Fundamentals of

Programming
Python

DRAFT

Richard L. Halterman
Southern Adventist University

Contents

Preface

Chapter 1

The Context of Software Development

A computer program, from one perspective, is a sequence of instructions that dictate the ?ow of electrical impulses within a computer system. These impulses affect the computer?s memory and interact with the display screen, keyboard, mouse, and perhaps even other computers across a network in such a way as to produce the ?magic? that permits humans to perform useful tasks, solve high-level problems, and play games. One program allows a computer to assume the role of a ?nancial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Powerful software construction tools hide the lower-level details from programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and ef?ciency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language. While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as software engineers develop software to drive particular systems. These systems are de?ned by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and pro?lers. This chapter examines the context of software development, including computer systems and tools.

A computer program is an example of computer software. One can refer to a program as a piece of software as if it were a tangible object, but software is actually quite intangible. It is stored on a medium. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used, software must be stored in the computer?s memory. Typically computer programs are loaded into memory from a medium like the computer?s hard disk. An electromagnetic pattern representing the program is stored on the computer?s hard drive. This pattern of electronic symbols must be transferred to the computer?s memory before the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

To the underlying computer hardware, speci?cally the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that it makes a certain part of the display screen red. Unfortunately, only a minuscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word for an Intel-based computer running the Windows 8.1 operating system. Further, almost none of those who could produce the binary sequence would claim to enjoy the task.

The Word program for older Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Core i7 in the Windows machine accepts a completely different binary language than the PowerPC processor in the older Mac. We say the processors have their own machine language.

Software can be represented by printed words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like Python allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, C++, Java, and C#. Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language.

One might think that ideally such a conversion tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called compilers that translate one computer language into another have been around for over 60 years, but natural language processing is still an active area of arti?cial intelligence research. Natural languages, as they are used by most humans, are inherently ambiguous. To understand properly all but a very limited subset of a natural language, a human (or arti?cially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today?s software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any problem that can be solved by a computer.

While these three lines do constitute a proper Python program, they more likely are merely a small piece of a larger program. The lines of text in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, subtotal, tax, and total, called variables, represent information. Mathematicians have used variables for hundreds of years before the ?rst digital computer was built. In programming, a variable represents a value stored in the computer?s memory. Instead of some cryptic binary instructions meant only for the processor, we see familiar-looking mathematical operators (= and +). Since this program is expressed in the Python language, not machine language, no computer processor can execute the program directly. A program called an interpreter translates the Python code into machine code when a user runs the program. The higher-level language code is called source code. The corresponding machine language code is called the target code. The interpreter translates the source code into the target machine language.

? Interpreters. An interpreter is like a compiler, in that it translates higher-level source code into target code (usually machine language). It works differently, however. While a compiler produces an executable program that may run many times with no additional translation needed, an interpreter translates source code statements into machine language each time a user runs the program. A compiled program does not need to be recompiled to run, but an interpreted program must be reinterpreted each time it executes. For this reason interpreted languages are often refered to as scripting languages. The interpreter in essence reads the script, where the script is the source code of the program. In general, compiled programs execute more quickly than interpreted programs because the translation activity occurs only once. Interpreted programs, on the other hand, can run as is on any platform with an appropriate interpreter; they do not need to be recompiled to run on a different platform. Python, for example, is used mainly as an interpreted language, but compilers for it are available. Interpreted languages are better suited for dynamic, explorative development which many people feel is ideal for beginning programmers. Popular scripting languages include Python, Ruby, Perl, and, for web browsers, Javascript.

? Debuggers. A debugger allows a programmer to more easily trace a program?s execution in order to locate and correct errors in the program?s implementation. With a debugger, a developer can simultaneously run a program and see which line in the source code is responsible for the program?s current actions. The programmer can watch the values of variables and other program elements to see if their values change as expected. Debuggers are valuable for locating errors (also called bugs) and repairing programs that contain errors. (See Section 3.6 for more information about programming errors.)

? Pro?lers. A pro?ler collects statistics about a program?s execution allowing developers to tune appropriate parts of the program to improve its overall performance. A pro?ler indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Developers also can use pro?lers for testing purposes to ensure all the code in a program is actually being used somewhere during testing. This is known as coverage. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing. The main purpose of pro?ling is to ?nd the parts of a program that can be improved to make the program run faster.

Python is used for software development at companies and organizations such as Google, Yahoo, Facebook, CERN, Industrial Light and Magic, and NASA. Experienced programmers can accomplish great things with Python, but Python?s beauty is that it is accessible to beginning programmers and allows them to tackle interesting problems more quickly than many other, more complex languages that have a steeper learning curve.

This book does not attempt to cover all the facets of the Python programming language. Experienced programmers should look elsewhere for books that cover Python in much more detail. The focus here is on introducing programming techniques and developing good habits. To that end, our approach avoids some of the more esoteric features of Python and concentrates on the programming basics that transfer directly to other imperative programming languages such as Java, C#, and C++. We stick with the basics and explore more advanced features of Python only when necessary to handle the problem at hand.

This is a Python statement. A statement is a command that the interpreter executes. This statement prints the message This is a simple Python program on the screen. A statement is the fundamental unit of execution in a Python program. Statements may be grouped into larger chunks called blocks, and blocks can make up more complex statements. Higher-order constructs such as functions and methods are composed of blocks. The statement

Figure 1.8 shows that WingIDE 101 displays a program?s output as black text on a white background. In order to better distinguish visually in this text program source code from program output, we will render the program?s output with white text on a black background, as it would appear in the command line interpreter under Windows as shown in Figure 1.12. This means we would show the output of Listing 1.1 (simple.py) as

If you try to enter each line one at a time into the interactive shell, the program?s output will be intermingled with the statements you type. In this case the best approach is to type the program into an editor, save the code you type to a ?le, and then execute the program. Most of the time we use an editor to enter and run our Python programs. The interactive interpreter is most useful for experimenting with small snippets of Python code.

In Listing 1.2 (arrow.py) each print statement ?draws? a horizontal slice of the arrow. All the horizontal slices stacked on top of each other results in the picture of the arrow. The statements form a block of Python code. It is important that no whitespace (spaces or tabs) come before the beginning of each statement. In Python the indentation of statements is signi?cant and the interpreter generates error messages for improper indentation. If we try to put a single space before a statement in the interactive shell, we get

Chapter 2

Values and Variables

beginning of a string, and the right ' symbol that follows speci?es the end of the string. If a single quote marks the beginning of a string value, a single quote must delimit the end of the string. Similarly, the double quotes, if used instead, must appear in pairs. You may not mix the two kinds of quotation marks when used to delimit a particular string, as the following interactive sequence shows:

It is important to note that the expressions 4 and '4' are different. One is an integer expression and the other is a string expression. All expressions in Python have a type. The type of an expression indicates the kind of expression it is. An expression?s type is sometimes denoted as its class. At this point we have considered only integers and strings. The built in type function reveals the type of any Python expression:

In mathematics, integers are unbounded; said another way, the set of mathematical integers is in?nite. In Python, integers may be arbitrarily large, but the larger the integer, the more memory required to represent it. This means Python integers theoretically can be as large or as small as needed, but, since a computer has a ?nite amount of memory (and the operating system may limit the amount of memory allowed for a running program), in practice Python integers are bounded by available memory.

This is an assignment statement. An assignment statement associates a value with a variable. The key to an assignment statement is the symbol = which is known as the assignment operator. The statement assigns the integer value 10 to the variable x. Said another way, this statement binds the variable named x to the value 10. At this point the type of x is int because it is bound to an integer value.

The meaning of the assignment operator (=) is different from equality in mathematics. In mathematics, = asserts that the expression on its left is equal to the expression on its right. In Python, = makes the variable on its left take on the value of the expression on its right. It is best to read x = 5 as ?x is assigned the value 5,? or ?x gets the value 5.? This distinction is important since in mathematics equality is symmetric: if x = 5, we know 5 = x. In Python this symmetry does not exist; the statement illustrates the print function accepting two parameters. The ?rst parameter is the string 'x =', and the second parameter is the variable x bound to an integer value. The print function allows programmers to pass multiple expressions to print, each separated by commas. The elements within the parentheses of the print function comprise what is known as a comma-separated list. The print function prints each element in the comma-separated list of parameters. The print function automatically prints a space between each element in the list so the printed text elements do not run together.

a

2

x, y, z is one tuple, and 100, -45, 0 is another tuple. Tuple assignment works as follows: The ?rst variable in the tuple on left side of the assignment operator is assigned the value of the ?rst expression in the tuple on the left side (effectively x = 100). Similarly, the second variable in the tuple on left side of the assignment operator is assigned the value of the second expression in the tuple on the left side (in effect y = -45). z gets the value 0.

a
a = 2
2

a = 2
b = 5

a

2

b

5

a = 2
b = 5
a = 3
a = b
b = 7

a = 2
b = 5
a = 3

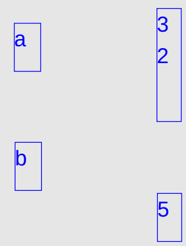
a

3
2

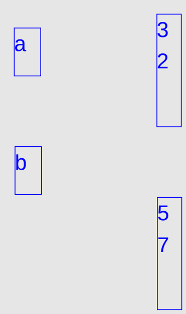
b

5

```
a = 2
b = 5
a = 3
a = b
```




```
a = 2
b = 5
a = 3
a = b
b = 7
```



```
x = 2
x
```

2

x

```
x = 2
del x
```

2

While mathematicians are content with giving their variables one-letter names like x, programmers should use longer, more descriptive variable names. Names such as sum, height, and sub_total are much better than the equally permissible s, h, and st. A variable?s name should be related to its purpose within the program. Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols more understandable.

Python is a case-sensitive language. This means that capitalization matters. if is a reserved word, but none of If, IF, or iF is a reserved word. Identi?ers also are case sensitive; the variable called Name is different from the variable called name. Note that three of the reserved words (False, None, and True) are capitalized.

The most important thing to remember about a variable?s name is that it should be well chosen. A variable?s name should re?ect the variable?s purpose within the program. For example, consider a program controlling a point-of-sale terminal (also known as an electronic cash register). The variable keeping track of the total cost of goods purchased might be named total or total_cost. Variable names such as a67_99 and fred would be poor choices for such an application.

