max` function from Chapter 2, which looks like this:

```
def max(x, y):
   if x > y:
      return x
   else:
      return y
```

This function has the name `max` and takes two parameters, `x` and `y`. The statements in the body decide which of these two values is larger and then return that value.

Functions, however, are often called simply for their effect and need not return a value. For example, the Python functions that implement complete programs don't include a `return` statement. Some languages distinguish a function that returns a value from one that doesn't by calling the latter a *procedure.* Python uses the term *function* for both types. This terminology is technically accurate because Python functions always return a value, which is the Python constant `None` if no `return` statement appears.

Parameter passing

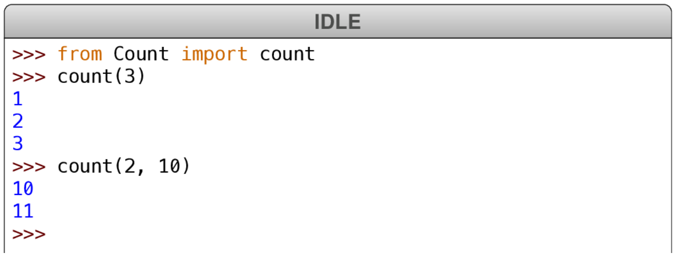
In the function calls you have seen so far, the arguments supplied by the caller are copied to the parameter variables in the order in which they appear. The first argument is assigned to the first parameter variable, the second argument to the second parameter variable, and so on. Parameters passed by their order in the argument list are called ***positional parameters.***

When you use positional parameters, the variable names in the caller and the called function are completely irrelevant to the process by which parameter values are assigned. There may well be a variable named `x` in both the calling function and in the parameter list for the function being called. That reuse of the same name, however, is merely a coincidence. Local variable names and parameter names are visible only inside the function in which their declarations appear.

Python allows a function to specify a value for a parameter that the caller fails to supply. Such parameters are called ***default parameters.*** Default parameters appear in the function header line with an equal sign and a default value. For example, the following function displays `n` consecutive integers, beginning with the value `start` if two arguments are supplied and with the value 1 if the second argument is missing:

```
def count(n, start=1):
    for i in range(n):
        print(start + i)
```

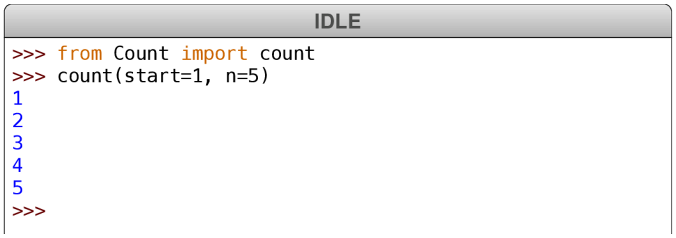
The following IDLE session illustrates the operation of `count`, both when it is given a second argument and when it is not:



IDLE

```
>>> from Count import count
>>> count(3)
1
2
3
>>> count(2, 10)
10
11
>>>
```

Python also allows callers to pass arguments by including the parameter name and an equal sign in the function call. For example, if you cannot remember the order of parameters for the `count` function, you can write the arguments in either order by including the parameter names, as follows:



IDLE

```
>>> from Count import count
>>> count(start=1, n=5)
1
2
3
4
5
>>>
```

Parameters identified by name are called ***keyword parameters,*** even though the names are not in any way related to Python keywords like `def` or `while`.

Default and keyword parameters are useful in designing library functions that are easy to use. The section entitled “Designing your own libraries” later in this chapter includes several examples of each of these styles.

5.2 The mechanics of function calls

Although you can certainly get by with an intuitive understanding of how the function-calling process works, it helps to understand precisely what happens when one function calls another in Python. The sections that follow describe the process in detail and then walk you through a simple example.

The steps in calling a function

Whenever a function call occurs, Python executes the following operations:

1. The calling function computes values for each argument using the bindings of local variables in its own context. Because the arguments are expressions, this computation can involve operators and other functions; the calling function evaluates these expressions before execution of the new function begins.
2. The system creates new space for all the local variables required by the new function, including the variables in the parameter list. These variables are allocated together in a block, which is called a ***stack frame.***
3. Each positional argument is copied into the corresponding parameter variable.
4. All keyword arguments are copied to the parameter with the same name.
5. For parameters that include default values, Python assigns those values to any arguments that are still unspecified. If any parameters are still unassigned after this step, Python reports an error.
6. The statements in the function body are executed until the program encounters a `return` statement or there are no more statements to execute.
7. The value of the `return` expression, if any, is evaluated and returned as the value of the function.
8. The stack frame created for this function call is discarded. In the process, all local variables disappear.
9. The calling program continues, with the returned value substituted in place of the call. The point to which the function returns is called the ***return address.***

Although this process may seem to make at least some sense, you probably need to work through an example or two before you understand it fully. Reading through the example in the next section will give you some insight into the process, but it will be even more helpful to take one of your own programs and walk through it at the same level of detail. And while you can trace through a program on paper or a whiteboard, it may be best to get yourself a supply of 3 × 5 index cards and then use a card to represent each stack frame. The advantage of the index-card model is that you can create a stack of index cards that closely models the operation of the computer. Calling a function adds a card; returning from the function removes it.

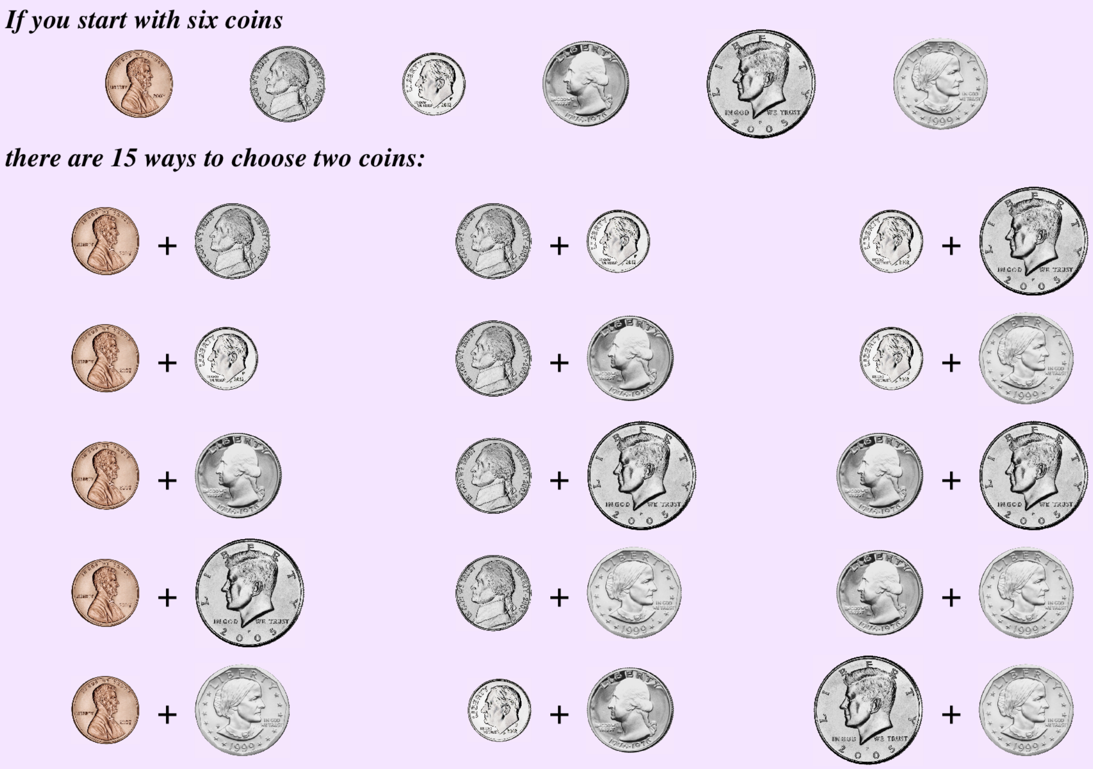
The combinations function

The function-calling process is most easily illustrated in the context of a specific example. Suppose that you have a collection of six coins, which in the United States might be a penny, a nickel, a dime, a quarter, a half-dollar, and a dollar. Given those six coins, how many ways are there to choose two of them? As you can see from the full enumeration of the possibilities in Figure 5-1, the answer is 15. However, as a computer scientist, you should immediately think about the more general question: given a set containing n distinct elements, how many ways can you choose a subset with k elements? The answer to that question is computed by the ***combinations function*** $C(n, k)$, which is defined as

$$C(n, k) = \frac{n!}{k! \times (n - k)!}$$

where the exclamation point indicates the factorial function, which you saw in Chapter 2. The code to compute the combinations function in Python appears in Figure 5-2.

FIGURE 5-1 Illustration of the combinations function



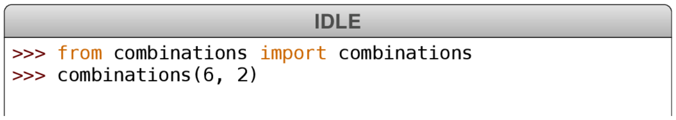
If you start with six coins

there are 15 ways to choose two coins:

FIGURE 5-2 Python implementation of the mathemematical combinations function $C(n, k)$

```
# File: combinations.py

"""
This module exports an implementation of the mathematical combinations
function C(n, k), which is the number of ways of selecting k objects
from a set of n distinct objects.
"""

def combinations(n, k):
    """
    Returns the mathematical combinations function C(n, k), which is
    the number of ways one can choose k elements from a set of size n.
    """
    return fact(n) // (fact(k) * fact(n - k))

def fact(n):
    """
    Returns the factorial of n, which is the product of all the
    integers between 1 and n, inclusive.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

As you can see from Figure 5-2, the `combinations.py` file contains two functions. The `combinations` function computes the value of $C(n, k)$, and the now-familiar `fact` function computes factorials. An IDLE session just before making the call to `combinations(6, 2)` might look like this:



```
>>> from combinations import combinations
>>> combinations(6, 2)
```

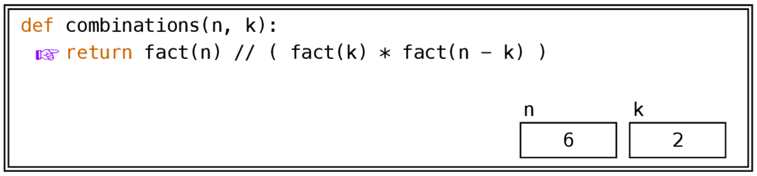
Tracing the combinations function

While the `combinations` function is interesting in its own right, the purpose of the current example is to illustrate the steps involved in calling functions. When the user enters a function call in the IDLE window, the Python interpreter invokes the standard steps in the function-calling process.

As always, the first step is to evaluate the arguments in the current context. In this example, the arguments are the numbers 6 and 2, so the evaluation process simply keeps track of these two values.

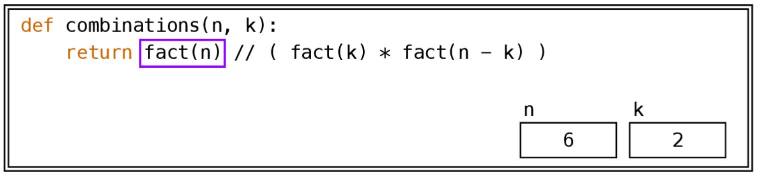
The second step is to create a frame for the `combinations` function that contains space for the variables that are stored as part of that frame, which are the parameters and any variables that appear in declarations within the function. The `combinations` function has two positional parameters and no local variables, so the frame only requires enough space for the parameter variables `n` and `k`. After the Python interpreter creates the frame, it copies the argument values into these variables in order. Thus, the parameter variable `n` is initialized to 6, and the parameter variable `k` is initialized to 2.

In the diagrams in this book, each stack frame appears as a rectangle surrounded by a double line. Each stack-frame diagram shows the code for the function along with a pointing-hand icon that makes it easy to keep track of the current execution point. The frame also contains labeled boxes for each of the local variables. The stack frame for the `combinations` function therefore looks like this after the parameters have been initialized but before execution of the function begins:



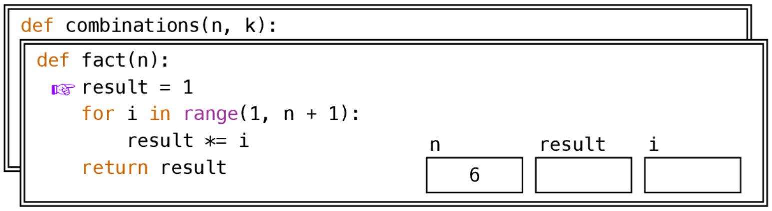
```
def combinations(n, k):
 ☞ return fact(n) // ( fact(k) * fact(n - k) )
```

To compute the value of the `combinations` function, the program must make three calls to the function `fact`. In Python, function calls are evaluated from left to right, so the first call is the one to `fact(n)`, as follows:



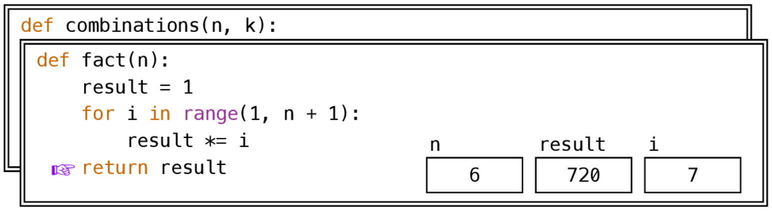
```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
```

To evaluate this function, the system must create yet another stack frame, this time for the function `fact` with an argument value of 6. The frame for `fact` has both parameters and local variables. The parameter `n` is initialized to the value of the calling argument and therefore has the value 6. The two local variables, `i` and `result`, have not yet been initialized, which is indicated in stack diagrams using an empty box. The new frame for `fact` gets stacked on top of the old one, which allows the Python interpreter to remember the values in the earlier stack frame, even though they are not currently visible. The situation after creating the new frame and initializing the parameters looks like this:



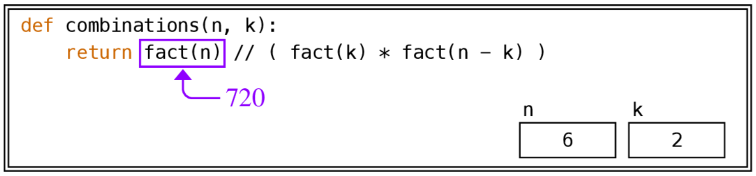
```
def combinations(n, k):
def fact(n):
  ☞ result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

The system then executes the statements in the function `fact`. In this instance, the body of the `for` loop is executed six times. On each cycle, the value of `result` is multiplied by the loop index `i`, which means that it will eventually hold the value 720 (1×2×3×4×5×6 or 6!). When the program reaches the `return` statement, the stack frame looks like this:



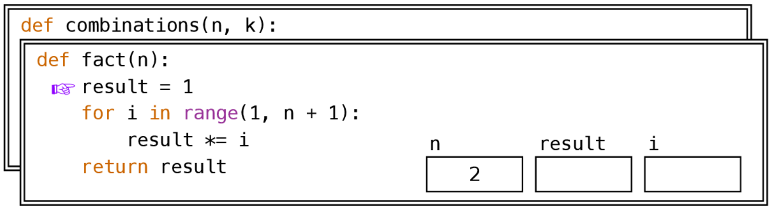
```
def combinations(n, k):
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
  ☞ return result
```

Returning from a function involves copying the value of the `return` expression (in this case the local variable `result`), to the point at which the call occurred. The frame for `fact` is then discarded, which leads to the following configuration:



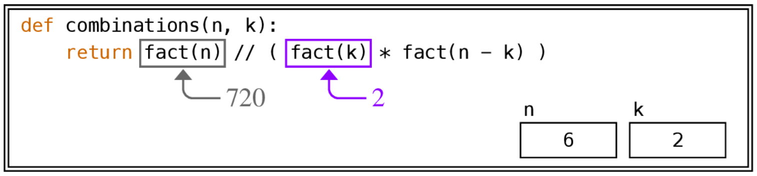
```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
```

The next step in the process is to make a second call to `fact`, this time with the argument `k`. In the calling frame, `k` has the value 2. That value is then used to initialize the parameter `n` in the new stack frame, as follows:



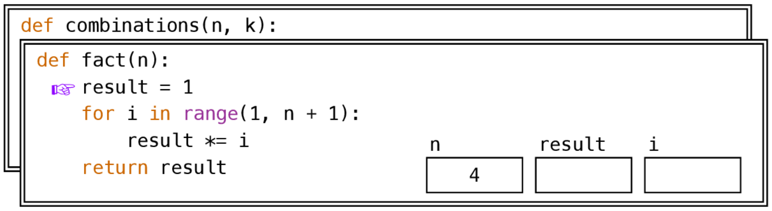
```
def combinations(n, k):
def fact(n):
  ☞ result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

The computation of `fact(2)` is easier to perform in one's head than the earlier call to `fact(6)`. This time around, the value of `result` will be 2, which is then returned to the calling frame, like this:



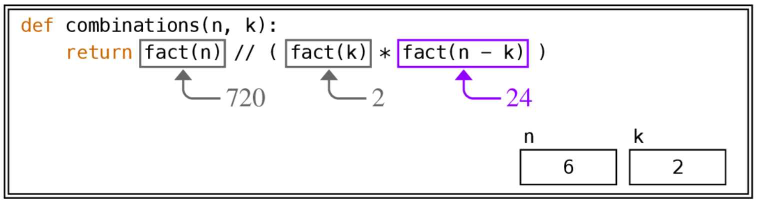
```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
```

The code for `combinations` makes one more call to `fact`, this time with the argument `n - k`. Evaluating this call therefore creates a new stack frame with `n` equal to 4:



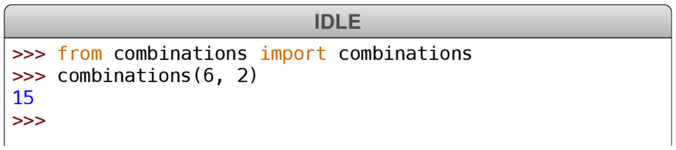
```
def combinations(n, k):
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

The value of `fact(4)` is 1×2×3×4, or 24. When this call returns, the system is able to fill in the last of the missing values in the calculation, as follows:



```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
```

The computer then divides 720 by the product of 2 and 24 to get the answer 15. This value is returned to the Python interpreter running in the IDLE console window. The interpreter prints that value on the console, like this:



```
>>> from combinations import combinations
>>> combinations(6, 2)
15
>>>
```

5.3 Libraries and interfaces

Writing a program to solve a large or difficult problem inevitably forces you to manage at least some amount of complexity. There are algorithms to design, special cases to consider, user requirements to meet, and innumerable details to get right. To make programming manageable, you must reduce the complexity of the programming process as much as possible. Functions reduce some of the complexity; libraries offer a similar reduction in programming complexity but at a higher level of detail. A function gives its caller access to a set of steps that implements a single operation. A library provides a collection of tools that share a common model. That model and its conceptual foundation constitute a ***programming abstraction.***

Clients and implementers

One of the goals of any programming abstraction is to hide the complexity involved in the underlying implementation. By exporting the `sqrt` function, the Python `math` library hides away the complexities involved in calculating a square root. When you call `math.sqrt`, you don't need to have any idea how the implementation works. Although it almost certainly uses a more modern algorithm that computes the result more quickly, the implementation might use the 3800-year-old Babylonian method described in Chapter 3. The caller doesn't need to know. The details of how the computation proceeds are relevant only to the programmers responsible for implementing the `math` library.

Knowing how to call the `math.sqrt` function and knowing how to implement it are both important skills. It is useful to keep in mind, however, that those two skills—calling a function and implementing one—are to a large extent independent. Successful programmers often use functions that they wouldn't have a clue how to write. Conversely, programmers who implement a library function can never anticipate all the potential uses for that function.

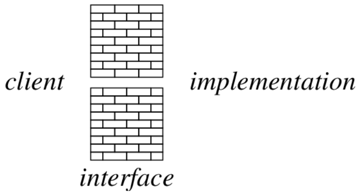
To emphasize the difference in perspective between programmers who implement a library and those who use it, computer scientists have assigned names to programmers working in each of these roles. Naturally enough, a programmer who implements a library is called an ***implementer.*** Conversely, a programmer who calls functions provided by a library is called a ***client*** of that library.

Both functions and libraries offer a tool for hiding lower-level implementation details so that clients need not worry about them. In computer science, this technique is called ***information hiding.*** The fundamental idea, championed by David Parnas in the early 1970s, is that the complexity of programming systems is best managed by making sure that details are visible only at those levels of the program at which they are relevant. For example, only the programmers who implement `math.sqrt` need to know the details of its operation. Clients who merely use `math.sqrt` can remain blissfully unaware of the underlying details.

The concept of an interface

In computer science, the understanding shared between a client and an implementer is called an ***interface.*** Conceptually, an interface contains the information that clients need to know about a library—and no more. For clients, getting too much information can be as bad as getting too little, because additional detail is likely to make the interface more difficult to understand. Often, the real value of an interface lies not in the information it *reveals* but rather in the information it *hides*.

When you design an interface for a library, you should try to protect the client from as many of the complicating details of the implementation as possible. In doing so, it is perhaps best to think of an interface not as a communication channel between the client and the implementation, but instead as a wall that divides them.



Like the wall that divided the lovers Pyramus and Thisbe in Greek mythology, the wall representing an interface contains an opening or chink that allows the two sides to communicate. In programming, that chink exposes the function definitions so that the client and implementation can share essential information. The main purpose of the wall, however, is to keep the two sides apart. Ideally, all the complexity involved in the realization of a library lies on the implementation side of the wall. An interface is successful if it supports the principle of information hiding by keeping as much complexity as possible away from the client side.

5.4 The random library

Before turning to the problem of creating new libraries, it makes sense to explore another Python library so that you have more examples than the `math` library you have already seen. The `random` library exports a set of functions that allow you to write programs that make seemingly random choices. Being able to simulate random behavior is necessary, for example, if you want to write a computer game that involves flipping a coin or rolling a die, but is also useful in more practical contexts. Programs that simulate random processes are said to be ***nondeterministic.***

As with the `math` library, you need to import the `random` library before you use it in a module. To do so, all you need is the statement

```
import random
```

at the beginning of your file. Including this statement gives you access to all the functions in the `random` library but requires you to refer to those functions using their fully qualified name, which includes the source module name and a dot, as in `random.randint`.

Functions in the random library

A subset of the functions exported by the `random` library appears in Figure 5-3. These functions are subdivided into categories depending on the type of value on which they operate. The first section offers a set of functions for working with random integers. The easiest function to use is `randint(`*min*`,` *max*`)`, which returns an integer between *min* and *max,* inclusive. You can, for example, use this function to generate a die roll like this:

```
    die = random.randint(1, 6)
```

Similarly, you could generate the outcome of spinning a European roulette wheel (unlike American roulette wheels, which have both a 0 and a 00 slot, European roulette wheels have slots numbered from 0 to 36) with the following statement:

```
    spin = random.randint(0, 36)
```

You can also generate random integers using the function `randrange`, which takes the same argument forms as the `range` function used in conjunction with the `for` loop. Thus, you could also simulate the die roll like this:

```
    die = random.randrange(1, 7)
```

FIGURE 5-3 Selected functions from the `random` library

Random integers

`randint(`*min*`,` *max*`)`	Returns a random integer between *min* and *max,* inclusive.
`randrange(`*limit*`)`	Returns a random integer from 0 up to but not including *limit.*
`randrange(`*start*`,` *limit*`)`	Returns a random integer from *start* up to but not including *limit.*

Random floating-point numbers

`random()`	Returns a random floating-point number in the range between 0 and 1.
`uniform(`*min*`,` *max*`)`	Returns a random floating-point number between *min* and *max.*

Random functions on lists and sequences

`choice(`*seq*`)`	Returns a random element from *seq,* which is any sequence.
`sample(`*seq*`,` *k*`)`	Returns a list with *k* elements randomly chosen from *seq.*
`shuffle(`*list*`)`	Rearranges the list in a random order.

Initialization functions

`seed()`	Randomizes the internal random number generator.
`seed(`*k*`)`	Sets the internal state of the random number generator so that it generates the same sequence for any specific value of the integer *k.*

Figure 5-3 includes two functions for generating random floating-point numbers, although the `random` library implements a much larger set of functions that are useful in statistical applications. The `random` function itself generates a number uniformly distributed over the range from 0 to 1. More generally, the function `uniform(`*min*`,` *max*`)` returns a floating-point value between *min* and *max.* For example, the function call

```
random.uniform(-1, 1)
```

generates a random floating-point number between –1 and 1.

You can use the `random` function to simulate random events that occur with some probability. The predicate function

```
def random_chance(p=0.5):
    return random.random() < p
```

returns `True` with probability `p`, where the argument is a statistical probability value between 0.0, which means that the event never happens, and 1.0, which means that the event always happens. If `p` is omitted, the `random_chance` function uses the default value 0.5, which signifies an event that happens with probability 0.5, or 50 percent of the time. Thus, you can simulate the process of flipping a coin using the following statements, which set the variable `flip` to `"Heads"` or `"Tails"` with equal probability:

```
if random_chance():
    flip = "Heads"
else:
    flip = "Tails"
```

The functions `random.choice`, `random.sample`, and `random.shuffle` are designed for use with lists, which are introduced in Chapter 8. As a preview of these coming attractions, you can use `random.choice` to choose a random character from a string. For example, you can use the following line to set the variable `letter` to a randomly chosen lowercase letter:

```
letter = random.choice("abcdefghijklmnopqrstuvwxyz")
```

Initializing the random number generator

The `seed` function in the last section of Figure 5-3 requires a little more explanation. Because computers are deterministic machines, random numbers are usually computed by going through a deterministic calculation that nonetheless appears random to the user. Random numbers computed in this way are called ***pseudorandom numbers.*** By default, modern versions of Python automatically call `seed` to initialize the generator, but it is still common practice to include an explicit call to `random.seed()` at the beginning of each program that uses the `random` library. The effect of this statement is to initialize the internal state of Python's random number generator to an unpredictable value based on the system clock. As Figure 5-3 shows, you can also call `seed` with an integer argument, which is used to set the internal state. If you initialize the random number generator to a particular value, it will always generate the same values every time the program is run.

At first, it may seem hard to understand why a random number package should return the same values on each run. After all, deterministic behavior of this sort seems to defeat the whole purpose of the package. There is, however, a good reason behind this behavior: programs that behave deterministically are easier to debug. To illustrate this fact, suppose you have just written a program to play an intricate game, such as Monopoly. As is always the case with newly written programs, the odds are good that your program has a few bugs. In a complex program, bugs can be relatively obscure, in the sense that they only occur in rare situations. Suppose you are playing the game and discover that the program is starting to behave in a bizarre way. As you begin to debug the program, it would be very convenient if you could regenerate the same state and take a closer look at what is going on. Unfortunately, if the program is running in a nondeterministic way, a second run of the program will behave differently from the first. Bugs that showed up the first time may not occur on the second pass.

In general, it is difficult to reproduce the conditions that cause a program to fail if the program is behaving in a truly random fashion. If, on the other hand, the program is operating deterministically, it will do the same thing each time. This behavior makes it possible for you to recreate the conditions under which the problem occurred. When you write a program that works with random numbers, it is usually best to call `random.seed` with an argument during the debugging phase. When the program seems to be working well, you can remove that argument to ensure that its behavior changes from one run to the next.

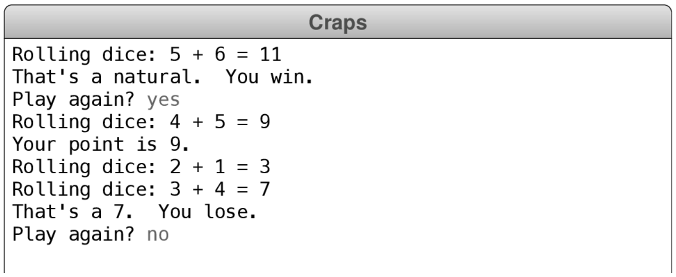
Using the random library

As an illustration of how clients might use the `random` library, the `Craps` program in Figure 5-4 plays the casino game called *craps.* The rules for craps appear in the comments at the beginning of the program. The code itself follows the outline imposed by the rules of the game. In particular, it rolls the dice initially and then chooses how to proceed according to the result of that first roll. Moreover, because the task of rolling two dice and determining their sum appears at different points in the program, it makes sense to make rolling two dice a separate function. The startup code at end of the `Craps.py` file runs the `Craps` function repeatedly, asking the user at the end of each game whether to play again.

FIGURE 5-4 A program to play the casino game of Craps

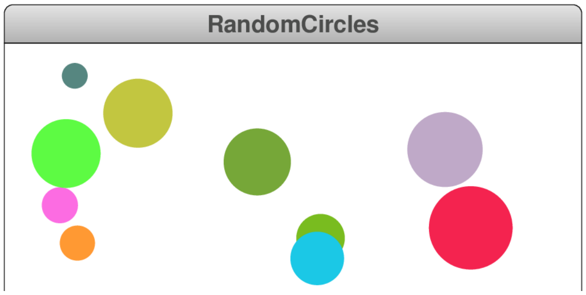
```
# File: Craps.py

import random

"""
This program plays the casino game of Craps.  At the beginning, the player
rolls a pair of dice and computes the total.  If the total is 2, 3, or 12
("craps"), the player loses.  If the total is 7 or 11 (a "natural"), the
player wins.  If the total is any other number, that number becomes the
"point."  The player then keeps rolling until the point comes up again, in
which case the player wins, or a 7 appears, in which case the player loses.
"""

def craps():
    """Plays Craps until the user quits."""
    finished = False
    while not finished:
        play_one_craps_game()
        finished = input("Play again? ") != "yes"

def play_one_craps_game():
    """Plays one craps game."""
    total = roll_two_dice()
    if total == 7 or total == 11:
        print("That's a natural.  You win.")
    elif total == 2 or total == 3 or total == 12:
        print("That's craps.  You lose.")
    else:
        point = total
        print("Your point is " + str(point) + ".")
        finished = False
        while not finished:
            total = roll_two_dice()
            if total == point:
                print("You made your point.  You win.")
                finished = True
            elif total == 7:
                print("That's a 7.  You lose.")
                finished = True

def roll_two_dice():
    """Rolls two dice, displays their values, and returns their sum."""
    d1 = random.randint(1, 6)
    d2 = random.randint(1, 6)
    total = d1 + d2
    print("Rolling dice:", d1, "+", d2, "=", total)
    return total

# Startup code

if __name__ == "__main__":
    craps()
```

Although the `Craps` function is nondeterministic and will therefore produce different results each time, the following console log shows two possible outcomes:

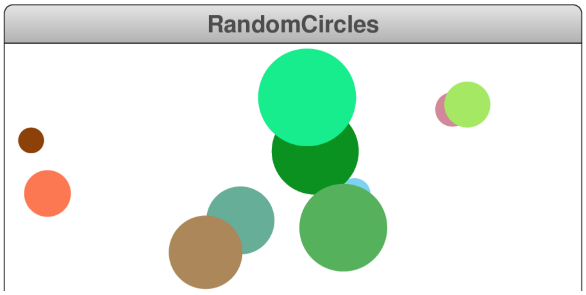


```
Rolling dice: 5 + 6 = 11
That's a natural.  You win.
Play again? yes
Rolling dice: 4 + 5 = 9
Your point is 9.
Rolling dice: 2 + 1 = 3
Rolling dice: 3 + 4 = 7
That's a 7.  You lose.
Play again? no
```

As a second example of a program that uses the `random` library, the `RandomCircles.py` program in Figure 5-5 on the next page displays circles of various random sizes, random colors, and random positions. The display will be different each time, but the code makes sure that the individual circles always fit inside the graphics window. A sample run of this program might look like this:



Because Python's random number generator produces different values each time, running the program again produces an image with different circles, as follows:



It is worth paying attention to the implementation of the `random_color` function in Figure 5-5, which uses the `random.choice` function to select one of the sixteen hexadecimal digits. The effect is to create a hexadecimal color value in the form "#*dddddd*", as described on page 106.

FIGURE 5-5 Program to display random circles on the screen

```
# File: RandomCircles.py

from pgl import GWindow, GOval
import random

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
N_CIRCLES = 10
MIN_RADIUS = 15
MAX_RADIUS = 50

def random_circles():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    for i in range(N_CIRCLES):
        gw.add(create_random_circle())

def create_random_circle():
    """
    Creates a randomly generated circle that fits in the window.
    """
    r = random.uniform(MIN_RADIUS, MAX_RADIUS)
    x = random.uniform(r, GWINDOW_WIDTH - r)
    y = random.uniform(r, GWINDOW_HEIGHT - r)
    return create_filled_circle(x, y, r, random_color())

def create_filled_circle(x, y, r, color):
    """
    Creates a circle of radius r centered at the point (x, y) filled
    with the specified color.
    """
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.set_color(color)
    circle.set_filled(True)
    return circle

def random_color():
    """
    Returns a random color expressed as a string consisting of a "#"
    followed by six random hexadecimal digits.
    """
    str = "#"
    for i in range(6):
        str += random.choice("0123456789ABCDEF")
    return str

# Startup code

if __name__ == "__main__":
    random_circles()
```

5.5 Creating your own libraries

One of the most important advantages of creating a library is that doing so allows you to reuse functions and definitions in new programs without having to copy the actual code. For example, you may have noticed that the RandomCircles program in Figure 5-5 included the function create_filled_circle, which first appeared in the Target.py program in Figure 4-14. If you discover that you are using a function several times in different applications, you should consider defining that function in a library module and then importing that module when you need it.

Python makes it extremely easy to import definitions from other modules. You could, for example, import create_filled_circle directly from the Target module by including the line

```
from Target import create_filled_circle
```

at the top of your program. Doing so, however, is likely to confuse clients who are not familiar with the Target application. In most cases, it makes sense to collect several similar functions together into a module that is used only as a library and not as an application. Client applications can then import the functions they need from a common source.

When you create a new library module from existing functions, it is usually wise to think about those functions carefully to see whether they meet the needs of as many clients as possible. While the create_filled_circle function does exactly what both the Target.py and RandomCircles.py programs need, other applications may want to control the color used for the border of the circle separately from the color used to fill it. If you decide that feature would be useful, you should redefine create_filled_circle so that it takes two arguments specifying color, one for the fill color and one for the border color, both of which are optional. If only one color argument appears, that argument should be used to set the color of the entire circle, just as it has in the existing applications. Clients that need to control the fill and border colors independently should two color names, ideally specified as keyword parameters under the names fill and border so that their order is unimportant.

The extended version of create_filled_circle appears in Figure 5-6 along with a similar function for rectangles called create_filled_rect and a useful function called create_centered_label that creates a GLabel centered at a specified point. Clients who need any of these functions can then import them from the gtools module. You can, moreover, go back and rewrite the Target.py and RandomCircles.py programs to import the create_filled_circle function from gtools because the new definition is compatible with the old one.

FIGURE 5-6 A library of simple graphical tools

```python
# File: gtools.py

"""
This module exports a library of three graphics functions
(create_filled_rect, create_filled_circle, and create_centered_label)
that are used in several different applications.  Each of these
functions is described in its docstring comment.
"""

from pgl import GRect, GOval, GLabel

def create_filled_rect(x, y, width, height, fill="Black", border=None):
    """
    Creates a GRect filled with the specified fill color.  If border is
    specified, the border appears in that color.
    """
    rect = GRect(x, y, width, height)
    rect.set_filled(True)
    if border is None:
        rect.set_color(fill)
    else:
        rect.set_color(border)
        rect.set_fill_color(fill)
    return rect

def create_filled_circle(x, y, r, fill="Black", border=None):
    """
    Creates a circle of radius r centered at the point (x, y) with the
    specified fill color.  If border is specified, the border appears
    in that color.
    """
    circle = GOval(x - r, y - r, 2 * r, 2 * r)
    circle.set_filled(True)
    if border is None:
        circle.set_color(fill)
    else:
        circle.set_color(border)
        circle.set_fill_color(fill)
    return circle

def create_centered_label(text, x, y, font=None):
    """
    Creates a new GLabel centered at the point (x, y) in both the
    horizontal and vertical directions.  If font is specified, it
    is used to set the font of the label.
    """
    label = GLabel(text)
    if font is not None:
        label.set_font(font)
    label.set_location(x - label.get_width() / 2, y + label.get_ascent() / 2)
    return label
```

The code in Figure 5-6 introduces one new feature of Python, which is important to understand if you are trying to write functions that take different argument patterns. To detect whether an argument is missing from the call, all three functions specify the Python constant None as a default value for the last parameter in the list. The new feature appears later in the function body when the code checks to see whether that argument was specified. Although the == and != operators also work for this purpose, the approved way to check for None is to use the Python operators is and is not, which test for exact identity rather than equality. The distinction is subtle and beyond the scope of this chapter, but you will have a chance later in this book to see examples where the differences between these operators matter.

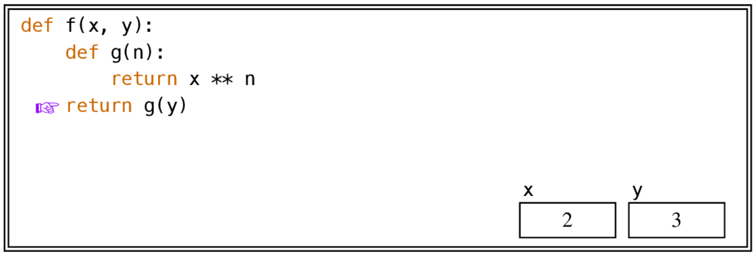
5.6 Inner functions

In Python, you can define one function inside another function simply by nesting the definitions just as you would nest any control structure. A function defined inside another one is called an ***inner function.*** The following admittedly artificial example defines a function g inside a function f:

```
def f(x, y):
    def g(n):
        return x ** n
    return g(y)
```

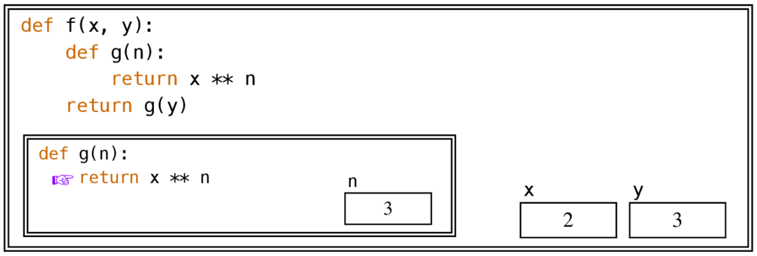
The function g is an inner function and is defined only inside the function f. It takes a single argument n and returns the value of the variable x raised to the nth power. Although that idea initially seems straightforward, it is important to ask yourself how the function g determines the value of x.

The answer is that x is defined in the enclosing context. The variable x is one of the parameters for the function f, which encloses the definition of g. If g cannot find a variable in its own collection of variables, it looks to see if that variable is defined in the enclosing function. When Python evaluates the expression x ** n, if find the value on n in its own stack frame and the value of x in the frame of the enclosing function f. Calling f(2, 3), for example, creates a frame for the function f, which looks like this when execution reaches the return statement:



```
def f(x, y):
    def g(n):
        return x ** n
 ☞  return g(y)
```

Executing the `return` statement requires Python to evaluate the function `g(y)` in the current frame. Since `g` is an inner function, its frame is conceptually nested inside the frame for `f` and has access to its variables. The frame diagram after invoking `g` therefore looks like this:



```
def f(x, y):
    def g(n):
        return x ** n
    return g(y)
```

```
def g(n):
☞return x ** n
```

Inside the frame for `g`, the variable `n` is in the local frame and has the value 3. The variable `x` appears in the enclosing frame and has the value 2. The result of the function call is therefore 2^3 or 8.

When Python needs to find the value of an identifier, it searches for the name in the following four contexts:

1. ***Local.*** The local context consists of all names defined within the current function. A name is defined in the function if it appears as a parameter, as the target of an assignment, as the index variable in a `for` loop, or as the name of a nested function definition.
2. ***Enclosing.*** The enclosing context consists of the names defined in a function that encloses the current one, as illustrated by the diagram showing the frame for `g` nested within the frame for `f`.
3. ***Global.*** The global context consists of names defined outside of any function or imported into the current module using the `from-import` statement.
4. ***Built-in.*** The last place that Python looks for a name is in the list of built-in functions like `abs`, `str`, and `print`.

Python searches each of these contexts in order, so that a local definition takes precedence over a definition in any of the other contexts. Names defined at one level can therefore hide names defined at lower levels of this hierarchy. For example, it is perfectly legal in Python to use `str` as the name of a local variable, but doing so means that it is impossible to call the built-in function `str` inside the function that defines the local variable. Hiding an existing identifier by defining its name in a more local context is called ***shadowing.***

The region of a program in which an identifier is defined is called its ***scope.*** The rules for determining the scope of an identifier in Python are a little more complicated than this section suggests, but these rules are sufficient in most cases.

One of the advantages of defining nested functions is that doing so allows a set of related functions to share local data without having to pass all of the information as arguments. For example, the `DrawHouse.py` program in Figure 4-11 can be simplified by defining the functions `draw_house_at`, `draw_frame`, `draw_door`, and `draw_window` as inner functions nested within the `draw_house` definition. Doing so makes it unnecessary to pass `gw` as an explicit parameter to each of these functions because those functions can see this variable in the enclosing scope. You will have a chance to implement this strategy in exercise 6.

5.7 Introduction to recursion

Most algorithmic strategies used to solve programming problems have counterparts outside the domain of computing. When you perform a task repeatedly, you are using iteration. When you make a decision, you exercise conditional control. Because these operations are familiar, most people learn to use the control statements `for`, `while`, and `if` with relatively little trouble.

Before you can solve certain more sophisticated programming tasks, you will need to master a powerful problem-solving strategy that has few direct counterparts in the real world. That strategy, called ***recursion,*** is defined as any solution technique in which large problems are solved by reducing them to smaller problems *of the same form.* The italicized phrase is crucial to the definition, which otherwise describes the basic strategy of stepwise refinement. Both strategies involve decomposition. What makes recursion special is that the subproblems in a recursive solution have the same form as the original problem.

If you are like most beginning programmers, the idea of breaking a problem down into subproblems of the same form does not make much sense when you first hear it. Unlike repetition or conditional testing, recursion is not a concept that comes up in day-to-day life. Because it is unfamiliar, learning how to use recursion can be difficult. To do so, you must develop the intuition necessary to make recursion seem as natural as all the other control structures. For most students of programming, reaching that level of understanding takes considerable time and practice. Even so, learning to use recursion is definitely worth the effort. As a problem-solving tool, recursion is so powerful that it at times seems almost magical. In addition, using recursion often makes it possible to write complex programs in simple and profoundly elegant ways.

A simple example of recursion

To gain a better sense of what recursion is, let’s imagine that you have been appointed as the funding coordinator for a political campaign, which is long on volunteers but short on cash. Your job is to raise $1,000,000 in contributions.

If you know someone who is willing to write a check for the entire $1,000,000, your job is easy. On the other hand, you may not be lucky enough to have friends who are generous millionaires. In that case, you must raise the $1,000,000 in smaller amounts. If the average contribution is $100, you might choose a different tack: call 10,000 friends and ask each of them for $100. But then again, you probably don't have 10,000 friends. So what can you do?

As is often the case when you are faced with a task that exceeds your own capacity, the answer lies in delegating part of the work to others. Your organization has a ready supply of volunteers. If you could find 10 dedicated supporters in different parts of the country and appoint them as regional coordinators, each of those 10 people could then take responsibility for raising $100,000.

Raising $100,000 is simpler than raising $1,000,000, but it hardly qualifies as easy. What should your regional coordinators do? If they adopt the same strategy, they will in turn delegate parts of the job. If they each recruit 10 fundraising volunteers, those people will only have to raise $10,000 each. The delegation process can continue until the volunteers are able to raise the money on their own; because the average contribution is $100, the volunteer fundraisers can probably raise $100 from a single donor, which eliminates the need for further delegation. If you express this fundraising strategy in pseudocode, it has the following structure:

```
def collect_contributions(n):
    if n <= 100:
        Collect the money from a single donor.
    else:
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
```

The most important thing to notice about this pseudocode function is that the line

```
Get each volunteer to collect n/10 dollars.
```

is simply the original problem reproduced at a smaller scale. The basic character of the task—raise *n* dollars—remains exactly the same; the only difference is that *n* has a smaller value. Moreover, because the problem is the same, you can solve it by calling the original function. Thus, the preceding line of pseudocode would eventually be replaced with the following line:

```
collect_contributions(n / 10)
```

It's important to note that the `collect_contributions` function ends up calling itself if the contribution level is greater than $100. In the context of programming, having a function call itself is the defining characteristic of recursion.

The structure of the `collect_contributions` function is typical of recursive functions. In general, the body of a recursive function has the following form:

```
if test for simple case:
    Compute a simple solution without using recursion.
else:
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the subproblem solutions into a solution for the whole.
```

This structure provides a template for writing recursive functions and is therefore called the ***recursive paradigm.*** You can apply this technique to programming problems as long as they meet the following conditions:

1. You must be able to identify ***simple cases*** for which the answer is easily determined.
2. You must be able to identify a ***recursive decomposition*** that lets you break any complex instance of the problem into simpler problems of the same form.

The `collect_contributions` example illustrates the power of recursion. As with any recursive technique, the original problem is solved by breaking it down into smaller subproblems that differ from the original only in their scale. Here, the original problem is to raise $1,000,000. At the first level of decomposition, each subproblem is to raise $100,000. These problems are then subdivided to create smaller problems until the problems are simple enough to be solved immediately without recourse to further subdivision.

A recursive formulation of the factorial function

The `combinations` module in Figure 5-2 includes a simple implementation of a function to compute factorials, which looks like this:

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

This implementation uses a `for` loop to cycle through the integers between 1 and `n`. Strategies based on looping are said to be ***iterative.***

You can, however, also implement the `fact` function recursively by taking advantage of an important mathematical property of factorials. Each factorial is related to the factorial of the next smaller integer in the following way:

$$n! = n \times (n - 1)!$$

Thus, 4! is 4 × 3!, 3! is 3 × 2!, and so on. To make sure that this process stops at some point, mathematicians define 0! to be 1. Thus, the conventional mathematical definition of the factorial function looks like this:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

This definition is recursive, because it defines the factorial of *n* in terms of a simpler instance of the factorial function: finding the factorial of *n* – 1. The new problem has the same form as the original, which is the fundamental characteristic of recursion. You can then use the same process to define (*n* – 1)! in terms of (*n* – 2)!. Moreover, you can carry this process forward step by step until the solution is expressed in terms of 0!, which is equal to 1 by definition.

From your perspective as a programmer, the most important consequence of the definition from mathematics is that it provides a template for a recursive solution. In Python, you can implement a function `fact` that computes the factorial of its argument as follows:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

If `n` is 0, the result of `fact` is 1. If not, the implementation computes the result by calling `fact(n - 1)` and then multiplying the result by `n`. This implementation follows directly from the mathematical definition of the factorial function and has precisely the same recursive structure.

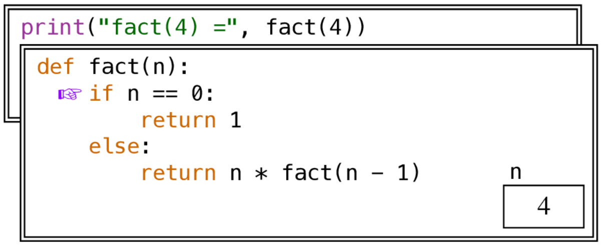
Tracing the recursive process

If you work from the mathematical definition, writing the recursive implementation of `fact` is straightforward. On the other hand, even though the definition is easy to write, the brevity of the solution may seem suspicious. When you are learning about recursion for the first time, the recursive implementation of `fact` seems to leave something out. Even though it clearly reflects the mathematical definition, the recursive formulation makes it hard to identify where the actual computational steps occur. When you call `fact`, for example, you want the computer to give you the answer. In the recursive implementation, all you see is a formula that transforms one call to `fact` into another one. Because the steps in that calculation are not explicit, it seems somewhat magical when the computer gets the right answer. If you trace through the logic the computer uses to evaluate any function call, however, you discover that no magic is involved. When the computer evaluates a call to the recursive `fact` function, it goes through the same process it uses to evaluate any other function call.

To visualize the process, suppose that you have executed the statement

```
print("fact(4) =", fact(4))
```

in the IDLE interpreter. When this statement calls `fact`, Python creates a new stack frame and copies the argument value into the formal parameter `n`. The frame for `fact` temporarily supersedes the frame executing the `print` call as shown in the following diagram:

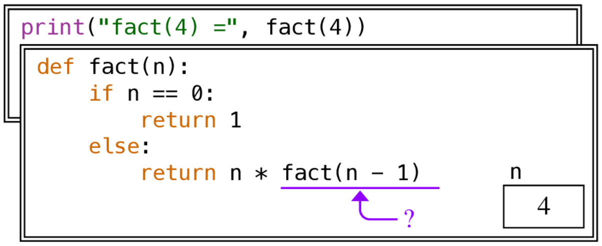


```
print("fact(4) =", fact(4))
def fact(n):
  ☞ if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

The computer now begins to evaluate the body of the function, starting with the `if` statement. Because `n` is not equal to 0, control proceeds to the `else` clause, where the program must evaluate and return the value of the expression

```
n * fact(n - 1)
```

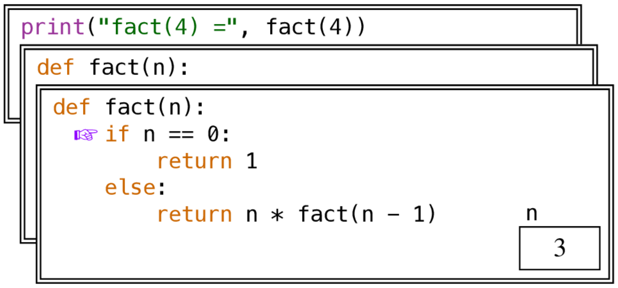
Evaluating this expression requires computing the value of `fact(n - 1)`, which introduces a recursive call. When that call returns, all the program has to do is to multiply the result by `n`. The current state of the computation can therefore be diagrammed as follows:



```
print("fact(4) =", fact(4))
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

As soon as the call to `fact(n - 1)` returns, the result is substituted for the expression underlined in the diagram, which allows computation to proceed.

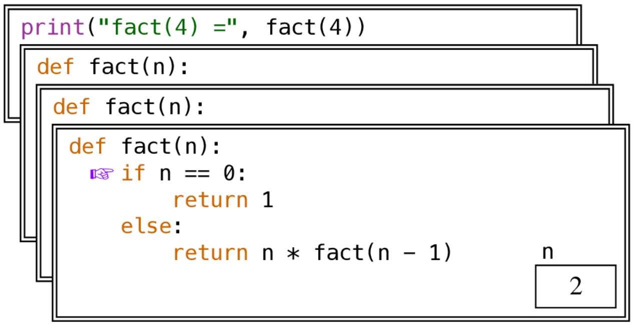
The next step in the computation is to evaluate the call to `fact(n - 1)`, beginning with the argument expression. Because the current value of `n` is 4, the argument expression n – 1 has the value 3. The computer then creates a new frame for fact in which n is initialized to this value. Thus, the next frame looks like this:



```
print("fact(4) =", fact(4))
 def fact(n):
  def fact(n):
   ☞ if n == 0:
         return 1
     else:
         return n * fact(n - 1)        n
                                       3
```

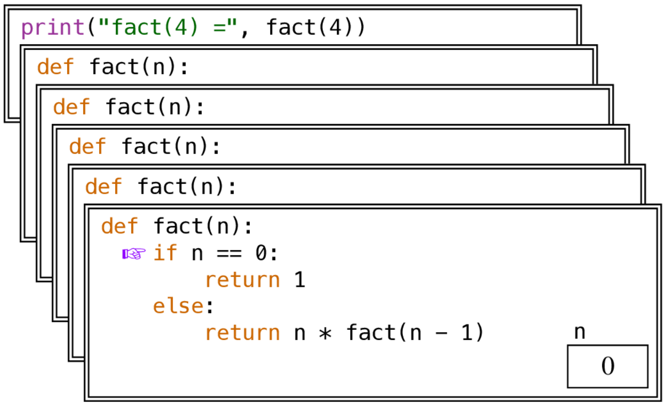
There are now two frames labeled fact. In the most recent one, the computer is just starting to calculate fact(3). This new frame hides the previous frame for fact(4), which will not reappear until the fact(3) computation is complete.

Computing fact(3) again begins by testing the value of n. Since n is still not 0, the else clause evaluates fact(n – 1), which creates another stack frame:



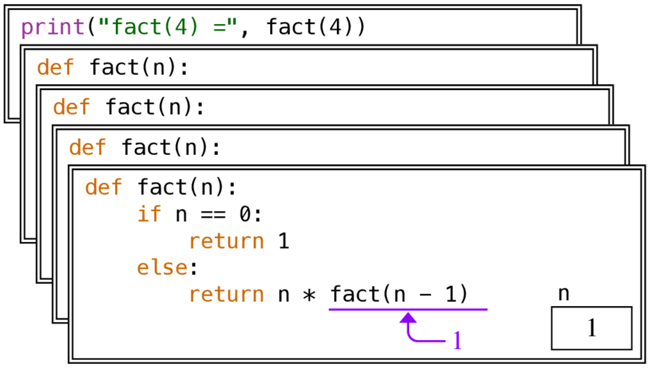
```
print("fact(4) =", fact(4))
 def fact(n):
  def fact(n):
   def fact(n):
    ☞ if n == 0:
          return 1
      else:
          return n * fact(n - 1)        n
                                        2
```

Following the same logic, the program must now call fact(1), which in turn calls fact(0), creating two new stack frames, as follows:



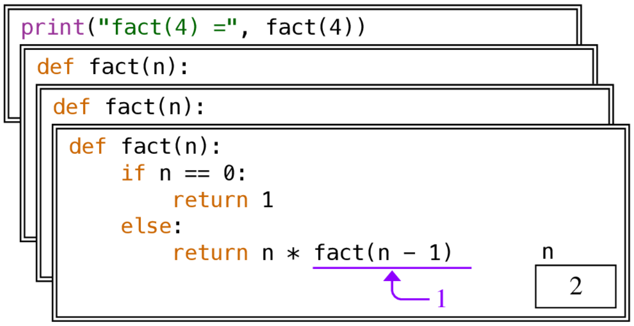
```
print("fact(4) =", fact(4))
 def fact(n):
  def fact(n):
   def fact(n):
    def fact(n):
     def fact(n):
      ☞ if n == 0:
            return 1
        else:
            return n * fact(n - 1)        n
                                          0
```

At this point, however, the situation changes. Because the value of n is 0, the function can return its result immediately. The value 1 is returned to the calling frame, which resumes its position on top of the stack, as shown:



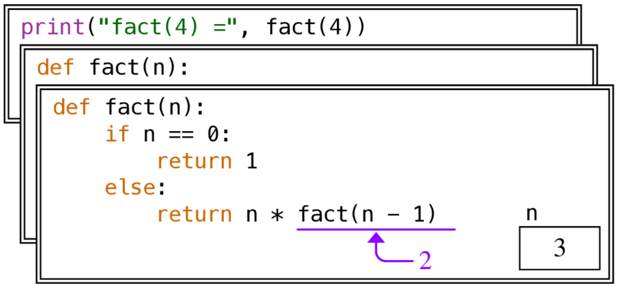
```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
def fact(n):
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

From this point, the computation proceeds back through each of the recursive calls, completing the calculation of the return value at each level. In this frame, for example, the call to fact(n - 1) can be replaced by the value 1, as shown in the stack frame. The code then computes the result by multiplying the current value of n by 1 and then returns that result to its caller, like this:



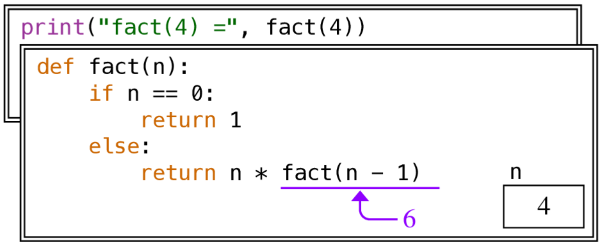
```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Because n is now 2, evaluating the return statement causes the value 2 to be passed back to the previous level, as follows:



```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

At this stage, the program returns 3 × 2 to the previous level, so that the frame for the initial call to `fact` looks like this:



```
print("fact(4) =", fact(4))
```

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

The final step in the calculation process consists of calculating 4 × 6 and returning the value 24 to the call to `print` in the IDLE interpreter.

The recursive leap of faith

The point of including the complete trace of the `fact(4)` computation is to convince you that the computer treats recursive functions just like all other functions. When you are faced with a recursive function, you can—at least in theory—mimic the operation of the computer and figure out what it will do. By drawing all the frames and keeping track of all the variables, you can duplicate the entire operation and come up with the answer. If you do so, however, you will usually find that the complexity of the process ends up making the computation much harder to follow.

Whenever you try to understand a recursive program, it is useful to put the underlying details aside and focus instead on a single level of the operation. At that level, you are allowed to assume that any recursive call automatically gets the right answer as long as the arguments to that call are in some sense simpler than the original arguments. This psychological strategy—assuming that any simpler recursive call will work correctly—is called the ***recursive leap of faith.*** Learning to apply this strategy is essential to using recursion in practical applications.

As an example, consider what happens when this implementation is used to compute `fact(n)` with `n` equal to 4. To do so, the recursive implementation must compute the value of the expression

```
n * fact(n - 1)
```

By substituting the current value of `n` into the expression, you know that the result is

```
4 * fact(3)
```

Stop right there. Computing `fact(3)` is simpler than computing `fact(4)`. Because it is simpler, the recursive leap of faith allows you to assume that it works. Thus, you should assume that the call to `fact(3)` correctly computes the value of 3!, which is 3 × 2 × 1, or 6. The result of calling `fact(4)` is therefore 4 × 6, or 24.

Whenever you try to understand a recursive function, it is better to focus on the big picture instead of the details. Once you have made the recursive decomposition and identified the simple cases, be satisfied that the computer can handle the rest.

The Fibonacci function

In a mathematical treatise entitled *Liber Abbaci* published in 1202, the Italian mathematician Leonardo Fibonacci proposed a problem that has had a wide influence on many fields, including computer science. The problem was phrased as an exercise in population biology—a field that has become increasingly important in recent years. Fibonacci's problem concerns how the population of rabbits would grow from generation to generation if the rabbits reproduced according to the following, admittedly fanciful, rules:

- Each pair of fertile rabbits produces a new pair of offspring each month.
- Rabbits become fertile in their second month of life.
- Old rabbits never die.

If a pair of newborn rabbits is introduced in January, how many pairs of rabbits are there at the end of the year?

You can solve Fibonacci's problem by keeping a count of the rabbits at each month during the year. At the beginning of January, there are no rabbits, since the first pair is introduced sometime in that month, which leaves one pair of rabbits on February 1st. Because the initial pair of rabbits is newborn, they are not yet fertile in February, which means that the only rabbits on March 1st are the original pair of rabbits. In March, however, the original pair is now of reproductive age, which means that a new pair of rabbits is born. The new pair increases the colony's population—counting by pairs—to two on April 1st. In April, the original pair goes right on reproducing, but the rabbits born in March are as yet too young. Thus, there are three pairs of rabbits at the beginning of May. From here on, with more rabbits becoming fertile each month, the rabbit population begins to explode.

At this point, it is useful to record the population data so far as a sequence of terms, indicated by the subscripted value t_i, each of which shows the number of rabbit pairs at the beginning of the i^{th} month from the start of the experiment on January 1st. The sequence itself is called the ***Fibonacci sequence*** and begins with the following terms, which represent the results of our calculation so far:

t_0	t_1	t_2	t_3	t_4
0	1	1	2	3

You can simplify the computation of further terms in this sequence by making an important observation. Because in this problem pairs of rabbits never die, all the rabbits that were around in the previous month are still around. Moreover, every pair of fertile rabbits has produced a new pair. The number of fertile rabbit pairs capable of reproduction is simply the number of rabbits that were alive in the month before the previous one. The net effect is that each new term in the sequence must simply be the sum of the preceding two. Thus, the next several terms in the Fibonacci sequence look like this:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
0	1	1	2	3	5	8	13	21	34	55	89	144

The number of rabbit pairs at the end of the year is therefore 144.

From a programming perspective, it helps to express the rule for generating new terms in the following more mathematical form:

$$t_n = t_{n\text{-}1} + t_{n\text{-}2}$$

An expression of this type, in which each element of a sequence is defined in terms of earlier elements, is called a ***recurrence relation.***

The recurrence relation alone is not sufficient to define the Fibonacci sequence. Although the formula makes it easy to calculate new terms in the sequence, the process has to start somewhere. In order to apply the formula, you need to have at least two terms already available, which means that the first two terms in the sequence—t_0 and t_1—must be defined explicitly. The complete specification of the terms in the Fibonacci sequence is therefore

$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ t_{n\text{-}1} + t_{n\text{-}2} & \text{otherwise} \end{cases}$$

This mathematical formulation is an ideal model for a recursive implementation of a function `fib(n)` that computes the n^{th} term in the Fibonacci sequence. All you need to do is plug the simple cases and the recurrence relation into the standard recursive paradigm. The recursive implementation of `fib(n)` looks like this:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

Now that you have a recursive implementation of the function `fib`, how can you go about convincing yourself that it works? You can always begin by tracing through the logic. Consider, for example, what happens if you call `fib(5)`. Because this is not one of the simple cases enumerated in the `if` statement, the implementation computes the result by evaluating the line

```
return fib(n - 1) + fib(n - 2)
```

which in this case is equivalent to

```
return fib(4) + fib(3)
```

At this point, the computer calculates the result of `fib(4)`, adds that to the result of calling `fib(3)`, and returns the sum as the value of `fib(5)`.

But how does the computer evaluate `fib(4)` and `fib(3)`? The answer, of course, is that it uses precisely the same strategy it did to calculate `fib(5)`. The essence of recursion is to break problems down into simpler ones that can be solved by calls to exactly the same function. Those calls get broken down into simpler ones, which in turn get broken down into even simpler ones, until at last the simple cases are reached.

Although you could certainly work through the necessary steps, it is best to regard this entire mechanism as irrelevant detail. Instead, all you need to do is remember the recursive leap of faith. Your job at this level is to understand how the call to `fib(5)` works. In the course of walking though the execution of that function, you have managed to transform the problem into computing the sum of `fib(4)` and `fib(3)`. Because the argument values are smaller, each of these calls represents a simpler case. Applying the recursive leap of faith, you can assume that the program correctly computes each of these values, without going through all the steps yourself.

Summary

In this chapter, you learned about *functions*, which enable you to refer to an entire set of operations with a single name. More importantly, by allowing the programmer to ignore the internal details and concentrate only on the effect of a function as a whole, functions provide a critical tool for reducing the conceptual complexity of programs.

The important points introduced in this chapter include:

- A *function* consists of a set of program statements that have been collected together and given a name. Other parts of the program can then *call* that function, possibly passing it information in the form of *arguments* and receiving a result *returned* by that function.
- A function that returns a value must have a `return` statement that specifies the result. Functions may return values of any type.

- Variables declared within a function are local to that function and cannot be used outside it. Internally, all the variables declared within a function are stored together in a *stack frame*.
- *Parameters* are local variables that act as placeholders for the argument values.
- Parameters in Python come in two types: *positional* and *keyword.* Python initializes positional parameter variables by copying the argument values in the order in which they appear. Arguments specified by keyword are copied into the parameter variable with the same name.
- Python allows a function to specify a default values for each parameter, which is used if the caller fails to supply a value.
- If any parameters remain uninitialized after positional, keyword, and default processing is complete, Python reports an error.
- When a function returns, it continues from precisely the point at which the call was made. Computer scientists refer to this point as the *return address.*
- You can create your own libraries by collecting the necessary code in a module whose name ends with the standard `.py` file type. You can then use the entries exported by this library by importing them into your application.
- In understanding the concept of a library, it is useful to differentiate the roles of the *client,* who uses the library, and the *implementer,* who writes the necessary code. The shared understanding between the client and the implementer is called the *interface.*
- Figure 5-3 lists some of the functions available in Python's `random` library. These functions enable you to write applications that simulate random behavior.
- In Python, function definitions can be nested inside other functions. The nested functions are called *inner functions.*
- An inner function has access to the local variables in the function that encloses it.
- Python determines the value associated with an identifier by looking at the following contexts in order: local, enclosing, global, and built-in.
- The portion of a program in which an identifier is defined is called its *scope.*
- Python's implementation of function calls makes it possible for a function to call itself, because the local variables for each call are stored in different stack frames. Functions that call themselves are said to be *recursive.*
- Before you can use recursion effectively, you must learn to limit your analysis to a single level of the recursive decomposition and to rely on the correctness of all simpler recursive calls without tracing through the entire computation. Trusting these simpler calls to work correctly is often called the *recursive leap of faith.*

Review questions

1. Define the following terms as they apply to functions: *call*, *argument*, *return*.

2. How do you specify the result of a function in Python?

3. Can there be more than one `return` statement in the body of a function?

4. Variables declared within a function are called *local variables.* What is the significance of the word *local* in this context?

5. What is a *stack frame?*

6. What do computer scientists mean by the term *return address?*

7. In your own words, describe the process by which Python uses the arguments in a function call to initialize the parameters variables. Be sure that your explanation covers positional, keyword, and default parameters

8. Describe the differences between the roles of *client* and *implementer.*

9. What is an *interface?*

10. How would you use the `random.randint` function to generate a randomly chosen integer between 1 and 100? How would you accomplish the same result using `random.randrange`?

11. If you run the `RandomCircles.py` program shown in Figure 5-5, you expect to see 10 circles on the graphics window because `N_CIRCLES` has the value 10. In fact, you sometimes see fewer circles. Why might this be?

12. What are the three functions exported by the `gtools` library in Figure 5-6?

13. What is an *inner function?*

14. True or false: An inner function has access to the local variables in the enclosing function.

15. What is meant by the term *scope* as it applies to variable names?

16. List the order in which Python searches contexts for the value on an identifier.

17. Describe the difference between the strategies of *iteration* and *recursion.*

18. What is meant by the phrase *recursive leap of faith?* Why is this concept important for you as a programmer?

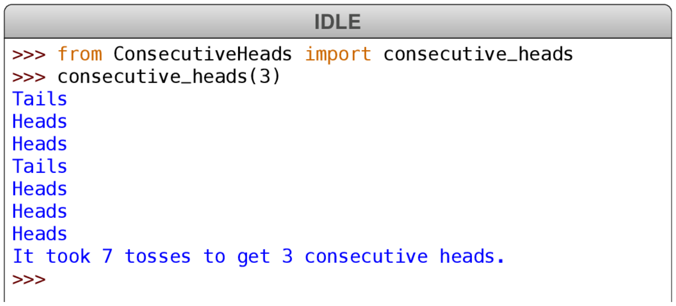
19. In the section entitled “Tracing the recursive process,” the text goes through a long analysis of what happens internally when `fact(4)` is called. Using this section as a model, trace the execution of `fib(3)`, sketching out each stack frame created in the process.

Exercises

1. Write a function `random_average(n)` that generates `n` random real numbers between 0 and 1 and then returns the average of those `n` values. Statistically, calling `random_average(n)` will produce results that become closer to 0.5 as the value of `n` increases. Write a main program that displays the result of calling `random_average` on 1, 10, 100, 1000, 10000, 100000, and 1000000.

2. *Heads. . . .*
Heads. . . .
Heads. . . .
A weaker man might be moved to re-examine his faith, if in
nothing else at least in the law of probability.
—Tom Stoppard, *Rosencrantz and Guildenstern Are Dead,* 1967

Write a function `consecutive_heads(number_needed)` that simulates tossing a coin repeatedly until the specified number of heads appear consecutively. At that point, your program should display a line on the console that indicates how many coin tosses were needed to complete the process. The following console log shows one possible execution of the program:



```
>>> from ConsecutiveHeads import consecutive_heads
>>> consecutive_heads(3)
Tails
Heads
Heads
Tails
Heads
Heads
Heads
It took 7 tosses to get 3 consecutive heads.
>>>
```

3. *I shall never believe that God plays dice with the world.*
—Albert Einstein, 1947

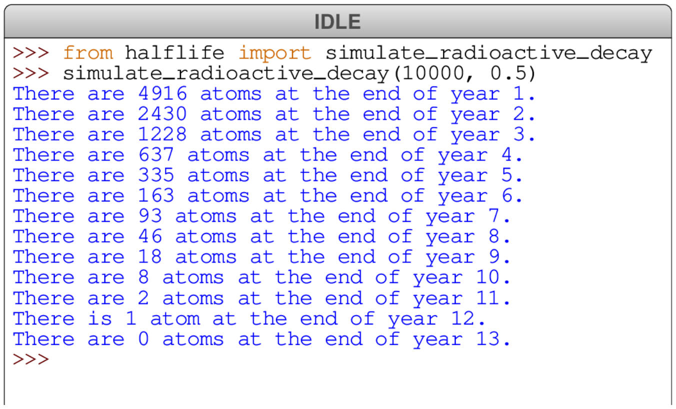
Despite Einstein’s metaphysical objections, the current models of physics, and particularly of quantum theory, strongly suggest that nature does indeed involve random processes. A radioactive atom, for example, does not decay for any specific reason that we mortals understand. Instead, that atom has a probability of decaying randomly within a particular period of time.

Because physicists consider radioactive decay a random process, it is not surprising that random numbers can be used to simulate it. Suppose you start with a collection of atoms, each of which has a certain probability of decaying in any unit of time. You can then approximate the decay process by taking each atom in turn and deciding randomly whether it decays.

Write a function `simulate_radioactive_decay` that models the process of radioactive decay. The first parameter is the initial population of atoms; the second is the probability that any of those atoms will decay within a year. For example, calling

```
simulate_radioactive_decay(10000, 0.5)
```

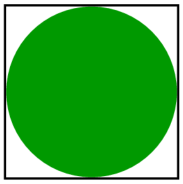
simulates what happens over time to a sample that contains 10,000 atoms of some radioactive material, where each atom has a 50 percent chance of decaying in a year. Your function should produce a trace on the console showing how many atoms remain at the end of each year until all of the atoms have decayed. For example, the output of your function might look like this:



```
>>> from halflife import simulate_radioactive_decay
>>> simulate_radioactive_decay(10000, 0.5)
There are 4916 atoms at the end of year 1.
There are 2430 atoms at the end of year 2.
There are 1228 atoms at the end of year 3.
There are 637 atoms at the end of year 4.
There are 335 atoms at the end of year 5.
There are 163 atoms at the end of year 6.
There are 93 atoms at the end of year 7.
There are 46 atoms at the end of year 8.
There are 18 atoms at the end of year 9.
There are 8 atoms at the end of year 10.
There are 2 atoms at the end of year 11.
There is 1 atom at the end of year 12.
There are 0 atoms at the end of year 13.
>>>
```

As the numbers indicate, roughly half the atoms in the sample decay each year. In physics, the conventional way to express this observation is to say that the sample has a ***half-life*** of one year.

4. Random numbers offer an interesting strategy for approximating the value of π. Imagine that you have a green dartboard hanging on your wall that consists of a circle painted on a square backdrop, as in the following diagram:



What happens if you throw a sequence of darts completely randomly, ignoring any darts that miss the board altogether? Some of the darts will fall inside the green circle, but some will be outside the circle in the white corners of the square. If the throws are random, the ratio of the number of darts landing inside the circle to the total number of darts hitting the square should be approximately equal to the ratio between the two areas. The ratio of the areas is independent of the actual size of the dartboard, as illustrated by the formula

$$\frac{\textit{darts falling inside the circle}}{\textit{darts falling inside the square}} \cong \frac{\textit{area inside the circle}}{\textit{area inside the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

To simulate this process in a program, imagine that the dartboard is drawn on the standard Cartesian coordinate plane with its center at the origin and a radius of 1 unit. The process of throwing a dart randomly at the square can be modeled by generating two random numbers, x and y, each of which lies between −1 and +1. This (x, y) point always lies somewhere inside the square. The point (x, y) lies inside the circle if

$$\sqrt{x^2 + y^2} < 1$$

This condition, however, can be simplified considerably by squaring each side of the inequality, which yields the following more efficient test:

$$x^2 + y^2 < 1$$

If you perform this simulation many times and compute what fraction of the darts falls inside the circle, the result will be an approximation of π/4.

Write a program that simulates throwing 10,000 darts and then uses the results to display an approximate value of π. Don't worry if your answer is correct only in the first few digits. The strategy used in this problem is not particularly accurate, even though it often provides useful approximations. In mathematics, this technique is called ***Monte Carlo integration,*** after the capital city of Monaco, famous for its casinos.

5. The combinations function $C(n, k)$ determines the number of ways you can choose k values from a set of n elements, ignoring the order of the elements. If the order of the value matters—so that, in the case of the coin example, choosing a penny and then a dime is seen as distinct from choosing a dime and then a penny—you need to use a different function, which computes the number of ***permutations,*** which are all the ways of ordering k elements taken from a collection of size n. This function is denoted as $P(n, k)$, and has the following mathematical formulation:

$$P(n, k) = \frac{n!}{(n - k)!}$$

Although this definition is mathematically correct, it is not well suited to implementation in practice because the factorials involved quickly get very large. For example, if you use this formula to calculate the number of ways to select two cards from a standard 52-card deck (assuming that the order matters), you would end up trying to evaluate the following fraction:

$$\frac{80{,}658{,}175{,}170{,}943{,}878{,}571{,}660{,}636{,}856{,}403{,}766{,}975{,}289{,}505{,}440{,}883{,}277{,}824{,}000{,}000{,}000{,}000}{30{,}414{,}093{,}201{,}713{,}378{,}043{,}612{,}608{,}166{,}064{,}768{,}844{,}377{,}641{,}568{,}960{,}512{,}000{,}000{,}000{,}000}$$

even though the answer is the much more manageable 2652 (52 × 51).

Write a function `permutations(n, k)` that computes the $P(n, k)$ function without calling the `fact` function. Part of your job in this problem is to figure out how to compute this value efficiently. To do so, you will probably find it useful to play around with some relatively small values to get a sense of how the factorials in the numerator and denominator of the formula behave.

6. Rewrite the `DrawHouse.py` program from Figure 4-11 so that the helper functions are defined as inner functions that share access to the `gw` variable.

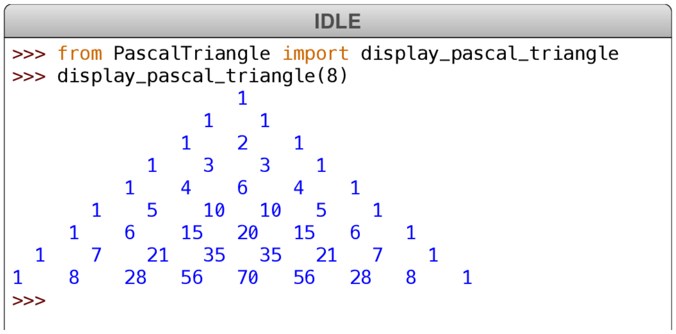
7. Create a library file called `alignment.py` that exports the `align_right` function from Chapter 1 along with the similar functions `align_left` and `align_center` that perform left and center alignment, respectively. In writing `align_center`, you will have to make some decision as to where it add the extra space if the number of spaces required is odd. The comments associated with the function should document your decision.

8. The values of the combinations function $C(n, k)$ described in this chapter are often displayed using a triangular arrangement that begins

$$\begin{array}{c} C(0,0) \\ C(1,0) \quad C(1,1) \\ C(2,0) \quad C(2,1) \quad C(2,2) \\ C(3,0) \quad C(3,1) \quad C(3,2) \quad C(3,3) \\ C(4,0) \quad C(4,1) \quad C(4,2) \quad C(4,3) \quad C(4,4) \end{array}$$

and then continues for as many rows as desired. This figure is called ***Pascal's Triangle*** after its inventor, the seventeenth-century French mathematician Blaise Pascal. Pascal's Triangle has the interesting property that every interior entry is the sum of the two entries above it.

Write a function `display_pascal_triangle(n)` that displays Pascal's Triangle from row 0 up to row `n`, as shown in the following IDLE session:



```
>>> from PascalTriangle import display_pascal_triangle
>>> display_pascal_triangle(8)
                    1
                 1    1
               1    2    1
            1    3    3    1
          1    4    6    4    1
       1    5    10   10   5    1
     1    6    15   20   15   6    1
  1    7    21   35   35   21   7    1
1    8    28   56   70   56   28   8    1
>>>
```

The interesting challenge in this assignment is aligning the output, for which the library module you wrote for the preceding exercise will come in handy.

9. The fact that every entry in Pascal's Triangle is the sum of the two entries above it makes it possible to calculate *C*(*n, k*) recursively. Use this insight to write a recursive implementation of the `combinations` function without using any loops or calls to `fact`.

10. Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Write a recursive function `cannonball` that takes as its argument the height of the pyramid and returns the number of cannonballs it contains. Your function must operate recursively and must not use any iterative constructs, such as `while` or `for`.

11. Rewrite the `fib` function so that it operates iteratively rather than recursively.

12. Rewrite the `digit_sum` function from page 45 so that it operates recursively instead of iteratively. To do so, you need to identify both the simple cases and the necessary recursive insight.

13. Rewrite the `gcd` function that uses Euclid's algorithm shown on page 72 so that it computes the greatest common divisor recursively using the following rules:

    - If *y* is zero, then *x* is the greatest common divisor.
    - Otherwise, the greatest common divisor of *x* and *y* is always equal to the greatest common divisor of *y* and the remainder of *x* divided by *y*.

CHAPTER 6

Writing Interactive Programs

Quit worrying about failure. Failure's easy. Worry about if you're successful, because then you have to deal with it.

— Adele Goldberg, interview with John Mashey, 2010

Adele Goldberg (1945–)

Adele Goldberg received her Ph.D. in Information Science from the University of Chicago and took a research position at the Xerox Palo Alto Research Center (PARC), which introduced the graphical user interface—an idea that has since become central to modern computing. Together with others in the Learning Research Group at PARC, Goldberg designed and implemented the programming language Smalltalk, which took the ideas of object-oriented programming developed in Scandinavia and integrated them into a programming environment designed to support constructivist learning in which students build knowledge from their experiences. Drawing on the state-of-the-art technology invented at PARC, Smalltalk was among the first programming environments designed for use with graphical displays. Along with her colleagues Alan Kay and Dan Ingalls, Goldberg received the Software Systems Award from the Association for Computing Machinery, the leading professional society for computer science, in 1987.

So far, the only interactions you have had with Python programs have taken place in the context of the IDLE interpreter. When you enter an expression on the IDLE console, the Python interpreter evaluates that expression and displays the result. When you run a program from the command line, your experience so far is that the program runs to completion with no further interaction with the user.

This style of interaction, whether executed in IDLE or from the console, is called ***synchronous,*** because user actions are synchronized with the program operation. A graphical user interface (often shortened to the acronym ***GUI,*** which is pronounced like *gooey*), by contrast, is ***asynchronous,*** in that it allows the user to intercede at any point, typically by using the mouse or the keyboard to trigger an action. Actions that occur asynchronously with respect to the program operation, such as clicking the mouse or typing on the keyboard, are generically referred to as ***events.*** Interactive programs that operate by responding to these events are said to be ***event-driven.*** The primary goal of this chapter is to teach you how to write simple event-driven programs.

Historically, the development of the graphical user interface has been closely associated with the object-oriented paradigm, which is itself commonly abbreviated as ***OOP.*** There are at least two reasons that the GUI and OOP have worked well together (beyond the fact that they have both become popular three-letter buzzwords in the computing industry). First, graphical displays are characterized by having many independent objects that form a hierarchical relationship that fits easily into the object-oriented paradigm. Second, it is easy to think of events as messages, which are a central foundation of the object-oriented model. Clicking the mouse, for example, sends a message to the application, which then responds in an appropriate way.

6.1 First-class functions

Before looking at the details of how event-driven programs are implemented in Python, it is useful to spend a little more time considering the question of how Python implements the idea of a function. In the programs you have seen so far in this book, the ideas of functions and data have remained separate. Functions provide the means for representing an algorithm. Those functions then operate on data values, which act as the raw material on which computation is performed. Functions have been part of the algorithmic structure, not part of the data structure. Being able to use functions as data values, however, often makes it much easier to design effective interfaces, because this facility allows clients to specify operations as well as data.

In Python, functions are values that are simultaneously part of both the algorithmic structure and the data structure of a program. Given a functional value, you can assign it to a variable, pass it as a parameter, or return it as a result. When a programming language allows functions to behave just like any other data value, computer scientists say that the language supports ***first-class functions.***

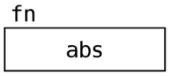
As noted in Chapter 1, a data type is defined as a combination of a domain and a set of operations. For the function data type, the domain is the vast spectrum of functions that you can to define in Python. The operation that is particular to the function data type is ***application,*** which is the process of calling that function with a list of arguments.

Assigning functions to variables

As a starting point in understanding the concept of treating functions as data objects, it makes sense to look at the process of assigning a function value to a variable. The function `abs` for example is one of Python's built-in functions. You could assign that function value to a variable using the assignment statement

```
fn = abs
```

Like any assignment statement, this line creates a variable named `fn` and assigns it a value, which is the built-in function `abs`. You can diagram the resulting situation like this:

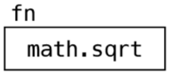


Since the variable `fn` contains a function as its value, you can call `fn`, just as you call the built-in function `abs`. Calling `fn(-3)`, for example, returns the integer 3.

As with any variable, you can also change the value that `fn` contains. If you have imported the `math` library, you could execute the assignment

```
fn = math.sqrt
```

which would change the value stored in the variable as follows:



Calling `fn(25)` at this point would return 5.0 as a floating-point value.

Although the idea of assigning functions to variable may initially seem rather esoteric, you have already seen it done. The Python statement

```
from math import sqrt
```

is equivalent to the statements

```
import math
sqrt = math.sqrt
```

The assignment stores the function value `math.sqrt` in the global variable `sqrt`, which is then available in the current module.

Closures

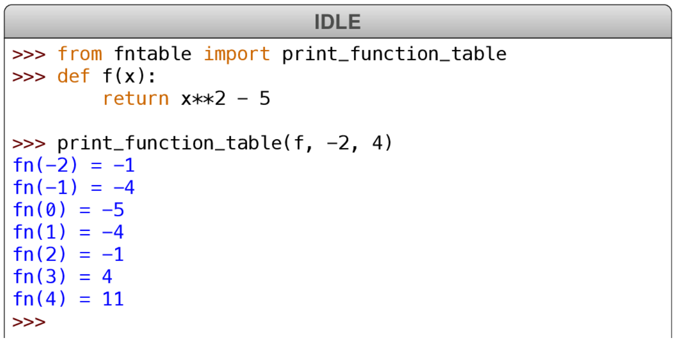
For functions defined in a library or at the top level of a module, assigning that function to a variable corresponds to storing its name in a variable, as the box diagrams in the previous section suggest. The situation is more interesting if you assign an inner function, as described in section 5-6, to a variable. Inner functions have access to the local variables declared in the enclosing function. In Python, that access is part of the function value, which combines the code that implements it with the variables in its scope. This combination of code and variables is called a ***closure.*** Closures are an amazingly powerful feature of languages like Python and will prove essential to writing interactive programs.

Passing functions as parameters

Since functions in Python are first-class values, they can be passed as parameters. One example of an application in which doing so makes intuitive sense is the following function:

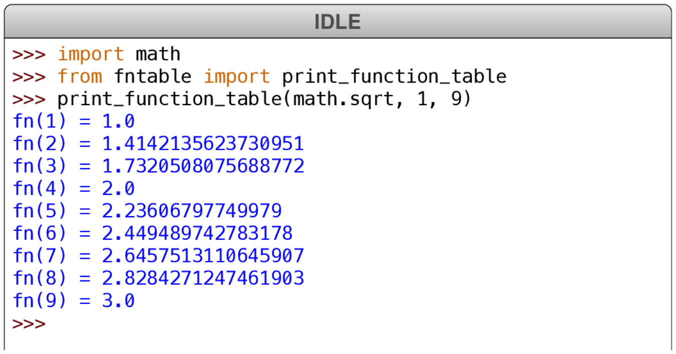
```
def print_function_table(fn, min, max):
    for i in range(min, max + 1):
        print("fn(" + str(i) + ") = " + str(fn(i)))
```

The first parameter is a function that takes a number and returns a result. The effect of `print_function_table` is to count from `min` to `max`, generating a line of output that shows the value of the function at each of those values. For example, if this definition of `print_function_table` is stored in the Python module `fntable`, you could generate the following IDLE session:



```
>>> from fntable import print_function_table
>>> def f(x):
        return x**2 - 5

>>> print_function_table(f, -2, 4)
fn(-2) = -1
fn(-1) = -4
fn(0) = -5
fn(1) = -4
fn(2) = -1
fn(3) = 4
fn(4) = 11
>>>
```

The first argument to `print_function_table` can be any function. For example, using `math.sqrt` as the argument generates the following IDLE session:



```
>>> import math
>>> from fntable import print_function_table
>>> print_function_table(math.sqrt, 1, 9)
fn(1) = 1.0
fn(2) = 1.4142135623730951
fn(3) = 1.7320508075688772
fn(4) = 2.0
fn(5) = 2.23606797749979
fn(6) = 2.449489742783178
fn(7) = 2.6457513110645907
fn(8) = 2.8284271247461903
fn(9) = 3.0
>>>
```

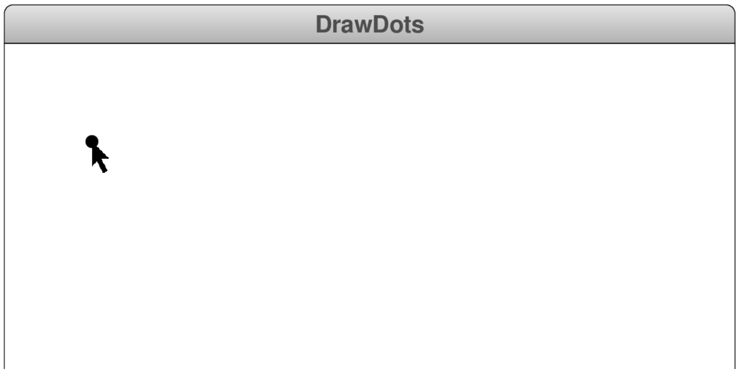
6.2 A simple interactive example

Before becoming immersed in the details, it helps to consider a simple example that illustrates the graphics library's model for user interaction. The `DrawDots.py` program in Figure 6-1 draws a small dot whenever the user clicks the mouse button.

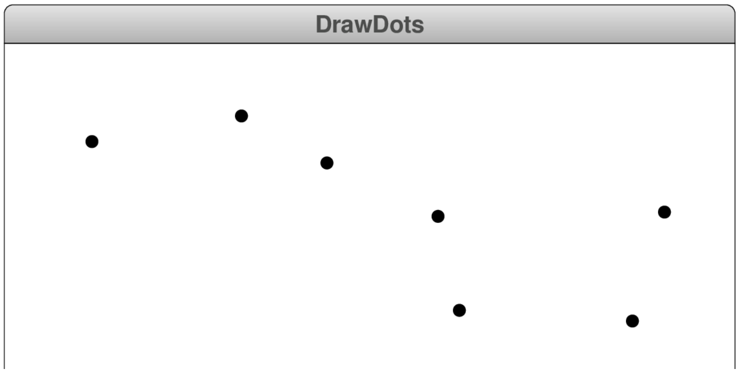
FIGURE 6-1 **Program to draw dots when the user clicks the mouse**

```
# File: DrawDots.py

"""This program draws a dot every time the user clicks the mouse."""

from pgl import GWindow, GOval
from gtools import create_filled_circle

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
DOT_SIZE = 6

# Function: draw_dots

def draw_dots():

    def click_action(e):
       gw.add(create_filled_circle(e.get_x(), e.get_y(), DOT_SIZE / 2))

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    gw.add_event_listener("click", click_action)

# Startup code

if __name__ == "__main__":
    draw_dots()
```

For example, if you click the mouse near the upper left corner of the window, the program will draw a dot in that position, as shown in the following diagram:



If you then go on to click the mouse in other positions, dots will appear there as well. You could, for example, draw a picture of the constellation Ursa Major, which is more commonly known as the Big Dipper. All you would have to do is click the mouse once in the position of each star, as follows:



Although the code in Figure 6-1 is extremely short, the program is different enough from the ones that you've seen so far that it makes sense to go through it in detail. The function begins by defining the function `click_action`, which specifies what happens when the user clicks the mouse. The next statement creates the graphics window, precisely as you always have. The final statement in the `DrawDots.py` program establishes the link between the graphics window and the behavior specified by `click_action`. Executing the line

```
gw.add_event_listener("click", click_action)
```

tells the graphics window that it wants to respond to mouse clicks. Moreover, the response to that mouse click is specified by `click_action`, which is called automatically whenever a click occurs.

It is useful to note that the code in Figure 6-1 never calls click_action explicitly. The call, when it happens, comes from the code that implements the graphics library. Functions that the program does not call directly but that instead occur in response to some event are referred to as ***callback functions.*** The name reflects the relationship between the client program and the libraries it uses. As a client, your program calls add_event_listener to register interest in a particular event. As part of that process, you provide the library with a function that it can call when the event occurs. It is, in a way, analogous to providing a callback number. When the library implementation needs to call you back, you've given it the means to do so.

Now that you have a sense of how callback functions work in general, you are in a better position to understand the click_action function, which looks like this:

```
def click_action(e):
    gw.add(create_filled_circle(e.get_x(), e.get_y(),
                                DOT_SIZE / 2))
```

The function takes a parameter e, which provides the function with data about the details of the event. In this case, e is a ***mouse event,*** which keeps track of the location of the mouse along with other data. Callback functions that respond to mouse events can determine the location of the mouse by invoking the methods e.get_x() and e.get_y(). Each of these methods returns a coordinate in pixels measured relative to the origin in the upper left corner of the window.

The click_action function calls create_filled_circle to create the dot and then adds it to the window so that its center appears as the current mouse position. The variable gw, which is a local variable inside draw_dots, is accessible to the click_action code because its definition appears in the enclosing function.

6.3 Controlling properties of objects

Before moving on to look at more sophisticated examples of interactivity, it is important to have a more complete understanding of how to manipulate graphical objects that have already been placed on the screen. So far, the objects that you've added to the graphics window retain their initial location and dimensions. When you build interactive programs, you need to be able to change these properties.

The classes in the graphics library export a richer set of methods than you have had a chance to use so far. Figure 6-2 lists the complete set of methods supported by every graphical object and a few that apply only to specific classes. Each of the method descriptions consists of a single line that offers an overview of what the method does. For more details, you can look up the documentation on the web.

FIGURE 6-2 **Expanded list of methods available in the graphics library**

Methods that control the location of the object

obj`.set_location(`*x*`, `*y*`)`	Sets the location of this object to the point (*x*, *y*).
obj`.move(`*dx*`, `*dy*`)`	Moves the object using the displacements *dx* and *dy*.
obj`.move_polar(`*r*`, `*theta*`)`	Moves the object *r* pixels in direction *theta*.

Methods that control the appearance of the object

obj`.set_color(`*color*`)`	Sets the color used to display this object.
obj`.set_line_width(`*width*`)`	Sets the width of the lines used to draw the object.
obj`.set_visible(`*flag*`)`	Sets whether this object is visible.
obj`.rotate(`*theta*`)`	Rotates the object *theta* degrees around its origin.
obj`.scale(`*sf*`)`	Scales the object by *sf* both horizontally and vertically.

Methods that control the stacking order

obj`.send_backward()`	Moves this object one step backward in the stacking order.
obj`.send_forward()`	Moves this object one step forward in the stacking order.
obj`.send_to_back()`	Moves this object to the back of the stacking order.
obj`.send_to_front()`	Moves this object to the front of the stacking order.

Methods that return properties of the object

obj`.get_x()`	Returns the *x* coordinate of the object.
obj`.get_y()`	Returns the *y* coordinate of the object.
obj`.get_width()`	Returns the width of this object.
obj`.get_height()`	Returns the height of this object.
obj`.get_color()`	Returns the color used to display this object.
obj`.get_line_width()`	Returns the width of the lines used to draw the object.
obj`.is_visible()`	Checks to see whether this object is visible.
obj`.contains(`*x*`, `*y*`)`	Checks to see whether the point (*x*, *y*) is inside the object.

Methods available only for the `GRect` and `GOval` classes

obj`.set_filled(`*flag*`)`	Sets whether this object is filled.
obj`.set_fill_color(`*color*`)`	Sets the color used to fill the interior of the object.
obj`.set_bounds(`*x*`, `*y*`, `*width*`, `*height*`)`	Resets the boundary rectangle for the object.

Methods available only for the `GLine` class

obj`.set_start_point(`*x*`, `*y*`)`	Changes the start point of the line without changing the end.
obj`.set_end_point(`*x*`, `*y*`)`	Changes the end point of the line without changing the start.

Instead of going through each of these methods in detail, this chapter presents several programming examples that introduce new methods only as they are needed. As a result, you have a chance to learn about each of the new methods in the context of an application that makes use of it.

6.4 Responding to mouse events

The "click" event used in the DrawDots.py program is only one of several mouse events that Python allows you to detect. The mouse events implemented by the GWindow class are shown in Figure 6-3. Each of these event names allows you to respond to a specific type of action with the mouse, most of which will seem familiar from using your computer. The "mousemove" event, for example, is generated when you move the mouse in the window without pressing the mouse button. The "drag" event occurs when you move the mouse while holding the button down. The name of the event comes from the fact that the interaction model of moving the mouse with the button down is often used to drag objects around on the window. You press the mouse button over an object to grab it and then drag it to the desired position.

The sections that follow offer several examples that illustrate conventional styles of using the mouse to create and reposition objects in the graphics window.

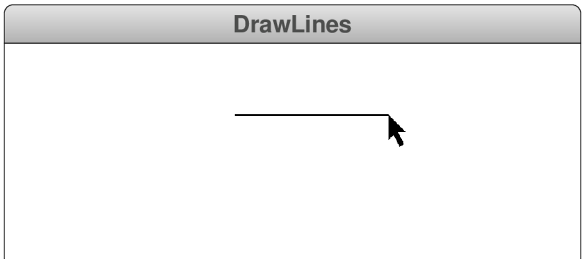
A simple line-drawing program

In all likelihood, you have already used some application that allows you to draw lines on the screen by dragging the mouse. To create a line, you press the mouse button at the point at which you'd like the line to start and then drag the mouse with the button down until you reach the point at which you want the line to end. As you drag the mouse, the application typically updates the line so that you can see what you have drawn so far. When you release the mouse button, the line stays in that position, and you can repeat the process to create as many new lines as you wish.

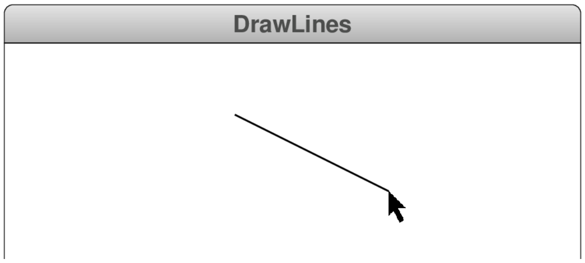
FIGURE 6-3 Common mouse event types

"click"	The user clicks the mouse in the window.
"dblclick"	The user double-clicks the mouse in the window.
"mousedown"	The user presses the mouse button.
"mouseup"	The user releases the mouse button.
"mousemove"	The user moves the mouse with the button up.
"drag"	The user drags the mouse (that is, moves the mouse with the button down).

Suppose, for example, that you press the mouse button somewhere on the screen and then drag the mouse rightward an inch, holding the button down. What you'd like to see is the following picture:

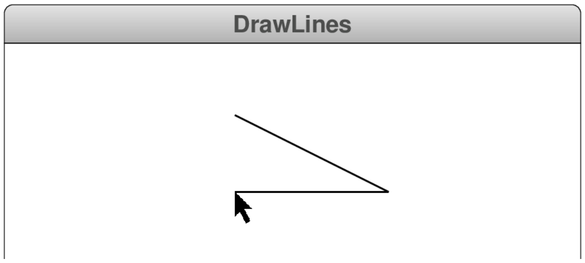


If you then move the mouse downward without releasing the button, the displayed line will track the mouse, so that you might see the following picture:



As you drag the mouse, the application repeatedly updates the line, making it appear to stretch as the mouse moves. Because the effect is precisely what you would expect if you joined the starting point and the mouse cursor with a stretchy elastic line, this technique is called ***rubber-banding.***

When you release the mouse, the line stays where it is. If you then press the mouse button again on that same point, you can go ahead and draw an additional line segment by dragging the mouse to the end point of the new line, as follows:



At least in terms of the conceptual strategy, this problem doesn't initially seem that different from the one used earlier in the `DrawDots.py` program. When the user presses the mouse button, the program creates a zero-length line that starts and ends at the current mouse position. As the user drags the mouse to a new position, all the program needs to do is change the endpoint of that line to the new mouse coordinates. The program therefore needs to listen for both the "mousedown" and "drag" events and then implement the necessary operations on a GLine object that is shared throughout the entire program.

Unfortunately, writing the code for the DrawLines.py program is not quite as simple as the preceding paragraph suggests. If you try to solve this program by storing the shared GLine object in the closure, you would be tempted to write a program that looks something like this:

```
def draw_lines():

    def mousedown_action(e):
        line = GLine(e.get_x(), e.get_y(),
                     e.get_x(), e.get_y())
        gw.add(line)

    def drag_action(e):
        line.set_end_point(e.get_x(), e.get_y())

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    line = None
    gw.add_event_listener("mousedown", mousedown_action)
    gw.add_event_listener("drag", drag_action)
```

As the bug symbol indicates, however, this strategy fails because of the way Python treats local variables.

Most modern programming languages require you to indicate what variables are local to a function using a specification called a ***declaration.*** Python doesn't. To make the language easier for novices, Python looks through the code for a function for all variables that appear on the left side of an assignment and then automatically declares those variables as local. Because the variable line appears on the left side of an assignment in mousedown_action, Python treats it as a new local variable with no connection to the line variable that appears in the enclosing draw_lines function. As a result, the line variable in mousedown_action is entirely separate from the line variable in draw_lines and drag_action.

Although Python now includes a mechanism for indicating that a function should use an enclosing definition instead of creating a new local variable, the syntax for doing so is confusing to new students and makes programs that use it more difficult to read and maintain. A better strategy—and certainly the one more likely to be adopted by professional programmers—is to collect the variables that need to be shared into a single object and then rely on closures to ensure that all the inner functions have access to those values. The only problem with adopting that strategy at this point in the text is that the process of creating objects and assigning to their components is not covered in detail until Chapter 10.

Fortunately, the fact that this problem usually arises in graphical programs makes it possible to implement a simple workaround. Every program that uses the Portable Graphics Library defines a `GWindow` object that is stored by convention in a variable named `gw`. Assuming that you define `gw` in the main program, any callback functions you define will have access to this variable. That fact means that you can store any data you need to share inside the `GWindow` object.

For example, instead of defining a variable named `line` as in the buggy version of `DrawLines.py`, you define a new component of the `gw` object called `gw.line`. The revised code appears in Figure 6-4.

FIGURE 6-4 Code for the DrawLines.py program

```
# File: DrawLines.py

"""This program lets the user draw lines by dragging the mouse."""

from pgl import GWindow, GLine

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300

# Implementation notes: draw_lines
# --------------------------------
# The variable line is the target of an assignment in mousedown_action and
# used in drag_action.  It must therefore be stored as a component of gw.

def draw_lines():

    def mousedown_action(e):
        gw.line = GLine(e.get_x(), e.get_y(), e.get_x(), e.get_y())
        gw.add(gw.line)

    def drag_action(e):
        gw.line.set_end_point(e.get_x(), e.get_y())

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    gw.add_event_listener("mousedown", mousedown_action)
    gw.add_event_listener("drag", drag_action)

# Startup code

if __name__ == "__main__":
    draw_lines()
```

The only differences between the code in Figure 6-4 and the earlier buggy version of DrawLines.py is that all occurrences of the variable line in the original program have been replaced by gw.line in place of the earlier variable line. Because no assignments are made to the gw variable itself outside of the main program, that variable is shared through the closure. Functions like mousedown_action and drag_action can refer to the individual components of gw without breaking the sharing arrangement that allows these programs to work.

Most of the graphical programs in this text use this strategy of storing shared state inside the gw variable. You will have a chance to learn more about objects and their uses in Chapter 10.

Dragging objects on the canvas

The DragObjects.py program in Figure 6-5 on the next page offers a slightly more sophisticated example of an event-driven program that uses the mouse to reposition objects on the display. This program begins by adding a blue rectangle and a red oval to the window, just as in the GRectPlusGOval.py program from Chapter 4. The rest of the program represents the code pattern for dragging objects.

As in the DrawLines.py program in Figure 6-4, the callback functions in the DragObjects.py program need to assign new values to shared state variables. The main function therefore must store the following information in components of the GWindow object so that the inner functions can manipulate those values:

1. gw.last_x, which is the *x* coordinate at which the last mouse event occurred
2. gw.last_y, which is the corresponding *y* coordinate
3. gw.gobj, which is the object being dragged

The mousedown_action function consists of the following code:

```
def mousedown_action(e):
    gw.last_x = e.get_x()
    gw.last_y = e.get_y()
    gw.gobj = gw.get_element_at(gw.last_x, gw.last_y)
```

The first two statements in simply record the *x* and *y* coordinates of the mouse in the state variables gw.last_x and gw.last_y. The third statement records the object being moved through the use of an important new method in the GWindow class called get_element_at, which takes an *x* and a *y* coordinate and then checks to see what object displayed on the window contains that location. Here, it is important to recognize that there are two possibilities. First, you could be pressing the mouse button

FIGURE 6-5 Code for the `DragObjects.py` program

```
# File: DragObjects.py

"""This program allows the user drag objects with the mouse."""

from pgl import GWindow, GOval, GRect

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 200
GOBJECT_WIDTH = 200
GOBJECT_HEIGHT = 100

# Main program

def drag_objects():

    def mousedown_action(e):
        gw.last_x = e.get_x()
        gw.last_y = e.get_y()
        gw.gobj = gw.get_element_at(gw.last_x, gw.last_y)

    def drag_action(e):
        if gw.gobj is not None:
            gw.gobj.move(e.get_x() - gw.last_x, e.get_y() - gw.last_y)
            gw.last_x = e.get_x()
            gw.last_y = e.get_y()

    def click_action(e):
        if gw.gobj is not None:
            gw.gobj.send_to_front()

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    x0 = (gw.get_width() - GOBJECT_WIDTH) / 2
    y0 = (gw.get_height() - GOBJECT_HEIGHT) / 2
    rect = GRect(x0, y0, GOBJECT_WIDTH, GOBJECT_HEIGHT)
    rect.set_filled(True)
    rect.set_color("Blue")
    gw.add(rect)
    oval = GOval(x0, y0, GOBJECT_WIDTH, GOBJECT_HEIGHT)
    oval.set_filled(True)
    oval.set_color("Red")
    gw.add(oval)
    gw.add_event_listener("mousedown", mousedown_action)
    gw.add_event_listener("drag", drag_action)
    gw.add_event_listener("click", click_action)

# Startup code

if __name__ == "__main__":
    drag_objects()
```

on top of an object, which means that you want to start dragging it. Second, you could be pressing the mouse button somewhere else on the canvas at which there is no object to drag. If just one object exists at the specified location, get_element_at returns that object. If more than one object covers that space, get_element_at chooses the one in front of the others in the stacking order. If no objects exist at that location, get_element_at returns the special value None. In any of those cases, the mousedown_action function assigns that value to the state variable gw.gobj.

The drag_action function consists of the following code:

```
def drag_action(e):
    if gw.gobj is not None:
        gw.gobj.move(e.get_x() - gw.last_x,
                     e.get_y() - gw.last_y)
        gw.last_x = e.get_x()
        gw.last_y = e.get_y()
```

The if statement checks to see whether there is an object to drag. If the value of gobj is None, there is nothing to drag, so the rest of the function can just be skipped. If there is an object, you need to move it by some distance in each direction. That distance does not depend on the current coordinates of the mouse but rather on how far it has moved from where it was when you last noted its position. Thus, the arguments to the move method are—for both the *x* and *y* components—the location where the mouse is now minus where the mouse was at the time of the last event. Those coordinates are stored in the variables gw.last_x and gw.last_y. Once you have moved the object, you must then update these values to ensure that they are correct for the next call to mouse_dragged.

The DragObjects.py program also registers its interest in "click" events, which trigger a call to the following function:

```
def click_action(e):
    if gw.gobj is not None:
        gw.gobj.send_to_front()
```

The point of adding this function is to allow the user to change the stacking order, which, as noted in Chapter 4, is the order in which objects are layered on the screen.

In the DragObjects.py program, clicking on an object has the effect of moving it to the front of the stacking order. Implementing this behavior correctly, however, requires understanding the rules that the Portable Graphics Library uses for mouse events. A "click" event occurs when a "mousedown" event is followed within a relatively short amount of time by a "mouseup" event. By the time Python processes the "click" event, the "mousedown" and "mouseup" events have already occurred. Although DragObjects.py does not specify any action for "mouseup", it responds to the "mousedown" event by calling mousedown_action. Thus, by the time the call to click_action occurs, the mousedown_action function will already have set the value of gw.gobj.

6.5 Timer-based animation

Interactive programs change their behavior not only in response to user events, but also over time. In a computer game, for example, objects on the screen typically move in real time. Updating the contents of the graphics window so that they change over time is called ***animation.***

The GWindow class in the Portable Graphics Library exports two methods that support animation by allowing the application to invoke a callback function after a specified delay. The method

```
gw.set_timeout(function, delay)
```

creates a ***one-shot timer*** that calls *function* after *delay* milliseconds. The method

```
gw.set_interval(function, delay)
```

creates an ***interval timer*** that calls *function* repeatedly every *delay* milliseconds. Each of these methods returns an object called a ***timer*** that makes it possible to control the animation process.

As an example, executing

```
timer = gw.set_interval(step, 20)
```

creates an interval timer and stores the resulting timer object in the variable timer. The interval timer then begins generating calls to the function step once every 20 milliseconds, or every fiftieth of a second. The name step is chosen here to suggest that each call represents a single step in the animation, which is called a ***time step.*** The step function takes no arguments, so any information it needs must be communicated through the closure of the function in which step is defined.

Timers that initiate events every 20 milliseconds allow you to change the state of the graphics window quickly enough so that the changes appear smooth to the human eye. You can therefore move an object on the screen by creating an interval timer that executes its callback function every 20 milliseconds and then having the callback function make an incremental change to the position of that object.

The reason for storing the timer object is that doing so allows you to invoke its stop method, which turns off the timing process and prevents any subsequent invocations of the callback function. In the context of an animation, for example, you can call `timer.stop()` when the animated object reaches its final location.

A simple example of animation

A simple example of timer-based of animation appears in Figure **Error! Reference source not found.**-6, which moves a square diagonally across the screen from its initial position in the upper left corner to its final position in the lower right, moving one pixel in each dimension on every time step.

The code for the callback function looks like this:

```
def step():
    square.move(dx, dy)
    if square.get_x() + SQUARE_SIZE > gw.get_width():
        timer.stop()
```

FIGURE 6-6 **Program to move a square diagonally across the screen**

```
# File: AnimatedSquare.py

from pgl import GWindow, GRect

# Constants

GWINDOW_WIDTH = 500            # Width of the graphics window
GWINDOW_HEIGHT = 300           # Height of the graphics window
N_STEPS = 100                  # Number of steps in the simulation
TIME_STEP = 20                 # Time step in milliseconds
SQUARE_SIZE = 50               # Size of the square in pixels

def animated_square():
    def step():
        square.move(dx, dy)
        if square.get_x() + SQUARE_SIZE >= gw.get_width():
            timer.stop()

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    dx = (gw.get_width() - SQUARE_SIZE) / N_STEPS
    dy = (gw.get_height() - SQUARE_SIZE) / N_STEPS
    square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE)
    square.set_filled(True)
    gw.add(square)
    timer = gw.set_interval(step, TIME_STEP)

# Startup code

if __name__ == "__main__":
    animated_square()
```

The first line adjusts the position of the square by the values `dx` and `dy`. The rest of the function tests whether the square is still inside the window and stops the timer if it has moved beyond the boundary.

Tracking the state of an animation

As animations become more complex, keeping track of the state of the animation becomes a bit tricky. Suppose, for example, that you want to add animation to the `RandomCircles.py` program in Figure 5-5 on page 154. Instead of having the circles all show up at once, what you want is for the circles to appear slowly, one at a time. Each circle begins as a single point and then grows until it reaches its desired size. The program should then creates the next circle and lets it grow, continuing in this fashion until all ten circles are displayed on the screen.

It is, of course, tempting to start this program by building on the earlier example. That strategy would suggest adopting the following pseudocode structure:

```
for i in range(N_CIRCLES):
    Create a circle.
    Animate that circle so that it grows to full size.
    Wait for that animation to complete.
```

Unfortunately, that strategy doesn't work well if you use the Portable Graphics Library, which expects all interactions to be ***event-driven,*** in the sense that all actions take place in response to events that occur asynchronously. This event model rules out the earlier pseudocode approach and requires a different strategy, in which all aspects of the animation are implemented inside the `step` function. The `step` function therefore has the following pseudocode form:

```
def step():
    if  the current circle is still growing:
        Increase the size of the current circle.
    elif  there are more circles to create:
        Create another circle.
    else:
        timer.stop()
```

The code for `GrowingCircles.py` appears in Figure 6-7 on the next page. The code for the `create_new_circle` function is largely the same as the code for `create_random_circle` in Figure 5-5. The only differences are that

1. The `create_new_circle` function creates circles whose initial size is 0.
2. The `create_new_circle` function records the eventual and current size of the circle in the state components `gw.desired_size` and `gw.current_size`.

FIGURE 6-7 Program to create ten circles that start as a point and then grow to full size

```
# File: GrowingCircles.py

from pgl import GWindow
from gtools import create_filled_circle
from RandomCircles import random_color
import random

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
N_CIRCLES = 10
MIN_RADIUS = 15
MAX_RADIUS = 50
TIME_STEP = 20
DELTA_SIZE = 1

# Main program

def growing_circles():

    def create_new_circle():
        r = random.uniform(MIN_RADIUS, MAX_RADIUS)
        x = random.uniform(r, GWINDOW_WIDTH - r)
        y = random.uniform(r, GWINDOW_HEIGHT - r)
        gw.circle = create_filled_circle(x, y, 0, random_color())
        gw.desired_size = 2 * r
        gw.current_size = 0
        gw.circles_created += 1
        return gw.circle

    def step():
        if gw.current_size < gw.desired_size:
            gw.current_size += DELTA_SIZE
            x = gw.circle.get_x() - DELTA_SIZE / 2
            y = gw.circle.get_y() - DELTA_SIZE / 2
            gw.circle.set_bounds(x, y, gw.current_size, gw.current_size)
        elif gw.circles_created < N_CIRCLES:
            gw.add(create_new_circle())
        else:
            timer.stop()

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    gw.circles_created = 0
    gw.current_size = 0
    gw.desired_size = 0
    timer = gw.set_interval(step, TIME_STEP)

# Startup code

if __name__ == "__main__":
    growing_circles()
```

The code for the step function follows the pseudocode outline shown earlier on this page. The only new feature is the call to set_bounds, which resets the location and size of the current circle so that it grows by one pixel in each time step.

It is worth noting that the main program explicitly initializes the state variables gw.desired_size and gw.current_size to 0. Setting these two variables to the same value ensures that create_new_circle is called on the first time step

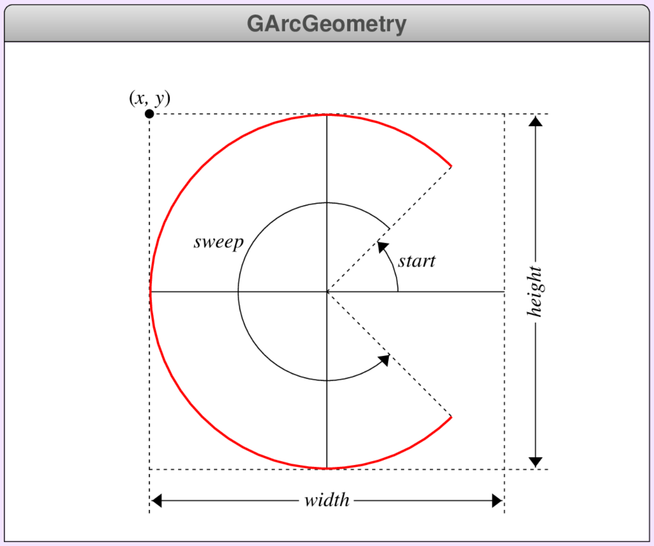
6.6 Expanding the graphics library

Ever since Chapter 4, you've been using classes from the Portable Graphics Library to create simple drawings on the screen. So far, however, you have seen only a small part of what the graphics library has to offer. Now that you know how to write programs that involve animation and interactivity, it makes sense to learn more about the graphics library and how to use it. This section introduces three new classes—GArc, GPolygon, and GCompound—that allow you to create more interesting graphical displays.

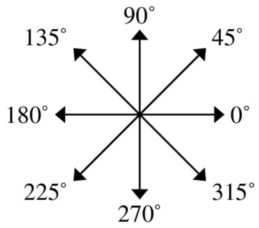
The GArc class

The GArc class is used to display an arc formed by selecting part of the perimeter of an oval. The GArc function itself takes six parameters—x, y, width, height, start, and sweep—which are illustrated in Figure 6-8. The first four parameters specify the location and size of the rectangle that encloses the arc and therefore have precisely

FIGURE 6-8 **The geometric interpretation of the GArc parameters**



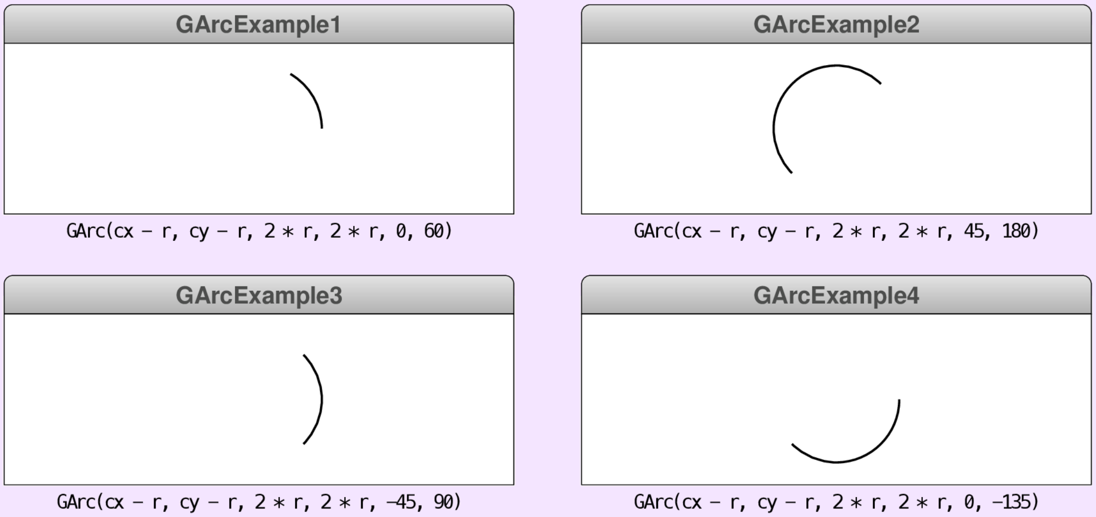
the same interpretation as those parameters in calls to GRect or GOval. The next two parameters specify the ***start angle,*** which is the angle at which the arc begins, and the ***sweep angle,*** which is the number of degrees through which the arc extends. In keeping with mathematical convention, angles in the graphics library are measured in degrees counterclockwise from the +*x* axis, as follows:



The effect of these parameters is most easily demonstrated by example. The four sample runs in Figure 6-9 show the effect of the code below each diagram. The code fragments create arcs using different values for start and sweep. Each of the arcs has a radius of r pixels and is centered at the point (cx, cy).

The GArc class implements the methods shown in Figure 6-10 on the next page. As you can see, these methods include set_filled and set_fill_color, just as GRect and GOval do. It is not immediately apparent, however, exactly what filling an arc means. In the interpretation of arc-filling used in the Portable Graphics Library, the unfilled version of a GArc is not simply the boundary of its filled counterpart. If you display an unfilled GArc, only the arc itself is shown. If you call

FIGURE 6-9 Examples of GArc objects



GArc(cx − r, cy − r, 2 * r, 2 * r, 0, 60)

GArc(cx − r, cy − r, 2 * r, 2 * r, 45, 180)

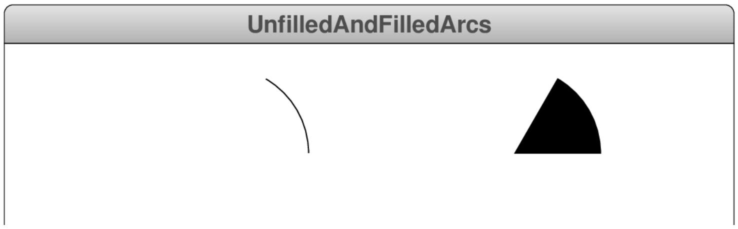
GArc(cx − r, cy − r, 2 * r, 2 * r, -45, 90)

GArc(cx − r, cy − r, 2 * r, 2 * r, 0, -135)

FIGURE 6-10 Methods implemented by the `GArc` class

arc`.set_filled(`*flag*`)`	Sets whether the wedge for this arc is filled.
arc`.set_fill_color(`*color*`)`	Sets the color used to fill the wedge for this arc.
arc`.set_start_angle(`*start*`)`	Sets the start angle to *start*.
arc`.get_start_angle()`	Returns the start angle.
arc`.set_sweep_angle(`*sweep*`)`	Sets the sweep angle to *sweep*.
arc`.get_sweep_angle()`	Returns the sweep angle.

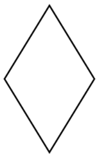
`set_filled(True)` on that arc, the graphics library connects the end points of the arc to the center from which the arc was drawn and then fills the interior of that region. The following sample run illustrates the difference by showing both unfilled and filled versions of the same 60-degree arc:



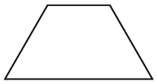
The important lesson to take from this example is that the geometric boundary of a `GArc` changes if you set it to be filled. A filled arc is a wedge-shaped region that has a well-defined interior. An unfilled arc is simply a section taken from the boundary of an ellipse. If you want to display the outline of the wedge that calling `set_filled` would generate, the simplest strategy is to call `set_filled(True)` and then use `set_fill_color("White")` to set the interior of the region to white.

The `GPolygon` class

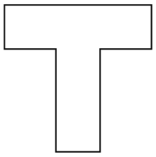
The `GPolygon` class makes it possible to display a ***polygon,*** which is simply the mathematical name for a closed shape whose boundary consists of straight lines**.** The line segments that form the outline of a polygon are called ***edges.*** The point at which a pair of edges meets is called a ***vertex.*** Many polygonal shapes are familiar from the real world. Each cell in a honeycomb is a hexagon, which is the common name for a polygon with six sides. A stop sign is an octagon with eight identical sides. Polygons, however, are not required to have equal sides and angles. The figures in the left margin, for example, illustrate four polygons that fit the general definition.



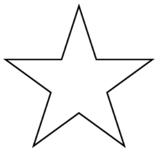
diamond



trapezoid



T-shape



five-pointed star

The GPolygon class is easy to use if you keep the following points in mind:

- Unlike the functions that create the other shapes, the GPolygon function does not create the entire figure. What happens instead is that calling GPolygon creates an empty polygon. Once you have created an empty polygon, you then add vertices to it by calling various other methods described later in this section.
- The origin of a GPolygon is not defined to be its upper left corner. Many polygons, after all, don't have an upper left corner. What happens instead is that you—as the programmer who is creating the specific polygon—choose a ***reference point*** that defines the location of the polygon as a whole. You then specify the coordinates for each vertex in terms of where they lie in relation to the reference point. This approach makes it easier to move the polygon as a unit.

The creation of a GPolygon is easiest to illustrate by example. Suppose that you want to create a GPolygon representing the diamond-shaped figure shown in the margin. Your first design decision consists of choosing where to put the reference point. For most polygons, the most convenient point is the geometric center of the figure. If you adopt that model, you then need to create an empty GPolygon and add four vertices to it, specifying the coordinates of each vertex relative to the coordinates of the center. Assuming that the width and height of the diamond are stored in the constants DIAMOND_WIDTH and DIAMOND_HEIGHT, you can create the diamond-shaped GPolygon using the following code:

```
diamond = GPolygon()
diamond.add_vertex(-DIAMOND_WIDTH / 2, 0)
diamond.add_vertex(0, DIAMOND_HEIGHT / 2)
diamond.add_vertex(DIAMOND_WIDTH / 2, 0)
diamond.add_vertex(0, -DIAMOND_HEIGHT / 2)
```

When you use the add_vertex method to construct a polygon, the coordinates of each vertex are expressed relative to the reference point. In some cases, it is easier to specify the coordinates of each vertex in terms of the preceding one. To enable this approach, the GPolygon class offers an add_edge method, which is similar to add_vertex except that the parameters specify the displacement from the previous vertex to the current one. You can therefore create exactly the same GPolygon by making the following sequence of calls:

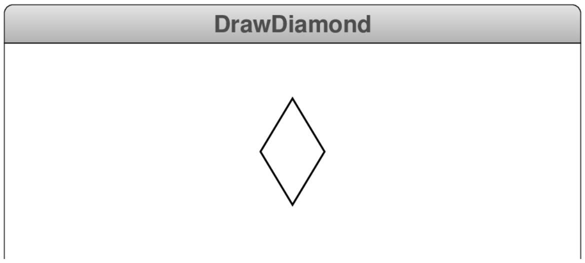
```
diamond = GPolygon()
diamond.add_vertex(-DIAMOND_WIDTH / 2, 0)
diamond.add_edge(DIAMOND_WIDTH / 2, DIAMOND_HEIGHT / 2)
diamond.add_edge(DIAMOND_WIDTH / 2, -DIAMOND_HEIGHT / 2)
diamond.add_edge(-DIAMOND_WIDTH / 2, -DIAMOND_HEIGHT / 2)
diamond.add_edge(-DIAMOND_WIDTH / 2, DIAMOND_HEIGHT / 2)
```

Note that the first vertex must still be added using add_vertex, but that subsequent ones can be defined by specifying the edge displacements.

Once you have defined the diamond shape by either of these methods, you can add the diamond at the center of the window using the following statement:

```
gw.add(diamond, gw.get_width() / 2, gw.get_height() / 2)
```

The graphics window then looks like this:



For many polygonal figures, it is easier to specify the edges using the method add_polar_edge. This method is identical to add_edge except that its arguments are the length of the edge and its direction, expressed in degrees counterclockwise from the +*x* axis.

The add_polar_edge method makes it easy to create figures in which you know the angles of the edges but would need trigonometry to calculate the vertices. The following function, for example, uses add_polar_edge to create a regular hexagon in which the length of each edge is determined by the parameter side:

```
def create_hexagon(side):
    hex = GPolygon()
    hex.add_vertex(-side, 0)
    angle = 60
    for i in range(6):
        hex.add_polar_edge(side, angle)
        angle -= 60
    return hex
```

As always, the first vertex is added using add_vertex. Here, the initial vertex is the one at the left edge of the hexagon. The first edge then extends from that point at an angle of 60 degrees. Each subsequent edge has the same length, but sets off at an angle 60 degrees to the right of the preceding one. When all six edges have been added, the final edge ends up at the original vertex, thereby closing the polygon.

Once you have defined this method, executing the statement

```
gw.add(create_hexagon(50), gw.get_width() / 2,
                           gw.get_height() / 2)
```

produces the following display:

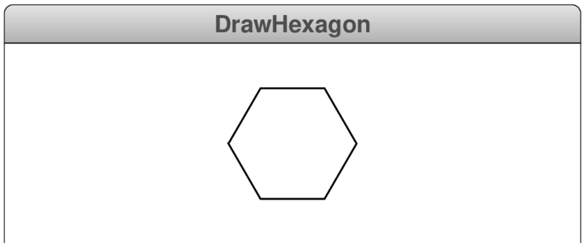
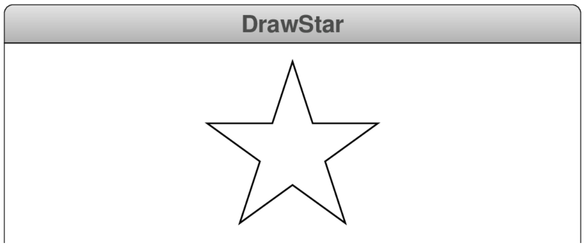


Figure 6-11 lists the methods that apply to the `GPolygon` class. As with the other bounded figures, `GPolygon` implements `set_filled` and `set_fill_color`.

As another example of using the `GPolygon` class, the `create_star` function in Figure 6-12 at the top of the next page creates a `GPolygon` whose edges form a five-pointed star, as follows:



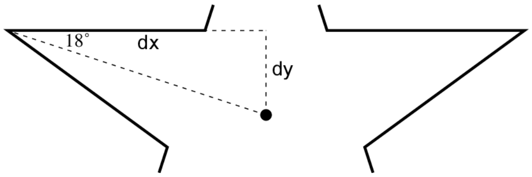
Although the star is more complicated mathematically than the earlier examples, the most difficult part is determining the coordinates of the starting point at the left edge of the star. Calculating the *x* coordinate is easy because the starting point is simply half the width of the star to the left of its center. Calculating the distance in the *y* direction requires a bit of trigonometry, which can be illustrated as follows:

FIGURE 6-11 Methods implemented by the GPolygon class

poly.`add_vertex(`*x*`,` *y*`)`	Adds a vertex at the point (*x*, *y*).
poly.`add_edge(`*dx*`,` *dy*`)`	Adds a vertex shifted by *dx* and *dy* from the preceding one.
poly.`add_polar_edge(`*r*`,` *theta*`)`	Adds a vertex shifted by *r* units in direction *theta.*
poly.`set_filled(`*flag*`)`	Sets whether the polygon is filled.
poly.`set_fill_color(`*color*`)`	Sets the color used to fill the polygon.

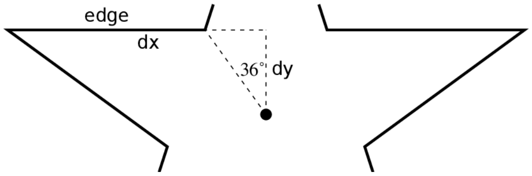
FIGURE 6-12 Function to create a five-pointed star

```
# Implementation notes: create_star
# --------------------------------
# The displacements dx and dy depend on the geometry of a pentagon.
# The details of that calculation are described in the text.

def create_star(size):
    """Creates a five-pointed star with its reference point at the center."""
    poly = GPolygon()
    dx = size / 2
    dy = dx * math.tan(18 * math.pi / 180)
    edge = dx - dy * math.tan(36 * math.pi / 180)
    poly.add_vertex(-dx, -dy)
    angle = 0
    for i in range(5):
        poly.add_polar_edge(edge, angle)
        poly.add_polar_edge(edge, angle + 72)
        angle -= 72
    return poly
```



Each of the points around the periphery of a five-pointed star forms an angle that is a tenth of a complete circle, which is 36 degrees. If you draw a line that bisects that angle—leaving 18 degrees on either side—that line will hit the geometric center of the star, forming the right triangle shown in the diagram. The value of `dy` is therefore equal to `dx` multiplied by the tangent of 18 degrees, as shown in the code.

The other tricky calculation is that of the edge length, which is illustrated in the following diagram:



To determine the value of `edge`, you need to subtract the dotted portion of the horizontal line from its entire length, which is given by `dx`. The length of the dotted portion is easily computed using trigonometry as `dy` multiplied by the tangent of 36 degrees. Once you have computed these values, the rest of the `create_star` function follows much the same pattern as the code for `create_hexagon`.

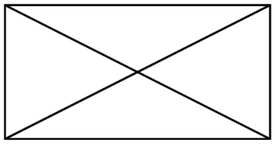
The `GCompound` class

The `GCompound` class makes it easy to assemble a collection of graphical objects into a single unit. As with `GPolygon`, calling `GCompound` creates an empty structure that you then have to fill by calling `add`, just as if you were adding those objects to the graphics window. Once you have assembled the objects, you can add the whole `GCompound` to the window, at which point it functions as a single object.

As a simple example, the function `create_crossed_box` shown in Figure 6-13 creates a `GCompound` consisting of a rectangle and the two diagonal lines that cross it. For example, the declaration

```
box = create_crossed_box(BOX_WIDTH, BOX_HEIGHT)
```

sets the variable `box` so that it holds a new `GCompound` object that looks like this:



Like the `GPolygon` class, the `GCompound` class defines its own coordinate system in which all coordinate values are expressed relative to a reference point. This design has two advantages. First, separating the process of defining the shape and setting its coordinates means that you can define a `GCompound` without having to know exactly where it will appear. That property is particularly useful if the location of an object in the graphics window depends on its size. Second, there are often more appropriate choices to use as a reference point than the conventional upper left corner. The

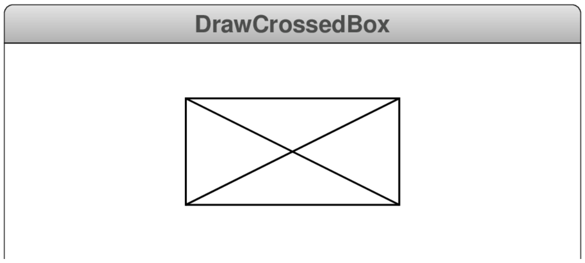
FIGURE 6-13 **Function to draw a GCompound consisting of a box and its diagonals**

```
def create_crossed_box(width, height):
    """Creates a crossed box with its reference point at the center."""
    compound = GCompound()
    compound.add(GRect(-width / 2, -height / 2, width, height))
    compound.add(GLine(-width / 2, -height / 2, width / 2, height / 2))
    compound.add(GLine(-width / 2, height / 2, width / 2, -height / 2))
    return compound
```

create_crossed_box function, for example, returns a GCompound in which the reference point is at the center, which is often a more convenient choice. You can then place the crossed box at the center of the window using the following code:

```
cx = gw.get_width() / 2
cy = gw.get_height() / 2
gw.add(box, cx, cy)
```

Executing these statements creates the following image on the graphics window:



Summary

In this chapter, you learned how to create interactive programs. The important points introduced in this chapter include:

- The Portable Graphics Library uses an *event-driven* model in which the user's actions generate events that occur asynchronously with respect to the operation of the program. Each event triggers a function call that responds to that event.
- Functions in Python are first-class values in the sense that they can be used in all the ways that any other value can. Functions can be assigned to variables, passed as parameters to other functions, and returned as a function result.
- The graphics library exports a large collection of methods that apply to every graphical object. A list of these methods appears in Figure 6-2 on page 184.
- Programs indicate their interest in responding to mouse events by calling the add_event_listener method on the graphics window.
- Mouse events are associated with an event type indicated by a string. The names of the different event types appear in Figure 6-3 on page 185.
- Each call to add_event_listener specifies the function that should respond to that type of event. These functions are generically known as *callback functions.*
- Callback functions used to respond to mouse events take a single parameter that includes information about the event. The only mouse-event properties used in this text are the methods get_x and get_y, which return the position in the window at which the mouse event occurred.

- Callback functions are conventionally declared within the body of an enclosing function so that the callback function has access to the local variables of the function in which the callback function is declared.
- A callback function can *examine* local variables in the enclosing context without taking any special action. If a callback function wants to *change* the value of a variable in the enclosing context, it cannot do so using an assignment statement, because Python would interpret the assignment as an implicit declaration of a new local variable. This text avoids that problem by embedding any shared variables that need to change inside the `GWindow` object.
- The `GWindow` class includes a method `get_element_at(x, y)` that returns the graphical object at that location in the window. If there is no object at that location, `get_element_at` returns the special value `None`.
- The usual strategy for implementing animation in the Portable Graphics Library is to use a *timer,* which executes a callback function after a specified delay. If the delay is 20 milliseconds or less, motion on the screen appears continuous.
- The `GWindow` class in the Portable Graphics Library exports two methods that support animation. The `set_timeout` method creates a *one-shot timer* that invokes a callback function after a specified delay. The `set_interval` method creates an *interval timer* that invokes the callback function repeatedly every time the delay time expires.
- The `set_timeout` and `set_interval` methods return a timer object, which is typically stored in a variable called `timer`. Invoking the `stop` method on the timer object turns off the timer and terminates the animation process.
- The `GArc` class makes it possible to display elliptical arcs defined by a bounding rectangle and two angles: a *start* angle that indicates where the arc starts and a *sweep* angle that indicates how far the arc extends. Filled arcs appear as wedges in which the endpoints of the arc are connected to the center.
- The `GPolygon` class makes it possible to display an arbitrary polygon. The `GPolygon` function itself creates an empty polygon; you create the actual polygon by calling some combination of the methods `add_vertex`, `add_edge`, and `add_polar_edge`.
- The `GCompound` class represents a graphical object that contains other graphical objects. Creating a `GCompound` allows the collection to be treated as a unit.
- Both the `GPolygon` and `GCompound` classes use an internal coordinate system relative to the object itself. This strategy makes it possible to create the object without knowing where it will appear in the window.

Review questions

1. In the context of the Portable Graphics Library, what is an *event?*

2. Are events in the Portable Graphics Library synchronous or asynchronous?

3. What reasons are offered in this chapter for the close association of graphical user interfaces and object-oriented programming?

4. Why are functions in Python said to be *first-class functions?*

5. True or false: In Python, you can pass a function as a parameter to some other function.

6. What are the two parameters to the `add_event_listener` method?

7. What event type do you use to respond to a mouse click?

8. What are the two methods used in this chapter to get more specific information about a mouse event?

9. What is a *callback function?*

10. How does a callback function usually share information with the function that defines it?

11. How does Python's reliance on implicit declarations complicate the definition of callback functions?

12. What strategy does this chapter recommend for ensuring that callback functions can change data values shared with other functions?

13. What is meant by the term *rubber-banding?*

14. What value does the `get_element_at` method return if no object exists at the specified location?

15. How does the `get_element_at` method decide which object to return if more than one object covers the specified location?

16. Describe in your own words the strategy for implementing animation in the Portable Graphics Library.

17. What is the difference between a *one-shot timer* and an *interval timer?* How do you specify which type you are creating?

18. How do you stop a timer?

19. Describe the significance of the *start* and *sweep* parameters in the call to the GArc function.

20. What does it mean if the sweep argument to the GArc function is negative?

21. Describe the arcs produced by each of the following calls to GArc, where cx and cy are the coordinates of the center of the window and r has the value 100:

    a) `GArc(cx, cy, 2 * r, 2 * r, 0, 270)`
    b) `GArc(cx, cy, 2 * r, 2 * r, 135, -90)`
    c) `GArc(cx, cy, 2 * r, 2 * r, 180, -45)`
    d) `GArc(cx, cy, 3 * r, r, -90, 180)`

22. How does the GArc class interpret the notion of a filled arc?

23. Describe the differences between the methods add_vertex, add_edge, and add_polar_edge in the GPolygon class.

24. Which of the three methods listed in the preceding question is conventionally used to add the first vertex to a GPolygon?

25. In your own words, describe the purpose of the GCompound class.

26. What advantages does the text cite for having the GPolygon and GCompound classes define their own reference point?

Exercises

1. Drawing on the print_function_table function for inspiration, implement a function

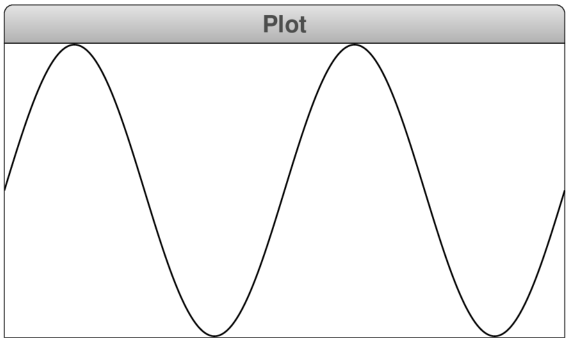
```
def plot(gw, fn, x_min, x_max, y_min, y_max)
```

that plots the function fn on the graphics window by creating small GLine segments and adding them to the graphics window. The parameters x_min, x_max, y_min, and y_max specify a translation between data values and window coordinates. The left edge of the window, for example, should correspond to the value x_min in the domain of the function.

For example, calling

```
plot(gw, math.sin, -2 * math.pi, 2 * math.pi, -1, 1)
```

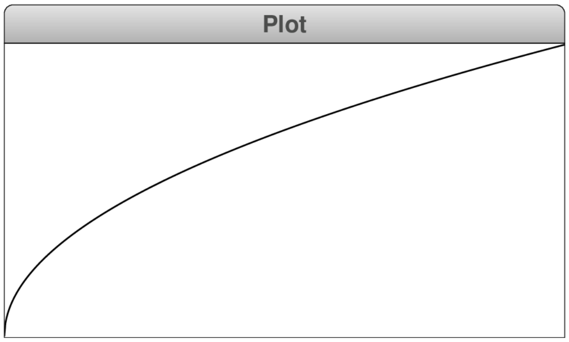
should generate a plot of the trigonometric sine function for values of *x* ranging from –2π to +2π and displayed so that the vertical space in the window runs from –1 at the bottom to +1 at the top (note that this interpretation requires you to flip Python’s coordinate system so that it matches the traditional Cartesian model in which *y* values increase as you move upward). After you make this call, the graphics window should look like this:



Similarly, calling

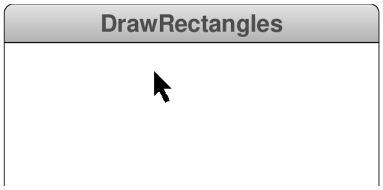
```
plot(gw, math.sqrt, 0, 4, 0, 2)
```

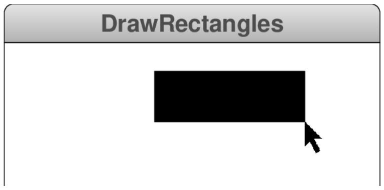
should plot the `math.sqrt` function on a graph that extends from 0 to 4 along the *x*-axis and from 0 to 2 along the *y*-axis, like this:



2. Modify the `DrawDots.py` program so that clicking the mouse draws a small × every time you click the mouse. The ×, which consists of two `GLine` objects, should be positioned so that the intersection appears at the point where the mouse was clicked.

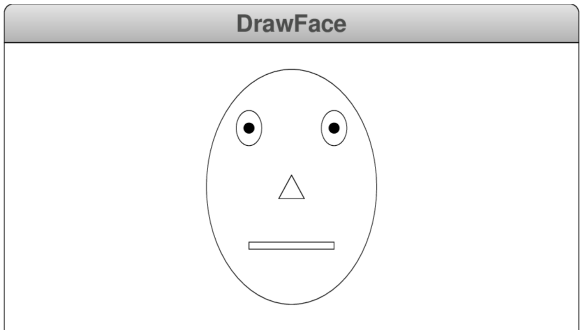
3. In addition to line drawings of the sort generated by the `DrawLines.py` program, interactive drawing programs allow you to add other shapes to the canvas. In a typical drawing application, you create a rectangle by pressing the mouse at one corner and then dragging it to the opposite corner. For example, if you press the mouse at the location in the left diagram and then drag it to the position where you see the cursor in the right diagram, the program creates the rectangle shown:



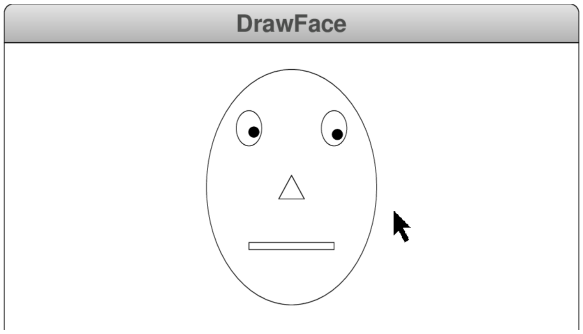


The rectangle grows as you drag the mouse. When you release the mouse button, the rectangle is complete and stays where it is. You can then go back and add more rectangles in the same way.

4. Use the `GOval`, `GLine`, and `GRect` classes to create a cartoon drawing of a face that looks like this:



Once you have this picture, add a callback function for the `"mousemove"` event so that the pupils in the eyes follow the cursor position. For example, if you move the cursor to the lower right side of the screen, the pupils should shift so that they appear to be looking at that point, as follows:



Although it doesn't matter much when the cursor is outside the face, it is important to compute the position of the pupil independently for each eye. If you move the mouse between the eyes, for example, the pupils should point in opposite directions so that the face appears cross-eyed.

5. Write a program that draws a filled black square in the center of the canvas. Once you have that part of the program working, animate your program so that the color of the square changes once a second to a new, randomly chosen color. Your program should run for a minute and then stop.

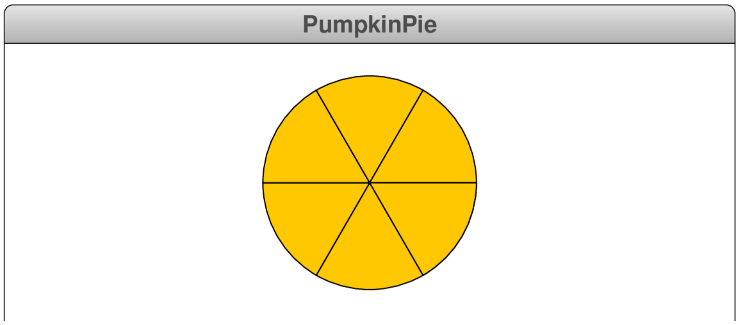
6. Using the `AnimatedSquare.py` program as a model, write a program `BouncingBall.py` that bounces a ball inside the boundaries of the graphics window. Your program should begin by placing a `GOval` in the center of the window to represent the ball. On each time step, your program should shift the position of the ball by `dx` and `dy` pixels, where both `dx` and `dy` initially have the value 1. Whenever the leading edge of the ball touches one of the boundaries of the window, your program should make the ball bounce by negating the value of `dx` or `dy`, as appropriate. Don't worry about getting your program to stop; just let it run until the user decides to terminate the program.

   Keep in mind that the values of `dx` and `dy` must be reassigned whenever a bounce occurs, which your program must detect inside the `step` function that runs each time the interval timer ticks. These variables must therefore be defined as components of the `GWindow` variable.

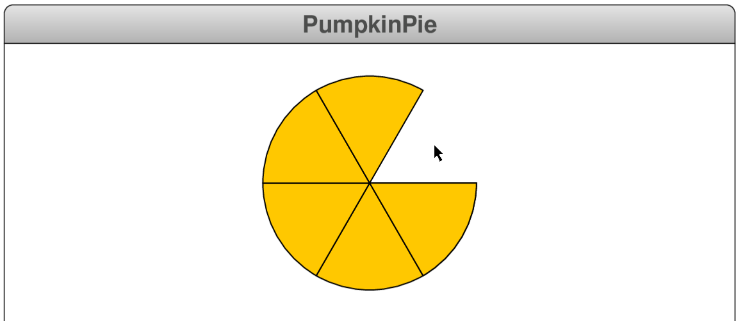
7. Rewrite the `BouncingBall.py` program from exercise 6 so that clicking the mouse starts and stops the motion of the ball. Although it is possible to implement this behavior by starting and stopping the timer, it is simpler to keep the timer running and use a Boolean flag variable called `ball_is_moving` to indicate whether the `step` function should update the position of the ball. Using this design, all you have to do in the `click_action` function is reverse the sense of this flag, changing `False` to `True` and vice versa.

8. Rewrite the `BouncingBall.py` program from exercise 7 so that the ball is implemented as a `GCompound` containing a `GOval` shifted by the radius of the ball in both the *x* and *y* directions. The advantage of making this change is that the coordinates of the `GCompound` now refer to the center of the ball, which makes the code to see whether the ball is bouncing more symmetrical and therefore easier to understand.

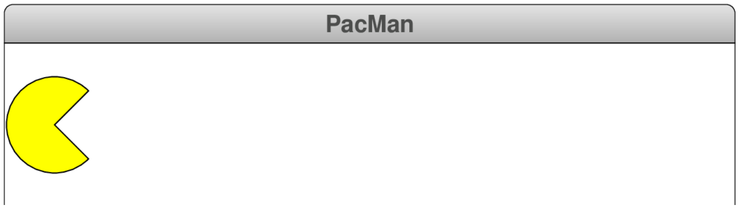
9. Write a program that draws a picture of a pumpkin pie divided into equal wedge-shaped pieces where the number of pieces is indicated by the constant `N_PIECES`. Each wedge should be a separate `GArc`, filled in orange and outlined in black. The following screen image, for example, shows the diagram when `N_PIECES` is 6.



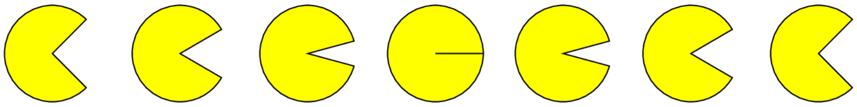
Once you have this display, add event processing to your application so that clicking on any of the wedges removes that wedge from the display. For example, if you click on the wedge in the upper right, the screen image should look like this:



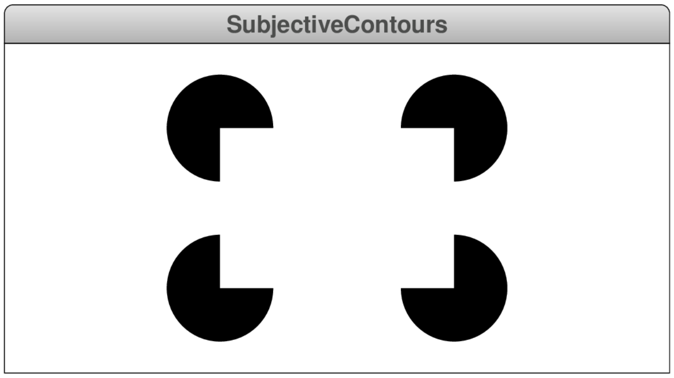
10. The title character in the PacMan series of games is easy to draw in Python using a filled `GArc`. As a first step, write a program that adds a PacMan figure at the left edge of the window, as follows:



Once you have this part working, add the code to make the PacMan figure move rightward until it reaches the right edge of the graphics window. As PacMan moves, your program should change the start and sweep angles so that the mouth appears to open and close as shown in the following image sequence:

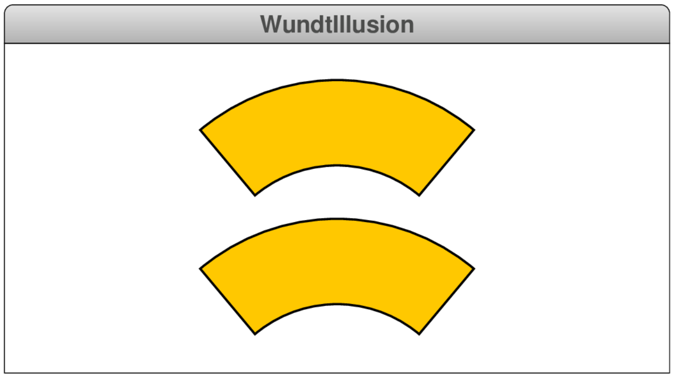


11. The PacMan shape appears in an optical illusion called a ***subjective contour,*** popularized in an article by the Italian psychologist Gaetano Kanizsa in the April 1976 issue of *Scientific American,* which includes this image:



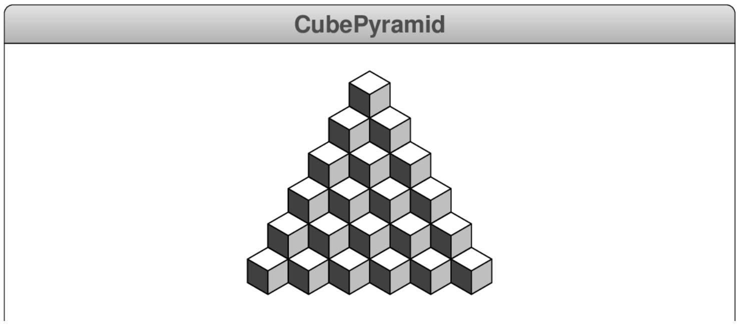
Although the simplest way to produce this picture is to draw a white rectangle on top of four complete circles, a skeptic might claim that the color of the rectangle is brighter than its background. Make it impossible to defend this claim by drawing this figure using only the four filled arcs.

12. Another illusion that uses filled arcs is the ***Wundt illusion,*** first described by Wilhelm Max Wundt in 1898.



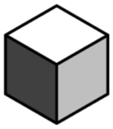
In this illusion, the lower curve looks longer than the upper curve, although the two are in fact the same size. Write a program that draws these segments using the graphics library. To do so, you need to draw a filled arc, overlay it with a smaller arc filled in white, and then complete the border with an unfilled arc.

13. Write a program that draws the following optical illusion on the graphics window:



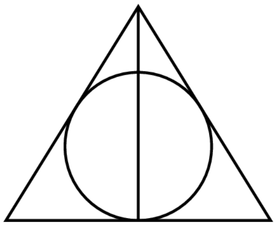
The illusion arises from the fact that it is possible to see the white surfaces as either the tops or the bottoms of cubes stacked to form a pyramid.

Each of the individual cubes is composed of three diamond-shaped polygons whose sides have different fill colors, as follows:



In writing this exercise, you should create a function that returns one of these cubes as a `GCompound` and then assemble the pyramid from those compounds.

14. In J. K. Rowling's *Harry Potter and the Deathly Hallows,* those who believe in the legend named in the title recognize one another through a symbol that combines three elements—a triangle representing the cloak of invisibility, a circle representing the stone of resurrection, and a line representing the elder wand—superimposed as follows:



Write a function `create_deathly_hallows_symbol` that takes the width and height of the figure and returns a `GCompound` that includes all three of these elements. The triangle should be a `GPolygon`, the circle should be a `GOval`, and the line should be a `GLine`. The geometry is straightforward for both the line and the triangle, but rather complicated for the circle, which must exactly touch the edges of the triangle. Although you could figure out the necessary relationships by using the Pythagorean theorem, you can instead simply use the following formula for the radius *r* as a function of the width *w* and the height *h:*

$$r = \frac{w\sqrt{4h^2 + w^2} - w^2}{4h}$$

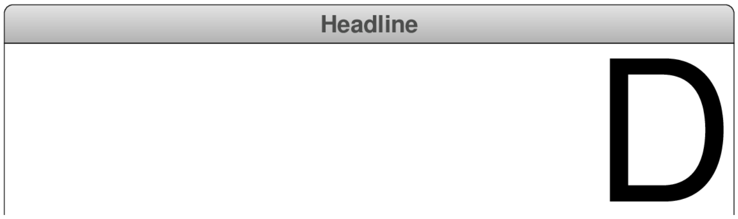
Use the `create_deathly_hallows_symbol` function to write a program that displays the symbol in the center of the window. Once you've done that, add the code needed to let the user drag the symbol around the window.

15. In New York's Times Square, you can get the news of the day by watching headlines on large display screens that show a single line of text. The headline initially begins to appear at the right edge of the screen and then moves quickly from right to left. Your job in this exercise is to write a program that simulates this type of headline display by moving a `GLabel` across the screen.

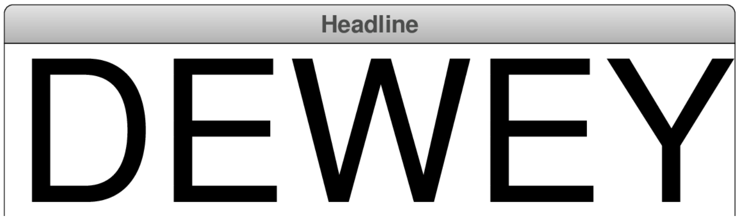
    Suppose, for example, that you want to use your program to display the famous *Chicago Tribune* headline from when the paper incorrectly called the result of the 1948 presidential election:

    DEWEY DEFEATS TRUMAN

    Your program should create a `GLabel` containing the headline and then position it so that the entire text of the label is clipped beyond the right edge of the screen. Your program should then implement a timer-based graphical animation that moves the `GLabel` a few pixels to the left on each time step. After a few time steps, the display will show the first letter of the headline, as follows:



    The headline continues to scroll across the screen, so that a few seconds later the entire first word is visible:



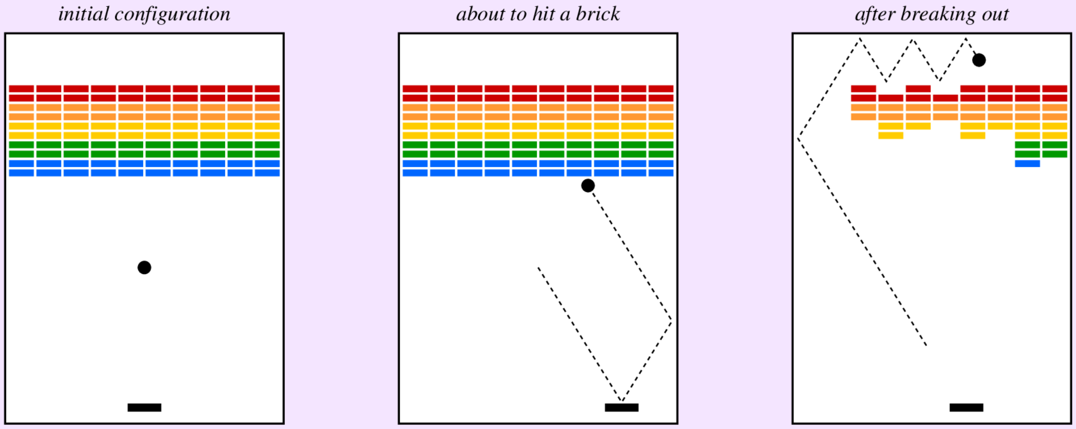
As the label continues to scroll, letters will disappear off the left edge of the screen as new letters appear on the right. Your program should continue to scroll letters toward the left until the entire `GLabel` disappears from view.

16. Write a program to play the classic arcade game of Breakout, which was developed in 1976 by Steve Wozniak, who would later become one of the founders of Apple. In Breakout, your goal is to clear a collection of bricks by hitting each of them with a bouncing ball.

    The initial configuration of the Breakout game appears in the leftmost diagram in Figure 5-14. The colored rectangles in the top part of the screen are bricks, two rows each of red, orange, yellow, green, and blue. The slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension, but moves back and forth across the screen along with the mouse until it reaches the edge of its space.

    A complete Breakout game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world. Thus, after two bounces—one off the paddle and one off the right wall—the ball might have the trajectory shown in the middle diagram.

FIGURE 6-14 Selected configurations in the Breakout game


initial configuration

about to hit a brick

after breaking out

As you can see from the middle diagram, the ball is about to collide with one of the bricks on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears.

The play continues in this way until one of the following conditions occurs:

- The ball hits the lower wall, which means that you must have missed it with the paddle. In this case, the turn ends and the next ball is served. After three turns, the game is over, and the player loses.
- The last brick is eliminated, in which case the player wins.

After all the bricks in a particular column have been cleared, a path will open to the top wall, as shown in the rightmost diagram in Figure 6-14. When this delightful situation occurs, the ball will often bounce back and forth several times between the top wall and the upper line of bricks without the user ever having to worry about hitting the ball with the paddle. This condition is called “breaking out.” It is important to note that, even though breaking out is a very exciting part of the player’s experience, you don’t have to do anything special in your program to make it happen. The game operates the same as always: balls bounce off walls, collide with bricks, and obey the laws of physics.

CHAPTER 7
Strings

> The work [of conducting the census should] be done so far as possible by mechanical means. In order to accomplish this the records must be put in such shape that a machine could read them. This is most readily done by punching holes in cards.
>
> — Herman Hollerith, *An Electric Tabulating System,* 1889

Herman Hollerith (1860–1929)

The idea of encoding text in machine-readable form dates back to the nineteenth century and the work of the American inventor Herman Hollerith. After studying engineering at City College of New York and the Columbia School of Mines, Hollerith spent a couple of years working as a statistician for the U.S. Census Bureau before accepting a teaching position at MIT. While at the Census Bureau, Hollerith had become convinced that the data produced by the census could be counted more quickly and accurately by machine. In the late 1880s, he designed and built a tabulating machine that was used to conduct the 1890 census in record time. The company he founded to commercialize his invention, originally called the Tabulating Machine Company, changed its name in 1924 to International Business Machines (IBM). Hollerith's card-based tabulating system pioneered the technique of textual encoding described in this chapter—a contribution that was reflected in the fact that early versions of the FORTRAN language used the letter `H` (for Hollerith) to indicate text data.

Although you have been using strings ever since Chapter 1, you have only scratched the surface of what you can do with string data. This chapter introduces the features available in Python's built-in string type, which provides a convenient abstraction for working with strings of characters. Understanding how to work with strings will make it much easier to write interesting applications. Before considering the details of how strings work, however, it helps to take a step back and look at how computers store data in the first place.

7.1 Binary representation

Today's computers represent information in a simple but powerful form that allows information—no matter how complex—to be stored as a sequence of primitive values that can exist in only one of two possible states. Each of those primitive values is called a ***bit.***

The interpretation of the values for each bit depends on how you choose to view the underlying information. If you think of the bits that form the internal circuitry of the machine as tiny light switches, you might label those states as *off* and *on*. If you think of each bit as a logical value, you might instead use the Boolean labels *false* and *true.* However, because the word *bit* comes from a contraction of the term *binary digit,* it is more common to label those states as **0** and **1**, which are the digits of the binary number system on which computer arithmetic is based.

Binary notation

Leibniz

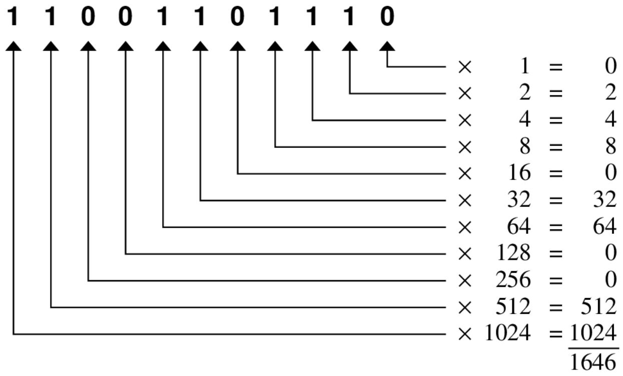
The idea of writing numbers in binary notation predates the development of the electronic computer by thousands of years. The Chinese *I Ching* from ... uses binary notation to number the 32 different symbols. The German mathematician Gottfried Wilhelm von Leibniz (1646–1716) offered a detailed account of the binary system in a paper published by the French Royal Academy of Science in 1703. In that paper (which cites the *I Ching* as an earlier source), Leibniz writes:

> Ordinary arithmetic calculation is performed following a progression by tens. One uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, which signify zero, one, and the following numbers up to nine, inclusive. On going up to ten, one starts again, and writes *ten* as 10; ten times ten, or *one hundred,* as 100; ten times one hundred, or *one thousand,* as 1000; and ten times a thousand as 10000. And so on.
>
> But instead of the progression by tens, I have used for several years the simplest progression of all, which goes by twos, which I find to be the perfection of the science of numbers. I therefore do not use any characters other than 0 and 1, and on going up to two, I start again. That is why *two* is written here as 10; and two times two or *four* as 100; and two times four or *eight* as 1000 . . .

Leibniz's second paragraph describes the binary system as "the simplest progression of all." Each digit in a binary number counts for twice as much as its neighbor on the right. That rule makes it easy to translate a number written in binary back to its decimal equivalent: all you need to do is add the place values of each digit in the number. For example, if Leibniz were to use binary notation to represent the year of his birth, he would write the number like this:

1 1 0 0 1 1 0 1 1 1 0

The following diagram shows that this value indeed corresponds to the value 1646:



For the most part, numeric representations in this book use decimal notation for readability. If the base is not clear from the context, the text follows the usual convention of using a subscript to denote the base. For example, the equivalence of the binary value 11001101110 and the decimal value 1646 can be made explicit by writing the numbers like this:

$11001101110_2 = 1646_{10}$

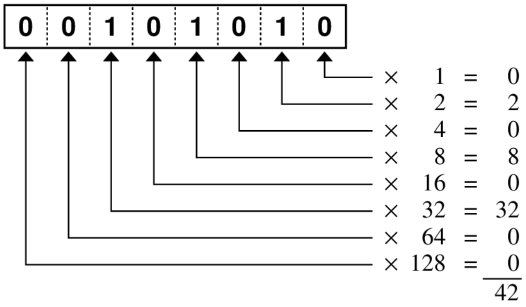
Storing integers as sequences of bits

The binary representation described by Leibniz makes it easy to store integers as a sequence of individual bits. In modern computer hardware, individual bits are collected together into larger units that are then treated as integral units of storage. The smallest such combined unit is called a ***byte,*** which consists of eight bits. Bytes are then assembled into larger structures called ***words,*** where a word is usually defined to be the size required to hold an integer value of the type most appropriate for the hardware. Today, machines typically organize their memory into words that are either four or eight bytes long (32 or 64 bits).

To get a sense of how computers can store integers internally, consider the byte containing the following binary digits:

0	0	1	0	1	0	1	0

That sequence of bits represents the number forty-two, which you can verify—just as Leibniz would have done—by calculating the contribution for each of the individual bits, as follows:



Bytes can store integers between 0 and 255, which is $2^8 - 1$. Numbers outside this range must be stored in larger units that use more bits of memory.

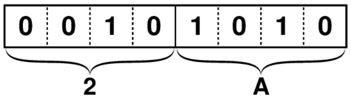
Hexadecimal notation

Although the bit diagrams make it clear how computers store integer values internally, these diagrams also demonstrate the fact that writing numbers in binary form is terribly inconvenient. Binary numbers are cumbersome, mostly because they tend to be so long. Decimal representations are intuitive and familiar but make it harder to understand how the number translates into bits.

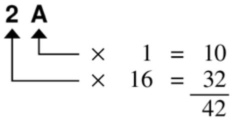
For applications in which it is useful to understand how a number translates into its binary representation without having to work with binary numbers that stretch all the way across the page, computer scientists use ***hexadecimal*** (base 16) notation instead. In hexadecimal notation, there are sixteen digits that represent the values from 0 to 15. Although the decimal digits 0 through 9 are perfectly adequate for the first ten digits, classical arithmetic does not define the extra symbols you need to represent the remaining six. Computer science traditionally uses the letters **A** through **F** for this purpose, as follows:

A	=	10
B	=	11
C	=	12
D	=	13
E	=	14
F	=	15

What makes hexadecimal notation useful is the fact that you can easily convert between hexadecimal values and the underlying binary representation. All you need to do is combine the bits into groups of four. For example, the number forty-two can be converted from binary to hexadecimal like this:



The first four bits represent the number 2, and the next four represent the number 10. Converting each of these to the corresponding hexadecimal digit gives **2A** as the hexadecimal form. You can then verify that this number still has the value 42 by adding up the digit values, as follows:



As noted earlier, the text follows the convention of using a subscript to denote the base if it is not clear from context. Thus, the number forty-two can be written down like this in decimal, binary, octal, and hexadecimal:

$$42_{10} = 00101010_2 = 52_8 = 2\text{A}_{16}$$

Python allows you to write integer constants in any of these bases. Decimal numbers require no special marker, but you can specify a number in binary, octal, or hexadecimal by adding the prefix `0b`, `0o`, or `0x` at the beginning of a string of digits. Thus, you can represent the integer forty-two in Python in any of these ways:

```
42    0b00101010    0o52     0x2A
```

The most important thing to remember is that the number itself is always the same; the numeric base affects only the representation. Forty-two has an intrinsic meaning that is independent of the base, which is perhaps easiest to see in the representation an elementary school student might use:

The number of tick marks in this representation is forty-two. The fact that a number is written in binary, decimal, or any other base is a property of the representation, not of the number itself. Numbers do not have bases; representations do.

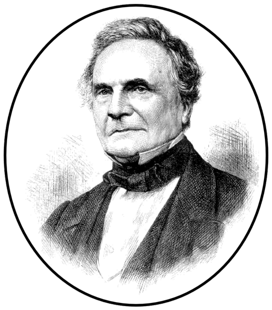
Representing nonnumeric data

Although the discussion so far has focused on how computers store numbers, this chapter is about strings, which are an important example of ***nonnumeric data.*** The challenge in having computers represent nonnumeric data lies in finding a way to store that information inside the computer.

The simplest strategy for representing nonnumeric data is to assign numbers to the individual data values you need to represent. For example, the conventional way to represent the months of the year—even without a computer—is to give each month a number: January has the value 1, February has the value 2, and so on, up to December, which has the value 12. This strategy is called ***enumeration.***

Once you have enumerated a set of values, you can represent those values in memory by using the appropriate numeric code. For example, the numeric value 12 corresponds to the month of December. Internally, that value is stored as an integer expressed—as you saw in the preceding sections—as a sequence of binary digits. There is no indication in the hardware as to whether that value represents the integer 12 or the numeric representation for the month of December. The meaning of a particular value depends on how it is used. If the program uses the value arithmetically, it is interpreted as the integer 12. If it instead uses that value to select from a list of month names, that value indicates December. In either case, the number stored inside the computer is exactly the same.

Ada Lovelace



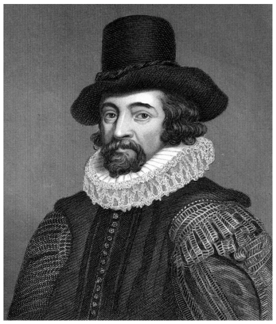
Charles Babbage

The strategy of using numbers to represent nonnumeric data is one of the most important ideas in the history of computation. One of the clearest and earliest expositions of that idea comes from Ada Lovelace, daughter of the poet Lord Byron and his wife Anna Isabella Byron. In the 1840s, Lady Lovelace collaborated with the English mathematician and inventor Charles Babbage on the design of his Analytical Engine, a calculating machine that anticipated several essential features of modern computers, including the ability to solve different tasks by changing its programming. Indeed, much of what we know about the Analytical Engine—which sadly was never completed—comes from Lovelace's translation of a detailed description of Babbage's work by the Italian engineer Luigi Menabrea. Her translation, entitled *Sketch of the Analytical Engine Invented by Charles Babbage, Esq.,* was published in 1843, along with her explanatory notes that were almost three times as long as the original paper. Lovelace recognized that the algebraic patterns for which the Analytical Engine was designed could be extended to include concepts beyond simple numbers. Her notes envision a world of possibilities for the Analytical Engine that someday "might compose elaborate and scientific pieces of music of any degree of complexity or extent."

In an interview for a film about Ada Lovelace's life and work, Doron Swade, who led the effort to rebuild Babbage's earlier Difference Engine for the Science Museum in London, offers the following description of Ada's contribution:

> Ada saw something that, in some sense, Babbage failed to see. In Babbage's world, his engines were bound by number. . . . What Lovelace saw—what Ada Byron saw—was that *number* could represent entities other than *quantity.* So, once you had a machine for manipulating numbers, if those numbers represented other things—letters, musical notes—then the machine could manipulate symbols of which number was one instance."

Representing characters


Francis Bacon

The primitive elements of string data are individual characters. Like the months of the year, characters can be represented inside the computer by assigning each character a numeric code. You could, for example, assign successive integers to represent each of the letters in the alphabet, using 0 for the letter **A**, 1 for letter **B**, and so on. In 1605, the English philosopher and scientist Francis Bacon did precisely that when he devised a technique for encoding messages that is now known as ***Bacon's cipher.*** What is, however, even more astonishing is that Bacon based his cipher on the binary representation of these numbers, almost a century before Leibniz published his paper on binary arithmetic. Bacon's cipher, however, was not used in practice and had little or no influence on the later development of computation.

Émile Baudot

The first binary encoding scheme for characters used extensively in practice was the ***Baudot code,*** which was invented in 1870 by the French engineer Émile Baudot, one of the pioneers of the telegraph. In Baudot's scheme, each of the 26 letters was assigned a numeric code. The encoding also included a few special characters to represent the space character, the two characters that telegraph printers used to designate the end of a line, and transitions to an alternate character set used for digits and punctuation. The letters of the alphabet did not appear in order, but were instead chosen so that the most common letters, such as **E** and **T**, would require pressing just one of the five keys on the input device that Baudot designed.

The fact that the letters do not appear consecutively in the Baudot code does not make the encoding scheme any less effective. The only essential characteristic of an encoding system is that the sender and receiver agree on how to convert letters to numeric codes. The need for a common encoding shared by senders and receivers increases the importance of standardization. As long as all telegraph operators used the same code, they were able to communicate with one another.

In the early years of the computing industry, standardization was complicated by the existence of incompatible character encodings. The American Standards Association (now known as the American National Standards Institute or ANSI) began work on a standardized character encoding in 1960, which was formalized in 1963 as the ***American Standard Code for Information Interchange*** or ***ASCII.*** Early IBM machines used a different character set derived from the coding system used for punched cards. That early character set evolved into a competing standard called the ***Extended Binary Coded Decimal Interchange Code*** or ***EBCDIC.*** Over time, ASCII and its successors have become the dominant standard in the industry.

In its original design, ASCII contained 128 characters, which is enough to store the uppercase and lowercase letters of the Latin alphabet, the standard decimal digits, a variety of punctuation symbols, and a set of nonprinting characters called ***control characters.*** The characters in the original ASCII set appear in Figure 7-1. The gray

FIGURE 7-1 **The first 128 characters in ASCII**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0*x*	`\0`								`\b`	`\t`	`\n`	`\v`	`\f`	`\r`		
1*x*																
2*x*	*space*	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3*x*	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4*x*	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5*x*	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6*x*	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7*x*	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

boxes in the table correspond to control characters that have lost their significance over time. The few remaining control characters recognized by Python are indicated using a backslash (`\`) followed by a letter that suggests that character's function. For example, the character `\n` represents the ***newline character,*** which marks the end of a line. None of the other control characters are used in this book.

The characters in Figure 7-1 are arranged according to their internal values, which are expressed in hexadecimal. The character **A**, for example, appears in the row labeled **4***x* and the column labeled **1**, so its internal representation is **41**$_{16}$, which is the decimal number 65. There is no need to learn these values, although certain patterns are important. This text, for example, relies on the following properties:

- The digit characters are consecutive.
- The uppercase and lowercase letters form two consecutive sequences.

The ASCII coding system quickly proved to be inadequate as computing expanded into the global environment. With the advent of the World Wide Web in the 1990s, it became necessary to expand the encoding system to embrace a broader collection of languages. The result of that expansion was a new standard called ***Unicode,*** which supports a much larger set of characters. The version of Unicode implemented in Python allows for 1,114,111 (110000_{16}) characters.

Converting between numeric codes and characters

Python makes it easy to convert back and forth between characters and their underlying numeric representation in Unicode. The built-in function `chr` takes an integer value and returns a one-character string that contains the character with that code. For example, you can see from Figure 7-1 that calling `chr(0x41)`—or, equivalently, `chr(65)`—returns the string `"A"`. The built-in function `ord` applies the same conversion in the opposite direction. It takes a one-character string and returns the corresponding Unicode value. Thus, calling `ord("A")` returns the integer 65.

As a modern-day programmer, you should never have to know any of the Unicode values and should certainly not write programs that use numeric values to represent characters. At the same time, it is important to know that characters have a numeric representation and that you can use the `ord` function to obtain it. Later examples in this chapter, for example, will need to use the character code for `"A"`. Those programs will not, however, use the explicit value 65, which would make the code difficult to read. Those programs will instead use `ord("A")` to indicate this value.

7.2 String functions and operators

From the brief discussion of strings in Chapter 1, you already know that Python uses the + operator to signify concatenation, which consists of joining the strings together end to end. You also know that you can determine the length of a string by calling the built-in function `len`. For example, if `ALPHABET` is defined as

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

calling `len(ALPHABET)` returns the value 26. Similarly, if you define a variable `empty` using the declaration