Many computational tasks require numbers that have fractional parts. For example, to compute the area of a circle given the circle?s radius, we use the value ?, or approximately 3.14159. Python supports such non-integer numbers, and they are called ?oating-point numbers. The name implies that during mathematical calculations the decimal point can move or ??oat? to various positions within the number to maintain the proper number of signi?cant digits. The Python name for the ?oating-point type is float. Consider the following interactive session:

The ?rst line in Listing 2.7 (pi-print.py) assigns an approximation of ? to the variable named pi, and the second line prints its value. The last line prints some text along with a literal ?oating-point value. Any literal numeric value with a decimal point in a Python program automatically has the type float. This means the Python literal 2.0 is a float, not an int, even though mathematically we would classify it as an integer.

Floating-point numbers are an approximation of mathematical real numbers. The range of ?oating-point numbers is limited, since each value requires a ?xed amount of memory. Floating-point numbers differ from integers in another, very important way. An integer has an exact representation. This is not true necessarily for a ?oating-point number. Consider the real number ?. The mathematical constant ? is an irrational number which means it contains an in?nite number of digits with no pattern that repeats. Since ? contains an in?nite number of digits, a Python program can only approximate ??s value. Because of the limited number of digits available to ?oating-point numbers, Python cannot represent exactly even some numbers with a ?nite number of digits; for example, the number 23.3123400654033989 contains too many digits for the float type. As the following interaction sequence shows, Python stores 23.3123400654033989 as 23.312340065403397:

We also can use the round function to round a ?oating-point number to a speci?ed number of decimal places. The round function accepts an optional argument that produces a ?oating-point rounded to fewer decimal places. The additional argument must be an integer and speci?es the desired number of decimal places to round. In the shell we see

The expression round(n, r) rounds ?oating-point expression n to the 10?r decimal digit; for example, round(n, -2) rounds ?oating-point value n to the hundreds place (102). Similarly, round(n, 3) rounds ?oating-point value n to the thousandths place (10?3).

The characters that can appear within strings include letters of the alphabet (A-Z, a-z), digits (0-9), punctuation (., :, ,, etc.), and other printable symbols (#, &, %, etc.). In addition to these ?normal? characters, we may embed special characters known as control codes. Control codes control the way the console window or a printer renders text. The backslash symbol (\) signi?es that the character that follows it is a control code, not a literal character. The string '\n' thus contains a single control code. The backslash is known as the escape symbol, and in this case we say the n symbol is escaped. The \n control code represents the newline control code which moves the text cursor down to the next line in the console window. Other control codes include \t for tab, \f for a form feed (or page eject) on a printer, \b for backspace, and \a for alert (or bell). The \b and \a do not produce the desired results in the interactive shell, but they work properly in a command shell. Listing 2.9 (specialchars.py) prints some strings containing some of these control codes.

A string with a single quotation mark at the beginning must be terminated with a single quote; similarly, A string with a double quotation mark at the beginning must be terminated with a double quote. A single-quote string may have embedded double quotes, and a double-quote string may have embedded single quotes. If you wish to embed a single quote mark within a single-quote string, you can use the backslash to escape the single quote (\'). An unprotected single quote mark would terminate the string. Similarly, you may protect a double quote mark in a double-quote string with a backslash (\"). Listing 2.10 (escapequotes.py) shows the various ways in which quotation marks may be embedded within string literals.

This statement expects the user to enter an integer value. If the user types 3, for example, all is well. The variable num then will refer to the integer object 3. The int function can convert the string '3' to the integer value 3. The word number is ambiguous, however, so the user might attempt to enter 3.4. In this case the input statement would return the string '3.4'. The int function cannot convert the string '3.4' to an integer directly, even though it can convert the ?oating-point number 3.4 to the integer 3. The following interactive sequence demonstrates:

The assignment statement here uses the input function to obtain the number from the user as a string. It then uses the float function to convert the received string to a ?oating-point number. Finally it converts the ?oating-point number to an integer via the int function. What if you wish to round the user?s input value instead of truncating it? The following function composition would work in that case:

In Listing 2.13 (addintegers.py) we would prefer that the cursor remain at the end of the printed line so when the user types a value it appears on the same line as the message prompting for the values. When the user presses the enter key to complete the input, the cursor automatically will move down to the next line. The print function as we have seen so far always prints a line of text, and then the cursor moves down to the next line so any future printing appears on the next line. The print statement accepts an additional argument that allows the cursor to remain on the same line as the printed text:

The expression end='' is known as a keyword argument. The term keyword here means something different from the term keyword used to mean a reserved word. We defer a complete explanation of keyword arguments until we have explored more of the Python language. For now it is suf?cient to know that a print function call of this form will cause the cursor to remain on the same line as the printed text. Without this keyword argument, the cursor moves down to the next line after printing the text.

Another keyword argument allows us to control how the print function visually separates the arguments it displays. By default, the print function places a single space in between the items it prints. print uses a keyword argument named sep to specify the string to use insert between items. The name sep stands for separator. The default value of sep is the string ' ', a string containing a single space. Listing 2.17 (printsep.py) shows the sep keyword customizes print?s behavior.

? '{0} {1}': This is known as the formatting string. It is a Python string because it is a sequence of characters enclosed with quotes. Notice that the program at no time prints the literal string {0} {1}. This formatting string serves as a pattern that the second part of the expression will use. {0} and {1} are placeholders, known as positional parameters, to be replaced by other objects. This formatting string, therefore, represents two objects separated by a single space.

becomes ?7 10000000?, since 7 replaces {0} and 107 = 10000000 replaces {1}. Figure 2.8 shows how the arguments of format substitute for the positional parameters in the formatting string.

10000000

7

The positional parameter {0:>3} means ?right-justify the ?rst argument to format within a width of three characters.? Similarly, the {1:>16} positional parameter indicates that format?s second argument is to be right justi?ed within 16 places. This is exactly what we need to properly align the two columns of numbers.

11. How is the value 2.45×10?5 expressed as a Python literal?

Chapter 3

Expressions and Arithmetic

A literal value like 34 and a variable like x are examples of simple expressions. We can use operators to combine values and variables and form more complex expressions. In Section 2.1 we saw how we can use the + operator to add integers and concatenate strings. Listing 3.1 (adder.py) shows we can use the addition operator (+) to add two integers provided by the user.

This statement prompts the user to enter some information. After displaying the prompt string Please enter an integer value:, this statement causes the program?s execution to stop and wait for the user to type in some text and then press the enter key. The string produced by the input function is passed off to the int function which produces an integer value to assign to the variable value1. If the user types the sequence 431 and then presses the enter key, value1 is assigned the integer 431.

All expressions have a value. The process of determining the expression?s value is called evaluation. Evaluating simple expressions is easy. The literal value 54 evaluates to 54. The value of a variable named x is the value stored in the memory location bound to x. The value of a more complex expression is found by evaluating the smaller expressions that make it up and combining them with operators to form potentially new values.

Table 3.1 contains the most commonly used Python arithmetic operators. The common arithmetic operations, addition, subtraction, multiplication, division, and power behave in the expected way. The // and % operators are not common arithmetic operators in everyday practice, but they are very useful in programming. The // operator is called integer division, and the % operator is the modulus or remainder operator. 25/3 is 8.3333. Three does not divide into 25 evenly. In fact, three goes into 25 eight times with a remainder of one. Here, eight is the quotient, and one is the remainder. 25//3 is 8 (the quotient), and 25%3 is 1 (the remainder).

Two operators, + and -, can be used as unary operators. A unary operator has only one operand. The - unary operator expects a single numeric expression (literal number, variable, or more complicated numeric expression within parentheses) immediately to its right; it computes the additive inverse of its operand. If the operand is positive (greater than zero), the result is a negative value of the same magnitude; if the operand is negative (less than zero), the result is a positive value of the same magnitude. Zero is unaffected. For example, the following code sequence

The // operator produces an integer result when used with integers. In the ?rst case above 25 divided by 4 is 6 with a remainder of 1, and in the second case 4 divided by 25 is 0 with a remainder of 4. Since integers are whole numbers, the // operator discards any fractional part of the answer. The process of discarding the fractional part of a number leaving only the whole number part is called truncation. Truncation is not rounding; for example, 13 divided by 5 is 2.6, but 2.6 truncates to 2.

6

25//4

4)25

-24

1

25%4

$$1 - \frac{1}{3} - \frac{1}{3} - \frac{1}{3} = 0$$

Floating-point numbers are not real numbers, so the result of 1.0/3.0 cannot be represented exactly without infinite precision. In the decimal (base 10) number system, one-third is a repeating fraction, so it has an infinite number of digits. Even simple nonrepeating decimal numbers can be a problem. One-tenth (0.1) is obviously nonrepeating, so we can express it exactly with a finite number of digits. As it turns out, since numbers within computers are stored in binary (base 2) form, even one-tenth cannot be represented exactly with floating-point numbers, as Listing 3.3 (imprecise10.py) illustrates.

In Listing 3.3 (imprecise10.py) lines 3–6 make up a single Python statement. If that single statement that performs nine subtractions were written on one line, it would flow well off the page or off the editing window. Ordinarily a Python statement ends at the end of the source code line. A programmer may break up a very long line over two or more lines by using the backslash (\) symbol at the end of an incomplete line. When the interpreter is processing a line that ends with a \, it automatically joins the line that follows. The interpreter thus sees a very long but complete Python statement.

The last closing parenthesis on the second line matches the ?rst opening parenthesis on the ?rst line. No backslash symbol is required at the end of the ?rst line. The interpreter will begin scanning the ?rst line, matching closing parentheses with opening parentheses. When it gets to the end of the line and has not detected a closing parenthesis to match an earlier opening parenthesis, the interpreter assumes it must appear on a subsequent line, and so continues scanning until it completes the long statement. If the interpreter does not ?nd expected closing parenthesis in a program, it issues a error. In the Python interactive shell, the interpreter keeps waiting until the user complies or otherwise types something that causes an error:

tional power, the mathematics will work out precisely. Python can represent the fraction 1 4 = 0.25 = 2?2

When should you use integers and when should you use ?oating-point numbers? A good rule of thumb is this: use integers to count things and use ?oating-point numbers for quantities obtained from a measuring device. As examples, we can measure length with a ruler or a laser range ?nder; we can measure volume with a graduated cylinder or a ?ow meter; we can measure mass with a spring scale or triple-beam balance. In all of these cases, the accuracy of the measured quantity is limited by the accuracy of the measuring device and the competence of the person or system performing the measurement. Environmental factors such as temperature or air density can affect some measurements. In general, the degree of inexactness of such measured quantities is far greater than that of the ?oating-point values that represent them.

x is an integer and y is a ?oating-point number. What type is the expression x + y? Except in the case of the / operator, arithmetic expressions that involve only integers produce an integer result. All arithmetic operators applied to ?oating-point numbers produce a ?oating-point result. When an operator has mixed operands?one operand an integer and the other a ?oating-point number?the interpreter treats the integer operand as ?oating-point number and performs ?oating-point arithmetic. This means x + y is a ?oating-point expression, and the assignment will make the variable sum bind to a ?oating-point value.

The assignment operator is a different kind of operator from the arithmetic operators. Programmers use the assignment operator only to build assignment statements. Python does not allow the assignment operator to be part of a larger expression or part of another statement. As such, the notions of precedence and associativity do not apply in the context of the assignment operator. Python does, however, support a special kind of assignment statement called chained assignment. The code

Good programmers annotate their code by inserting remarks that explain the purpose of a section of code or why they chose to write a section of code the way they did. These notes are meant for human readers, not the interpreter. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. Well-chosen identi?ers (see Section 2.3) and comments can aid this assessment process. Also, in practice, teams of programmers develop software. A different programmer may be required to ?nish or ?x a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or un?nished code. While it may seem dif?cult to believe, even the same programmer working on her own code months later can have a dif?cult time remembering what various parts do. Comments can help greatly.

The interpreter is designed to execute all valid Python programs. The interpreter reads the Python source ?le and translates it into a executable form. This is the translation phase. If the interpreter detects an invalid program statement during the translation phase, it will terminate the program?s execution and report an error. Such errors result from the programmer?s misuse of the language. A syntax error is a common error that the interpreter can detect when attempting to translate a Python statement into machine language. For example, in English one can say

The interpreter detects syntax errors immediately. Syntax errors never make it out of the translation phase. Sometimes run-time exceptions do not reveal themselves immediately. The interpreter issues a run-time exception only when it attempts to execute the faulty statement. In Chapter 4 we will see how to write programs that optionally execute some statements only under certain conditions. If those conditions do not arise during testing, the faulty code does not get a chance to execute. This means the error may lie undetected until a user stumbles upon it after the software is deployed. Run-time exceptions, therefore, are more troublesome than syntax errors.

Undiscovered run-time errors and logic errors that lurk in software are commonly called bugs. The interpreter reports execution errors (exceptions) only when the conditions are right that reveal those errors. The interpreter is of no help at all with logic errors. Such bugs are the major source of frustration for developers. The frustration often arises because in complex programs the bugs sometimes reveal themselves only in certain situations that are dif?cult to reproduce exactly during testing. You will discover this frustration as your programs become more complicated. The good news is that programming experience and the disciplined application of good programming techniques can help reduce the number of logic errors. The bad news is that since software development in an inherently human intellectual pursuit, logic errors are inevitable. Accidentally introducing and later ?nding and eliminating logic errors is an integral part of the programming process.

?C = 5 9 ×(?F ?32)

The right side of the assignment operator (=) is ?rst evaluated. The statement assigns back to the seconds variable the remainder of seconds divided by 3,600. This statement can alter the value of seconds if the current value of seconds is greater than 3,600. A similar statement that occurs frequently in programs is one like

This statement increments the variable x to make it one bigger. A statement like this one provides further evidence that the Python assignment operator does not mean mathematical equality. The following statement from mathematics

x = x+1

A variation on Listing 3.9 (timeconv.py), Listing 3.10 (enhancedtimeconv.py) performs the same logic to compute the time components (hours, minutes, and seconds), but it uses simpler arithmetic to produce a slightly different output?instead of printing 11,045 seconds as 3 hr, 4 min, 5 sec, Listing 3.10 (enhancedtimeconv.py) displays it as 3:04:05. It is trivial to modify Listing 3.9 (timeconv.py) so that it would print 3:4:5, but Listing 3.10 (enhancedtimeconv.py) includes some extra arithmetic to put leading zeroes in front of single-digit values for minutes and seconds as is done on digital clock displays.

Have you ever tried to explain to someone how to perform a reasonably complex task? The task could involve how to make a loaf of bread from scratch, how to get to the zoo from city hall, or how to factor an algebraic expression. Were you able to explain all the steps perfectly without omitting any important details critical to the task?s solution? Were you frustrated because the person wanting to perform the task obviously was misunderstanding some of the steps in the process, and you believed you were making everything perfectly clear? Have you ever attempted to follow a recipe for your favorite dish only to discover that some of the instructions were unclear or ambiguous? Have you ever faithfully followed the travel directions provided by a friend and, in the end, found yourself nowhere near the intended destination?

Because many real-world tasks involve a number of factors, people sometimes get lucky and can complete a complex task given less-than-perfect instructions. A person often can use experience and common sense to handle ambiguous or incomplete instructions. If fact, humans are so good at dealing with ?fuzzy? knowledge that in most instances the effort to produce excruciatingly detailed instructions to complete a task is not worth the effort.

When a computer executes the instructions found in software, it has no cumulative experience and no common sense. It is a slave that dutifully executes the instructions it receives. While executing a program a computer cannot ?ll in the gaps in instructions that a human naturally might be able to do. Further, unlike with humans, executing the same program over and over does not improve the computer?s ability to perform the task. The computer has no understanding.

An algorithm is a ?nite sequence of steps, each step taking a ?nite length of time, that solves a problem or computes a result. A computer program is one example of an algorithm, as is a recipe to make lasagna. In both of these examples, the order of the steps matter. In the case of lasagna, the noodles must be cooked in boiling water before they are layered into the ?lling to be baked. It would be inappropriate to place the raw noodles into the pan with all the other ingredients, bake it, and then later remove the already baked noodles to cook them in boiling water separately. In the same way, the ordering of steps is very important in a computer program. While this point may be obvious, consider the following sound argument:

$$?C = 5\;9 \times (?F\ ?32)$$

The problem with this section of code is that after the ?rst statement is executed, x and y both have the same value (y?s original value). The second assignment is super?uous and does nothing to change the values of x or y. The solution requires a third variable to remember the original value of one the variables before it is reassigned. The correct code to swap the values is

The algorithms we have seen so far have been simple. Statement 1, followed by Statement 2, etc. until every statement in the program has been executed. Chapters 4 and 5 introduce some language constructs that permit optional and repetitive execution of some statements. These constructs allow us to build programs that do much more interesting things, but the algorithms that take advantage of them are more complex. We must not lose sight of the fact that a complicated algorithm that is 99% correct is not correct. An algorithm?s design and implementation can be derailed by inattention to the smallest of details.

i1 = 2
i2 = 5
i3 = -3
d1 = 2.0
d2 = 5.0
d3 = -0.5

i1 = 2
i2 = 5
i3 = -3
d1 = 2.0
d2 = 5.0
d3 = -0.5

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

Chapter 4

Conditional Execution

All the programs in the preceding chapters execute exactly the same statements regardless of the input, if any, provided to them. They follow a linear sequence: Statement 1, Statement 2, etc. until the last statement is executed and the program terminates. Linear programs like these are very limited in the problems they can solve. This chapter introduces constructs that allow program statements to be optionally executed, depending on the context of the program?s execution.

Arithmetic expressions evaluate to numeric values; a Boolean expression, sometimes called a predicate, may have only one of two possible values: false or true. The term Boolean comes from the name of the British mathematician George Boole. A branch of discrete mathematics called Boolean algebra is dedicated to the study of the properties and the manipulation of logical expressions. While on the surface Boolean expressions may appear very limited compared to numeric expressions, they are essential for building more interesting and useful programs.

We have seen that the simplest Boolean expressions are False and True, the Python Boolean literals. A Boolean variable is also a Boolean expression. An expression comparing numeric expressions for equality or inequality is also a Boolean expression. The simplest kinds of Boolean expressions use relational operators to compare two expressions. Table 4.1 lists the relational operators available in Python.

Table 4.2 shows some simple Boolean expressions with their associated values. An expression like 10 < 20 is legal but of little use, since 10 < 20 is always true; the expression True is equivalent, simpler, and less likely to confuse human readers. Since variables can change their values during a program?s execution, Boolean expressions are most useful when their truth values depend on the values of one or more variables.

if : condition

block

Why is indentation that mixes tabs and spaces a problem and thus forbidden in Python 3? Consider creating a Python source ?le in one editor and then viewing it in a different editor with tab stops set differently. Lines that appear perfectly indented in the original editor would be misaligned in the new editor. Instead, code indented with four spaces within one editor would appear exactly the same in any other editor.

The assignment statement and ?rst printing statement are both a part of the block of the if. Given the truth value of the Boolean expression divisor != 0 during a particular program run, either both statements will be executed or neither statement will be executed. The last statement is not indented, so it is not part of the if block. The program always prints Program ?nished, regardless of the user?s input.

never prints anything. Python considers the integer value zero to be false and treats every other integer value, positive and negative, to be true. Similarly, the ?oating-point value 0.0 is false, but any other ?oating-point value is true. The empty string ('' or "") is considered false, and any nonempty string is interpreted as true. Any Python expression can serve as the condition for an if statement. In later chapters we will explore additional kinds of expressions and see how they relate to Boolean conditions.

```
if          : condition
        if-block
else:
        else-block
```

To introduce compound Boolean expressions, consider a computer science degree that requires, among other computing courses, Operating Systems and Programming Languages. If we isolate those two courses, we can say a student must successfully complete both Operating Systems and Programming Languages to qualify for the degree. A student that passes Operating Systems but not Programming Languages will not have met the requirements. Similarly, Programming Languages without Operating Systems is insuf?cient, and a student completing neither Operating Systems nor Programming Languages surely does not qualify.