```
empty = ""
```

the expression `len(empty)` has the value 0. The string containing no characters at all, which comes up frequently in programming, is called the ***empty string.***

In addition to the these operations, you have also used the relational operators (==, `!=`, <, <=, >, and >=) to compare string values. For example, the programs in Chapter 7 that used a blank line to signal the end of the input used the == operator to test whether the variable `line` was equal to the empty string. The relational operators compare strings using ***lexicographic order,*** which is similar to traditional alphabetical order but which uses the underlying Unicode values of each character to make the comparison. Lexicographic order means that case is significant, so `"a"` is not equal to `"A"`. In lexicographic order, `"a"` is greater than `"A"` because the Unicode value for a lowercase **a** (**61**$_{16}$ or 97) is greater than the Unicode value for an uppercase **A** (**41**$_{16}$ or 65).

Beyond these operations you have already seen, Python defines several other tools for working with strings. The sections that follow describe the built-in operators and functions organized into logically related groups. The methods that apply to string objects are introduced in section 7.4.

Repeating a string

In Python, you can use the * operator to specify a string composed by concatenating multiple copies of a shorter string. For example, the expression

```
"ab" * 3
```

returns the six-character string "ababab". Python's use of the * operator seems appropriate, not only because it suggests multiplicity but also because it corresponds to the mathematical definition of multiplication as repeated addition, as follows:

```
"ab" * 3   is the same as   "ab" + "ab" + "ab"
```

As it does in arithmetic expressions, the * operator takes precedence over the + operator, so that the expression

```
"Rose" + " is a rose" * 3 + "."
```

performs the * operator first and therefore returns the string

```
"Rose is a rose is a rose is a rose."
```

This sentence appears in Gertrude Stein's poem "Sacred Emily" from 1913.

Selecting an individual character

You can select an individual character from a Python string by enclosing its index in square brackets. Character positions in a string are numbered starting from 0. For example, the characters in the constant ALPHABET defined on the previous page are numbered as in the following diagram:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

The expression ALPHABET[10], for example, is the one-character string "K".

It is often useful, however, to specify a character by indicating how far that character is from the end of the string. Python allows a string index to be negative, in which case the position is determined by counting backwards from the end. The characters in ALPHABET can therefore also be numbered like this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
–26	–25	–24	–23	–22	–21	–20	–19	–18	–17	–16	–15	–14	–13	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

Using this numbering scheme, the expression ALPHABET[-3] selects the third character from the end, or "X".

Negative index numbers are never necessary but in some cases turn out to be convenient. In particular, it is more concise to select the last character in the string `s` by writing `s[-1]` than the longer and less evocative `s[len(s) - 1]`.

Slicing

While concatenation makes longer strings from shorter pieces, you often need to do the reverse: separate a string into the shorter pieces it contains. A string that is part of a longer string is called a ***substring.*** Python makes it easy and convenient to extract substrings by extending the square-bracket notation for character selection so that you can specify not only a single index position but also a range of index positions marking the boundaries of a substring. In Python, using square brackets to select a range of characters is called ***slicing.***

In its simplest form, a slice in Python is written using two indices separated by a colon inside the square brackets, like this:

str[*start*:*limit*]

As with the `range` function defined in Chapter 2, the index expressions inside the square brackets specify a half-open interval in the sense that the index range includes *start* but stops just before *limit.* Thus, the expression

```
ALPHABET[1:4]
```

returns the three-character substring `"BCD"`, which starts at index position 1 and ends just before index position 4. Similarly, the expression

```
ALPHABET[1:-1]
```

returns the 24-character substring `"BCDEFGHIJKLMNOPQRSTUVWXY"`, which stops just short of the index position indicated by −1, which uses negative indexing to specify the last character in the string.

Python allows you to leave out the index expressions on either side of the colon. If the first index is missing, it is assumed to be the beginning of the string, so that

```
ALPHABET[:5]
```

selects the substring `"ABCDE"` consisting of the first five characters in `ALPHABET`. If the second expression is missing, it is taken to be the length of the string. Thus,

```
ALPHABET[13:]
```

selects the substring `"NOPQRSTUVWXYZ"`, which contains the characters from index position 13 up to the end of the string.

The square-bracket syntax also accepts an optional third component, as follows:

str[*start*:*limit*:*stride*]

When the *stride* component appears, it indicates the distance between characters chosen for inclusion in the substring. For example, the expression

```
ALPHABET[9:20:5]
```

selects characters from `ALPHABET` starting at position 9, ending before position 20, and moving ahead five characters on each stride. This expression therefore selects the characters in index positions 9, 14, and 19 to produce the string `"JOT"`. The expression

```
ALPHABET[::2]
```

uses the default values for *start* and *limit* but uses a *stride* of 2 to select every other character from `ALPHABET`, which produces the string `"ACEGIKMOQSUWY"`.

As with the built-in `range` function, the *stride* component can be negative, in which case the characters are selected by counting backwards through the string. When the *stride* value is negative, the *start* component defaults to the last character in the string, and the *limit* component defaults to the beginning of the string. For example, the expression

```
ALPHABET[::-1]
```

returns the characters in `ALPHABET`, chosen from back to front to produce the 26-character string `"ZYXWVUTSRQPONMLKJIHGFEDCBA"`.

Programmers entranced by Python's particularly succinct style of expression are often tempted to use this form of slicing to reverse a string. Doing so, however, makes the resulting program difficult to follow for programmers unfamiliar with this Python-specific idiom. One way to restore the desired readability is to embed this operation in a function whose name makes the effect of the operation clear, like this:

```
def reverse_string(s):
    return s[::-1]
```

Although some readers may be mystified as to how this implementation achieves the desired effect, those readers can use the name of the function to understand the program on a more holistic level.

7.3 Common string patterns

Although section 7.2 gives you a sense of what string operators Python offers, the discussion gives you little guidance as to how you can use these operators most effectively. When you are learning to program, it is often easier to ignore as many details as possible and instead write your programs by relying on code patterns that implement common operations. The two most important string patterns are iterating through the characters in a string and growing a string by concatenation. The sections that follow describe these patterns.

Iterating through the characters in a string

When you work with strings, one of the most important patterns involves iterating through the characters in a string. In its simplest form, which you have already seen in Chapter 1, iterating through the characters in a string requires the following code:

```
for ch in s:
    . . . body of loop that uses the character ch . . .
```

On each loop cycle, the variable `ch` is bound to a one-character string chosen from successive index positions of the string `s`. The body of the loop then uses that character to perform some computation. You can, for example, count the number of spaces in a string using the following function:

```
def count_spaces(s):
    ns = 0
    for ch in s:
        if ch == " ":
            ns += 1
    return ns
```

Growing a string through concatenation

The other string pattern that is important to memorize involves creating a new string one character at a time. The details of the loop depend on the application, but the general pattern for creating a string by concatenation looks like this:

```
result = ""
for whatever loop header line fits the application:
    result += the next piece of the result
```

For example, the `n_copies` function returns a string consisting of `n` copies of `s`, achieving the effect of the expression `s * n` without using the `*` operator:

```
def n_copies(n, s):
    result = ""
    for i in range(n):
        result += s
    return result
```

Combining the iteration and concatenation patterns

Many string-processing functions use the iteration and concatenation patterns together. For example, the following function returns a copy of the string s with all spaces removed:

```
def remove_spaces(s):
    result = ""
    for ch in s:
        if ch != " ":
            result += ch
    return result
```

As a second example, the following function offers another strategy—arguably more readable but certainly less efficient—for implementing the reverse_string function first defined on page 228:

```
def reverse_string(s):
    result = ""
    for ch in s:
        result = ch + result
    return result
```

This implementation builds up the reversed string by concatenating each character onto the front of the existing result. For example, calling reverse("stressed") assigns the following values to result as it goes through the for loop:

```
""
"s"
"ts"
"rts"
"erts"
"serts"
"sserts"
"esserts"
"desserts"
```

7.4 String methods

Strings in Python support a range of operations beyond those you have seen so far in this chapter. Those operations, however, are implemented in a slightly different style. In Python, the built-in string type is implemented as a class similar to those in the Portable Graphics Library. Like all classes in an object-oriented language, the string class implements most of its operations in the form of methods that are applied to string objects.

As with the methods you've used in the graphics library, method calls on the string class are written using the receiver syntax:

receiver.*name*(*arguments*)

Figure 7-2 lists the most common methods that Python defines as part of its built-in string class. These methods under the first four subheadings in Figure 7-2 are explored in detail in the sections that follow. The methods under the subheading "Splitting and joining strings" are discussed in Chapter 8.

Finding patterns

From time to time, you will find it useful to search a string to see whether it contains a particular substring. To support such search operations, Python's string class exports a method called `find`, which comes in two forms. The simplest form of the call is

`s.find(`*pattern*`)`

where *pattern* is the substring you're looking for. When called, the `find` method searches through `s` looking for the first occurrence of the pattern. If the search value is found, `find` returns the index position at which the match begins. If the character does not appear before the end of the string, `find` returns –1.

The `find` method takes optional `start` and `end` arguments that limit the range of the search. These arguments are illustrated by the following examples, where `s` contains the string `"hello, world"`:

```
s.find("o")       →        4
s.find("o", 5)    →        8
s.find("o", 5, 7)   →     −1
```

The Python string class also includes an `rfind` method that works like `find`, except that it searches backward from the end of the specified range for the last instance of the pattern.

FIGURE 7-2 Common methods in Python's string class

Finding patterns

s.find(*pattern*, *start*, *end*)	Searches the string *s* for the string *pattern* in the index range [*start*:*end*], where *start* and *end* default to 0 and the length of the string. The function returns the first index of *pattern*, or –1 if not found.
s.rfind(*pattern*, *start*, *end*)	Searches backward in *s* for the last instance of *pattern* in the index range [*start*:*end*], which have the same defaults as `find`. The function returns the last index of *pattern*, or –1 if it is not found.
s.startswith(*prefix*)	Returns `True` if *s* starts with the characters in *prefix*.
s.endswith(*suffix*)	Returns `True` if *s* ends with the characters in *suffix*.

Creating transformed strings

s.lower()	Returns a copy of *s* converting all letters to lowercase.
s.upper()	Returns a copy of *s* converting all letters to uppercase.
s.capitalize()	Returns a lowercase copy of *s* with the first letter capitalized.
s.lstrip()	Returns a copy of *s* after removing whitespace from the left end.
s.rstrip()	Returns a copy of *s* after removing whitespace from the right end.
s.strip()	Returns a copy of *s* after removing whitespace from both ends.
s.replace(*old*, *new*)	Returns a copy of *s* replacing all instances of *old* with *new*.

Testing for character properties

s.isalpha()	Returns `True` if *s* is nonempty and contains only letters.
s.isdigit()	Returns `True` if *s* is nonempty and contains only digits.
s.isalnum()	Returns `True` if *s* is nonempty and contains only letters or digits.
s.islower()	Returns `True` if *s* has at least one letter and all letters are lowercase.
s.isupper()	Returns `True` if *s* has at least one letter and all letters are uppercase.
s.isspace()	Returns `True` if *s* is nonempty and contains only whitespace characters.

Formatting methods

s.ljust(*width*)	Returns `s` flush to the left in a field of the specified width.
s.rjust(*width*)	Returns `s` flush to the right in a field of the specified width.
s.center(*width*)	Returns `s` centered in a field of the specified width.
s.format(. . .)	Returns a copy of *s* after inserting formatted values.

Splitting and joining strings

s.split(*pattern*)	Splits the string into a list of strings by dividing it at *pattern*.
s.splitlines()	Splits a multiline string into a list of the individual lines.
sep.join(*list*)	Joins the elements of *list* into a string using *sep* to separate the elements.

The startswith method returns True if the receiver string begins with the specified prefix. For example, the expression

```
answer.startswith("y") or answer.startswith("Y")
```

is True if answer begins with either "y" or "Y". The endswith method is symmetric and returns True if the string ends with the specified suffix.

Creating transformed strings

Python's string class exports several methods for changing the case of characters within a string. The method upper, for example, returns a string in which all lowercase characters in the original string have been converted to their uppercase equivalents. Thus, if s contains the string "hello, world", calling s.upper() returns "HELLO, WORLD". The lower method performs the case conversion in the opposite direction. If the constant ALPHABET is defined as shown on page 225, calling ALPHABET.lower() returns "abcdefghijklmnopqrstuvwxyz". The capitalize method returns a string in which the first character is capitalized and all other letters are converted to their lowercase forms.

The "Creating transformed strings" subheading of Figure 7-2 lists several other methods that often come in handy. The lstrip, rstrip, and strip methods return a copy of the receiver string after removing all ***whitespace characters*** ("invisible" characters such as spaces or tabs) from one or both ends of the string. The replace method returns a copy of the receiver string after replacing all instances of the first argument with the second. This function makes it possible to simplify the definition of remove_spaces from page 230 as follows:

```
def remove_spaces(s):
    return s.replace(" ", "")
```

It is important to remember that the methods in Python's string class do not change the value of the receiver but instead return an entirely new string. Thus, calling s.upper() doesn't change the value of the variable s. If you want to change the value of s to its uppercase equivalent, you need to use an assignment statement to store the value back into the variable, as in

```
s = s.upper()
```

The upper method makes it easy to write a predicate function called equals_ignore_case that checks whether two strings are equal if the comparison ignores the distinction between uppercase and lowercase characters, as follows:

```
def equals_ignore_case(s1, s2):
    return s1.upper() == s2.upper()
```

Testing for character properties

When you work with individual characters in a string, it is often useful to determine whether those characters fall into particular categories, such as letters or digits. The string class in Python includes methods that check to see whether the receiver string fits one of the specified categories. For example, the expression `ch.isdigit()` has the value `True` if `ch` contains a digit character and the value `False` if `ch` contains any other type of character. Similarly, `ch.isspace()` returns `True` if `ch` is one of the whitespace characters, as defined on page 233.

In most cases, the methods that fall under the "Testing for character properties" subheading are applied to a single character. These methods, however, can also be applied to longer strings. For example, if `line` contains the string `"1729"`, calling `line.isdigit()` returns `True` because every character in line is a digit. The rules for applying these methods to multicharacter strings are summarized in the short descriptions provided in Figure 7-2.

Formatting methods

The first three methods under the "Formatting methods" subheading in Figure 7-2 implement left, right, and center padding for strings. The `ljust`, `rjust`, and `center` methods in the string class therefore serve as library counterparts to the `align_left`, `align_right`, and `align_center` functions you implemented in Chapter 5, exercise 7. That exercise suggests that these functions are important enough to include in a library, and Python has done just that. The *f*-string model offers a sophisticated facility for specifying how numbers and other types are formatted for display on the console or other output devices. That method and other strategies for controlling how strings appear are described in more detail in the section on "Formatting strings" later in this chapter.

7.5 Building string applications

The easiest way to improve your understanding of strings is to look at several sample applications. The sections that follow walk you through four applications that use strings in different ways.

Checking for palindromes

A ***palindrome*** is a word that reads identically backward and forward, such as *level* or *noon*. The goal of this section is to write a predicate function `is_palindrome` that checks whether a string is a palindrome. Calling `is_palindrome("level")` should return `True`; calling `is_palindrome("xyz")` should return `False`.

As with most programming problems, there is more than one strategy for solving this problem. The following code illustrates one strategy:

```
def is_palindrome(s):
    for i in range(len(s) // 2):
        if s[i] != s[-(i + 1)]:
            return False
    return True
```

This implementation uses a for loop to run through each index position in the first half of the string, checking whether the character in that position matches the one in the symmetric position relative to the end of the string.

If, however, you make use of the functions you already have, you can code is_palindrome in a much simpler form, as follows:

```
def is_palindrome(s):
   return s == reverse_string(s)
```

Although both implementations of is_palindrome return the correct result, there are various tradeoffs that may lead you to choose one over the other. The first implementation is likely to be more efficient because it doesn't require creating any new strings. Despite the difference in efficiency, the second version has many advantages, particularly as an example for new programmers. For one thing, it takes advantage of existing code by making use of the reverse_string function. For another, it hides the complexity involved in calculating the index positions required by the first version. It takes at least a minute or two for most students to figure out why the code includes the selection expression s[-(i + 1)] or why the upper limit of the for range is len(s) // 2. By contrast, the line

```
return s == reverse_string(s)
```

reads as fluidly as English: a string is a palindrome if it is equal to the same string when you reverse it. That, after all, is precisely the definition of a palindrome.

Particularly as you are learning about programming, it is better to work toward the clarity shown in the second implementation of is_palindrome than to try and match the efficiency of the first. Given the speed of modern computers, it is almost always worth sacrificing some efficiency to make a program easier to understand.

Generating acronyms

An ***acronym*** is a new word formed by combining, in order, the initial letters of a series of words. For example, *NATO* is an acronym formed from the first letters in *North Atlantic Treaty Organization.* The goal of this section is to write a function called acronym that takes a string and returns its acronym. For example, calling

```
acronym("North Atlantic Treaty Organization")
```

should return the string "NATO". Similarly, calling

```
acronym("port out starboard home")
```

should return the acronym "posh".

When you first look at the problem, it might seem that the obvious approach is to start with the first character and then search for spaces in a while loop. Each time the function finds a space, it can concatenate the next character onto the end of the string variable used to hold the result. When no more spaces appear in the string, the acronym is complete. This strategy can be translated into a Python implementation as follows:

```
def acronym(s):
    result = s[0]
    sp = s.find(" ")
    while sp != -1:
        result += s[sp + 1]
        sp = s.find(" ", sp + 1)
    return result
```

Although this implementation works for some strings, it fails for others. For example, it produces the correct algorithm only if each pair of words is separated by exactly one space. If some of the words are separated using hyphens—as in "self-contained underwater breathing apparatus", which produces the acronym "scuba"—this implementation will fail to return the correct result. Worse still, the function will generate an error condition if the word ends with a space, because the selection expression s[sp + 1] will try to select the character after the end of the string, which doesn't exist.

Although the following implementation is not as easy to follow, it correctly handles the special cases in which the earlier version fails:

```
def acronym(s):
    result = ""
    in_word = False
    for ch in s:
        if ch.isalpha():
            if not in_word:
                result += ch
            in_word = True
        else:
            in_word = False
    return result
```

This implementation uses the standard idiom to go through the string character by character, looking at each one. It determines the word boundaries by using the variable `in_word`, which is `True` if the process is scanning letters and `False` if it is scanning nonletters. New letters get added to the acronym only if the code sees a letter when `in_word` was previously `False`.

Translating English to Pig Latin

To give you more of a sense of how to implement string-processing applications, this section describes a Python function that takes a line of text and translates each word in that line from English to Pig Latin, a made-up language familiar to most children in the English-speaking world. In Pig Latin, words are formed from their English counterparts by applying the following rules:

1. If the word contains no vowels, no translation is done, which means that the Pig Latin word is the same as the original.
2. If the word begins with a vowel, the Pig Latin translation consists of the original word followed by the suffix *way.*
3. If the word begins with a consonant, the Pig Latin translation is formed by extracting the string of consonants up to the first vowel, moving that collection of consonants to the end of the word, and then adding the suffix *ay.*

As an example, suppose that the English word is *scram*. Because the word begins with a consonant, you divide it into two parts: one consisting of the letters before the first vowel and one consisting of that vowel and the remaining letters:

$$\boxed{\texttt{scr}}\ \boxed{\texttt{am}}$$

You then interchange these two parts and add *ay* at the end, as follows:

$$\boxed{\texttt{am}}\ \boxed{\texttt{scr}}\ \boxed{\texttt{ay}}$$

Thus the Pig Latin word for *scram* is *amscray*. For a word that begins with a vowel, such as *apple,* you simply add *way* to the end, which leaves you with *appleway.*

The code for the `PigLatin.py` program appears in Figure 7-3. The file exports two functions for clients to use. The `word_to_pig_latin` function converts a word to its Pig Latin equivalent. The `to_pig_latin` function takes a line of text and converts the entire line to Pig Latin by divides the line into words and then converting each word. Characters that are not part of a word are copied directly to the output line so that punctuation and spacing remain unaffected. The following IDLE session shows a few calls to the function `to_pig_latin`, which also works for single words:

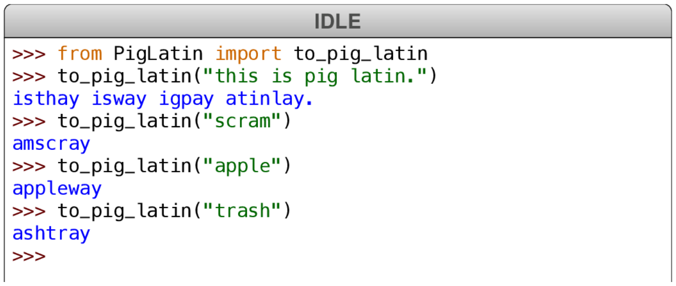
FIGURE 7-3 Functions to translate English to Pig Latin

```
# File: PigLatin.py

"""
This file converts from English to Pig Latin using the following rules:
  1. If the word begins with a vowel, add "way" to the end of the word.
  2. If the word begins with a consonant, extract the leading consonants
     up to the first vowel, move them to the end, and then add "ay".
  3. If the word contains no vowels, return the word unchanged.
"""

def to_pig_latin(line):
    """Converts a multi-word string from English to Pig Latin."""
    result = ""
    start = -1
    for i in range(len(line)):
        ch = line[i]
        if ch.isalpha():
            if start == -1:
                start = i
        else:
            if start >= 0:
                result += word_to_pig_latin(line[start:i])
            start = -1
            result += ch
    if start >= 0:
        result += word_to_pig_latin(line[start:])
    return result

def word_to_pig_latin(word):
    """Translates a word to Pig Latin."""
    vp = find_first_vowel(word)
    if vp == -1:
        return word
    elif vp == 0:
        return word + "way"
    else:
        head = word[0:vp]
        tail = word[vp:]
        return tail + head + "ay"

def find_first_vowel(word):
    """Returns the index of the first vowel in the word, or -1 if none."""
    for i in range(len(word)):
        if is_english_vowel(word[i]):
            return i
    return -1
```

☞

FIGURE 7-3 Functions to translate English to Pig Latin (continued)

```
def is_english_vowel(ch):
    """Returns True if ch is an English vowel (A, E, I, O, or U)."""
    return len(ch) == 1 and "AEIOUaeiou".find(ch) != -1

# Main program

def pig_latin():
    finished = False
    while not finished:
        line = input("Enter a string: ")
        if line == "":
            finished = True
        else:
            print("Pig Latin form: " + to_pig_latin(line))

# Startup code

if __name__ == "__main__":
    pig_latin()
```



```
>>> from PigLatin import to_pig_latin
>>> to_pig_latin("this is pig latin.")
isthay isway igpay atinlay.
>>> to_pig_latin("scram")
amscray
>>> to_pig_latin("apple")
appleway
>>> to_pig_latin("trash")
ashtray
>>>
```

It is worth taking a careful look at the implementations of `to_pig_latin` and `word_to_pig_latin` in Figure 7-3. The `to_pig_latin` function finds the word boundaries in the input, which provides a useful pattern for separating a string into individual words. The `word_to_pig_latin` function uses slicing to extract pieces of the English word and then uses concatenation to put them back together in their Pig Latin form.

Implementing simple ciphers

Codes and ciphers have been around in some form or another for most of recorded history. There is evidence to suggest that coded messages were used in ancient Egypt, China, and India, possibly as early as the third millennium BCE, although few details of the cryptographic systems have survived. In Book 6 of the *Iliad,* Homer suggests the existence of a coded message when King Proitos, seeking to have the young Bellerophontes killed,

> sent him to Lykia, and handed him murderous symbols, which
> he inscribed on a folding tablet, enough to destroy life . . .

Shakespeare's Hamlet, of course, has Rosencrantz and Guildenstern carry a similarly dangerous missive, but Hamlet's message is secured under a royal seal. In the *Iliad,* nothing in the text suggests that the message is sealed, which implies that the meaning of the "murderous symbols" must somehow be disguised.

	1	2	3	4	5
1	**A**	**B**	**C**	**D**	**E**
2	**F**	**G**	**H**	**IJ**	**K**
3	**L**	**M**	**N**	**O**	**P**
4	**Q**	**R**	**S**	**T**	**U**
5	**V**	**W**	**X**	**Y**	**Z**

Polybius square

One of the first encryption systems whose details survive is the ***Polybius square,*** developed by the Greek historian Polybius in the second century BCE. In this system, the letters of the alphabet are arranged to form a 5×5 grid in which each letter is represented by its row and column number. Suppose, for instance, that you want to transmit following English version of Pheidippides's message to Sparta:

THE ATHENIANS BESEECH YOU TO HASTEN TO THEIR AID

This message can be transmitted as a series of numeric pairs, as follows:

44 23 15 11 44 23 15 33 24 11 33 43 12 15 43 15 15 13 23 54
34 45 44 34 23 11 43 44 15 33 44 34 44 23 15 24 42 11 24 14

The real advantage of the Polybius square is not that it allows for secret messages, but that it simplifies the problem of transmission. Each letter in the message can be represented by holding between one and five torches in each hand, which allows a message to be communicated visually over a great distance. By reducing the alphabet to an easily transmittable code, the Polybius square anticipates such later developments as Morse code and semaphore, not to mention modern digital encodings such as ASCII or Unicode.

In *De Vita Caesarum,* written sometime around 110 CE, the Roman historian Suetonius describes an encryption system used by Julius Caesar, as follows:

> If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely *D,* for *A,* and so with the others.

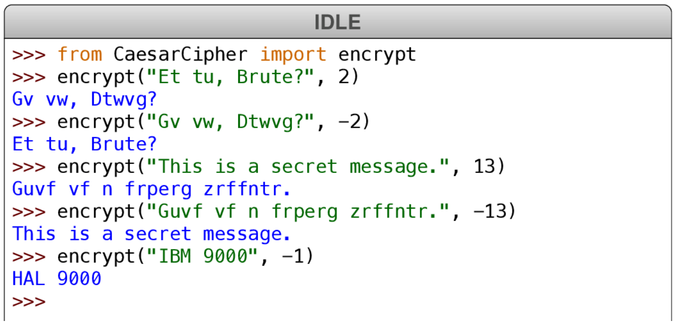
Even today, the technique of encoding a message by shifting letters a certain distance in the alphabet is called a ***Caesar cipher***. According to the passage from Suetonius, each letter is shifted three positions ahead in the alphabet. For example, if Caesar had had time to translate his final words according to his coding system, **ET TU BRUTE** would have come out as **HW WX EUXWH**, because **E** gets moved three letters ahead to **H**, **T** gets moved three to **W**, and so on. Letters that get advanced past the end of the alphabet wrap around back to the beginning, so that **X** becomes **A**, **Y** becomes **B**, and **Z** becomes **C**.

FIGURE 7-4 **Function to encrypt a message using a Caesar cipher**

```
# Implementation notes: encrypt
# -----------------------------
# Calling encrypt(s, key) encrypts the string s by adding key to
# each letter, wrapping around the end of the alphabet if necessary.

def encrypt(s, key):
    """Encrypts s using a Caesar cipher with the specified key value."""
    if key < 0:
        key = 26 - (-key % 26)
    result = ""
    for ch in s:
        if ch >= "A" and ch <= "Z":
            base = ord("A")
            code = ord(ch)
            ch = chr(base + (code - base + key) % 26)
        elif ch >= "a" and ch <= "z":
            base = ord("a")
            code = ord(ch)
            ch = chr(base + (code - base + key) % 26)
        result += ch
    return result
```

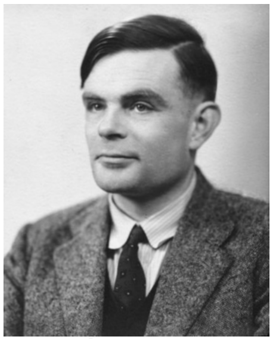
The `caesar_cipher` function in Figure 7-4 translates the letters in a string according to the rules for constructing a Caesar cipher. The code uses `ord` to convert characters into their Unicode values and then uses the remainder operator to implement the cyclical shift that wraps around to the beginning of the alphabet. Once the `caesar_cipher` function has computed the new character code, it uses `chr` to convert the Unicode value back into a string. The code makes sure that the operands to `%` are positive to avoid relying on Python's mathematical assumptions.

The following IDLE session demonstrates the operation of `caesar_cipher`:



```
>>> from CaesarCipher import encrypt
>>> encrypt("Et tu, Brute?", 2)
Gv vw, Dtwvg?
>>> encrypt("Gv vw, Dtwvg?", -2)
Et tu, Brute?
>>> encrypt("This is a secret message.", 13)
Guvf vf n frperg zrffntr.
>>> encrypt("Guvf vf n frperg zrffntr.", -13)
This is a secret message.
>>> encrypt("IBM 9000", -1)
HAL 9000
>>>
```

Cryptography played an important role in the early history of computing. During World War II, a team of mathematicians and engineers at Bletchley Park in England



Alan Turing

used electromechanical devices to break the German Enigma code. That accomplishment, in which the pioneering computer scientist Alan Turing played a major role, proved vital to the Allied war effort. Although this work was kept secret for many years after the war, it has recently been popularized in a series of films including *Breaking the Code*, *Enigma*, and *The Imitation Game*.

7.6 Formatting strings

Modern computing has its roots in machines designed for processing and tabulating data. As noted in the introduction to this chapter, Hermann Hollerith designed a tabulating machine that was used to process the 1890 census in the United States, thereby making it possible to complete the process in substantially less time. The company that Hollerith founded went on to become International Business Machines (IBM), which dominated the computing industry for most of the 20th century.

The early IBM machines were used primarily to automate such record-keeping operations as accounting, inventory control, and payroll processing. Those applications produced printed reports designed for human readers, which required arranging the output in a tabular format with fixed-width columns. The need to produce easily readable output meant that programming languages designed to support data processing included features that allowed programmers to specify the exact format in which the various output values should appear.

Although precisely formatted output is no longer as important as it was in the past, modern languages typically offer some strategy for controlling output format. Python is particularly generous in this regard. Over its history, Python has embraced three different strategies for controlling output formatting:

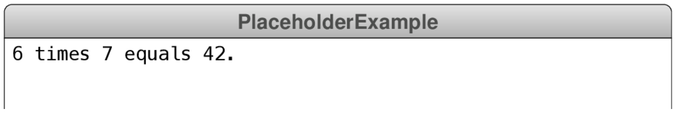
1. ***Percent-sign formatting.*** Early versions of Python redefined the remainder operator (`%`) for the string class, turning it into a general tool for substituting values into an existing string. Because the Python style guidelines now discourage this approach, it does not make sense to cover it in detail.
2. ***The* `format` *method in the string class.*** Beginning with Python version 2.0, the string class includes a method called `format` that allows callers to create new strings by replacing placeholders with formatted values.
3. ***Formatted string literals.*** In December 2016, Python version 3.6 introduced formatting at the language level. The model is similar to the `format` method, but more concise and easier to use, as you saw in Chapter 1.

Formatted string literals (or ***f-strings*** for short) are so much easier to use that they have rendered the earlier models largely obsolete. You may at some point find that you need to *read* programs that use the older styles, but it doesn't make sense to use those models in the programs you *write*.

Suppose that you have two variables, n1 and n2, each of which contains an integer. What you would like to do is produce an output line of the form

```
___ times ___ equals ___.
```

in which the underscored components are replaced by n1, n2, and their product. For example, if n1 and n2 contain 6 and 7, the output should look like this:



```
6 times 7 equals 42.
```

Although you can solve this problem by concatenating the components of the output line in a call to the print function, you've known since Chapter 1 that the simplest way to produce this output line is to use the following line:

```
print(f"{n1} times {n2} equals {n1 * n2}.")
```

The expressions inside the curly braces are called ***placeholders.*** When Python evaluates an *f*-string, it replaces the text inside the braces with the value of the expression evaluated at that point in the execution of the program, automatically converting the result to a string if necessary.

So far, the examples of the *f*-string model have done nothing more than substitute values for placeholders. The real power of these the placeholder model lies in the ability to control how the inserted values are formatted. In addition to the expression that indicates what value should be inserted, Python allows programmers to specify a ***format specification*** consisting of a colon and a sequence of format-control options just before the closing brace. The complete list of options is long, but the following are among the most useful:

- ***Fill.*** If the converted value is shorter than the minimum field width described later in this list, the output needs to be padded to reach the required width. By default, the value is padded with spaces, but you can change this behavior by starting the options specification with some other character.
- ***Alignment.*** The most common values for the alignment option are <, ^, >, which specify left, center, and right alignment, respectively. By default, Python uses right alignment for numeric values and left alignment for other values.
- ***Sign.*** The sign option is usually omitted, in which case negative numbers are preceded by a minus sign. Specifying + as the sign option ensures that all numbers include either a plus or a minus sign. Using a space ensures that positive numbers are preceded by a space, which helps to maintain alignment.
- ***Grouping.*** The most common grouping option is the comma, which indicates that commas should be used to separate numeric output at three-digit boundaries.

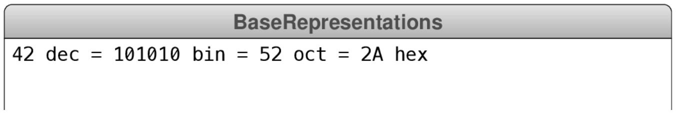
- ***Width.*** The width option is an integer that indicates the minimum width of the field in terms of the numbers of characters. If the converted value is shorter than the width, it will be padded as specified by the alignment option.
- ***Precision.*** The precision option appears after the width option, preceded by a period so that the pair look like a floating-point number. The interpretation of the precision option depends on the conversion type but usually controls the number of digits after the decimal point.
- ***Type.*** The last character in the format specification indicates the type of conversion. The most common conversion types appear in Figure 7-5. Uppercase format codes force uppercase letters in the conversion.

The number of possible combinations of the various formatting options is so large that it is impossible to cover them all. The easiest way to familiarize yourself with the available formatting options is through experimentation. Look through the descriptions for a feature that sounds like something you might want to use and then give it a try. Even so, it helps to offer a couple of simple examples.

The following function takes an integer and then prints the representation of that integer in decimal, binary, octal, and hexadecimal notation:

```
def base_representations(v):
    print(f"{v:d} dec = {v:b} bin = {v:o} oct = {v:X} hex")
```

Calling `base_representations(42)` produces the following output:



```
42 dec = 101010 bin = 52 oct = 2A hex
```

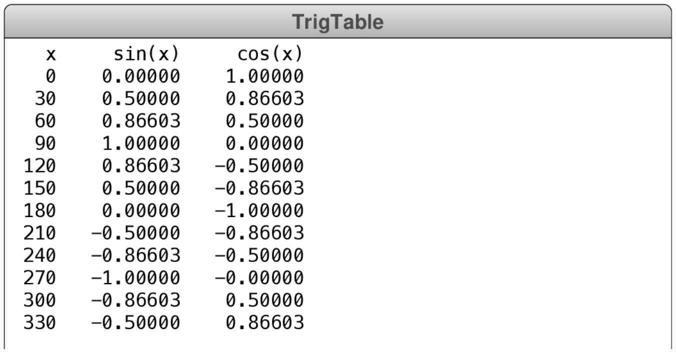
FIGURE 7-5 Conversion type codes for use with *f*-strings and the `format` method

`b`	Converts an integer into its binary representation as a string of **0**s and **1**s.
`d`	Converts an integer into its conventional decimal (base 10) representation.
`e` *or* `E`	Converts a number into its representation using scientific notation.
`f` *or* `F`	Converts a number into a floating-point string.
`g` *or* `G`	Converts a number using either `e` or `f`, choosing the best fit.
`o`	Converts an integer into an octal (base 8) string.
`s`	Converts any value into its string representation.
`x` *or* `X`	Converts an integer into a hexadecimal (base 16) string.

As a second example, the function

```
def trig_table(step):
    print("   x     sin(x)     cos(x)")
    for x in range(0, 360, step):
        r = math.radians(x)
        sinr = math.sin(r)
        cosr = math.cos(r)
        print(f" {x:3d}   {sinr: 7.5f}   {cosr: 7.5f}")
```

produces a table of sines and cosines for angles ranging from 0 to 360 degrees in increments specified by the `step` parameter. The results of each mathematical function are displayed in a fixed-point format that is seven characters wide, with five digits after the decimal point, allowing extra space for a negative sign. Calling `trig_table(30)`, for example, produces the following table:



```
   x     sin(x)     cos(x)
   0    0.00000    1.00000
  30    0.50000    0.86603
  60    0.86603    0.50000
  90    1.00000    0.00000
 120    0.86603   -0.50000
 150    0.50000   -0.86603
 180    0.00000   -1.00000
 210   -0.50000   -0.86603
 240   -0.86603   -0.50000
 270   -1.00000   -0.00000
 300   -0.86603    0.50000
 330   -0.50000    0.86603
```

Python's *f*-strings are useful in writing test programs for the code you write. By reducing the amount of code you need to write, *f*-strings make the process less onerous, which in turn makes it more likely that programmers will write those tests. Here, for example, is a test program for the `acronym` function:

```
def test_acronym():

    def test(s, expected):
        result = acronym(s)
        print(f"acronym(\"{s}\") -> \"{result}\"" +
              f" (should be \"{expected}\")")

    test("self-contained underwater breathing apparatus",
         "scuba")
    test("port out starboard home", "posh")
    test("North Atlantic Treaty Organization", "NATO")
```

Summary

In this chapter, you have learned how to use Python's built-in string type, which makes it easy to write string-processing applications without worrying about the details of the underlying representation. Important points in this chapter include:

- The fundamental unit of information in a modern computer is a *bit,* which can be in one of two possible states. The state of a bit is usually represented in memory diagrams using the binary digits **0** and **1**, but it is equally appropriate to think of these values as *off* and *on* or *false* and *true,* depending on the application.
- Sequences of bits are combined inside the hardware to form larger structures, including *bytes,* which are eight bits long, and *words,* which are large enough to contain a standard integer.
- Computer scientists tend to record the values of bit sequences in *hexadecimal* (base 16), which allows binary values to be represented in a more compact form.
- Numbers don't have bases; representations do.
- Nonnumeric data values are represented by numbering the elements in the domain and then using those numbers as codes for the original values.
- Characters are represented internally using a coding scheme called *Unicode,* which assigns numeric values to characters from a wide range of languages.
- Python allows you to convert between Unicode values and strings using the built-in functions `ord` and `chr`.
- The string class represents a type that is conceptually a sequence of characters. The character positions in a string are assigned index numbers that start at 0 and extend up to one less than the length of the string.
- Python allows negative index numbers for strings, which count backward from the end of the string.
- Python allows you to concatenate `n` copies of a string `s` using the notation `s * n`.
- You can extract a substring in Python using *slicing,* which ordinarily appears in the form *str*`[`*start*`:`*limit*`]`. This form of slicing produces a substring that begins at index position *start* and continues up to but not including index position *limit.* If *start* is missing, it defaults to 0. If *limit* is missing, it defaults to the end of the string.
- The square-bracket syntax for slicing also accepts a third component called the *stride,* which specifies how many characters to move ahead while composing the substring.
- The *stride* component can be negative, in which case the selection of characters occurs backward from the end of the string.

- The idiomatic pattern for iterating through the elements of a string is

```
for ch in s:
    . . . body of loop that manipulates ch . . .
```

- The standard pattern for growing a string by concatenation is

```
result = ""
for whatever loop header line fits the application:
    result += the next piece of the result
```

- Python implements several strategies that support inserting formatted values into a string. This chapter describes the use of *formatted string literals* or *f-strings,* which were added to Python's version 3.6 release in December 2016 and quickly became the most popular model for format control.
- When you use the *f*-string model, you add the letter `f` before the initial quotation mark and then include *placeholders* enclosed in curly braces. Evaluation of the *f*-string triggers replacement of those placeholders with the values of the expressions enclosed within the braces. The *f*-string model allows the user to specify additional formatting options such as the conversion type, alignment, field width, and precision.

Review questions

1. Define the following terms: *bit, byte,* and *word.*

2. What is the etymology of the word *bit?*

3. Convert each of the following decimal numbers to its hexadecimal equivalent:

   a. 17
   b. 256
   c. 1729
   d. 2766

4. Convert each of the following hexadecimal numbers to decimal:

   a. 17
   b. 64
   c. **CC**
   d. **FAD**

5. In his "New Math" song mentioned on page 4, Tom Lehrer notes that "the book I got this problem out of wants you to do it in base 8." What is $342_8 - 173_8$?

6. What Python functions allow you to convert back and forth between an integer and the corresponding Unicode character?

7. What does *ASCII* stand for?

8. What is the relationship between ASCII and Unicode?

9. By consulting Figure 7-1, determine the Unicode values of the characters `"$"`, `"@"`, `"0"`, and `"x"`.

10. True or false: In Python, you can determine the length of the string stored in the variable `s` by calling `len(s)`.

11. True or false: The index positions in a string begin at 1 and extend up to the length of the string.

12. How do you extract the character at position *k* in a string?

13. What are the three components of the square-bracket notation used to indicate slicing? What are the default values of each component?

14. What is *lexicographic ordering?*

15. What value does `find` return if the pattern string does not appear?

16. What is the significance of the optional second argument to `find`?

17. Suppose that you have declared and initialized the variable `s` as follows:

```
s = "hello, world"
```

Given that declaration, what value is produced by each of the following calls:

a. `len(s)`
b. `s[5]`
c. `s[-3]`
d. `s.find("l")`
e. `s.find("l", 5)`
f. `s.replace("h", "j")`
g. `s[3:5]`
h. `s[7:]`
i. `s[3:3]`
j. `s[::-2]`

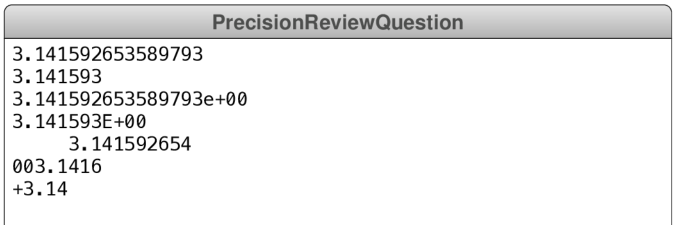
18. What is the pattern for iterating through each character in a string?

19. What is the pattern for growing a string through concatenation?

20. What value is produced by each of the following *f*-strings:

a. `f"{17} + {25} = {17 + 25}"`
b. `f"{127:X}"`
c. `f"{2.7182818:6.4f}"`
d. `f"{'L':<3s}{'C':^3s}{'R':>3s}"`

21. To 16 significant digits, the constant `math.pi` is 3.141592653589793. What *f*-string conversion would you use to produce each line in the following sample run:



```
3.141592653589793
3.141593
3.141592653589793e+00
3.141593E+00
     3.141592654
003.1416
+3.14
```

Exercises

1. In exercise 10 from Chapter 3, you wrote a program to find perfect numbers. Rewrite that program so that it also displays the binary form of these numbers. As you can see if you run this program, the first few perfect numbers follow an interesting pattern when you write them out in binary. Euclid discovered this pattern more than 2000 years ago, and the 18th-century Swiss mathematician Leonhard Euler proved that all even perfect numbers follow this pattern. The question of whether any odd perfect numbers exist remains unresolved in mathematics.

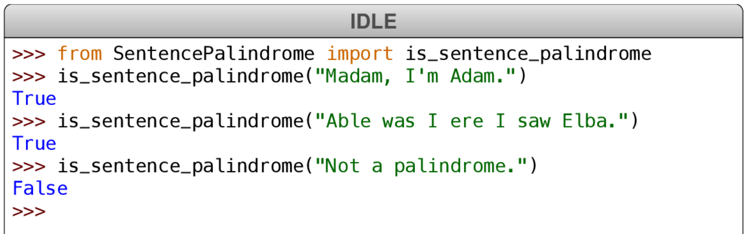
2. Suppose that the `startswith` and `endswith` methods were not defined in Python. Implement the same functionality by defining the more readably named functions `starts_with(s, prefix)` and `ends_with(s, suffix)`.

3. Implement the function `is_english_consonant(ch)`, which returns `True` if `ch` is a consonant in English, that is, any alphabetic character except one of the five vowels: `"a"`, `"e"`, `"i"`, `"o"`, and `"u"`. As with the `is_english_vowel` function presented in the text, your method should recognize both lower- and uppercase consonants.

4. Rewrite the `is_palindrome` function so that it operates recursively, taking advantage of the fact that a string is a palindrome if (a) its length is less than two or (b) its first and last characters match and the substring between those characters is a palindrome.

5. The concept of a palindrome is often extended to full sentences by ignoring punctuation, spacing, and differences in the case of letters. For example, the string `"Madam, I'm Adam."` is a sentence palindrome, because if you look only at the letters and ignore any case distinctions, it reads identically backward and forward.

   Write a predicate function `is_sentence_palindrome(s)` that returns `True` if `s` fits this definition of a sentence palindrome. For example, you should be able to use your function to reproduce the following IDLE session:



```
>>> from SentencePalindrome import is_sentence_palindrome
>>> is_sentence_palindrome("Madam, I'm Adam.")
True
>>> is_sentence_palindrome("Able was I ere I saw Elba.")
True
>>> is_sentence_palindrome("Not a palindrome.")
False
>>>
```

6. Write a function `create_regular_plural(word)` that returns the plural of `word` formed by following these standard English rules:

   a. If the word ends in *s, x, z, ch,* or *sh,* add *es* to the word.

   b. If the word ends in a *y* preceded by a consonant, change the *y* to *ies.*

   c. In all other cases, add just an *s.*

   Design a set of test cases to verify that your function works.

7. In English, the notion of an ongoing action is expressed using the present progressive tense, which involves the addition of an *ing* suffix to the verb. For example, the sentence *I think* conveys a sense that one is capable of thinking; by contrast, the sentence *I am thinking* conveys the impression that one is currently doing so. The *ing* form of the verb is called the ***present participle.***

   Unfortunately, creating the present participle is not always as simple as adding the *ing* ending. One common exception is a word like *cogitate* that ends in a silent *e*. In such cases, the *e* is usually dropped, so that the participle form becomes *cogitating*. Another common exception involves words that end with a single consonant, which typically gets doubled in the participle form. For example, the verb *run* becomes *running*.

   Although there are many exceptions, you can construct a large fraction of the legal participle forms in English by applying the following rules:

   a. If the word ends in an *e* preceded by a consonant, take the *e* away before adding *ing.* Thus, *move* should become *moving*. If the *e* is not preceded by a consonant, it should remain in place, so that *see* becomes *seeing*.

   b. If the word ends in a consonant preceded by a vowel, insert an extra copy of that consonant before adding *ing.* Thus, *jam* should become *jamming*. If, however, there is more than one consonant at the end of the word, no such doubling takes place, so that *walk* becomes *walking*.

   c. In all other circumstances, simply add the *ing* suffix.

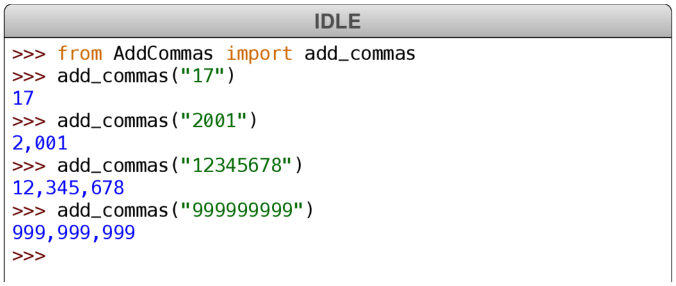
   Write a function `create_present_participle(verb)` that takes an English verb, which you may assume is entirely lowercase and at least two characters long, and forms the participle using these rules.

8. As in most languages, English includes two types of numbers. The ***cardinal numbers*** (such as *one, two, three,* and *four*) are used in counting; the ***ordinal numbers*** (such as *first, second, third,* and *fourth*) are used to indicate a position in a sequence. In text, ordinals are usually indicated by writing the digits in the number, followed by the last two letters of the English word that names the corresponding ordinal. Thus, the ordinal numbers *first, second, third,* and *fourth* often appear in print as *1st, 2nd, 3rd,* and *4th*. The ordinals for 11, 12, and 13, however, are *11th, 12th,* and *13th.* Devise a rule that determines what suffix should be added to each number, and then use this rule to write a function `create_ordinal_form(n)` that returns the ordinal form of the number `n` as a string.

9. > *The waste of time in spelling imaginary sounds and their history*
   > *(or etymology as it is called) is monstrous in English . . .*
   >
   > —George Bernard Shaw, 1941

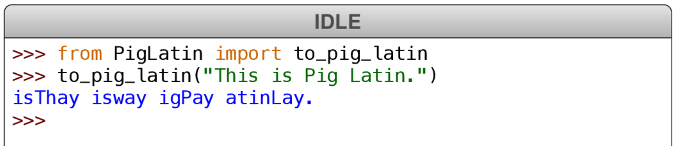
   In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was to eliminate all doubled letters, so that *bookkeeper* would be written as *bokeper* and *committee* would become *comite*. Write a function `remove_doubled_letters(s)` that returns a new string in which any duplicated characters in `s` have been replaced by a single copy.

10. When large numbers are written on paper, it is traditional—at least in the United States—to use commas to separate the digits into groups of three. For example, the number one million is usually written as 1,000,000. Implement a function `add_commas(digits)` that takes a string of digits representing a number and returns the string formed by inserting commas at every third position, starting on the right. Your implementation of `add_commas`—which should not use the grouping option in the *f*-string model to perform this operation—should be able to reproduce the following IDLE session:



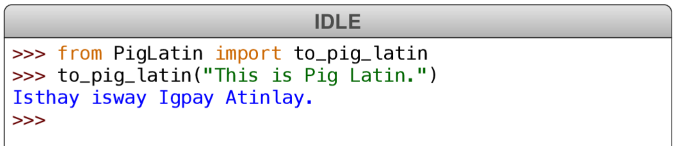
```
>>> from AddCommas import add_commas
>>> add_commas("17")
17
>>> add_commas("2001")
2,001
>>> add_commas("12345678")
12,345,678
>>> add_commas("999999999")
999,999,999
>>>
```

11. As written, the `PigLatin.py` program in Figure 7-3 behaves oddly if you enter a string that includes words beginning with an uppercase letter. For example, if you were to capitalize the first word in the sentence and the name of the Pig Latin language, you would see the following output:



```
>>> from PigLatin import to_pig_latin
>>> to_pig_latin("This is Pig Latin.")
isThay isway igPay atinLay.
>>>
```

Rewrite the `word_to_pig_latin` function so that any word that begins with a capital letter in the English line still begins with a capital letter in Pig Latin. Thus, after you make the necessary changes in the program, the output should look like this:



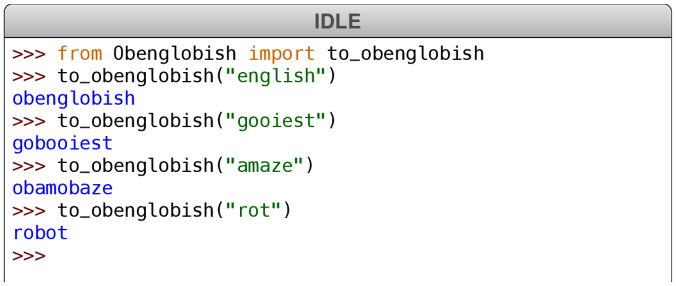
```
>>> from PigLatin import to_pig_latin
>>> to_pig_latin("This is Pig Latin.")
Isthay isway Igpay Atinlay.
>>>
```

12. Most people in English-speaking countries have played the Pig Latin game at some point in their lives. There are other invented "languages" in which words are created using some simple transformation of English. One such language is called *Obenglobish,* in which words are created by adding the letters *ob* before the vowels (*a, e, i, o,* and *u*) in an English word. For example, under this rule, the word *english* gets the letters *ob* added before the *e* and the *i* to form *obenglobish*, which is how the language got its name.

    In official Obenglobish, the `ob` characters are added only before vowels that are pronounced, which means that a word like *game* would become *gobame* rather than *gobamobe* because the final *e* is silent. While it is impossible to implement this rule perfectly, you can do a pretty good job by adopting the rule that the *ob* should be added before every vowel in the English word *except*

    - Vowels that follow other vowels
    - An *e* that occurs at the end of the word

    Write a function `obenglobish` that takes an English word and returns its Obenglobish equivalent, using the translation rule given above. Your function should allow you to generate the following IDLE session:



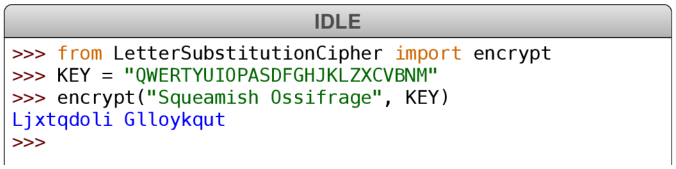
```
>>> from Obenglobish import to_obenglobish
>>> to_obenglobish("english")
obenglobish
>>> to_obenglobish("gooiest")
gobooiest
>>> to_obenglobish("amaze")
obamobaze
>>> to_obenglobish("rot")
robot
>>>
```

13. Although Caesar ciphers are simple, they are also extremely easy to break. A somewhat more secure scheme allows each letter in the message to be represented consistently by some other letter, but not one chosen by shifting the character a fixed distance in the alphabet. This kind of coding scheme is called a ***letter-substitution cipher.***

    The key in a letter-substitution cipher is a 26-character string that shows the enciphered counterpart of each of the 26 letters of the alphabet. For example, if the communicating parties choose `"QWERTYUIOPASDFGHJKLZXCVBNM"` as the key (which is unimaginatively generated by typing the letter keys on the keyboard in order), that key then corresponds to the following mapping:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
Q W E R T Y U I O P A S D F G H J K L Z X C V B N M
```

    Write a function `encrypt` that takes a string and a 26-character key and returns the string after applying a letter-substitution cipher with that key. For example, your function should be able to produce the following sample run:



```
>>> from LetterSubstitutionCipher import encrypt
>>> KEY = "QWERTYUIOPASDFGHJKLZXCVBNM"
>>> encrypt("Squeamish Ossifrage", KEY)
Ljxtqdoli Glloykqut
>>>
```

    The words *squeamish ossifrage* were part of the solution to a cryptographic puzzle published in *Scientific American.* The puzzle was developed by Ron Rivest, Adi Shamir, and Leonard Adleman, who invented the widely used RSA encryption algorithm, named from the first letters of their surnames.

14. Write a predicate function `is_key_legal`, which takes a string and returns `True` if that string would be a legal key in a letter-substitution cipher. A key is legal only if it meets the following two conditions:

    1. The key is exactly 26 characters long.
    2. Every uppercase letter appears in the key.

These conditions automatically rule out the possibility that the key contains invalid characters or duplicated letters. After all, if all 26 uppercase letters appear and the string is 26 characters long, there isn't room for anything else.

15. Letter-substitution ciphers require the sender and receiver to use different keys: one to encrypt the message and one to decrypt it when it reaches its destination. Your task in this exercise is to write a function `invert_key` that takes an encryption key and returns the corresponding decryption key. In cryptography, that operation is called ***inverting*** the encryption key.

    The idea of inverting a key is most easily illustrated by example. Suppose, for example, that the key is `"QWERTYUIOPASDFGHJKLZXCVBNM"` as in exercise 14. That key represents the following translation rule:

    ```
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
    Q W E R T Y U I O P A S D F G H J K L Z X C V B N M
    ```

    The translation table shows that `A` maps into `Q`, `B` maps into `W`, `C` maps into `E`, and so on. To turn the encryption process around, you have to read the translation table from bottom to top, looking to see what letter in the original text would have produced each letter in the encrypted version. For example, if you look for the letter `A` in the bottom line of the key, you discover that the corresponding letter in the original must have been `K`. Similarly, the only way to get a `B` in the encrypted message is to start with an `X` in the original one. The first two entries in the inverted translation table therefore look like this:

    ```
    A B
    ↓ ↓
    K X
    ```

    If you continue this process by finding each letter of the alphabet on the bottom of the original translation table and then looking to see what letter appears on top, you will eventually complete the inverted table, as follows:

    ```
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
    K X V M C N O P H Q R S Z Y I J A D L E G W B U F T
    ```

    The inverted key is simply the 26-character string on the bottom row, which in this case is `"KXVMCNOPHQRSZYIJADLEGWBUFT"`.

16. Rewrite the `FactorialTable.py` program from Figure 2-4 on page 54 so that it uses the formatting features of *f*-strings instead of the `align_right` function.

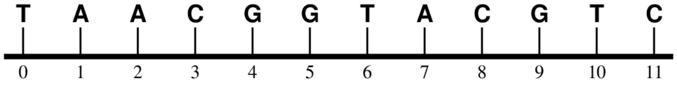
17. The genetic code for all living organisms is carried in its DNA—a molecule with the remarkable capacity to replicate its own structure. The DNA molecule itself consists of a long strand of chemical bases wound together with a similar strand in a double helix. DNA's ability to replicate comes from the fact that its four

constituent bases—adenosine, cytosine, guanine, and thymine—combine with each other only in the following ways:

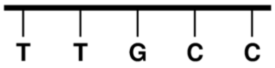
- Cytosine on one strand links only with guanine on the other, and vice versa.
- Adenosine links only with thymine, and vice versa.

Biologists abbreviate the names of the bases to a single letter: **A**, **C**, **G**, or **T**.

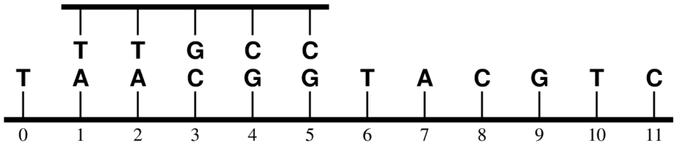
Inside the cell, a DNA strand acts as a template to which other DNA strands can attach themselves. As an example, suppose that you have the following DNA strand, in which the position of each base has been numbered as in a string:



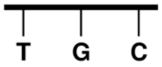
Your mission in this exercise is to determine at what point a shorter DNA strand can attach itself to the longer one. If, for example, you are trying to find a match for the strand



the rules dictate that this strand can bind to the longer one only at position 1:



By contrast, the strand

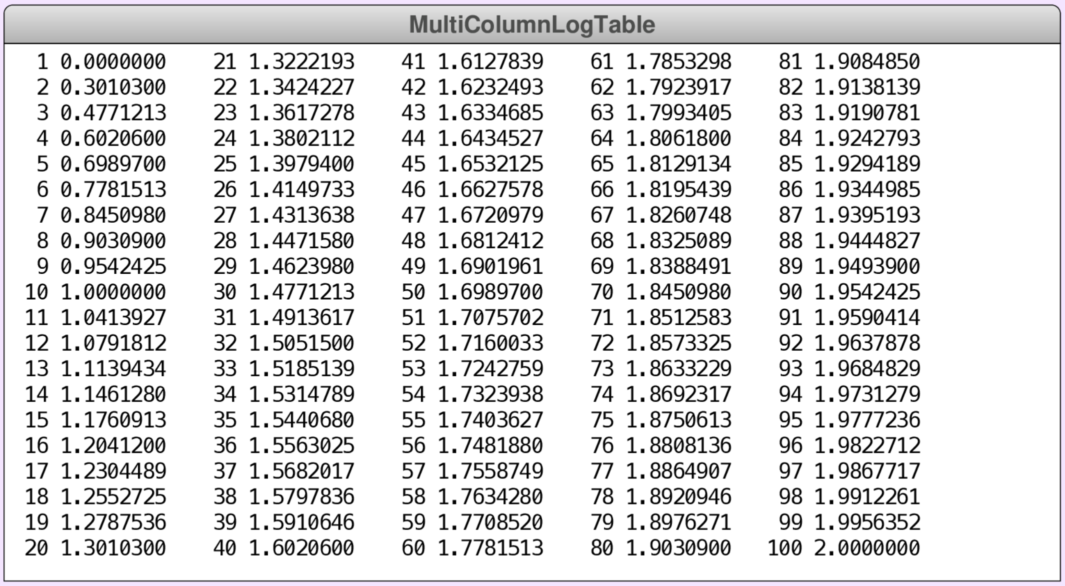


matches at either position 2 or position 7.

Write a function `find_dna_match(s1, s2, start=0)` that returns the first position at which the DNA strand `s1` can attach to the strand `s2` after the `start` position, which defaults to 0. If there is no match, `find_dna_match` should return −1.

18. When Charles Babbage designed his computing machines in the early 19th century, his initial motivation was to automate the production of mathematical tables, which were produced by hand and often contained frequent errors.

FIGURE 7-6 Formatted version of Charles Babbage's multicolumn table of logarithms



```
  1 0.0000000    21 1.3222193    41 1.6127839    61 1.7853298    81 1.9084850
  2 0.3010300    22 1.3424227    42 1.6232493    62 1.7923917    82 1.9138139
  3 0.4771213    23 1.3617278    43 1.6334685    63 1.7993405    83 1.9190781
  4 0.6020600    24 1.3802112    44 1.6434527    64 1.8061800    84 1.9242793
  5 0.6989700    25 1.3979400    45 1.6532125    65 1.8129134    85 1.9294189
  6 0.7781513    26 1.4149733    46 1.6627578    66 1.8195439    86 1.9344985
  7 0.8450980    27 1.4313638    47 1.6720979    67 1.8260748    87 1.9395193
  8 0.9030900    28 1.4471580    48 1.6812412    68 1.8325089    88 1.9444827
  9 0.9542425    29 1.4623980    49 1.6901961    69 1.8388491    89 1.9493900
 10 1.0000000    30 1.4771213    50 1.6989700    70 1.8450980    90 1.9542425
 11 1.0413927    31 1.4913617    51 1.7075702    71 1.8512583    91 1.9590414
 12 1.0791812    32 1.5051500    52 1.7160033    72 1.8573325    92 1.9637878
 13 1.1139434    33 1.5185139    53 1.7242759    73 1.8633229    93 1.9684829
 14 1.1461280    34 1.5314789    54 1.7323938    74 1.8692317    94 1.9731279
 15 1.1760913    35 1.5440680    55 1.7403627    75 1.8750613    95 1.9777236
 16 1.2041200    36 1.5563025    56 1.7481880    76 1.8808136    96 1.9822712
 17 1.2304489    37 1.5682017    57 1.7558749    77 1.8864907    97 1.9867717
 18 1.2552725    38 1.5797836    58 1.7634280    78 1.8920946    98 1.9912261
 19 1.2787536    39 1.5910646    59 1.7708520    79 1.8976271    99 1.9956352
 20 1.3010300    40 1.6020600    60 1.7781513    80 1.9030900   100 2.0000000
```

Figure 7-6 shows a table of logarithms (using the `math.log10` function) for the integers between 1 and 100, arranged—just as in Babbage's *Table of the Logarithms of the Natural Numbers*—into five vertical columns on the page.

Write a Python program to produce a multicolumn table of logarithms in the style of Figure 7-6.

19. The first program for Babbage's Analytical Engine—and therefore presumably the first program ever—was written by Ada Lovelace in 1843. Its purpose was to calculate the Bernoulli numbers, a mathematical series discovered by the Swiss mathematician Jakob Bernoulli at the beginning of the 18th century. Bernoulli numbers can be defined recursively as follows, where $C(n, k)$ is the combinations function introduced on page 142:

$$B(n) = \begin{cases} 1 & \text{if } n = 0 \\ \dfrac{\sum\limits_{i=0}^{n-1} C(n+1, i)\, B(i)}{n+1} & \text{otherwise} \end{cases}$$

Write a Python program to list the Bernoulli numbers, displaying the results in tabular form so that each value shows six digits after the decimal point.

CHAPTER 8

Lists

I'm not rich because I invented VisiCalc, but I feel that I've made a change in the world. That's a satisfaction money can't buy.

— Dan Bricklin, November 1985, as quoted in Robert Slater, *Portraits in Silicon*

Bob Frankston and Dan Bricklin

In modern computing, one of the most visible applications of the data structures described in this chapter is the electronic spreadsheet, which uses a two-dimensional array to store tabular information. The first electronic spreadsheet was VisiCalc, which was released in 1979 by Software Arts, Incorporated, a small startup company founded by MIT graduates Dan Bricklin and Bob Frankston. VisiCalc proved to be a popular application, leading many larger firms to develop competing products, including Lotus 1 2 3 and, more recently, Microsoft Excel.

Up to now, the programs in this book have worked with individual data items. The real power of computing, however, comes from the ability to work with collections of data. This chapter introduces the idea of an ***array,*** which is the general term that programmers use to indicate an ordered collection of values. Arrays are important in programming largely because ordered collections occur quite frequently in the real world. Whenever you want to represent a set of values in which it makes sense to think about those values as forming a sequence, arrays are likely to play a role in the solution.

At the same time, arrays—as a separate data type—are becoming less important because most programming languages today offer more powerful types that provide not only the limited set of operations historically associated with arrays but also a range of more sophisticated operations that programmers find extremely useful. Python, for example, does not include arrays in their traditional form but instead implements the array idea using a built-in data structure called a ***list.***

8.1 Introduction to arrays and lists

An array is a collection of individual values in which the elements are identified by a position number. You must be able to enumerate the individual values of an array in order: here is the first, here is the second, and so on. Conceptually, it is easiest to think of an array as a sequence of boxes, with one box for each data value. Each of the values in an array is called an ***element.***

As noted in the introduction to this chapter, Python implements the array concept using the more powerful `list` class, which is one of Python's built-in data types. Like every other data type in Python, lists can be stored in variables, passed as arguments to a function, and returned from functions as a result. And like every other data type, lists in Python support a set of operations appropriate to the type. For lists, that set of operations allows you to manipulate both the contents and the ordering of elements. These operations are outlined in the sections that follow.

Python list notation

Creating a list in Python is much easier than it is in most other programming languages. As you know from Chapter 1, all you need to do is enclose the elements of the list in square brackets, using commas to separate the elements. For example, the following constant declaration defines `COINS` as a list of integers that corresponds to coins in the United States:

```
COINS = [ 1, 5, 10, 25, 50, 100 ]
```

After you make this definition, the value of the constant `COINS` is a list that corresponds to the following box diagram:

COINS

1	5	10	25	50	100
0	1	2	3	4	5

The small numbers underneath the boxes in this diagram represent the position of that value in the list, which is called its ***index.*** As you already know, index numbers in Python always begin with 0 and run up to one less than the number of elements, just as they do for strings.

You can apply the built-in `len` function to a list to determine the number of elements it contains. The expression

```
len(COINS)
```

therefore has the value 6.

The elements of a list need not be numbers but can instead be any Python value. For example, the following variable declaration defines `hogwarts` as a list containing the names of the four houses at the Hogwarts School of Witchcraft and Wizardry from J. K. Rowling's Harry Potter novels:

```
hogwarts = [
    "Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"
]
```

The box diagram for this list looks like this:

hogwarts

"Gryffindor"	"Hufflepuff"	"Ravenclaw"	"Slytherin"
0	1	2	3

The expression `len(hogwarts)` has the value 4.

As with strings, you can select an individual element of a list by writing the name of the list and following it with the index written in square brackets. For example, given the earlier definitions in this section, the expression `COINS[2]` is 10, because that is the value at index 2 in the `COINS` list. Similarly, `hogwarts[0]` has the value `"Gryffindor"`. If you select an index position that falls outside the limits of a list, Python treats that selection as an error.

Sequence types

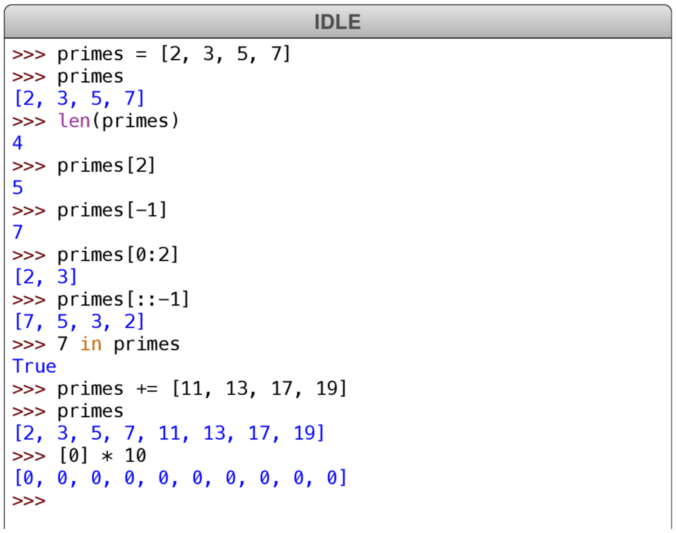
Unless you've already forgotten everything you learned in Chapter 7, you must have noticed by now that the discussion of lists from the preceding section is starting to sound familiar. Strings and lists share several fundamental properties. Both types number their elements beginning with index position 0, both types use the `len` function to determine the number of elements, and both types indicate selection using square brackets.

As it happens, the similarities between strings and lists extend well beyond these particular characteristics. In Python, strings and lists are examples of a more general class of objects called a ***sequence,*** which refers to any ordered collection. Strings are sequences of characters; lists are sequences of any Python value.

In keeping with the general principles of object-oriented programming—which you will have more of a chance to explore in Chapter 10—the fact that strings and lists are both sequences mean that strings and lists implement a common set of operations that apply to all sequences. In particular, most of the string operations you learned about in Chapter 7 apply equally well to lists. For example, lists support all of the following operations, which you already know from working with strings:

- The `len` function
- Index numbering that begins at 0 and extends up to the length minus 1
- Negative index numbering that counts from the end of the sequence
- Selection of an individual element using square brackets
- Slicing in all its forms
- Checking whether a value is contained in a list using the `in` operator
- Concatenation using the + or += operator
- Repetition using the ∗ operator

The effect of each of these operations is illustrated in the following IDLE session:

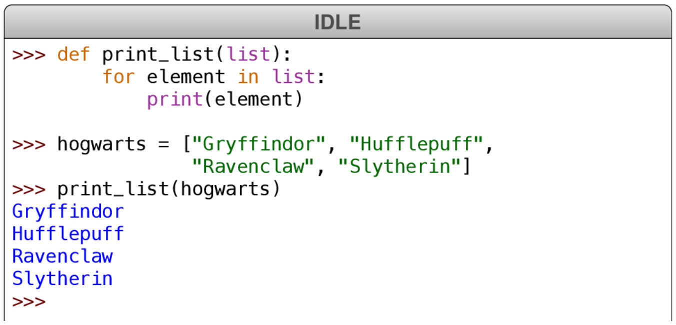


```
>>> primes = [2, 3, 5, 7]
>>> primes
[2, 3, 5, 7]
>>> len(primes)
4
>>> primes[2]
5
>>> primes[-1]
7
>>> primes[0:2]
[2, 3]
>>> primes[::-1]
[7, 5, 3, 2]
>>> 7 in primes
True
>>> primes += [11, 13, 17, 19]
>>> primes
[2, 3, 5, 7, 11, 13, 17, 19]
>>> [0] * 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>>
```

Perhaps more importantly, both strings and lists support iteration using the `for` statement. For example, you can use the following function to print the elements of a list on the console, one per line:

```
def print_list(list):
    for element in list:
        print(element)
```

This function allows you to reproduce the following IDLE session:



```
>>> def print_list(list):
        for element in list:
            print(element)

>>> hogwarts = ["Gryffindor", "Hufflepuff",
                "Ravenclaw", "Slytherin"]
>>> print_list(hogwarts)
Gryffindor
Hufflepuff
Ravenclaw
Slytherin
>>>
```

Assigning to list elements

Despite their many similarities, strings and lists differ in one important respect. In Python, strings are ***immutable,*** which means that you can't change their elements. Lists, by contrast, are ***mutable,*** which means, among other things, that you can assign a new value to a list element. For example, if some future leaders of Hogwarts decided that they might need to honor a more worthy wizard, evaluating the expression

```
hogwarts[3] = "Dumbledore"
```

would change the value of the `hogwarts` list to

"Gryffindor"	"Hufflepuff"	"Ravenclaw"	"Dumbledore"
0	1	2	3

Passing lists as parameters

When you pass a list as a parameter to a function, it is important to keep in mind that all objects in Python are represented internally as references, as described in Chapter 4. As a result, if a function makes changes to the elements of a list, the caller will see those changes because the caller and the function share access to the same list.

It is important to note that this behavior is not a violation of the rules for parameter passing presented in Chapter 5. In the list of rules presented in the section entitled “The steps in calling a function,” rules 3 and 4 read as follows:

3. Each positional argument is copied into the corresponding parameter variable.
4. All keyword arguments are copied to the parameter with the same name.

These rules say explicitly that both positional and keyword arguments are *copied* into the appropriate parameter variable, but passing a list as a parameter results in the values being shared. The explanation of this apparent inconsistency is that Python does indeed copy the argument, but that argument is just a reference. Python copies the reference but does not copy the internal data. The effect of this strategy is that a function and its caller have access to the same elements.

8.2 List methods

As was true for strings, Python’s list class also exports several methods that provide additional operations beyond those that apply to all sequences. Figure 8-1 at the top of the next page describes several of the most important methods implemented by the list class. These methods are detailed in the sections that follow.

Methods that leave the original list unchanged

Python’s list class exports several methods that leave the original list unchanged. The `index` method, for example, returns the first index at which the argument appears in the list. The `index` method is therefore in some ways analogous to the `find` method for strings. Like `find`, the `index` method takes an optional argument to specify the index position at which to start the search for a matching element. Unlike `find`, the `index` method raises a `ValueError` exception if no matching element is found. You can test for this condition using a `try` statement, which is described in the section entitled “Exception handling” later in this chapter.

The `count` method returns the number of list elements that match the argument. For example, if `scores` contains a list of the scores students received on an exam, the expression `scores.count(100)` would return the number of students who achieved a perfect score of 100.

The `copy` method returns a new list that contains the same elements as the original. The result of the `copy` method occupies a different address in memory and therefore no longer shares the list elements with the original. The use of `copy` is therefore different from assignment, which copies the *reference* to a list, leaving its values shared. Although the `copy` method creates a new set of memory cells for the elements of the list, it initializes the values of those cells by assignment from the old cells. If

FIGURE 8-1 **Methods exported by Python's list class**

Methods that leave the original list unchanged

list`.index(`*value*`)` *list*`.index(`*value*`,` *start*`)`	Returns the first index matching *value,* beginning at *start* if specified. This method raises a `ValueError` exception if no match is found.
list`.count(`*value*`)`	Returns the number of times *value* appears in the list.
list`.copy()`	Returns a shallow copy of the original list.

Methods that add and remove elements

list`.append(`*value*`)`	Adds *value* to the end of the list.
list`.extend(`$list_2$`)`	Adds the elements in $list_2$ to the end of the list.
list`.insert(`*index*`,` *value*`)`	Inserts *value* before the specified index position.
list`.remove(`*value*`)`	Removes the first instance of *value* from the list, raising a `ValueError` exception if it does not appear.
list`.pop()` *list*`.pop(`*index*`)`	Removes and returns the element at the specified index position. If no parameters are specified, `pop` removes and returns the last element.
list`.clear()`	Removes all the elements from the list.

Methods that reorder the elements of a list

list`.reverse()`	Reverses the order of the elements in the list.
list`.sort()` *list*`.sort(key=`*fn*`)` *list*`.sort(reverse=`*flag*`)`	Sorts the elements of the list. The `sort` method takes two optional keyword parameters: `key`, which is a function that produces the sort key, and `reverse`, which, if `True`, reverses the sort order.

String methods that involve lists

str`.split()` *str*`.split(`*separator*`)`	Splits *str* into a list of substrings by dividing it at each instance of the string *separator.* If *separator* is not specified, `split` divides the string at each sequence of whitespace characters.
str`.splitlines()`	Splits *str* into a list of substrings by dividing it at each line break.
sep`.join(`*list*`)`	Joins the elements of *list* into a string using *sep* to separate the elements.

those elements are themselves lists or any other object, the list produced by `copy` will contain the same internal references as the original. Computer scientists say that the `copy` method makes a ***shallow copy*** of the original because the copy operation does not descend below the top level of the list.

Methods that add and remove elements

The list class includes six methods that add or remove elements from a list. The `append` and `extend` methods support adding new elements to the end of the list; `append` adds a single element, and `extend` adds all the elements in a second list. The `insert` method makes it possible to add elements in the middle of a list. The index number in the `insert` method specifies the index position *before* the insertion. For example, calling

```
    list.insert(0, 17)
```

inserts the value 17 at the beginning of the list before element 0.

The `clear`, `remove`, and `pop` methods remove elements from a list. The `clear` method is the easiest to describe, since it has the effect of removing all the elements, leaving the list empty. The `remove` and `pop` methods each remove one element from the list. The primary difference is that `remove` takes the *value* of the element you want to remove, while `pop` takes the *index*. Another difference is that `pop` returns the value that was removed. As noted in Figure 8-1, the index number is optional in the `pop` method. If it is missing, `pop` removes and returns the last element.

Methods that reorder the elements of a list

The list class includes two methods—`reverse` and `sort`—that reorder the elements of an existing list. The `reverse` method simply reverses the elements of the list without allocating any new list storage. Although you can easily implement a function that duplicates its operation, the `reverse` method in the list class operates more efficiently because it is built into the Python language.

The `sort` method rearranges the elements of a list so that they appear in ascending order. If you call `sort` with no argument, Python sorts the list using the style of comparison that is appropriate to its elements. If you sort a list of numbers, Python reorders the elements so that they increase numerically. If you sort a list of strings, Python arranges the elements in lexicographic order.

The `sort` method takes two optional parameters, which must be specified as keyword arguments. The `key` parameter allows you to specify a ***key function,*** which takes a single argument and returns the value that `sort` should use in the comparison. For example, if `lines` is a list of strings, you can sort `lines` by increasing length by calling `lines.sort(key=len)`. Similarly, you can sort `lines` alphabetically ignoring case by calling `lines.sort(key=str.upper)`. In this example, `upper` is a method belonging to Python's string class, which means that you need to include the class name `str` so that Python knows where to find its definition. The `reverse` parameter allows you to invert the order. For example, calling `lines.sort(reverse=True)` sorts `lines` in reverse lexicographic order.

In many cases, you don't need to change the order of a list but can instead simply iterate through the list in a different order. The built-in functions `reversed` and sorted take any iterable object and return a new iterable object that cycles through the elements in a different order. For example, the statement

```
for line in reversed(lines):
```

leaves lines unchanged but cycles through them backward starting at the end. The sorted function also takes the optional key and reversed arguments from sort.

String methods that involve lists

The list operations described in Figure 8-1 include three methods that are part of the string class but which are covered here because they also require an understanding of lists. The split method separates a string into substrings by dividing it at each instance of a separator string and then returns a list of the individual substrings. For example, if date contains "16-Jul-1969" (the date on which Apollo 11 landed on the moon), calling date.split("-") returns the following list:

"16"	"Jul"	"1969"

If you leave out the separator string from the call to split, Python divides the string at any sequence of whitespace characters. For example, suppose that the variable line is defined as

```
line = "abc   def ghi "
```

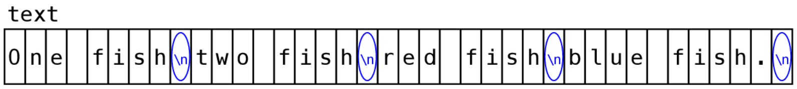
where three spaces separate the substrings "abc" and "def" and a space appears at the end of the string. Calling line.split() produces the three-element list

"abc"	"def"	"ghi"

If you instead call line.split(" ") using an explicit space character as the separator, Python splits the string at each individual occurrence of a space character to produce the following six-element list:

"abc"	""	""	"def"	"ghi"	""

The splitlines method is handy when you are working with a string that consists of a sequence of lines ending with the newline character. As an example, the first page of Dr. Seuss's *One Fish, Two Fish, Red Fish, Blue Fish* contains the words of the title divided into lines, which are stored internally like this:


text
O n e   f i s h \n t w o   f i s h \n r e d   f i s h \n b l u e   f i s h . \n

Calling `text.split("\n")` returns a list with five elements, the last of which is the empty string corresponding to the characters after the final newline. What you almost certainly want to do instead is to divide the title into its four components, which is exactly what `splitlines` does.