Related to the logical and operator is the logical or operator. To illustrate the logical or operator, consider two mathematics courses, Differential Equations and Linear Algebra. A computer science degree requires at least one of those two courses. A student who successfully completes Differential Equations but does not take Linear Algebra meets the requirement. Similarly, a student may take Linear Algebra but not Differential Equations. A student that takes neither Differential Equations nor Linear Algebra certainly has not met the requirement. It is important to note the a student may elect to take both Differential Equations and Linear Algebra (perhaps on the way to a mathematics minor), but the requirement is no less ful?lled.

Logical or works in a similar fashion. Given our Boolean expressions e1 and e2, the compound expression e1 or e2 is false only if e1 and e2 are both false; if either one is true or both are true, the compound expression is true. Note that the or operator is an inclusive or, not an exclusive or. In informal conversion we often imply exclusive or in a statement like ?Would you like cake or ice cream for dessert?? The implication is one or the other, not both. In computer programming the or is inclusive; if both subexpressions in an or expression are true, the or expression is true.

Logical logical not operator reverses the truth value of the expression to which it is applied. If e is a true Boolean expression, not e is false; if e is false, not e is true. In mathematics, if the expression x = y is false, it must be true that x ?= y. In Python, the expression not (x == y) is equivalent to the expression x != y. If also is the case that the Python expresion not (x != y) is just a more complicated way of expressing x == y. In mathematics, if the expression x < y is false, it must be the case that x ?y. In Python, not (x < y) has the same truth value as x >= y. The expression not (x >= y) is equivalent to x < y. You may be able to see from these examples that if e is a Boolean expression, it always is true that not not e is equivalent to e (this is known as the double negative property of mathematical logic).

Table 4.4 lists the Python operators we have seen so far. Table 4.4 shows that operator not has higher precedence than both and and or. The and operator has higher precedence than or. Both the and and or operators are left associative; not is right associative. The and and or operators have lower precedence than any other binary operator except assignment. This means the expression

In the expression e1 and e2 both subexpressions e1 and e2 must be true for the overall expression to be true. Since the and operator evaluates left to right, this means that if e1 is false, there is no need to evaluate e2. If e1 is false, no value of e2 can make the expression e1 and e2 true. The and operator ?rst tests the expression to its left. If it ?nds the expression to be false, it does not bother to check the right expression. This approach is called short-circuit evaluation. In a similar fashion, in the expression e1 or e2, if e1 is true, then e2?s value is irrelevant?an or expression is true unless both subexpressions are false. The or operator uses short-circuit evaluation also.

? The order of the subexpressions can affect performance. When a program is running, complex expressions require more time for the computer to evaluate than simpler expressions. We classify an expression that takes a relatively long time to evaluate as an expensive expression. If a compound Boolean expression is made up of an expensive Boolean subexpression and an less expensive Boolean subexpression, and the order of evaluation of the two expressions does not effect the behavior of the program, then place the more expensive Boolean expression second. In the context of the and operator, if its left operand is False, the more more expensive right operand need not be evaluated. In the context of the or operator, if the left operand is True, the more expensive right operand may be ignored.

If x is a numeric value less than 10, this statement will query the user to print or not print the value of x. If x ?10, the program need not stop and wait for the user?s input. If x ?10, the user?s input is super?uous anyway. Now consider the statement with the Boolean expressions ordered the other way:

In this case as well, both subconditions must be true to print the value of x. The difference here is that the program always pauses its execution to accept the user?s input regardless of x?s value. This statement bothers the user for input even when the second subcondition ensures the user?s answer will make no difference.

If the value of x is less than zero, this section of code should print nothing. Unfortunately, the code fragment above is not legal Python. The if/else statement contains an else block, but it does not contain an if block. The comment does not count as a Python statement. Both if and if/else statements require an if block that contains at least one statement. Additionally, an if/else statement requires an else block that contains at least one statement.

? If the executing program ?nds the value variable to be greater than or equal to zero, it executes the statement within the if-block. This statement is itself an if statement. The program thus checks the second (inner) condition. If the second condition is satis?ed, the program displays the In range message; otherwise, it does not. Regardless, the program eventually prints the Done message.

We say that the second if (with the comment Second check) is nested within the ?rst if (First check). We call the ?rst if the outer if and the second if the inner if. Notice the entire inner if statement is indented one level relative to the outer if statement. This means the inner if?s block, the print("In range") statement, is indented two levels deeper than the outer if statement. Remember that if you use four spaces as the distance for a indentation level, you must consistently use this four space distance for each indentation level throughout the program.

Computers store all data internally in binary form. The binary (base 2) number system is much simpler than the familiar decimal (base 10) number system because it uses only two digits: 0 and 1. The decimal system uses 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Despite the lack of digits, every decimal integer has an equivalent binary representation. Binary numbers use a place value system not unlike the decimal system. Figure 4.3 shows how the familiar base 10 place value system works.

With only two digits to work with, the binary number system distinguishes place values by powers of two. Since both binary and decimal numbers share the digits 0 and 1, we will use the subscript 2 to indicate a binary number; therefore, 100 represents the decimal value one hundred, while 1002 is the binary number four. Sometimes to be very clear we will attach a subscript of 10 to a decimal number, as in 10010.

Listing 4.12 (binaryconversion.py) uses an if statement containing a series of nested if statements to print a 10-bit binary string representing the binary equivalent of a decimal integer supplied by the user. We use if/else statements to print the individual digits left to right, essentially assembling the sequence of bits that represents the binary number.

? prints appends the digit (actually character) 1 to the binary string under construction, and
? removes via the remainder operator that power of two?s contribution to the value.

The sole if statement in Listing 4.13 (simplerbinaryconversion.py) ensures that the user provides an integer in the proper range. The other if statements that originally appeared in Listing 4.12 (binaryconversion.py) are gone. A clever sequence of integer arithmetic operations replace the original conditional logic. The two programs?binaryconversion.py and simplerbinaryconversion.py?behave identically but simplerbinaryconversion.py?s logic is simpler.

This very simple troubleshooting program attempts to diagnose why a computer does not work. The potential for enhancement is unlimited, but this version deals only with power issues that have simple ?xes. Notice that if the computer has power (fan or disk drive makes sounds or lights are visible), the program indicates that help should be sought elsewhere! The decision tree capturing the basic logic of the program is shown in Figure 4.6. The steps performed are:

In Listing 4.16 (timeconvcond2.py) each code segment responsible for printing a time value and its English word unit is protected by an if statement that only allows the code to execute if the time value is greater than zero. The exception is in the processing of seconds: if all time values are zero, the program should print 0 seconds. Note that each of the if/else statements responsible for determining the singular or plural form is nested within the if statement that determines whether or not the value will be printed at all.

A simple if/else statement can select from between two execution paths. Listing 4.11 (enhancedcheckrange.py) showed how to select from among three options. What if exactly one of many actions should be taken? Nested if/else statements are required, and the form of these nested if/else statements is shown in Listing 4.17 (digittoword.py).

? Notice that each if block contains a single printing statement and each else block, except the last one, contains an if statement. The control logic forces the program execution to check each condition in turn. The ?rst condition that matches wins, and its corresponding if body will be executed. If none of the conditions are true, the program prints the last else?s Too large message.

As a consequence of the required formatting of Listing 4.17 (digittoword.py), the mass of text drifts to the right as more conditions are checked. Python provides a multi-way conditional construct called if/elif/else that permits a more manageable textual structure for programs that must check many conditions. Listing 4.18 (restyleddigittoword.py) uses the if/elif/else statement to avoid the rightward code drift.

```
if condition-1:
    block-1
elif condition-2:
    block-2
elif condition-3:
    block-3
elif condition-4:
    block-4
    ...
else:
    default-block
```

Other checks skipped

Some argue that the conditional expression is not as readable as a normal if/else statement. Regardless, many Python programmers use it sparingly because of its very speci?c nature. Standard if/else blocks can contain multiple statements, but contents in the conditional expression are limited to single, simple expressions.

What values of x make the expression true, and what values of x make the expression false? This expression is always true, no matter what value is assigned to the variable x. A Boolean expression that is always true is known as a tautology. Think about it. If x is a number, what value could the variable x assume that would make this Boolean expression false? Regardless of its value, one or both of the subexpressions will be true, so the compound or expression is always true. This particular or expression is just a complicated way of expressing the value True.

In the end the meaning of the max variable remains the same??maximum I have determined so far,? but, after comparing max to all the input variables, we now know it is the maximum value of all four input numbers. The extra variable max is not strictly necessary, but it makes thinking about the problem and its solution easier.

What changes would we need to make to both Listing 4.28 (max4a.py) and Listing 4.29 (max4b.py) if we must extend them to handle ?ve input values instead of four? Adding this capability to Listing 4.28 (max4a.py) forces us to modify every condition in the program, adding a check against a new num5 variable. We also must provide an additional elif check since we will need to select from among ?ve possible assignments to the max variable. In Listing 4.29 (max4b.py), however, we need only add an extra sequential if statement with a new simple condition to check.

Chapter 5 introduces loops, the ability to execute statements repeatedly. You easily can adapt the sequential if approach to allow users to type in as many numbers as they like and then have the program report the maximum number the user entered. The multiway if/elif/else approach with the more complex logic cannot be adapted in this manner. Not only is our sequential if version cleaner and simpler; we more easily can extend its capabilities.

```
if j > k:
j = i
else:
i = k
print("i =", i, " j =", j, " k =", k)
```

```
if val != 5:
print("wow ", end='')
else:
val += 1
else:
```

```
if val == 17:
val += 10
else:
print("whoa ", end='')
print(val)
```

```
n = int(input())
if n < 1000:
print('*', end='')
if n < 100:
print('*', end='')
if n < 10:
print('*', end='')
if n < 1:
print('*', end='')
print()
```

```
n = int(input())
if n < 1000:
print('*', end='')
elif n < 100:
print('*', end='')
elif n < 10:
print('*', end='')
elif n < 1:
print('*', end='')
print()
```

18. Write a Python program that requests ?ve integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4 2, 3 and 3, the program should report 2 as the minimum and 4 as maximum.

Chapter 5

Iteration

How would you write the code to count to 10,000? Would you copy, paste, and modify 10,000 printing statements? You could, but that would be impractical! Counting is such a common activity, and computers routinely count up to very large values, so there must be a better way. What we really would like to do is print the value of a variable (call it count), then increment the variable (count += 1), and repeat this process until the variable is large enough (count == 5 or maybe count == 10000). This process of executing the same section of code over and over is known as iteration, or looping. Python has two different statements, while and for, that enable iteration.

begins the while statement. The expression following the while keyword is the condition that determines if the statement block is executed or continues to execute. As long as the condition is true, the program executes the code block over and over again. When the condition becomes false, the loop is ?nished. If the condition is false initially, the program will not execute the code block within the body of the loop at all.

while : condition

block

Except for the reserved word while instead of if, while statements look identical to if statements. Sometimes beginning programmers confuse the two or accidentally type if when they mean while or vice-versa. Usually the very different behavior of the two statements reveals the problem immediately; however, sometimes, especially in nested, complex logic, this mistake can be hard to detect.

The executing program checks the condition before executing the while block and then checks the condition again after executing the while block. As long as the condition remains truth, the program repeatedly executes the code in the while block. If the condition initially is false, the program will not execute the code within the while block. If the condition initially is true, the program executes the block repeatedly until the condition becomes false, at which point the loop terminates.

In the beginning we initialize entry to zero for the sole reason that we want the condition entry >= 0 of the while statement to be true initially. If we fail to initialize entry, the program will produce a run-time error when it attempts to compare entry to zero in the while condition. The entry variable holds the number entered by the user. Its value can change each time through the loop.

The variable sum is known as an accumulator because it accumulates each value the user enters. We initialize sum to zero in the beginning because a value of zero indicates that it has not accumulated anything. If we fail to initialize sum, the program will generate a run-time error when it attempts to use the += operator to modify the (non-existent) variable. Within the loop we repeatedly add the user?s input values to sum. When the loop ?nishes (because the user entered a negative number), sum holds the sum of all the nonnegative values entered by the user.

The initialization of entry to zero coupled with the condition entry >= 0 of the while guarantees that the program will execute the body of the while loop at least once. The if statement ensures that the program will not add a negative entry to sum. (Could the if condition have used > instead of >= and achieved the same results?) When the user enters a negative value, the executing program will not update the sum variable, and the condition of the while will no longer be true. The loop then terminates and the program executes the print statement.

We can use a while statement to make Listing 4.14 (troubleshoot.py) more convenient for the user. Recall that the computer troubleshooting program forces the user to rerun the program once a potential program has been detected (for example, turn on the power switch, then run the program again to see what else might be wrong). A more desirable decision logic is shown in Figure 5.2.

A while block makes up the bulk of Listing 5.5 (troubleshootloop.py). The Boolean variable done controls the loop; as long as done is false, the loop continues. A Boolean variable like done used in this fashion is often called a ?ag. You can think of the ?ag being down when the value is false and raised when it is true. In this case, when the ?ag is raised, it is a signal that the loop should terminate.

real numbers, if we add $\frac{1}{10}$ to 0 ten times, the result equals 1. Unfortunately, Listing 5.6 (stopatone.py) bypasses 1 and keeps going! The first few lines of its execution are

The program never stops printing numbers. Note that $\frac{1}{10}$ does not have an exact internal binary floating-point representation. If you change the statement

the program stops as expected. This is because $\frac{1}{8}$ happens to have an exact internal binary floating-point representation. Since such exact representations of floating-point values are rare, you should avoid using the == and != operators with floating-point numbers to control the number of loop iterations. Listing 5.7 (stopatonefixed.py) uses <= to control the loop instead of !=, and it behaves as expected.

Looking at the source code of Listing 5.9 (de?nite2.py), we cannot predict how many times the loop will repeat. The number of iterations depends on the input provided by the user. However, at the program?s point of execution after obtaining the user?s input and before the start of the execution of the loop, we would be able to determine the number of iterations the while loop would perform. Because of this, the loop in Listing 5.9 (de?nite2.py) is considered to be a de?nite loop as well.

range(begin,end,step)

The ?rst number is i?s value at the beginning of the block, and the parenthesized number is i?s value at the end of the block before the next iteration. The code within the block can reassign i, but this binds i to a different integer object (20). The next time through the loop the for statement obtains the next integer served by the range object and binds i to this new integer.

If you look in older Python books or at online examples of Python code, you probably will encounter the xrange expression. Python 2 has both range and xrange, but Python 3 (the version we use in this text) does not have the xrange expresson. The range expression in Python 3 is equivalent to the xrange expression in Python 2. The range expression in Python 2 creates a data structure called a list, and this process can involve considerable overhead for an executing program. The xrange expression in Python 2 avoids this overhead, making it more ef?cient than range, especially for a large sequence. When building loops with the for statement, Python 2 programmers usually use xrange rather than range to improve their code?s ef?ciency. In Python 3, we can use range without compromising run-time performance. In Chapter 10 we will see it is easy to make a list out of a Python 3 range expression, so Python 3 does not need two different range expressions that do almost exactly the same thing.

We initially emphasize the for loop?s ability to iterate over integer sequences because this is a useful and common task in software construction. The for loop, however, can iterate over any iterable object. As we have seen, a tuple is an iterable object, and a range object is an iterable object. A string also is an iterable object. We can use a for loop to iterate over the characters that comprise a string. Listing 5.13 (stringletters.py) uses a for loop to print the individual characters of a string.

Listing 5.16 (timestable1.py) does indeed print each row in its proper place?it just does not supply the needed detail for each row. Our next step is to re?ne the way the program prints each row. Each row should contain size numbers. Each number within each row represents the product of the current row and current column; for example, the number in row 2, column 5 should be 2 × 5 = 10. In each row, therefore, we must vary the column number from from 1 to size. Listing 5.17 (timestable2.py) contains the needed re?nement.

Normally, a while statement executes until its condition becomes false. A running program checks this condition ?rst to determine if it should execute the statements in the loop?s body. It then re-checks this condition only after executing all the statements in the loop?s body. Ordinarily a while loop will not immediately exit its body if its condition becomes false before completing all the statements in its body. The while statement is designed this way because usually the programmer intends to execute all the statements within the body as an indivisible unit. Sometimes, however, it is desirable to immediately exit the body or recheck the condition from the middle of the loop instead. Said another way, a while statement checks its condition only at the ?top? of the loop. It is not the case that a while loop ?nishes immediately whenever its condition becomes true. Listing 5.22 (whileexitattop.py) demonstrates this top-exit behavior.

of the while can never be false, the break statement is the only way to get out of the loop. Here, the break statement executes only when it determines that the number the user entered is negative. When the program encounters the break statement during its execution, it skips any statements that follow in the loop?s body and exits the loop immediately. The keyword break means ?break out of the loop.? The placement of the break statement in Listing 5.23 (addmiddleexit.py) makes it impossible to add a negative number to the sum variable.

Some software designers believe that programmers should use the break statement sparingly because it deviates from the normal loop control logic. Ideally, every loop should have a single entry point and single exit point. While Listing 5.23 (addmiddleexit.py) has a single exit point (the break statement), some programmers commonly use break statements within while statements in the which the condition for the while is not a tautology. Adding a break statement to such a loop adds an extra exit point (the top of the loop where the condition is checked is one point, and the break statement is another). Most programmers ?nd two exits point perfectly acceptable, but much above two break points within a single loop is particularly dubious and you should avoid that practice.

the
break
statement

The break statement is handy when a situation arises that requires immediate exit from a loop. The for loop in Python behaves differently from the while loop, in that it has no explicit condition that it checks to continue its iteration. We must use a break statement if we wish to prematurely exit a for loop before it has completed its speci?ed iterations. The for loop is a de?nite loop, which means programmers can determine up front the number of iterations the loop will perform. The break statement has the potential to disrupt this predictability. For this reason, programmers use break statements in for loops less frequently, and they often serve as an escape from a bad situation that continued iteration might make worse.

When a program?s execution encounters a break statement inside a loop, it skips the rest of the body of the loop and exits the loop. The continue statement is similar to the break statement, except the continue statement does not necessarily exit the loop. The continue statement skips the rest of the body of the loop and immediately checks the loop?s condition. If the loop?s condition remains true, the loop?s execution resumes at the top of the loop. Listing 5.26 (continueexample.py) shows the continue statement in action.

Programmers do not use the continue statement as frequently as the break statement since it is very easy to transform the code that uses continue into an equivalent form that does not. Listing 5.27 (nocontinueexample.py) works exactly like Listing 5.26 (continueexample.py), but it avoids the continue statement.

the
continue

Listing 5.29 (whilenoelse.py) uses two distinct Python constructs, the while statement followed by an if/else statement, whereas Listing 5.28 (whileelse.py) uses only one, a while/else statement. Listing 5.29 (whilenoelse.py) also must check the count < 5 condition twice, once in the while statement and again in the if/else statement.

A for statement with an else block works similarly to the while/else statement. When a for/else loop exits because it has considered all the values in its range or all the characters in its string, it executes the code in its associated else block. If a for/else statement exits prematurely due to a break statement, it does not execute the code in its else block. Listing 5.30 (countvowelselse.py) shows how the else block works with a for statement.

An in?nite loop is a loop that executes its block of statements repeatedly until the user forces the program to quit. Once the program ?ow enters the loop?s body it cannot escape. In?nite loops sometimes are by design. For example, a long-running server application like a Web server may need to continuously check for incoming connections. The Web server can perform this checking within a loop that runs inde?nitely. Beginning programmers, unfortunately, all too often create in?nite loops by accident, and these in?nite loops represent logic errors in their programs.

? The condition of a while must be true initially to gain access to its body. The code within the body must modify the state of the program in some way so as to in?uence the outcome of the condition that is checked at each iteration. This usually means the body must be able to modify one of the variables used in the condition. Eventually the variable assumes a value that makes the condition false, and the loop terminates.