The `join` method reverses the operation of `split` and allows you to create a string from a list of substrings, inserting a separator between each one. You apply the `join` method to the separator string, passing in the list as an argument. Thus, the expression

```
" ".join(["this", "is", "a", "test"])
```

joins the four strings in the list together to produce the string `"this is a test"`, in which the substrings are separated by the space character.

8.3 List comprehensions

Although Python's notation for initializing a list by enclosing its elements in square brackets is very convenient, it suffers from a lack of flexibility. Suppose, for example, you are writing an application and discover that it would be useful to have a list of the powers of ten. If you know in advance how many powers of ten you need, you can initialize the list using Python's square-bracket syntax. For example, if you know that your application uses only the powers of ten up to a million (10^6) , you can use the following declaration:

```
powers_of_ten = [ 1, 10, 100, 1000, 10000, 100000, 1000000 ]
```

But what if you aren't sure how many powers of ten your clients will need? It would be better if you could structure your program so that the number of elements is specified by a constant called `N_POWERS`, making it easy to change. To produce the list that includes the first seven powers of ten, you would set `N_POWERS` to 7.

Given that the number of elements in `powers_of_ten` might change, you can no longer simply list the elements inside square brackets but must instead compute the desired values. One strategy for doing so is to initialize `powers_of_ten` to be an empty list and then append `N_POWERS` elements to it using a `for` loop, like this:

```
powers_of_ten = [ ]
for i in range(N_POWERS):
    powers_of_ten.append(10 ** i)
```

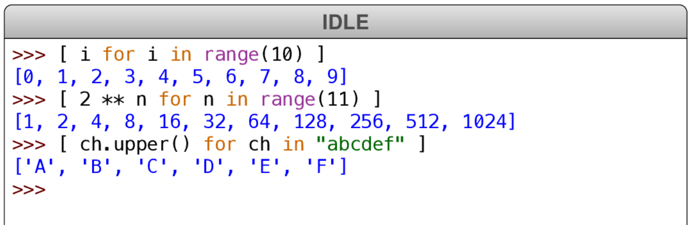
Although this code is already reasonably concise, Python supports an even more compact syntactic form called a ***list comprehension,*** which allows the programmer to specify a `for` loop inside the square brackets, thereby combining the processes of creating the list and assigning the elements. The code to initialize `powers_of_ten` using list comprehension looks like this:

```
powers_of_ten = [ 10 ** i for i in range(N_POWERS) ]
```

In its simplest form, list comprehension uses the syntactic pattern

```
[ exp for var in iterable ]
```

which returns a list consisting of the value of *exp* repeated for each of the cycles of the `for` loop. The following IDLE session shows three examples of this pattern:



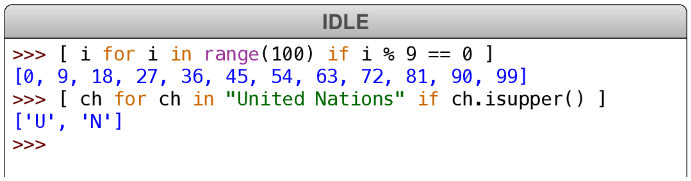
IDLE

```
>>> [ i for i in range(10) ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [ 2 ** n for n in range(11) ]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
>>> [ ch.upper() for ch in "abcdef" ]
['A', 'B', 'C', 'D', 'E', 'F']
>>>
```

You can also include an `if` clause in the list-comprehension pattern to select elements that satisfy some condition. Using this option, the pattern looks like this:

```
[ exp for var in iterable if condition ]
```

Once again, the simplest way to illustrate the use of this pattern is with a couple of examples in an IDLE session:



IDLE

```
>>> [ i for i in range(100) if i % 9 == 0 ]
[0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99]
>>> [ ch for ch in "United Nations" if ch.isupper() ]
['U', 'N']
>>>
```

It is always possible to expand a list comprehension into a `for` statement that produces a variable with the same value. For example, the pattern

```
[ exp for var in iterable if condition ]
```

produces a value that is the same as the contents of the variable `seq` after executing the following code:

```
seq = [ ]
for var in iterable:
    if condition:
        seq.append(exp)
```

Even though it is possible to achieve the same result using statements that you already know, using list comprehensions offer the following advantages:

- *List comprehensions are shorter.* The expanded form of this pattern requires four lines to accomplish what the list-comprehension form manages in one.
- *List comprehensions are typically more efficient.* Python interpreters are usually able to optimize the performance of a list comprehension more effectively than code written in an expanded form. In particular, Python can often allocate space for the list at the beginning rather than having to add an element on each cycle.
- *List comprehensions can be embedded as expressions.* A list comprehension is a standalone expression that can be used as part of a more general Python computation. For example, you can pass a list comprehension as an argument to a function without having to create a variable to hold the result.
- *List comprehensions are more likely to match algorithmic descriptions.* Formal descriptions of algorithms are often written in a form that is easily translated into a list comprehension.

List comprehensions in Python are actually more powerful than the examples here suggest. A single list comprehension can include several nested `for` loops and any number of `if` clauses. The simple pattern, however, covers the vast majority of applications you are likely to encounter in practice.

8.4 Using lists for tabulation

The data structure of a program is typically designed to reflect the organization of data in the real-world domain of the application. In general, whenever an application involves data that can be represented in the form of a list with elements a_0, a_1, a_2, and so on, a list is the natural choice for the underlying representation. It is also quite common for programmers to refer to the index of a list element as a ***subscript,*** reflecting the fact that lists are used to hold data that would typically be written with subscripts in mathematics.

There are, however, important uses of lists in which a different relationship exists between the data in the application domain and the data in the program. This section shows how you can use a list to count how many times each character in the alphabet appears in a string by using the Unicode values of each character to derive an index position.

The exercises for Chapter 7 ask you to implement a letter-substitution cipher, which encrypts a message by replacing each letter with an encoded version of that letter determined through the use of a secret key. Although *implementing* a letter-substitution cipher is a good exercise, a more interesting problem is figuring out how to *break* a letter-substitution cipher if you do not have access to the key.

The problem of breaking a letter-substitution cipher often appears in recreational puzzles called ***cryptograms.*** Edgar Allan Poe was a great fan of cryptograms and described a technique for solving them in his 1843 novel *The Gold Bug:*

Edgar Allan Poe

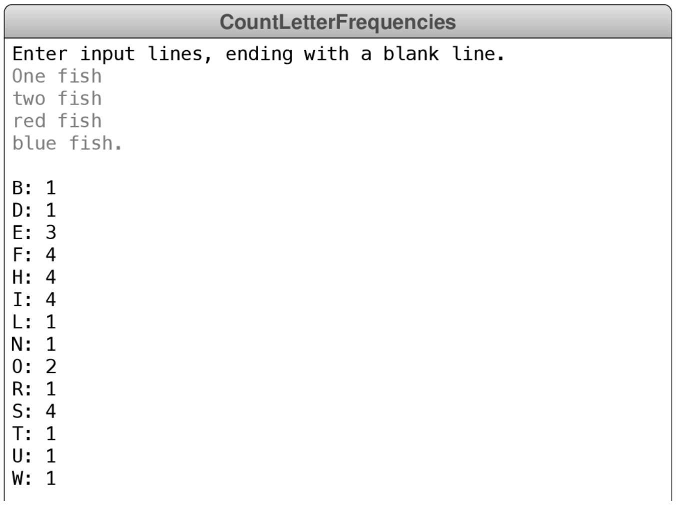
> My first step was to ascertain the predominant letters. . . . Now, in English, the letter which most frequently occurs is *e*. Afterwards, the succession runs thus: *a o i d h n r s t u y c f g l m w b k p q x z*. *E* however predominates so remarkably that an individual sentence of any length is rarely seen, in which it is not the prevailing character.

As it happens, Poe's list of the most common letters is by no means accurate. Computerized analysis reveals that the 12 most common letters in English are

E T A O I N S H R D L U

Given that computerized analyses of English text were not available in his day, Poe can perhaps be excused for making a few mistakes. Poe was, however, entirely correct in his claim that the first step in discovering the hidden meaning of a cryptogram is to construct a table showing how often each letter appears. A program that does just that appears in Figure 8-2 on the next page.

The following sample run shows the output of `CountLetterFrequencies.py` using the first page of *One Fish, Two Fish, Red Fish, Blue Fish* as input:



CountLetterFrequencies

```
Enter input lines, ending with a blank line.
One fish
two fish
red fish
blue fish.

B: 1
D: 1
E: 3
F: 4
H: 4
I: 4
L: 1
N: 1
O: 2
R: 1
S: 4
T: 1
U: 1
W: 1
```

The output shows that the file contains four copies of the letters *F, I, S,* and *H*, three *E*'s, two *O*'s, and a smattering of letters that each appear exactly once. Note that letters that never appear in the input are not shown in the output.

FIGURE 8-2 **Program to count letter frequencies in lines entered by the user**

```python
# File: CountLetterFrequencies.py

"""This program tabulates the frequencies of letters in user input."""

def count_letter_frequencies():
    counts = create_frequency_table()
    print("Enter input lines, ending with a blank line.")
    finished = False
    while not finished:
        line = input()
        if line == "":
            finished = True
        else:
            update_frequency_table(counts, line)
    print_frequency_table(counts)

def create_frequency_table():
    """Creates an empty frequency table."""
    return [ 0 ] * 26

def update_frequency_table(counts, source):
    """Updates the frequency table by scanning the source string."""
    for ch in source:
        if ch.isalpha():
            counts[ord(ch.upper()) - ord("A")] += 1

def print_frequency_table(counts):
    """Prints a frequency table using the data from counts."""
    for i in range(len(counts)):
        count = counts[i]
        if count > 0:
            ch = chr(ord("A") + i)
            print(f"{ch}: {count}")

# Test function

def test_count_letter_frequencies():
    counts = create_frequency_table()
    for n in counts:
        assert n == 0
    update_frequency_table(counts, "hello, world")
    assert counts[ord("D") - ord("A")] == 1
    assert counts[ord("E") - ord("A")] == 1
    assert counts[ord("H") - ord("A")] == 1
    assert counts[ord("L") - ord("A")] == 3
    assert counts[ord("O") - ord("A")] == 2
    assert counts[ord("R") - ord("A")] == 1
    assert counts[ord("W") - ord("A")] == 1

# Startup code

if __name__ == "__main__":
    count_letter_frequencies()
```

The implementation strategy used in CountLetterFrequencies.py is to create a list of 26 integers in which each index position contains the current count of the letter in the alphabet corresponding to that index. The element at the beginning of the list contains the number of *A*'s, and the element at the end of the list contains the number of *Z*'s. The code then subdivides the tasks of initializing, updating, and printing out the letter-frequency data contained in that list into three functions. The function create_frequency_table creates a list in which each of the 26 letter counts is set to 0. The update_frequency_table function then updates the contents of the list by running through every character in a string and incrementing the count associated with each letter. When all the updates have been made, the program calls the function print_frequency_table to display the results.

Of these functions, create_frequency_table is by far the simplest. All it has to do is return a list with 26 elements, each of which is set to 0. The following function definition uses Python's repetition operator to accomplish this task:

```
def create_frequency_table():
    return [ 0 ] * 26
```

The code for the count_letter_frequencies function assigns this result to the variable counts, which can be diagrammed as follows:

counts

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Each time a letter appears in the input, you need to increment the corresponding element in counts, which occurs in update_frequency_table. As you can see from Figure 8-2, update_frequency_table uses the standard for-loop pattern to iterate through the characters in line and then executes the following statement for each character that passes the isalpha test, marking it as a letter:

```
counts[ord(ch.upper()) - ord("A")] += 1
```

This statement is sufficiently complex that it is worth going through it in some detail. When Python executes this statement, it has already determined that ch is a letter, but it might be an uppercase letter or a lowercase one. Calling ch.upper() produces the uppercase value. Calling ord on this character returns the Unicode value for the character. Subtracting the Unicode value for the character "A" produces the desired index in the counts array, which is then incremented.

The code for display_frequency_table performs the same conversion in the opposite direction. The values of i in the for loop run from 0 to 25. To convert that index into a character requires the following code:

```
ch = chr(ord("A") + i)
```

8.5 Using files

In a practical application to count letter frequencies, you would not want the user to have to enter the text of each line but would instead like to read the data from a input file. Before you can change `CountLetterFrequencies.py` to take its input from a file, you need know more about how Python works with files.

The concept of a file

Programs use variables to store information: input data, calculated results, and any intermediate values generated along the way. The information in variables, however, is ephemeral and disappears when the program stops running. For many applications, it is important to store data in a more permanent way.

Whenever you want to store information on the computer for longer than the running time of a program, the usual approach is to store that information in a ***file,*** which in computing contexts refers to any collection of data stored in electronic form and distinguished from other files by an identifying name. Files are ordinarily stored on your computer's hard disk but can also reside on the network or on a removable storage device, such as a flash memory drive. Files also come in a variety of types. Computers use files to store music, images, movies, formatted documents, statistical information, and a wide variety of other data types. The most common type of file—and the only one considered in this book—is a ***text file,*** which contains a sequence of characters.

To get a sense of how you might use text files, suppose that you want to collect your favorite quotations from Shakespeare and store them on your computer. One approach is to store each quotation in a separate file. You might begin your collection with the following lines from *Hamlet:*

`Hamlet.txt`

```
To be, or not to be: that is the question.
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them?
```

For your second quotation, you might choose the following lines from *Romeo and Juliet:*

`Juliet.txt`

```
What's in a name?
That which we call a rose
By any other name would smell as sweet.
```

These diagrams show the name outside the file, in much the same way that variable diagrams show the variable name on the outside and the value on the inside.

When you look at a file, it often makes sense to regard it as a two-dimensional structure—a sequence of lines composed of individual characters. Internally, however, text files are represented as a continuous sequence of characters. In addition to the printing characters you can see, files also contain newline characters that mark the end of each line.

Suppose, for example, that you have also created a file called `Macbeth.txt` containing the following rhymed couplet:

Macbeth.txt

```
A drum, a drum!
Macbeth doth come.
```

Although it is entirely reasonable to think of this file as containing two lines of characters, it is important to remember that the internal structure of the file is represented as a single sequence of characters, like this:

Macbeth.txt

A		d	r	u	m	,		a		d	r	u	m	!	\n	M	a	c	b	e	t	h		d	o	t	h		c	o	m	e	.	\n

As this most recent example makes clear, a text file is similar to a string. Both are ordered sequences of characters. The critical differences are that

- *The information stored in a file is permanent.* The value of a string variable persists only as long as the variable does. All program variables disappear when the program exits. Information stored in a file exists until the file is deleted.
- *Files are usually read sequentially.* When you read data from a file, you usually start at the beginning and read through them in order, typically line by line. You read the first line, then the second, and so on until you reach the end of the file.

Reading text files

The standard pattern for reading a text file in Python uses a new feature called the `with` statement, the details of which are beyond the scope of this book. Even without knowing precisely what it does, you can use the `with` statement in the file-reading pattern, which looks like this:

```
with open(filename) as handle:
    Call methods on the file handle to read the data
```

In this pattern, *filename* specifies the name of the file, and *handle* is a variable used to hold a reference to the file, which is commonly called a ***file handle.*** The statements in the body then call methods on the file handle to read the data. There are several strategies for doing so, including the following:

1. Reading the entire file as a single string
2. Reading the file one line at time, processing each line as you go
3. Reading the entire file into a list of strings and then working with that list

The rest of this section describes each of these strategies in more detail.

In many ways, the simplest approach is to read the entire file as a single string by calling the `read` method on the file handle. The return value includes the embedded newline characters. For example, if you execute the code

```
with open("Macbeth.txt") as f:
    text = f.read()
```

the variable `text` is set to the entire contents of the file, like this:

```
text
"A drum, a drum!\nMacbeth doth come.\n"
```

The disadvantage of using the `read` method is that it has to read the entire contents of the file into memory at once. If a file is large, the need to allocate storage for a single string that includes its entire contents can slow your program down. Most computers today have large memories, which means that you are unlikely to run into trouble with this strategy unless you are working with extremely large files.

You can avoid having to read an entire file at once by calling the `readline` method to read the next line from the file. The string returned by the `readline` method includes the newline character at the end of the line. Thus, the code

```
with open("Macbeth.txt") as f:
    line1 = f.readline()
    line2 = f.readline()
```

sets the variables `line1` and `line2` to the two lines in the file, as follows:

```
line1
"A drum, a drum!\n

line2
Macbeth doth come.\n"
```

Calling the `readline` method after you have read the last line in a file returns the empty string. You can therefore process every line in a file by adapting the code for the read-until-sentinel loop presented in Chapter 2 as follows:

```
with open("Macbeth.txt") as f:
    finished = False
    while not finished:
        line = f.readline()
        if line == "":
            finished = True
        else:
            Perform whatever process is required for the line
```

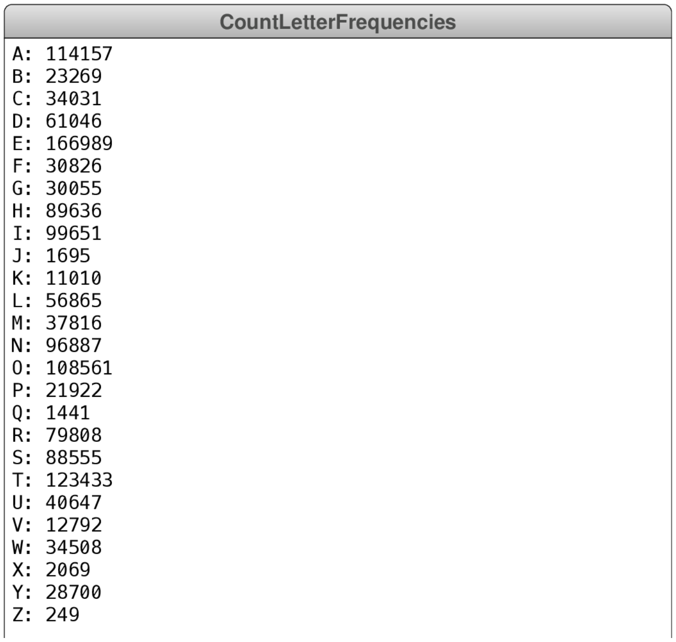
Although this implementation works, the code is hard to read. You can use the fact that file handles are iterable objects to shorten the code considerably, like this:

```
with open("Macbeth.txt") as f:
    for line in f:
        Perform whatever process is required for the line
```

You can combine this pattern with `CountLetterFrequencies.py` to count the letter frequencies in George Eliot's 1871 novel *Middlemarch,* as follows:

```
with open("Middlemarch.txt") as f:
    counts = create_frequency_table()
    for line in f:
        update_frequency_table(counts, line)
    print_frequency_table(counts)
```

Running this program produces the following output:



```
A: 114157
B: 23269
C: 34031
D: 61046
E: 166989
F: 30826
G: 30055
H: 89636
I: 99651
J: 1695
K: 11010
L: 56865
M: 37816
N: 96887
O: 108561
P: 21922
Q: 1441
R: 79808
S: 88555
T: 123433
U: 40647
V: 12792
W: 34508
X: 2069
Y: 28700
Z: 249
```

If you sort this output by letter frequency in descending order, you discover that the 12 most common letters in *Middlemarch* are

E T A O I N H S R D L U

The only difference between this frequency table and the statistical results for modern English presented on page 269 is that the *H* and the *S* are reversed. In general, the more text you analyze, the closer the frequencies will align with those calculated for modern English.

The `CountLetterFrequencies.py` program can accomplish its task without ever having to keep track of more than one line from the file at a time. Other programs, by contrast, may make it necessary to read all the lines of the file before performing any computation. The simplest example is that of an application that reverses the order of the lines that file contains. When you read the first line, for example, you can't yet do anything useful with it. What your program has to do is store that line away so that it can print it at the end.

If you are implementing an application that needs access to all the lines in a file, the usual strategy is to read the contents of the file into a list. Python's file class includes a `readlines` method that works for many applications, but is not ideal for others. The problem with the `readlines` method is that each line includes the newline character that marks the end of the line, which most applications would prefer not to see. And while it is possible to remove these characters by going through each element of the list and stripping off the newline at the end, doing so at the client level is far less efficient than having Python implement this function.

The following pattern offers the simplest strategy for reading a list of lines without the newlines, which is usually exactly what you want:

```
with open(filename) as f:
    lines = f.read().splitlines()
```

The `splitlines` method in the string class removes the newline characters.

Suppose that `WitchesBrew.txt` contains the following lines from *Macbeth:*

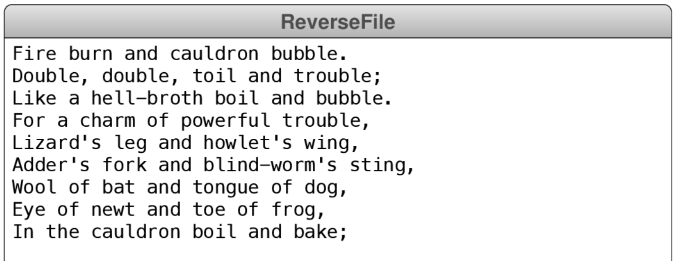
WitchesBrew.txt

```
In the cauldron boil and bake;
Eye of newt and toe of frog,
Wool of bat and tongue of dog,
Adder's fork and blind-worm's sting,
Lizard's leg and howlet's wing,
For a charm of powerful trouble,
Like a hell-broth boil and bubble.
Double, double, toil and trouble;
Fire burn and cauldron bubble.
```

You can display these lines in reverse order using

```
with open("WitchesBrew.txt") as f:
    lines = f.read().splitlines()
    lines.reverse()
    for line in lines:
        print(line)
```

which generates the following output:



```
Fire burn and cauldron bubble.
Double, double, toil and trouble;
Like a hell-broth boil and bubble.
For a charm of powerful trouble,
Lizard's leg and howlet's wing,
Adder's fork and blind-worm's sting,
Wool of bat and tongue of dog,
Eye of newt and toe of frog,
In the cauldron boil and bake;
```

Writing text files

Although reading a file is a more common operation, Python also allows you to write data to a file using the following pattern:

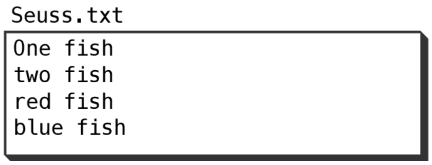
```
with open(filename, "w") as handle:
    Call methods on the file handle to write the data
```

The only difference from the pattern used for reading is that the `open` function takes a second argument that specifies how the file is used. This optional argument to `open` is called the ***mode.*** By default, the `mode` parameter is `"r"`, which signifies that the file should be opened for reading. If you instead specify `"w"`, the file is opened for writing. If the file does not exist, Python creates it; if it does, Python deletes the existing contents of the file and prepares to write new data, just as if the file had been created from scratch. Another useful mode is `"a"`, which opens a file for writing without deleting its contents so that you can append new content.

The body of the `with` statement specifies the code used to write the contents of the file. The most common method is `write`, which writes a string to the file. The following code, for example, creates a file named `Seuss.txt` containing the text from the first page of *One Fish, Two Fish, Red Fish, Blue Fish:*

```
with open("Seuss.txt", "w") as f:
    f.write("One fish\n")
    f.write("two fish\n")
    f.write("red fish\n")
    f.write("blue fish.\n")
```

As this code example illustrates, you must specify line breaks in the file by including the newline character in the string you pass to `write`. The file created by this code looks like this:



```
Seuss.txt
One fish
two fish
red fish
blue fish
```

You can also use the `writelines` method, which writes data from a list of lines.

Exception handling

When you are writing an application that works with files, it is important to keep in mind that the call to `open` might fail. For example, if you request the name of an input file from the user, and the user types the filename incorrectly, the `open` function will be unable to find the mistyped filename. To signal a failure of this sort, the implementation of `open` responds by ***raising an exception,*** which is the phrase Python uses to describe the process of reporting an exceptional condition outside the normal program flow. Python includes a special statement form called `try` that allows you as a programmer to specify an interest in responding to exceptions of a particular type. The simplest form of the `try` statement appears in the syntax box on the left.

```
try:
    body
except type:
    response
```

When Python encounters a `try` statement, it executes the statements in the body. Those statements, of course, may call other functions, which may in turn call other functions, as the computation descends through the levels that implement its decomposition strategy. In any function nested at any level inside the body raises an exception, Python stops its normal execution and looks to see whether any function in the chain of callers has used a `try` statement that responds to exceptions of that type. If so, Python executes the `except` clause, which defines the response to that exception.

The following definition offers a simple illustration of the `try` statement in the form of a function that asks the user to specify the name of a file:

```
def open_input_file(prompt="Input file: "):
    while True:
        filename = input(prompt)
        try:
            with open(filename):
                return filename
        except IOError:
            print("That file cannot be opened")
```

If the user-specified file exists, open_input_file returns its name. If not, the function continues to ask for a file name until the user enters one that succeeds.

The code for open_input_file includes a couple of details that are worth noting. First, the with statement only opens the file to see whether it exists and never reads any data from the file, which means there is no need to store the file handle. Second, the except clause specifies the exception type IOError, which is the type that Python uses to report any errors in the file package. Python defines many other error types, but those are beyond the scope of this book.

Choosing files interactively

If you are creating a general-purpose application, it doesn't make sense to include the name of the file explicitly in the code as in the examples you have seen so far. What you want to do instead is have Python open a file dialog that allows the user to select a file. The modules supplied with this book include a filechooser library that lets the user select files interactively. Figure 8-3 illustrates the use of the filechooser module by creating a CountLetterFrequenciesInFile.py application that adds interactive file selection on top of the earlier version of the code. The functions that perform the actual counts are imported from the CountLetterFrequencies.py application in Figure 8-2.

FIGURE 8-3 **Program to count letter frequencies in a file selected using a dialog**

```
# File: CountLetterFrequenciesInFile.py

"""This program counts the frequencies of letters in a file."""

from filechooser import choose_input_file
from CountLetterFrequencies import create_frequency_table
from CountLetterFrequencies import update_frequency_table
from CountLetterFrequencies import print_frequency_table

def count_letter_frequencies_in_file():
    filename = choose_input_file()
    if filename != "":
        with open(filename) as f:
            counts = create_frequency_table()
            for line in f:
                update_frequency_table(counts, line)
            print_frequency_table(counts)

# Startup code

if __name__ == "__main__":
    count_letter_frequencies_in_file()
```

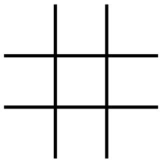
The `filechooser` library exports two functions: `choose_input_file` and `choose_output_file`. Each of these functions pops up a dialog, which then allows the user to move through the directory structure to select a particular file. Double-clicking on the filename or highlighting a file and then clicking the **Open** or **Save** button completes the action of the dialog and returns the complete pathname of the selected file to the calling function. Clicking the **Cancel** button dismisses the dialog and returns the empty string to the caller.

8.6 Multidimensional arrays

In Python, the elements of a list can be of any type. In particular, the elements of a list can themselves be lists. Lists of lists are used to model the concept of a ***multidimensional array,*** which is an array in which the elements are laid out in more than one dimension and therefore require more than one index to select an individual element.

The most common form of multidimensional array is the two-dimensional array, which is most often used to represent data in which the individual entries form a rectangular structure marked off into rows and columns. This type of two-dimensional structure is called a ***matrix.*** Arrays with three or more dimensions are also legal in Python but occur less frequently.

As an example of a two-dimensional array, suppose you want to represent a game of Tic-Tac-Toe as part of a program. As you probably know, Tic-Tac-Toe is played on a board consisting of three rows and three columns, as follows:



Players take turns placing the letters *X* and *O* in the empty squares, trying to line up three identical symbols horizontally, vertically, or diagonally.

The most natural strategy for representing the Tic-Tac-Toe board is to use a two-dimensional array with three rows and three columns. Each of the elements is a string, which must be one of the following: `""` (representing an empty square), `"X"`, and `"O"`. Since the board is initially empty, you can initialize it like this:

```
board = [ [ "" ] * 3 for i in range(3) ]
```

Given this declaration, you can refer to the characters in the individual squares by supplying two indices, one specifying the row number and another specifying the column number. In this representation, each number varies over the range 0 to 2, and the individual positions on the board have the following designations:

board[0][0]	board[0][1]	board[0][2]
board[1][0]	board[1][1]	board[1][2]
board[2][0]	board[2][1]	board[2][2]

It is important to note that you can't initialize board like this:

```
board = [ [ "" ] * 3 ] * 3
```

While this statement might seem to have the same effect as the earlier one, the three rows of this board are the same object, so that you can't change one element without having that change reflected in the other rows.

8.7 Image processing

In modern computing, one of the most important applications of two-dimensional arrays occurs in the field of computer graphics. As you learned in Chapter 4, graphical images are composed of individual pixels. Figure 4-3 on page 104 offers a magnified view of the screen that shows how the pixels create the image as a whole. Those images are most easily represented using two-dimensional arrays.

The GImage class

The Portable Graphics Library defines the GImage class as a graphical object that contains image data encoded using one of the standard formats. The three most common are the Portable Network Graphics (PNG) format, the Joint Photographic Experts Group (JPEG) format, and the Graphics Interchange Format (GIF).

Displaying an image requires two steps. The first is to create or download an image file. The name of the image file should end with an extension that identifies the encoding format, which is.png for the images in this book. The second step is to create a GImage object and add it to the graphics window, just as you would with any other graphical object. For example, if you have an image file called MyImage.png, you can display that image in the upper left corner of the graphics window using the following line:

```
gw.add(GImage("MyImage.png"))
```

If you want to center an image in the graphics window, you must first create the GImage and then use its size to determine where you need to position it, much as you do with a GLabel. This technique is illustrated by the following code:

```
image = GImage("MyImage.png")
x = (gw.get_width() - image.get_width()) / 2
y = (gw.get_height() - image.get_height()) / 2
gw.add(image, x, y)
```

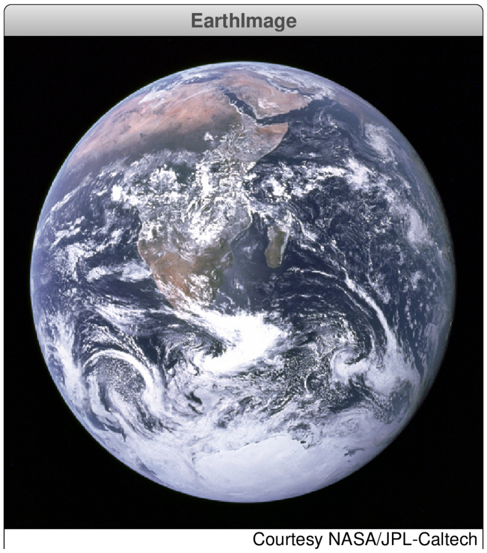
In many cases, you will find that you want to display an image at a different size on the screen from the size that appears in the file. The easiest way to do so is to use the scale method, which changes the size of a GObject by the specified scale factor. If image contains a GImage, calling image.scale(0.5) makes it half as big in each dimension. Similarly, calling image.scale(2) doubles its size.

The code for the EarthImage.py program in Figure 8-4 at the top of the next page illustrates the use of scaling to display an image so that it fills the available space. The image, which shows the earth as seen by the Apollo 17 astronauts on their way to the moon in December 1972, is stored in an image file named EarthImage.png. The EarthImage.py program reads that image file into a GImage object and then adds that object to the window. The line

```
image.scale(gw.get_width() / image.get_width())
```

scales the image so that it fills the entire width of the window.

Running the EarthImage.py program produces the following display:



Courtesy NASA/JPL-Caltech

FIGURE 8-4 **Program to draw an image of the earth taken from Apollo 17**

```
# File: EarthImage.py

"""
This program draws a picture of the earth taken by Apollo 17 along with
a photo credit.
"""

from pgl import GWindow, GImage, GLabel

# Constants

GWINDOW_WIDTH = 350
GWINDOW_HEIGHT = 365
CITATION_FONT = "12px 'Helvetica Neue'"
CITATION_Y = 3

def earth_image():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    image = GImage("images/EarthImage.png")
    image.scale(gw.get_width() / image.get_width())
    gw.add(image, 0, 0)
    citation = GLabel("Courtesy NASA/JPL-Caltech ")
    citation.set_font(CITATION_FONT)
    x = gw.get_width() - citation.get_width()
    y = gw.get_height() - CITATION_Y
    gw.add(citation, x, y)

# Startup code

if __name__ == "__main__":
    earth_image()
```

This screenshot also illustrates the inclusion of a citation along with an image. When you use existing images, you need to be aware of possible restrictions on the use of intellectual property. Most of the images you find on the web are protected by copyright. Under copyright law, you must obtain the permission of the copyright holder in order to use the image, unless your use of the image satisfies the guidelines for “fair use”—a doctrine that has unfortunately become more murky in the digital age. Under “fair use” guidelines, you could almost certainly use a copyrighted image in a paper that you write for a class. On the other hand, you could not put that same image into a commercially published work without first securing—and probably paying for—the right to do so.

Even in cases in which your use of an image falls within the “fair use” guidelines, it is important to give proper credit to the source. As a general rule, whenever you find an image on the web that you would like to use, you should first check to see whether that website explains its usage policy. Many of the best sources for images on the web have explicit guidelines for using their images. Some images are absolutely free, some are free for use with citation, some can be used in certain contexts but not others, and some are completely restricted. For example, the website for the National Aeronautics and Space Administration (`https://www.nasa.gov`) has an extensive library of images about the exploration of space. As the website explains, you can use these images freely as long as you include the citation "Courtesy NASA/JPL-Caltech" along with the image. The `EarthImage.py` program follows these guidelines and includes the requested citation on the graphics window.

Representation of images

In Python, an image is a two-dimensional array in which the image as a whole is a sequence of rows, and each row is a sequence of individual pixel values. The value of each element indicates the color that should appear in the corresponding pixel position on the screen. From Chapter 4, you know that you can specify a color in Python by indicating the intensity of each of the primary colors. Each of those intensities ranges from 0 to 255 and therefore fits in an eight-bit byte. The color as a whole is stored in a 32-bit integer that contains the red, green, and blue intensity values along with a measure of the transparency of the color, represented by the Greek letter alpha (α). For the opaque colors used in most images, the value of α is always 255 in decimal, which is **11111111** in binary or **FF** in hexadecimal.

As an example, the following diagram shows the four bytes that form the color `"Pink"`, which Python defines using the hexadecimal values **FF**, **C0**, and **CB** as the red, green, and blue components. Translating those values to their binary form gives you the following:

α	*red*	*green*	*blue*
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 0 0 0 0 0 0	1 1 0 0 1 0 1 1

The fact that Python packs all the information about a color into a 32-bit integer means that you can store an image as a two-dimensional array of integers. Each element of the entire array contains one row of the image. In keeping with Python's coordinate system, the rows of an image are numbered from 0 starting at the top. Each row is an array of integers representing the value of each pixel as you move from left to right.

Using the `GImage` class to manipulate images

The `GImage` class in the graphics library exports several methods that make it possible to perform basic image processing. As long as certain conditions are met concerning the source of the image, you can obtain the two-dimensional array of pixel values by calling `get_pixel_array`. Thus, if the variable `image` contains a `GImage`, you can retrieve its pixel array by calling

```
array = image.get_pixel_array()
```

The height of the image is equal to the number of rows in the pixel array. The width is the number of elements in any of the rows, each of which has the same length in a rectangular image. Thus, you can initialize variables to hold the height and width of the pixel array like this:

```
height = len(array)
width = len(array[0])
```

If you need to create a new pixel array with dimensions `width` and `height`, the best approach is to use list comprehension like this:

```
array = [ [ 0 for i in height ] for j in width ]
```

The `GImage` class includes several methods to simplify the task of manipulating image data. These methods appear in Figure 8-5. As you can see from the first section of the figure, the `GImage` class supports two constructors, one for reading data from a file and one to construct a `GImage` from a two-dimensional array. Given an initialized image, the `get_pixel_array` method returns the array of pixels stored within the image. The `GImage` class also exports class methods for retrieving the red, green, and blue components of a pixel from an integer and for assembling red, green, and blue values into the corresponding integer form.

FIGURE 8-5 Useful methods in the `GImage` class

Constructors to create a `GImage`

`GImage(`*filename*`)`	Creates a new `GImage` by reading the image data from the specified file.
`GImage(`*array*`)`	Creates a new `GImage` from the pixel array.

Method to read the individual pixels in an image

image`.get_pixel_array()`	Returns the pixel array for this image.

Static methods

`GImage.get_red(`*pixel*`)`	Returns the red component of the pixel as an integer between 0 and 255.
`GImage.get_green(`*pixel*`)`	Returns the green component of the pixel as an integer between 0 and 255.
`GImage.get_blue(`*pixel*`)`	Returns the blue component of the pixel as an integer between 0 and 255.
`GImage.create_rgb_pixel(`*r*`,` *g*`,` *b*`)`	Creates a pixel value from the specified *r, g,* and *b* components, each of which is between 0 and 255.

These new capabilities in the `GImage` class make it possible for you to write programs to manipulate images, in much the same way that a commercial system like Adobe Photoshop™ does. The general strategy looks like this:

1. Use `get_pixel_array` to obtain the array of pixel values.
2. Perform the desired transformation by manipulating the values in the array.
3. Call the `GImage` function to create a new object from the modified array.

The following function definition uses this pattern to flip an image vertically:

```
def flip_vertical(image):
    array = image.get_pixel_array()
    array.reverse()
    return new GImage(array)
```

A more substantive problem is that of converting a color image to ***grayscale,*** a format in which all the pixels are either black, white, or some intermediate shade of gray. To do so, you need to go through each element in the pixel array and replace each pixel with a shade of gray that approximates the apparent brightness of that color. In computer graphics, that apparent brightness is called ***luminance.***

The goal of a grayscale conversion is to produce a shade of gray that approximates the brightness of each pixel to the eye. As it turns out, luminance is controlled much more strongly by how much green appears in the pixel than by the amount of red or blue. Red and blue tend to make an image appear darker, while green tends to lighten it up. The formula for luminance adopted by the standards committee responsible for television signals in the United States looks like this:

$$\textit{luminance} = 0.299 \times \textit{red} + 0.587 \times \textit{green} + 0.114 \times \textit{blue}$$

A complete program to produce a grayscale image appears in Figure 8-6 on the next page. The main program begins by allowing the user to choose an image file. It then loads the image and displays the original and grayscale images side by side:

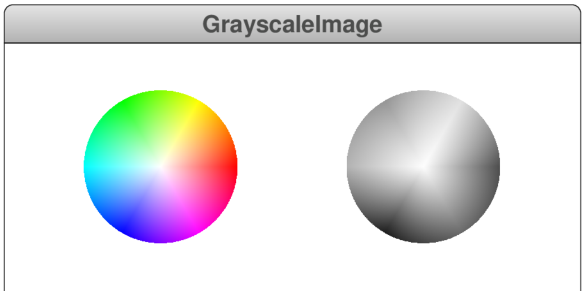


FIGURE 8-6 **Program to convert an image to grayscale**

```
# File: GrayscaleImage.py

"""This program displays an image together with its grayscale version."""

from pgl import GWindow, GImage

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 400
IMAGE_FILENAME = "images/ColorWheel.png"
IMAGE_SEP = 50

# Main program

def grayscale_image():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    image = GImage(IMAGE_FILENAME)
    gw.add(image, (gw.get_width() - IMAGE_SEP) / 2 - image.get_width(),
                  (gw.get_height() - image.get_height()) / 2)
    grayscale = create_grayscale_image(image)
    gw.add(grayscale, (gw.get_width() + IMAGE_SEP) / 2,
                      (gw.get_height() - image.get_height()) / 2)

def create_grayscale_image(image):
    """Creates a grayscale image based on the luminance of each pixel."""
    array = image.get_pixel_array()
    height = len(array)
    width = len(array[0])
    for i in range(height):
        for j in range(width):
            gray = luminance(array[i][j])
            array[i][j] = GImage.create_rgb_pixel(gray, gray, gray)
    return GImage(array)

# Implementation notes: luminance
# -------------------------------
# This function returns the subjective brightness of a pixel value,
# which depends differently on the three color components.  For example,
# a high green component makes a pixel appear brighter that the same
# value for the other components.  The luminance formula is a standard
# adopted by the NTSC.

def luminance(pixel):
    """Returns the luminance of a pixel."""
    r = GImage.get_red(pixel)
    g = GImage.get_green(pixel)
    b = GImage.get_blue(pixel)
    return round(0.299 * r + 0.587 * g + 0.114 * b)

# Startup code

if __name__ == "__main__":
    grayscale_image()
```

Summary

In this chapter, you have learned how to use *lists,* which are the data structure that Python uses to represent an ordered collection of data. The important points introduced in this chapter include:

- Most programming languages define a type for storing a sequence of elements. Historically, this type of sequential structure is called an *array.* Python supports the idea of an array using a more general structure called a *list* that implements additional operations.
- Each element in a list is identified by an integer index that indicates its position in the list. Index numbers begin with 0.
- Python lists are most often created by enclosing a list of the elements in square brackets, separated by commas.
- You can select a particular element of a list by indicating the index of the desired element in square brackets after the list name. This operation is called *selection.*
- Both lists and strings are examples of a more general class of objects called *sequences,* which means that these classes share a set of common operations associated with sequences.
- Lists in Python are stored as *references* to the memory containing the values of the list. An important implication of this design is that passing a list as a parameter means that the function ands its caller see the same list of elements.
- Lists support a variety of operations implemented as methods. The most important list methods are listed in Figure 8-1 on page 245.
- Python includes a syntactic form called *list comprehension,* which makes it easy to create a list using a `for` loop and an optional conditional test.
- A *file* is a collection of data stored in electronic form and distinguished from other files by an identifying *filename*.
- Files can contain data of different types. This book works only with *text files,* which are sequences of characters.
- The `open` method creates a *file handle,* which you can then use to read data from or write data to a file.
- The methods `read`, `readline`, and `readlines` are useful in reading data from a text file; the methods `write` and `writelines` are used to write data to a file.
- Python's `try-except` statement makes it possible for programs to respond to exceptional conditions, such as a requested input file that doesn't exist.
- Python supports *multidimensional arrays* with any number of subscripts. Those arrays are represented as lists of lists.

- Images are represented as two-dimensional arrays of integers, each of which specifies the red, green, and blue components of the pixel color.
- The Portable Graphics Library includes a class called `GImage` that supports images in a way that gives clients access to the underlying pixel array.

Review questions

1. Define the following terms as they apply to lists: *element, index, length,* and *selection.*

2. How would you create a list called `dwarves` containing the names of the 13 dwarves who arrived at Bilbo's doorstep in J. R. R. Tolkien's fantasy *The Hobbit?* Their names, in the order in which they appeared, are Dwalin, Balin, Kili, Fili, Dori, Nori, Ori, Oin, Gloin, Bifur, Bofur, Bombur, and Thorin.

3. How do you determine the length of a list?

4. True or false: Lists violate the rules for parameter passing by sharing values rather than copying them.

5. Describe the Python syntax for *list comprehension.* What are the advantages of using this syntactic shorthand?

6. What are the principal differences between a *text file* and a *string?*

7. Suppose that you have a variable `filename` that contains the name of a file. What code would you use to read the contents of that file as a single string?

8. If you are using the `readline` method to read a file line by line, how do you tell the difference between a blank line in the file and the end of the file?

9. What is an *exception?*

10. What is the general form of the `try` statement?

11. What is a multidimensional array?

12. In your own words, explain why the text attaches a bug symbol to the following code, which is attempting to initialize a Tic-Tac-Toe board to a 3 × 3 matrix, each of whose elements is the empty string:

    ```
    board = [ [ "" ] * 3 ] * 3
    ```

    What is the correct statement for accomplishing this task?

13. What class from the graphics library makes it possible to display images on the graphics window?

14. Describe how images are represented internally.

15. How do you extract the pixel array from an image?

16. Given a pixel array, how do you determine the width and height of the image?

Exercises

1. In statistics, a collection of data values is usually referred to as a ***distribution.*** A primary purpose of statistical analysis is to find ways to compress the complete set of data into summary statistics that express properties of the distribution as a whole. The most common statistical measure is the ***mean*** (traditionally denoted by the Greek letter μ), which is simply the traditional average. Another common measure is the ***standard deviation*** (traditionally denoted as σ), which provides an indication of how much the values in a distribution $x_1, x_2, \ldots, x_n$ differ from the mean. If you are computing the standard deviation of a complete distribution as opposed to a sample, the standard deviation can be expressed as follows:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (\mu - x_i)^2}{n}}$$

The uppercase sigma (Σ) indicates a summation of the quantity that follows, which in this case is the square of the difference between the mean and each individual data point.

Create a library module called `stats` that exports the functions `mean` and `stdev`, each of which takes a list of numbers representing a distribution and returns the corresponding statistical measure. Make sure that the comments are sufficient for clients to understand how to use these functions.

2. Implement a function `create_index_array(n)` that returns a list containing `n` integer elements, each of which is its own index. For example, calling `create_index_array(8)` should return the list

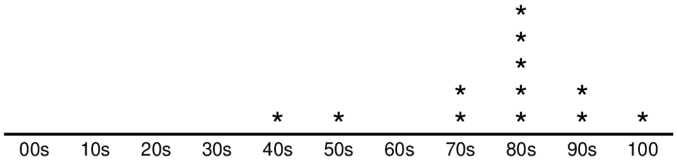
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

3. Use the `list`, `sort`, and `join` methods to write a function `sort_letters` that rearranges the characters in a string so that they appear in lexicographic order. For example, calling `sort_letters("cabbage")` should return the string `"aabbceg"`.

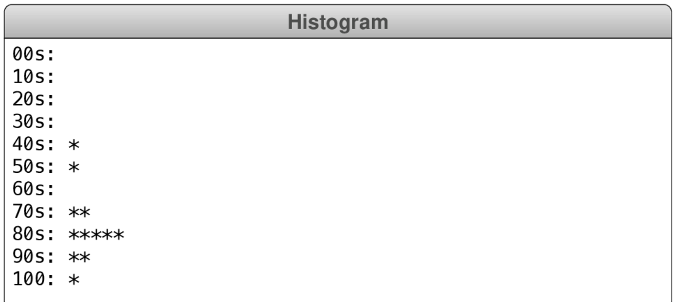
4. A ***histogram*** is a graph that displays a set of values by dividing the data into separate ranges and then indicating how many data values fall into each range. For example, given the set of exam scores

   100, 95, 47, 88, 86, 92, 75, 89, 81, 70, 55, 80

   a traditional histogram would have the following form:



   The asterisks in the histogram indicate one score in the 40s, one in the 50s, five in the 80s, and so forth. When you generate histograms on the console, however, it is easier to display them sideways on the page, like this:



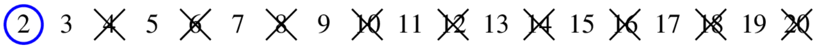
```
00s:
10s:
20s:
30s:
40s: *
50s: *
60s:
70s: **
80s: *****
90s: **
100: *
```

   Write a program called `Histogram` that allows the user to select a file containing exam scores ranging from 0 to 100 and then displays a histogram of those scores, divided into the ranges 0–9, 10–19, 20–29, and so forth, up to the range containing only the value 100. Your function should match the format shown in the sample run as closely as possible.

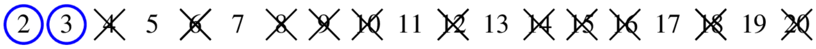
5. In the third century BCE, the Greek astronomer Eratosthenes developed an algorithm for finding all the prime numbers up to some upper limit *N*. To apply the algorithm, you start by writing down a list of the integers between 2 and *N*. For example, if *N* were 20, you would begin by writing the following list:

   2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

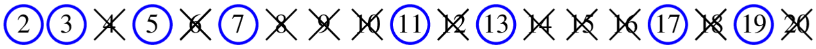
   You then circle the first number in the list, indicating that you have found a prime. Whenever you mark a number as a prime, you go through the rest of the list and cross off every multiple of that number, since none of those multiples can itself be prime. Thus, after executing the first cycle of the algorithm, you will have circled the number 2 and crossed off every multiple of 2, as follows:



To complete the algorithm, you simply repeat the process by circling the first number in the list that is neither crossed off nor circled, and then crossing off its multiples. In this example, you would circle 3 as a prime and cross off all multiples of 3 in the rest of the list, which would result in the following state:



Eventually, every number in the list will either be circled or crossed out, as shown in this diagram:



The circled numbers are the primes; the crossed-out numbers are composites. This algorithm is called the ***sieve of Eratosthenes.***

Write a program that uses the sieve of Eratosthenes to generate a list of the primes between 2 and 1000.

6. Write a function `create_identity_matrix(n)` that returns an `n` × `n` matrix in which the elements are 0 except for the main diagonal in which the value is 1. If you use list comprehensions and the fact that the built-in `int` function converts Boolean values to 0 and 1, the implementation should be a single line.

7. Write a program that uses the `filechooser` library to let the user select an input file and then prints on the console the longest line contained in that file.

8. Modify the program from the preceding exercise so that your program prints out the lines of the selected file, sorted in decreasing order by length.

9.

> *Books were bks and Robin Hood was Rbinhd. Little Goody Two Shoes lost her Os and so did Goldilocks, and the former became a whisper, and the latter sounded like a key jiggled in a lck. It was impossible to read "cockadoodledoo" aloud, and parents gave up reading to their children, and some gave up reading altogether. . . .*
>
> —James Thurber, *The Wonderful O,* 1957

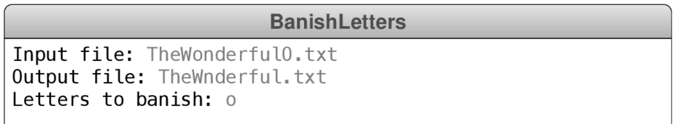
In James Thurber's children's story *The Wonderful O,* the island of Ooroo is invaded by pirates who set out to banish the letter *O* from the alphabet. Such censorship would be much easier with modern technology. Write a program that asks the user for an input file, an output file, and a string of letters to be eliminated. The program should then copy the input file to the output file, deleting any of the letters that appear in the string of censored letters, no matter whether they appear in uppercase or lowercase form.

As an example, suppose that you have a file containing the first few lines of Thurber’s novel, as follows:

TheWonderfulO.txt

```
Somewhere a ponderous tower clock slowly
dropped a dozen strokes into the gloom.
Storm clouds rode low along the horizon,
and no moon shone.  Only a melancholy
chorus of frogs broke the soundlessness.
```

If you run your program with the input



```
Input file: TheWonderfulO.txt
Output file: TheWnderful.txt
Letters to banish: o
```

it should write the following file:

TheWnderful.txt

```
Smewhere a pnderus twer clck slwly
drpped a dzen strkes int the glm.
Strm cluds rde lw alng the hrizn,
and n mn shne.  nly a melanchly
chrus f frgs brke the sundlessness.
```

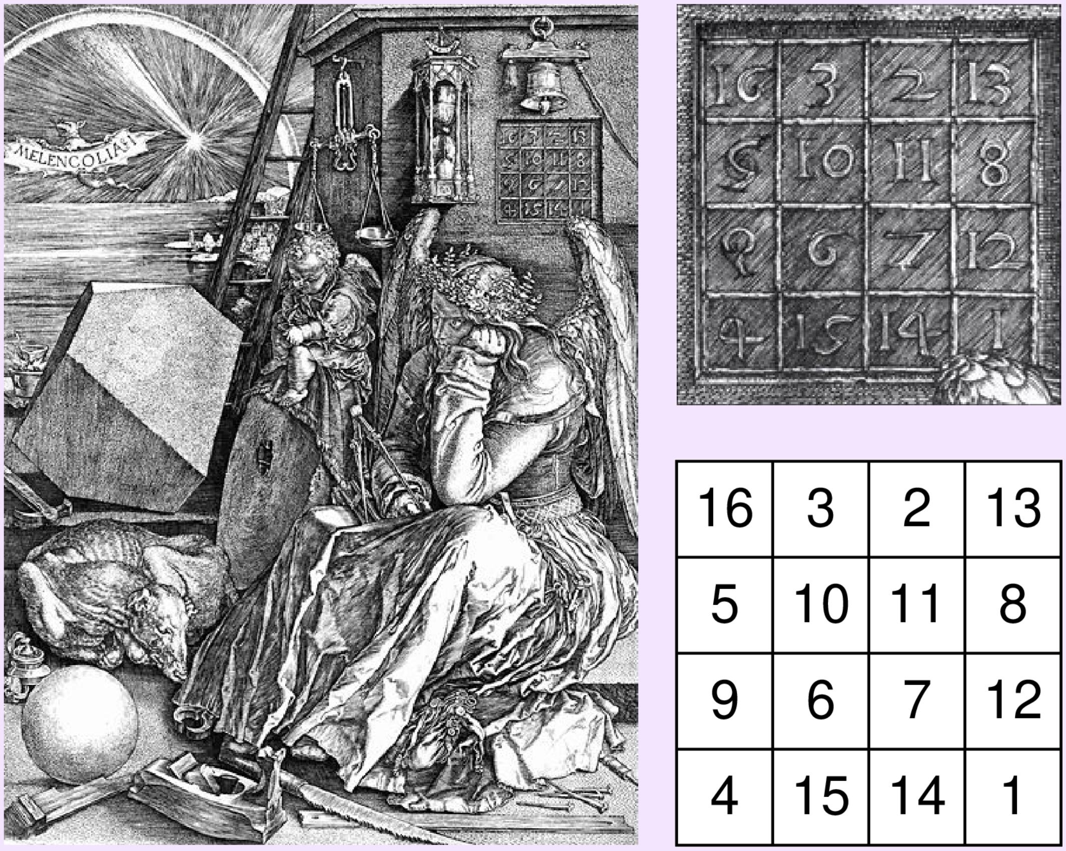
If you get greedy and remove all the vowels by supplying the string "aeiou" as the letters to banish, the contents of the output file corresponding to TheWonderfulO.txt would be

```
Smwhr  pndrs twr clck slwly
drppd  dzn strks nt th glm.
Strm clds rd lw lng th hrzn,
nd n mn shn.  nly  mlnchly
chrs f frgs brk th sndlssnss.
```

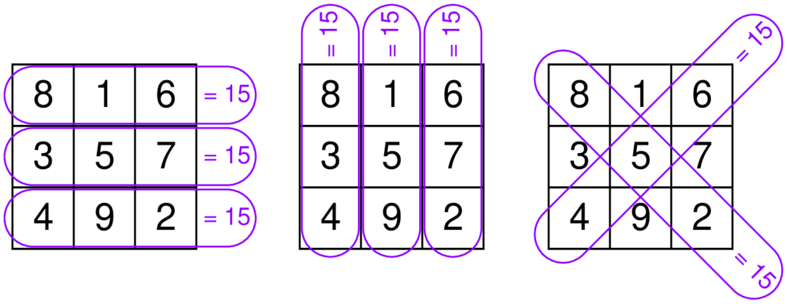
10. A ***magic square*** is a two-dimensional array of integers in which the rows, columns, and diagonals all add up to the same value. One of the most famous magic squares appears in the 1514 engraving *Melencolia I* by Albrecht Dürer shown in Figure 8-7 at the top of the next page, in which a 4×4 magic square appears at the upper right, just under the bell. In Dürer’s square, which can be read more easily in the magnified inset shown at the right of the figure, all four rows, all four columns, and both diagonals add up to 34.

    A more familiar example is the following 3×3 magic square in which each of the rows, columns, and diagonals add up to 15, as shown:

FIGURE 8-7 Magic square in Albrecht Dürer's *Melencolia I* (1514)



16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1



Implement a function `is_magic_square` that takes a two-dimensional array of integers and then tests whether those integers form a magic square. Your function should return `False` if the two-dimensional array is not square.

11. In the game of Minesweeper, a player searches for hidden mines on a rectangular grid that might—for a very small board—look like this:

●					●
					●
●	●		●		●
●					
		●			

One way to represent that grid in Python is to use a list of Boolean values marking mine locations, where `True` indicates the location of a mine. You could, for example, initialize the variable `mine_locations` to this list by writing the following declaration:

```
mine_locations = [
    [  True, False, False, False, False,  True ],
    [ False, False, False, False, False,  True ],
    [  True,  True, False,  True, False,  True ],
    [  True, False, False, False, False, False ],
    [ False, False,  True, False, False, False ],
    [ False, False, False, False, False, False ]
]
```

Write a function `count_mines` that takes a two-dimensional Boolean array representing the location of the mines and returns a new array with the same dimensions that indicates how many mines are in the neighborhood of each location. If a location contains a mine, the corresponding entry in the matrix returned by `count_mines` should be –1. Thus, the assignment statement

```
counts = count_mines(mine_locations)
```

should initialize `counts` as follows:

−1	1	0	0	2	−1
3	3	2	1	4	−1
−1	−1	2	−1	3	−1
−1	4	3	2	2	1
1	2	−1	1	0	0
0	1	1	1	0	0

FIGURE 8-8 Typical Sudoku puzzle and its solution

		2	**4**		**5**	**8**		
	4	**1**	**8**				**2**	
6				**7**			**3**	**9**
2				**3**			**9**	**6**
		9	**6**		**7**	**1**		
1	**7**			**5**				**3**
9	**6**			**8**				**1**
	2				**9**	**5**	**6**	
		8	**3**		**6**	**9**		

3	9	**2**	**4**	6	**5**	**8**	1	7
7	**4**	**1**	**8**	9	3	6	**2**	5
6	8	5	2	**7**	1	4	**3**	**9**
2	5	4	1	**3**	8	7	**9**	**6**
8	3	**9**	**6**	2	**7**	**1**	5	4
1	**7**	6	9	**5**	4	2	8	**3**
9	**6**	7	5	**8**	2	3	4	**1**
4	**2**	3	7	1	**9**	**5**	**6**	8
5	1	**8**	**3**	4	**6**	**9**	7	2

12. Over the last couple of decades, a logic puzzle called *Sudoku* has become popular throughout the world. In Sudoku, you start with a 9×9 array of integers in which some of the cells have been filled with a digit between 1 and 9 as shown on the left side of Figure 8-8. Your job in the puzzle is to fill each of the empty spaces with a digit between 1 and 9 so that each digit appears exactly once in each row, each column, and each of the smaller 3×3 squares. The solution appears at the right side of Figure 8-8. Each Sudoku puzzle is carefully constructed so that there is only one solution.

    Although the algorithmic strategies you need to solve Sudoku puzzles lie beyond the scope of this book, you can easily write a function that checks to see whether a proposed solution follows the Sudoku rules against duplicating values in a row, column, or outlined 3×3 square. Write a function named `check_sudoku_solution` that takes a 9×9 array and returns `True` if that array obeys all the rules for a Sudoku square.

13. Write a method `flip_horizontal` that works similarly to `flip_vertical` as presented in the chapter except that it reverses the image in the horizontal dimension.

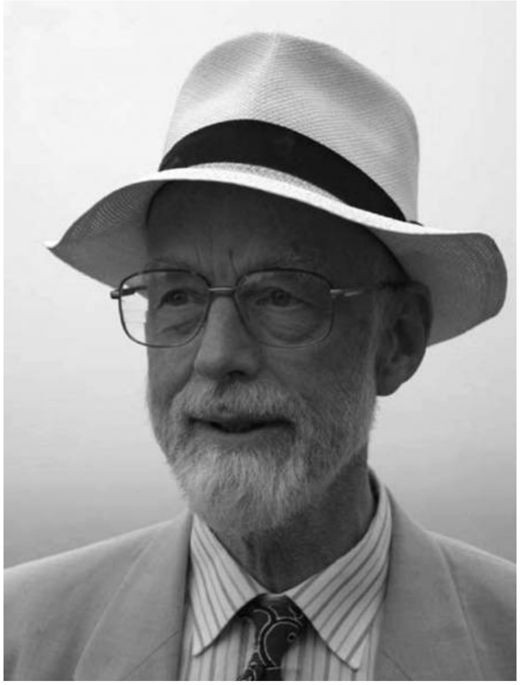
14. Write a method `rotate_left` that takes a `GImage` and produces a new `GImage` in which the original has been rotated 90 degrees to the left.

CHAPTER 9

Searching and Sorting

I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult.

—C. A. R. Hoare, Turing Award Lecture, 1981



C. A. R. Hoare

Charles Antony Richard (Tony) Hoare is Professor *emeritus* of Computer Science at Oxford University and a senior researcher at Microsoft's Research Laboratory in Cambridge, England. After completing a degree in philosophy at Oxford in 1956, Hoare became fascinated by the emerging world of computer science. During his graduate-school years, Hoare developed a highly efficient sorting algorithm called Quicksort, which is described in this chapter and remains in active use today. He also led the effort during the 1960s to create the first commercial compiler for Algol 60, a programming language that served as an important model for subsequent languages. Professor Hoare received the ACM Turing Award in 1980.

As you have certainly discovered by this point in your study of programming, there are usually many different strategies that you can use to solve a problem. Choosing among those different strategies is a central part of the programming process and typically requires you to evaluate several possible approaches to determine which is most appropriate to the problem at hand. One important consideration is efficiency. Algorithms that run more efficiently will presumably require less computing time and deliver the desired results more quickly.

This chapter explores the topic of algorithmic efficiency in the context of two fundamental operations in which the choice of algorithm has a significant impact on the program's running time. Those operations are

- ***Searching,*** which is the process of finding a particular element in an array
- ***Sorting,*** which is the process of rearranging the elements in an array so that they are stored in a well-defined order

Historically, each of these operations is defined in the context of arrays. This chapter is therefore in some sense a continuation of the discussion of arrays and lists from Chapter 8. This chapter, however, has another central theme that links it not only to that chapter but also to the discussion of algorithmic strategies in Chapter 3.

As you will see as you go through the programs in this chapter, there are many different strategies you can use to implement searching and sorting. These strategies vary enormously in their efficiency and therefore raise more general issues about precisely what the term *efficiency* means in an algorithmic context and how one might go about measuring that efficiency. These questions form the foundation for the subfield of computer science known as ***analysis of algorithms.*** Although a detailed understanding of algorithmic analysis requires a reasonable facility with mathematics and a lot of careful thought, you can get a sense of how it works by investigating the performance of several different algorithms in an important programming domain.

9.1 Searching

Although the search problem is usually framed as one of finding the index at which a particular element occurs in a list, the properties of search algorithms can be illustrated just as well in the context of the simpler problem of determining whether an element exists in the list at all. This operation is precisely the one implemented by Python's `in` operator. The expression

```
key in list
```

has the value `True` if `key` exists at any position in the list and the value `False` if it doesn't.

To make this version of the search problem more concrete—and more useful as well—suppose that you are writing a program to play a word game like Scrabble and need to know whether a particular string is a valid English word. To do so, you will need to have access to a list of English words so that you can compare the word you have against the elements of that list. That list, of course, has conceptual similarities to a dictionary, but lacks the associated definitions. Computer scientists commonly refer to a word list without definitions as a ***lexicon.***

As you know from 3, the libraries associated with this book include a module called `english` that exports a list called `ENGLISH_WORDS` that contains 127,145 words along with a function `is_english_word(s)` that checks to see whether `s` is a valid English word. Although you could simply use the `in` operator to ask whether

```
s in ENGLISH_WORDS
```

determining the answer would take considerable time bccause Python would have to check through the entire list in `ENGLISH_WORDS`. The goal of the next two sections is to explore alternative strategies for implementing this test to find one that offers considerably better performance.

The linear-search algorithm

The simplest strategy for writing `is_english_word`—although not necessarily the most efficient one—is captured in the following advice that the King of Hearts gives the White Rabbit in Lewis Carroll's *Alice's Adventures in Wonderland:*

> *Begin at the beginning, and go on till you come to the end: then stop.*

Turning that informal statement into an algorithm for searching is not particularly difficult. The only modification that you need to make is that the algorithm should also stop if it finds the value it is searching for. Thus, you might express a more complete account of Lewis Carroll's searching algorithm as follows:

> *Begin at the beginning, and go on till you either find the element you're looking for or come to the end. If you find the element, you know that it exists; if you reach the end, you know that the element does not appear.*

Because the process starts at the beginning and proceeds in a straight line through the elements of the array, this algorithm is called ***linear search.***

Turning this informal strategy into a Python definition for is_english_word requires little more than a direct translation of Lewis Carroll's approach:

```
def is_english_word(s):
    s = s.lower()
    for i in range(len(ENGLISH_WORDS)):
        if s == ENGLISH_WORDS[i]:
            return True
    return False
```

The for loop begins at the beginning and continues until it comes to the end of the ENGLISH_WORDS array. The function returns when it finds the word in the array or, failing that, at the end of the for loop.

The binary-search algorithm

The version of is_english_word from the previous section runs slowly because the linear-search algorithm has to looks at each array element in turn to check for a match. Fortunately, the fact that the ENGLISH_WORDS array is stored in alphabetical order makes it possible to do much better. A useful strategy for improving the efficiency of is_english_word is to compare the word you're looking for against the entry in the middle of the array. If your word precedes in alphabetical order the value you find at the middle position, you only have to search the first half of the array. Conversely, if your word follows the value in the center position, you only have to search the second half. Repeating this process means that you can throw away half of the values in the array on each cycle of the search loop. This algorithm, which appears in Figure **Error! Reference source not found.**-1 on the next page, is called ***binary search.***

After converting the parameter s to lower case to ensure that it matches the words in the lexicon, the binary-search implementation of is_english_word begins by setting the variables lh and rh to the leftmost and rightmost index positions in the array. At the beginning, the leftmost index is 0, and the rightmost index is one less than the length of the ENGLISH_WORDS array. If the string contained in s is in the lexicon, it must lie somewhere in this range of indices. The rest of the function consists of a loop that successively narrows this range by comparing s against the entry in the middle of the index range and using the result of that comparison to decide how to adjust the index bounds. The loop continues until the word is found or until there are no elements left in the range, which means that the word is not in the lexicon.

FIGURE 9-1 **Implementation of `is_english_word` using binary search**

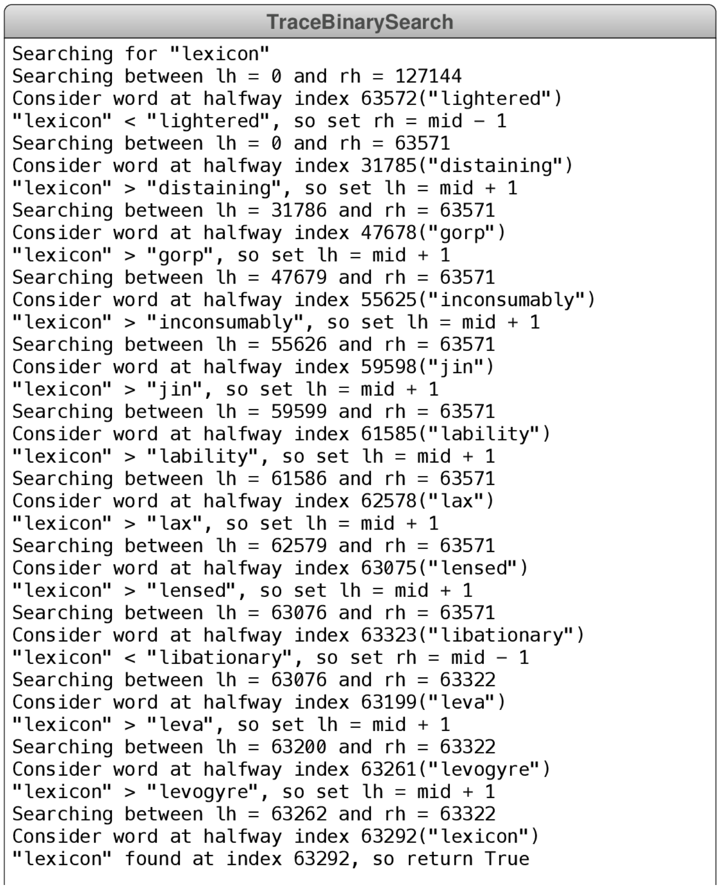
```
# File: IsEnglishWord.py

"""This module implements is_english_word using binary search."""

from english import ENGLISH_WORDS

# Implementation notes: is_english_word
# -------------------------------------
# This code looks at the middle element of the sorted range and can
# therefore discard half the elements on each cycle.

def is_english_word(s):
    """Returns True if s is a valid English word."""
    s = s.lower()
    lh = 0
    rh = len(ENGLISH_WORDS) - 1
    while lh <= rh:
        mid = (lh + rh) // 2
        if s == ENGLISH_WORDS[mid]:
            return True
        if s < ENGLISH_WORDS[mid]:
            rh = mid - 1
        else:
            lh = mid + 1
    return False
```

The binary-search implementation of `is_english_word` is important enough that it makes sense to go through an example. Suppose that you want to check whether *lexicon* is really an English word or simply part of a technical vocabulary that computer scientists use. To do so, you can execute the following code:

```
if is_english_word("lexicon"):
    print("lexicon is a valid word")
```

The `ENGLISH_WORDS` array contains 127145 words, which means that the initial values of `lh` and `rh` are 0 and 127144. On the first cycle of the loop, the code computes the midpoint of the remaining range by averaging `lh` and `rh`, using the `//` operator to ensure that the result is an integer. The code then stores this position in the variable `mid`. The word at index 63572 is the unusual but nonetheless legitimate word *lightered.* Since *lexicon* comes before *lightered* in lexicographic order, `is_english_word` needs to search only the indices between `lh` and `mid - 1`. Substituting the current values of these variables shows that the new search range is limited to the indices between 0 and 63571. The process then continues until it either finds the specified word or there are no elements left in the index range.

Calling `is_english_word("lexicon")` makes the sequence of comparisons shown in the following console log:



```
Searching for "lexicon"
Searching between lh = 0 and rh = 127144
Consider word at halfway index 63572("lightered")
"lexicon" < "lightered", so set rh = mid - 1
Searching between lh = 0 and rh = 63571
Consider word at halfway index 31785("distaining")
"lexicon" > "distaining", so set lh = mid + 1
Searching between lh = 31786 and rh = 63571
Consider word at halfway index 47678("gorp")
"lexicon" > "gorp", so set lh = mid + 1
Searching between lh = 47679 and rh = 63571
Consider word at halfway index 55625("inconsumably")
"lexicon" > "inconsumably", so set lh = mid + 1
Searching between lh = 55626 and rh = 63571
Consider word at halfway index 59598("jin")
"lexicon" > "jin", so set lh = mid + 1
Searching between lh = 59599 and rh = 63571
Consider word at halfway index 61585("lability")
"lexicon" > "lability", so set lh = mid + 1
Searching between lh = 61586 and rh = 63571
Consider word at halfway index 62578("lax")
"lexicon" > "lax", so set lh = mid + 1
Searching between lh = 62579 and rh = 63571
Consider word at halfway index 63075("lensed")
"lexicon" > "lensed", so set lh = mid + 1
Searching between lh = 63076 and rh = 63571
Consider word at halfway index 63323("libationary")
"lexicon" < "libationary", so set rh = mid - 1
Searching between lh = 63076 and rh = 63322
Consider word at halfway index 63199("leva")
"lexicon" > "leva", so set lh = mid + 1
Searching between lh = 63200 and rh = 63322
Consider word at halfway index 63261("levogyre")
"lexicon" > "levogyre", so set lh = mid + 1
Searching between lh = 63262 and rh = 63322
Consider word at halfway index 63292("lexicon")
"lexicon" found at index 63292, so return True
```

As you can see from the trace output, the binary-search algorithm is able to find the word *lexicon* by making just 12 comparisons, even though the lexicon contains 127,145 words. The linear-search algorithm has to look at every one of those words, which means that the binary-search approach reduces the number of required comparisons by a factor of 10,000.

9.2 Simple strategies for sorting

Although the differences in efficiency between linear search and binary search are certainly significant, the importance of choosing the right algorithm is even more evident in the problem of ***sorting,*** which consists of rearranging the elements in a list so that they appear in some well-defined order. For example, suppose you have stored the following integers in the variable `array`, which is implemented as a list in Python:

array

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

Your mission is to write a function `sort(array)` that rearranges the elements into ascending order, like this:

19	25	30	37	56	58	73	95
0	1	2	3	4	5	6	7

The selection sort algorithm

There are many algorithms you could choose to sort an array into ascending order. One of the simplest is called ***selection sort,*** which is implemented in Figure 9-2. Given an array of size *N*, the selection sort algorithm goes through each element position and finds the value that should occupy that position in the sorted array. When it finds the appropriate element, the algorithm exchanges that element with the value previously occupying the desired position to ensure that no elements are lost. Thus, on the first cycle, the algorithm finds the smallest element and swaps it with the first element, which appears at index position 0. On the second cycle, it finds the smallest remaining element and swaps it with the second element. Thereafter, the algorithm continues this strategy until all positions in the array are correctly ordered.

FIGURE 9-2 **Implementation of the selection sort algorithm**

```python
# File: SelectionSort.py

"""This module implements the sort function using selection sort."""

# Implementation notes: sort
# --------------------------
# This function implements the selection sort algorithm, which swaps
# each element in turn with the element that belongs in that position.

def sort(array):
    """Sorts the array in ascending order."""
    for lh in range(len(array)):
        rh = lh
        for i in range(lh + 1, len(array)):
            if array[i] < array[rh]:
                rh = i
        array[lh],array[rh] = array[rh],array[lh]
```

For example, if the initial contents of the array are

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

the first cycle through the outer `for` loop identifies the 19 in index position 5 as the smallest value in the entire array and then swaps it with the 56 in index position 0 to leave the following configuration:

19	25	37	58	95	56	73	30
0	1	2	3	4	5	6	7

On the second cycle, the algorithm finds the smallest element between positions 1 and 7, which turns out to be the 25 in position 1. The program goes ahead and performs the exchange operation, leaving the array unchanged from that in the preceding diagram. On each subsequent cycle, the algorithm performs a swap operation to move the next smallest value into its appropriate final position. When the `for` loop is complete, the entire array is sorted.

Empirical measurement of performance

How efficient is the selection sort algorithm as a strategy for sorting? To answer questions of this kind, it helps to collect empirical data about how long it takes a computer to complete a task for problems of varying size. When I ran the selection sort algorithm on my MacBook Pro laptop, for example, I observed the following running times, where *N* represents the number of elements in the array:

N	**Running time**
10	0.000013 sec
100	0.000581 sec
1000	0.0579 sec
10,000	5.738 sec
100,000	574.2 sec
1,000,000	57,395.0 sec

For an array of 100 integers, the selection sort algorithm completes its work in less than a millisecond. Even for 10,000 integers, this implementation of `sort` takes just a few seconds, which seems fast enough in terms of our human sense of time. As the array sizes get larger, however, the performance of selection sort begins to go downhill. For an array of 100,000 integers, the algorithm requires 574 seconds, which is almost 10 minutes. If you're sitting in front of your computer waiting for it to reply, that seems a long time. But that number pales into insignificance when you compare it to the time required to sort 1,000,000 integers, which—at least when coded in Python using the selection sort algorithm—takes almost 16 *hours.*

The performance of selection sort rapidly gets worse as the array size increases. As you can see from the timing data, every time you multiply the number of values by 10, the time required to sort the array goes up a hundredfold. Sorting a list of ten million numbers would therefore take somewhere around 1600 hours, which is on the order of 66 *days*. If your business required sorting arrays on this scale, you would have no choice but to find a more efficient approach.

Analyzing the performance of selection sort

What makes selection sort perform so badly as the number of values to be sorted becomes large? To answer this question, it helps to think about what the algorithm has to do on each cycle of the outer loop. To correctly determine the first value in the array, the selection sort algorithm must consider all N elements as it searches for the smallest value. Thus, the time required on the first cycle of the loop is presumably proportional to N. For each of the other elements in the array, the algorithm performs the same basic steps but looks at a smaller number of elements each time. It looks at $N-1$ elements on the second cycle, $N-2$ on the third, and so on, so the total running time is roughly proportional to

$$N + N-1 + N-2 + \ . \ . \ . + 3 + 2 + 1$$

Because it is difficult to work with an expression in this expanded form, it is useful to simplify it by applying a bit of mathematics. As you may have learned in an algebra course, the sum of the first N integers is given by the formula

$$\frac{N \times (N+1)}{2}$$

or, after evaluating the multiplication sign in the numerator,

$$\frac{N^2 + N}{2}$$

If you write out the values of this function for various values of N, you get a table that looks like this:

N	$\frac{N^2 + N}{2}$
10	55
100	5050
1000	500,500
10,000	50,005,000
100,000	5,000,050,000
1,000,000	500,000,500,000

Because the running time of the selection sort algorithm is presumably related to the amount of work the algorithm needs to do, the values in this table should be roughly proportional to the observed execution time of the algorithm, which turns out to be true. If you look at the measured timing data for selection sort in Figure 9-3, for example, you discover that the algorithm requires 574.2 seconds to sort 100,000 numbers. In that time, the selection sort algorithm has to perform 50,005,000 operations in its innermost loop. Assuming that there is indeed a proportionality relationship between these two values, dividing the time by the number of operations gives the following estimate of the proportionality constant:

$$\frac{574.2\text{ seconds}}{5{,}000{,}050{,}000} \approx 1.15 \times 10^{-7}\text{ seconds}$$

If you apply this same proportionality constant to the other entries in the table, you discover that the formula

$$1.22 \times 10^{-7}\text{ seconds} \times \frac{N^2 + N}{2}$$

offers a reasonable approximation of the running time for all but the smallest values of *N,* where the time required for other statements in the program have more significance. The table in Figure 9-3 includes these estimated times and the relative error between the observed and estimated times.

9.3 Computational complexity

The problem with carrying out a detailed analysis like the one shown in Figure 9-3 is that you end up with too much information. Although it is occasionally useful to have a formula for predicting exactly how long a program will take, you can usually get away with more qualitative measures. The reason that selection sort is impractical for large values of *N* has little to do with the precise timing characteristics of a particular implementation running on the laptop I happen to have at the moment. The problem is simpler and more fundamental. At its essence, the problem with selection sort is that doubling the size of the input array increases the running time of the selection sort algorithm by a factor of four, which means that the running time grows more quickly than the number of elements in the array.

FIGURE 9-3 Observed and estimated times for selection sort

N	Observed time	Estimated time	Error
10	0.000013 sec	0.0000063 sec	51%
100	0.000581 sec	0.000580 sec	< 1%
1000	0.0579 sec	0.0576 sec	< 1%
10,000	5.738 sec	5.751 sec	< 1%
100,000	574.2 sec	575.0 sec	< 1%
1,000,000	57,395.0 sec	57,500.0 sec	< 1%

The most valuable qualitative insights you can obtain about algorithmic efficiency are usually those that help you understand how the performance of an algorithm responds to changes in problem size. For algorithms that operate on numbers, it generally makes sense to let the numbers themselves represent the problem size. For algorithms that operate on arrays, you can use the number of elements. When evaluating algorithmic efficiency, computer scientists traditionally use the letter N to indicate the size of the problem, no matter how it is calculated. The relationship between N and the performance of an algorithm as N becomes large is called the ***computational complexity*** of that algorithm.

Big-O notation

Computer scientists use a special formulation called ***big-O notation*** to denote the computational complexity of algorithms. Big-O notation was introduced by the German mathematician Paul Bachmann in 1892—long before the development of computers. The notation itself is very simple and consists of the letter *O,* followed by a formula enclosed in parentheses. When it is used to specify computational complexity, the formula is usually a simple function involving the problem size N. For example, in this chapter you will soon encounter the big-O expression

$$O(N^2)$$

which reads aloud as "big-oh of N squared."

Big-O notation is used to specify qualitative approximations and is ideal for expressing the computational complexity of an algorithm. Coming as it does from mathematics, big-O notation has a precise definition, which appears later in this chapter in the section entitled "A formal definition of big-O." At this point, however, it is more important for you to gain some intuition into what big-O means.

Standard simplifications of big-O

When you use big-O notation to express computational complexity, the goal is to offer a *qualitative* insight as to how changes in N affect the algorithmic performance as N becomes large. Because big-O notation is not intended to be a quantitative measure, it is not only appropriate but desirable to make the formula inside the parentheses as simple as possible. The most common simplifications you can make when using big-O notation to express computational complexity are as follows:

1. *Eliminate any term whose contribution to the total ceases to be significant as N becomes large.* When a formula involves several terms added together, one of the terms often grows much faster than the others and ends up dominating the entire expression as N becomes large. For large values of N, this term alone will control the running time of the algorithm, and you can ignore the other terms in the formula entirely.

2. *Eliminate any constant factors.* When you calculate computational complexity, your main concern is how running time changes as a function of the problem size *N*. Constant factors have no effect on the overall pattern. If you bought a machine that was twice as fast as your old one, any algorithm that you executed on your machine would run twice as fast as before for every value of *N*. The growth pattern, however, would remain exactly the same. Thus, you can ignore constant factors when you use big-O notation.

The computational complexity of selection sort

You can apply the simplification rules from the preceding section to derive a big-O expression for the computational complexity of selection sort. From the analysis in the section "Analyzing the performance of selection sort" earlier in the chapter, you know that the running time of the selection sort algorithm for an array of *N* elements is proportional to

$$\frac{N^2 + N}{2}$$

Although it would be mathematically correct to use this formula directly in the big-O expression

$$O\left(\frac{N^2 + N}{2}\right)$$

you would never do so in practice because the formula inside the parentheses is not expressed in the simplest form.

The first step toward simplifying this relationship is to recognize that the formula is actually the sum of two terms, as follows:

$$\frac{N^2}{2} + \frac{N}{2}$$

You then need to consider the contribution of each of these terms to the total formula as *N* increases in size, which is illustrated by the following table:

N	$\frac{N^2}{2}$	$\frac{N}{2}$	$\frac{N^2+N}{2}$
10	50	5	55
100	5000	50	5050
1000	500,000	500	500,500
10,000	50,000,000	5000	50,005,000
100,000	5,000,000,000	50,000	5,000,050,000

As N increases, the term involving N^2 quickly dominates the term involving N. As a result, the simplification rule allows you to eliminate the smaller term from the expression. Even so, you would not write that the computational complexity of selection sort is

$$O\left(\frac{N^2}{2}\right)$$

because you can eliminate the constant factor. The simplest expression you can use to indicate the complexity of selection sort is

$$O(N^2)$$

This expression captures the essence of the performance of selection sort. As the size of the problem increases, the running time tends to grow by the square of that increase. Thus, if you double the size of the array, the running time goes up by a factor of four. If you instead multiply the number of input values by 10, the running time explodes by a factor of 100.

Deducing computational complexity from code

It is often possible to determine the computational complexity of a function simply by looking at the code, as in the following function that computes the average of the elements in an array:

```
def average(array):
    total = 0
    for value in array:
        total += value
    return total / n
```

When you call this function, some parts of the code are executed only once, such as the initialization of `total` to 0 and the division operation in the `return` statement. These computations take a certain amount of time, but that time is constant in the sense that it doesn't depend on the size of the array. Code whose execution time does not depend on the problem size is said to run in ***constant time,*** which is expressed in big-O notation as $O(1)$.

The designation $O(1)$ can seem confusing, because the expression inside the parentheses does not depend on N. In fact, this lack of any dependency on N is the whole point of the $O(1)$ notation. As you increase the size of the problem, the time required to execute code whose running time is $O(1)$ increases in exactly the same way that 1 increases; in other words, the running time does not increase at all.

There are, however, other parts of the `average` function that are executed exactly N times, once for each cycle of the `for` loop. These components include the internal calculations in the `for` loop to produce the next element and the statement

```
total += value
```

that constitutes the loop body. Although any single execution of this part of the computation takes a fixed amount of time, the fact that these statements are executed N times means that their total execution time is directly proportional to the array size. The computational complexity of this part of the `average` function is $O(N)$, which is commonly called ***linear time.***

The total running time for `average` is therefore the sum of the times required for the constant parts and the linear parts of the algorithm. As the size of the problem increases, however, the constant term becomes less and less relevant. By exploiting the simplification rule that allows you to ignore terms that become insignificant as N gets large, you can assert that the `average` function runs in $O(N)$ time.

You could also predict this result just by looking at the loop structure of the code. For the most part, the individual expressions and statements—unless they involve function calls that must be accounted separately—run in constant time. What matters in terms of computational complexity is how often those statements are executed. For many programs, you can determine the computational complexity simply by finding the piece of the code that is executed most often and determining how many times it runs as a function of N. In the case of the `average` function, the body of the loop is executed `n` times. Because no part of the code is executed more often than this, you can predict that the computational complexity will be $O(N)$.

The selection sort function can be analyzed in a similar way. The most frequently executed part of the code is the comparison in the statement

```
if array[i] < array[rh]:
```

That statement is nested inside two `for` loops whose limits depend on the value of N. The inner loop runs N times as often as the outer loop, which implies that the inner loop body is executed $O(N^2)$ times. Algorithms like selection sort that exhibit $O(N^2)$ performance are said to run in ***quadratic time.***

Worst-case versus average-case complexity

In some cases, the running time of an algorithm depends not only on the size of the problem but also on the specific characteristics of the data. For example, consider the function

```
def linear_search(key, array):
    for i in range(len(array)):
        if key == array[i]:
            return i
    return -1
```

which uses the linear-search algorithm to return the first index position in `array` at which the value `key` appears or the sentinel value −1 if the value `key` does not appear anywhere in the array. Because the `for` loop in the implementation executes N times, you expect the performance of `linear_search`, as its name implies, to be $O(N)$.

On the other hand, some calls to `linear_search` can be executed very quickly. Suppose, for example, that the key element you are searching for happens to be in the first position in the array. In that case, the body of the `for` loop will run only once. If you're lucky enough to search for a value that always occurs at the beginning of the array, `linear_search` will run in constant time.

When you analyze the computational complexity of a program, you are rarely interested in the minimum possible time. In general, computer scientists tend to be concerned about analyzing the following two types of complexity:

- *Worst-case complexity.* The most common type of complexity analysis consists of determining the performance of an algorithm in the worst possible case. Such an analysis is useful because it allows you to set an upper bound on the computational complexity. If you analyze for the worst case, you can guarantee that the performance of the algorithm will be at least as good as your analysis indicates. You might sometimes get lucky, but you can be confident that the performance will not get any worse.
- *Average-case complexity.* From a practical point of view, it is often useful to consider how well an algorithm performs if you average its behavior over all possible sets of input data. Particularly if you have no reason to assume that the specific input to your problem is in any way atypical, the average-case analysis provides the best statistical estimate of actual performance. Unfortunately, average-case analysis is usually more difficult to carry out and typically requires considerable mathematical sophistication.

The worst case for the `linear_search` function occurs when the key is not in the array at all. When the key is not there, the function must complete all `n` cycles of the `for` loop, which means that its performance is $O(N)$. If the key is known to be in the array, the `for` loop will be executed about half as many times on average, which implies that average-case performance is also $O(N)$. As you will discover in the section on "The Quicksort algorithm" later in this chapter, the average-case and worst-case performances of an algorithm sometimes differ in qualitative ways, which means that in practice it is often important to take both performance characteristics into consideration.

9.4 Divide-and-conquer strategies

At this point, you know considerably more about complexity analysis than you did when you started the chapter. However, you are no closer to solving the practical problem of how to write a sorting algorithm that is more efficient for large arrays. The selection sort algorithm is clearly not up to the task, because the running time increases in proportion to the square of the input size. The same is true for most sorting algorithms that process the elements of an array in a linear order. To develop a better sorting algorithm, you need a qualitatively different approach.

Oddly enough, the key to finding a better sorting strategy lies in recognizing that the quadratic behavior of algorithms like selection sort has a hidden virtue. The basic characteristic of quadratic complexity is that, as the size of a problem doubles, the running time increases by a factor of four. The reverse, however, is also true. If you divide the size of a quadratic problem by two, you decrease the running time by that same factor of four. This fact suggests that dividing an array in half and then applying a recursive divide-and-conquer approach might reduce the required sorting time.

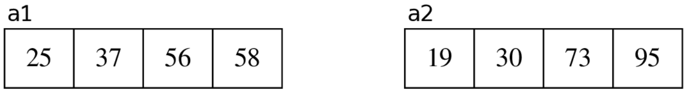
To make this idea more concrete, suppose you have a large array that you need to sort. What happens if you divide the array into two halves and then use the selection sort algorithm to sort each of those pieces? Because selection sort is quadratic, each of the smaller arrays requires one quarter of the original time. You need to sort both halves, of course, but the total time required to sort the two smaller arrays is still only half the time that would have been required to sort the original array. If it turns out that sorting two halves of an array simplifies the problem of sorting the complete array, you will be able to reduce the total time substantially. More importantly, once you discover how to improve performance at one level, you can use the same algorithm recursively to sort each half.

To determine whether a divide-and-conquer strategy is applicable to the sorting problem, you need to answer the question of whether dividing an array into two smaller arrays and then sorting each one helps to solve the general problem. As a way to gain some insight into this question, suppose that you start with an array containing the following eight elements:

`array`

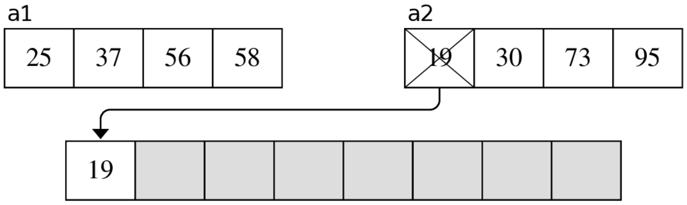
56	25	37	58	95	19	73	30

If you divide the array of eight elements into two arrays of length four and then sort each of those smaller arrays—keep in mind that the recursive leap of faith means you can assume that the recursive calls work correctly—you get the following situation in which each of the smaller arrays is sorted:

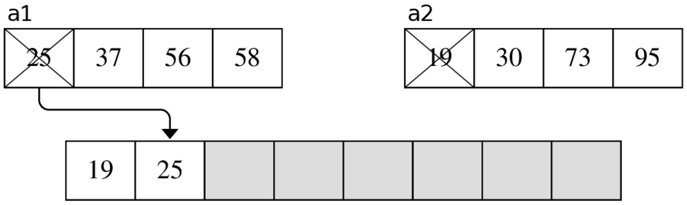


How useful is this decomposition? Remember that your goal is to take the values out of these smaller arrays and put them back into the original array in the correct order. How does having these smaller sorted arrays help you accomplish that goal?

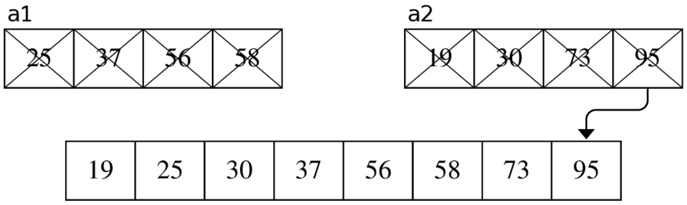
As it happens, reconstructing the complete array from the smaller sorted arrays is a simpler problem than sorting itself. The required technique, called ***merging,*** depends on the fact that the first element in the complete ordering must be either the first element in `a1` or the first element in `a2`, whichever is smaller. In this example, the first element you want is the 19 in `a2`. If you add that element to an `array` of the right length and cross it out of `a2`, you get the following configuration:



Once again, the next element can only be the first unused element in `a1` or `a2`. This time, you compare the 25 from `a1` against the 30 in `a2` and choose the former:



You can easily continue this process of choosing the smaller value from `a1` or `a2` until you have reconstructed the entire array, which will look like this:



The merge sort algorithm

The merge operation, combined with recursive decomposition, gives rise to a sorting algorithm called ***merge sort,*** which requires the following steps:

1. Check to see if the length of the array is 0 or 1. If so, it must already be sorted.
2. Divide the array into two smaller arrays, each of which is half the size.
3. Sort each of the smaller arrays recursively.
4. Merge the two sorted arrays back into the original one.

The code for the merge sort algorithm, shown in Figure 9-4, divides neatly into two functions: `sort` and `merge`. The code for `sort` follows directly from the outline

FIGURE 9-4 **Implementation of the merge sort algorithm**

```python
# File: MergeSort.py

"""This module implements the sort function using merge sort."""

# Implementation notes: sort
# --------------------------
# This implementation divides the array in half, recursively sorts each
# of the resultinng subarrays, and then merges the sorted subarrays by
# choosing the smallest remaining element for the two possibilities.

def sort(array):
    """Rearranges the elements of array in ascending order."""
    if len(array) > 1:
        mid = len(array) // 2
        a1 = array[:mid]
        a2 = array[mid:]
        sort(a1)
        sort(a2)
        merge(array, a1, a2)

def merge(array, a1, a2):
    """Merges the sorted arrays a1 and a2 into array."""
    n1 = len(a1)
    n2 = len(a2)
    p1 = 0
    p2 = 0
    for i in range(len(array)):
        if p2 == n2 or (p1 < n1 and a1[p1] < a2[p2]):
            array[i] = a1[p1]
            p1 += 1
        else:
            array[i] = a2[p2]
            p2 += 1
```

of the algorithm. After checking for the special case, the algorithm uses slicing to divide the original array into two smaller ones, sorts these arrays recursively, and then calls `merge` to reassemble the complete solution.

Most of the work is done by the `merge` function, which takes the original `array` along with the smaller arrays `a1` and `a2`. The heart of the `merge` function is the `for` loop that fills each position in `array`. On each cycle of the loop, the function selects the smaller element from `a1` or `a2` (after first checking whether any elements are left) and copies that value to the next free slot in the original array.

The computational complexity of merge sort

You now have an implementation of the `sort` function based on the strategy of divide-and-conquer. How efficient is it? You can measure its efficiency by sorting arrays of numbers and timing the result, but it is helpful to start by thinking about the algorithm in terms of its computational complexity.

When you call the merge sort implementation of `sort` on a list of N numbers, the running time can be divided into two components:

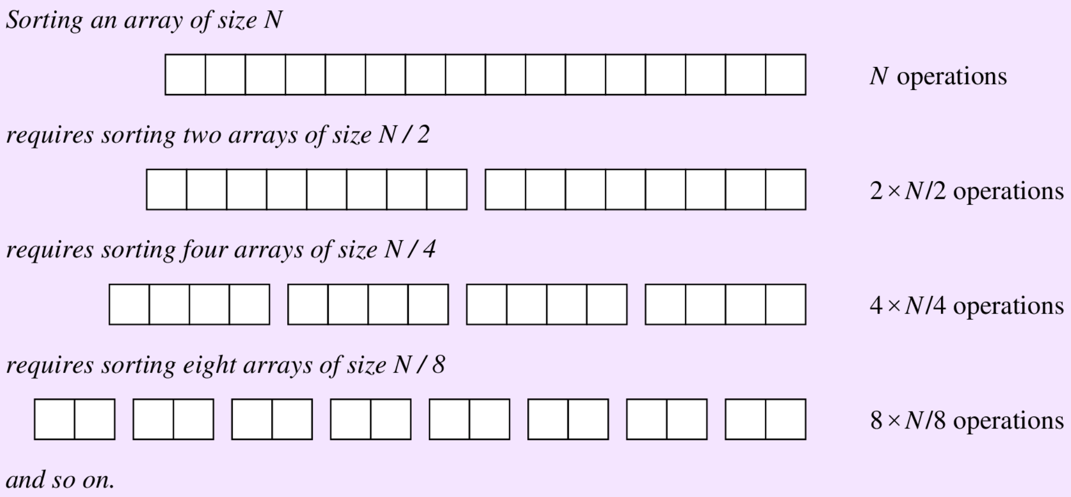
1. The amount of time required to execute the operations at the current level of the recursive decomposition
2. The time required to execute the recursive calls

At the top level of the recursive decomposition, the cost of performing the nonrecursive operations is proportional to N. The loop to fill the subsidiary arrays accounts for N cycles, and the call to `merge` has the effect of refilling the original N positions in the array. If you add these operations and ignore the constant factor, you discover that the complexity of any single call to `sort`—not counting the recursive calls within it—requires $O(N)$ operations.

But what is the cost of the recursive operations? To sort an array of size N, you must recursively sort two arrays of size $N / 2$. Each of these operations requires some amount of time. If you apply the same logic, you quickly determine that sorting each of these smaller arrays requires time proportional to $N / 2$ at that level of the recursive decomposition, plus whatever time is required by any further recursive calls. The same process then continues until you reach the simple case in which the arrays consist of a single element or no elements at all.

The total time required to solve the problem is the sum of the time required at each level of the recursive decomposition. In general, the decomposition has the structure shown in Figure 9-5. As you move down through the recursive hierarchy, the arrays get smaller, but more numerous. The amount of work done at each level, however,

FIGURE 9-5 **Recursive decomposition of merge sort**


Sorting an array of size N
N operations
requires sorting two arrays of size N / 2
$2 \times N/2$ operations
requires sorting four arrays of size N / 4
$4 \times N/4$ operations
requires sorting eight arrays of size N / 8
$8 \times N/8$ operations
and so on.

is always directly proportional to N. Determining the total amount of work is thus a question of finding out how many levels there will be.

At each level of the hierarchy, the value of N is divided by 2. The total number of levels is therefore equal to the number of times you can divide N by 2 before you get down to 1. Rephrasing this problem in mathematical terms, you need to find a value of k such that

$$N = 2^k$$

Solving the equation for k gives you

$$k = \log_2 N$$

Because the number of levels is $\log_2 N$ and the amount of work done at each level is proportional to N, the total amount of work is proportional to $N \log_2 N$.

Unlike other scientific disciplines, in which logarithms are expressed in terms of powers of 10 (common logarithms) or the mathematical constant e (natural logarithms), computer science tends to use ***binary logarithms,*** which are based on powers of 2. Logarithms computed using different bases differ only by a constant factor, and it is therefore traditional to omit the logarithmic base when you talk about computational complexity. Thus, the computational complexity of merge sort is usually written as

$$O(N \log N)$$

FIGURE 9-6 Empirical comparison of selection and merge sorts

N	Selection sort	Merge sort
10	0.000013 sec	0.000025 sec
100	0.000581 sec	0.000378 sec
1000	0.0579 sec	0.00493 sec
10,000	5.738 sec	0.0605 sec
100,000	574.2 sec	0.763 sec
1,000,000	57,395.0 sec	9.05 sec

Comparing N^2 and N log N performance

But how much better is an algorithm that runs in $O(N \log N)$ time than one that requires $O(N^2)$? One way to assess the level of improvement is to look at empirical data to get a sense of how the running times of the selection and merge sort algorithms compare. That timing information appears in Figure 9-6. For 10 items, this implementation of selection sort is twice as fast as merge sort. By the time you get up to 100,000 items, however merge sort is faster by nearly three orders of magnitude. For large arrays, merge sort represents a significant improvement. The numbers in both columns grow as N becomes larger, but the N^2 column grows much faster than the N log N column. Sorting algorithms based on an N log N algorithm will therefore be useful over a much larger range of array sizes.

9.5 Standard complexity classes

In programming, most algorithms fall into one of several common complexity classes. The most important complexity classes are shown in Figure 9-7, which gives the common name of the class along with the corresponding big-O expression and a representative algorithm in that class.

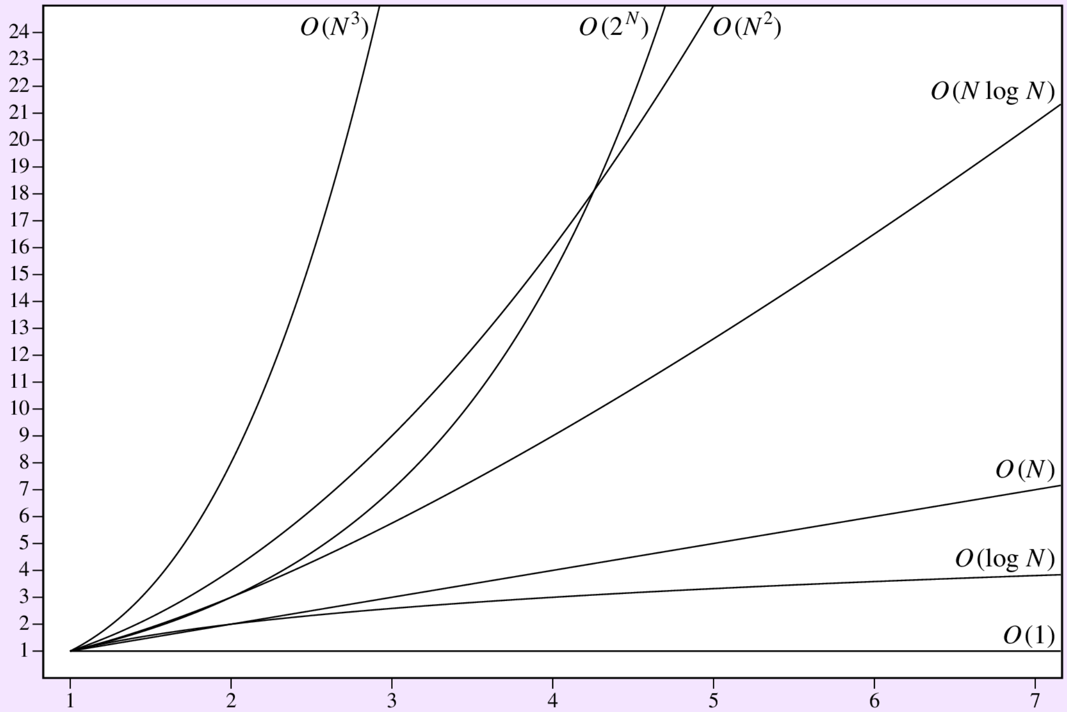
FIGURE 9-7 Standard complexity classes

Constant	$O(1)$	Returning the first element in an array
Logarithmic	$O(\log N)$	Binary search in a sorted array
Linear	$O(N)$	Linear search in an array
N log N	$O(N \log N)$	Merge sort
Quadratic	$O(N^2)$	Selection sort
Cubic	$O(N^3)$	Straightforward algorithms for matrix multiplication
Exponential	$O(2^N)$	Solving a problem requiring a series of N branching decisions

The classes in Figure 9-7 are presented in strictly increasing order of complexity. If you have a choice between one algorithm that requires $O(\log N)$ time and another that requires $O(N)$ time, the first will always outperform the second as N grows large. For small values of N, terms that are discounted in the big-O calculation may allow a theoretically less efficient algorithm to outperform one that has a lower computational complexity. On the other hand, as N grows larger, there will always be a point at which the theoretical difference in efficiency becomes the deciding factor.

The differences in efficiency between these classes are in fact profound. You can begin to get a sense of how the different complexity functions stand in relation to one another by looking at the graph in Figure 9-8, which plots these complexity functions on a traditional linear scale. Unfortunately, this graph tells an incomplete and somewhat misleading part of the story, because the values of N are all very small. Complexity analysis, after all, is primarily relevant as the values of N become large. Figure 9-9 shows the same data plotted on a logarithmic scale, which gives you a better sense of how these functions grow over a more extensive range of values.

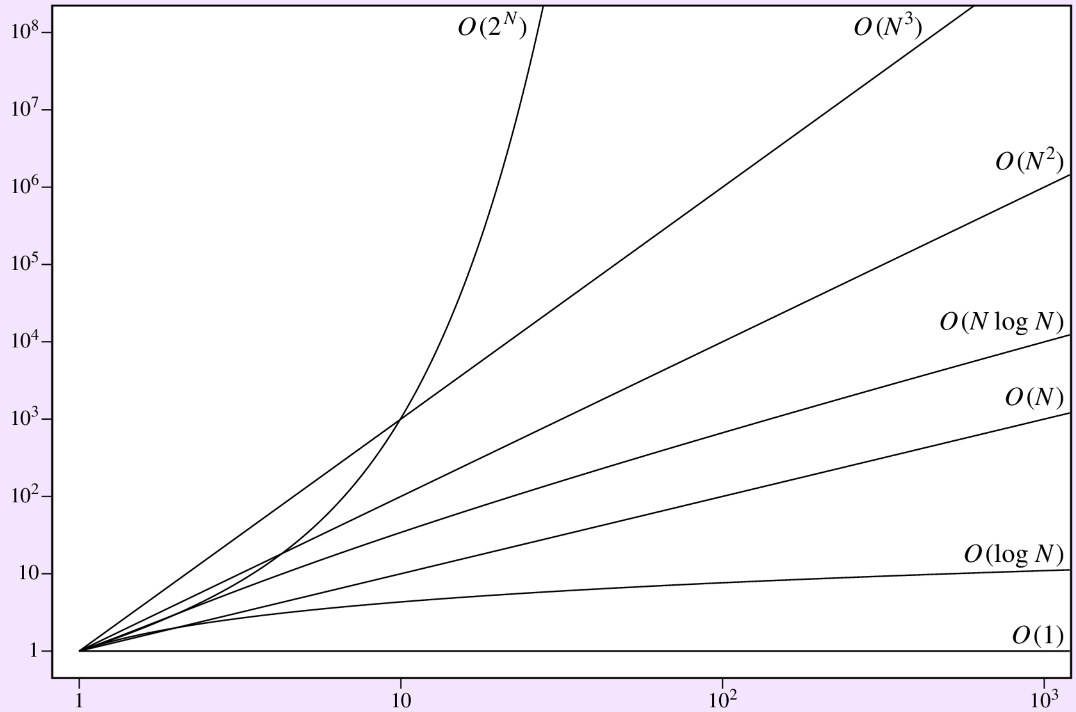
FIGURE 9-8 **Growth characteristics of the standard complexity classes: linear plot**



Algorithms that fall into the constant, linear, quadratic, and cubic complexity classes are all part of a more general family called ***polynomial algorithms,*** which execute in time N^k for some constant k. One of the useful properties of the logarithmic plot shown in Figure 9-9 is that the graph of any function N^k always comes out as a straight line whose slope is proportional to k. If you look at the figure, it is clear that N^k—no matter how big k happens to be—invariably grows more slowly than the exponential function represented by 2^N, which continues to curve upward as the value of N increases. This property has important implications in terms of finding practical algorithms for real-world problems. Even though the selection sort example demonstrates that quadratic algorithms have substantial performance problems for large values of N, algorithms whose complexity is $O(2^N)$ are considerably less efficient. As a general rule of thumb, computer scientists classify problems that can be solved using algorithms that run in polynomial time as ***tractable,*** in the sense that they are amenable to implementation on a computer. Problems for which no polynomial-time algorithm exists are regarded as ***intractable.***

Unfortunately, there are many commercially important problems for which all known algorithms require exponential time. One of these is the ***subset-sum problem,*** which consists of determining whether any subset of a set of N integers adds up to a

FIGURE 9-9 **Growth characteristics of the standard complexity classes: logarithmic plot**



given target value. Another is the ***traveling salesperson problem,*** which consists of finding the shortest route by which one can visit a set of *N* cities connected by some transportation system and then return to the starting point. As far as anyone knows, it is not possible to solve either the subset-sum problem or the traveling salesman problem in polynomial time. The best-known approaches all have exponential performance in the worst case and are equivalent in efficiency to generating all possible routings and comparing the cost. At least for the moment, the optimal solution to each of these problems is to try every possibility, which requires exponential time. On the other hand, no one has been able to prove conclusively that no polynomial-time algorithm for these problems exist. There might be some clever algorithm that would make these problems tractable. If so, many problems currently believed to be difficult would move into the tractable range as well.

The question of whether problems like subset-sum or the traveling salesman problem can be solved in polynomial time is one of the most important open questions in computer science and indeed in mathematics. This question is known as the ***P* = *NP* problem** and carries a million-dollar prize for its solution.

9.6 The Quicksort algorithm

Even though the merge sort algorithm presented earlier in this chapter performs well in theory and has a worst-case complexity of $O(N \log N)$, it is not used much in practice. Instead, most sorting programs in use today are based on an algorithm called Quicksort, developed by the British computer scientist C. A. R. (Tony) Hoare profiled on the first page of this chapter.

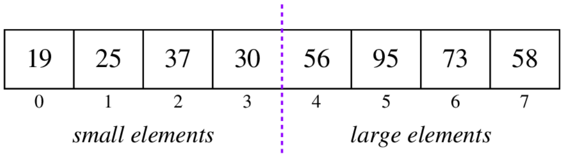
Both Quicksort and merge sort employ a divide-and-conquer strategy. In the merge sort algorithm, the original array is divided into two halves, each of which is sorted independently. The resulting sorted arrays are then merged together to complete the sort operation for the entire array. Suppose, however, that you took a different approach to dividing up the array. What would happen if you started the process by making an initial pass through the array, changing the positions of the elements so that "small" values come at the beginning of the array and "large" values come at the end, for some definition of the words *large* and *small?*

For example, suppose that the original array you wanted to sort was the following one, presented earlier in the discussion of merge sort:

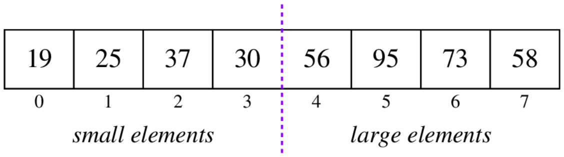
`array`

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

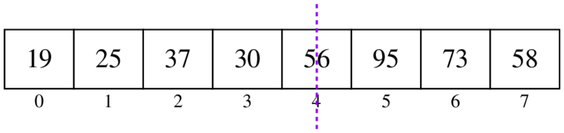
Since half of these elements are larger than 50 and half are smaller, it might make sense to define *small* in this case as being less than 50 and *large* as being 50 or more. If you could then find a way to rearrange the elements so that all the small elements appear at the beginning and all the large ones at the end, you would wind up with an array that looks something like the following diagram, which shows one of many possible orderings in which the small and large elements appear on opposite sides of the boundary:



When the elements are divided into parts in this fashion, all that remains to be done is to sort each of the parts, using a recursive call to the function that does the sorting. Since all the elements on the left side of the boundary line are smaller than all those on the right, the final result will be a completely sorted array:



If you could always choose the optimal boundary between the small and large elements on each cycle, this algorithm would divide the array in half each time and end up demonstrating the same qualitative characteristics as merge sort. In practice, the Quicksort algorithm selects some existing element in the array and uses that value to represent the dividing line between the small and large elements. Although you will have a chance to explore more effective strategies in the exercises, one strategy is to pick the first element (56 in the original array) and use that to represent the boundary value. When the array is reordered, the boundary will fall at a particular index position rather than between two positions, as follows:



From this point, the recursive calls must sort the array between positions 0 and 3 and the array between positions 5 and 7, leaving index position 4 right where it is.

As in merge sort, the simple case of the Quicksort algorithm is an array of size 0 or 1, which must already be sorted. The recursive part of the Quicksort algorithm consists of the following steps:

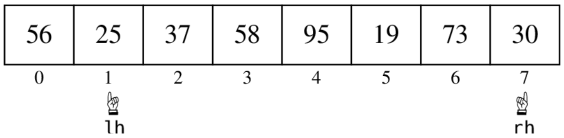
1. *Choose an element to serve as the boundary between the small and large elements.* This element is called the ***pivot.*** For the moment, the simplest strategy is to select the first element in the array.
2. *Rearrange the elements in the array so that large elements are moved toward the end of the array and small elements toward the beginning.* More formally, the goal of this step is to divide the elements around a boundary position so that all elements to the left of the boundary are less than the pivot and all elements to the right are greater than or possibly equal to the pivot. This processing is called ***partitioning*** the array and is discussed in detail in the next section.
3. *Sort the elements in each of the partial arrays.* Because all elements to the left of the pivot boundary are strictly less than all those to the right, sorting each of the arrays must leave the entire array in sorted order. Moreover, since the algorithm uses a divide-and-conquer strategy, these smaller arrays can be sorted using a recursive application of Quicksort.

Partitioning the array

In the partition step of the Quicksort algorithm, the goal is to rearrange the elements so that they are divided into three classes: those that are smaller than the pivot; the pivot element itself, which is situated at the boundary position; and those elements that are at least as large as the pivot. The tricky part about partitioning is to rearrange the elements without using any extra storage, which is typically done by swapping pairs of elements.

Tony Hoare's original approach to partitioning is easy to explain in English. As in the preceding section, the discussion that follows assumes that the pivot is stored in the initial element position. Because the pivot value has already been selected when you start the partitioning phase of the algorithm, you can tell immediately whether a value is "small" or "large" by comparing it to the pivot. Hoare's partitioning algorithm then proceeds as follows:

1. For the moment, ignore the pivot element at index position 0 and concentrate on the remaining elements. Use two index values, `lh` and `rh`, to record the index positions of the first and last elements in the rest of the array, as shown:



56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

2. Move the `rh` index to the left until it either coincides with `lh` or points to an element containing a small value. In this example, the value 30 is already a small value, so the `rh` index does not need to move.

3. Move the `lh` index to the right until it coincides with `rh` or points to an element containing a value that is larger than or equal to the pivot. In this example, the `lh` index must move to the right until it points to an element larger than 56, which leads to the following configuration:

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7
			lh				rh

4. If the `lh` and `rh` index values have not yet reached the same position, exchange the elements in the `lh` and `rh` positions, which leaves the array looking like this:

56	25	37	30	95	19	73	58
0	1	2	3	4	5	6	7
			lh				rh

5. Repeat steps 2 through 4 until the `lh` and `rh` positions coincide. On the next pass, for example, the exchange operation in step 4 swaps the 19 and the 95. As soon as that happens, the next execution of step 2 moves the `rh` index to the left, where it ends up matching the `lh`, as follows:

56	25	37	30	19	95	73	58
0	1	2	3	4	5	6	7
				lh+rh			

6. Unless the chosen pivot just happened to be the smallest element in the entire array (and the code includes a special check for this case), the point at which the `lh` and `rh` index positions coincide will be the small value that is furthest to the right in the array. The only remaining step is to exchange that value with the pivot element at the beginning of the array, like this:

19	25	37	30	56	95	73	58
0	1	2	3	4	5	6	7
				↑ boundary			

   Note that this configuration meets the requirements of the partitioning step. The pivot value is at the marked boundary position, with every element to the left being smaller and every element to the right being at least as large.

An implementation of `sort` using the Quicksort algorithm is shown in Figure 9-10.

FIGURE 9-10 **Implementation of the Quicksort algorithm**

```
# File: Quicksort.py

"""This module implements the sort function using Quicksort."""

def sort(array):
    """Rearranges the elements of array in ascending order."""
    quicksort(array, 0, len(array))

# Implementation notes: quicksort
# -------------------------------
# This function sorts the array by executing the following steps:
#   1. Partition the array so that small elements precede large elements.
#   2. Recursively sort the small and large subarrays.

def quicksort(array, p1, p2):
    """Recursively sorts the elements in array[p1:p2]."""
    if p2 - p1 > 1:
        boundary = partition(array, p1, p2)
        quicksort(array, p1, boundary)
        quicksort(array, boundary + 1, p2)

# Implementation notes: partition
# -------------------------------
# This function chooses the first element as the "pivot" value.  It then
# works inward from both ends of the array looking for pairs of elements
# that are on the "wrong" side of the array relative to the pivot, at
# which point it swaps them.  At the end of the pass, the function swaps
# the pivot to the middle and returns its index as the boundary.  The
# following conditions apply when the function returns:
#   1. array[i] < pivot for all i < boundary
#   2. array[i] = pivot for i = boundary
#   3. array[i] >= pivot for all i > boundary

def partition(array, p1, p2):
    """Rearranges array[p1:p2] so small elements preceed large ones."""
    pivot = array[p1]
    lh = p1 + 1
    rh = p2 - 1
    while lh < rh:
        while lh < rh and array[rh] >= pivot:
            rh -= 1
        while lh < rh and array[lh] < pivot:
            lh += 1
        if lh < rh:
            array[lh],array[rh] = array[rh],array[lh]
    if array[lh] >= pivot:
        return p1
    array[p1] = array[lh]
    array[lh] = pivot
    return lh
```

Analyzing the performance of Quicksort

A head-to-head comparison of the actual running times for the merge sort and Quicksort algorithms appears in Figure 9-11. As you can see, this implementation of Quicksort tends to run slightly faster than the implementation of merge sort given in Figure 9-4, which is one of the reasons why programmers use it more frequently in practice. Moreover, the running times for both algorithms appear to grow in roughly the same way.

The empirical results presented in Figure 9-11, however, obscure an important point. As long as the Quicksort algorithm chooses a pivot that is close to the median value in the array, the partition step will divide the array into roughly equal parts. If the pivot value does not actually fall near the middle of the range of values, one of the two partial arrays may be much larger than the other, which defeats the purpose of the divide-and-conquer strategy. In an array with randomly chosen elements, Quicksort tends to perform well, with an average-case complexity of $O(N \log N)$. In the worst case—which paradoxically consists of an array that is already sorted—the performance degenerates to $O(N^2)$. Despite this inferior behavior in the worst case, Quicksort is so much faster in practice than most other algorithms that it has become the standard choice for general sorting procedures.

There are several strategies you can use to increase the likelihood that the pivot is in fact close to the median value in the array. One simple approach is to have the Quicksort implementation choose the pivot element at random. Although it is still possible that the random process will choose a poor pivot value, it is unlikely that it would make the same mistake repeatedly at each level of the recursive decomposition. Moreover, there is no distribution of the original array that is always bad. Given any input, choosing the pivot randomly ensures that the average-case performance for that array will be $O(N \log N)$. Another possibility, which you can explore in more detail in exercise 11, is to select a few values, typically three or five, from the array and choose the median of those values as the pivot.

FIGURE 9-11 Empirical comparison of merge sort and Quicksort

N	Merge sort	Quicksort
10	0.000025 sec	0.000013 sec
100	0.000378 sec	0.000209 sec
1000	0.00493 sec	0.00325 sec
10,000	0.0605 sec	0.0430 sec
100,000	0.763 sec	0.561 sec
1,000,000	9.05 sec	7.66 sec

You do have to be somewhat careful as you try to improve the algorithm in this way. Picking a good pivot improves performance, but also costs some time. If the algorithm spends more time choosing the pivot than it gets back from making a good choice, you will end up slowing down the implementation rather than speeding it up.

9.7 A formal definition of big-O

Because understanding big-O notation is critical to modern computer science, it is important to offer a more formal definition to help you understand why the intuitive model of big-O works and why the suggested simplifications of big-O formulas are in fact justified. Doing so, however, inevitably requires some mathematics. If mathematics scares you, try not to worry. It is more important to understand what big-O means in practice than it is to follow all the steps presented in this section.

In computer science, big-O notation is used to express the relationship between two functions, typically in an expression like this:

$$t(N) = O(f(N))$$

The formal meaning of this expression is that $f(N)$ is an approximation of $t(N)$ with the following characteristic: it must be possible to find a constant N_0 and a positive constant C so that for every value of $N \geq N_0$ the following condition holds:

$$t(N) \leq C \times f(N)$$

In other words, as long as N is sufficiently large, the function $t(N)$ is always bounded by a constant multiple of the function $f(N)$.

When it is used to express computational complexity, the function $t(N)$ represents the actual running time of the algorithm, which is usually difficult to compute. The function $f(N)$ is a much simpler formula that nonetheless provides a reasonable qualitative estimate of how the running time changes as a function of N, because the condition expressed in the mathematical definition of big-O ensures that the actual running time cannot grow faster than $f(N)$.

To see how the formal definition applies, it is useful to return to the selection sort example. Analyzing the loop structure of selection sort showed that the operations in the innermost loop were executed

$$\frac{N^2 + N}{2}$$

times and that the running time was presumably roughly proportional to this formula. When this complexity was expressed in terms of big-O notation, the constants and low-order terms were eliminated, leaving only the assertion that the execution time was $O(N^2)$, which is in fact an assertion that

$$\frac{N^2 + N}{2} = O(N^2)$$

To show that this expression is indeed true under the formal definition of big-O, all you need to do is come up with values for the constants C and N_0 such that

$$\frac{N^2 + N}{2} \leq C \times N^2$$

for all values of $N \geq N_0$. This example is unusually simple, since the inequality always holds if you set the constants C and N_0 both to 1. After all, as long as N is no smaller than 1, you know that $N \leq N^2$. It must therefore be the case that

$$\frac{N^2 + N}{2} \leq \frac{N^2 + N^2}{2}$$

But the right side of this inequality is simply N^2, which means that

$$\frac{N^2 + N}{2} \leq N^2$$

for all values of $N \geq 1$, as required by the definition.

You can use a similar argument to show that any polynomial of degree k, which can be expressed in general terms as

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_2 N^2 + a_1 N + a_0$$

is $O(N^k)$. Once again, your goal is to find constants C and N_0 such that

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_2 N^2 + a_1 N + a_0 \leq C \times N^k$$

for all values of $N \geq N_0$. As in the preceding example, you can start by choosing 1 for the value of the constant N_0. For all values of $N \geq 1$, each successive power of N is at least as large as its predecessor, so

$$N^k \geq N^{k-1} \geq N^{k-2} \geq \ . \ . \ . \geq N \geq 1$$

This property in turn implies that

$$\begin{aligned} & a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ . \ . \ . + a_1 N + a_0 \\ & \quad \leq |a_k| N^k + |a_{k-1}| N^k + |a_{k-2}| N^k + \ . \ . \ . + |a_1| N^k + |a_0| N^k \end{aligned}$$

where the vertical bars surrounding the coefficients on the right side of the equation indicate absolute value. By factoring out N^k, you can simplify the right side of this inequality to

$$(|a_k| + |a_{k-1}| + |a_{k-2}| + \ . \ . \ . + |a_1| + |a_0|) N^k$$

Thus, if you define the constant C to be

$$|a_k| + |a_{k-1}| + |a_{k-2}| + \ldots + |a_1| + |a_0|$$

you have established that

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \ldots + a_2 N^2 + a_1 N + a_0 \leq C \times N^k$$

This result proves that the entire polynomial is $O(N^k)$.

Although big-O notation is commonly used because it provides an upper bound on computational complexity, computer scientists also define two other measures that make it possible to express tighter bounds on computational complexity:

- ***Big-omega notation*** uses the Greek letter Ω to define a lower bound on the complexity of a computation. For example, the notation $\Omega(f(N))$ indicates that the running time is always at least $C \times f(N)$ for all $N \geq N_0$.
- ***Big-theta notation*** uses the Greek letter Θ to express a tight bound in which both the lower and upper bounds apply.

Summary

The most valuable concept to take with you from this chapter is that algorithms for solving a problem can vary widely in their performance characteristics. Choosing an algorithm that has better computational properties can often reduce the time required to solve a problem by many orders of magnitude. This chapter illustrates those differences by implementing several algorithms for searching and sorting.

Other important points in this chapter include:

- The problem of *searching* consists of finding a particular element in an array.
- The linear-search algorithm looks at each element in the array and therefore runs in time proportional to the size of the array.
- Binary search offers much better performance than linear search but requires that the elements of the array are sorted.
- Most algorithmic problems can be characterized by an integer N that represents the size of the problem. For algorithms that operate on arrays, it is conventional to define the problem size as the number of elements.
- The most useful qualitative measure of efficiency is *computational complexity,* which is defined as the relationship between problem size and algorithmic performance as the problem size becomes large.

- *Big-O notation* provides an intuitive way of expressing computational complexity because it allows you to highlight the most important aspects of the complexity relationship in the simplest possible form.
- When you use big-O notation, you can simplify the formula by eliminating any term in the formula that becomes insignificant as N becomes large, along with any constant factors.
- You can often predict the computational complexity of a program by looking at the nesting structure of the loops it contains.
- Two useful measures of complexity are *worst-case* and *average-case* analysis. Average-case analysis is usually more difficult to conduct.
- Divide-and-conquer strategies make it possible to reduce the complexity of sorting algorithms from $O(N^2)$ to $O(N \log N)$, which is a significant reduction.
- Many common algorithms fall into one of several *complexity classes* that include the *constant, logarithmic, linear, N log N, quadratic, cubic,* and *exponential* classes. Algorithms whose complexity class appears earlier in this list are more efficient than those that come later, at least when the problems being considered are sufficiently large.
- Problems that can be solved in *polynomial time,* which is defined to be $O(N^k)$ for some constant value k, are considered to be *tractable*. Problems for which no polynomial-time algorithm exists are considered *intractable* because solving such problems requires prohibitive amounts of time, even for problems of relatively modest size.
- Because it tends to perform extremely well in practice, most sorting programs are based on the *Quicksort algorithm,* developed by Tony Hoare, even though its worst-case complexity is $O(N^2)$.

Review questions

1. Describe the algorithmic problems of *searching* and *sorting* in your own words.

2. Estimate the number of comparisons that the binary-search algorithm would need to perform in order to find an element in an array of 1000 elements.

3. The implementation of `sort` shown in Figure 9-2 exchanges the values at positions `lh` and `rh` even if these values happen to be the same. If you change the program so that it checks to make sure `lh` and `rh` are different before making the exchange, it is likely to run more slowly than the original algorithm. Why might this be so?

4. Suppose that you are using the selection sort algorithm to sort an array of 500 values and you find that it takes 60 milliseconds to complete the operation. What

would you expect the running time to be if you used the same algorithm to sort an array of 1000 values on the same machine?

5. What is the closed-form expression that computes the sum of the series

$$N + N{-}1 + N{-}2 + \cdot\cdot\cdot + 3 + 2 + 1$$

6. In your own words, define the concept of computational complexity.

7. What are the two rules given in this chapter for simplifying big-O notation?

8. Is it technically correct to say that selection sort runs in

$$O\left(\frac{N^2 + N}{2}\right)$$

time? What, if anything, is wrong with expressing computational complexity in this form?

9. Is it technically correct to say that selection sort runs in $O(N^3)$ time? Again, what, if anything, is wrong with characterizing selection sort in this way?

10. What is the computational complexity of the following function:

```python
def mystery1(n):
    sum = 0
    for i in range(n):
        for j in range(i):
            sum += i * j
    return sum
```

11. What is the computational complexity of this function:

```python
def mystery2(n):
    sum = 0
    for i in range(10):
        for j in range(i):
            sum += j * n
    return sum
```

12. Why is it customary to omit the base of the logarithm in big-O expressions such as $O(N \log N)$?

13. What is the difference between worst-case and average-case complexity? In general, which of these measures is harder to compute?

14. Explain the roles of the constants C and N_0 in the formal definition of big-O.

15. Why does the `merge` function run in linear time?

16. Explain each of the lines in the following loop from the merge function:

```
for i in range(len(array)):
    if p2 == n2 or (p1 < n1 and a1[p1] < a2[p2]):
        array[i] = a1[p1]
        p1 += 1
    else:
        array[i] = a2[p2]
        p2 += 1
```

17. What are the seven complexity classes identified in this chapter as the most common classes encountered in practice?

18. What does the term *polynomial algorithm* mean?

19. What is the difference between a *tractable* and an *intractable* problem?

20. In Quicksort, what conditions must be true at the end of the partitioning step?

21. What are the worst- and average-case complexities for Quicksort?

Exercises

1. It is easy to write a recursive function

```
def raise_to_power(x, n)
```

that calculates x^n for a nonnegative integer n by relying on the recursive insight that

$$x^n = x \times x^{n-1}$$

Such a strategy leads to an implementation that runs in linear time. You can, however, adopt a recursive divide-and-conquer strategy that takes advantage of the fact that

$$x^{2n} = x^n \times x^n$$

Use this fact to write a recursive version of raise_to_power that runs in $O(\log N)$ time.

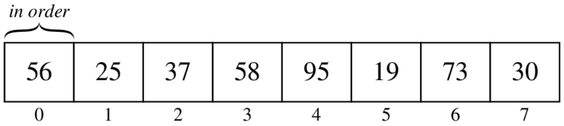
2. Write a program to produce a trace of the binary-search algorithm of the sort shown on page 302.

3. When you convert English to Pig Latin, most words turn into something that sounds vaguely Latinate but different from conventional English. There are, however, a few words whose Pig Latin equivalents just happen to be English words. For example, the Pig Latin translation of *trash* is *ashtray,* and the

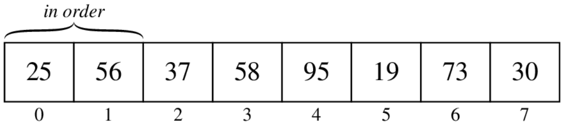
translation for *express* is *expressway.* Use the `PigLatin.py` program from Chapter 7 together with the `english` library to write a program that displays a list of all such words.

4. There are several other sorting algorithms that exhibit the $O(N^2)$ behavior of selection sort. Of these, one of the most important is ***insertion sort,*** which operates as follows. You go through each element in the array in turn, as with the selection sort algorithm. At each step in the process, however, the goal is not to find the smallest remaining value and switch it into its correct position, but rather to ensure that the values considered so far are correctly ordered with respect to each other. Although these values may shift as more elements are processed, they form an ordered sequence in and of themselves.

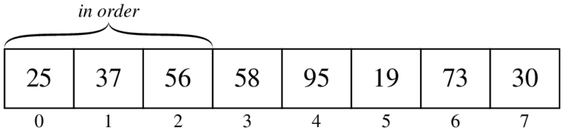
   For example, if you consider again the data used in the sorting examples from this chapter, the first cycle of the insertion sort algorithm requires no work, because an array of one element is always sorted:



On the next cycle, you need to put 25 in the correct position with respect to the elements you have already seen, which means that you need to exchange the 56 and 25 to reach the following configuration:



On the third cycle, you need to find where the value 37 should go. To do so, you must move backward through the earlier elements—which you know are in order with respect to each other—looking for the position where 37 belongs. As you go, you need to shift each of the larger elements one position to the right, which eventually makes room for the value you're trying to insert. In this case, the 56 gets shifted by one position, and the 37 winds up in position 1. Thus, the configuration after the third cycle looks like this:

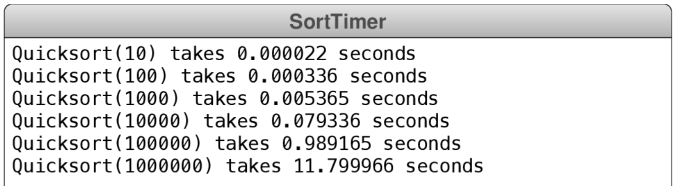


After each cycle, the initial portion of the array is always sorted, which implies that cycling through all the positions in this way will sort the entire array.

The insertion sort algorithm is important in practice because it runs in linear time if the array is already more or less in the correct order. It therefore makes sense to use insertion sort to restore order to a large array in which only a few elements are out of sequence.

Write an implementation of `sort` that uses the insertion sort algorithm. Construct an informal argument to show that the worst-case behavior of insertion sort is $O(N^2)$.

5. Write a function that keeps track of the elapsed time as it executes the `sort` procedure on a randomly chosen array. Use that function to write a program that produces a table of the observed running times for a predefined set of sizes, as shown in the following sample run:



```
Quicksort(10) takes 0.000022 seconds
Quicksort(100) takes 0.000336 seconds
Quicksort(1000) takes 0.005365 seconds
Quicksort(10000) takes 0.079336 seconds
Quicksort(100000) takes 0.989165 seconds
Quicksort(1000000) takes 11.799966 seconds
```

The best way to measure elapsed system time for programs of this sort is to call the standard function `time.perf_counter`, which returns the most accurate time value available on the computer expressed in seconds. The time base for the `perf_counter` function is not defined, but you can nonetheless measure elapsed time using the following code pattern:

```
start = time.perf_counter()
. . . Perform some calculation . . .
elapsed = time.perf_counter() - start
```

Unfortunately, calculating the time requirements for a program that runs quickly requires some subtlety because there is no guarantee that the system clock unit is precise enough to measure elapsed time accurately. For example, if you used this strategy to time the process of sorting 10 integers, it might well turn out that the value of `elapsed` at the end of the code fragment is 0. The reason is that the processing unit on most machines can execute many instructions in the space of a single clock tick—almost certainly enough to get the entire sorting process done for an array of 10 elements. Because the system's internal clock may not tick in the interim, the two values returned by `time.perf_counter` are likely to be the same.

The best way to get around this problem is to repeat the calculation many times between the two calls to `time.perf_counter`. For example, if you want to determine how long it takes to sort 10 numbers, you can perform the sort-10-numbers experiment 1000 times in a row and then divide the total elapsed time by 1000. This strategy gives you a timing measurement that is much more accurate.

6. Write a function `elapsed_time` that generalizes the computation from the preceding exercise by returning how many seconds are required to call a client-supplied function. The `elapsed_time` function takes two parameters. The first is a callback function that implements the operation whose running time you want to measure. The second argument, which is optional and defaults to 1, indicates how many repetitions of the function call should occur between the time measurements.

7. Suppose you know that all the values in an integer array fall into the range 0 to 9999. Show that it is possible to write a $O(N)$ algorithm to sort arrays with this restriction. Implement your algorithm and evaluate its performance by taking empirical measurements using the strategy outlined in exercise 8. Explain why the algorithm is less efficient than selection sort for small values of *N*.

8. Change the implementation of the Quicksort algorithm so that, instead of picking the first element in the array as the pivot, the `partition` function chooses the median of the first, middle, and last elements.

9. Although $O(N \log N)$ algorithms are more efficient than $O(N^2)$ algorithms if you are sorting a large array, the simplicity of quadratic algorithms like selection sort often means that they perform better for small values of *N*. This fact raises the possibility of developing a strategy that combines the two algorithms, using Quicksort for large arrays but selection sort whenever the size of the arrays becomes less than some threshold called the ***crossover point.*** Approaches that combine two different algorithms to exploit the best features of each are called ***hybrid strategies.***

   Reimplement `sort` using a hybrid of the Quicksort and selection sort strategies. Experiment with different values of the crossover point and determine what value gives the best performance. The optimal value of the crossover point depends on the specific timing characteristics of your computer and will change from system to system.

10. Another interesting hybrid strategy for the sorting problem is to start with a recursive implementation of Quicksort that simply returns when the size of the array falls below a certain threshold. When this function returns, the array is not sorted, but all the elements are relatively close to their final positions. At this point, you can use the insertion sort algorithm presented in exercise 7 on the entire array to fix any remaining problems. Because insertion sort runs in linear time on arrays that are mostly sorted, this two-step process may run more quickly

than either algorithm alone. Write an implementation of the `sort` function that uses this hybrid approach.

11. Suppose you have two functions, f and g, for which $f(N)$ is less than $g(N)$ for all values of N. Use the formal definition of big-O to prove that

$$15f(N) + 6g(N)$$

is $O(g(N))$.

12. Use the formal definition of big-O to prove that N^2 is $O(2^N)$.

13. Exercise 1 shows that it is possible to compute x^n in $O(\log N)$ time. This fact in turn makes it possible to write an implementation of the function `fib(n)` that also runs in $O(\log N)$ time, which is much faster than the traditional iterative version.

To do so, you need to rely on the somewhat surprising fact that the Fibonacci function is closely related to a value called the ***golden ratio,*** which has been known since the days of Greek mathematics. The golden ratio, which is usually designated by the Greek letter phi (φ), is defined to be the value that satisfies the equation

$$\varphi^2 - \varphi - 1 = 0$$

Because this is a quadratic equation, it actually has two roots. If you apply the quadratic formula, you will discover that these roots are

$$\varphi = \frac{1+\sqrt{5}}{2}$$

$$\hat{\varphi} = \frac{1-\sqrt{5}}{2}$$

In 1718, the French mathematician Abraham de Moivre discovered that the n^{th} Fibonacci number can be represented in closed form as

$$\frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$$

Moreover, because $\hat{\varphi}^n$ is always very small, the formula can be simplified to

$$\frac{\varphi^n}{\sqrt{5}}$$

rounded to the nearest integer.

Use this formula and the `raise_to_power` function from exercise 1 to write an entirely recursive implementation of `fib(n)` that runs in $O(\log N)$ time.

14. If you're ready for a real algorithmic challenge, write the function

```
def find_majority_element(array)
```

that takes an array of nonnegative integers and returns the ***majority element,*** which is defined to be a value that occurs in a majority (at least 50 percent plus one) of the element positions. If no majority element exists, the function should return –1. Your function must also meet the following conditions:

- It must run in $O(N)$ time.
- It must use $O(1)$ additional space. In other words, it may use individual temporary variables but may not allocate any additional array storage. This condition also rules out recursive solutions, because the space required for stack frames grows with the depth of the recursion.
- It must not change any of the values in the array.

15. If you enjoyed the previous problem, here's an even more challenging one that was at one time an interview question at Microsoft. Suppose that you have an array of N elements, in which each element has a value in the range 1 to $N-1$, inclusive. Since there are N elements in the array and only $N-1$ possible values to store in each slot, there must be at least one value that is duplicated in the array. There may, of course, be many duplicated values, but you know that there must be at least one by virtue of what mathematicians call the ***pigeonhole principle:*** if you have more items to put into a set of pigeonholes than the number of pigeonholes, there must be some pigeonhole that ends up with more than one item.

Your task in this problem is to write a function

```
def find_duplicate(array)
```

that takes an array whose elements are constrained to be in the 1 to $N-1$ range and returns one of the duplicated elements. As in the previous exercise, your solution must meet the following conditions:

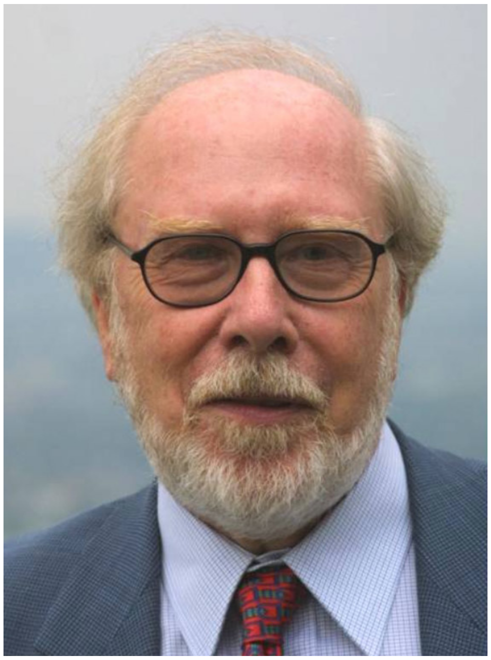
- It must run in $O(N)$ time.
- It must use $O(1)$ additional space.
- It must not change any of the values in the array.

CHAPTER 10

Classes and Objects

I have always tried to identify and focus in on what is essential and yields unquestionable benefits. For example, the inclusion of a coherent and consistent scheme of data type declarations in a programming language I consider essential.

—Niklaus Wirth, Turing Award Lecture*,* 1984



Niklaus Wirth (1934–)

Swiss computer scientist Niklaus Wirth designed and engineered several early programming languages including Euler, PL360, Algol-W, and Pascal, which became the standard language for introductory computer science throughout the 1970s and 1980s. Although Grace Hopper's COBOL language described on page 35 included support for data records, Pascal was the first programming language to integrate the record concept into the type system in a consistent way. In 1975, Wirth published an influential book entitled *Algorithms + Data Structures = Programs,* which offers an eloquent defense of the idea that data structures are as fundamental to programming as algorithms. Niklaus Wirth received the ACM Turing Award in 1984.

When you learned about lists in Chapter 8, you took your first steps toward understanding an extremely important idea in computer programming: the use of compound data structures to represent collections of information. When you use a list in the context of a program, you are able to combine an arbitrarily large number of data values into a single structure that has conceptual integrity as a whole. If you need to do so, you can select particular elements of the list and manipulate them individually. But you can also treat the list as a unit and manipulate it all at once.

The ability to take individual values and organize them into coherent units is an essential feature of modern programming. Functions allow you to unify many independent operations under a single name. Compound data structures—of which lists are only one example—offer the same facility in the data domain. In each case, being able to aggregate the tiny pieces of a program into a single, higher-level structure provides both conceptual simplification and a significant increase in your power to express ideas. The power of unification is hardly a recent discovery; it has given rise to social movements and to nations, as reflected in the labor anthem that proclaims “the union makes us strong” and the motto *“E Pluribus Unum*”—“out of many, one”—on the Great Seal of the United States.

Although lists are a powerful tool when you need to model real-world data that can be represented as a sequence of ordered elements, it is also important to be able to combine unordered data values into a single unit. This chapter describes how classes and objects enable Python programmers to use the object-oriented paradigm, which is defined by two main principles. The first is ***encapsulation,*** which is the technique of combining data values and methods into a single structure. The second is ***inheritance,*** which allows programmers to define hierarchies in which classes automatically acquire behavior from their ancestors in the hierarchy. Encapsulation is discussed in this chapter, and inheritance is covered in Chapter 13 after you have had more of a chance to work with objects.

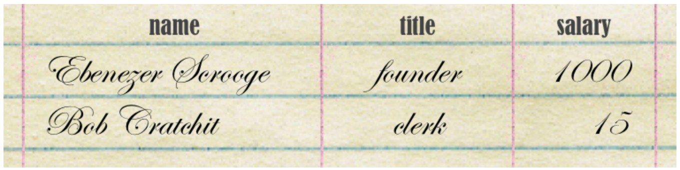
10.1 Records and tuples

As you learned in Chapter 8, Python’s lists are an extension of an earlier, more primitive concept called arrays. In much the same way, Python’s implementation of classes and objects grows out of an older programming concept called a ***record,*** which is any data structure that combines several distinct values into an integrated whole. In this section, you will learn how to create Python objects that model traditional records so that you have a foundation from which to build a more comprehensive understanding of how objects in Python work.

The term *record* has its origin in the world before computing, where it refers to a collection of data values pertaining to a single entity. For example, census records—which were kept on paper until Hermann Hollerith invented punch cards for the 1880 census—collects the information pertaining to a single individual. Each census record presumably includes for the person's name, age, address, occupation, and any other data collected by the census bureau. Loan records for a bank would presumably include information such as the date of the loan, the name of the borrower, the amount of the loan, and the interest rate. Employee records for a firm presumably include information like the employee's name, job title, and salary. The individual components of a record are generically called ***fields.***

A simple example of records

The concepts of records and fields are best illustrated by example. At the rather small firm of Scrooge and Marley that appears in *A Christmas Carol* by Charles Dickens, the employee ledger might look something like this:



name	title	salary
Ebenezer Scrooge	founder	1000
Bob Cratchit	clerk	15

The preprinted top line in the ledger provides names for the three data fields: one for the employee's name, one for the job title, and one for the weekly salary in shillings. Each subsequent line represents a record for a single employee, giving the value of these three fields. The entries in the paper copy of the ledger book show that Scrooge and Marley has two employees: a founder named Ebenezer Scrooge who earns 1000 shillings per week and a clerk named Bob Cratchit struggling to make ends meet on the pitiful salary of 15 shillings a week.

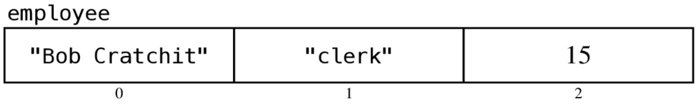
Representing records as tuples

The simplest way to represent a record in Python is to use a built-in data structure called a ***tuple,*** which is used in both computer science and mathematics to refer to an ordered, immutable collection of elements. In Python, you create a tuple by enclosing its elements in parentheses. For example, the assignment statement

```
employee = ("Bob Cratchit", "clerk", 15)
```

creates a tuple with three elements and assigns that tuple to the variable `employee`.

In its internal structure, a tuple is similar to a list. If, for example, you use square brackets instead of parentheses to define this variable, the box diagram for the variable has exactly the same form:



You can also apply many of the same operators to these two types. Both tuples and lists are sequences and therefore implement the operations for sequences described in Chapter 8. Tuples, however, differ from lists in two critical ways:

1. *Tuples are immutable.* Although you can change the elements of a list through direct assignment or by calling any of the methods that manipulate their elements, tuples are immutable. You cannot, for example, assign new values to the elements of a tuple or add values to the end using the += operator. Once you have created a tuple, its elements remain unchanged.
2. *Tuples are used primarily to represent records rather than sequences of values.* In Python, lists are used primarily to represent ordered sequences of objects, all of which have the same type. Tuples, by contrast, represent collections of values in which order is often not important and in which the types of the individual elements can vary.

The easiest way to understand the functional similarity of tuples and records is to think about the etymology of *tuple,* which comes from the suffix of more specific words like *quintuple, sextuple, septuple, octuple,* and so on. If you need to represent a data structure that contains a particular number of elements—which might well be a smaller number for which the English word does not end with the suffix *tuple,* such as *pair, triple, or quadruple*—using a tuple probably makes sense. By contrast, programmers use lists when they don't know how many elements a list contains or when instances of that list can differ in their number of elements.

The primary advantage of representing a record as a Python tuple is that doing so eliminates the overhead of defining a new class. At the same time, representing records as tuples has several disadvantages that often make the class-based strategy worth the additional cost. If you need to change the fields of a record, the fact that tuples are immutable makes it necessary to choose a different strategy. In addition, the fact that elements of a tuple are identified using an index number rather than a name can make programs that use a tuple-based strategy more difficult to understand.

As it happens, Python programmers rarely refer to the elements of a tuple by their index number because the language offers a convenient syntactic form called ***destructuring assignment*** for splitting a tuple into its component elements. If, for example, the variable `employee` contains the tuple defined on page 339, destructuring assignment allows you to extract the individual fields like this:

```
name, title, salary = employee
```

When Python notices that the left side of the assignment consists of three variables and the right side is a tuple of length 3, it automatically unpacks the component values, assigning the first element of the tuple to the variable `name`, the second element to the variable `title`, and the third element to the variable `salary`. The fact that these elements have numeric indices is no longer explicit in the program but is simply an implementation detail.

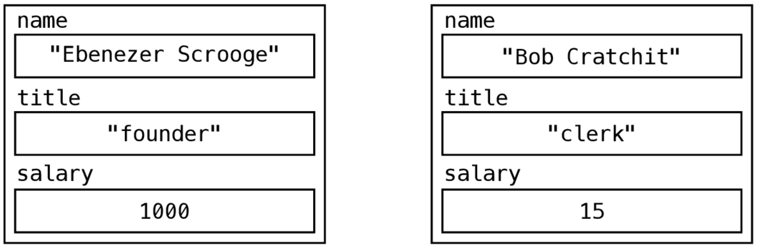
Because tuples are typically used to model structures that appear in the real world as pairs, triples, quadruples, and so on, there are not many applications in which it makes sense to have a tuple with just one element or no elements at all. On those rare occasions when you want to create a tuple containing a single value, Python requires a comma after that value to differentiate a tuple with one element from a parenthesized expression. For example, to create a tuple containing the number 42 as its only element, Python requires you to write

```
    (42,)
```

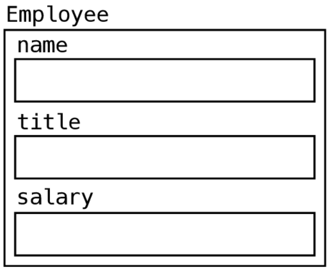
If you leave out the comma, Python interprets `(42)` as a parenthesized expression.

Representing records as objects

A more sophisticated strategy for representing the employee records for Scrooge and Marley is to encode the information for each record as an object containing the necessary fields, which are more often called ***attributes*** in the context of a Python object. The name and title attributes are presumably strings, and the salary attribute is a number. If you draw each of the records as a box containing interior boxes for each attribute, the information for Scrooge and Marley looks like this:



As you know from the discussion of classes and objects in Chapter 4, objects in Python are instances of a class, which provides a template for all objects that belong to that class. The `GRect` class in the Portable Graphics Library, for example, acts as a template for all `GRect` objects. Thus, while you can display many distinct `GRect` objects on the graphics window, there is only one `GRect` class. In much the same way, each of the objects representing an employee of Scrooge and Marley is an instance of an `Employee` class that defines the general structure shared by all employees. The Employee class therefore acts as a fill-in-the-blanks template that looks like this:



Defining an empty class template

Python's model for defining classes is sufficiently complex that it makes sense to present it in two stages. To make the general process of creating new objects clear, the next few paragraphs show how you can define a class without specifying any of its attributes. And although the rest of this chapter will show you how to add more sophisticated behavior to a class, defining an empty class with no internal structure is not without its uses. The GState class introduced in Chapter 6 works in exactly this way.

In this simplified model, the class definition for Employee looks like this:

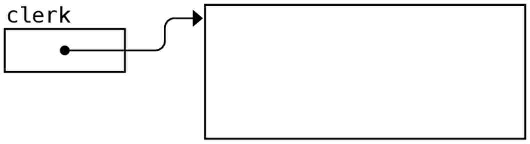
```
class Employee:
    """Fill in the details later"""
```

Python's syntactic rules require that the body of a class contain at least one line, but the docstring comment is sufficient for this purpose, although you can also use the keyword pass for this purpose.

Defining a new class automatically creates a function that acts as a constructor for that class. For example, the statement

```
clerk = Employee()
```

creates a new instance of the Employee class. Given the current empty definition of Employee, that instance contains no data but instead serves as a blank slate:



As the diagram makes clear, the value stored in `clerk` is not the object itself but is instead a reference to the blank-slate object. In Python, having a reference to an object makes it possible to create new attributes within it. That fact makes it possible for the client to fill in the values of the missing attributes.

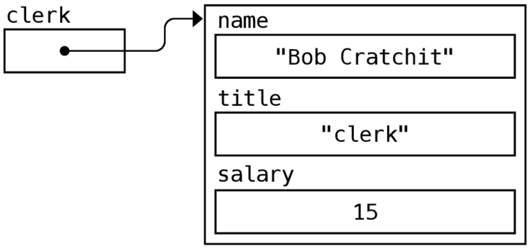
Properties within an object are indicated by name using the syntax

object.*name*

in which the object reference and the name of the attribute are separated by a period, which in programming contexts is more often called a ***dot.*** Because attributes are assignable, you can fill in the details for Bob Cratchit like this:

```
clerk.name = "Bob Cratchit"
clerk.title = "clerk"
clerk.salary = 15
```

Those statements create the necessary attributes and assign values to them so that the diagram looks like this:



Defining a constructor

The point of the example in the preceding section is to introduce the notions of object references and dot selection so that you are in a better position to understand how to initialize the attributes of an `Employee` object from the implementation side rather than as a client operation. As noted in Chapter 4, functions that create new instances of a class are called *constructors.* The division of responsibility between client and implementation becomes much clearer if the `Employee` class defines a constructor that takes the values of name, title, and salary and uses those to create a fully initialized object in which those values are stored in the appropriate internal attributes.

To create a constructor in Python, you need to define a special method called `__init__` inside the body of the class. The header for the `__init__` method has the following form:

```
def __init__(self, parameters):
```

where *parameters* is a list of the parameter names passed to the constructor. The first parameter to `__init__` is conventionally named `self` and contains a reference to the same blank-slate object described in the preceding section. The body of the constructor typically assigns the parameter values to the corresponding attributes in the object, using `self` as a reference to the object state.

For example, the following code defines an enhanced version of the `Employee` class that includes the constructor:

```
class Employee:
    """This class represents a simple employee object"""

    def __init__(self, name, title, salary):
        self.name = name
        self.title = title
        self.salary = salary
```

With this definition, the client of the `Employee` class can create the entry for Bob Cratchit like this:

```
clerk = Employee("Bob Cratchit", "clerk", 15)
```

If you look only at the increased complexity of the `Employee` class, this change may not seem like much of an improvement. Arguably, the code in the preceding section is easier to understand, if for no other reason than it does not require knowing anything about the `__init__` method or the `self` parameter. Those details, however, exist only on the implementation side and are not exposed to clients. On the client side, creating and initializing a new `Employee` object now takes a single line instead of four. The savings is even more evident if, for example, you want to create a list containing the employee records for everyone at Scrooge and Marley, which requires only the following code:

```
SCROOGE_AND_MARLEY = [
    Employee("Ebenezer Scrooge", "founder", 1000),
    Employee("Bob Cratchit", "clerk", 15)
]
```

Getters and setters

Although it is possible for clients to use dot selection to refer directly to attributes within an object, the conventions of modern object-oriented programming favor a different approach that seeks to maintain the integrity of the object. If a client needs access to a particular attribute, the class defines a method that returns that value. Such methods are called ***getters.*** In the Portable Graphics Library, for example, you don’t get the *x* coordinate of a GRect stored in the variable box by using a selection expression like box.x but instead by calling box.get_x().

It is easy to imagine that clients of the Employee class would need access to each of the attributes, which means that the Employee class should define the getter methods get_name, get_title, and get_salary. Like the constructor, these methods take a parameter named self, which Python supplies automatically whenever you make a method call using the receiver syntax. The definition of the get_name method, for example, looks like this:

```
def get_name(self):
    """Returns the name of the employee"""
    return self.name
```

In addition to getters, classes that define mutable types typically include methods called ***setters*** that set individual attributes. Setters are less common than getters, and it is important to think carefully about whether you need a setter for an individual attribute. In the case of the Employee class, for example, it is more likely that clients would want to be able to change an employee's salary or job title than the employee's name, which is easier to think about as a permanent part of the employee's record. The employee management application can implement name changes, on the infrequent occasions when they occur, by removing the record for the old name and replacing it with a new one. This argument suggests that the Employee class should export the methods set_title and set_salary, but not set_name. The set_salary method has the following form:

```
def set_salary(self, salary):
    """Resets the salary of this employee"""
    self.salary = salary
```

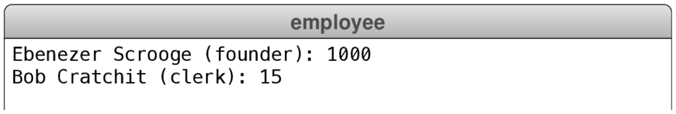
Converting objects to strings

In addition to the constructor and any methods necessary to define the behavior of a class, most Python classes will also define a special method called __str__, which specifies how to convert the object to a string. Defining this method makes it possible to see the value of an object and is invaluable for debugging. Figure 10-1 shows an implementation of the Employee class that includes a __str__ method, along with the constructors, getters, and setters defined earlier.

FIGURE 10-1 Definition of an Employee class

```python
# File: employee.py

"""
This module defines a simple Employee class.
"""

class Employee():
    """This class encapsulates information about an employee."""

    def __init__(self, name, title, salary):
        """Creates a new Employee object from the arguments"""
        self.name = name
        self.title = title
        self.salary = salary

    def __str__(self):
        """Returns a string representation of this employee"""
        return self.name + " (" + self.title + "): " + str(self.salary)

    def get_name(self):
        """Returns the name of this employee"""
        return self.name

    def get_title(self):
        """Returns the job title of this employee"""
        return self.title

    def get_salary(self):
        """Returns the weekly salary of this employee"""
        return self.salary

    def set_title(self, title):
        """Resets the job title of this employee"""
        self.title = title

    def set_salary(self, salary):
        """Resets the salary of this employee"""
        self.salary = salary

# Constant employee roster for testing

SCROOGE_AND_MARLEY = [
    Employee("Ebenezer Scrooge", "founder", 1000),
    Employee("Bob Cratchit", "clerk", 15)
]

# Startup code

if __name__ == "__main__":
    for emp in SCROOGE_AND_MARLEY:
        print(emp)
```

The `employee` module in Figure 10-1 also includes a test program that uses the roster of employees at Scrooge and Marley to test the `__init__` constructor and the `__str__` method. Running `employee` as a program produces the following output:



```
Ebenezer Scrooge (founder): 1000
Bob Cratchit (clerk): 15
```

A more complete test program would also test the implementation of the getters and setters, but that program listing would no longer fit on a single page.

10.2 Representing points

One of the advantages of using records—no matter whether they are implemented as tuples or as objects—is that doing so makes it possible to combine several related pieces of information into a composite value that can be manipulated as a unit. An important practical application of this principle arises when you need to represent a point in two-dimensional space, such as the drawing surface of the graphics window. So far, the graphical programs in this text have kept track of independent *x* and *y* coordinates, which is sufficient for many applications. As you move on to more complex graphical programs, however, it is useful to store the *x* and *y* values in an integrated unit called a ***point.***

Combining the *x* and *y* coordinates into a single data structure makes it possible to work with points as composite values. You can assign a point to a variable, create an array of points, pass a point as an argument to a function, and return a point as a result. This last example—returning a point as the result of a function call—adds a new capability that would otherwise be difficult to achieve. A Python function is allowed to return only a single value, so there is no way for a function to return the *x* and *y* coordinates independently. A function can, however, return a point, from which the caller can extract the *x* and *y* coordinates, if necessary.

Representing points as tuples

Although the Python implementation of the `pgl` module defines points using a class called `GPoint` to maintain consistency with other implementations of the graphics library other Python packages (including the one used to implement the `pgl` module) implement a point in two-dimensional space as a tuple with two elements representing the *x* and *y* coordinates. Thus, the tuple `(0,0)` represents the origin of the graphics window and the tuple `(1,4)` represents the point whose *x* coordinate is 1 and whose *y* coordinate is 4.

From the client's point of view, it is convenient to represent points as tuples because the Python syntax precisely matches the conventional mathematical form.

And although it might initially seem odd to select the individual coordinates of a point using the index numbers 0 and 1 instead of the names *x* and *y*, it is not at all hard to get used to that convention, particularly if you use destructuring assignment to separate the component values.

Defining points as a class

The Portable Graphics Library defines a class called GPoint that encapsulates an *x*-*y* coordinate pair. The definition of this class appears in Figure 10-2. Like the definition of the Employee class in Figure 10-1, the implementation of the GPoint class defines a constructor, getter methods for the *x* and *y* coordinates, and a method to convert a GPoint to a string. The code also illustrates a technique that is useful in maintaining an effective separation between the implementation and its clients. The

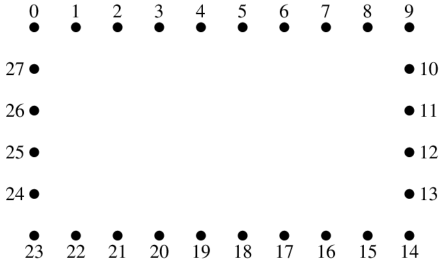
FIGURE 10-2 Implementation of the GPoint class in pgl.py

```
# Class: GPoint

class GPoint:
    """
    This class contains real-valued x and y fields. It is used to
    represent a location on the graphics plane.
    """

# Constructor: GPoint

    def __init__(self, x=0, y=0):
        """Initializes a point with the specified coordinates."""
        self._x = x
        self._y = y

# Public method: get_x

    def get_x(self):
        """Returns the x component of the point."""
        return self._x

# Public method: get_y

    def get_y(self):
        """Returns the y component of the point."""
        return self._y

# Public method: __str__

    def __str__(self):
        """Returns the string representation of a point."""
        return "(" + str(self._x) + ", " + str(self._y) + ")"
```

names of the attributes that hold the *x* and *y* coordinates are not called x and y but instead have the names _x and _y, which begin with an underscore. Python uses the leading-underscore convention to mark variables and methods that belong to the implementation and are considered off-limits to clients. Although Python cannot prevent clients from referring directly to these variables using the attribute names _x and _y, the underscore warns clients that any such access comes at their own risk. The implementer is free to remove or change the names of these variables, even though doing so would likely break client programs that ignored the warning signs these underscores represent.

The GPoint class is often useful in graphical programs because it allows a program to treat a coordinate pair as a single object. The YarnPattern.py program in Figure 10-3, for example, creates beautiful patterns using only GLine objects. Each of the GLine objects connects two points stored in a list using a process that you can easily carry out in the real world. Conceptually, the process begins by arranging pegs around the perimeter of the window so that they are evenly spaced along all four edges.

To get a sense of how this program operates, imagine that you start with a small graphics window in which the pegs are numbered clockwise from the upper left:



From here, you create a figure by winding a single piece of yarn through the pegs, starting at peg 0 and then moving ahead DELTA spaces on each cycle. For example, if DELTA is 11, the yarn goes from peg 0 to peg 11, then from peg 11 to peg 22, and then (counting past the beginning) from peg 22 to peg 5, as follows:

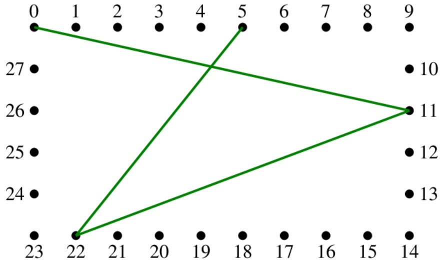
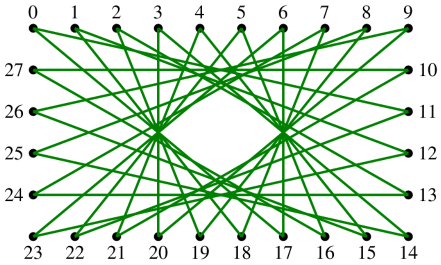


FIGURE 10-3 **Program to simulate threading yarn around a series of pegs**

```
# File: YarnPattern.py

"""
This program uses the GLine class to simulate winding a piece of yarn
around an array of pegs along the edges of the graphics window.
"""

from pgl import GWindow, GLine, GPoint

# Constants

PEG_SEP = 12               # The separation between pegs in pixels
N_ACROSS = 80              # Number of PEG_SEP units horizontally
N_DOWN = 50                # Number of PEG_SEP units vertically
DELTA = 113                # Number of pegs to skip on each cycle

# Derived constants

GWINDOW_WIDTH = N_ACROSS * PEG_SEP
GWINDOW_HEIGHT = N_DOWN * PEG_SEP

def yarn_pattern():
    """Creates the yarn pattern on the graphics window."""
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    pegs = create_pegs()
    this_peg = 0
    next_peg = -1
    while this_peg != 0 or next_peg == -1:
        next_peg = (this_peg + DELTA) % len(pegs)
        p0 = pegs[this_peg]
        p1 = pegs[next_peg]
        line = GLine(p0.get_x(), p0.get_y(), p1.get_x(), p1.get_y())
        line.set_color("Green")
        gw.add(line)
        this_peg = next_peg

def create_pegs():
    pegs = [ ]
    for i in range(N_ACROSS):
        pegs.append(GPoint(i * PEG_SEP, 0))
    for i in range(N_DOWN):
        pegs.append(GPoint(N_ACROSS * PEG_SEP, i * PEG_SEP))
    for i in range(N_ACROSS, 0, -1):
        pegs.append(GPoint(i * PEG_SEP, N_DOWN * PEG_SEP))
    for i in range(N_DOWN, 0, -1):
        pegs.append(GPoint(0, i * PEG_SEP))
    return pegs

# Startup code

if __name__ == "__main__":
    yarn_pattern()
```

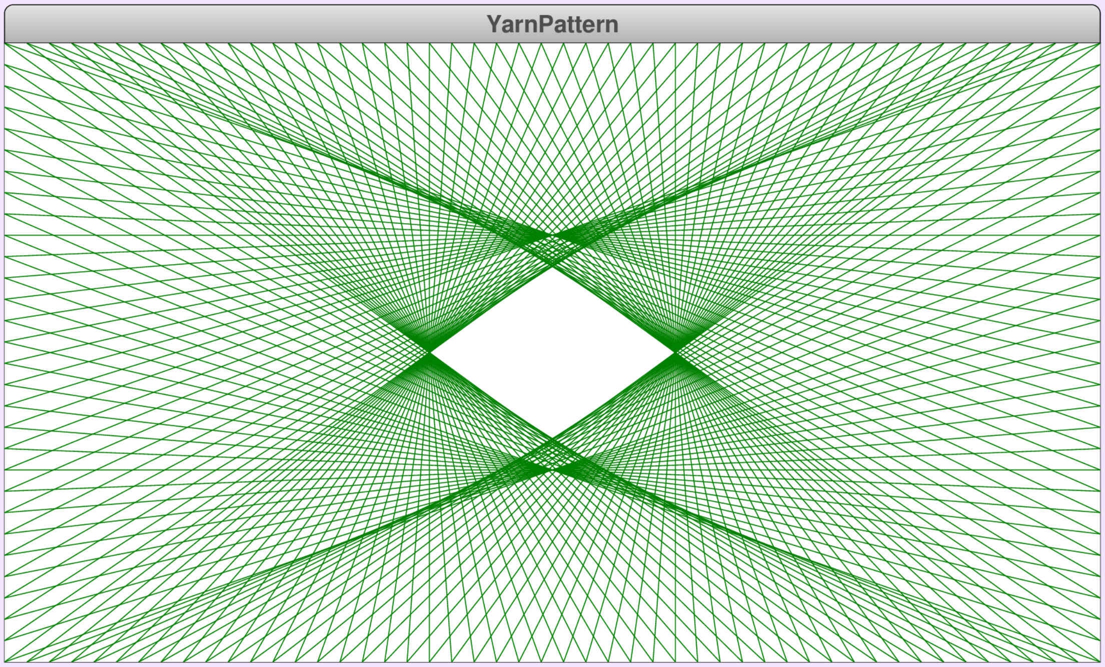
The process continues until the yarn returns to peg 0, creating the following pattern:



The program in Figure 10-3 begins by calling `create_pegs` to create the array of points around the perimeter. The code creates pegs from left to right across the top, from top to bottom along the right side, from right to left across the bottom, and finally from bottom to top along the left side. When `create_pegs` returns, the `YarnPattern.py` program starts at peg 0 and then advances `DELTA` steps on each cycle until the index loops back to 0. On each cycle, the implementation creates a `GLine` object to connect the current point in the array with the previous one.

Figure 10-4 shows a larger example of the output produced by `YarnPattern.py` that uses the values of `N_ACROSS` and `N_DOWN` shown in the program listing.

FIGURE 10-4 **Sample run of the yarn-pattern program**



10.3 Rational numbers

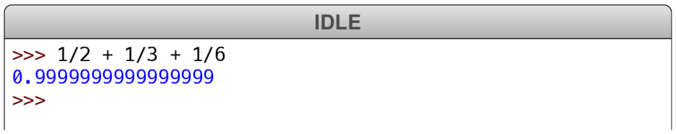
Although the `GPoint` class from section 10.2 illustrates the basic mechanics used to define a new class, developing a solid understanding of the topic requires you to consider more sophisticated examples. This section walks you through the design of a class to represent ***rational numbers,*** which are those numbers that can be represented as the quotient of two integers.

In some respects, rational numbers are similar in concept to the `float` class in Python. Both rational numbers and floating-point numbers can represent fractional values, such as 1.5, which is the rational number 3/2. The difference is that rational numbers are exact, while the built-in implementation of floating-point numbers relies on approximations limited by the hardware precision.

To get a sense of why this distinction might be important, consider the arithmetic problem of adding together the following fractions:

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{6}$$

Basic arithmetic makes it clear that the answer is 1, but Python's floating-point arithmetic gives a different result, as the following IDLE session shows:



```
>>> 1/2 + 1/3 + 1/6
0.9999999999999999
>>>
```

The problem is that the memory cells used to store numbers have a limited storage capacity, which in turn restricts the precision they can offer. Within the limits of Python's standard arithmetic, the sum of one-half plus one-third plus one-sixth is closer to 0.9999999999999999 than it is to 1.0. By contrast, rational numbers are not subject to this type of rounding error because no approximations are involved. What's more, rational numbers obey well-defined arithmetic rules, which are summarized in Figure 10-5. Since Python does not include rational numbers among its predefined types, you have to implement `Rational` as a new class.

FIGURE 10-5 Rules for rational arithmetic

Addition

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Subtraction

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

Multiplication

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

Division

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

A general strategy for defining new classes

When you work in object-oriented languages, designing new classes is the most important skill you need to master. As with much of programming, designing a new class is as much an art as it is a science. Designing a class requires a strong sense of aesthetics and sensitivity to the needs of any clients who will use that class as a tool. Experience and practice are the best teachers, but following a general design framework can help get you started along this path.

From my own experience, I've often found the following approach helpful:

1. *Think generally about how clients are likely to use the class.* From the very beginning of the process, it is essential to remember that library classes are designed to meet the needs of clients and not for the convenience of the implementer. In a professional context, the most effective way to ensure that a new class meets those needs is to involve clients in the design process. At a minimum, however, you need to put yourself in the client role as you sketch the outlines of the class design.
2. *Determine what information belongs in the private state of each object.* Although the private data maintained in the attributes of the object is conceptually part of the implementation, it simplifies the later design phases if you have an intuitive sense of what information objects of this class contain.
3. *Determine the parameters needed for the constructor.* Whenever a client creates a new instance of your class, the first step in the process is making a call to the constructor. As part of the design phase, you need to decide what information the client will want to supply at the time the object is created, which in turn determines what parameters the constructor will need. In the case of the `GPoint` class, the client must supply the *x* and *y* coordinates.
4. *Enumerate the operations that will become the public methods of the class.* In this phase, the goal is to define the names and parameters for the methods exported by the class, thereby adding specificity to the general outline you developed at the beginning of the process.
5. *Code and test the implementation.* Once you have completed the overall design, you need to implement it. Writing the actual code is not only essential to having a working program but also offers validation for the design. As you write the implementation, it is sometimes necessary to revisit the interface design if, for example, you discover that a particular feature is difficult to implement at an acceptable level of efficiency. As the implementer, you also have a responsibility to test your implementation to ensure that the class delivers the functionality it claims.