In Listing 5.31 (?ndfactors.py) the outer loop?s condition involves the variables n and MAX. We observe that we assign 20 to MAX before the loop and never change it afterward, so to avoid an in?nite loop it is essential that n be modi?ed within the loop. Fortunately, the last statement in the body of the outer loop increments n. n is initially 1 and MAX is 20, so unless the circumstances arise to make the inner loop in?nite, the outer loop eventually should terminate.

The inner loop?s condition involves the variables n and factor. No statement in the inner loop modi?es n, so it is imperative that factor be modi?ed in the loop. The good news is factor is incremented in the body of the inner loop, but the bad news is the increment operation is protected within the body of the if statement. The inner loop contains one statement, the if statement. That if statement in turn has two statements in its body:

Listing 5.32 (?ndfactorsfor.py) is a different version of our factor ?nder program that uses nested for loops instead of nested while loops. Not only is it slightly shorter, but it avoids the potential for the misplaced increment of the factor variable. This is because the for statement automatically handles the loop variable update.

As a final note on infinite loops, Section 1.4 mentioned the preference for using the Debug option under the WingIDE-101 integrated development environment when running our programs. When executing the program under the Run option, the IDE can become unresponsive if the program encounters an infinite loop. At that point, terminating the IDE is the only solution. Under the debugger, we very easily can interrupt a wayward program's execution via WingIDE-101's Stop action.

? The second inner loop prints the row of asterisks that make up the tree. The ?rst time through the outer loop, row is zero, so it prints no left side asterisks, one central asterisk, and no right side asterisks. Each time through the loop the number of left-hand and right-hand stars to print both increase by one, but there remains just one central asterisk to print. This means the tree grows one wider on each side for each line moving down. Observe how the 2*row + 1 value expresses the needed number of asterisks perfectly.

For comparison, Listing 5.35 (startreefor.py) uses for loops instead of while loops to draw our star trees. The for loop is a better choice for this program since once the user provides the height, the program can calculate exactly the number of iterations required for each loop. This number will not change during the rest of the program?s execution, so the de?nite loop (for) is better a better choice than the inde?nite loop (while).

The expression value % trial_factor is zero when trial_factor divides into value with no remainder?exactly when trial_factor is a factor of value. If the program discovers a value of trial_factor that actually is a factor of value, then it sets is_prime false and exits the loop via the break statement. If the loop continues to completion, the program will not set is_prime to false, which means it found no factors, and, so, value is indeed prime.

In order to enter the body of the inner loop, trial_factor must be less than value. value does not change anywhere in the loop. trial_factor is not modi?ed anywhere in the if statement within the loop, and it is incremented within the loop immediately after the if statement. trial_factor is, therefore, incremented during each iteration of the loop. Eventually, trial_factor will equal value, and the loop will terminate.

In order to enter the body of the outer loop, value must be less than or equal to max_value. max_value does not change anywhere in the loop. The last statement within the body of the outer loop increases value, and no where else does the program modify value. Since the inner loop is guaranteed to terminate as shown in the previous answer, eventually value will exceed max_value and the loop will end.

This version without the break introduces a slightly more complicated condition for the while but removes the if statement within its body. is_prime is initialized to true before the loop. Each time through the loop it is reassigned. trial_factor will become false if at any time value % trial_factor is zero. This is exactly when trial_factor is a factor of value. If is_prime becomes false, the loop cannot continue, and

If the inner for loop completes its iteration over all the values in its range, it will execute the print statement in its else block. The only way the inner for loop can be interrupted is if it discovers a factor of value. If it does ?nd a factor, the premature exit of the inner for loop prevents the execution of its else block. This logic enables it to print only prime numbers?exactly the behavior we want.

We initialize the variable in_value at the top of the program only to make sure the loop?s body executes at least one time. A de?nite loop (for) is inappropriate for a program like Listing 5.39 (betterinputonly.py) because the program cannot determine ahead of time how many attempts the user will make before providing a value in range.

```
a = 0
while a < 100:
print('*', end='')
a += 1
print()
```

```
a = 0
while a > 100:
print('*', end='')
a += 1
print()
```

```
a = 0
while a < 100:
b = 0
while b < 55:
print('*', end='')
b += 1
print()
a += 1
```

```
if (a + b) % 2 == 0:
print('*', end='')
b += 1
print()
a += 1
```

```
a = 0
while a < 100:
b = 0
while b < 100:
c = 0
while c < 100:
print('*', end='')
c += 1
b += 1
a += 1
print()
```

```
a = 0
while a < 100:
print(a)
a += 1
print()
```

```
a = 0
while a > 100:
print(a)
a += 1
print()
```

```
done = False
n, m = 0, 100
while not done and n != m:
n = int(input())
if n < 0:
done = True
print("n =", n)
```

```
a = 0
while a < 100:
print(a, end=' ')
a += 1
print()
```

Chapter 6

Using Functions

While this code may be acceptable for many applications, better algorithms exist that work faster and produce more precise answers. Another problem with the code is this: What if you are working on a signi?cant scienti?c or engineering application and must use different formulas in various parts of the source code, and each of these formulas involve square roots in some way? In mathematics, for example, we use square root to compute the distance between two geometric points (x1,y1) and (x2,y2) as

(x2 ?x1)2 +(y2 ?y1)2

and, using the quadratic formula, the solution to the equation ax2 +bx+c = 0 is

b2 ?4ac
2a

Suppose we are writing one big program that, among many other things, needs compute distances and solve quadratic equations. Must we copy and paste the relevant portions of our square root code found in Listing 5.33 (computesquareroot.py) to each location in our source code that requires a square root computation? Also, what if we develop another program that requires computing a root mean square? Will we need to copy the code from Listing 5.33 (computesquareroot.py) into every program that needs to compute square roots, or is there a better way to package the square root code and reuse it?

One way to make code more reusable is by packaging it in functions. A function is a unit of reusable code. In Chapter 7 we will see how to write our own reusable functions, but in this chapter we examine some of the functions available in the Python standard library. Python provides a collection of standard functions stored in libraries called modules. Programmers can use the functions from these libraries within their own code to build sophisticated programs.

16 = 4, so when presented with 16.0, sqrt responds with 4.0. Figure 6.1 illustrates the conceptual view of the sqrt function. The square root function is like a black box to the code that uses it. Callers do not need to know the details of the code inside the function in order to use it. Programmers are concerned more about what the function does, not how it does it.

Unlike the other functions we have used earlier, the interpreter is not automatically aware of the sqrt function. The sqrt function is not part of the small collection of functions (like type, int, and str) always available to Python programs. The sqrt function is part of separate module within the standard library. A module is a collection of Python code that can used in other programs. The import keyword makes a module available to the interpreter. The ?rst statement in Listing 6.1 (standardsquareroot.py) shows one way to use the import keyword:

num is the information the function needs to do its work. We say num is the argument, or parameter, passed to the function. We also can say ?we are passing num to the sqrt function.? The function uses the variable num?s value to perform the computation. Parameters enable callers to communicate information to a function during the function?s execution.

As noted in Figure 6.1, the square root function is a black box to the caller. The caller is concerned strictly about what the function does, not how the function accomplishes its task. We safely can treat all functions like black boxes. We can use the service that a function provides without being concerned about its internal details. Ordinarily we can in?uence the function?s behavior only via the parameters that we pass, and that nothing else we do can affect what the function does or how it does it. Furthermore, for the types of objects we have considered so far (integers, ?oating-point numbers, and strings), when a caller passes data to a function, the function cannot affect the caller?s copy of that data. The caller is, however, free to

? Parameters. A function must be called with a certain number of parameters, and each parameter must be the correct type. Some functions like print permit callers to pass a variable number of arguments, but most functions, like sqrt, specify an exact number. If a caller attempts to call a function with too many or too few parameters, the interpreter will issue an error message and refuse to run the program. Consider the following misuse of sqrt in the interactive shell:

Like mathematical functions that must produce a result, a Python function always produces a value to return to the caller. Some functions are not designed to produce any useful results. Clients call such a function for the effects provided by the executing code within a function, not for any value that the function computes. The print function is one such example. The print function displays text in the console window; it does not compute and return a value to the caller. Since Python requires that all functions return a value, print must return something. Functions that are not meant to return anything return the special object None. We can show this in the Python shell:

A Python module is simply a ?le that contains Python code. The name of the ?le dictates the name of the module; for example, a ?le named math.py contains the functions available from the standard math module. The modules of immediate interest to us are the standard modules that contain functions that our programs can use. The Python standard library contains thousands of functions distributed throughout more than 230 modules. These modules cover a wide range of application domains. One of the modules, known as the built-ins module (actual name __builtins__), contains all the functions we have been using in earlier chapters: print, input, etc. These built-in functions make up only a very small fraction of all the functions the standard library provides. Programmers must use one or more import statements within a program or within the interactive interpreter to gain access to the remaining standard functions.

and function name. Many programmers prefer this approach because the complete name unambiguously identi?es the function with its module. A large, complex program could import the math module and a different, third-party module called extramath. Suppose the extramath module provided its own sqrt function. There can be no mistaking the fact that the sqrt being called in the expression math.sqrt(16) is the one provided by the math module. It is impossible for a program to import the sqrt functions separately from both modules and use their simple names simultaneously within a program. Does

As programs become larger and more complex, the import entire module approach becomes more compelling. The quali?ed function names improve the code?s readability and avoids name clashes between modules that provide functions with identical names. Soon we will be writing our own, custom functions. Quali?ed names ensure that names we create ourselves will not clash with any names that modules may provide.

Section 6.1 observed that we have been using functions in Python since the ?rst chapter. These functions include print, input, int, float, str, and type. These functions and many others reside in a module named __builtins__. The __builtins__ module is special because its components are automatically available to any Python program with?no import statement is required. The full name of the print function is __builtins__.print, although chances are you will never see its full name written in a Python program. We can verify its fully quali?ed name in the interpreter:

This interactive sequence veri?es that the names print and __builtins__.print refer to the same function object. The id function is another __builtins__ function. The expression id(x) evaluates to the address in memory of object named x. Since id(print) and id(__builtins__.print) evaluate to the same value, we know both names correspond to the same function object.