The sections that follow carry out these steps for the `Rational` class.

Adopting the client perspective

As a first step toward the design of the `Rational` class, you need to think about what features your clients are likely to need. In a large company, you might have various implementation teams that need to use rational numbers and could give you a good sense of what operations should be part of that class. In that setting, it would be useful to work together with those clients to agree on a set of design goals.

Since this example is a textbook scenario, however, it isn't possible for you to schedule meetings with prospective clients. The primary purpose of the example is to illustrate the structure of class definitions in Python. Given these limitations and the need to manage the complexity of the example, it makes sense to implement only the arithmetic operations defined in Figure 10-5.

Specifying the private state of the `Rational` class

For the `Rational` class, the private state is easy to specify. A rational number is defined as the quotient of two integers. Each rational object must therefore keep track of these two values. In the implementation, these variables are called `_num` and `_den`. The names are abbreviations of the mathematical terms ***numerator*** and ***denominator*** used to refer to the upper and lower parts of a fraction. The underscore at the beginning of these names indicates that these variables are the attribute of the implementation and should not be examined or changed by clients.

Defining the `Rational` constructor

Given that a rational number represents the quotient of two integers, the constructor will presumably take two numbers representing the components of the fraction. Defining the constructor in this way makes it possible, for example, to define the rational number one-third by calling `Rational(1, 3)`. To make it easier to work with integers—which are just rational numbers whose denominator is 1—it is useful to allow clients to call the constructor with a single integer argument, letting Python supply the default value of 1 for the denominator. Given this design, the header line for the constructor will look like this:

```
def __init__(self, num, den=1):
```

To help clients discover errors in the use of the `Rational` class, the constructor should check that the supplied parameters correspond to a legal rational number. For example, the constructor should check that that `den` is not 0, because division by zero is not a legal operation. The code can check that this condition holds by making the following test:

```
if den == 0:
    raise ValueError("Illegal denominator value")
```

If the client calls the constructor with `den` equal to 0, the constructor responds by raising the built-in `ValueError` exception with an explanatory string.

Beyond checking for zero in the denominator, there are other restrictions that you might want to impose on the values of the parameters `num` and `den` to ensure that the implementation operates correctly. The code becomes much simpler if the constructor can guarantee that every rational number has a consistent, unique representation, which is not true if the client can supply any values for `num` and `den`. The rational number one-third, for example, can be written as a fraction in any of the following ways:

$$\frac{1}{3} \qquad \frac{2}{6} \qquad \frac{100}{300} \qquad \frac{-1}{-3}$$

Because these fractions all represent the same rational number, it is appropriate for them to have the same internal representation. Mathematicians achieve this goal by insisting on the following rules:

- The denominator is always positive, which means that the sign of the value is stored with the numerator.
- The rational number 0 is always represented as the fraction 0/1.
- The fraction is always expressed in lowest terms, which means that both the numerator and the denominator are divided by their greatest common divisor.

Enumerating the methods for the `Rational` class

Along with the constructor, the `Rational` class must define additional methods that implement the behavior of the class. As with the `Employee` and `GPoint` classes defined earlier in the chapter, it is good practice to define a `__str__` method that converts a `Rational` object to a string. More importantly, the `Rational` class must define methods for the four arithmetic operators. For the moment, the simplest strategy is to define methods called `add`, `sub`, `mul`, and `div` that perform the necessary computations. Each of these methods takes two parameters. The `self` parameter that appears first in the parameter list of every method holds a reference to the `Rational` number that is implementing the operator. The second parameter, which is called `r` in the implementation, contains a reference to the other `Rational` value involved in the computation.

Implementing the `Rational` class

Figure 10-6 shows the code for a `rational` module that implements a simple version of the `Rational` class. As suggested in the preceding section, the class includes a constructor, an implementation of the `__str__` method that converts a `Rational` object to a string, and the code for the methods that implement the operations.

FIGURE 10-6 Implementation of the `Rational` class

```
# File: rational.py

"""This module defines a class for representing rational numbers."""

import math

class Rational:

# Implementation note
# --------------------
# The Rational class ensures that every number has a unique internal
# representation by guaranteeing that the following conditions hold:
#    1. The denominator must be greater than 0.
#    2. The number 0 is always represented as 0/1.
#    3. The fraction is always reduced to lowest terms.

    def __init__(self, num, den=1):
        """Creates a new Rational object from the integers num and den."""
        if den == 0:
            raise ValueError("Illegal denominator value")
        if num == 0:
            den = 1
        elif den < 0:
            den = -den
            num = -num
        g = math.gcd(abs(num), den)
        self._num = num // g
        self._den = den // g

    def __str__(self):
        """Returns the string representation of this Rational object."""
        if self._den == 1:
            return str(self._num)
        else:
            return str(self._num) + "/" + str(self._den)

    def add(self, r):
        """Creates a new Rational by adding r to this object."""
        return Rational(self._num * r._den + self._den * r._num,
                        self._den * r._den)

    def sub(self, r):
        """Creates a new Rational by subtracting r from this object."""
        return Rational(self._num * r._den - self._den * r._num,
                        self._den * r._den)

    def mul(self, r):
        """Creates a new Rational by multiplying this object by r."""
        return Rational(self._num * r._num, self._den * r._den)

    def div(self, r):
        """Creates a new Rational by dividing this object by r."""
        return Rational(self._num * r._den, self._den * r._num)
```

The code for the arithmetic operators follows directly from the mathematical definition. The implementation of the add method, for example, looks like this:

```
def add(self, r):
    return Rational(self._num * r._den + self._den * r._num,
                    self._den * r._den)
```

The definition of the add method creates a new Rational object by calling the constructor with the values required by the addition formula:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

In this method, the values a and b refer to the numerator and denominator of the current Rational object, which are available in the attributes _num and _den inside the self object. The values c and d refer to the corresponding components of the Rational object passed as the variable r.

The return statement in the add method calls the Rational constructor with the computed values of the numerator and denominator for the result. Calling the constructor ensures that the result is properly reduced to lowest terms and meets the other requirements maintained inside each Rational object.

10.4 Operator overloading

In the implementation of the Rational class shown in Figure 10-6, the arithmetic operators are implemented as the methods add, sub, mul, and div. This design decision means that you must invoke these methods using the receiver syntax. Thus, if you want to set the variable sum to the sum of the rational values stored in the variables a, b, and c, you would need to use the following statement:

```
sum = a.add(b).add(c)
```

Although this syntax makes sense to anyone familiar with Python's implementation of objects, it is certainly less expressive than the statement you would use to add three numbers, which looks like this:

```
sum = a + b + c
```

Unlike most object-oriented languages, Python makes it possible to define the Rational class so that this more natural syntax has the desired effect. To do so, you need to define a method for each operator that Python can then use to determine what that operator means in the context of the Rational class. This technique is called ***operator overloading.***

Each of the standard operators in Python is associated with a method whose name, like the name of the `__str__` method you have already seen, begins and ends with two underscores. The association between method names and operators appears in Figure 10-7. As an example, the first entry in the table shows that the method name `__add__` implements Python's + operator. Although addition is initially undefined for new classes, the implementation of that class can specify the meaning of addition by defining an `__add__` method. Moreover, since the `Rational` class already defines

FIGURE 10-7 **Standard methods used to support operator overloading**

Arithmetic operators

`__add__(self, rhs)`	Implements the + operator.
`__sub__(self, rhs)`	Implements the – operator.
`__mul__(self, rhs)`	Implements the * operator.
`__truediv__(self, rhs)`	Implements the / operator.
`__floordiv__(self, rhs)`	Implements the // operator.
`__mod__(self, rhs)`	Implements the % operator.
`__pow__(self, rhs)`	Implements the ** operator.
`__neg__(self)`	Implements the unary – operator.

Arithmetic operators applied in reverse

`__radd__(self, lhs)`	Called by the right operand to implement the == operator.
`__rsub__(self, lhs)`	Called by the right operand to implement the – operator.
`__rmul__(self, lhs)`	Called by the right operand to implement the * operator.
`__rtruediv__(self, lhs)`	Called by the right operand to implement the / operator.
`__rfloordiv__(self, lhs)`	Called by the right operand to implement the // operator.
`__rmod__(self, lhs)`	Called by the right operand to implement the % operator.
`__rpow__(self, lhs)`	Called by the right operand to implement the ** operator.

Relational operators

`__eq__(self, rhs)`	Implements the == operator.
`__ne__(self, rhs)`	Implements the != operator. Defaults to the inverse of `__eq__`.
`__lt__(self, rhs)`	Implements the < operator.
`__gt__(self, rhs)`	Implements the > operator.
`__le__(self, rhs)`	Implements the <= operator.
`__ge__(self, rhs)`	Implements the >= operator.

an add method that uses the traditional receiver syntax, you can enable addition for Rational objects by including the following method definition:

```
def __add__(self, rhs):
    return self.add(rhs)
```

When Python encounters an expression like a + b in which a and b are instances of the Rational class, it automatically translates that expression into the method call a.__add__(b). The implementation of the __add__ method corresponding to the operator then calls the regular add method in the Rational class to compute the answer.

You can, however, include additional code in the __add__ method so that it is possible to use the + operator to add a Rational value and an integer without forcing the client to convert the integer to a Rational explicitly. To do so, the easiest way is to have the definition of __add__ use the built-in type function to determine the type of the value that appears as the operand on the right side of the + operator. If this value is an integer, the code can first convert it to a Rational and then complete the addition. If this value is another Rational object, the code can apply the add method to the two rational numbers it has in hand. If this value is anything else, Python's conventions for operator overloading require the __add__ method to return the built-in Python value NotImplemented. This strategy leads to the following definition of the __add__ method:

```
def __add__(self, rhs):
    if type(rhs) is int:
        return self.add(Rational(rhs))
    elif type(rhs) is Rational:
        return self.add(rhs)
    else:
        return NotImplemented
```

This definition makes it possible, for example, to evaluate the expression

```
Rational(1, 2) + 1
```

which adds one to the Rational value 1/2, which produces the Rational value 3/2.

But what happens if the operands to + appear in the opposite order? If you ask Python to evaluate the expression

```
1 + Rational(1, 2)
```

it should get the same answer. Python, however, decides what to do based on the methods associated with the left operand, which in this case is the int value 1. The `__add__` method associated with the built-in type `int` has no knowledge of the `Rational` class and therefore is unable to perform the addition. Recognizing that it can't produce an answer, the `__add__` method for `int` returns `NotImplemented`, which Python takes as a signal to try a different strategy.

It is at this point that the methods in Figure 10-7 labeled "Arithmetic operators applied in reverse" come into play. For each of the arithmetic operators, Python defines a method that is used only if the left operand fails to implement the operation. These methods determine the result if the defining class appears on the right side of the operator. The `__radd__` method for `Rational` looks like this:

```
def __radd__(self, lhs):
    if type(lhs) is int:
        return Rational(lhs).add(self)
    elif type(lhs) is Rational:
        return lhs.add(self)
    else:
        return NotImplemented
```

Once this method is in place, the expression

```
1 + Rational(1, 2)
```

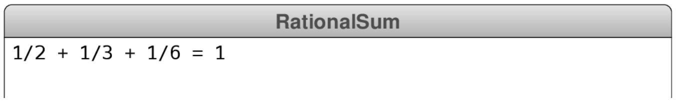
correctly produces the `Rational` value 3/2.

The definitions for the other arithmetic operators and the comparison operators look very much like the definitions for `__add__` and `__radd__`. The only differences are the method names. The code for the expanded `Rational` class is available on the web site for this book, but is too long to be useful as an example.

Before turning away from the topic of rational numbers, it is worth going back to the example at the beginning of the chapter to show that rational arithmetic is exact. Running the function

```
def rational_sum():
    a = Rational(1, 2)
    b = Rational(1, 3)
    c = Rational(1, 6)
    print("1/2 + 1/3 + 1/6 = " + str(a + b + c))
```

produces the following output:



```
1/2 + 1/3 + 1/6 = 1
```

10.5 Implementing a token scanner

The Point and Rational classes defined earlier in this chapter are both examples of immutable classes in which the internal attributes of the class never change after an object is created. While immutable classes are ideal for many data types—such as the built-in str class Python uses to represent strings—other types derive much of their utility from allowing clients to manipulate the internal data. For example, a list is more flexible than its immutable tuple counterpart precisely because it offers operations like insertion, deletion, and setting individual elements. The graphical programs you have built using the Portable Graphics Library depend on the fact that the objects in the GObject are mutable and that you can change their color, position, and size by invoking methods on those objects.

The rest of this chapter goes through the design of a class that allows clients to divide a string into substrings that form a logically connected unit that can be larger than a single character, such as a word or a number. In computer science, such units are called ***tokens,*** and a class that delivers tokens sequentially from a string is called a ***token scanner.*** A token scanner has to be implemented as a mutable class because its state changes as you use. When you call a method to read the next token, the token scanner has to update its internal state so the next call will return the following token.

In terms of its operation, a token scanner accomplishes much the same task as the to_pig_latin function in the PigLatin.py shown in Figure 7-3, which was responsible for dividing the input into words and then calling word_to_pig_latin to convert each word to its Pig Latin form. The goal in the next several sections is to reimplement word_to_pig_latin using a more general TokenScanner class that is flexible enough to use in a variety of applications.

What clients want from a token scanner

The best way to begin the design of the TokenScanner class is to look at the problem from the client perspective. Every client that wants to use a scanner starts with a source of tokens, which might be a string but might also be an input stream for applications that read data from files. In either case, what the client needs is some way to retrieve individual tokens from that source.

Although there are other strategies that offer the necessary functionality, the conventional design for a token scanner uses the following pseudocode form:

```
Initialize a token scanner object and set its input source.
while more tokens are available:
    Read and process the next token.
```

This pseudocode pattern immediately suggests what methods the TokenScanner class will have to support. From this example, you would expect TokenScanner to export the following methods:

- A TokenScanner constructor that creates a token scanner object from a source.
- A has_more_tokens method that tests if there are more tokens left to read.
- A next_token method that scans and returns the next token.

These methods define the operational structure of a token scanner and are largely independent of the specifics of the applications. Different applications, however, define tokens in all sorts of different ways, which means that the TokenScanner class must give the client some control over what types of tokens are recognized.

The need to recognize different types of tokens is easiest to illustrate by offering a few examples. As a starting point, it is instructive to revisit the problem of translating English into Pig Latin. If you rewrite the PigLatin.py program to use the token scanner, you can't ignore the spaces and punctuation marks, because those characters need to be part of the output. In the context of the Pig Latin problem, tokens fall into one of two categories:

1. A string of consecutive alphanumeric characters representing a word
2. A single-character string consisting of a space or punctuation mark

If you gave the token scanner the input

```
this is pig latin.
```

calling next_token repeatedly would return the following sequence of eight tokens:

`this` □ `is` □ `pig` □ `latin` `.`

Other applications, however, are likely to define tokens in different ways. The Python interpreter, for example, uses a token scanner to break programs into tokens that make sense in the programming context, including identifiers, constants, operators, and other symbols that define the syntactic structure of the language. For example, if you typed the line

```
print("The sum is " + str(sum))
```

into the Python interpreter, you would like its token scanner to deliver the following sequence of tokens:

`print` `(` `"The sum is "` `+` `str` `(` `sum` `)` `)`

There are several differences between these two application domains in the definition of a token. In the Pig Latin translator, anything that's not a sequence of alphanumeric characters is returned as a single-character token. In the example of the Python interpreter, the situation is more complicated. For one thing, the string constant `"The sum is "` has the correct meaning only if the token scanner treats it as a single entity. Perhaps less obviously, the compiler's token scanner ignores spaces in the input entirely, unless they appear inside string constants.

As you will learn if you go on to take a course on programming languages, it is possible to build a token scanner that allows the client to specify what constitutes a legal token, typically by supplying a precise set of rules. That design offers the greatest possible generality. Generality, however, sometimes comes at the expense of simplicity. If you force clients to specify the rules for token formation, they need to learn how to write those rules, which is similar in many respects to learning a new language. Worse still, the rules for token formation—particularly if you are trying to specify, for example, the rules that a compiler uses to recognize numbers—are complicated and difficult for clients to get right.

If your goal in the interface is to maximize simplicity, it is probably better to design the `TokenScanner` class so that clients can enable options that allow it to recognize the type of tokens used in specific application contexts. If all you want is a token scanner that collects consecutive alphanumeric characters into words, you use the `TokenScanner` class in its simplest possible configuration. If you instead want the `TokenScanner` to identify the units in a Python program, you enable options that tell the scanner, for example, to ignore whitespace characters, to treat quoted strings as single units, and to recognize particular combinations of punctuation marks as multicharacter operators.

The `tokenscanner` module

Because token scanners are so useful, the library files provided with this book include a `tokenscanner.py` module that offers considerable flexibility without sacrificing simplicity. The `tokenscanner.py` module exports a `TokenScanner` class that implements the methods shown in appear in Figure 10-8 on the next page. Many of the methods in the interface are used to enable options that change the default behavior of the scanner so that it serves the needs of a wider range of clients.

The `tokenscanner` module makes it easier to write a variety of applications, including several you have already seen in this book. You can, for example, use it to simplify the `PigLatin.py` program from Figure 7-3 by rewriting the function `to_pig_latin` as follows:

FIGURE 10-8 **Methods exported by the library implementation of the TokenScanner class**

Constructor

TokenScanner(*source*)	Initializes a scanner object. The source for the tokens is either a string or an open file object. If no source is provided, the client must call set_input before reading tokens.

Methods for reading tokens

has_more_tokens()	Returns True if there are more tokens to read.
next_token()	Returns the next token from this scanner. If called when there are no more tokens, next_token returns the empty string.
save_token(token)	Saves the specified token so that it will be returned on the next call to next_token.

Methods for controlling scanner options

ignore_whitespace()	Tells the scanner to ignore whitespace characters.
ignore_comments()	Tells the scanner to ignore comments.
scan_numbers()	Tells the scanner to recognize any legal Python number as a single token.
scan_strings()	Tells the scanner to return a string enclosed in quotation marks as a single token. The quotation marks (which may be either single or double quotes) are included in the result.
add_word_characters(*str*)	Adds the characters in str to the characters legal in a word.
add_operator(*op*)	Defines a new multicharacter operator. The scanner will return the longest defined operator or a single character.

Miscellaneous methods

set_input(*source*)	Sets the input source for this scanner to the specified source, which is either a string or an open file object. Any tokens remaining in the previous source are lost.
get_token_type(*token*)	Returns the token type, which must be one of the following: TokenScanner.EOF TokenScanner.NUMBER TokenScanner.SEPARATOR TokenScanner.STRING TokenScanner.WORD TokenScanner.OPERATOR
get_position()	Returns the current position of the scanner in the input stream.
get_string_value(*token*)	Removes the quotation marks from a string token and interprets any escape characters.
get_number_value(*token*)	Returns the numeric value of a token.
verify_token(*expected*)	Reads the next token and makes sure it matches *expected*.
is_word_character(*ch*)	Returns True if the character *ch* is valid in a word.
is_valid_identifier(*token*)	Returns True if token is a valid word token.

```
def to_pig_latin(line):
    scanner = TokenScanner(ine)
    result = ""
    while scanner.has_more_tokens():
        token = scanner.next_token()
        if token.isalpha():
            token = word_to_pig_latin(token)
        result += token
    return result
```

While the new implementation of to_pig_latin is shorter than the original, the real simplification is conceptual. The original code had to operate at the level of individual characters; the new version gets to work with complete words, because the TokenScanner class takes care of the low-level details.

Implementing the TokenScanner class

Particularly given the number of options it supports, the complete implementation of the TokenScanner class is too complicated to serve as an effective example. Figure 10-9, which extends over the next three pages, therefore presents a simplified version of the token scanner package that defines only the following methods:

- A constructor that accepts an optional string argument as the initial source
- The set_input method, which sets the scanner input to a string
- The next_token method, which returns the next token from the string
- The has_more_tokens method, which lets clients see if tokens are available
- The ignore_whitespace method, which tells the scanner to ignore spaces

As the largest example of a class definition you have seen so far, the code for TokenScanner in Figure 10-9 is worth studying in some detail. As you do so, it is important that you don't skip over the first page of the figure, which is composed almost entirely of comments. While it is true that the Python interpreter ignores these comments, you should keep in mind that comments are intended for human readers of the program—readers like you. Particularly when you are designing a class that you hope other programmers will use, you have a responsibility to give those programmers the information they need to use that class effectively. If potential clients are unable to figure out how to use a class, they will refrain from doing so. The comments on the first page of Figure 10-9 give the reader a tour of the facilities provided by the TokenScanner class along with examples of its use.

Another feature of the code that is worth your notice is that all private identifiers begin with an underscore. These private identifiers include not only the properties of the TokenScanner object defined in the constructor but also the private method _skip_whitespace, which is called internally by the implementation.

FIGURE 10-9 Simplified implementation of the **TokenScanner** class

```
# File: tokenscanner.py

"""This file implements a simple version of a token scanner class."""

# A token scanner is an abstract data type that divides a string into
# individual tokens, which are strings of consecutive characters that
# form logical units.  This simplified version recognizes two token types:
#
#   1. A string of consecutive letters and digits
#   2. A single character string
#
# To use this class, you must first create a TokenScanner instance by
# calling its constructor, passing in the string s to be scanned:
#
#     scanner = TokenScanner(s)
#
# You can also set the token source by calling
#
#     scanner.set_input(s)
#
# Once you have created the scanner, you can get the next token by calling
#
#    token = scanner.next_token()
#
# To determine whether any tokens remain to be read, you can either
# call the predicate method scanner.has_more_tokens() or check to see
# whether next_token returns the empty string.
#
# The following code fragment serves as a pattern for processing each
# token in the string stored in the variable source:
#
#     scanner = TokenScanner(source)
#     while scanner.has_more_tokens():
#         token = scanner.next_token()
#         . . . code to process the token . . .
#
# By default, the TokenScanner class treats whitespace characters
# as operators and returns them as single-character tokens.  You
# can set the token scanner to ignore whitespace characters by
# making the following call:
#
#     scanner.ignore_whitespace()

class TokenScanner:
    """This class implements a simple token scanner."""

    def __init__(self, source=""):
        """Creates a TokenScanner object that scans the specified string."""
        self.set_input(source)
        self._ignore_whitespace_flag = False
```

☞

FIGURE 10-9 **Simplified implementation of the `TokenScanner` class (continued)**

```
    def set_input(self, source):
        """Resets the input so that it comes from source."""
        self._source = source
        self._nch = len(source)
        self._cp = 0

    def next_token(self):
        """Returns the next token or the empty string at the end."""
        if self._ignore_whitespace_flag:
            self._skip_whitespace()
        if self._cp == self._nch:
            return ""
        token = self._source[self._cp]
        self._cp += 1
        if token.isalnum():
            while self._cp < self._nch and self._source[self._cp].isalnum():
                token += self._source[self._cp]
                self._cp += 1
        return token

    def has_more_tokens(self):
        """Returns True if there are more tokens to read."""
        if self._ignore_whitespace_flag:
            self._skip_whitespace()
        return self._cp < self._nch

    def ignore_whitespace(self):
        """Tells the scanner to ignore whitespace characters."""
        self._ignore_whitespace_flag = True

# Private methods

    def _skip_whitespace(self):
        """Skips over any whitespace characters before the next token."""
        while self._cp < self._nch and self._source[self._cp].isspace():
            self._cp += 1

# Interactive test program for the TokenScanner class

def interactive_token_scanner_test():
    finished = False
    while not finished:
        line = input("Enter a string: ")
        if line == "":
            finished = True
        else:
            scanner = TokenScanner(line)
            if line.startswith(" "):
                scanner.ignore_whitespace()
            while scanner.has_more_tokens():
                print('"' + scanner.next_token() + '"')
```

☞

FIGURE 10-9 Simplified implementation of the TokenScanner class (continued)

```python
# Unit test

def test_token_scanner():
    scanner = TokenScanner("abc  123")
    assert scanner.next_token() == "abc"
    assert scanner.next_token() == " "
    assert scanner.next_token() == " "
    assert scanner.next_token() == "123"
    assert scanner.next_token() == ""
    scanner = TokenScanner()
    scanner.ignore_whitespace()
    scanner.set_input("x = 2 * y")
    assert scanner.has_more_tokens()
    assert scanner.next_token() == "x"
    assert scanner.next_token() == "="
    assert scanner.next_token() == "2"
    assert scanner.next_token() == "*"
    assert scanner.next_token() == "y"
    assert not scanner.has_more_tokens()

# Startup code

if __name__ == "__main__":
    test_token_scanner()
```

Perhaps the most important feature to notice about the TokenScanner class is the way in which it encapsulates data values and methods into a single object. Like other data values, a TokenScanner object maintains information about its *state.* This information is stored in the values of the property variables _source, _nch, _cp, and _ignore_whitespace_flag. The client, however, need not be aware of those variables and indeed is warned by their names against making any direct reference to their values. What the client sees in a TokenScanner object are its methods, which define its behavior without revealing the implementation details.

Summary

This chapter introduces the concept of an *object,* which is a data structure that encapsulates state and behavior. Like arrays, objects combine multiple values into a single unit. In an array, individual elements are selected using a numeric index; in an object, individual attributes are selected by name.

The important points introduced in this chapter include:

- All Python objects are instances of a *class,* which provides a template for all objects of that conceptual type.

- Class definitions begin with a header line containing the `class` keyword and the name of class. The body of the class consists primarily of method definitions that define the behavior of the class.
- Like arrays, objects are treated as *references*, which means that their internal structure is not copied when the object is assigned or passed as a parameter.
- Given a Python object, you can select individual attributes using the dot operator, which is followed by the name of the attribute.
- Most classes define a method called `__init__`, which acts as the constructor for the class. Like all class methods, the first parameter to `__init__` is a reference to the object being created. That parameter should always be called `self`. The constructor can take additional parameters, which allows the client to pass other information to the constructor.
- Modern programming style discourages clients from manipulating the values of individual attributes directly. Classes instead provide mediated access in the form of methods. Methods that retrieve the value of an attribute are called *getters;* methods that set the value of an attribute are called *setters.*
- Classes typically define a method called `__str__`, which converts an instance of the class into a string that humans can recognize.
- To warn clients against looking too closely at parts of the implementation that were not intended to be seen by clients, Python uses the convention of adding an underscore at the beginning of names of any private attributes or methods.
- Designing new classes is as much an art as a science. This chapter outlines a general strategy to guide you in this process on page 353, but experience and practice are the best teachers.
- Python allows classes to define new implementations for the standard operators. This technique is called *operator overloading.* Figure 10-7 on page 358 provides a list of the methods that correspond to the built-in operators.
- A sequence of characters that has integrity as a unit is called a *token.* This chapter presents a simple implementation of a `TokenScanner` class that divides a string into its component tokens. The libraries included with this text include a `TokenScanner` class that offers clients more flexibility. The methods exported by the expanded `TokenScanner` class appear in Figure 8-1.

Review questions

1. What word does Python use for the individual components of an object?

2. True or false: If you pass a Python object as a parameter to a function, the function receives a copy of the object and therefore cannot change the components of the original.

3. What is the *dot operator* and how is it used?

4. What is the name of the special function used to implement a constructor?

5. True or false: Modern programming practice discourages direct access to the data attributes in an object.

6. What are *getters* and *setters?*

7. What is the purpose of the `__str__` method in a Python class?

8. What convention does Python adopt to discourage clients from referring directly to private attributes and methods?

9. What happens if you change the value of `DELTA` in the `YarnPattern.py` program from 113 to 104? Does the picture look as striking? Why or why not?

10. What is a *rational number?*

11. What restrictions does the constructor for the `Rational` class place on the values of the `num` and `den` variables?

12. What steps does the chapter propose as a useful approach to designing a class?

13. What is *operator overloading?*

14. What method would you define to change the definition of the `%` operator?

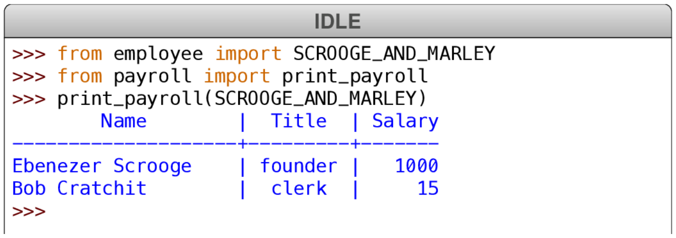
15. What is the difference between the `__add__` and `__radd__` methods?

16. In your own words, describe the function of a token scanner.

17. Given the `TokenScanner` class presented in this chapter, what statements would you use to list every token from a string stored in the variable `line`?

Exercises

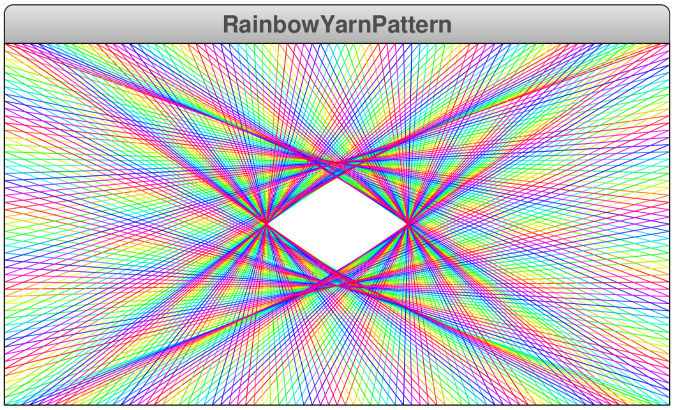
1. Write a function `print_payroll` that takes an array of employees, each of which is defined as a simple Python object, and prints on the console a list for each employee showing the name, title, and salary. For example, if `SCROOGE_AND_MARLEY` has been initialized as a two-element list containing the entries for Ebenezer Scrooge and Bob Cratchit shown in the chapter, your function should be able to reproduce the following IDLE session:



```
>>> from employee import SCROOGE_AND_MARLEY
>>> from payroll import print_payroll
>>> print_payroll(SCROOGE_AND_MARLEY)
        Name        |  Title   | Salary
--------------------+---------+-------
Ebenezer Scrooge    | founder |   1000
Bob Cratchit        |  clerk  |     15
>>>
```

2. Rewrite the `YarnPattern.py` program from Figure 10-3 so that it uses tuples to represent the points instead of `GPoint` objects.

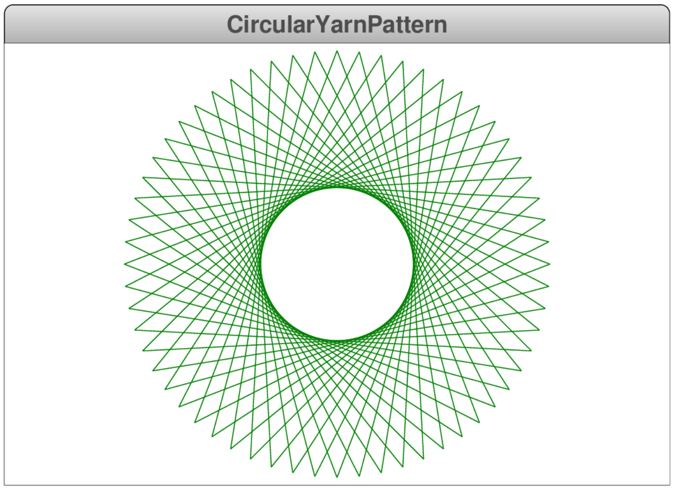
3. You can make more interesting yarn patterns by changing the color of each segment as you cycle through the pegs, like this:



Here, the yarn starts off red and then moves around the spectrum to orange, yellow, green, and so on. Cycling through the colors of the rainbow is tricky using the standard RGB color model but relatively easy if you instead use the ***HSV model,*** which defines colors in terms of their hue, saturation, and value. The hue value ranges from 0 and 1 that indicates a position on the color wheel that begins with red and then cycles through the spectrum colors.

This problem is made much easier by the fact that Python supports a `colorsys` library that exports a function `hsv2rgb` that converts between these color models. Look up the definition of `hsv2rgb` on the web and figure out what you need to do to produce this colored yarn pattern.

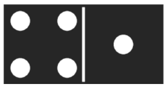
4. You can also produce interesting yarn patterns by arranging the pegs in a circle instead of a rectangle. Here, for example, is the result of arranging 60 pegs around the circumference of a circle and then using a `DELTA` value of 23 to string yarn around the pegs:



Modify the YarnPattern.py program to produce this figure. To save yourself the trouble of working out the trigonometric calculations, you can use the following function to return a tuple containing the coordinates of a point which is r units away from the point (x, y), moving in the direction specified by angle, which is measured in degrees, just as in the GArc class:

```
def polar_point(x, y, r, angle):
    dx = r * math.cos(math.radians(angle))
    dy = -r * math.sin(math.radians(angle))
    return (x + dx, y + dy)
```

5. The game of ***dominos*** is played using pieces that are usually black rectangles with some number of white dots on each side. For example, the domino



is called the 4-1 domino, with four dots on its left side and one on its right.

Define a simple Domino class that exports the following entries:

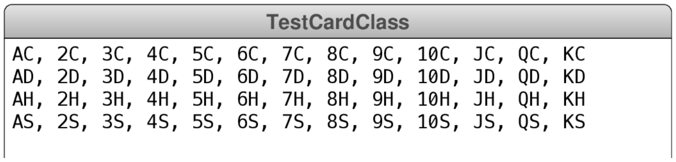
- A constructor that takes the number of dots on each side
- A __str__ method that creates a string representation of the domino
- Two getter methods named get_left_dots and get_right_dots

Test your implementation of the Domino class by writing a program that creates a full set of dominos from 0-0 to 6-6 and then displays those dominos on the console. A full set of dominos contains one copy of each possible domino in that range, disallowing duplicates that result from flipping a domino over. Thus, a domino set has a 4-1 domino but not a separate 1-4 domino.

6. Define a `Card` class suitable for representing a standard playing card, which is identified by two components: a ***rank*** and a ***suit.*** The rank is stored as an integer between 1 and 13 in which an ace is a 1, a jack is an 11, a queen is a 12, and a king is a 13. For the convenience of clients, the `Card` class exports constants named `Card.ACE`, `Card.JACK`, `Card.QUEEN`, and `Card.KING`. The suit is also represented as an integer between 0 and 3, which are exported as the constants `Card.CLUBS`, `Card.DIAMONDS`, `Card.HEARTS`, and `Card.SPADES`, respectively.

   Along with the constants, the `Card` class should export the following methods:

   - A constructor that takes either of two forms. If `Card` is called with two arguments, as in `Card(10, Card.DIAMONDS)`, it should create a card from those components. If `Card` is called with one argument, it should interpret the argument as a string composed of a rank (either a number or the first letter of a symbolic name) and the first letter of the suit, as in `"10D"` or `"QS"`.
   - A `__str__` method that converts the card to a string as described in the outline of the constructor. The card `Card(Card.QUEEN, Card.SPADES)`, for example, should have the string representation `"QS"`.
   - The getter methods `get_rank` and `get_suit`.
   - A main program that displays the string representation of every card, with each suit appearing on a separate line. The output of this program should look like this:



```
AC, 2C, 3C, 4C, 5C, 6C, 7C, 8C, 9C, 10C, JC, QC, KC
AD, 2D, 3D, 4D, 5D, 6D, 7D, 8D, 9D, 10D, JD, QD, KD
AH, 2H, 3H, 4H, 5H, 6H, 7H, 8H, 9H, 10H, JH, QH, KH
AS, 2S, 3S, 4S, 5S, 6S, 7S, 8S, 9S, 10S, JS, QS, KS
```

7. Write a function `midpoint` that takes two values of type `GPoint` and returns a new `GPoint` object whose coordinates define the midpoint of the line segment specified by the two parameters. For example, if the variables `upper_left` and `lower_right` are defined as

```
upper_left = GPoint(0, 0)
lower_right = GPoint(GWINDOW_WIDTH, GWINDOW_HEIGHT)
```

   calling `midpoint(upper_left, lower_right)` should return a point whose coordinates mark the center of the window.

8. Design and implement a `Date` class that exports the following resources:

   - Constants for the names of the months, so that clients can use the constant `Date.DECEMBER` instead of writing the number 12.

- A constructor that takes parameters named year, month, and day, and then uses those values to initialize the internal attributes of the Date object. For example, the statement

  ```
  moon_landing = Date(1969, Date.JULY, 20)
  ```

  should initialize moon_landing so that it represents July 20, 1969. The constructor should also check that the date is valid and raise a ValueError exception if any value is out of range. Note that making this check means that the module needs to know how many days are in each month.
- The getter methods get_day, get_month, and get_year.
- A __str__ method that returns the date in the form *dd–mmm–yyyy,* where *dd* is a one- or two-digit date, *mmm* is the three-letter English abbreviation for the month, and *yyyy* is the four-digit year. Thus, the string version of moon_landing is "20–Jul–1969".

9. Extend the Rational class by implementing overloaded versions of the relational operators ==, !=, <, <=, >, and >=. The names of these overloaded methods are listed in Figure 10-7 on page 358.

10. Write a program that uses the TokenScanner class to display the longest word that appears in a file chosen by the user. A word should be defined as any consecutive string of letters and digits, as in the TokenScanner class.

11. For certain applications, it is useful to be able to generate a series of names that form a sequential pattern. For example, if you wanted to number figures in a paper, having some mechanism to return the sequence of strings "Figure 1", "Figure 2", "Figure 3", and so on, would be very handy. You might also need to label points in a geometric diagram, in which case you would want a similar but independent set of labels for points such as "P0", "P1", "P2", and so forth.

    If you think about this problem more generally, what clients would like is a LabelGenerator class that allows them to specify a prefix string ("Figure " or "P" for the examples in the preceding paragraph) coupled with an integer used as a sequence number. To initialize a new generator, the client provides the prefix string and the initial index as arguments to the LabelGenerator constructor. Once the generator has been created, the client can return new labels in the sequence by calling next_label on the LabelGenerator object.

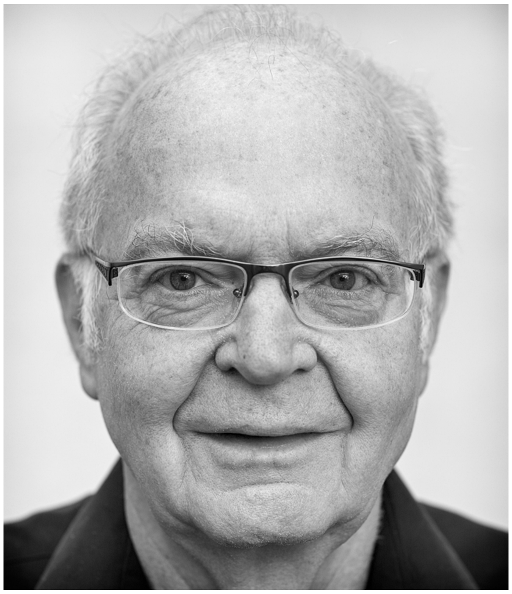
    Design and implement the LabelGenerator class along with a suitable program to test your implementation.

CHAPTER 11

Dictionaries and Sets

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

—Donald Knuth, Turing Award lecture, 1974



Donald E. Knuth

Donald Knuth got his introduction to computing during his undergraduate years at the Case Institute of Technology (now part of Case Western Reserve University) when he worked with the IBM 650 mainframe. He received his Ph.D. in mathematics from the California Institute of Technology in 1963 and in 1968 joined the computer science faculty at Stanford University. Knuth is best known for writing an extraordinarily comprehensive series of books entitled *The Art of Computer Programming,* which focuses considerable attention on the topics in this chapter. Throughout his career, Knuth has sought to weave the notions of aesthetics and elegance into the practice of computing. When he became convinced that conventional typesetting was unable to produce books that would be beautiful as well as comprehensive, Knuth implemented the typesetting language TEX, which remains in widespread use today. Professor Knuth received the ACM Turing Award in 1974 for his many contributions to computer science.

Python defines several built-in types beyond those you have already seen. Of these, the types covered in this chapter—dictionaries and sets—turn out to be especially valuable as tools for writing programs. The primary goal of this chapter is to give you a sense of when and how to use these structures as a client. In addition, the chapter explores a strategy called *hashing* makes it possible to look up dictionary entries in constant time.

11.1 Dictionaries

Chapter 8 introduced the concept of a *lexicon,* which was a list of words without associated definitions. While a lexicon is exactly what you need to write a spelling checker or for playing games like Scrabble, some applications will require you to associate each word with a definition. Providing those definitions turns a lexicon into a ***dictionary,*** which is a data structure in which relatively small identifying tags—the words in a physical dictionary, for example—are linked to additional information, often larger or more complex, such as a dictionary definition. In computer science, the identifying tag is called a ***key*** and the associated data structure is called the ***value*** for that key. Although computer scientists often use the term *map* to describe the general concept of a data structure that implements the key-to-value association, Python uses the term *dictionary.* This terminology is also reflected in the name of the Python type used to implement a dictionary, which is the built-in type `dict`.

Symbol tables

Dictionaries have many applications in programming. As an example, the Python interpreter needs to assign values to variables, which can then be identified by name. Python uses dictionaries—which are usually called ***symbol tables*** in the context of a programming language—to maintain the association between the name of a variable and its corresponding value. Python keeps track of several symbol tables simultaneously and looks through them in a specified order, looking first in the symbol table associated with the current function, then in the symbol tables associated with each of the calling functions, then in the symbol table for the global variables for the current module, and finally in the symbol table of built-in functions shared across all modules.

Because you already have a mental model of how variables work in the context of a programming language, symbol tables provide a good model for illustrating the operation of a dictionary. In Python, you create an empty dictionary by writing a matched set of curly braces with nothing inside them. Thus, you can initialize the variable `symtab` to an empty dictionary like this:

```
symtab = { }
```

This line sets up an empty dictionary that can be diagrammed as follows:

```
symtab
┌──────────────────┐
│                  │
│                  │
└──────────────────┘
```

Once you have created a dictionary, you can add a new definition by writing an assignment statement with the following general form:

dictionary[*key*] = *value*

For example, making the assignment

```
symtab["pi"] = 3.14159
```

adds a new association between the key `"pi"` and the value 3.14159, as follows:

```
symtab
┌──────────────────┐
│ pi → 3.14159     │
│                  │
└──────────────────┘
```

Similarly, executing the statement

```
symtab["e"] = 2.71828
```

adds a new association between the key `"e"` and the value 2.71828, like this:

```
symtab
┌──────────────────┐
│ pi → 3.14159     │
│ e  → 2.71828     │
└──────────────────┘
```

Once you have assigned a value to a key in a dictionary, you can retrieve that value by using the key as if it were an array index. The expression `symtab["e"]` has the value 2.71828, and the expression `symtab["pi"]` has the value 3.14159.

Although it hardly makes sense in the case of mathematical constants, you can change the values associated with keys in the dictionary using a new assignment statement. You could, for example, reset the value associated with `"pi"` (as an 1897 bill before the Indiana State General Assembly sought to do) by calling

```
symtab["pi"] = 3.0
```

which would lead to the following state:

```
symtab
┌──────────────────┐
│ pi → 3.0         │
│ e  → 2.71828     │
└──────────────────┘
```

As these examples illustrate, dictionaries act in much the same way that arrays do. The only difference is that the index value is no longer restricted to being an integer. Any Python type can serve as keys in a dictionary as long as two conditions are met: (1) the type must be immutable and (2) the type must implement the `hash` function described in the section on "Hashing" later in this chapter.

The program in Figure 11-1 simulates a tiny bit of the IDLE interpreter by reading assignment statements in the form

var = *value*

and requests to display the value of a variable in the form

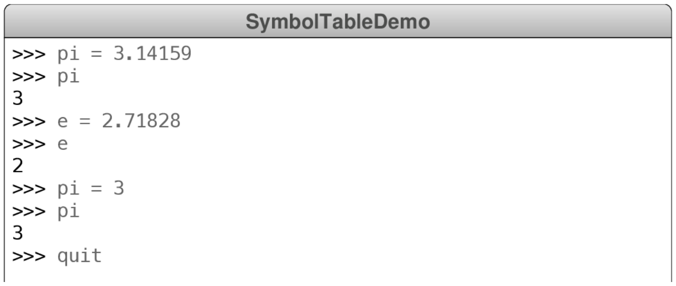
var

FIGURE 11-1 Program to illustrate the operation of a symbol table

```python
# File: SymbolTableDemo.py

"""
This program simulates a tiny part of the IDLE interpreter by modeling
the processes of assigning values to global variables and then looking
up those values by name.
"""

from tokenscanner import TokenScanner

def symbol_table_demo():
    symtab = { }
    scanner = TokenScanner()
    scanner.ignore_whitespace()
    finished = False
    while not finished:
        line = input(">>> ")
        scanner.set_input(line)
        name = scanner.next_token()
        if name == "quit":
            finished = True
        elif name == "list":
            for key, value in sorted(symtab.items()):
                print(f"{key}={value}")
        elif name != "":
            if scanner.has_more_tokens():
                if scanner.next_token() == "=":
                    symtab[name] = scanner.next_token()
                else:
                    print("Missing = in assignment")
            else:
                print(symtab.get(name))
```

The program treats *value* simply as a string and performs no calculations at all. It does, however, illustrate the general process of assignment to a global variable in which setting a new value overwrites any previous definition.

A sample run of the `SymbolTableDemo.py` program might look like this:



SymbolTableDemo

```
>>> pi = 3.14159
>>> pi
3
>>> e = 2.71828
>>> e
2
>>> pi = 3
>>> pi
3
>>> quit
```

Creating lookup tables

As it does for lists and tuples, Python includes a special syntactic form that creates a dictionary by listing its keys and values explicitly as part of the program. All you need to do is enclose a list of the desired definitions inside curly braces, where each definition is a ***key-value pair*** in the following form:

key: *value*

For example, the following statement defines a dictionary that defines an association between the name of a month and its conventional numeric value:

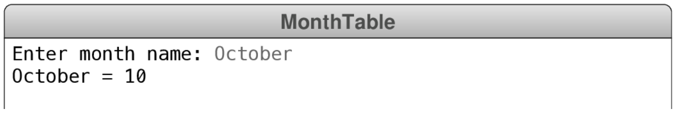
```
MONTH_TABLE = {
    "January": 1, "February": 2, "March": 3,
    "April": 4, "May": 5, "June": 6,
    "July": 7, "August": 8, "September": 9,
    "October": 10, "November": 11, "December": 12
}
```

Dictionaries of this sort that implement a mapping between a predetermined list of keys and their associated values is often called a ***lookup table.***

You can use the definition of `MONTH_TABLE` to convert the name of a month into its numeric form by using the month name as an index. Thus, the expression `MONTH_TABLE["June"]` has the value 6. More importantly, you can use this dictionary to convert the name of a month entered by the user to its numeric value using the following statements:

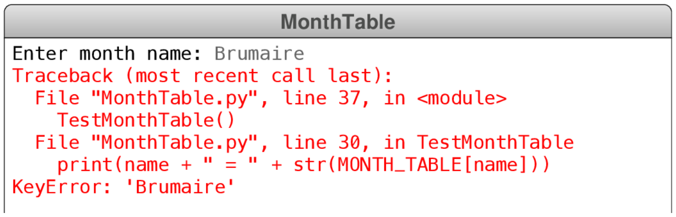
```
name = input("Enter month name: ")
print(f"{name} = {MONTH_TABLE[name]}")
```

Running this code snippet might generate the following console session:



```
Enter month name: October
October = 10
```

If you try to select a definition from a dictionary using a key that doesn't exist, Python raises a KeyError exception. For example, if you ran the same lines of code and entered one of the new month names invented (and soon abandoned) during the French Revolution, you might see the following session:



```
Enter month name: Brumaire
Traceback (most recent call last):
  File "MonthTable.py", line 37, in <module>
    TestMonthTable()
  File "MonthTable.py", line 30, in TestMonthTable
    print(name + " = " + str(MONTH_TABLE[name]))
KeyError: 'Brumaire'
```

Although you can use a try statement to catch this error, it is generally easier to use Python's in operator to check whether a key is defined, as in the following code:

```
name = input("Enter month name: ")
if name in MONTH_TABLE:
    print(f"{name} = {MONTH_TABLE[name]}")
else:
    print(f"{name} is not a valid month name")
```

Reading a dictionary from a data file

If you fly at all frequently, you quickly learn that every airport in the world has a three-letter code assigned by the International Air Transport Association (IATA). For example, the John F. Kennedy airport in New York City is assigned the three-letter code JFK. Other codes, however, are considerably harder to recognize. Most web-based travel systems offer some means of looking up these codes as a service to their customers.

A simple way to implement this facility is to create a dictionary whose keys are the airport codes and whose values are the city names. If you can create such a dictionary, all you need to do to find the city corresponding to the three-letter airport code is use the three-letter code as an index. There are, however, more than 2000 assigned airport codes, and the list of codes changes over time as new airports open and old airports close. For these reasons, it doesn't make sense to define a lookup table like the MONTH_NAMES constant from the preceding section. A much better strategy is to initialize the dictionary using a list of airport codes stored in a data file that can be easily updated whenever changes occur.

Suppose, for example, that the IATA organization maintains a downloadable file called `AirportCodes.txt` containing one line of data for each airport. Those lines all start with a three-letter code, which is followed immediately by a colon and the name of the city and country in which that airport is located. If the entries are sorted in descending order by passenger traffic as compiled by Airports Council International in 2017, the file would begin with the lines in Figure 11-2.

The first step in writing a program that allows users to look up the location of an airport from its three-letter code is to read the data from `AirportCodes.txt` into a dictionary. The problem of reading key-value pairs from a data file, however, is more general than the airport application and is worth making into a module of its own, as shown in Figure 11-3 at the top of the next page. Clients can use the `dictfile` module to read in a file in which keys and values appear on the same line. For the airport application, the client might call

```
airports = read_dictionary("AirportCodes.txt")
```

For a different application, the client would call `read_dictionary` on a different data file. The optional `separator` parameter makes it possible to use a character other than a colon to mark the division between the key and the value.

FIGURE 11-2 Beginning of a data file containing airport codes and locations

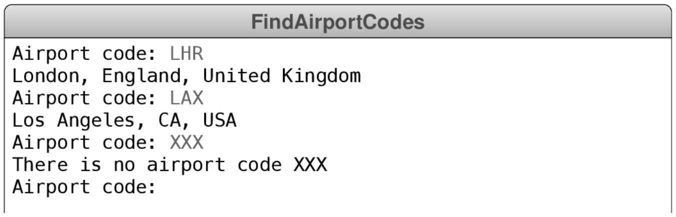
AirportCodes.txt

```
ATL:Atlanta, GA, USA
PEK:Beijing, China
DXB:Dubai, United Arab Emirates
LAX:Los Angeles, CA, USA
HND:Tokyo, Japan
ORD:Chicago, IL, USA
LHR:London, England, United Kingdom
HKG:Hong Kong, Hong Kong
PVG:Shanghai, China
CDG:Paris, France
DFW:Dallas/Ft Worth, TX, USA
AMS:Amsterdam, Netherlands
FRA:Frankfurt, Germany
IST:Istanbul, Turkey
CAN:Guangzhou, China
   .
   .
   .
```

FIGURE 11-3 **General-purpose module to read a dictionary from a data file**

```python
# File: dictfile.py

"""
This module contains code to read a dictionary from a data file.
The data file contains key-value pairs listed on separate lines.
"""

def read_dictionary(filename, separator=":"):
    """
    Creates a dictionary by reading key-value pairs from the
    specified file in which the division between the key and
    value is marked by the specified separator, which defaults
    to a colon.  The function discards any leading and trailing
    whitespace from both the key and the value.
    """
    dictionary = { }
    with open(filename) as f:
        for line in f:
            index = line.find(separator)
            key = line[:index].strip()
            value = line[index + len(separator):].strip()
            dictionary[key] = value
    return dictionary
```

The `FindAirportCodes.py` program in Figure 11-4 at the top of the next page reads three-letter codes from the user and displays the corresponding city name, as shown in the following sample run:



```
Airport code: LHR
London, England, United Kingdom
Airport code: LAX
Los Angeles, CA, USA
Airport code: XXX
There is no airport code XXX
Airport code:
```

Iterating through keys in a dictionary

In some applications, it is useful to be able to iterate through all the keys in a dictionary. For example, you can list the airports serving a particular country or city by going through the keys in the dictionary and listing every entry for which the value contains the desired country or city name. To make such applications possible, Python dictionaries support iteration using the following `for` loop pattern:

```
for variable in dictionary:
    . . . body of the loop . . .
```

FIGURE 11-4 **Program to translate airport codes to city names**

```
# File: FindAirportCodes.py

"""
This program looks up three-letter airport codes in a dictionary
read in from a file.
"""

from dictfile import read_dictionary

# Main program

def find_airport_codes():
    airports = read_dictionary("AirportCodes.txt")
    finished = False
    while not finished:
        code = input("Enter airport code: ")
        if code == "":
            finished = True
        else:
            if code in airports:
                print(airports[code])
            else:
                print(f"There is no airport code {code}")

# Startup code

if __name__ == "__main__":
    find_airport_codes()
```

This version of the pattern iterates through all the keys in *dictionary* so that, in each cycle, *variable* is assigned to the next key.

In versions of Python since 3.6, the `for` loop processes the keys in the same order in which they were entered into the dictionary. In earlier versions of Python—and in the similar constructs used in other programming languages—the order in which the elements are processed is unpredictable. If you want your program to run in as many versions of Python as possible or you think that it might be at some point translated into a different programming language, it is wise not to depend on the order in which the `for` loop processes the keys.

In many cases, you need both the key and the corresponding value in each cycle of the loop. One approach to iterating through the keys and values together is to look up the value on each cycle of the loop as shown in the following example:

```
for key in dict:
    value = dict[key]
    . . . rest of the loop body, which has access to both the key and the value . . .
```

While this pattern has the desired effect, it requires an extra step to look up the value associated with the key even though that value was presumably accessible as Python cycled through the keys. You can eliminate this step by using the `for` loop to iterate instead over all the key-value pairs like this:

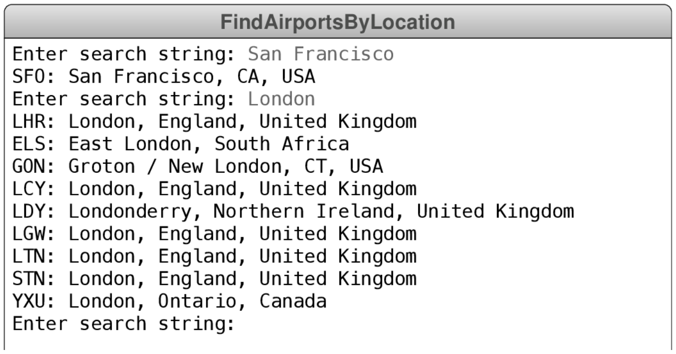
```
for key, value in dict.items():
    . . . the loop body, which has access to both the key and the value . . .
```

The `items` method returns an iterable value that is conceptually a list of pairs, each of which is a tuple containing a key and its corresponding value. The `for` loop then uses destructuring assignment to split the tuple into its two components, assigning the key to the variable `key` and the value to the variable `value`.

The `FindAirportsByLocation.py` program in Figure 11-5 uses this iteration pattern to implement a console-based application to find the airports serving a particular location. On each cycle, the program checks to see whether the user-supplied search string appears in the location stored as the value corresponding to each key. If so, the application prints that entry on the console. A sample run of this application might look like this:

FIGURE 11-5 Program to list the airport codes serving a particular location

```python
# File: FindAirportsByLocation.py

"""This program lists all the airports in a specified location."""

from dictfile import read_dictionary

def find_airports_by_location():
    airports = read_dictionary("AirportCodes.txt")
    finished = False
    while not finished:
        s = input("Enter search string: ")
        if s == "":
            finished = True
        else:
            for code, location in airports.items():
                if s in location:
                    print(f"{code}: {location}")

# Startup code

if __name__ == "__main__":
    find_airports_by_location()
```



```
Enter search string: San Francisco
SFO: San Francisco, CA, USA
Enter search string: London
LHR: London, England, United Kingdom
ELS: East London, South Africa
GON: Groton / New London, CT, USA
LCY: London, England, United Kingdom
LDY: Londonderry, Northern Ireland, United Kingdom
LGW: London, England, United Kingdom
LTN: London, England, United Kingdom
STN: London, England, United Kingdom
YXU: London, Ontario, Canada
Enter search string:
```

Dictionary methods

Dictionaries implement several additional methods that will prove useful in certain applications. The most important of these methods appear in Figure 11-6. The only one of these methods that may need additional explanation is the `get` method, which makes it possible to supply a default value for keys that are not found in the dictionary. The selection operation `dict[key]` raises a `KeyError` exception if the key is not found. In many applications, it is more useful to call `dict.get(key)`, which returns the constant `None` if the key is not found. The `get` method takes an optional second argument, which allows clients to specify some default value other than `None`.

11.2 Using dictionaries as records

The preceding section presents Python's `dict` class in order to emphasize the use of dictionaries as maps that associate a key with a value. That interpretation remains important in Python. Increasingly, however, Python programmers use dictionaries to

FIGURE 11-6 Common methods in Python's `dict` class

`len(`*dict*`)`	Returns the number of key-value pairs in the dictionary.
dict`.get(`*key*`)` *dict*`.get(`*key*`,` *value*`)`	Returns the value associated with *key* in the dictionary, or the specified default value if *key* is not found. In the first form of this call, the default value is `None`.
dict`.pop(`*key*`)`	Removes the key-value pair corresponding to *key* and returns the associated value. If the key is not found, `pop` raises a `KeyError` exception.
dict`.clear()`	Removes all key-value pairs from the dictionary, leaving it empty.
dict`.items()`	Returns an iterable object that cycles through successive tuples consisting of a key-value pair.

implement the idea of a record. After all, a record associates the name of a field with its value, which is pretty much just what a dictionary does.

For example, instead of using either a tuple or a class to represent a point, you could use a dictionary for that purpose. Under this interpretation, you could initialize the variable pt to the point (3, 4) by writing

```
pt = { "x": 3, "y": 4 }
```

After making this assignment, you could select the *x* and *y* components of the point by writing pt["x"] and pt["y"], respectively.

Although this model seems somewhat more verbose than the earlier approaches, it has the advantage of reducing the number of structures you need to consider when defining a new data structure. This advantage will become much more evident in Chapter 12, particularly in the context of using JavaScript Object Notation (JSON) to represent nested data structures.

11.3 Designing an efficient dictionary

The dictionary abstraction is used widely in programming applications, which gives the programmers responsible for implementing that abstraction an incentive to make dictionaries as efficient as possible. After all, if programmers can use clever algorithmic techniques to improve the performance of Python's dictionaries, every client that uses those dictionaries will benefit from the change.

Implementing dictionaries using lists

Before moving on to consider more efficient strategies, it is useful to start with a simple list-based implementation just to make sure that you understand what each of the required operations does. Since the goal is to implement the operations required for a dictionary, it is hardly appropriate to use Python's dict class in the solution. This section instead implements a Dictionary class that exports the necessary operations under the method names put and get. Calling put(*key*, *value*) adds a new value for the specified key, overwriting any previous value. Calling get(*key*) returns the value associated with the key if it exists. Like its counterpart in the built-in dict class, the get method takes an optional default value to use if the key is undefined.

Figure 11-7 on the next page shows an implementation of the Dictionary class in which the key-value pairs are stored in a list of tuples. That list of tuples is stored in an attribute of the Dictionary object called _bindings. The constructor sets the

FIGURE 11-7 Implementation of the `Dictionary` class using a list of tuples

```
# File: ListDictionary.py

"""
This module implements the Dictionary class, which simulates Python's
dict class.
"""

# Implementation notes
# --------------------
# This version of the Dictionary class stores the key-value pairs in a
# list of tuples.

class Dictionary:

    def __init__(self):
        """Initializes an empty dictionary."""
        self._bindings = [ ]

    def get(self, key, value=None):
        """Retrieves the binding for key, using value as a default."""
        for k,v in self._bindings:
            if k == key:
                return v
        return value

    def put(self, key, value):
        """Sets the binding for key to value."""
        for i in range(len(self._bindings)):
            k,_ = self._bindings[i]
            if k == key:
                self._bindings[i] = (key, value)
                return
        self._bindings.append((key, value))

    def __getitem__(self, key):
        """Implements square-bracket selection for this class."""
        value = self.get(key)
        if value is None:
            raise KeyError("No such key")
        return value

    def __setitem__(self, key, value):
        """Implements square-bracket assignment for this class."""
        self.put(key, value)

    def __iter__(self):
        """Returns an iterator for the keys in the order of insertion."""
        keys = [ ]
        for key,value in self._bindings:
            keys.append(key)
        return keys.__iter__()
```

private attribute `_bindings` to be an empty list. The `put` method searches through the elements of the list looking for a tuple that contains the requested key. If it finds one, the `put` method replaces that tuple with one that reflects the new binding. If not, the `put` method adds a new key-value pair to the end of the list. The `get` method operates similarly. If it finds the requested key while searching the list, it returns the value from that key-value pair. If it doesn't, `get` returns the default value.

The last three methods in Figure 11-7 illustrate new features that allow the `Dictionary` implementation to function more like Python's built-in `dict` class. The `__getitem__` method tells Python how to implement square-bracket selection. The `__setitem__` method has a symmetric interpretation that overrides Python's treatment of assignment to a selected object. Defining new implementations of these methods mean that you can retrieve items from a dictionary by writing

dict[*key*]

and set a new value by writing

dict[*key*] = *value*

The third method is in many ways more interesting. If you define an `__iter__` method in a class, Python considers instances of that class to be iterable objects, which means you can use them in a `for` statement. Thus, if you have stored an instance of the `Dictionary` class in a variable called `bindings`, you can iterate over its keys by writing

```
for key in bindings:
```

The `__iter__` method returns an object called an ***iterator,*** which is the data type Python uses to track the progress of stepping through an iteration. The details of iterators—and their more counterparts called ***generators***—are beyond the scope of an introductory computer science course. Even so, it is easy to create an iterator from any iterable object by calling the `__iter__` method for that object. The implementation of the `__iter__` method in the list-based `Dictionary` class creates a list of the keys and then result of calling the `__iter__` method on that list.

Improving the running time

In the implementation shown in Figure 11-7, both `put` and `get` run in $O(N)$ time. If you can keep the keys in the list in some kind of sorted order, you can reduce the running time of the `get` method to $O(\log N)$ by using binary search to find a matching key. Unfortunately, there is no obvious way to apply that same optimization to the `put` method. Although it is possible to check whether a key already exists in the dictionary—and even to determine exactly where a new key needs to be added—in

O(log *N*) time, inserting the new key-value pair at that position requires shifting every subsequent entry forward. Thus, put requires *O*(*N*) time, even in a sorted list.

To get some insight into how you might improve the performance of looking up words in a Python dictionary, it may help to think more concretely about how you might look up a word in a physical dictionary that is printed on paper instead of being stored in electronic form. The strategies that humans use to look up a dictionary entry don't look anything like the implementation in the preceding section. The algorithmic strategy embodied in that implementation is to check each successive word in the dictionary to see if it matches the one you're looking for. You start with the first entry, go on to the second, and then the third, until you find the word or determine that it is not in the dictionary at all.

No one, of course, would use this strategy for a dictionary of any significant size. But it is also unlikely that you would apply the *O*(log *N*) binary search algorithm, which corresponds to opening the dictionary exactly at the middle, deciding whether the word you're searching for appears in the first or second half, and then repeatedly applying this algorithm to smaller and smaller parts of the dictionary. You would instead try to anticipate more accurately where in the dictionary you should start looking for a word. If, for example, the word you're looking for starts with the letter *A,* you will start looking near the beginning of the alphabet. By contrast, if the word you're looking for begins with the letter *Z,* you would start looking closer to the end.

Printed dictionaries often try to help you with this process by including cutaway tabs along the side, each of which is labeled with the starting letter of words in that section. If you are lucky enough to have this kind of dictionary, you would look for words starting with *A* in the section marked with the *A* tab and for words starting with *Z* in the section marked with the *Z* tab. These tabs ensure that your search begins in the right section, which reduces the number of words you need to check.

To get a sense of how the metaphor of dictionary tabs might help in the design of a Python-based implementation, it is useful to work with a smaller example than the one using airport codes presented earlier in the chapter. In 1963, the United States Postal Service introduced a set of two-letter codes for the individual states, districts, and territories of the United States. The codes for the 50 states appear in Figure 11-8 at the top of the next page.

If you enter the key-value pairs from Figure 11-8 into the list-based dictionary presented in Figure 11-7, the list will have 50 elements, one for each state. Because the process of searching for a specific key requires looking at every element, the implementation will, in the worst case, have to look at every one of those 50 keys.

The most direct way to apply the idea of dictionary tabs to the search process is to divide the list of all the states into 26 shorter lists, one for each possible starting letter.

FIGURE 11-8 **USPS abbreviations for the 50 states**

AK	Alaska	HI	Hawaii	ME	Maine	NJ	New Jersey	SD	South Dakota
AL	Alabama	IA	Iowa	MI	Michigan	NM	New Mexico	TN	Tennessee
AR	Arkansas	ID	Idaho	MN	Minnesota	NV	Nevada	TX	Texas
AZ	Arizona	IL	Illinois	MO	Missouri	NY	New York	UT	Utah
CA	California	IN	Indiana	MS	Mississippi	OH	Ohio	VA	Virginia
CO	Colorado	KS	Kansas	MT	Montana	OK	Oklahoma	VT	Vermont
CT	Connecticut	KY	Kentucky	NC	North Carolina	OR	Oregon	WA	Washington
DE	Delaware	LA	Louisiana	ND	North Dakota	PA	Pennsylvania	WI	Wisconsin
FL	Florida	MA	Massachusetts	NE	Nebraska	RI	Rhode Island	WV	West Virginia
GA	Georgia	MD	Maryland	NH	New Hampshire	SC	South Carolina	WY	Wyoming

As in the physical dictionary, the states whose two-letter codes begin with *A* will show up in the *A* list, and so forth. At least in theory, this strategy should reduce the length of the individual lists—and therefore the running time—by a factor of 26.

Unfortunately, keys in a dictionary, like the first letters of English words, are not uniformly distributed. For example, many more English words begin with *C* than with *X*. The same is true for the state codes. Fully 64 percent of the state codes start with *A, I, M, N,* or *W,* while no state names begin with *B, E, J, Q, X, Y,* or *Z.* The fact that the first letters in the state name are poorly distributed across the alphabet means that some of the search lists will be relatively long and others will be empty. The divide-up-the-list-by-first-letter strategy offers some increase in efficiency, but nothing like the hoped-for factor of 26.

On the other hand, there is no reason that you have to use the first character of the key to divide up the keys in a dictionary. The first-character strategy is simply the closest analogue to what you do if you have a physical dictionary sitting in front of you. What you need is a strategy that divides the keys into groups in a way that does a better job of ensuring that the keys are distributed more evenly. That idea can be implemented in an elegant way using a technique called *hashing,* which is described in the following section.

Hashing

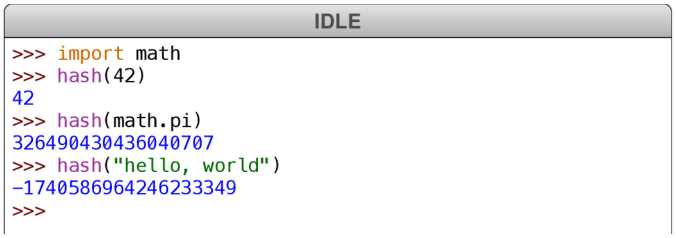
The best way to improve the efficiency of the dictionary implementation is to come up with a way of using the key to determine, at least fairly closely, where to look for the corresponding value. Choosing any obvious property of the key, such as its first character, runs into the problem that keys are not equally distributed with respect to that property.

Given that you are using a computer, however, there is no reason that the property you use to locate the key has to be something easy for a *human* to figure out. To maintain the efficiency of the implementation, the only thing that matters is whether the property is easy for a *computer* to determine. Since computers are better at computation than humans are, allowing for algorithmic computation opens a much wider range of possibilities.

The computational strategy called ***hashing*** operates as follows:

1. Select a function *f* that transforms a key into an integer value, which is called the ***hash code*** of that key. The function that computes the hash code is called, naturally enough, a ***hash function.*** An implementation of the dictionary abstraction that uses this strategy is conventionally called a ***hash table.***
2. Use the hash code for a key to determine the starting point as you search for a matching key in the table.

Python includes a built-in function called `hash` that returns the hash code for any immutable value. The following IDLE session shows the result of calling `hash` on the integer 42, the constant value `math.pi`, and the string `"hello, world"`, all of which are immutable:



```
>>> import math
>>> hash(42)
42
>>> hash(math.pi)
326490430436040707
>>> hash("hello, world")
-1740586964246233349
>>>
```

Although the hash code for 42 seems simple enough, the other hash codes listed in this example seem completely random. As it happens, the fact that these values seem random is not at all surprising. The implementation of the `hash` function in Python uses much the same techniques as the `random` library to ensure that the chance that two keys collide is as small as possible. To achieve that goal, the hash function tries to scatter the results over as wide a range of integers as possible.

If you are running a recent version of Python on modern 64-bit computer, the result of the `hash` function is a 64-bit integer, which means that its value lies somewhere between –9,223,372,036,854,775,808 and 9,223,372,036,854,775,807, which is an enormous range of values. The probability that two strings chosen at random produce the same hash code is 1 in 2^{64}, which means that one would never expect such an event to happen in a lifetime.

Unfortunately, the number of possible values of the `hash` function is so gigantic that there is no way to use the hash code itself as an index into a smaller list of values. No computer that exists now or in the foreseeable future could hold an array of that size. What you need to do is compress the hash codes into a narrower range in which each of these smaller values can serve as index into an array containing some fraction of the key-value pairs.

Although other representations are possible, a common strategy is to use the hash code to compute an index into an array of lists, where each list holds all the key-value pairs corresponding to that hash code. When you use this strategy to implement a hash table, the elements of the array containing the lists are traditionally called ***buckets.*** To find the key you're looking for, all you need to do is search through the list of key-value pairs in the bucket whose index is specified by the hash code.

As a general rule, the number of possible hash codes is considerably larger than the number of buckets. You can, however, convert an arbitrarily large hash code into a bucket number by computing the remainder of the absolute value of the hash code divided by the number of buckets. Thus, if the array of buckets is stored in the attribute `_buckets` of the dictionary object, you can compute the bucket number for a particular key like this:

```
bucket = abs(hash(key)) % len(self._buckets)
```

A bucket number represents an index into the `_buckets` array, each of whose elements is a list of key-value pairs. Colloquially, computer scientists say that a key ***hashes to a bucket*** if the hash function applied to the key returns that bucket number after applying the remainder operation. Thus, the common property that links all the keys in a single linked list is that they all hash to the same bucket. Having two or more different keys hash to the same bucket is called ***collision.***

The reason that hashing works is that the hash function always returns the same value for any particular key. If a key hashes to bucket #13 when you call `put` to enter it into the dictionary, that key will still hash to bucket #13 when you call `get` to find its value. Figure 11-9 shows the code for the `HashDictionary` module, which implements the `Dictionary` class using a hash table.

Tracing the hash table implementation

The easiest way to understand the implementation of the hash table in Figure 11-9 shows is to go through a simple example.

FIGURE 11-9 Implementation of the `Dictionary` class using a hash table

```
# File: HashDictionary.py

"""
This module implements the Dictionary class, which simulates Python's
dict class.  This version uses a hash table to store key-value pairs.
"""

class Dictionary:

    N_BUCKETS = 7

    def __init__(self):
        """Initializes an empty dictionary."""
        self._buckets = [ [ ] for i in range(Dictionary.N_BUCKETS) ]

    def get(self, key, value=None):
        """Returns the binding for key, using value as a default."""
        bucket = abs(hash(key)) % len(self._buckets)
        chain = self._buckets[bucket]
        for k,v in chain:
            if k == key:
                return v
        return value

    def put(self, key, value):
        """Sets the binding for key to value."""
        bucket = abs(hash(key)) % len(self._buckets)
        chain = self._buckets[bucket]
        for i in range(len(chain)):
            k,_ = chain[i]
            if k == key:
                chain[i] = (key, value)
                return
        chain.append((key, value))

    def __getitem__(self, key):
        """Implements square-bracket selection for this class."""
        value = self.get(key)
        if value is None:
            raise KeyError("No such key")
        return value

    def __setitem__(self, key, value):
        """Implements square-bracket assignment for this class."""
        self.put(key, value)

    def __iter__(self):
        """Returns an iterator for the keys in an unspecified order."""
        keys = [ ]
        for chain in self._buckets:
            for key,value in chain:
                keys.append(key)
        return keys.__iter__()
```

Suppose that you have written a program that needs to initialize a dictionary that maps the two-letter codes for each state into the corresponding state name. Your program would begin by calling the `Dictionary` constructor like this:

```
state_dictionary = Dictionary()
```

The constructor creates a list called `_buckets` with 16 elements, each of which is an empty list.

To add the first state in the list to the dictionary, your program would then execute the call

```
state_dictionary.put("AK", "Alaska")
```

If you look at the code for `put`, you will see that the first statement is

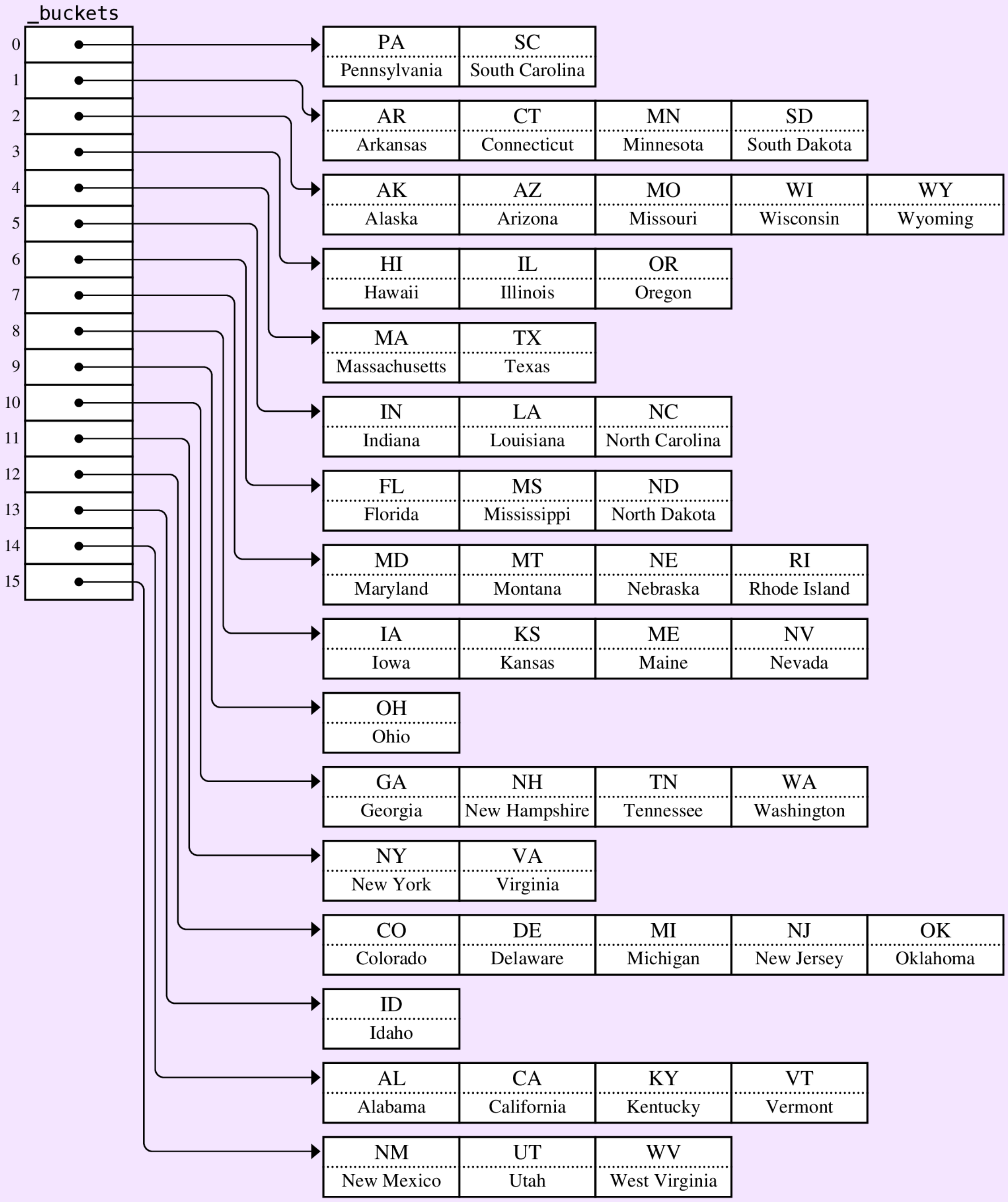
```
bucket = abs(hash(key)) % len(self._buckets)
```

which computes the bucket number. Calling `hash("AK")` in this example produces the 64-bit integer –5,249,979,066,121,302,514. While that number is difficult for people to comprehend, the computer has no more difficulty performing arithmetic on that number than on any other integer that fits inside a single memory location. The statement then divides the absolute value of the hash code by the number of buckets and then assigns the remainder to the local variable `bucket`. Trusting in the computer to carry out that division, it turns out that `"AK"` hashes to bucket #2. The remaining code in the `put` method goes through the key-value pairs in the list from bucket #2 looking for a matching key. The list is currently empty, so the code simply adds the tuple `("AK", "Alaska")` to the end of the list.

Since there are only 16 buckets in the array and 50 states whose codes need to be stored in the dictionary, it must be the case that the codes for some of the states collide. This fact is an application of what mathematicians call the ***pigeonhole principle,*** which simply says that if you have more pigeons than pigeonholes, you can't house all the pigeons without having at least two pigeons in some pigeonhole.

As it happens, the first collision in this example comes when you try to insert the entry for the state code `"AZ"`, which also hashes to bucket #2. The `put` method looks through the list of values that have already been added to bucket #2 to see if `"AZ"` is already there. Since it isn't, the `put` method adds a new entry to the list. If you carry out this process for all 50 states, you end up with the diagram shown in Figure 11-10.

FIGURE 11-10 Hash table containing the two-letter codes for the 50 states


_buckets
0 PA Pennsylvania, SC South Carolina
1 AR Arkansas, CT Connecticut, MN Minnesota, SD South Dakota
2 AK Alaska, AZ Arizona, MO Missouri, WI Wisconsin, WY Wyoming
3 HI Hawaii, IL Illinois, OR Oregon
4 MA Massachusetts, TX Texas
5 IN Indiana, LA Louisiana, NC North Carolina
6 FL Florida, MS Mississippi, ND North Dakota
7 MD Maryland, MT Montana, NE Nebraska, RI Rhode Island
8 IA Iowa, KS Kansas, ME Maine, NV Nevada
9 OH Ohio
10 GA Georgia, NH New Hampshire, TN Tennessee, WA Washington
11 NY New York, VA Virginia
12 CO Colorado, DE Delaware, MI Michigan, NJ New Jersey, OK Oklahoma
13 ID Idaho
14 AL Alabama, CA California, KY Kentucky, VT Vermont
15 NM New Mexico, UT Utah, WV West Virginia

The implementation of the hash table works because get and put use the same code to determine the bucket number. When you are looking up a key using get, you can rely on the fact that it must be in the bucket you calculate from the hash code if it is going to be anywhere in the table at all. After all, the only way that key could have found its way into the table is if a previous call to put added it to the list stored in that bucket. If the keys are the same, then get and put will calculate the same bucket number.

Adjusting the number of buckets

If you look at the distribution of keys in the hash table pictured in Figure 11-10, you will see that distribution is reasonably uniform. None of the buckets contain more than five elements and none of them happen to be empty. As you would expect, the lists in each buckets are of slightly different lengths because of the way random processes work. Even so, the effect of the hashing strategy is to reduce the time needed to find a key in the table by a factor roughly equal to the number of buckets.

In terms of computational complexity, however, it is not yet clear that the reduction in time will matter much as the size of the dictionary grows. Big-O notation allows you to throw away constant factors. If the number of buckets is always 16, all this most recent implementation has done is divide the average running time by 16, which means that the computational complexity of the put and get methods is still $O(N)$.

Although the details of the hash function are also important, the efficiency of a hash table depends on the number of buckets. If the number of buckets is small, collisions occur more frequently. In particular, if there are more entries in the hash table than buckets, collisions are inevitable. Collisions affect the efficiency of the hash table because put and get have to search through longer lists. As the hash table fills up, the number of collisions rises, which in turn reduces performance.

It is important to remember that the goal of using a hash table is to optimize the put and get methods so that they run in constant time, at least in the average case. Achieving this goal requires that the lists stored in each bucket remain short, which in turn implies that the number of buckets must always be large in comparison to the number of entries. Assuming that the hash function does a good job of distributing the keys evenly among the buckets, the average length of each bucket chain is given by the formula

$$\lambda = \frac{N_{\text{entries}}}{N_{\text{buckets}}}$$

For example, if the total number of entries in the table is three times the number of buckets, the average chain will contain three entries, which in turn means that three string comparisons will be required, on average, to find a key. This ratio, usually indicated by the Greek letter lambda (λ), is called the ***load factor*** of the hash table.

For good performance, you want to make sure that the value of λ remains small. Although the mathematical details are beyond the scope of this text, maintaining a load factor of 0.7 or less means that the average cost of looking up a key in a dictionary is $O(1)$. Smaller load factors imply that there will be lots of empty buckets in the hash table array, which wastes a certain amount of space. You can usually reduce the running time of a dictionary implemented as a hash table by being willing to consume more memory. Conversely, you can often save memory by being willing to accept slightly slower performance. Considerations of this sort come in many applications and are called a ***time-space tradeoff.***

Unless the hashing algorithm is engineered for a particular application in which the number of keys is known in advance, it is impossible to choose a fixed value for the number of buckets that works well for all clients. If a client keeps entering more and more entries into a hash table, the performance will eventually degrade. If you want to ensure good performance, the best approach is to allow the implementation to increase the number of buckets dynamically. For example, you can design the implementation so that it allocates a larger hash table if the load factor in the table ever reaches a certain threshold. Unfortunately, if you increase the number of buckets, the bucket numbers all change, which means that the code to expand the table must reenter every key from the old table into the new one. This process is called ***rehashing.*** Although rehashing can be time-consuming, it is performed infrequently and therefore has minimal impact on the overall running time of the application. You will have a chance to implement rehashing in exercise 9.

Hashing and computer security

The techniques involved in hashing play an important role in several applications involving computer security. The most common strategies used to prevent a malicious third party from making unauthorized changes to the content of a message is to include a ***digital signature*** as part of the message, which is essentially a hash code of its original contents. When digital signatures are combined with secure encryption technology, forging a message become extremely difficult.

Interestingly, hash tables have also been a source of vulnerability that allow hackers to overwhelm the capacity of a system by sending it time-consuming requests that prevent the system from responding to legitimate traffic. This type of intrusion is called a ***denial-of-service attack.*** Because hash tables are used in most implementations of web-based protocols, knowing how to slow those hash tables down becomes a useful tool in the hacker's arsenal. For example, if a hacker knows exactly how a hash function works, it is not difficult to send requests in which all the keys collide.

Python eliminates this problem by randomizing the hash function it uses for strings so that calling `hash` gives different values each time the interpreter is run. This strategy prevents a hacker from exploiting collisions to reduce server performance. Within any single run of the interpreter, however, the hash function will give consistent results, which means that the hashing strategy continues to function correctly.

11.4 Sets

The last built-in type covered in this chapter is the `set` class, which is an interesting object of study for a variety of reasons. From the perspective of the Python programmer, sets often provide just the right tool for applications in which you need to keep track of a collection of distinct objects. Beyond their practical value, however, sets provide a powerful mental model for thinking about programs, mostly because the properties of sets have been studied for so many years in the context of mathematics. If mathematicians have known for centuries that some theorem is true in the context of mathematical sets, adopting that theoretical model can often make it easier to design, implement, and debug a program. Finally, many intellectually exciting algorithms in computer science today use sets in their implementation. If you code those algorithms in a language that includes—as Python does—a powerful set abstraction, the translation from an abstract algorithmic description to a working program is a much more straightforward process.

Sets as a mathematical abstraction

In all likelihood, you have already encountered sets at some point in your study of mathematics. In general terms, it is easiest to think of a ***set*** as an unordered collection of distinct elements. For example, the set whose elements are the names of the primary colors of light looks like this:

```
{ "Red", "Green", "Blue" }
```

The individual elements appear in this order only because it is conventional. If the names were in a different order, it would still be the same set. A set never contains multiple copies of the same element.

The set of primary colors is a ***finite set*** because it contains a finite number of elements. In mathematics, there are also ***infinite sets,*** such as the set of all integers. In a computer system, sets are usually finite, even if they correspond to infinite sets in mathematics.

To understand the fundamental operations on sets, it is useful to have a few sets to use as a foundation. In keeping with mathematical convention, this text uses the following symbols to refer to the indicated sets:

$\varnothing$	The ***empty set,*** which contains no elements
$\mathbf{N}$	The set of ***natural numbers*** (0, 1, 2, 3, . . .)
$\mathbf{Z}$	The set of all integers
$\mathbf{R}$	The set of all real numbers

In mathematics, sets are most often written using a single uppercase letter. Sets whose membership is defined—like **N, Z,** and **R**—are denoted using boldface letters. Names that refer to some unspecified set are written using italic letters, such as S and T.

The fundamental property that defines a set is that of ***membership,*** which has the same intuitive meaning in mathematics that it does in English. Mathematicians express membership symbolically using the notation $x \in S$, which indicates that the value x is an element of the set S. For example, $17 \in \mathbf{N}$, $-4 \in \mathbf{Z}$, and $\pi \in \mathbf{R}$. Conversely, the notation $x \notin S$ indicates that x is *not* an element of S. For example, $-4 \notin \mathbf{N}$ because the set of natural numbers does not include the negative integers.

Mathematical set theory defines several operations on sets, of which the following are the most common:

- ***Union.*** The union of two sets is written as $A \cup B$ and consists of all elements belonging to the set A, the set B, or both.

  $\{1, 3, 5, 7, 9\} \cup \{2, 4, 6, 8\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  $\{1, 2, 4, 8\} \cup \{2, 3, 5, 7\} = \{1, 2, 3, 4, 5, 7, 8\}$
  $\{2, 3\} \cup \{1, 2, 3, 4\} = \{1, 2, 3, 4\}$

- ***Intersection.*** The intersection of two sets is written as $A \cap B$ and consists of the elements belonging to both A and B.

  $\{1, 3, 5, 7, 9\} \cap \{2, 4, 6, 8\} = \varnothing$
  $\{1, 2, 4, 8\} \cap \{2, 3, 5, 7\} = \{2\}$
  $\{2, 3\} \cap \{1, 2, 3, 4\} = \{2, 3\}$

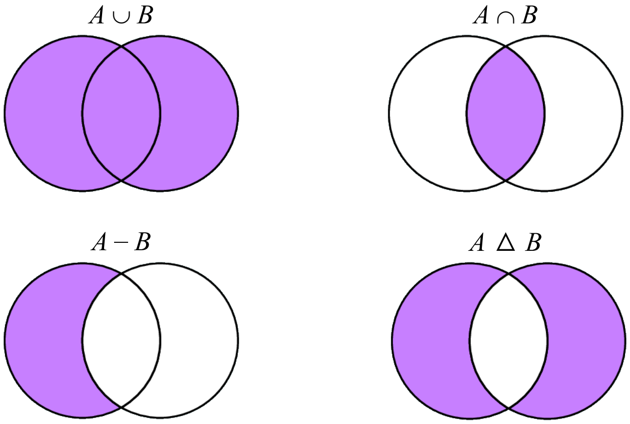
- ***Set difference.*** The difference of two sets is written as $A - B$ and consists of the elements belonging to A except for those that are also contained in B.

  $\{1, 3, 5, 7, 9\} - \{2, 4, 6, 8\} = \{1, 3, 5, 7, 9\}$
  $\{1, 2, 4, 8\} - \{2, 3, 5, 7\} = \{1, 4, 8\}$
  $\{2, 3\} - \{1, 2, 3, 4\} = \varnothing$

- ***Symmetric set difference.*** The symmetric difference of two sets is written as $A \bigtriangleup B$ and consists of the elements belonging to either A or B but not both.

  $\{1, 3, 5, 7, 9\} \bigtriangleup \{2, 4, 6, 8\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  $\{1, 2, 4, 8\} \bigtriangleup \{2, 3, 5, 7\} = \{1, 3, 4, 5, 7, 8\}$
  $\{2, 3\} \bigtriangleup \{1, 2, 3, 4\} = \{1, 4\}$

Set operations are often illustrated by drawing ***Venn diagrams,*** which are named for the British logician John Venn. In a Venn diagram, the individual sets are represented as geometric figures that overlap to indicate regions in which they share elements. For example, the results of the set operations union, intersection, set difference, and symmetric set difference are indicated by the shaded regions in the following Venn diagrams:



In addition to set-producing operations like union, intersection, set difference, and symmetric set difference, the mathematical theory of sets also defines several operations that determine whether a property holds between two sets. Operations that test a particular property are the mathematical equivalent of predicate methods and are usually called ***relations.*** The most important relations on sets are the following:

- ***Equality.*** The sets A and B are equal if they have the same elements. The equality relation for sets is indicated by the standard equal sign used to denote equality in other mathematical contexts. Thus, the notation $A = B$ indicates that the sets A and B contain the same elements.
- ***Subset.*** The subset relation is written as $A \subseteq B$ and is true if all the elements of A are also elements of B. For example, the set $\{2, 3, 5, 7\}$ is a subset of the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Similarly, the set **N** of natural numbers is a subset of the set **Z** of integers. From the definition, it is clear that every set is a subset of itself. Mathematicians use the notation $A \subset B$ to indicate that A is a ***proper subset*** of B, which means that the subset relation holds but that the sets are not equal.

One of the useful bits of knowledge you can derive from mathematical set theory is that the set operations are related to each other in various ways. These relationships are usually expressed as ***identities,*** which are rules indicating that two expressions are invariably equal. In this text, identities are written in the form

$$lhs \equiv rhs$$

which means that the set expressions *lhs* and *rhs* are equal by definition. The most common set identities are shown in Figure 11-11.

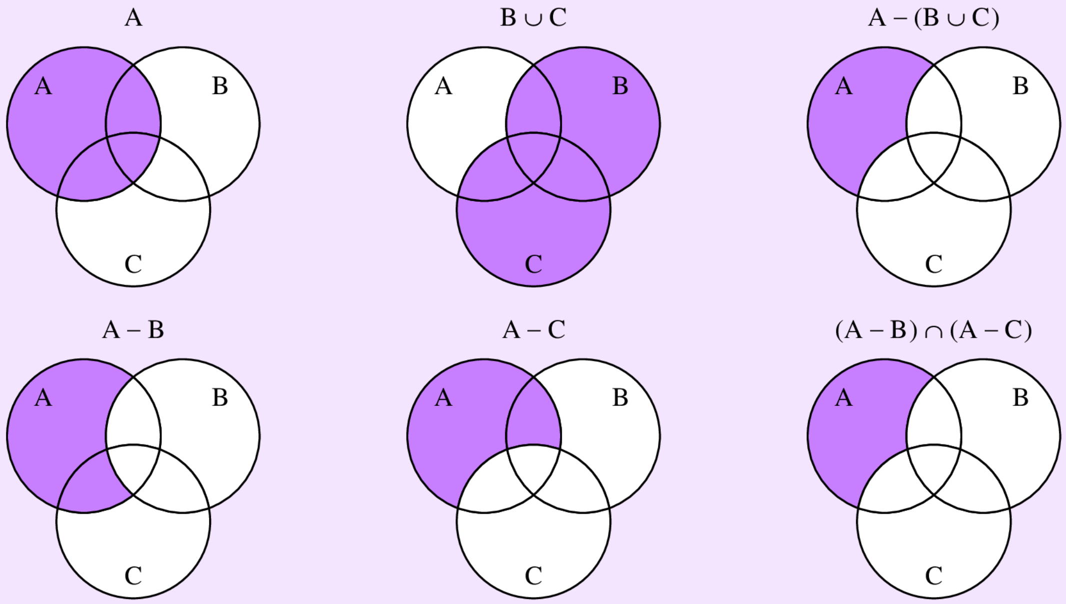
You can get a sense of how these identities work by drawing Venn diagrams to represent individual stages in the computation. Figure 11-12, for example, verifies the first of De Morgan's laws listed in Figure 11-11, which are named after the British mathematician Augustus De Morgan, who first formalized these identities. The shaded areas represent the value of each subexpression in the identity. The fact that the Venn diagrams along the right edge of Figure 11-12 have the same shaded region demonstrates that the set $A - (B \cup C)$ is the same as the set $(A - B) \cap (A - C)$.

Mathematical techniques are important to computer science for several reasons. For one thing, theoretical knowledge is useful in its own right because it deepens your understanding of the foundations of computing. Moreover, this type of theoretical knowledge often has direct application to programming practice. By relying on data structures whose mathematical properties are well established, you can use the theoretical underpinnings of those structures to your advantage. For example, if you write a program that uses sets, you may be able to simplify your code by applying one of the standard set identities from Figure 11-11. Choosing to use sets as a programming abstraction, as opposed to designing some less formal structure of your own, makes it easier for you to apply theory to practice.

FIGURE 11-11 Fundamental set identities

$S \cup S \equiv S$ $S \cap S \equiv S$	Idempotence
$A \cup (A \cap B) \equiv A$ $A \cap (A \cup B) \equiv A$	Absorption
$A \cup B \equiv B \cup A$ $A \cap B \equiv B \cap A$ $A \triangle B \equiv B \triangle A$	Commutative laws
$A \cup (B \cup C) \equiv (A \cup B) \cup C$ $A \cap (B \cap C) \equiv (A \cap B) \cap C$ $A \triangle (B \triangle C) \equiv (A \triangle B) \triangle C$	Associative laws
$A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$ $A \cap (B \triangle C) \equiv (A \cap B) \triangle (A \cap C)$	Distributive laws
A ▯ $(B \cap C) \equiv (A$ ▯ $B) \cup (A$ ▯ $C)$ A ▯ $(B \cup C) \equiv (A$ ▯ $B) \cap (A$ ▯ $C)$	De Morgan's laws

FIGURE 11-12 Illustration of the first of De Morgan's laws using Venn diagrams



Sets in Python

The beginning of the preceding section used the following notation to define a set consisting of the three primary colors of light:

```
{ "Red", "Green", "Blue" }
```

That expression is written in its conventional mathematical form, in which the elements of the set are enclosed in curly braces and separated by commas. Python uses precisely that syntax. You can, for example, assign that set to the variable `primaries` by writing

```
primaries = { "Red", "Green", "Blue" }
```

You can also create sets using ***set comprehensions,*** which are analogous to the list comprehensions introduced in Chapter 8. For example, the expression

```
{ i for i in range(10) }
```

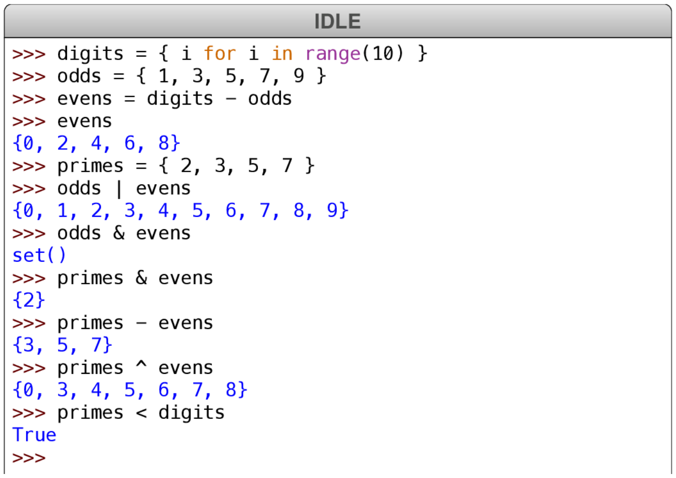
creates a set containing the integers from 0 to 9.

Python's set class includes built-in operators for all the mathematical operators on sets described in the preceding section. These operators are listed, along with several

FIGURE 11-13 Common operators and methods in Python's `set` class

`set()`	Creates an empty set. The expression `{ }` creates an empty dictionary.
`set(`*list*`)`	Creates a set from the elements of *list* or any iterable object.
`len(`*set*`)`	Returns the number of elements in a set.
set_1 `|` set_2	Returns the union of set_1 and set_2.
set_1 `&` set_2	Returns the intersection of set_1 and set_2.
set_1 `-` set_2	Returns the difference of set_1 and set_2.
set_1 `^` set_2	Returns the symmetric difference of set_1 and set_2.
set_1 `==` set_2	Returns `True` if set_1 and set_2 contain the same elements.
set_1 `<=` set_2	Returns `True` if set_1 is a subset of set_2.
set_1 `<` set_2	Returns `True` if set_1 is a proper subset of set_2.
element `in` *set*	Returns `True` if *element* is in the set.
set_1`.isdisjoint(`set_2`)`	Returns `True` if set_1 and set_2 contain no elements in common.
set`.clear()`	Removes all elements from the set, leaving it empty.
set`.copy()`	Creates and returns a shallow copy of the set.
set`.add(`*element*`)`	Adds the specified element to the set.
set`.remove(`*element*`)`	Removes the element from the set, raising a `ValueError` if it is missing.
set`.discard(`*element*`)`	Removes the element from the set, doing nothing if it is missing.

methods that apply to the `set` class in Figure 11-13. Several of the most common set operators are illustrated in the following IDLE session:



```
>>> digits = { i for i in range(10) }
>>> odds = { 1, 3, 5, 7, 9 }
>>> evens = digits - odds
>>> evens
{0, 2, 4, 6, 8}
>>> primes = { 2, 3, 5, 7 }
>>> odds | evens
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> odds & evens
set()
>>> primes & evens
{2}
>>> primes - evens
{3, 5, 7}
>>> primes ^ evens
{0, 3, 4, 5, 6, 7, 8}
>>> primes < digits
True
>>>
```

Summary

This chapter explored two built-in types—*dictionaries* and *sets*—that are useful in a variety of applications. The important points introduced in this chapter include:

- A *dictionary* associates *keys* with *values* in a way that enables clients to retrieve those associations efficiently. Python's syntax for working with a dictionary is similar to the one it uses for lists. You can retrieve the value for a particular key by enclosing the key in square brackets after the dictionary. You can set a new value for a key by assigning to that same selection expression.
- Python allows you to create a dictionary by enclosing a list of key-value pairs in curly braces. The individual entries in that list consist of a key and a value separated by a colon.
- The `for` statement makes it easy to iterate through the keys in a dictionary. Since Python 3.6, the `for` statement returns the keys in the order in which they were inserted, but relying on that behavior makes your programs less portable. You can also cycle through the key-value pairs in a dictionary by using the `items` method, which returns an iterable object that delivers each key-value pair as a tuple.
- Python programmers often use dictionaries to implement data structures that are more traditionally thought of as records. Several examples in Chapter 12 illustrate this model in more detail.
- It is possible to implement the fundamental dictionary operations by storing key-value pairs in a list. Keeping the list in sorted order by key makes it possible to find a key in $O(\log N)$ time, but this representation still requires $O(N)$ time to insert a new key.
- Dictionaries can be implemented very efficiently using a strategy called *hashing,* in which keys are converted to an integer that tells the implementation precisely where it should look for the matching key.
- A common implementation of the hashing algorithm is to allocate an array of *buckets,* each of which contains a list of the keys that hash to that bucket. As long as the ratio of the number of entries to the number of buckets does not exceed about 0.7, the operations of adding a new key or finding an existing one both run in $O(1)$ time on average. Maintaining this performance as the number of entries grows requires periodic *rehashing* to increase the number of buckets.
- A set is an unordered collection of distinct elements. You can create a set in Python by listing its elements inside curly braces. The empty set must be written as `set()` because Python interprets the expression `{}` as an empty dictionary.
- Sets provide a powerful mental model for thinking about programs because sets have a solid mathematical foundation. The fundamental operations on sets are summarized in Figure 11-14 at the top of the next page.

FIGURE 11-14 Summary of the mathematical notation for sets

Empty set	$\varnothing$	The set containing no elements
Membership	$x \in S$	True if *x* is an element of *S*
Nonmembership	$x \notin S$	True if *x* is not an element of *S*
Equality	$A = B$	True if *A* and *B* contain exactly the same elements
Subset	$A \subseteq B$	True if all elements in *A* are also in *B*
Proper subset	$A \subset B$	True if A is a subset of B but the sets are not equal
Union	$A \cup B$	The set of elements in *A, B,* or both
Intersection	$A \cap B$	The set of elements in both *A* and *B*
Set difference	$A - B$	The set of elements in *A* that are not also in *B*
Symmetric set difference	$A \triangle B$	The set of elements in *A* that are not also in *B*

Review questions

1. In your own words, define the concept of a *dictionary* as Python uses the term.

2. List at least three application contexts in which the dictionary data structure is likely to prove useful?

3. What happens if you select a key that doesn't exist in a dictionary?

4. How do you iterate over the keys in a dictionary?

5. How does Python allow you to iterate over the keys and their associated values at the same time?

6. What guarantees do the most recent versions of Python make about the order in which the `for` loop iterates through the keys in a dictionary? Why might it be unwise to rely on this behavior?

7. For the list-based implementation of a dictionary, what algorithmic strategy does the chapter suggest for reducing the cost of finding a key to $O(\log N)$ time?

8. If you implement the strategy suggested in the preceding question, why does it still require $O(N)$ time to insert a new key?

9. What is meant by the term *bucket* in the implementation of a hash table?

10. What is a *collision?*

11. In your own words, define the *pigeonhole principle.*

12. In tracing through the code that enters state abbreviations into a hash table, the text notes that the entries for `"AZ"` and `"AK"` collide in bucket #2. By looking at the diagram in Figure 11-10, determine what state abbreviations are involved in the next collision that occurs.

13. What is a *time-space tradeoff?* How does that concept apply to hash tables?

14. What is meant by the term *load factor?*

15. How does a hash table keep the load factor small as the number of keys grows.

16. What is the approximate threshold for the load factor that ensures that the average performance of a hash table is $O(1)$?

17. How does a set differ from a list?

18. What sets are denoted by each of the following symbols: $\varnothing$, **Z**, **N**, and **R?**

19. What do the symbols $\in$ and $\notin$ mean?

20. What are the mathematical symbols for the operations union, intersection, set difference, and symmetric set difference?

21. What is the difference between a subset and a proper subset?

22. Give an example of an infinite set that is a proper subset of another infinite set.

23. Evaluate the following set expressions expressed in mathematical notation:

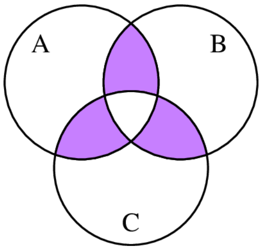
    a. $\{1, 2, 3\} \cup \{1, 3, 4\}$
    b. $\{1, 2, 3\} \cap \{1, 3, 4\}$
    c. $\{1, 2, 3\} - \{1, 3, 4\}$
    d. $(\{1, 2, 3\} - \{1, 3, 4\}) \cup (\{1, 2, 3\} - \{1, 3, 4\})$

24. For each of the following set operations, draw Venn diagrams whose shaded regions illustrate the contents of the specified set expression:

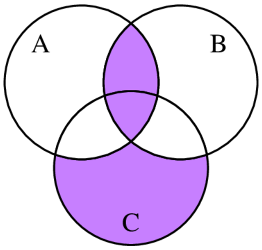
    a. $A \cup (B \cap C)$
    b. $(A - C) \cap (B - C)$
    c. $(A - B) \cup (B - A)$
    d. $(A \cup B) - (A \cap B)$

25. Write set expressions that describe the shaded region in each of the following Venn diagrams:

a.



b.



26. How would you create a Python set containing 6, 28, 496, and 8128?

27. True or false: You can use the syntax `{ }` to designate the empty set in Python.

28. What are the Python operators that correspond to the mathematical operators $\in$, $\cup$, $\cap$, Δ, $\subset$, and $\subseteq$?

29. What is the value of the Python expression `len(set("hello"))`?

Exercises

1. In many word games, letters are scored according to their point values, which are inversely proportional to their frequency in English words. In Scrabble™, the points are allocated as follows:

Points	Letters
1	*A, E, I, L, N, O, R, S, T, U*
2	*D, G*
3	*B, C, M, P*
4	*F, H, V, W, Y*
5	*K*
8	*J, X*
10	*Q, Z*

For example, the word `"FARM"` is worth 9 points in Scrabble: 4 for the *F*, 1 each for the *A* and the *R*, and 3 for the *M*. Write a function `scrabble_score` that takes a word and returns its score in Scrabble, not counting any of the other bonuses that occur in the game. You should ignore any characters other than uppercase letters in computing the score.

2. In Roman numerals, characters of the alphabet are used to represent integers as shown in this table:

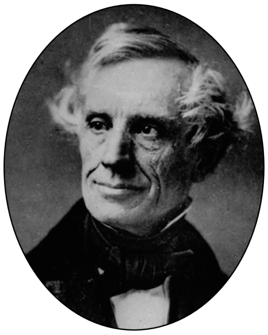
symbol	value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Each character in a Roman numeral stands for the corresponding value. Ordinarily, the value of the Roman numeral as a whole is the sum of the individual character values in the table. Thus, the string `"LXXVI"` denotes $50 + 10 + 10 + 5 + 1$, or 76. The only exception occurs when a character corresponding to a smaller value precedes a character representing a larger one, in which case the value of the first letter is subtracted from the total, so that the string `"IX"` corresponds to $10 - 1$, or 9.

Write a function `roman_to_decimal` that takes a string representing a Roman numeral and returns the corresponding decimal number. To find the values of each Roman numeral character, your function should find that character in a dictionary that implements a lookup table. If the string contains characters that are not in the table, `roman_to_decimal` should return -1.

3. Even though the `CountLetterFrequencies.py` program in Chapter 8 was designed to show how lists can be used for tabulation, its operation is conceptually more closely related to the idea of a dictionary in which each individual letter serves as a key whose corresponding value is the letter count. Rewrite the `CountLetterFrequencies.py` program so that it uses a dictionary rather than a list in its implementation. As before, the table of letter frequencies should appear in alphabetical order.

4. In May of 1844, Samuel F. B. Morse sent the message "What hath God wrought!" by telegraph from Washington to Baltimore, heralding the beginning of the age of electronic communication. In his 1998 book, *The Victorian Internet,* British journalist Tom Standage goes so far as to argue quite plausibly that the impact of the telegraph on the 19th-century world was in many ways more profound than the impact of the Internet on the 20th.


Samuel F. B. Morse

To make it possible to communicate information using only the presence or absence of a single tone, Morse designed a coding system in which letters and other symbols are represented as coded sequences of short and long tones, traditionally called *dots* and *dashes*. In Morse code, the 26 letters are represented by the codes in Figure 11-15.

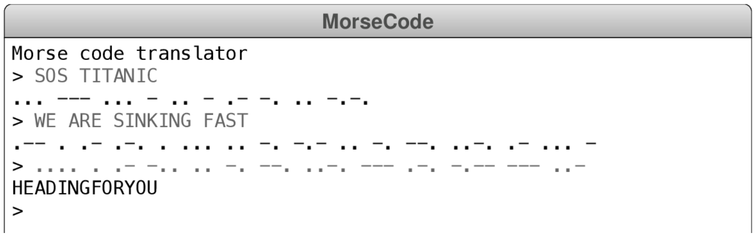
FIGURE 11-15 Morse code

A	•▬	H	••••	O	▬▬▬	V	•••▬
B	▬•••	I	••	P	•▬▬•	W	•▬▬
C	▬•▬•	J	•▬▬▬	Q	▬▬•▬	X	▬••▬
D	▬••	K	▬•▬	R	•▬•	Y	▬•▬▬
E	•	L	•▬••	S	•••	Z	▬▬••
F	••▬•	M	▬▬	T	▬		
G	▬▬•	N	▬•	U	••▬		

Write a program that reads in lines from the user and translates each line either to or from Morse code, depending on the first character of the line:

- If the line starts with a letter, you need to translate it to Morse code. Any characters other than the 26 letters should simply be ignored.
- If the line starts with a period (dot) or a hyphen (dash), it should be read as a series of Morse code characters that you need to translate back to letters. You may assume that each sequence of dots and dashes in the input string will be separated by spaces, and you are free to ignore any other characters that appear. Because there is no encoding for the space between words, the characters of the translated message will be run together.

The program should end when the user enters a blank line. A sample run of this program (taken from the messages between the Titanic and the Carpathia in 1912) might look like this:



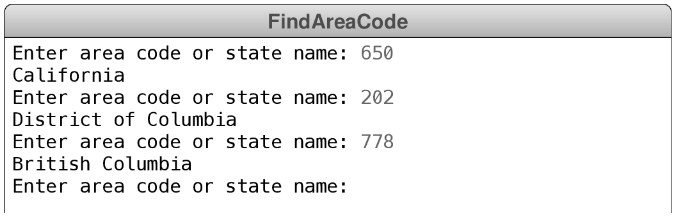
```
Morse code translator
> SOS TITANIC
... --- ... - .. - .- -. .. -.-.
> WE ARE SINKING FAST
.-- . .- .-. . ... .. -. -.- .. -. --. ..-. .- ... -
> .... . .- -.. .. -. --. ..-. --- .-. -.-- --- ..-
HEADINGFORYOU
>
```

5. Telephone numbers in the United States and Canada are organized into various three-digit ***area codes.*** A single state or province often has many area codes, but a single area code never crosses a state or provincial boundary. This rule makes it possible to list the geographical locations of each area code in a data file. For this problem, assume that you have a file `AreaCodes.txt`, which lists all the area codes paired with their locations, as illustrated by the first few lines of that file:

AreaCodes.txt

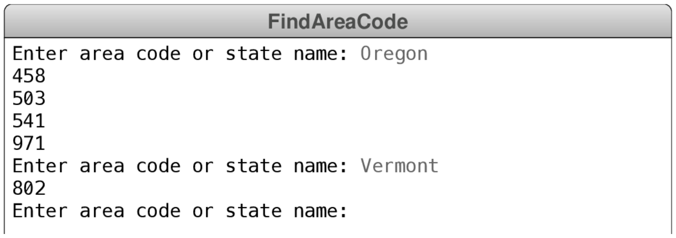
```
201–New Jersey
202–District of Columbia
203–Connecticut
204–Manitoba
205–Alabama
206–Washington
```

Using the `FindAirportCodes.py` program from Figure 11-4 as a model, write the code necessary to read this file into a dictionary where the key is the area code and the value is the location. Once you've read in the data, write a program that repeatedly asks the user for an area code and then looks up the corresponding location, as illustrated in the following sample run:



```
Enter area code or state name: 650
California
Enter area code or state name: 202
District of Columbia
Enter area code or state name: 778
British Columbia
Enter area code or state name:
```

As the prompt suggests, however, your program should also allow users to enter the name of a state or province and have the program list all the area codes that serve that area, as illustrated by the following sample run:



```
Enter area code or state name: Oregon
458
503
541
971
Enter area code or state name: Vermont
802
Enter area code or state name:
```

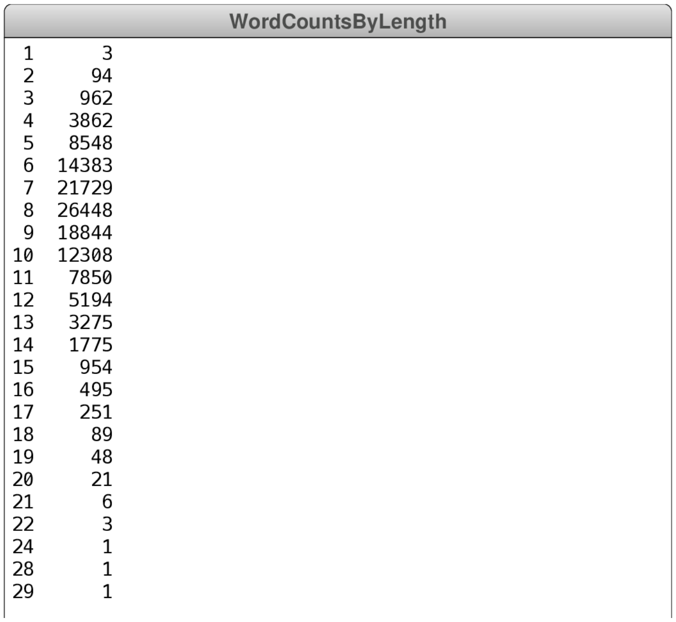
6. Poets who write rhymed verse sometimes make use of a rhyming dictionary, which lists all words with a particular ending. Although rhyming dictionaries find words based on their sound, it is still helpful to match words by spelling.

   Write a function `create_suffix_dictionary` that uses the `english` library to create a dictionary in which the keys are every string that appears at the end of some word in the dictionary and whose length is between two and five letters. The corresponding value in the dictionary should be a list of all the words in the dictionary that end with that suffix. For example, the value associated with the key `"lege"` should be the list

   `[ "allege", "college", "privilege", "sacrilege" ]`

   because those are the only English words ending in `"lege"`.

7. Write a program that displays a table showing the number of words that appear in the english module introduced in Chapter 3, sorted by the length of the word. The output of the program should look like this:



```
 1      3
 2     94
 3    962
 4   3862
 5   8548
 6  14383
 7  21729
 8  26448
 9  18844
10  12308
11   7850
12   5194
13   3275
14   1775
15    954
16    495
17    251
18     89
19     48
20     21
21      6
22      3
24      1
28      1
29      1
```

8. The two implementations of Dictionary class in this chapter—the list-based dictionary in Figure 11-7 and the hash-based dictionary in Figure 11-9—do not implement the full set of methods from Figure 11-6. Add the necessary code to implement the pop and clear methods as well as the len function, which happens automatically if you define a __len__ method for the class.

9. Modify the code for the Dictionary class in Figure 11-9 so that it implements rehashing. Your implementation should keep track of the load factor for the hash table and perform a rehashing operation that doubles the number of buckets whenever the load factor exceeds the limit indicated by a constant defined as follows:

```
REHASH_THRESHOLD = 0.7
```

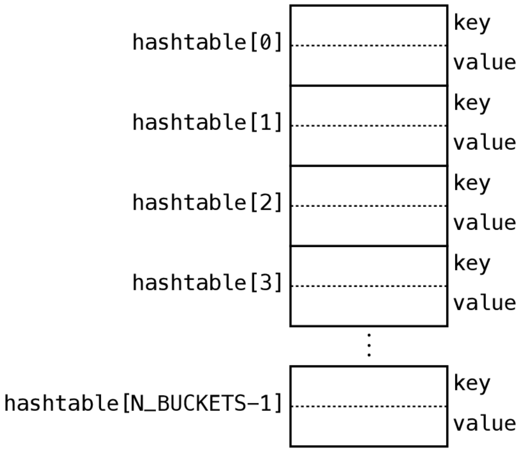
10. Write a program to evaluate the performance of the hashing algorithm by adding a display_statistics method to the Dictionary class in the HashDictionary module. This method should report the number of items, the number of buckets, and the load factor, along with the mean and standard deviation of the lengths of the bucket chains. The mean is equivalent to the traditional average. The standard deviation is a measure of how much the

individual values tend to differ from the mean. The formula for calculating the standard deviation of the lengths of the chains is

$$\sqrt{\frac{\sum_{i=1}^{N}(len_{\text{ave}} - len_{\text{i}})^2}{N}}$$

where N is the number of buckets, len_i is the length of the list stored in bucket i, and len_{ave} is the average list length. If the hash function is working well, the standard deviation should be relatively small in comparison to the mean, particularly as the number of symbols increases.

11. Although the bucket-chaining approach described in the text works well in practice, other strategies exist for resolving collisions in hash tables. In the early days of computing—when memories were small enough that the cost of introducing extra reference was taken seriously—hash tables often used a more memory-efficient strategy called ***open addressing,*** in which the key-value pairs are stored directly in a list, like this:



For example, if a key hashes to bucket #2, the open-addressing strategy tries to put that key and its value directly into the entry at `hashtable[2]`.

The problem with this approach is that `hashtable[3]` may already be assigned to another key that hashes to the same bucket. The simplest approach to dealing with collisions of this sort is to store each new key in the first free cell at or after its expected hash position. Thus, if a key hashes to bucket #2, the `put` and `get` functions first try to find or insert that key in `hashtable[2]`. If that entry is filled with a different key, however, these functions move on to try `hashtable[3]`, continuing the process until they find an empty entry or an entry with a matching key. If the index advances past the end of the list, it should wrap around back to the beginning. This strategy for resolving collisions is called ***linear probing.***

12. Sets are straightforward to implement because most of the operations can be layered on top of Python's dictionary abstraction. The basic idea is that the elements of the sets are the keys in the dictionary. An element is in the set if its corresponding value is `True`; an element is not in the set if that element does not appear as a key in the dictionary or if it has the value `False`.

    Define a `Set` class that implements the operators and methods shown in Figure 11-16. The implementation requires overloading several special methods for the various operators. The names of those methods are included in the descriptions.

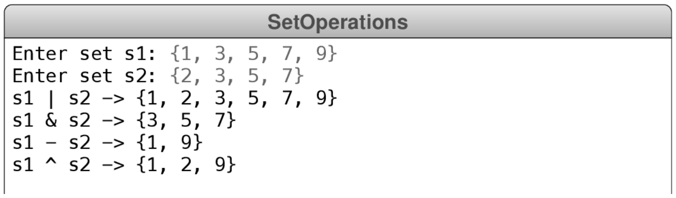
    In writing your own implementation of the `Set` class, you should keep the following points in mind:

    - You should use Python's `dict` class to store the internal dictionary.
    - Your `Set` class should support iteration.
    - Note that sets raise `ValueError` when a dictionary would raise `KeyError`.

FIGURE 11-16 Operators and methods in the dictionary-based implementation of a `Set` class

`Set()`	Creates an empty set.
`Set(`*list*`)`	Creates a set from the elements of *list* or any iterable object.
set`.clear()`	Removes all elements from the set, leaving it empty.
set`.add(`*element*`)`	Adds the specified element to the set.
set`.remove(`*element*`)`	Removes the element from the set, raising a `ValueError` if it is missing.
set`.discard(`*element*`)`	Removes the element from the set, doing nothing if it is missing.
`len(`*set*`)`	Implements the `len` function by overloading the `__len__` method.
element `in` *set*	Implements the `in` operator by overloading the `__contains__` method.
set_1 | set_2	Implements union by overloading the `__or__` method.
set_1 & set_2	Implements intersection by overloading the `__and__` method.
set_1 - set_2	Implements set difference by overloading the `__sub__` method.
set_1 ^ set_2	Implements symmetric set difference by overloading the `__xor__` method.
set_1 == set_2	Implements equality testing by overloading the `__eq__` method.
set_1 <= set_2	Implements subset testing by overloading the `__le__` method.
set_1 < set_2	Implements proper subset testing by overloading the `__lt__` method.

13. Write a Python program that reads in two sets from the user and then displays the result of applying the union, intersection, set difference, and symmetric set difference operators on those sets. A sample run of the program might look like this:



```
Enter set s1: {1, 3, 5, 7, 9}
Enter set s2: {2, 3, 5, 7}
s1 | s2 -> {1, 2, 3, 5, 7, 9}
s1 & s2 -> {3, 5, 7}
s1 - s2 -> {1, 9}
s1 ^ s2 -> {1, 2, 9}
```

The easiest way to let the user enter the sets is to use the `input` function to read in a line and then call the built-in `eval` method to evaluate the result as a Python expression.

14. The ***power set*** of a set is defined as the set of all its subsets. For example, the power set of {a, b, c} is

$$\{ \varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\} \}$$

Write a Python function `power_set` that takes a set `s` and returns a list of the subsets of `s`. For example, calling `power_set({1, 2})` should return the following Python list:

```
[ set(), {1}, {2}, {1, 2} ]
```

The return value must be a list because Python does not support sets of sets, given the implementation restriction that set elements must be immutable.

The hard part of this problem is coming up with a recursive strategy for generating the subsets. As you try to solve this problem, you need to think about how being able to generate all the subsets of a set containing $N-1$ elements might help you generate all the subsets of a set with N elements.

15. Write a program that lets the user choose an input file and then checks the spelling of all words in the file, where a word is any token that consists entirely of alphabetic characters. When your program finishes scanning the file, it should print out an alphabetical list of the words in the file that don't appear in the English dictionary. This program is surprisingly short as long as you make use of the library modules you have seen.

CHAPTER 12

Designing Data Structures

Modularity based on abstraction is the way things are done.
—Barbara Liskov, Turing Award Lecture, 2009

Barbara Liskov (1939–)

Barbara Liskov earned her bachelor's degree in mathematics from the University of California at Berkeley in 1961. After being introduced to computers and programming through jobs at the MITRE Corporation and Harvard University, Liskov returned to California, where she received her Ph.D. in Computer Science from Stanford University in 1968. For most of her career, Liskov was Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, where she conducted pioneering work on data abstraction in programming languages. The ideas that she championed about the importance of encapsulation have since become commonplace. For her many contributions, Liskov received the ACM Turing Award in 2009. On that occasion, MIT Provost L. Rafael Rife observed "every time you exchange e-mail with a friend, check your bank statement online or run a Google search, you are riding the momentum of her research."

Chapters 8, 10, and 11 introduced you to several compound types—lists, tuples, dictionaries, sets, and programmer-defined classes—which make it possible to represent collections of data values in your Python programs. Those chapters, however, concentrate on how those types are used in isolation. This chapter focuses instead on how you can combine these individual data models into data structures that are useful in applications, which requires thinking about data representation in a more holistic way.

12.1 Abstract data types

In Chapter 10, you learned about encapsulation and how to use it to define classes in which the variables that maintain the internal data are stored in attributes of the class that are marked as off-limits to clients. Those classes are examples of a more general concept in computer science called an ***abstract data type*** or ***ADT,*** which corresponds in Python to a class that uses encapsulation to separate its behavior from the details of its representation. As a client of an ADT, you know what the methods of that class do but not how those methods are implemented.

As a programming model, ADTs offer the following advantages:

- *Simplicity.* Hiding the internal representation from the client means that there are fewer details for the client to understand.
- *Flexibility.* Because a class is defined in terms of its public behavior, the programmer who implements a class is free to change the internal representation. As with any abstraction, it is appropriate to change the implementation as long as the interface remains the same.
- *Security.* The interface boundary acts as a wall that separates the client and the implementation. If a client program has access to the representation, it can change values in the underlying data structure in unexpected ways. Keeping the representation private prevents the client from making such changes.

The first two examples of ADTs from Chapter 10—the `GPoint` class from section 9.2 and the `Rational` class from section 10.3—are relatively simple. Those classes are also ***immutable,*** which means that, once you have created an object of that class, the internal state never changes. Immutable classes have many advantages, particularly in applications that use more than one processor.

In practice, many classes—including the `TokenScanner` class from section 10.5, which is the other ADT you've seen—need to maintain internal state information that changes over time. The variables that keep track of that state information must be included in the object state accessible through the `self` parameter.

12.2 Representing real-world data

One of the most important skills that software developers need to learn is how to represent real-world information in a form that computers can easily manipulate. As a concrete example, let's suppose that you have been hired by a political party to store voting data from past presidential elections on the theory that understanding the historical data may yield important insights that affect elections in the future. As a starting point, it is a useful exercise to design a data structure to represent the information shown in Figure 12-1 on the next page, which lists the popular vote for the four largest parties in the 2020 presidential election in the United States.

Given the data in Figure 12-1, the important question to ask is how to represent the electoral information in a way that preserves the relationships among the individual data values. In doing so, it is important to avoid jumping to conclusions based on the way in which the information is presented to human readers. For example, the two-dimensional structure of a printed table does not necessarily imply that the best representation is a two-dimensional array, but may simply indicate that this representation is easiest to display on the printed page.

As you design a data structure for the state-by-state electoral tallies—or any data structure, for that matter—it is important to keep in mind that Python's data structures are *tools.* Designing effective data structures requires you to think in a holistic way that takes a more abstract view. Thinking holistically makes it easier to recognize the relationships that define the overall structure.

You have already seen examples of the following abstract structures:

- *Sequences.* A ***sequence*** is an abstract data structure in which the individual elements form a logical sequence in which you can identify each element by its position. In Python, sequences are implemented using the built-in `list` type.
- *Records.* A ***record*** is an abstract data structure in which the elements are part of a logical whole but not in an ordered relationship. Python supports several strategies for implementing records. If the records are small, tuples are often an appropriate choice. If the record has more complex structure or if you want to associate methods with the data, the usual approach is to define a class to represent each record of a particular type. A third alternative is to use a dictionary in which the keys are the attribute names and the values are the corresponding attribute values.
- *Map.* A ***map*** is an abstract data structure in which a set of keys is associated with a corresponding set of values. Python implements maps using the built-in `dict` class.

FIGURE 12-1 Votes by state in the 2020 presidential election

	Electoral votes	Democratic	Republican	Libertarian	Green	Other
Alabama	9	849,624	1,441,170	25,176		7,312
Alaska	3	153,778	189,951	8,897		6,904
Arizona	11	1,672,143	1,661,686	51,465	1,557	475
Arkansas	6	423,932	760,647	13,133	2,980	18,377
California	55	11,110,250	6,006,429	187,895	81,029	115,278
Colorado	9	1,804,352	1,364,607	52,460	8,986	26,575
Connecticut	7	1,080,831	714,717	20,230	7,538	541
Delaware	3	296,268	200,603	5,000	2,139	336
District of Columbia	3	317,323	18,586	2,036	1,726	4,685
Florida	29	5,297,045	5,668,731	70,324	14,721	16,635
Georgia	16	2,473,633	2,461,854	62,229	1,013	1,231
Hawaii	4	366,130	196,864	5,539	3,822	2,114
Idaho	4	287,021	554,119	16,404	407	10,063
Illinois	20	3,471,915	2,446,891	66,544	30,494	17,900
Indiana	11	1,242,416	1,729,519	59,232	989	965
Iowa	6	759,061	897,672	19,637	3,075	11,426
Kansas	6	570,323	771,406	30,574	669	1,014
Kentucky	8	772,474	1,326,646	26,234	716	10,698
Louisiana	8	856,034	1,255,776	21,645		14,607
Maine	4	435,072	360,737	14,152	8,230	1,270
Maryland	10	1,985,023	976,414	33,488	15,799	26,306
Massachusetts	11	2,382,202	1,167,202	47,013	18,658	16,327
Michigan	16	2,804,040	2,649,852	60,381	13,718	11,311
Minnesota	10	1,717,077	1,484,065	34,976	10,033	31,020
Mississippi	6	539,398	756,764	8,026	1,498	8,073
Missouri	10	1,253,014	1,718,736	41,205	8,283	4,724
Montana	3	244,786	343,602	15,252		34
Nebraska	5	374,583	556,846	20,283		4,671
Nevada	6	703,486	669,890	14,783		17,217
New Hampshire	4	424,937	365,660	13,236	217	2,155
New Jersey	14	2,608,335	1,883,274	31,677	14,202	11,865
New Mexico	5	501,614	401,894	12,585	4,426	3,446
New York	29	5,230,985	3,244,798	60,234	32,753	26,056
North Carolina	15	2,684,292	2,758,775	48,678	12,195	20,864
North Dakota	3	114,902	235,595	9,393		1,929
Ohio	18	2,679,165	3,154,834	67,569	18,812	1,822
Oklahoma	7	503,890	1,020,280	24,731		11,798
Oregon	7	1,340,383	958,448	41,582	11,831	22,077
Pennsylvania	20	3,458,229	3,377,674	79,380		
Rhode Island	4	307,486	199,922	5,053		5,296
South Carolina	9	1,091,541	1,385,103	27,916	6,907	1,862
South Dakota	3	150,471	261,043	11,095		
Tennessee	11	1,143,711	1,852,475	29,877	4,545	23,243
Texas	38	5,259,126	5,890,347	126,243	33,396	5,944
Utah	6	560,282	865,140	38,447	5,053	19,367
Vermont	3	242,820	112,704	3,608	1,310	6,986
Virginia	13	2,413,568	1,962,430	64,761		19,765
Washington	12	2,369,612	1,584,651	80,500	18,289	34,579
West Virginia	5	235,984	545,382	10,687	2,599	79
Wisconsin	10	1,630,866	1,610,184	38,491	1,089	17,411
Wyoming	3	73,491	193,559	5,768		3,947

Choosing which of these structures to use depends on the characteristics of the data you are trying to model. If the data collection has a first element, a second element, and so on, a sequence is usually the most appropriate choice. If instead the data collection consists of independent pieces, you presumably want to use a record. Finally, if the data collection contains a set of values each of which is marked with a unique key, you are likely to choose a map.

There are at least two reasons why a two-dimensional array is probably not the best option for storing the voting data. First, elements of an array are usually of the same type, even though Python does not enforce that restriction. In the table of election results, the rows have the same structure, but the columns do not. The first column in each row is the number of electoral votes assigned to that state, while the other columns list vote totals by party. This distinction suggests that each row is best represented as a record with three properties:

1. The name of the state, which appears in Figure 12-1 as the row heading
2. The number of electoral votes
3. A dictionary that links party names and the vote totals for that party

One approach, which you will have a chance to explore in the exercises, is to define a class that encapsulates these three attributes into a single structure. That strategy has several important advantages, particularly if the classes you design include methods that associate behavior with the underlying data. In Python, however, it is often more convenient to represent data structures using only Python's built-in types. This strategy avoids the overhead of defining new classes by finding a way to represent your real-world data by assembling the complete structure from its individual pieces.

If you adopt this strategy, the data for the election would at the top level be a list in which each element corresponds to one of the fifty states plus the District of Columbia. These elements therefore represent the rows in Figure 12-1. Each of those elements is then a record containing the three items listed at the bottom of the previous page: the state name, the number of electoral votes, and the dictionary linking party names and vote totals.

Given that one of the goals of this example is to show how to create abstract data types without defining new classes, the remaining options for representing this record are to use a tuple or to create a dictionary with keys for each attribute, which might be `"name"`, `"electoral_votes"`, and `"party_totals"`. Although the first option is simpler, the second makes it possible to encode the election information in an easily readable form, as described in the following section.

Representing structured data in text form

Historically, applications that needed to read and write data to and from files did so using a representation specific to each application. The various applications that are part of Microsoft Office, for example, used to store files in binary format. Word files ending with `.doc` used one format, Excel files ending with `.xls` used a different format, and PowerPoint files ending with `.ppt` used yet another format. Following a general push toward more open standards in the industry, Microsoft changed its entire suite of applications in 2007 to use a text-based form called ***XML,*** which stands for ***Extensible Markup Language.*** This change is reflected in the new file types `.docx`, `.xlsx`, and `.pptx`. Applications that use XML to represent data files are significantly easier to write and maintain. As a result, XML has become increasingly common as a model for representing data.

XML, however, is not the only text-based model to become popular in recent years. The growing popularity of JavaScript as a language for writing web-based applications has generated increasing interest in JavaScript's data model. The growing interest has led in turn to the creation of a new standard for representing compound objects that simplifies the process of sharing data between applications, even if those applications are coded in different languages. This model is called ***JavaScript Object Notation,*** which is typically shortened to ***JSON.***

Although JSON was derived from JavaScript's standard notation for objects, it turns out to be remarkably similar to the notation that Python uses for the same purpose. Strings, numbers, lists, and dictionaries are represented identically in Python and JavaScript. As a result, JSON notation is instantly recognizable to Python programmers. Figure 12-2, for example, shows the election data from Figure 12-1 as it appears in JSON form. The code displayed in Figure 12-2 is legal JSON and legal Python at the same time.

Although there are minor differences between the syntactic rules used by JSON and Python, it is straightforward to translate one to the other. Modern versions of Python include a built-in library called `json` that translates back and forth between files written in JSON and Python's data structures. The function

`json.load(`*file*`)`

reads the next JSON data structure from the specified file object into its equivalent representation in Python. To translate in the other direction, you can call

```
json.dump(object, file)
```

which writes a text representation of the object to the specified data file.

If the file `PresidentialElection2020.json` contains the data shown in Figure 12-3, you can read that data structure into Python using the following code, assuming that you have imported the `json` library at the beginning of the module:

```
with open("PresidentialElection2020.json") as f:
    election_data = json.load(f)
```

The variable `election_data` now contains a list with 51 elements, one for each state and the District of Columbia. Each of those elements is a dictionary whose three keys define a record with the fields `name`, `electoral_votes`, and `party_totals`. The `party_totals` fields are dictionaries that map the name of a party to the number of votes it received in the election for that state.

The `CountVotes` module in Figure 12-3 on the next two pages defines several functions that work with election data along with a program to generate a report.

FIGURE 12-2 **Votes in the 2020 presidential election in JSON form**

```
[
    {
        "name": "Alabama",
        "electoral_votes": 9,
        "party_totals": {
            "Democratic": 849624,
            "Republican": 1441170,
            "Libertarian": 25176,
            "Other": 7312
        }
    },
   . . . Entries for the other states . . .
   {
        "name": "Wyoming",
        "electoral_votes": 3,
        "party_totals": {
            "Democratic": 55973,
            "Republican": 174419,
            "Libertarian": 13287,
            "Green": 2515,
            "Other": 9655
        }
    }
]
```

FIGURE 12-3 **Program to count the popular and electoral votes by state**

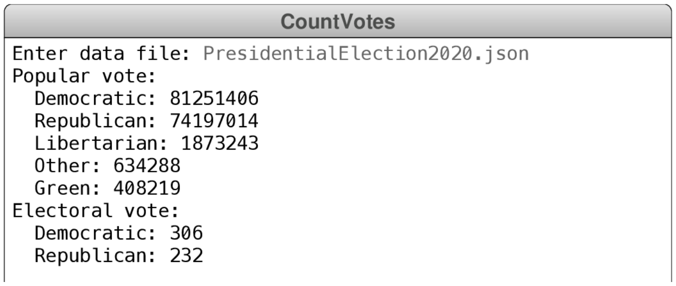
```
# File: CountVotes.py

"""
This module exports several functions for working with presidential
election data in the United States along with a main program that
produces an election report.
"""

import json

# Implementation notes: read_election_data
# ----------------------------------------
# This function reads a data file on an election stored using
# JavaScript Object Notation (JSON) n the following hierarchy:
#   + A list containing records for all 50 states in alphabetical order,
#     where each record is a dictionary with the following information:
#       - A name field, which specifies the name of the state
#       - An electoral_votes field, which is the number of electoral votes
#       - A party_totals field, which maps party names to vote totals

def read_election_data(filename):
    """Reads the data for a presidential election from the specified file."""
    with open(filename) as f:
        return json.load(f)

def count_popular_votes(election_data):
    """Returns a dictionary mapping parties to popular vote totals."""
    popular_votes = { }
    for state_data in election_data:
        for party, votes in state_data["party_totals"].items():
            if party not in popular_votes:
                popular_votes[party] = 0
            popular_votes[party] += votes
    return popular_votes

def count_electoral_votes(election_data):
    """Returns a dictionary mapping parties to electoral vote totals."""
    electoral_votes = { }
    for state_data in election_data:
        party = determine_winner(state_data["party_totals"])
        if party not in electoral_votes:
            electoral_votes[party] = 0
        electoral_votes[party] += state_data["electoral_votes"]
    return electoral_votes

def print_election_results(election_data):
    """Generates an election report with the popular and electoral votes."""
    print("Popular vote:")
    print_totals(count_popular_votes(election_data))
    print("Electoral vote:")
    print_totals(count_electoral_votes(election_data))
```

☞

FIGURE 12-3 Program to count the popular and electoral votes by state (continued)

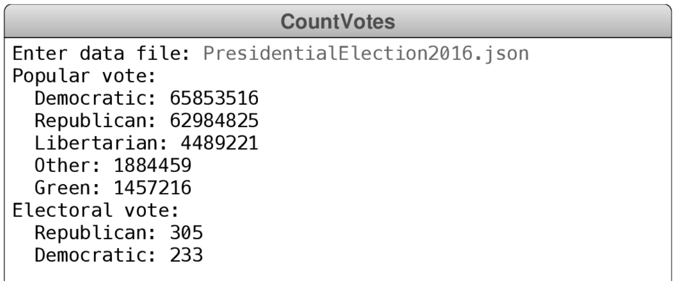
```
def print_totals(vote_totals):
    """Generates a report given a dictionary mapping parties to votes."""
    for party,votes in sorted(vote_totals.items(), key=get_vote_count,
                                                   reverse=True):
        print(f"  {party}: {votes}")

# Implementation notes: determine_winner
# --------------------------------------
# This function takes a dictionary mapping parties to votes and sorts
# it by the vote count.  It then returns immediately on the first cycle
# of the for loop.

def determine_winner(vote_totals):
    """Determines which party has the largest total in vote_totals."""
    for party,votes in sorted(vote_totals.items(), key=get_vote_count,
                                                   reverse=True):
        return party

def get_vote_count(pair):
    """Returns the number of votes from the tuple (party, votes)."""
    party,votes = pair
    return votes

def count_votes():
    """Generates an election report."""
    filename = input("Enter data file: ")
    election_data = read_election_data(filename)
    print_election_results(election_data)

# Startup code

if __name__ == "__main__":
    count_votes()
```

The effect of running `CountVotes.py` as a program is illustrated in the following sample run:



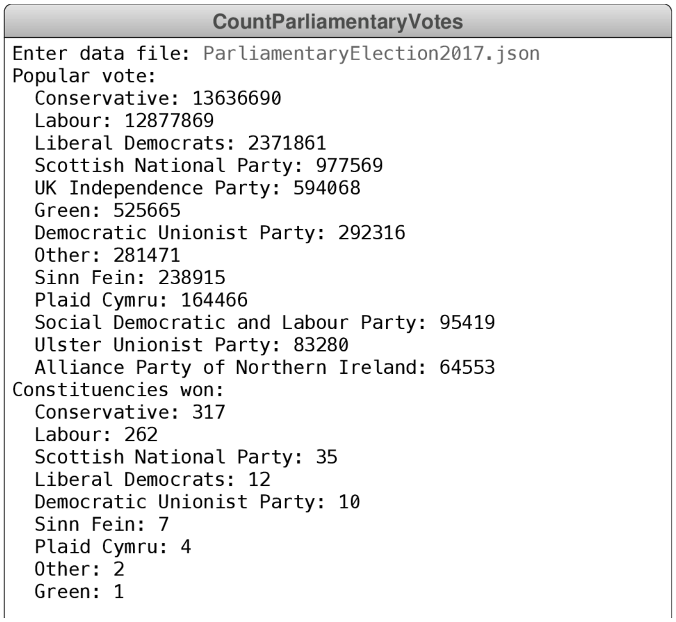
```
Enter data file: PresidentialElection2020.json
Popular vote:
  Democratic: 81251406
  Republican: 74197014
  Libertarian: 1873243
  Other: 634288
  Green: 408219
Electoral vote:
  Democratic: 306
  Republican: 232
```

The structure of the CountVotes.py application is surprisingly flexible. It is easy to substitute election data for any other year just by entering the name of a different data file. For example, if you were to create a new JSON file for the 2016 presidential election, you would see the following result:



```
Enter data file: PresidentialElection2016.json
Popular vote:
  Democratic: 65853516
  Republican: 62984825
  Libertarian: 4489221
  Other: 1884459
  Green: 1457216
Electoral vote:
  Republican: 305
  Democratic: 233
```

In much the same way, you can use this code with minimal modification to prepare reports for other elections. For example, if you prepare a JSON data file in which every electoral_votes field has the value 1, you can use the same program with minimal modifications to display the results of a parliamentary election in which the states are replaced by parliamentary constituencies. The following sample run, for example, shows the results of the June 2017 election in the United Kingdom:



```
Enter data file: ParliamentaryElection2017.json
Popular vote:
  Conservative: 13636690
  Labour: 12877869
  Liberal Democrats: 2371861
  Scottish National Party: 977569
  UK Independence Party: 594068
  Green: 525665
  Democratic Unionist Party: 292316
  Other: 281471
  Sinn Fein: 238915
  Plaid Cymru: 164466
  Social Democratic and Labour Party: 95419
  Ulster Unionist Party: 83280
  Alliance Party of Northern Ireland: 64553
Constituencies won:
  Conservative: 317
  Labour: 262
  Scottish National Party: 35
  Liberal Democrats: 12
  Democratic Unionist Party: 10
  Sinn Fein: 7
  Plaid Cymru: 4
  Other: 2
  Green: 1
```

There are more parties, but the structure of the program is the same.

The code in Figure 12-5 includes another feature that is worth noting. The functions `print_totals` and `determine_winner` each use the built-in `sorted` function to ensure that the `for` loop iterates through the sequence of party-vote pairs in descending order by the vote component of the tuple. The `key` and the `reverse` parameters have exactly the same effect for `sorted` as they do for the built-in `sort` function, which was described in Chapter 9.

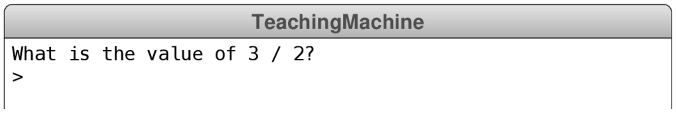
12.3 Data-driven programs

Computers got their start as machines designed to process data. Today, programs often use data not as passive information to be processed but instead to control the program's operation. Programs that allow data to control their execution are said to be ***data-driven.*** In a typical data-driven program, the source of the data is *external* to the program, in the sense that it is not actually part of the code. The data may be stored in a text file or in some more highly structured form. The program then operates in two phases. In the first phase, the program reads the data from the external source into an *internal* data structure that represents the same information. In the second, the program uses the internal structure to control its operation.

Programmed instruction courses

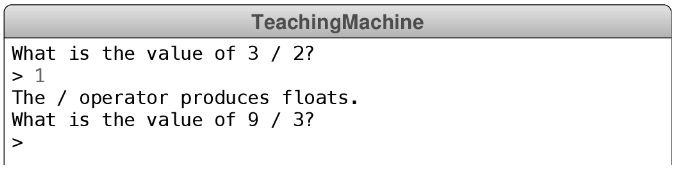
As with most programming concepts, the idea of a data-driven program is easiest to illustrate by example. The goal of this section is to create a "teaching machine" that uses a strategy called ***programmed instruction*** in which a computerized teaching tool asks a series of questions so that previous answers determine the order of subsequent questions. As long as a student is getting the right answers, the programmed instruction process skips the easy questions and moves on to more challenging topics. For the student who is having trouble, the process moves more slowly, leaving time for repetition and review. Although the idea of programmed instruction was quite the rage some 40 years ago, it didn't live up to the potential its proponents claimed. Even so, building a simple teaching machine based on the programmed-instruction model offers a useful illustration of data-driven programs.

To make the idea of a teaching-machine application more concrete, it helps to imagine how the student might use it. When the program starts, it begins by asking the student a question. For example, a programmed-instruction course on Python might begin like this:



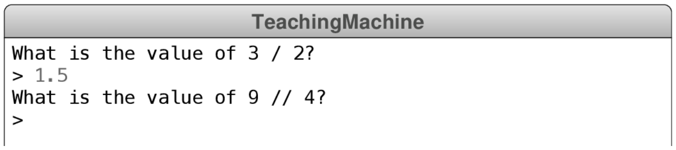
```
What is the value of 3 / 2?
>
```

The program then waits for the student to enter an answer. Depending on the response, the program will choose the next question either to provide more review or to let the student move ahead more quickly. For example, if the student enters an incorrect answer, the program will provide feedback and continue with another question about the / operator, which might look like this:



```
What is the value of 3 / 2?
> 1
The / operator produces floats.
What is the value of 9 / 3?
>
```

If the student instead supplies the correct response, the program moves on to a different topic, as shown in the following console session:



```
What is the value of 3 / 2?
> 1.5
What is the value of 9 // 4?
>
```

Students who continue to supply the correct answers can proceed through the course very quickly. Those who are having more trouble with the material have to make their way through a larger set of questions.

Designing for flexibility

It is possible to design a programmed instruction application by writing a set of Python functions. Each function asks a question, reads in an answer, and then calls another function appropriate to the answer the student supplies. Such a program, however, would be difficult to change. Someone who wanted to add questions or design an entirely new course would need to write new functions. Writing functions is simple enough for someone who understands programming, but not everyone does. The designers of programmed instruction courses are typically teachers in a specific discipline with little programming expertise. Forcing them to work in the programming domain—or even to use the JSON format introduced earlier in the chapter—limits the number of people who can use your application.

As the software developer for the teaching machine project, your goal is to write an application that presents a programmed instruction course to the student but allows teachers without programming skills to supply the questions, expected answers, and sequencing information that allows the application to ask the questions in the appropriate order. To do so, the best approach is to design your application as a general tool that takes all data pertaining to the programmed instruction course from a data file. If you adopt this approach, the same program can present many different courses by using different files.

Framing the problem

At one level, it is easy to outline the operation of the teaching machine application. When the program runs, it repeatedly executes the following steps:

1. Ask the student the current question. A question consists of one or more lines of text, which you can represent as strings.
2. Request an answer from the student, which can also be represented as a string.
3. Look up the answer in a list of possibilities provided for that question. If the answer appears in the list, consult the data structure to choose what question should become the new current question. If the student's answer does not match any of the possibilities provided by the course data file, the student should be informed of that fact and given another chance at the same question.

Many details are missing from this outline, but it is a start. Even at this level, the outline provides some insight into the eventual implementation. For example, you know that you need to keep track of what the "current question" is, which means that you need to have some way of identifying individual questions.

Coding the actual program turns out to be one of the easier pieces of the task; the harder problems arise in designing an appropriate data structure. For the program to be general and flexible, all the information that pertains to an actual course cannot be built into the program but must instead be stored in a data file. The program's job is to read that data file, store the information in an internal data structure, and then process that structure as outlined earlier in this section. Thus, your next major task is to design the data structures required for the problem so that you have a context for building the program as a whole.

The process of designing the data structure has two distinct components. First, you have to design an *internal* data structure for use by the program. The internal data structure consists of type definitions that use Python's data structures so that the resulting types mirror the organization of the real-world information you seek to represent. Second, you must design an *external* data structure that indicates how the information is stored in the data file. These two processes are closely related, mostly because they each represent the same information. Even so, the two structures are tailored to meet different purposes. The internal structure must provide all the information necessary to write the program. The external structure must allow someone without much programming knowledge to write a course.

Designing the internal representation

The first step in the process is to design a data structure that incorporates the necessary information. If you spend a little time thinking about what data must be stored for the teaching machine application, it quickly becomes clear that there are two levels at which creating an abstract data type makes sense. The first is at the level of the course as a whole. The second is at the level of an individual question. Each of these abstract types can be implemented as a Python class. The `TMCourse` class models the complete course and therefore conceptually contains a list of questions. The `TMQuestion` class models a single question and contains the text of the question, along with a list of possible answers and their associated transitions.

The process of identifying what classes you need and giving those classes names helps enormously in refining the data structure design. Even so, there are still many details that you need to fill in before you can write the implementation. The preceding paragraph observes that a course contains what is conceptually a list of questions, but it is not yet clear whether the underlying implementation should use Python's `list` class or some other structure. To make that decision, you need to think more carefully about how to identify each individual question.

One of the dangers in making a premature commitment to a particular data structure is that doing so may lead you to make other decisions that are not the best for the application. For example, if you somehow get it into your head that the questions in a `TMCourse` object need to be stored in a Python list, the natural assumption is that the questions will be numbered. That idea certainly seems reasonable at first. The questions, of course, do get written down in some order, and it is not immediately clear why a list is not precisely the structure you need.

Using sequential numbers to identify each question makes it much more difficult to edit the course. Each question must be able to refer to other questions so that the teaching machine program can move through the questions in a sequence controlled by the student's answers. Given a particular question, a correct response might take the student to question 6, and an incorrect answer might direct the student to question 17. But what happens if a course designer decides to add a new question? In that case, any question numbers that follow the inserted question would increase by one. The designer would have to go back through the course and fix any references to questions whose sequence numbers had changed.

This problem can be avoided entirely by giving the questions names instead of numbers. The course designer gives each question a name, which is then used to identify that question in the sequencing information. Given that design, adding a new question doesn't require changing any existing ones, except for those that refer to the new question. If questions are named rather than numbered, the underlying structure inside TMCourse is presumably not a list but instead a dictionary that maps question names to the corresponding TMQuestion objects.

The TMQuestion class encapsulates several different attributes: the question name, the text of the question, and a structure that associates each possible answer with the name of the question that follows this one if the student gives that answer. The name attribute is a string, the text attribute is presumably a list of strings to accommodate multiline questions, and the structure containing the answers is almost certainly a dictionary since it associates answers with question names.

Designing the external structure

Before you turn to the details that will allow you to write the definitions of TMCourse and TMQuestion, it helps to think about how that information is stored in the data file. Files are simply text, and the organization provided by the Python data structures must be expressed in the design of the file format. The file format must also make it easy for someone to write and edit, even if that person is not a programmer. Thus, you should choose a representation that is as simple as possible. In this case, it seems easiest to write out each question, one after another, along with its likely answers. So that the computer can tell where the one question stops and the next question begins, you must define some convention for separating the individual questions. A blank line works well in this context, as it does in most file structures. Thus, in individual units separated by blank lines, you have the data for each question and its answers.

But what goes into the text representation for the information pertaining to a single question? First of all, you need the text of the question, which consists of individual lines from the file. You also need some way to indicate the end of the question text, and the easiest way, both for you and for the course designer, is to define a sentinel. The program defined later in this chapter uses a line of five dashes for this purpose. Furthermore, you must allow the course designer to specify pairs that link an answer with the name of the next question. A simple approach is to specify both these values on a single line consisting of the answer text, a colon, and the name of the next question. Other formats are certainly possible, but this design seems as if it would be easy for a course designer to learn. Thus, the data for an individual question entry in the file looks like this:

```
DivQ1
What is the value of 3 / 2?
-----
1: DivQ2
1.5: DivQ4
```

The name of the question is "DivQ1", which presumably indicates that this is the first question about Python's division operator. The text of the question consists of a single line, after which there are two acceptable answers. If the student types in the incorrect answer 1, the program should go to the question named "DivQ2". If the student types in the correct answer of 1.5, the program should move on to the question named "DivQ4".

But what if the student types in some other answer like 17 or 3/2? In the original informal design, the proposal was to have the application tell the student that it didn't recognize the answer and then repeat the same question. As is often the case, working through a problem reveals weaknesses in the initial design. It would be better if the data structure allowed the course designer to specify a default next question if the student offers any other answer. Computer scientists often use an asterisk to match an arbitrary string of characters, so this notation seems as if it would be a reasonably intuitive choice. Adding in an any-other-answer transition leads to the following file format for the first question:

```
DivQ1
What is the value of 3 / 2?
-----
1: DivQ2
1.5: DivQ4
*: DivQ3
```

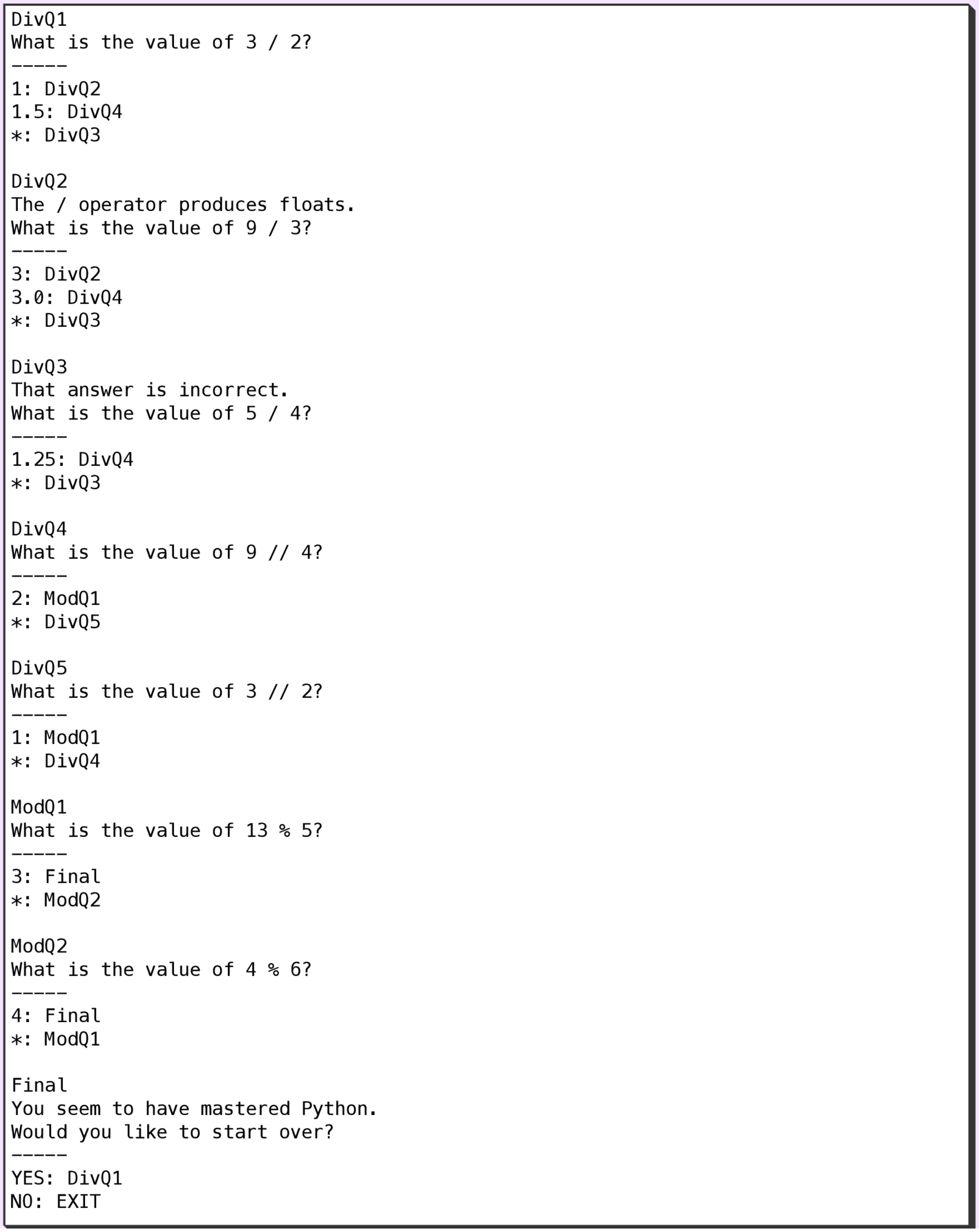
The data file entries for DivQ2 and DivQ3 might then look like this:

```
DivQ2
The / operator produces floats.
What is the value of 9 / 3?
-----
3: DivQ2
3.0: DivQ4
*: DivQ3

DivQ3
That answer is incorrect.
What is the value of 5 / 4?
-----
1.25: DivQ4
*: DivQ3
```

Each of these questions tells the student that the answer is incorrect, but DivQ2 can be more specific about the reasons.

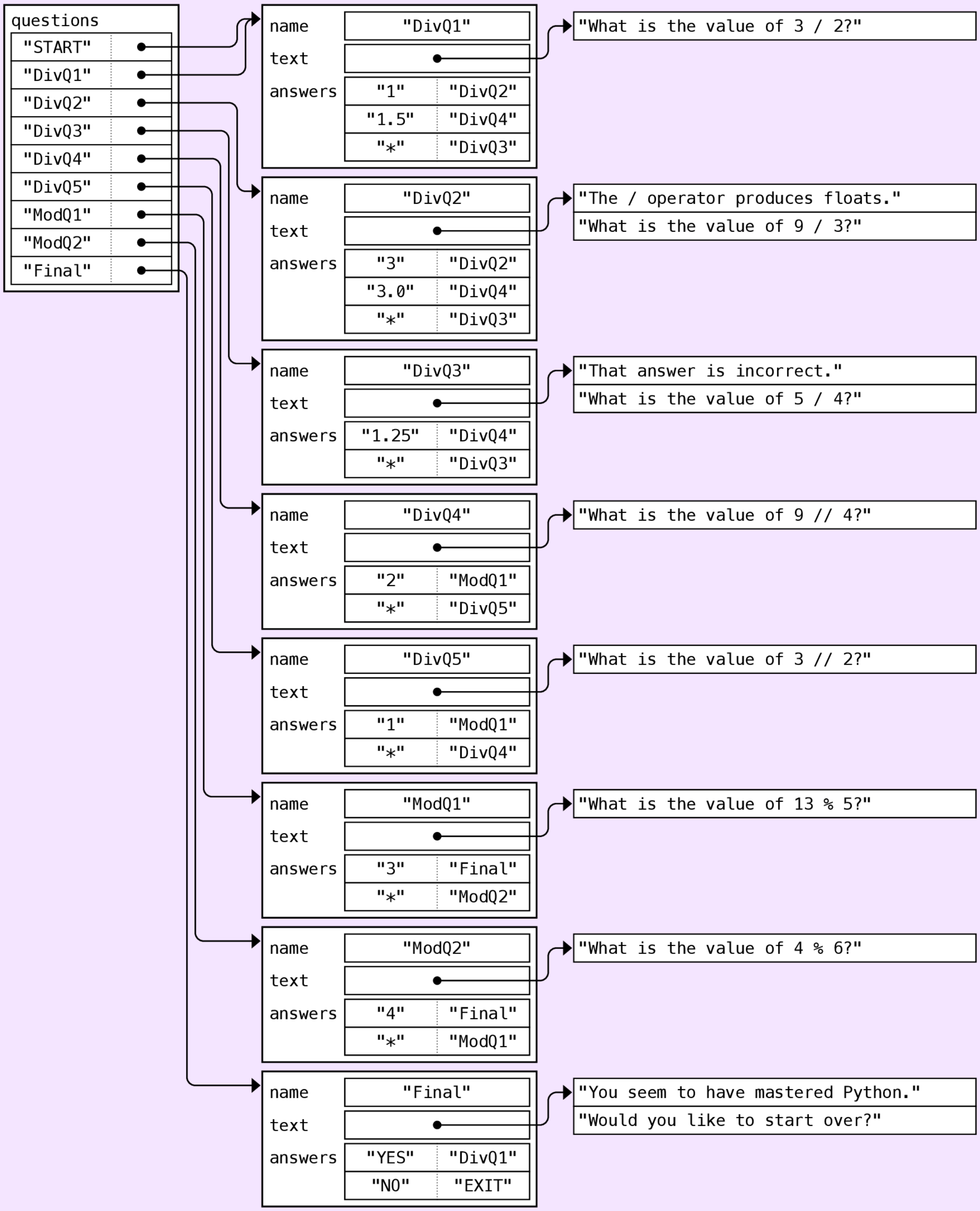
Figure 12-6 on the next page shows a complete data file for a short course on Python. Figure 12-7 on page 432 shows the internal form of the same information.

FIGURE 12-4 The external data structure as it appears in the `Python.txt` file



```
DivQ1
What is the value of 3 / 2?
-----
1: DivQ2
1.5: DivQ4
*: DivQ3

DivQ2
The / operator produces floats.
What is the value of 9 / 3?
-----
3: DivQ2
3.0: DivQ4
*: DivQ3

DivQ3
That answer is incorrect.
What is the value of 5 / 4?
-----
1.25: DivQ4
*: DivQ3

DivQ4
What is the value of 9 // 4?
-----
2: ModQ1
*: DivQ5

DivQ5
What is the value of 3 // 2?
-----
1: ModQ1
*: DivQ4

ModQ1
What is the value of 13 % 5?
-----
3: Final
*: ModQ2

ModQ2
What is the value of 4 % 6?
-----
4: Final
*: ModQ1

Final
You seem to have mastered Python.
Would you like to start over?
-----
YES: DivQ1
NO: EXIT
```

FIGURE 12-5 The corresponding internal data structure



Coding the program

Once you have defined the internal data structure and the external file format, the process of writing the code for the teaching machine program is reasonably straightforward. The main program for the teaching machine application appears in Figure 12-6, at the bottom of this page. Like most data-driven programs, the teaching machine applications runs in two phases. The first phase reads in the data from the data file and translates it into its internal form. The second phase uses the internal structure to step through the program operation as specified by the data. The TeachingMachine.py module delegates responsibility for reading the data files and translating them into the internal representation to two other classes—TMCourse and TMQuestion—each of which is defined in its own module.

The code for the TMCourse.py module appears in Figure 12-7, which begins at the top of the next page. The module includes both the definition of the TMCourse class and a top-level function called read_course that reads an open data file and returns a TMCourse object. As the comments indicate, the read_course function is logically associated with the TMCourse class but is not applied to an object.

FIGURE 12-6 **Main program for the teaching machine**

```python
# File: TeachingMachine.py

from TMCourse import read_course

# Implementation notes: TeachingMachine
# -------------------------------------
# This program implements a simple teaching machine that walks the user
# through a series of questions read from a data file.

def teaching_machine():
    course = choose_course()
    course.run()

def choose_course():
    """Returns a course chosen by the user."""
    while True:
        try:
            filename = input("Enter course name: ")
            with open(filename + ".txt") as f:
                return read_course(f)
        except IOError:
            print("Please enter a valid course name.")

# Startup code

if __name__ == "__main__":
    teaching_machine()
```

FIGURE 12-7 Class that represents a course for the teaching machine

```python
# File: TMCourse.py

"""This module defines a class to represent a TeachingMachine course."""

# Implementation notes: TMCourse
# ------------------------------
# The data file for a course consists of questions, each of which has
# the following format:
#
#     Name of the question (a single line)
#     The text of the question (one or more lines)
#     -----
#     Lines for the answers in the form <answer>:<next-question>

from TMQuestion import read_question

class TMCourse:

    def __init__(self, questions):
        """Creates a new TMCourse object with the specified questions."""
        self._questions = questions

    def run(self):
        """Steps through the questions in this course."""
        current = "START"
        while current != "EXIT":
            question = self._questions[current]
            for line in question.get_text():
                print(line)
            response = input("> ").strip().upper()
            answers = question.get_answers()
            next_question = answers.get(response, None)
            if next_question is None:
                next_question = answers.get("*", None)
            if next_question is None:
                print("I don't understand that response.")
            else:
                current = next_question

# Implementation notes: read_course
# ---------------------------------
# The read_course function logically belongs to the TMCourse.py module
# but not a part of the TMCourse class.  Unlike the other methods in
# the class, read_course is not applied to a TMCourse instance but
# instead returns a new one.
#
# To ensure that the program starts with the first question, the
# read_course function always stores a reference to the first
# question in the file under the key "START" as well as its name.
```

FIGURE 12-7 Class that represents a course for the teaching machine (continued)

```
def read_course(f):
    """Reads the entire TMCourse from the data file f."""
    questions = { }
    finished = False
    while not finished:
        question = read_question(f)
        if question is None:
            finished = True
        else:
            name = question.get_name()
            if len(questions) == 0:
                questions["START"] = question
            questions[name] = question
    return TMCourse(questions)
```

The TMCourse class exports two methods. The first is get_question, which returns the TMQuestion that corresponds to a question name. The second is the run method, which guides the user through the questions, adapting the order of questions to the user's responses.

Figure 12-8 shows the code for the TMQuestion class, which stores the data for a single question. The TMQuestion constructor takes the internal data values, which are the question name, the list of lines containing the question text, and the dictionary mapping answers to the name of the next question. The class exports getter methods for the name and text, along with a lookup_answer method, which checks the user's response against the expected answer and returns the name of the next question. The lookup_answer method also checks to see if the answer dictionary contains the special key "*", which matches any response. If there is no "*" option and no answer matches the response, lookup_answer returns None.

FIGURE 12-8 Class that represents a single question for the teaching machine

```
# File: TMQuestion.py

"""This module defines a class to represent a single question."""

class TMQuestion:

    def __init__(self, name, text, answers):
        """Creates a new TMQuestion object with these attributes."""
        self._name = name
        self._text = text
        self._answers = answers
```

FIGURE 12-8 **Class that represents a single question for the teaching machine (continued)**

```
    def get_name(self):
        """Returns the name of this question."""
        return self._name

    def get_text(self):
        """Returns the list containing the text of this question."""
        return self._text

    def get_answers(self):
        """Returns the map between a response and the next question."""
        return self._answers.copy()

# Implementation notes: read_question
# -----------------------------------
# This method is defined within the TMQuestion.py module but outside the
# TMQuestion class.  Unlike the other methods in the class, read_question
# is not applied to a TMQuestion instance but instead returns a new one.

MARKER = "-----"                             # Marker at end of question text

def read_question(f):
    """Reads the next TMQuestion, returning None at the end."""

    name = f.readline().rstrip()             # Read the question name
    if name == "":                           # Returning None at the end
        return None

    text = [ ]                               # Read the question text
    finished = False
    while not finished:
        line = f.readline().rstrip()
        if line == MARKER:
            finished = True
        else:
            text.append(line)

    answers = { }                            # Read the answer dictionary
    finished = False
    while not finished:
        line = f.readline().rstrip()
        if line == "":
            finished = True
        else:
            colon = line.find(":")
            if colon == -1:
                raise ValueError("Missing colon in " + line)
            response = line[:colon].strip().upper()
            next_question = line[colon + 1:].strip()
            answers[response] = next_question

    return TMQuestion(name, text, answers)  # Return the completed object
```

Changing the application domain

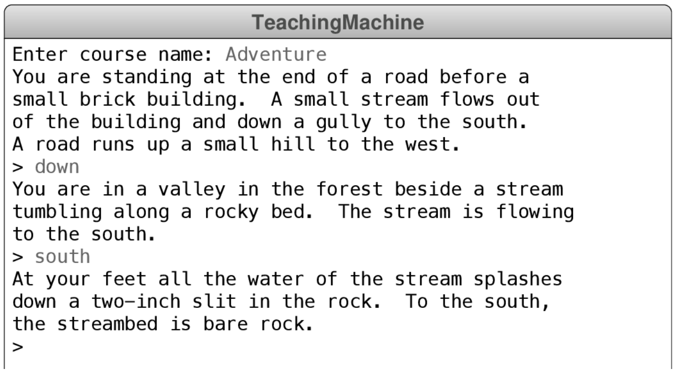
The fact that the `TeachingMachine.py` application takes its data from a data file makes it possible to use the program in entirely different contexts. As an example, if you ask the program to use the data file shown in Figure 12-9, the same teaching machine program plays a game reminiscent of the Adventure program created by Willie Crowther in the early 1970s.

As a player in Crowther's Adventure game, you assume the role of an adventurer wandering through a ***cave.*** The individual locations in the cave are generically called ***rooms,*** even though they might be outside. You move through the cave by typing simple commands, which the program uses to move you from room to room. The original game also includes objects that you can pick up along the way, some of which give you access to otherwise inaccessible rooms. The teaching machine doesn't support that feature, but you will have the chance to implement it in exercise 6.

FIGURE 12-9 The beginning of the `Adventure.txt` file

```
OutsideBuilding
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
-----
SOUTH: Valley
DOWN: Valley
NORTH: InsideBuilding
IN: InsideBuilding
WEST: EndOfRoad
UP: EndOfRoad

Valley
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
-----
SOUTH: SlitInRock
DOWN: SlitInRock
NORTH: OutsideBuilding

SlitInRock
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
-----
NORTH: Valley
SOUTH: OutsideGrate
DOWN: OutsideGrate
```

If you run the TeachingMachine.py program with the Adventure.txt data file, you might see a console session that looks like this:



```
Enter course name: Adventure
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
> down
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
> south
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
>
```

As the sample run shows, someone who uses the program with the Adventure.txt file will perceive the program's purpose very differently than someone who runs it with the Python.txt file. Even though the TeachingMachine program has not changed at all, the programmed instruction course has become an adventure game. The only difference is the data file.