The __builtins__ module provides a common core of general functions useful to any Python program regardless of its application area. The other standard modules that Python provides are aimed at speci?c application domains, such as mathematics, text processing, ?le processing, system administration, graphics, and Internet protocols, and multimedia. Programs that require more domain-speci?c functionality must

math Module
sqrt
Computes the square root of a number: sqrt(x) = ?x
exp

The parameter passed by the caller is known as the actual parameter. The parameter speci?ed by the function is called the formal parameter. During a function call the ?rst actual parameter is assigned to the ?rst formal parameter, the second actual parameter is assigned to the second formal parameter, etc. Callers must be careful to put the arguments they pass in the proper order when calling a function; for example, the call math.pow(10,2) computes 102 = 100, but the call math.pow(2,10) computes 210 = 1,024.

(x2,y2)

(px,py)

d2

d1

?

(x1,y1)

(0,0)

The functions in the math module are ideal for solving problems like the one shown in Figure 6.4. Suppose a spacecraft is at a ?xed location in space some distance from a planet. A satellite is orbiting the planet in a circular orbit. We wish to compute how much farther away the satellite will be from the spacecraft when it has progressed ? degrees along its orbital path.

We will let the origin of our coordinate system (0,0) be located at the center of the planet. This location corresponds also to the center of the satellite?s circular orbital path. The satellite is located as some point, (x,y) and the spacecraft is stationary at point (px, py). The spacecraft is located in the same plane as the satellite?s orbit. We wish to compute the distances between the moving point (satellite) and the ?xed point (spacecraft) as the satellite orbits the planet.

(x?px)2 +(y?py)2

This is because the value of x used in the second assignment statement is the new value of x computed by the ?rst assignment statement. The tuple assignment version uses the original x value in both computations. If we really wanted to use two assignment statements rather than a single tuple assignment, we would need to introduce an extra variable so we do not lose x?s original value:

and Mac OS X), time.clock returns the numbers of seconds elapsed since the program began executing. Under Microsoft Windows, time.clock returns the number of seconds since the ?rst call to time.clock. In either case, with two calls to the time.clock function we can measure elapsed time. Listing 6.5 (timeit.py) measures how long it takes a user to enter a character from the keyboard.

Some applications require behavior that appears random. Random numbers are particularly useful in games and simulations. For example, many board games use a die (one of a pair of dice?see Figure 6.5) to determine how many places a player is to advance. A die or pair of dice are used in other games of chance. A die is a cube containing spots on each of its six faces. The number of spots range from one to six. A player rolls a die or sometimes a pair of dice, and the side(s) that face up have meaning in the game being played. The value of a face after a roll is determined at random by the complex tumbling of the die. A software adaptation of a game that involves dice would need a way to simulate the random roll of a die.

All algorithmic random number generators actually produce pseudorandom numbers, not true random numbers. A pseudorandom number generator has a particular period, based on the nature of the algorithm used. If the generator is used long enough, the pattern of numbers produced repeats itself exactly. A sequence of true random numbers would not contain such a repeating subsequence. All practical algorithmic pseudorandom number generators have periods that are large enough for most applications.

randomfunctions Module
random

The random.seed function establishes the initial value from which the sequence of pseudorandom numbers is generated. Each call to random.random or random.randrange returns the next value in the sequence of pseudorandom values. Listing 6.10 (simplerandom.py) prints 100 pseudorandom integers in the range 1...100.

The numbers Listing 6.10 (simplerandom.py) prints appear to be random. The program begins its pseudorandom number generation with a seed value, 23. The seed value determines the exact sequence of numbers the program generates; identical seed values generate identical sequences. If you run the program again, it displays the same sequence. In order for the program to display different sequences, the seed value must be different for each run.

If we omit the call to the random.seed function, the program derives its initial value in the sequence from the time kept by the operating system. This usually is adequate for simple pseudorandom number sequences. Being able to specify a seed value is useful during development and testing when we want program executions to exhibit reproducible results.

Notice that when the user enters the text consisting of a single digit 4, the eval function interprets it as integer 4 and assigns an integer to the variable x1. When the user enters the text 4.0, the assigned variable is a ?oating-point variable. For x3, the user supplies the string "x3" (note the quotes), and the variable?s type is string. The more interesting situation is x4. The user enters x1 (no quotes). The eval function evaluates the unquoted text as a reference to the name x1 established by the ?rst assignment statement. The program bound the name x1 to the integer value 4 when executing the ?rst line of the program. This statement thus binds x4 to the same integer; that is, 4. Finally, the user enters x6 (no quotes). Since the quotes are missing, the eval function does not interpret x6 as a literal string; instead eval treats x6 as a name and attempts to evaluate it. Since no variable named x6 exists, the eval function prints an error message.

The exec function, also from the __builtins__ module, is similar to the eval function. The exec function accepts a string parameter that consists of a Python source statement. The exec function interprets the statement and executes it. Listing 6.17 (myinterpreter.py) plays the role of a rudimentary Python interpreter.

In fact, the examples above that use the eval and exec functions are not advisable in practice. This is because they enable the user to make the program do things the programmer never intended. Python contains functions that call on the operating system to perform tasks. This functionality includes the possibility of erasing ?les or formatting entire disk drives. If the user knows the required Python code to accomplish such devious tasks, he or she could hijack the program and cause havoc. As simple, harmless example, consider the following example run of Listing 6.14 (evalfunc.py):

Turtle graphics on a computer display mimics these actions of placing, moving, and turning a pen on a sheet of paper. It is called Turtle graphics because originally the pen was represented as a turtle moving within the display window. Seymour Papert originated the concept of Turtle graphics in his Logo programming language in the late 1960s (see http://en.wikipedia.org/wiki/Turtle graphics for more information about Turtle graphics). Python includes a Turtle graphics library that is relatively easy to use.

The speed function accepts an integer in the range 0...10. The value 1 represents the slowest speed, and the turtle?s speed increases as the arguments approach 10. Counterintuitively, 0 represents the fastest turtle speed. The speed function will accept a string argument in place of an integer value; the permissible strings correspond to the following numeric values:

The delay function provides another way to affect the time it takes to render an image. Rather than controlling the overall speed of the turtle?s individual movements and/or turns, the delay function speci?es the time delay in milliseconds between drawing incremental updates of the image to the screen. Listing 6.20 (speedvsdelay.py) demonstrates the subtle difference in the way speed and delay functions effect the time to render an image.

Figure 6.9 Listing 6.20 (speedvsdelay.py) demonstrates the different effects of the turtle.speed versus turtle.delay functions. The program renders the bottom 10 lines using the default speed and delay settings. In the next 10 lines the program draws with the slowest speed but without a delay. The program next draws 10 lines at the fastest speed with a 100 millisecond delay between screen updates. The top 10 lines represent the fastest speed with no delay. Note that the animation is still smooth at a slower speed, but the delay function makes the animation choppy at regular time intervals.

When animation is disabled with the tracer function, you should call update at the point the complete image should appear. When the tracer is active it explicitly draws the penstrokes to the screen as the turtle moves. With the tracer turned off, the programmer must ensure all the image becomes visible by calling the update function. Turning the tracer off is the ultimate way to speed up Turtle graphics rendering in Python.

Recall that we can use the from ...import ...notation to import some of the functions that a module has to offer. This allows callers to use the base names of the functions without prepending the module name. This technique becomes unwieldy when a programmer wishes to use a large number of functions from a particular module.

This ?import all? statement is in some ways the easiest to use. The mindset is, ?Import everything because we may need some things in the module, but we are not sure exactly what we need starting out.? The source code is shorter: * is quicker to type than a list of function names, and, within the program, short function names are easier to type than the longer, quali?ed function names. While in the short term the ?import all? approach may appear to be attractive, in the long term it can lead to problems. As an example, suppose a programmer is writing a program that simulates a chemical reaction in which the rate of the reaction is related logarithmically to the temperature. The statement

this statement imports everything from the math module, including a function named degrees which converts an angle measurement in radians to degrees (from trigonometry, 360?= 2? radians). Given the nature of the program, the word degrees is a good name to use for a variable that represents temperature. The two words are the same, but their meanings are very different. Even though the import statement brings in the degrees function, the programmer is free to rede?ne degrees to be a ?oating-point variable (recall rede?ning the print function in Section 2.3). If the program does rede?ne degrees, the math module?s degrees function is unavailable if the programmer later discovers its need. A name collision results if the programmer tries to use the same name for both the angle conversion and temperature representation. The same name cannot be used simultaneously for both purposes.

Observe how assigning the variable x adds the name x to the interpreter?s namespace. Importing just math.sqrt adds sqrt to the namespace. Finally, importing everything from the math module adds many more names. If we attempt to use any of these names in a different way, they will lose their original purpose; for example, the following continues the above interactive sequence:

We say that the ?import everything? statement pollutes the program?s namespace. This kind of import adds many names (variables, functions, and other objects) to the collection of names managed by the program. This can cause name collisions as in the example above with the name degrees, and it makes it more dif?cult to work with larger programs. When adding new functionality to such a program we must be careful not to tread on any names that already exist in the program?s namespace.

Note the m. pre?x attached to the calls of the sqrt and log10 functions. Programmers sometimes use this module renaming import to simplify typing when a module name is long. Listing 6.22 (octogon2.py) is a rewrite of Listing 6.19 (octogon.py) that introduces a new name for the turtle module: t. We say that turtle and t are aliases for the same module. This in effect shortens the quali?ed names for each of the function calls. The fact that Listing 6.22 (octogon2.py) runs faster than Listing 6.19 (octogon.py) has nothing to do with the module name aliasing or shorter quali?ed function names; Listing 6.22 (octogon2.py) runs much faster because it calls the turtle.delay function to speed up the drawing.

The practice of using an alias merely to shorten module name is questionable. It essentially hides a recognized standard name and so renders the code less readable. Fortunately, the ability to alias module names is convenient for other purposes besides shortening module names. Suppose a third-party software vendor develops an alternative to Python?s standard math module. It includes all the functions provided by the math module, with exactly the same names. The company markets it as a drop-in replacement for the math module. The vendor names this module fastmath because the algorithms it uses to implement the mathematical functions are more ef?cient than those used in the math module. As an example, when invoked with the same arguments the fastmath.sqrt and math.sqrt functions compute identical results, but fastmath.sqrt returns its result quicker than math.sqrt.