Summary

This chapter explores how to create more sophisticated data structures using the tools that Python provides. Important points in this chapter include the following:

- Classes that implement a set of operations without revealing the internal data structures are called *abstract data types.* Abstract data types offer several advantages including simplicity, flexibility, and security.
- Separating behavior and representation in an abstract data type allows the implementer to change that representation without adversely affecting clients.
- In designing the data structure for an application, it is usually better to think in terms of abstract conceptual models—sequences, records, and maps—rather than the concrete structures—lists, classes, and dictionaries—used to represent them.
- It is important to think carefully about structural relationships and avoid jumping to premature conclusions arising from how information is presented.
- Modern applications tend to use text-based representations to store complex data structures in files. Two common text-based strategies are *XML* (Extended Markup Language) and *JSON* (JavaScript Object Notation). The JSON model is particularly useful in Python because the syntax for strings, numbers, lists, and dictionaries in Python matches the syntax used in JSON.

- Programs in which the control flow is determined by the data structure are said to be *data driven.* Data-driven programs are usually shorter, more flexible, and easier to maintain than programs that incorporate the same information directly into the program design.
- Data-driven programs typically have two formats for the data: an *external format* stored in a data file and an *internal format* stored as a hierarchical combination of objects and built-in types.

Review questions

1. What is an *abstract data type?*

2. True or false: One of advantages of separating the behavior of an abstract data type from its underlying representation is that doing so makes it possible to change that representation without forcing clients to change their programs.

3. What do the abbreviations *XML* and *JSON* stand for?

4. True or false: The syntax for strings, numbers, lists, and dictionaries is the same in both Python and JSON.

5. What is a *data-driven program?*

6. In your own words, describe the differences between the internal and external data representations used by data-driven programs.

Exercises

1. Use the data from the `PresidentialElection2020.json` file to find all states in which the winning candidate got less than 50 percent of the vote. In 2020, three of these states were won by Democrats and one by Republicans.

2. Following the suggestion on page 419, rewrite the `CountVotes.py` program so that it uses a class to store the information for each state. Your revised program must then read data from a text file instead of a JSON file.

3. Suppose that a bank has hired you as a programmer and given you the task of automating the process of converting between different foreign currencies at the prevailing rate of exchange. Every day, the bank receives a file called `ExchangeRates.json` containing the current exchange rates stored in JSON format as shown in Figure 12-10 at the top of the next page. Each value in the `currencies` dictionary is itself a dictionary that specifies the name of the

FIGURE 12-10 Data file containing exchange rates expressed as a JSON structure

```
{
   "date": "13-Sep-2017",
   "currencies": {
      "USD": { "name": "US dollar", "rate": 1.00000 },
      "EUR": { "name": "European Euro", "rate": 1.07397 },
      "JPY": { "name": "Japanese yen", "rate": 0.00889 },
      "GBP": { "name": "Pound sterling", "rate": 1.23586 },
      "AUD": { "name": "Australian dollar", "rate": 0.77300 },
       . . . Entries for the other currencies . . .
      "THB": { "name": "Thai baht", "rate": 0.02885 },
      "MYR": { "name": "Malaysian ringgit", "rate": 0.22590 }
   }
}
```

currency and its current exchange rate relative to the dollar. For example, the entry

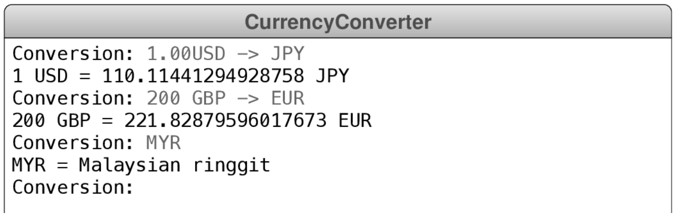
```
"GBP": { "name": "Pound sterling", "rate": 1.23586 }
```

indicates that the three-letter code `"GBP"` has the name `"Pound sterling"` and is currently trading at 1.23586 dollars to the pound.

Your task in this problem is to write a program that reads conversion requests from the user in the form

amount XXX `->` *YYY*

where *amount* is the monetary value you want to convert, and *XXX* and *YYY* are the three-letter codes for the old and new currency. Alternatively, the input line may consist of a three-letter currency code, in which case the program should report the full name of that currency. A sample run that illustrates both input forms might look like this:



```
Conversion: 1.00USD -> JPY
1 USD = 110.11441294928758 JPY
Conversion: 200 GBP -> EUR
200 GBP = 221.82879596017673 EUR
Conversion: MYR
MYR = Malaysian ringgit
Conversion:
```

4. In J. K. Rowling’s *Harry Potter* series, the students at Hogwarts School of Witchcraft and Wizardry study many forms of magic. One of the most difficult fields of study is potions, which is taught by Harry’s least favorite teacher, Professor Snape. Mastery of potions requires students to learn complex lists of

ingredients for creating the desired magical concoctions. Presumably to protect those of us in the Muggle world, Rowling does not give us a complete ingredient list for most of the potions used in the series, but we do learn about a few, including those shown in Figure 12-11.

Design a data structure that encodes the information shown in Figure 12-11 and then create a Python file that stores that information in JSON form. Test your data structure by writing a console-based program that requests a potion name from the user and then displays a list of its ingredients.

5. In recent years, the globalization of the world economy has put increasing pressure on software developers to make their programs operate in a wide variety of languages. That process used to be called *internationalization,* but is now more often referred to (perhaps somewhat paradoxically) as *localization.* In particular, the menus and buttons that you use in a program should appear in a language that the user knows.

   Your task in this problem is to write a definition for a class called `Localizer` designed to help with the localization process. The constructor for the class has the form

   ```
   def __init__(self, filename):
   ```

   The constructor creates a new `Localizer` object and initializes it by reading the contents of the data file. The data file consists of an English word, followed by any number of lines of the form

   xx=translation

   where *xx* is a standardized two-letter language code, such as `de` for German, `es` for Spanish, and `fr` for French. Part of such a data file, therefore, might look like this:

   Localizations.txt

   ```
   Cancel
   de=Abbrechen
   es=Cancelar
   fr=Annuler
   Close
   de=Schließen
   es=Cerrar
   fr=Fermer
   OK
   fr=Approuver
   Open
   de=Öffnen
   es=Abrir
   fr=Ouvrir
   ```

This file tells us, for example, that the English word `Cancel` should be rendered in German as `Abbrechen`, in Spanish as `Ayudar`, and in French as `Annuler`.

Beyond the implementation of the constructor, the only public method you need to define for `Localizer` is

```
def localize(self, word, language)
```

which returns the translation of the English word as specified by the two-letter language parameter. For example, if you have initialized a variable `localizer` by calling

```
localizer = Localizer("Localizations.txt")
```

you could then call

```
localizer.localize("Open", "de")
```

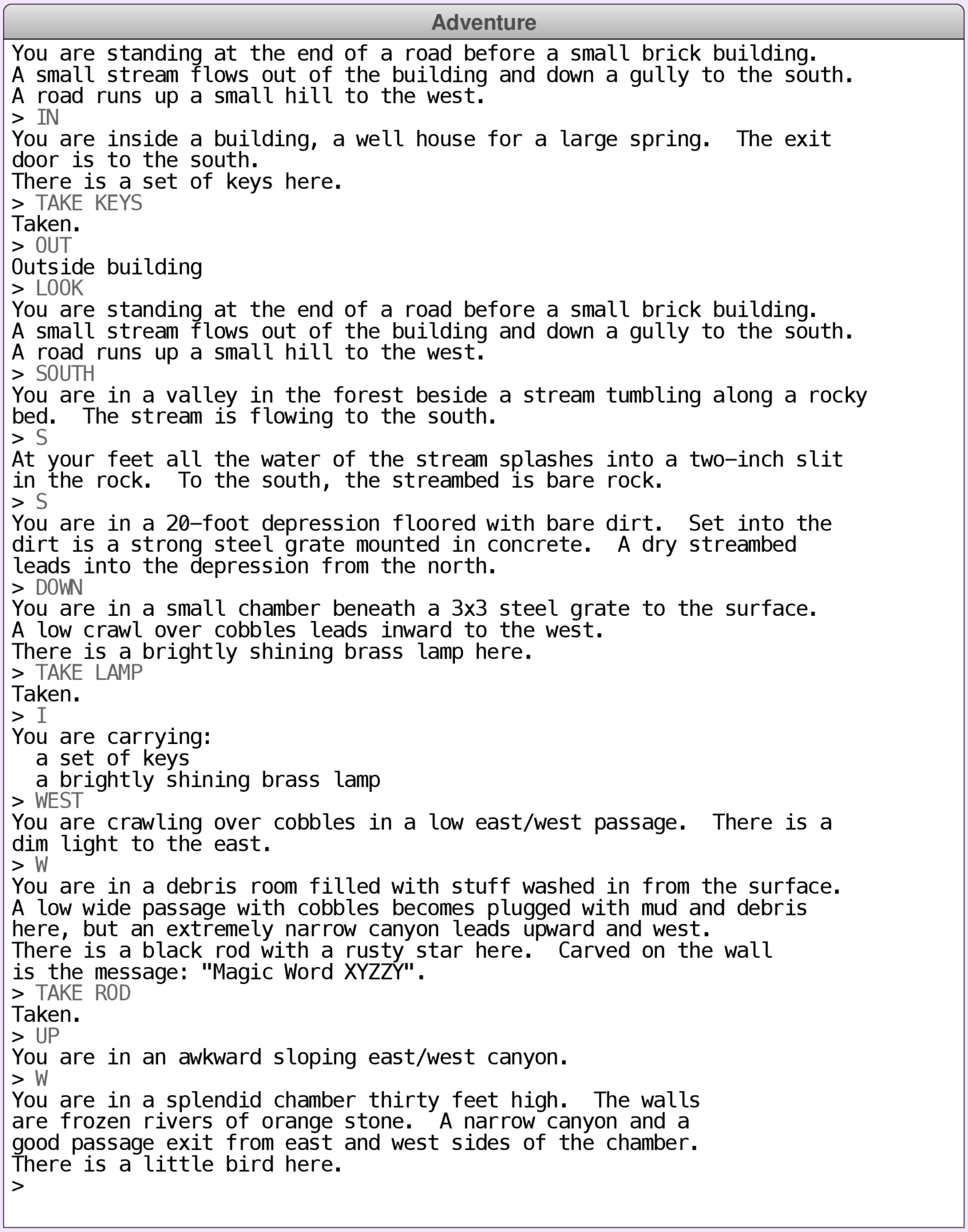
and expect it to return the string `"Öffnen"`. If no entry appears in the table for a particular word, `localize` should return the English word unchanged. Thus, `OK` becomes `Approuver` in French, but would remain as `OK` in Spanish or German.

6. In its current implementation, the `TeachingMachine.py` program provides no feedback when the user gives an incorrect answer. Design a strategy to allow the course designer to specify an optional message along with each possible response. If this message exists in the data file, the program should display that message before asking the next question.

7. Modify and extend the `TeachingMachine.py` program so that it plays a more interesting Adventure game. Implementing the following changes should allow you to reproduce the transcript from Willie Crowther's original Adventure game shown in Figure 12-12 on the next page:

   - Change the names of the modules and data structures so that they are appropriate for the Adventure game. Your code, for example, should define classes like `AdventureGame` and `AdventureRoom` instead of `TMCourse` and `TMQuestion`, which don't make sense in the context of Adventure.
   - Add a short description to the data file for the rooms so that the player sees the long description of a room only on the first visit. You should also add a `LOOK` command that prints the long description.
   - Add objects to the game. The code that works with objects should be data-driven, which means that you need to design an external data structure that keeps track of at least the following information: the word used to refer to the object, a description of the object, and the name of the room in which the object is initially located.

FIGURE 12-12 **Transcript from Willie Crowther's Adventure game**



```
You are standing at the end of a road before a small brick building.
A small stream flows out of the building and down a gully to the south.
A road runs up a small hill to the west.
> IN
You are inside a building, a well house for a large spring.  The exit
door is to the south.
There is a set of keys here.
> TAKE KEYS
Taken.
> OUT
Outside building
> LOOK
You are standing at the end of a road before a small brick building.
A small stream flows out of the building and down a gully to the south.
A road runs up a small hill to the west.
> SOUTH
You are in a valley in the forest beside a stream tumbling along a rocky
bed.  The stream is flowing to the south.
> S
At your feet all the water of the stream splashes into a two-inch slit
in the rock.  To the south, the streambed is bare rock.
> S
You are in a 20-foot depression floored with bare dirt.  Set into the
dirt is a strong steel grate mounted in concrete.  A dry streambed
leads into the depression from the north.
> DOWN
You are in a small chamber beneath a 3x3 steel grate to the surface.
A low crawl over cobbles leads inward to the west.
There is a brightly shining brass lamp here.
> TAKE LAMP
Taken.
> I
You are carrying:
  a set of keys
  a brightly shining brass lamp
> WEST
You are crawling over cobbles in a low east/west passage.  There is a
dim light to the east.
> W
You are in a debris room filled with stuff washed in from the surface.
A low wide passage with cobbles becomes plugged with mud and debris
here, but an extremely narrow canyon leads upward and west.
There is a black rod with a rusty star here.  Carved on the wall
is the message: "Magic Word XYZZY".
> TAKE ROD
Taken.
> UP
You are in an awkward sloping east/west canyon.
> W
You are in a splendid chamber thirty feet high.  The walls
are frozen rivers of orange stone.  A narrow canyon and a
good passage exit from east and west sides of the chamber.
There is a little bird here.
>
```

- Implement the user commands TAKE, DROP, and INVENTORY that allow the player to work with objects. For example, the command TAKE KEYS should take the keys from the current room and add them to the player's collection, DROP KEYS should leave the keys in the current room, and INVENTORY should display the descriptions of the objects the player is carrying.
- Make it possible to create interesting puzzles by allowing the player to move through a passage only if the player is carrying some object. In the original Adventure game, for example, it is possible to go down from the room named OutsideGrate only if you are carrying the object whose name is KEYS. The required object must be included in the data file as part of the passage description. One approach is to allow the pairs of directions and destination names to include an optional third component, as in

  ```
  DOWN: BeneathGrate/KEYS
  ```

- Design and implement a mechanism for defining synonyms so that, for example, the player can use the single-letter compass points N, E, S, and W instead of having to enter the entire word.
- Add any other features that you think would make for an exciting game.

CHAPTER 13

Inheritance

[I remember the exact moment] when the concept of "inheritance" (or classes and subclasses) had been created. I realized immediately that this was the solution to a very important problem Ole-Johan Dahl and I had been struggling with for months and weeks. And sure enough, inheritance has become a key concept in object-oriented programming, and thus in programming in general.

—Kristen Nygaard, address at the IRIS 19 conference, 1996

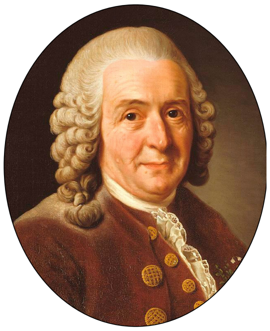
Kristen Nygaard (1926–2002)

Ole-Johan Dahl (1931–2002)

Norwegian computer scientists Kristen Nygaard and Ole-Johan Dahl developed the central ideas of object-oriented programming more than 50 years ago as part of their work on the programming language SIMULA. Early versions of SIMULA appeared in the early 1960s, but the stable version of the language that brought these concepts to the attention of the world appeared in 1967. The initial work on SIMULA was carried out at the Norwegian Computing Center, a state-funded research laboratory in Norway focusing on developing better software-engineering techniques. Both later joined the faculty at the University of Oslo. Although their work took several decades to become established in the industry, interest in object-oriented techniques has grown considerably in the last three decades, particularly after the release of modern object-oriented languages like C++ and Java. For their contributions, Nygaard and Dahl received both the 2001 Turing Award from the Association for Computing Machinery and the 2001 John von Neumann Medal from the Institute of Electrical and Electronic Engineers.

Object-oriented languages like Python are characterized by two properties: encapsulation and inheritance. Chapter 10 covers encapsulation in detail, but you have not as yet had the opportunity to learn about Python's model of ***inheritance*** in which a class acquires characteristics from other classes at higher levels in the programming analogue of a family tree. This chapter begins by introducing the concept of inheritance in the biological world and then moves on to show how the biological metaphor applies in the programming domain.

13.1 Class hierarchies



Carl Linnaeus

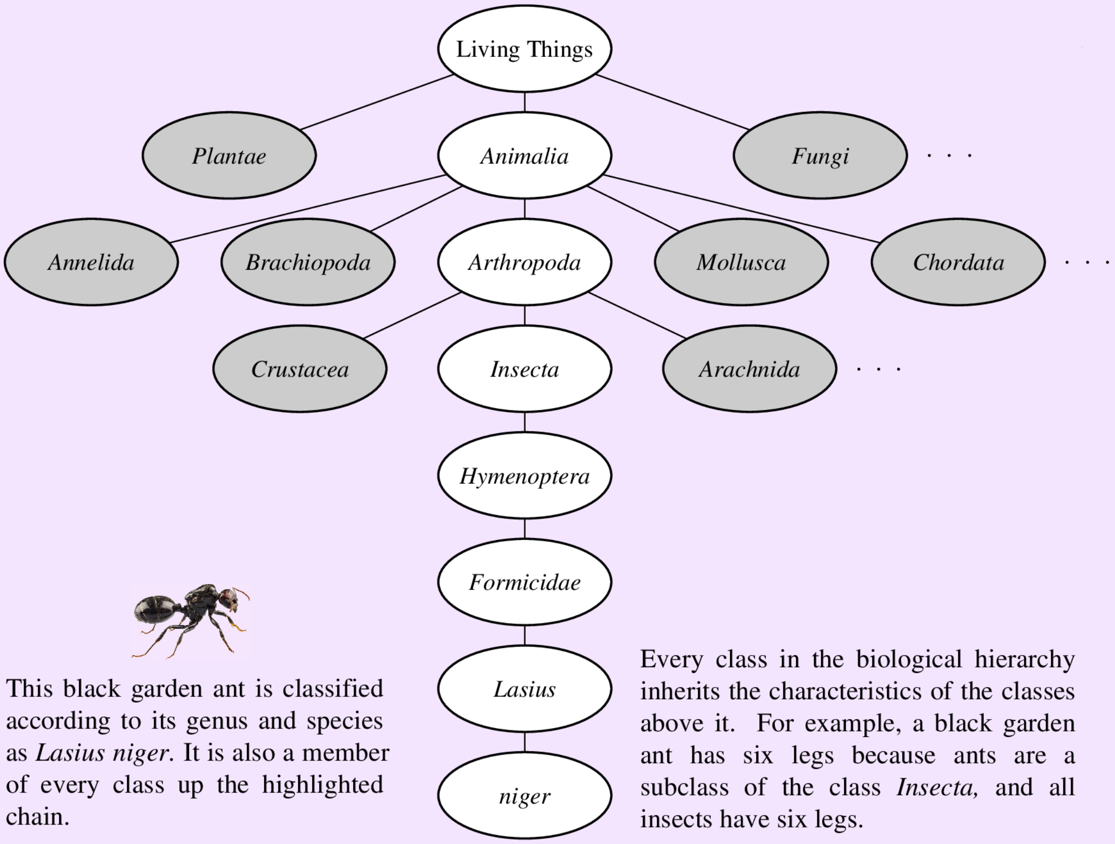
One of the defining properties of object-oriented languages is that they allow you to specify hierarchical relationships among classes. Those hierarchies are reminiscent of the biological classification system developed by the eighteenth-century Swedish botanist Carl Linnaeus as a means of representing the structure of the biological world. In Linnaeus's conception, living things are first subdivided into *kingdoms.* Each kingdom is further broken down into the hierarchical categories of *phylum, class, order, family, genus,* and *species.* Every species belongs not only to its own category at the bottom of the hierarchy but also to a category at each higher level.

This biological classification system is illustrated in Figure 13-1 at the top of the next page, which shows the classification of the common black garden ant, whose scientific name, *Lasius niger,* corresponds to its genus and species. This species of ant, however, is also part of the family *Formicidae,* which is the classification that identifies it as an ant. If you move upward in the hierarchy from there, you discover that *Lasius niger* is also of the order *Hymenoptera* (which includes bees and wasps), the class *Insecta* (which consists of the insects), and the phylum *Arthropoda* (which also includes, for example, shellfish and spiders).

One of the properties that makes this system of biological classification useful is that all living things belong to a category at every level in the hierarchy. Each individual life form therefore belongs to several categories simultaneously and inherits the properties that are characteristic of each one. The species *Lasius niger,* for example, is an ant, an insect, an arthropod, and an animal—all at the same time. Moreover, each individual ant shares the properties that it inherits from each of those categories. One of the defining characteristics of the class *Insecta* is that insects have six legs. All ants must therefore have six legs because ants are members of that class.

The biological metaphor also helps to illustrate the distinction between classes and objects. Although every common black garden ant has the same biological classification, there are many individuals of the common-black-garden-ant variety. In the language of object-oriented programming, *Lasius niger* is a class and each individual ant is an object.

FIGURE 13-1 Class hierarchies in the biological world



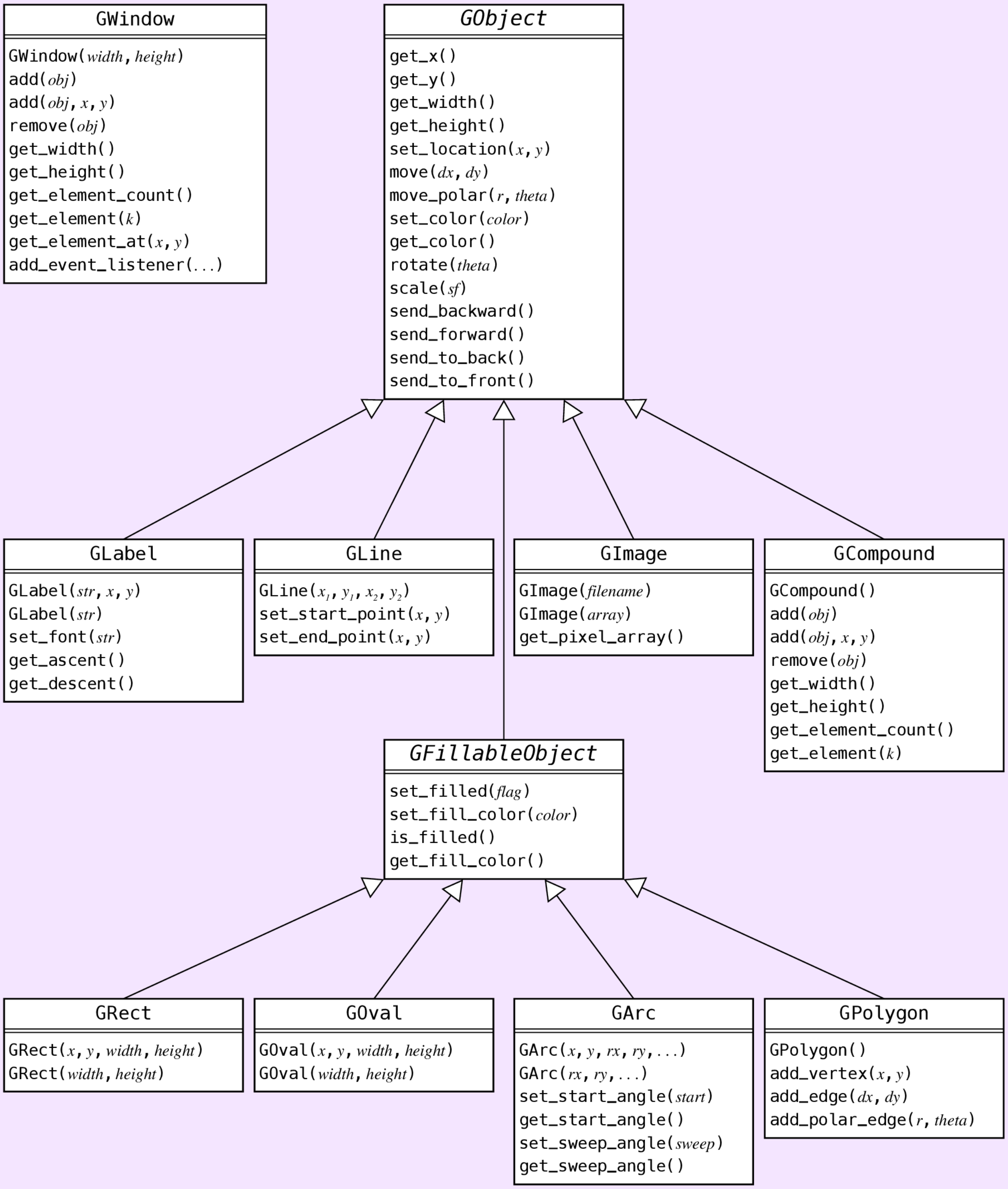
This black garden ant is classified according to its genus and species as *Lasius niger*. It is also a member of every class up the highlighted chain.

Every class in the biological hierarchy inherits the characteristics of the classes above it. For example, a black garden ant has six legs because ants are a subclass of the class *Insecta,* and all insects have six legs.

Class structures in Python follow much the same hierarchical pattern, as illustrated in Figure 13-2 at the top of the next page, which shows the relationships among the classes in the graphics library. The `GWindow` class is in a category by itself. The other class at the top of the diagram is a class called `GObject`, which you have not yet seen but in some sense have been using all along. The `GObject` class forms the top of a hierarchy that encompasses every graphical object that can be displayed in a `GWindow`. The classes that represent graphical objects are descendants of `GObject`, some directly and some through an intermediate `GFillableObject` class that encompasses the classes that have a fillable interior.

The diagram in Figure 13-2 illustrates several aspects of a standard methodology for illustrating class hierarchies called the ***Universal Modeling Language*** or ***UML.*** In a UML diagram, each class appears as a rectangular box whose upper portion

FIGURE 13-2 UML diagram for the classes in the graphics library



contains the name of the class. The methods implemented by that class appear in the lower portion of the box. The hierarchical relationships among the classes are indicated using arrows with open arrowheads that point from one class to another class at a higher level of the hierarchy. The class that appears lower in the hierarchy is a ***subclass*** of the class to which it points, which is called its ***superclass.***

In an object-oriented language, each subclass ***inherits*** the methods that apply to its superclasses all the way up through the top of the hierarchy. The GRect class, for example, inherits all the methods in GFillableObject, which in turn inherits all the methods from GObject. Given an instance of the GRect class, you can call set_fill_color because that method appears in GFillableObject. Similarly, you can call set_color, which is defined one level further up in GObject.

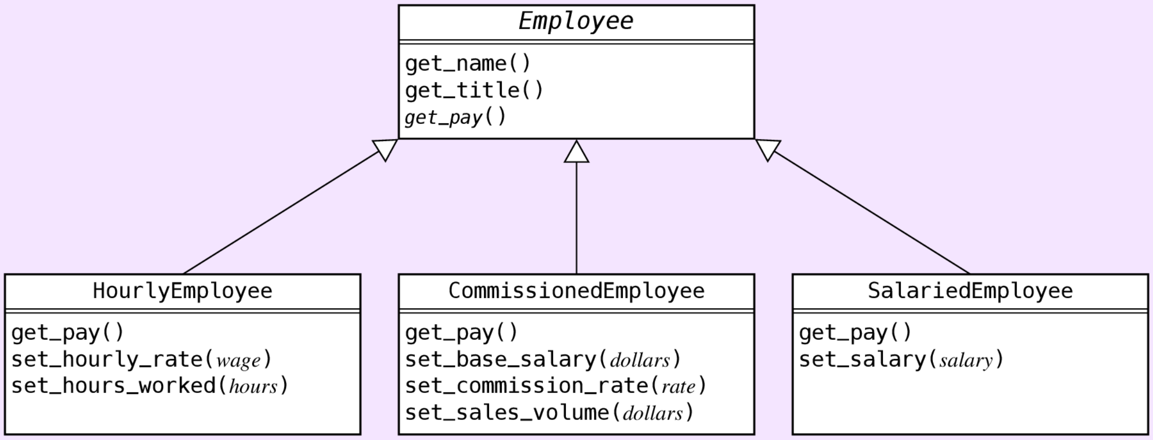
In the UML diagram in Figure 13-2, the names of the classes GObject and GFillableObject appear in italics. This notation is used to define an ***abstract class,*** which is a class that is never used to create an object but instead acts as a common superclass for ***concrete classes*** that appear beneath it in the hierarchy. Because GObject is abstract, you never create a GObject but instead create one of its concrete subclasses.

In addition to the methods it inherits from its superclass, each class in a hierarchy can implement additional methods that are specific to that class. For example, the idea of a font applies only to the GLabel class, which means that the set_font method is defined for that class and not at some higher level of the inheritance hierarchy. By contrast, the set_filled method applies only to the classes that descend from GFillableObject and not to the other GObject subclasses. It therefore makes sense to define set_filled in GFillableObject so that the definition is inherited by the GRect, GOval, GArc, and GPolygon classes.

13.2 Defining an employee hierarchy

Although the simple model for keeping track of employee data used in Chapter 10 might work for a two-person firm like Scrooge and Marley, large companies have different classes of employees that are similar in some ways but different in others. For example, a company might have hourly, commissioned, and salaried employees. Since those employee categories will share some information, it makes sense to define methods like get_name and get_title that work for all employees. By contrast, calculating the pay for each class of employee differs according to the employee type. A get_pay method must therefore be implemented separately for each subclass of Employee. This model suggests that the class hierarchy used to represent employees might look something like the UML diagram in Figure 13-3 at the top of the next page.

FIGURE 13-3 A simple class hierarchy for employees



The root of this hierarchy is `Employee`, which defines the methods common to all employees. The `Employee` class exports `get_name` and `get_title`, which the subclasses then inherit. Each subclass, however, must define its own `get_pay` method, because the computation is different. Hourly employees are paid based on the number of hours and an hourly rate, commissioned employees receive a base salary plus a commission on sales, and salaried employees receive a fixed salary.

Even though `get_pay` is defined in each subclass, it is useful to record the fact that every employee has a `get_pay` method, even though its implementation differs. The UML diagram therefore includes a `get_pay` method in the `Employee` class, even though that method is implemented at a lower level. The names of the `Employee` class and the `get_pay` method are set in italic type to indicate that these are abstract entities that act as placeholders for the concrete definitions.

Figures 13-4 and 13-5 on the next two pages define a simple `Employee` class and its `HourlyEmployee` subclass. (You will have a chance to implement the other two subclasses in exercise 1.) The `Employee` class has much the same form as it did in Figure 10-1. The structure of the subclass definition is similar, but includes the name of the superclass in parentheses after the subclass name, like this:

```
class HourlyEmployee(Employee):
```

In most cases, a subclass constructor must explicitly invoke the constructor for its superclass to ensure that the object is properly initialized. The first line of the `__init__` method for the `HourlyEmployee` subclass therefore looks like this:

```
def __init__(self, name, title):
    Employee.__init__(self, name, title)
```

FIGURE 13-4 Definition of the Employee class

```
# File: employee.py

"""
This module defines the Employee class, which is the top of a class
hierarchy with three subclasses: HourlyEmployee, CommissionedEmployee,
and SalariedEmployee.
"""

# Implementation notes:
# ---------------------
# The Employee class defines the attributes and methods that are common
# to all employees along with an abstract get_pay method that must be
# implemented in each subclass.  The individual subclasses define
# additional methods and override the get_pay method to compute the
# pay for the employee according to formulas specific to each subclass.
#
# The HourlyEmployee subclass is defined in this file; the other
# two subclasses are left as exercises.

# Class: Employee

class Employee:
    """
    This class represents the superclass of the various categories of
    employees.  It defines the methods that are common to all employees.
    """

# Constructor

    def __init__(self, name, title):
        """Initializes the common fields of an employee."""
        self._name = name
        self._title = title

# Methods

    def __str__(self):
        """Returns a string representation of this employee."""
        return self._name + " (" + self._title + ")"

    def get_name(self):
        """Returns the name of this employee."""
        return self._name

    def get_title(self):
        """Returns the job title of this employee."""
        return self._title

    def get_pay(self):
        """Returns the pay due to this employee."""
        raise NotImplementedError("Must be implemented in each subclass")
```

☞

FIGURE 13-5 Definition of the `HourlyEmployee` subclass

```
# Subclass: HourlyEmployee

class HourlyEmployee(Employee):
    """
    This subclass represents an hourly employee.
    """

# Constructor

    def __init__(self, name, title):
        """Initializes the fields of an HourlyEmployee."""
        Employee.__init__(self, name, title)
        self._hourly_rate = 0
        self._hours_worked = 0

# Methods

    def set_hourly_rate(self, wage):
        """Sets the hourly rate for this employee."""
        self._hourly_rate = wage

    def set_hours_worked(self, hours):
        """Sets the number of hours worked in the current pay period."""
        self._hours_worked = hours

# Override the default get_pay method with one specific to this subclass

    def get_pay(self):
        """Returns the pay due to this employee."""
        return self._hours_worked * self._hourly_rate
```

Because the `get_pay` method is part of the specification for the `Employee` class itself, the definition of `Employee` in Figure 13-4 also defines a `get_pay` method, which raises the built-in `NotImplementedError` exception to indicate that the `get_pay` method is not defined at the level of the `Employee` class. Fortunately, this error condition will never occur as long as the client uses the `Employee` class hierarchy correctly. Because `Employee` is an abstract class, clients will not call its constructor but will instead create one of its concrete subclasses. Each of those subclasses must replace the default version of `get_pay` with one that calculates the employee's pay appropriately. In object-oriented programming, the process of having a subclass replace an inherited method definition is called ***overriding.***

Python includes a built-in function `isinstance`, which is the preferred strategy for checking the type of a value. A call to `isinstance(`*value*`,` *type*`)` returns `True` if *value* is an instance of *type* or any of its subclasses, and `False` otherwise.

13.3 Extending the graphics classes

As you saw in Figure 13-2, the classes in the graphics library form an inheritance hierarchy in which classes like GRect, GOval, and GLabel extend a more general class called GObject. You can easily extend this hierarchy by defining new classes that build on the existing ones. For this purpose, the two classes that offer the greatest possibilities for extension are GPolygon and GCompound, and you will have the opportunity to see examples of both in the sections that follow.

Extending the GPolygon class

In a way, you have already seen examples of programs that create new GObject subclasses, although you didn't at the time have the necessary vocabulary to see them in that light. Consider, for example, the create_star function, which appears in Figure **Error! Reference source not found.**-12 on page 184. That function creates an empty GPolygon object and then adds the necessary edges to create a five-pointed star, which is then returned to the client. It is, however, equally reasonable to think of this function as a constructor for a new GPolygon subclass that appears on the graphics window as a star. Figure 13-6 at the top of the next page contains much the same code as Figure 6-12 but defines the operation as creating an instance of a new GObject subclass instead of a new graphical object.

As a subclass of GPolygon, the GStar class implements the set_filled and set_fill_color methods, which makes it possible to display a gold star outlined in black at the center of the graphics window by executing the following function:

```
def draw_outlined_gold_star():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    cx = gw.get_width() / 2
    cy = gw.get_height() / 2
    star = GStar(STAR_SIZE)
    star.set_filled(True)
    star.set_fill_color("Gold")
    gw.add(star, cx, cy)
```

Calling this function produces the following output on the graphics window:

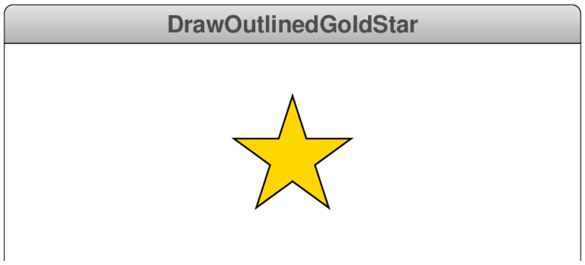


FIGURE 13-6 **Extended class representing a five-pointed star**

```
# File: GStar.py
"""
This file exports a new GObject subclass depicting a five-pointed star.
"""

from pgl import GPolygon
import math

class GStar(GPolygon):
    """
    This class represents a five-pointed star.
    """

    def __init__(self, size):
        """
        Creates a five-pointed star with its reference point at the
        center.  The size parameter indicates the width of the star
        at its widest point.
        """
        GPolygon.__init__(self)
        dx = size / 2
        dy = dx * math.tan(18 * math.pi / 180)
        edge = dx - dy * math.tan(36 * math.pi / 180)
        self.add_vertex(-dx, -dy)
        angle = 0
        for i in range(5):
            self.add_polar_edge(edge, angle)
            self.add_polar_edge(edge, angle + 72)
            angle -= 72
```

Extending the GCompound class

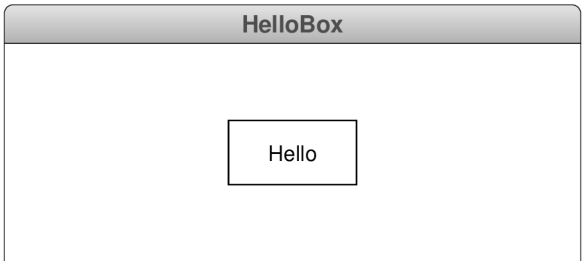
The GCompound class turns out to be an even more useful platform for designing extended classes than GPolygon because it allows you to combine several graphical objects into a single object that acts as an independent unit. As a simple example, you can extend GCompound to create a new class called GTextBox that consists of a rectangular box that includes a text string centered inside the frame. The code for the GTextBox class itself appears in Figure 13-7. Once you have defined the GTextBox class, you can use the following program to display a box containing the string "Hello" in the middle of the window:

```
def hello_box():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    cx = gw.get_width() / 2
    cy = gw.get_height() / 2
    box = GTextBox(BOX_WIDTH, BOX_HEIGHT, "Hello")
    gw.add(box, cx - BOX_WIDTH / 2, cy - BOX_HEIGHT / 2)
```

FIGURE 13-7 **Extended class representing a box containing text**

```
# File: GTextBox.py

"""
This module exports a GTextBox class that extends GCompound to display
text inside a rectangular box.
"""

from pgl import GCompound, GRect, GLabel

class GTextBox(GCompound):

    DEFAULT_FONT = "18px 'Helvetica Neue','Arial','Sans-Serif'"

    def __init__(self, width, height, text):
        """Creates a GTextBox that displays text inside a box."""
        GCompound.__init__(self)
        self.frame = GRect(width, height)
        self.label = GLabel(text)
        self.frame.set_filled(True)
        self.frame.set_fill_color("White")
        self.label.set_font(GTextBox.DEFAULT_FONT)
        self.add(self.frame)
        self.add(self.label, (width - self.label.get_width()) / 2,
                         (height + self.label.get_ascent()) / 2)

    def set_line_color(self, color):
        """Sets the line color of the frame surrounding the text."""
        self.frame.set_color(color)

    def set_fill_color(self, color):
        """Sets the fill color of the frame surrounding the text."""
        self.frame.set_fill_color(color)

    def set_text_color(self, color):
        """Sets the color of the text in the box."""
        self.label.set_color(color)
```

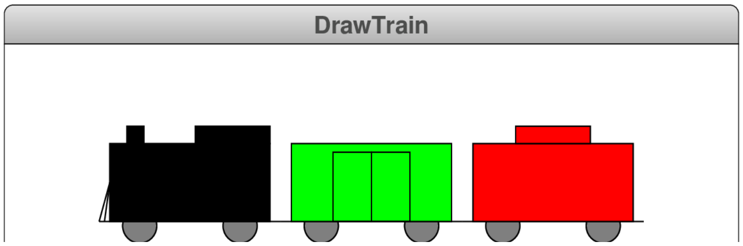
The output of this program is an 80×40 box containing the string "`Hello`" at the center of the graphics window, like this:



In addition to the constructor that creates the GCompound along with the GRect and GLabel objects it contains, the GTextBox class exports three additional methods—set_line_color, set_fill_color, and set_text_color—that control the colors of internal components. Each of these methods redirects the client's request to the graphical object that is responsible for displaying that feature. The set_line_color and set_fill_color methods pass those messages along to the GRect object stored in the local variable frame, and the set_text_color method sends the appropriate message to the GLabel stored in the variable label. Passing an operation along to an object stored inside a class is called ***forwarding.***

13.4 Decomposition and inheritance

The DrawHouse.py program in Figure 4-11 offers an illustration of how to apply the idea of decomposition to drawing a house, by dividing the program into smaller functions to draw the frame, the doors, and the windows. Suppose instead that you want to write a graphical program that creates the following picture of a three-car train consisting of a black engine, a green boxcar, and a red caboose:

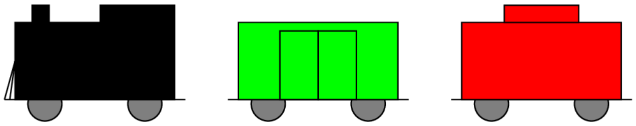


How would you go about designing such a program?

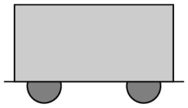
If you use a decomposition strategy similar to the one described in Chapter 4, you would implement this program by dividing it up into separate functions such as draw_engine, draw_boxcar, and draw_caboose. Each of these functions could in turn be broken down into functions that draw parts of each car, particularly when the same code can be shared by more than one type of car. That strategy, however, has a serious drawback that was not so serious in drawing a house. While houses tend to stay in one place, trains are designed to *move.* If you want to animate the train, you need to have your program change the position of *every* graphical object in the diagram on each time step. It would be much better if the train were a GCompound that you could animate as a single unit.

Fortunately, the strategy of decomposition is not limited to functions. In many cases, it is equally useful to decompose a problem by creating a hierarchy of classes whose structure reflects the relationships among the objects. For this application, it makes sense to define a Train class as a subclass of GCompound so that it acts as a single graphical object. The individual cars that form a train can then be objects of a class called `TrainCar`, which is also a subclass of `GCompound`. The three different types of train cars then become subclasses of `TrainCar`.

At this point, it helps to think carefully about the decomposition. In particular, it often makes sense to look for subtasks that recur in multiple subclasses. To see how that strategy might apply in the current problem, it's worth taking another look at the three different types of cars:



If you look at the diagrams for these three cars, you will see that they share a number of common features. The wheels are the same, as are the connectors that link the cars together. In fact, the body of the car itself is the same except for the color. Each type of car shares a common framework that looks like this:



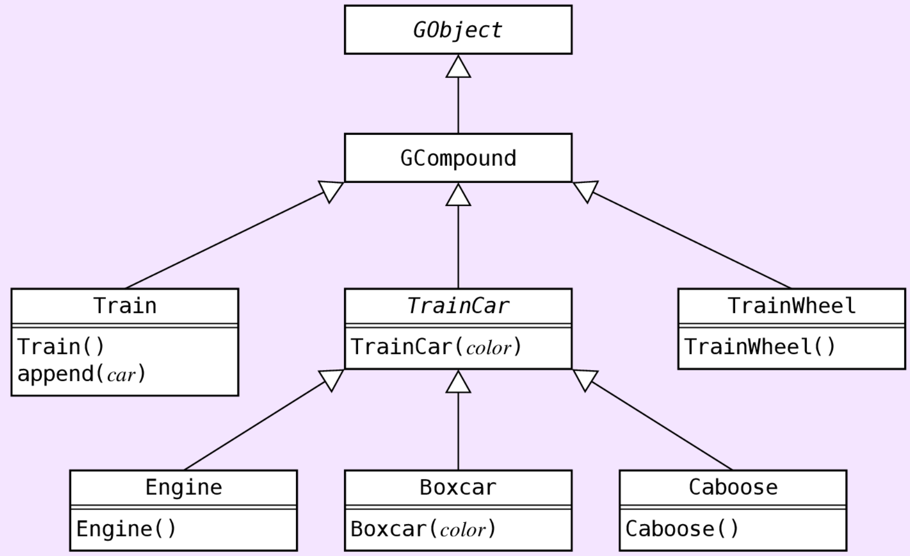
Thus, if you fill the interior of the car with the appropriate color, you can use it as the foundation for any of the three car types. For the engine, you need to add a smokestack, a cab, and a cowcatcher. For the boxcar, you need to add doors. For the caboose, you need a cupola.

To make it possible to draw cars in any color, the simplest approach is to have the `TrainCar` constructor take a `color` parameter that specifies the fill color of the gray box shown in the most recent diagram. Individual subclasses can then choose whether to make a specific decision about color, which might be that engines are always black and cabooses are always red, or to pass that choice on to the subclass. The `Boxcar` subclass, for example, can also take a `color` parameter and then pass it along to the `TrainCar` constructor.

The fact that each car has two wheels suggests that defining a `TrainWheel` class will simplify the `TrainCar` class by allowing the code for creating a wheel to be shared. Putting all these ideas together gives rise to the class hierarchy shown in Figure 13-8. Every class in the UML diagram is a `GObject` and can therefore be displayed on the graphics window.

Given this design, you can assemble the three-car train shown at the beginning of this section using the following code:

FIGURE 13-8 **UML diagram for the train classes**



```
train = Train()
train.append(Engine())
train.append(Boxcar("Green"))
train.append(Caboose())
```

The first line creates an empty train, and the remaining lines add an engine, a green boxcar, and a caboose to the end of the train. To center the train at the base of the window, you can take advantage of the fact that the `Train` class inherits the `get_width` method from `GObject`. You can therefore simply ask the train how long it is and then subtract half its width from the coordinates of the center of the window.

The `Train` object created in this snippet of code is a `GCompound` that contains every graphical object that appears in the window. If you want the train to move, all you have to do is animate the location of the `GCompound`, since all of its pieces will move together with the top-level compound.

The code to create and animate this train appears in Figure 13-9 on the next two pages. The implementation includes the class definitions for `Train`, `TrainCar`, `TrainWheel`, and `Boxcar`. You will have the chance to implement the `Engine` and `Caboose` subclasses in exercise 8.

FIGURE 13-9 Program to draw a train using a class hierarchy

```
# File: DrawTrain.js

"""
This program starts the process of drawing a three-car train consisting
of an engine, a boxcar, and a caboose.  Only the boxcar is implemented
here.  The engine and the caboose are left to the reader as an exercise.
"""

from pgl import GWindow, GCompound, GLine, GRect, GOval

# Constants

GWINDOW_WIDTH = 500           # Width of the graphics window
GWINDOW_HEIGHT = 300          # Height of the graphics window
CAR_WIDTH = 113               # Width of the frame of a train car
CAR_HEIGHT = 54               # Height of the frame of a train car
CAR_BASELINE = 15             # Distance of car base to the track
CONNECTOR = 6                 # Width of the connector on each car
WHEEL_RADIUS = 12             # Radius of the wheels on each car
WHEEL_INSET = 24              # Distance from frame to wheel center
CAB_WIDTH = 53                # Width of the cab on the engine
CAB_HEIGHT = 12               # Height of the cab on the engine
SMOKESTACK_WIDTH = 12         # Width of the smokestack
SMOKESTACK_HEIGHT = 12        # Height of the smokestack
SMOKESTACK_INSET = 12         # Distance from smokestack to front
DOOR_WIDTH = 27               # Width of the door on the boxcar
DOOR_HEIGHT = 48              # Height of the door on the boxcar
CUPOLA_WIDTH = 53             # Width of the cupola on the caboose
CUPOLA_HEIGHT = 12            # Height of the cupola on the caboose
TIME_STEP = 20                # Time step for the animation

def draw_train():

    def click_action(e):
        timer = gw.set_interval(step, TIME_STEP)

    def step():
        train.move(-1, 0)

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    train = Train()
    train.append(Boxcar("Green"))
    x = (gw.get_width() - train.get_width()) / 2
    y = gw.get_height()
    gw.add(train, x, y)
    gw.add_event_listener("click", click_action)
```

☞

FIGURE 13-9 Program to draw a train using a class hierarchy (continued)

```
class Train(GCompound):
    """This class represents an entire train"""

    def __init__(self):
        GCompound.__init__(self)

    def append(self, car):
        self.add(car, self.get_width(), 0)

class TrainCar(GCompound):
    """This class is the top of the hierarchy for all train cars"""

    def __init__(self, color):
        GCompound.__init__(self)
        r = WHEEL_RADIUS
        x = CONNECTOR
        y = -CAR_BASELINE
        self.add(GLine(0, y, CAR_WIDTH + 2 * CONNECTOR, y))
        self.add(Wheel(), x + WHEEL_INSET,  -r)
        self.add(Wheel(), x + CAR_WIDTH - WHEEL_INSET, -r)
        frame = GRect(x, y - CAR_HEIGHT, CAR_WIDTH, CAR_HEIGHT)
        frame.set_filled(True)
        frame.set_fill_color(color)
        self.add(frame)

class Wheel(GCompound):
    """This class implements a wheel on a train car"""

    def __init__(self):
        GCompound.__init__(self)
        r = WHEEL_RADIUS
        circle = GOval(-r, -r, 2 * r, 2 * r)
        circle.set_filled(True)
        circle.set_fill_color("Gray")
        self.add(circle)

class Boxcar(TrainCar):
    """This class displays a boxcar in a specified color"""

    def __init__(self, color):
        TrainCar.__init__(self, color)
        x = CONNECTOR + CAR_WIDTH / 2
        y = -(CAR_BASELINE + DOOR_HEIGHT)
        self.add(GRect(x - DOOR_WIDTH, y, DOOR_WIDTH, DOOR_HEIGHT))
        self.add(GRect(x, y, DOOR_WIDTH, DOOR_HEIGHT))

# Startup code

if __name__ == "__main__":
    draw_train()
```

13.5 Unit testing

In software engineering, a ***unit test*** is a section of code associated with a single module that checks if it is functioning correctly, independent of its connections to other modules in an application. You have been writing unit tests ever since Chapter 2 using test functions with assert statements to demonstrate that the other functions in the module produce the correct results. And although the assertion-based strategy is all you need in the context of an introductory programming course, Python supports a more sophisticated model for writing such tests through the unittest library, which uses inheritance in an interesting way.

The unittest library exports a TestCase class, which you then extend to write the unit tests for your own modules. Each subclass defines one or more test methods beginning with the prefix test. When you call unittest.main() at the end of the module, the unittest framework goes through the Python environment, finds all the subclasses of TestCase; for each of those, the framework finds all the test methods, and then calls each one to see whether it succeeds.

Each of the test methods performs its checking by calling one of the methods defined in the TestCase class, which are therefore inherited in your subclass. Figure 13-10 lists the various assert methods available in the TestCase class.

FIGURE 13-10 Methods in Python's unittest class

assertEqual(*x*, *y*)	Tests the condition *x* == *y*.
assertNotEqual(*x*, *y*)	Tests the condition *x* != *y*.
assertTrue(*x*)	Tests the condition bool(*x*).
assertFalse(*x*)	Tests the condition not bool(*x*).
assertIs(*x*, *y*)	Tests the condition *x* is *y*.
assertIsNot(*x*, *y*)	Tests the condition *x* is not *y*.
assertIsNone(*x*)	Tests the condition *x* is None.
assertIsNotNone(*x*)	Tests the condition *x* is not None.
assertIn(*x*, *y*)	Tests the condition *x* in *y*.
assertNotIn(*x*, *y*)	Tests the condition *x* not in *y*.
assertIsInstance(*x*, *y*)	Tests the condition isinstance(*x*, *y*).
assertNotIsInstance(*x*, *y*)	Tests the condition not isinstance(*x*, *y*).
assertRaises(*exception*)	Tests whether the associated with body raises *exception*.

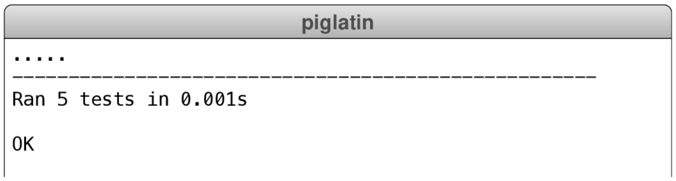
As an example, the following test program verifies a few simple properties of Python's arithmetic operators:

```
import unittest

class TestArithmeticOperators(unittest.TestCase):

    def test_arithmetic_operators(self):
        self.assertEqual(2 + 2, 4)
        self.assertEqual(10 - 5, 5)
        self.assertEqual(2 * 3, 6)
        self.assertEqual(6 / 4, 1.5)
        self.assertEqual(6 // 4, 1)
        self.assertIsInstance(4 / 2, float)
        self.assertIsInstance(4 // 2, int)
        with self.assertRaises(ZeroDivisionError):
            4 // 0

unittest.main()
```

The last two lines in the `test_arithmetic_operators` method illustrate the standard usage pattern for the `assertRaises` method, which checks whether the code inside the `with` body raises the specified exception type.

You have probably noticed that the names of the various `assert` methods in the `unittest` library use camel-case names instead of the standard snake-case style that Python coders prefer. Python's guidelines offer at least some flexibility on this point. Camel-case function and method names are allowed "in contexts where that's already the prevailing style to retain backwards compatibility." Python's `unittest` library is adapted from Java's `JUnit` library developed by Kent Beck and Erich Gamma, which is in turn derived from a similar package in Smalltalk. Those packages for other languages use camel-case names, and maintaining that style makes it easier to implement unit testing in multiple languages.

The code in Figure 13-11 on the next page offers a more extensive example in the form of a test suite for the Pig Latin translator. The unit test appears entirely within the startup boilerplate, which ensures that none of the test code—including the import of the `unittest` library—is executed unless the module is run from the command line, which triggers the test operation. Running the `piglatin.py` module as a main program generates the following output:



```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

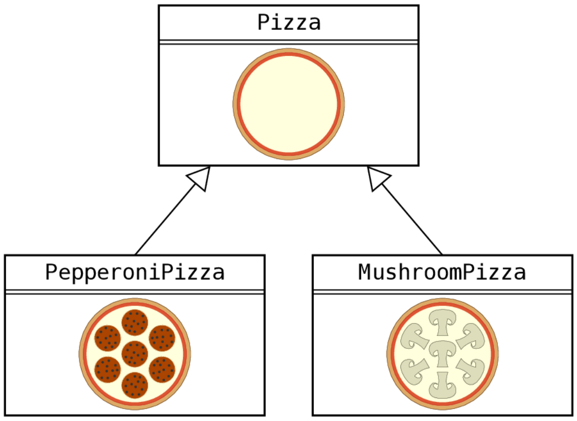
FIGURE 13-11 Changes to the Pig Latin program to accommodate unit testing

```
# File: piglatin.py

# This version of the piglatin.py module includes a unit test.
# All other functions remain the same.

def word_to_pig_latin(word):
    """Translates a word to Pig Latin."""
    if not word.isalpha():
        raise ValueError("Word is not alphabetic")
    vp = find_first_vowel(word)
    if vp == -1:
        return word
    elif vp == 0:
        return word + "way"
    else:
        head = word[0:vp]
        tail = word[vp:]
        return tail + head + "ay"

# Unit test

if __name__ == "__main__":

    import unittest

    class TestPigLatin(unittest.TestCase):

        def test_sentences(self):
            self.assertEqual(to_pig_latin("this is pig latin."),
                             "isthay isway igpay atinlay.")

        def test_starts_with_consonant(self):
            self.assertEqual(word_to_pig_latin("pig"), "igpay")
            self.assertEqual(word_to_pig_latin("latin"), "atinlay")
            self.assertEqual(word_to_pig_latin("trash"), "ashtray")

        def test_starts_with_vowel(self):
            self.assertEqual(word_to_pig_latin("is"), "isway")
            self.assertEqual(word_to_pig_latin("express"), "expressway")

        def test_has_no_vowels(self):
            self.assertEqual(word_to_pig_latin("tsktsks"), "tsktsks")

        def test_illegal_word(self):
            with self.assertRaises(ValueError):
                word_to_pig_latin("")
            with self.assertRaises(ValueError):
                word_to_pig_latin("123")

    unittest.main()
```

13.6 Deciding when to use inheritance

Although inheritance is a powerful concept that is ideal for many applications, it can easily be overused. Before deciding to implement a class hierarchy, it is important to think carefully about whether that design is appropriate. Quite often, it is not.

The best illustration I can offer of where inheritance is misused comes from several textbooks that suggest the example of a “pizza” class hierarchy, presumably because some students pay more attention when pizza is involved. In this model, the base of the hierarchy is a Pizza class, which takes care of those features—such as a crust, a tomato base, and cheese—that are common to all pizzas, or at least to those pizzas supported by the model. Different types of pizzas are then represented as subclasses of the Pizza class. For example, if your list of available toppings included pepperoni and mushrooms, you might envision a class hierarchy like this:



So far, so good. I can believe that pepperoni pizzas and mushroom pizzas are both subclasses of a more general Pizza class. The situation, however, gets more complicated if your customers want pizzas with more than one ingredient. What happens if someone orders a pizza with pepperoni *and* mushrooms. In languages like Python that support multiple inheritance (which is beyond the scope of this book), you might try to extend the class hierarchy by defining a PepperoniMushroomPizza class that inherits from both the PepperoniPizza and MushroomPizza classes.

The problem with this strategy is that the PepperoniMushroomPizza class is not really a subclass of those parents. The subclass relationship should allow you to substitute the words “is a” in place of the subclass arrow. In the Portable Graphics Library, for example, a GRect is a GFillableObject, which in turn is a GObject. Similarly, an ant in the biological hierarchy shown in Figure 13-1 is an insect and an animal. If for no other reason than the fact that it is no longer vegetarian, a pepperoni-and-mushroom pizza is not a mushroom pizza, no matter how you slice it.

What you really want to represent a range of different pizzas is not a hierarchy at all but instead a single `Pizza` class that includes a list of ingredients as part of its state. The `Pizza` class could then simply export an `add_topping` method that would allow the client to add any number of toppings to the base. Inheritance is simply out of place in this example.

As an example that stands in contrast to the misguided attempt to create a pizza hierarchy, consider how you might build a hierarchy of the various droids that exist in the *Star Wars* universe. The original movie introduced two lovable droids, R2-D2 and C-3PO, who provided a sense of continuity by appearing in all of the nine films that made up George Lucas's original vision. The two droids, however, are different in many ways. C-3PO introduces itself as a "cyborg" in that first film, and the many *Star Wars* sites on the web inform us that R2-D2 is an "astromech." In later films, we meet other droids in each of these classes. *The Force Awakens* brings us into contact with another astromech, the roly-poly BB-8. *Rogue One* introduces a cyborg with the designation K-2SO.

These droid subclasses have different behavior. Cyborgs can communicate in human languages, while astromechs communicate in a binary language of beeps and whistles. Unsurprisingly, give their structural resemblance to humans, droids walk upright on two legs, while astromechs exhibit a variety of strategies for movement.

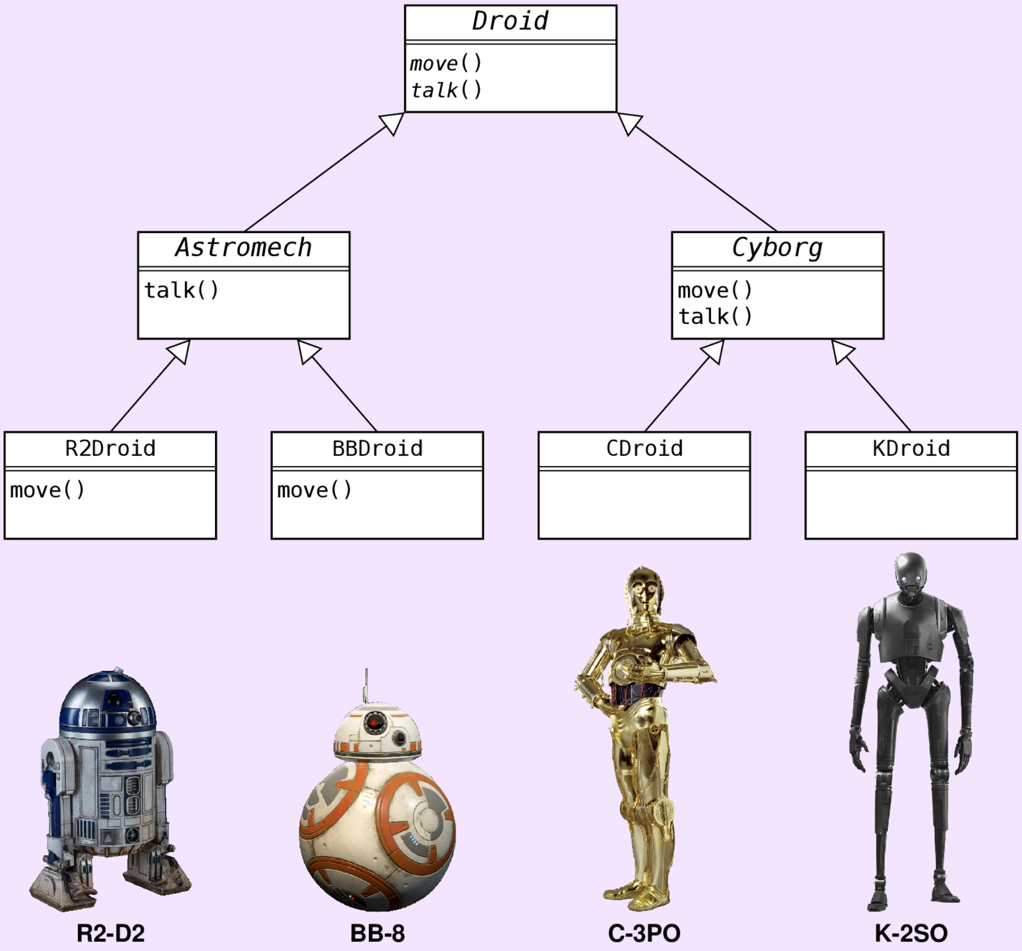
The fact that these behaviors are often associated with a class of droid rather than a specific model makes using a class hierarchy much more appropriate. All droids "talk" in some fashion, but that communication is implemented differently for cyborgs and astromechs suggests that those two classes would require a different implementation of a `talk` method. And while a single implementation of a `move` method could presumably be shared by all cyborgs, that method would have to be specified at a lower level to account for R2-D2's strategy of rotating and then rolling forward and BB-8's more flexible model of rolling in any direction.

Figure 13-12 at the top of the next page suggests a hierarchy for droids that accommodates both their differences and their commonalities. All droids respond to the methods `move` and `talk`, even though the implementations of those methods appear at different levels in the hierarchy.

As a general rule, inheritance makes sense when the classes you are working with have the following two properties:

1. The classes exhibit a clear hierarchical structure in which the "is a" relationship holds between every subclass and its parent.
2. You can identify shared behavior at higher levels of the hierarchy, which makes it appropriate to define methods that can then be inherited by the subclasses.

FIGURE 13-12 Droid hierarchy from the *Star Wars* universe



Summary

This chapter includes a brief introduction to the idea of inheritance in Python along with some appropriate examples. Important points in this chapter include the following:

- Classes in an object-oriented language form hierarchies in which classes at lower levels *inherit* the methods defined by the classes above them in the hierarchy.
- The immediate descendants of a class are called its *subclasses.* The immediate ancestor of a class is called its *superclass.*

- Classes that form part of the inheritance hierarchy but do not correspond to any actual objects are called *abstract classes.*
- The *Universal Modeling Language* or *UML* provides a notational structure for representing the relationships in a class hierarchy. Each subclass in a UML diagram is connected to its superclass using an arrow with an open arrowhead.
- The graphics library presented in Chapters 4 and 6 uses the class hierarchy shown in the UML diagram in Figure 13-2 on page 360. That hierarchy includes two abstract classes—`GObject` and `GFillableObject`—that serve to unify graphical objects that share a set of common operations.
- You can implement a subclass in Python by including the name of its superclass in parentheses as part of the class definition. The constructor for a subclass should begin by calling the constructor of the superclass.
- The best way to check the type of a value is to call `isinstance(`*value*`,` *type*`)`, which returns `True` if *value* is an instance of *type* or any of its subclasses.
- The classes in the graphics library, especially `GPolygon` and `GCompound`, offer useful starting points for inheritance relationships as illustrated by the `GStar`, `GTextBox`, and `DrawTrain` programs in Figures 13-6, 13-7, and 13-9.
- Inheritance allows you to apply the principles of top-down design and stepwise refinement in the data domain.
- Testing the behavior of a single module independently is called *unit testing.* Python supports unit testing through the `unittest` library. Clients of the `unittest` library write a suite of test methods as part of a class that extends `unittest.TestCase`. The test methods use the various `assert` methods listed in Figure 13-10 to check that the module is producing correct results.
- Inheritance can easily be overused. In general, class hierarchies make sense only when the classes exhibit a clear hierarchical structure and when you can identify shared behavior that can be inherited by classes at lower levels in the hierarchy.

Review questions

1. Choose a favorite animal and add it to the biological hierarchy in Figure 13-1. To find the appropriate place in the hierarchy, you will need to use the web to look up its phylum, class, order, family, genus, and species.
2. Define the terms *subclass* and *superclass.*
3. True or false: A subclass inherits the methods in its superclass along with those of all its superclasses in the inheritance hierarchy.
4. What does *UML* stand for?

5. How is the relationship between subclasses and superclasses represented in a UML diagram?

6. How can you determine what methods a class in a UML diagram supports?

7. What is an *abstract class?*

8. In your own words, explain the purpose of the `GFillableObject` class in the graphics hierarchy shown in Figure 13-2.

9. The implementation of `get_pay` in the `Employee` class signals failure by raising an error exception. What keeps this error from occurring if the client uses the employee hierarchy correctly?

10. When you are defining a subclass in Python, how do you specify its superclass?

11. How does a subclass trigger the initialization of its superclass?

12. What term is used to describe the process of providing a new definition for a method to replace one defined in a superclass?

13. What built-in Python method should you use to determine whether a value is an instance of a particular type?

14. Which two classes in the Portable Graphics Library are most likely to serve as the basis for extension?

15. What is meant by the term *forwarding?*

16. The `Train` class in Figure 13-9 exports a method called `append` to add a car to the end of the train. Would it have worked just as well to use `add` as the name of this method?

17. How does unit testing differ from any other kind of testing?

18. What class in the `unittest` library forms the foundation of the unit tests you write for your own modules?

19. How do you check that a computation raises an expected exception in a test method constructed using the `unittest` library?

20. Why does the `unittest` library use camel case for the `assert` methods instead of snake case, which is more conventional in Python?

21. True or false: Inheritance is often overused in programming applications.

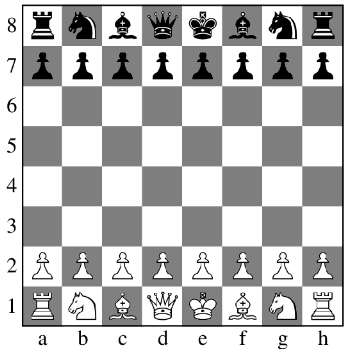
22. What two criteria does the chapter suggest for determining whether inheritance is an appropriate strategy when defining a data structure?

Exercises

1. Complete the definition of the Employee hierarchy from Figures 13-4 and 13-5 by defining CommissionedEmployee and SalariedEmployee.

2. Inheritance comes up naturally in many games. If you are writing a chess program, for example, you can represent the pieces by defining an abstract ChessPiece class along with the subclasses King, Queen, Rook, Bishop, Knight, and Pawn for the different piece types, as shown in The ChessPiece class keeps track of the color and location of the piece. Each of the subclasses extends ChessPiece by implementing the moves for that piece.

   The following diagram shows the pieces in their initial locations:



   Chess players identify each square on the board using a two-character string that combines a letter indicating the column and a digit indicating the row. For example, the white queen is initially on square **d1**.

FIGURE 13-13 UML diagram for the ChessPiece hierarchy

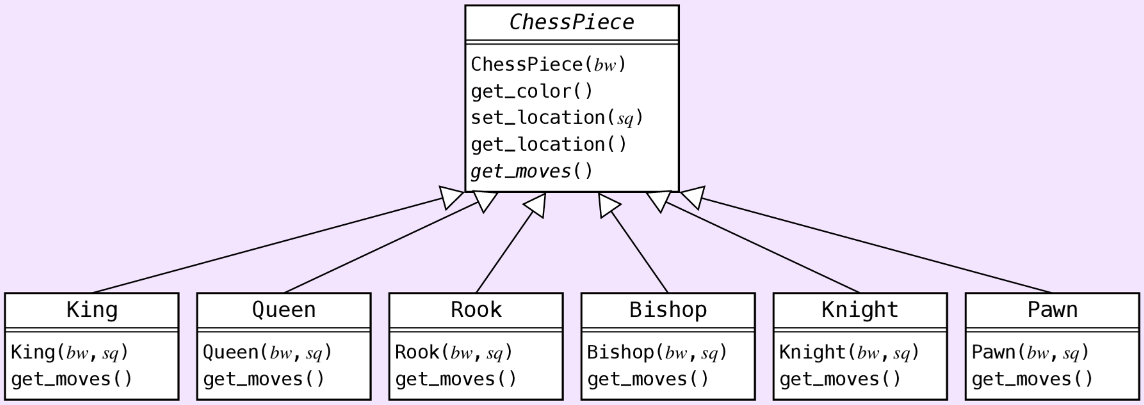
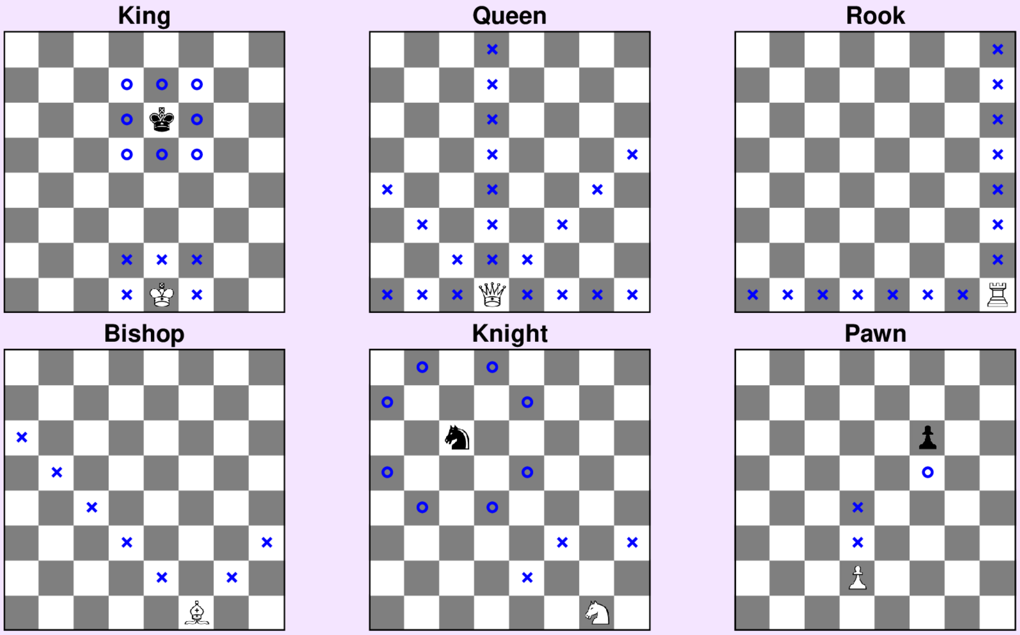
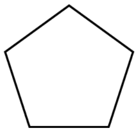


FIGURE 13-14 Moves for the chess pieces



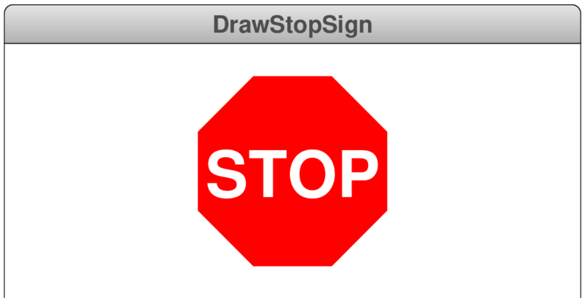
Implement the classes shown in the UML diagram in Figure 13-13. The constructors for the concrete classes take an argument `bw`, which is either `"B"` or `"W"`, and an argument `sq`, which is the two-character string identifying the square. The `get_moves` method should return an array of all the two-character locations to which that piece could move from its current square, assuming that the rest of the board were empty. Figure 13-14 shows how the different pieces move, in case you are unfamiliar with the rules of chess. The white pieces can move to any of the squares marked with an ×, and the black pieces can move to any square marked with an O. The white pawn in the last diagram can move either one or two squares because it is in its initial position on row 2, but the black pawn can only move one square because it not in its starting position.

3. Create a new `GRegularPolygon` class that extends `GPolygon` so that it is easy to represent a ***regular polygon,*** which is a polygon whose sides all have the same length and whose angles are equal. The `GRegularPolygon` constructor should take two parameters: `sides`, which indicates the number of sides, and `radius`, which indicates the distance from the reference point at the center to any of its vertices. The polygon should be oriented so that it is flat along the bottom. For example, calling `GRegularPolygon(5, 25)` should create a `GRegularPolygon` object that looks like this:



Similarly, calling `GRegularPolygon(200, 25)` should create a 200-sided polygon whose appearance—at least at the scale of the graphics window—is indistinguishable from that of a circle of radius 25.

4. Use the `GRegularPolygon` class from the preceding exercise to create a `GStopSign` class that create a picture that looks like this:



5. Extend the `GTextBox` class from Figure 13-7 so that it exports a `set_font` method that resets the font used for the text string. Because changing the font typically changes the dimensions of the label, the implementation of `set_font` will need to adjust the position of the label within the box.

6. Implement a `GCompound` subclass called `GVariable` class that makes it easy to draw box diagrams of variables on the graphics window. The methods implemented for `GVariable` appear in Figure 13-15. The reference point for the `GVariable` should be the upper left corner of the variable box.

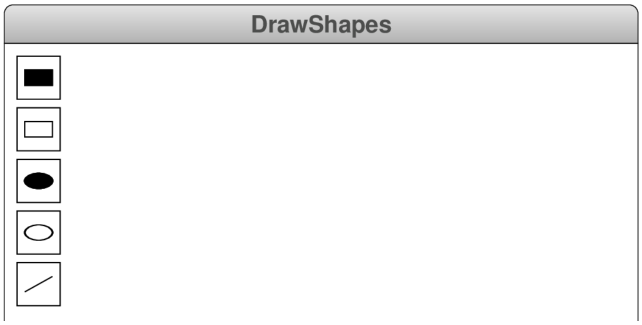
FIGURE 13-15 Methods implemented by the `GVariable` class

`GVariable(`*name*`)`
`GVariable(`*name*`,` *width*`)`
`GVariable(`*name*`,` *width*`,` *height*`)`
	Creates a new `GVariable` object with the specified name and dimensions, for which the constructor should supply suitable default values. The initial value of a `GVariable` is `None`, which should appear as blank space inside the variable box.

gvar`.get_name()`	Returns the name of the `GVariable`.
gvar`.set_value(`*value*`)`	Sets the value of the variable. The *value* parameter can be of any type, and its string representation should then appear in the center of the variable box.
gvar`.get_value()`	Returns the most recent value to which the variable was set.
gvar`.set_font(font)`	Sets the font of the internal `GLabel` used to display the variable.

7. Complete the implementation of the `DrawTrain.py` program in Figure 13-9 by writing definitions for the `Engine` and `Caboose` classes. Update the main program so that the train includes an engine, a green boxcar, an orange boxcar, and a caboose.

8. In Chapter 6, you had the chance to work with several programs that let you create shapes by dragging the mouse on the graphics window. Using those programs as a starting point, create a more elaborate `DrawShapes.py` program that displays an onscreen menu of five shapes—a filled rectangle, an outlined rectangle, a filled oval, an outlined oval, and a straight line—along the left side of the window, as shown in the following diagram:



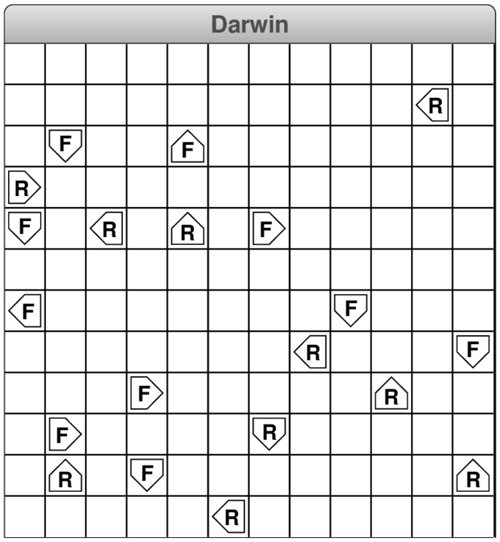
Clicking one of the squares in the menu chooses that shape as a drawing tool. Thus, if you click the filled oval in the middle of the menu area, your program should draw filled ovals. Clicking and dragging outside the menu should draw the currently selected shape.

9. Use the `unittest` library to update the testing code in the `quadratic.py` module presented in Figure 3-10. You should also change the implementation of `find_quadratic_roots` so that it raises a `ValueError` exception if the equation has no real roots and then check for that behavior in your unit test code.

   Use the `unittest` library to write a comprehensive unit test for your revised implementation of `solve_quadratic`.

10. Use the `unittest` library to write a unit test for the `rational.py` module that appears in Figure 10-6 on page 338. The hard part of this exercise consists of designing a good set of test methods that cover enough possibilities to give both the implementers and the clients of the `rational` library confidence that it is behaving correctly.

11. In the 1990s, my Stanford colleague Nick Parlante developed a wonderful simulation game that not only involves inheritance but also pays tribute to the evolutionary metaphor from which the idea of inheritance is derived. The Darwin game operates in a rectangular grid that looks like this:



This sample world is populated with twenty creatures, ten of a species called *Flytrap* and ten of a species called *Rover*, each of which is identified by the first letter in its name. The orientation is indicated by the shape surrounding the letter; the creature points in the direction of the arrow. Each creature runs a program that is particular to its species. Thus, all Rovers behave in the same way, as do all Flytraps, but the behavior of one species is different from that of the other.

As the simulation proceeds, every creature gets a turn. On its turn, a creature executes one of the actions shown in the first section of Figure 13-16 at the top of the next page. As soon as one of these actions is completed, the turn for that creature ends. When every creature has had a turn, the process begins again.

If one creature is facing another creature of a different species in the next square, the first creature can "infect" the second, which turns the infected creature into an instance of the infecting one. The goal for each species in the Darwin game is to infect as many creatures as possible.

The program for each species is represented as an array of strings, each of which is one of the statements in Figure 13-16. The program for the Flytrap creature, for example, consists of the following five-element array:

```
[                         # Index
    "if_different 3",     #   0
    "turn_right",         #   1
    "goto 0",             #   2
    "infect",             #   3
    "goto 0"              #   4
]
```

FIGURE 13-16 Instructions for programming Darwin creatures

Darwin actions

`move`	The creature moves forward as long as the square it is facing is empty. If the creature is blocked, the `move` instruction does nothing.
`turn_left`	The creature turns left 90 degrees.
`turn_right`	The creature turns right 90 degrees.
`infect`	If the square immediately in front of this creature is occupied by a different species that creature is “infected” so that it becomes an instance of this species. The old creature is removed and replaced by a new creature in the same orientation.

Control instructions

`goto` *n*	Take the next instruction from index *n* in the program array. Like all control actions, executing a `goto` instruction does not count as a turn.
`if_empty` *n*	If the square in front of this creature is unoccupied, take the next instruction from index *n* in the program array. If not, continue on to the next instruction.
`if_wall` *n*	If this creature is facing one of the border walls, take the next instruction from index *n* in the program array. If not, continue on to the next instruction.
`if_same` *n*	If the square in front of this creature is occupied by the same species, take the next instruction from index *n* in the program array. If not, continue on to the next instruction.
`if_different` *n*	If the square in front of this creature is occupied by a different species, take the next instruction from index *n* in the program array. If not, continue on to the next instruction.
`if_random` *n*	Randomly choose with equal probability whether to take the next instruction from index *n* in the program array or to continue on to the next instruction.

Each creature instance keeps track of the index of the current instruction in this program, which always begins at 0 when a creature is created. On a turn, the creature executes instructions until one of the Darwin actions occurs. The Flytrap creature, for example, begins by executing the `"if_different 3"` instruction at index 0 in the array. If the Flytrap is facing a creature of a different species, it goes to the `"infect"` instruction at index 3 and executes that operation. If the `if_different` instruction does not apply, the creature continues with the `"turn_right"` instruction at index 1. In either case, this creature’s turn ends after executing the action. On its next turn, the creature begins by executing one of the `"goto 0"` instructions, which sends the program back to the top. The Flytrap creature therefore rotates clockwise until it sees a creature of a different species, in which case it infects it to make it a Flytrap.

Your job in this exercise is to implement both the `Darwin` file that runs the simulation and the `Creature` class, which is the abstract superclass of `Flytrap` and `Rover` as well as any other creatures you design. The definitions of these two subclasses appear in Figure 13-17. Your definition of `Creature` must provide the methods used in these subclasses. The `Creature` class is also responsible for managing the display of the creature on the graphics window, which is most easily accomplished by making `Creature` a subclass of `GCompound`.

FIGURE 13-17 Code for the `Flytrap` and `Rover` subclasses

```
# This class implements the Flytrap creature.  A Flytrap never moves
# but instead spins clockwise, infecting any creatures it sees.

class Flytrap(Creature):

    def __init__(self):
        Creature.__init__(self, "F", FLYTRAP_PROGRAM)

    FLYTRAP_PROGRAM = [      # Index
        "if_different 3",    #   0
        "turn_right",        #   1
        "goto 0",            #   2
        "infect",            #   3
        "goto 0"             #   4
    ]

# This class implements the Rover creature.  A Rover constantly moves
# forward.  When it runs into a wall or another Rover, it randomly
# turns left or right.

class Rover(Creature):

    def __init__(self):
        Creature.__init__(self, "R", ROVER_PROGRAM)

    ROVER_PROGRAM = [        # Index
        "if_different 10",   #   0
        "if_wall 5",         #   1
        "if_same 5",         #   2
        "move",              #   3
        "goto 0",            #   4
        "if_random 9",       #   5
        "turn_left",         #   6
        "goto 0",            #   7
        "turn_right",        #   8
        "goto 0",            #   9
        "infect",            #  10
        "goto 0"             #  11
    ]
```

The main Darwin program is responsible for the following actions:

- Setting up the graphics window and drawing the grid
- Initializing the grid by creating 10 creatures of each species and positioning them randomly in the grid facing in a random direction
- Iterating through the grid giving each creature a turn

The most interesting part of this problem is designing new creatures that perform well in the survival-of-the-fittest challenge the Darwin game provides.

12. Many applications require some part of the implementation to read commands from the user and then perform some operation in response. If, for example, you implemented the Adventure game that appeared as exercise 7 in Chapter 12, you needed to implement commands such as LOOK, TAKE, and DROP.

As long as the number of commands you need to recognize is small, you can implement the task of interpreting commands using a sequence of if and elif statements that compare the command word entered by the user against the names of the various commands, like this:

```
if word.upper() == "LOOK":
    execute_look_command()
elif word.upper() == "TAKE":
    execute_take_command()
elif word.upper() == "DROP":
    execute_drop_command()
. . . and so on . . .
```

This kind of control structure is called a ***command dispatch.***

A more elegant approach is to use inheritance to create a class hierarchy of commands, particularly if several related commands are similar enough that they can share parts of their implementation. The abstract class Command represents the top of the hierarchy. Clients of the command-dispatch library then define subclasses, each of which knows how to execute that particular type of command. In the case of the Adventure game, for example, you might define concrete subclasses called LookCommand, TakeCommand, and DropCommand. All instances of the Command class and its subclasses define a method called execute that performs the necessary operation. If you put each Command object into a dictionary with its name as the key, all you need to do to implement the command-dispatch operation is select its name from the dictionary and call its execute method.

Reimplement the Adventure game (exercise 7 from Chapter 12) using this model to execute the predefined commands.