The names of standard functions are well known to experienced Python developers, so such renaming renders a program immediately less less readable. We should not consider renaming standard functions unless we have a very good reason. Some have found this technique useful for resolving name clashes between two modules that de?ne one or more functions with the same name. Returning to our example from above, suppose we wish to compare directly the performance of math.sqrt to fastmath.sqrt. The process of measuring the relative performance of software is known as benchmarking. We need to have both math.sqrt and fastmath.sqrt available in the same program. The following code performs the benchmark and avoids quali?ed function names:

Side 2

Hypotenuse

(a) math.sqrt(4.5)
(b) math.sqrt(4.5, 3.1)

(c) random.rand(4)
(d) random.seed()

Chapter 7

Writing Functions

As programs become more complex, programmers must structure their programs in such a way as to effectively manage their complexity. Most humans have a dif?cult time keeping track of too many pieces of information at one time. It is easy to become bogged down in the details of a complex problem. The trick to managing complexity is to break down the problem into more manageable pieces. Each piece has its own details that must be addressed, but these details are hidden as much as possible within that piece. These pieces assemble to form the problem?s complete solution.

So far all of the code we have written has been placed within a single block of code. That single block may have contained sub-blocks for the bodies of structured statements like if and while, but the program?s execution begins with the ?rst statement in the block and ends when the last statement in that block is ?nished. Even though all of the code we have written has been limited to one, sometimes big, block, our programs all have executed code outside of that block. All the functions we have used?print, input, sqrt, randrange, etc.?represent blocks of code that some other programmers have written for us. These blocks of code have a structure that makes them reusable by any Python program.

? It is dif?cult to write correctly. Complicated monolithic code attempts to do everything that needs to done within the program. The indivisible nature of the code divides the programmer?s attention amongst all the tasks the block must perform. In order to write a statement within a block of monolithic code the programmer must be completely familiar with the details of all the code in that block. For instance, we must use care when introducing a new variable to ensure that variable?s name is not already being used within the block.

? It is dif?cult to debug. If the sequence of code does not work correctly, it may be dif?cult to ?nd the source of the error. The effects of an erroneous statement that appears earlier in a block of monolithic code may not become apparent until a possibly correct statement later uses the erroneous statement?s incorrect result. Programmers naturally focus their attention ?rst to where they observe the program?s misbehavior. Unfortunately, when the problem actually lies elsewhere, it takes more time to locate and repair the problem.

We can write our own functions to divide our code into more manageable pieces. Using a divide and conquer strategy, we can decompose a complicated block of code into several simpler functions. The original code then can do its job by delegating the work to these functions. This process of is known as functional decomposition. Besides their code organization aspects, functions allow us to bundle functionality into reusable parts. In Chapter 6 we saw how library functions can dramatically increase the capabilities of our programs. While we should capitalize on library functions as much as possible, often we need a function exhibiting custom behavior unavailable in any standard function. Fortunately, we can create our own functions. Once created, we can use (call) these functions in numerous places within a program. If the function?s purpose is general enough and we write the function properly, we can reuse the function in other programs as well.

block

Figure 7.2 Calling relationships among functions during the execution of Listing 7.1 (doublenumber.py). Time ?ows from top to bottom. A vertical bar represents the time in which a block of code is active. Observe that functions are active only during their call. The shaded area within in block represents the time that block is idle, waiting for a function call to complete. Right arrows (?) represent function calls. Function calls show parameters, where applicable. Left arrows (?) represent function returns. Function returns show return values, if applicable.

Program Execution (Time)

1. The program?s execution begins with the ?rst line in the ?naked? block; that is, the block that is not part of the function de?nition. The program thus executes the assignment statement that calls the double function with the argument 3. Before the assignment can happen, the program?s execution transfers to the body of the double function. The code within double executes, which simply returns the product of 2 and the parameter passed in (in this case 3).

The empty parentheses in count_to_10?s de?nition indicates that the function does not accept any parameters from its caller. Also, the absence of a return statement indicates that this function communicates no information back to its caller. Such functions that get no information in and provide no results can be useful for the effects they achieve (in this case just printing the numbers 1 to 10.

Our doublenumber and count_to_10 functions are a bit underwhelming. The doublenumber function could be eliminated, and each call to doublenumber could be replaced with a simple variation of the code in its body. The same could be said for the count_to_10 function, although it is convenient to have the simple one-line statement that hides the complexity of the loop. These examples serve simply to familiarize us with the mechanics of function de?nitions and invocations. Functions really shine when our problems become more complex.

If a function is written to accept information from the caller, the caller must supply the information in order to use the function. The caller communicates the information via one or more parameters as required by the function. The count_to_10 function does us little good if we sometimes want to count up to a different number. Listing 7.4 (countton.py) generalizes Listing 7.3 (countto10func.py) to count as high as the caller needs.

we refer to n as the formal parameter. A formal parameter is used like a variable within the function?s body, and it is local to the function. A formal parameter is the parameter from the perspective of the function de?nition. During an invocation of double, such as double(2), the caller passes actual parameter 2. The actual parameter is the parameter from the caller?s point of view. A function invocation, therefore, binds the actual parameters sent by the caller to their corresponding formal parameters.

The polygon function does not have a return statement, so it does not communicate a result back to its caller. This program introduces the begin_fill and end_fill functions from the turtle module. When placed around code that draws a closed ?gure, these functions ?ll the shape with the current drawing color. Figure 7.3 shows a screenshot of a sample run of Listing 7.6 (regularpolygon.py).

The midpoint function returns only one result, but that result is a tuple containing two pieces of data. The mid variable in Listing 7.7 (midpoint.py) refers to a single tuple object. We will examine tuples in more detail in Chapter 11, but for now it is useful to note that we also can extract the components of the returned tuple into individual numeric variables as follows:

consider the fraction $\frac{18}{24}$. The greatest common divisor of 18 and 24 is 6, and we can compute the reduced fraction by dividing the numerator and the denominator by 6: $\frac{18\div 6}{24\div 6} = \frac{3}{4}$. The GCD function has applications in other areas besides reducing fractions to lowest terms. Consider the problem of dividing a piece of plywood 24 inches long by 18 inches wide into square pieces of maximum size in integer dimensions, without wasting any material. Since the GCF(24, 18) = 6, we can cut the plywood into twelve 6 inch × 6 inch square pieces as shown in Figure 7.4. If we cut the plywood into squares of any other size without wasting the any of the material, the squares would have to be smaller than 6 inches × 6 inches; for example, we could make forty-eight 3 inch × 3 inch squares as shown in pieces as shown in Figure 7.5. If we cut squares larger than 6 inches × 6 inches, not all the plywood can be used to make the squares. Figure 7.6. shows

```
# Determine the smaller of num1 and num2
min = num1 if num1 < num2 else num2
# 1 definitely is a common factor to all ints
largest_factor = 1
for i in range(1, min + 1):
```

This function is named gcd and expects two integer arguments. Its formal parameters are named num1 and num2. It returns an integer result. The function uses three local variables: min, largest_factor, and i. Local variables have meaning only within their scope. The scope of a local variable is the point within the function?s block after its ?rst assignment until the end of that block. This means that when you write a function you can name a local variable without concern that its name may be used already in another part of the program. Two different functions can use local variables named x, and these are two different variables that have no in?uence on each other. Anything local to a function de?nition is hidden to all code outside that function de?nition. Since a formal parameter also is local to its function, you can reuse the names of formal parameters in different functions without a problem.

Notice that Listing 7.9 (localplay.py) numbers each print statement in the order of its appearance in the program?s source code. When executing the program the interpreter will executes the source code line by line, top to bottom. It executes each statement in turn, but function de?nitions are special?a function de?nition packages code into an executable unit to be executed later. The code within a function de?nition executes only when invoked by a caller. Listing 7.9 (localplay.py), therefore, will not execute the print statements in the order listed in the source code. Listing 7.9 (localplay.py) prints

The printing statements 1, 2, 5, and 6 all refer to the variable x de?ned outside of functions fun1 and fun2. When fun1 and fun2 assign to a variable named x, this x is local to its respective function. The assignments within fun1 and fun2 do not affect the variable involved in printing statements 1, 2, 5, or 6. Note that the printing statements within fun1 and fun2 do not execute until the program actually calls the functions. That is why printing statements 2 and 4 appear out of numerical order in the program?s execution. In the end, the last printing statement, number 6, prints the value of the original x variable that the program assigned in its ?rst line. The code within fun1 and fun2, as they currently are written, cannot disturb the value of this external variable. (In Section 8.1 we shall see how a function can gain access to a variable de?ned outside the function?s de?nition.)

In the code we have considered in earlier chapters, the name of a variable uniquely identi?ed it and distinguished that variable from another variable. It may seem strange that now we can use the same name in two different functions within the same program to refer to two distinct variables. The block of statements that makes up a function de?nition constitutes a context for local variables. A simple analogy may help. In the United States, many cities have a street named Main Street; for example, there is a thoroughfare named Main Street in San Francisco, California. Dallas, Texas also has a street named Main Street. Each city and town provides its own context for the use of the term Main Street. A person in San Francisco asking ?How do I get to Main Street?? will receive the directions to San Francisco?s Main Street, while someone in Dallas asking the same question will receive Dallas-speci?c instructions. In a similar manner, assigning a variable within a function block localizes its identity to that function. We can think of a program?s execution as a person traveling around the U.S. When in San Francisco, all references to Main Street mean San Francisco?s Main Street, but when the traveler arrives in Dallas, the term Main Street means Dallas? Main Street. A program?s thread of execution cannot execute more than one statement at a time, which means it uses its current context to interpret any names it encounters within a statement. Similarly, at the risk of overextending the analogy, a person cannot be physically located in more than one city at a time. Furthermore, Main Street may be a bustling, multi-lane boulevard in one large city, but a street by the same name in a remote, rural township may be a narrow dirt road! Similarly, two like-named variables may mean two completely different things. A variable named x is one function may represent an integer, while a different function may use a string variable named x.

variables and parameters when the function begins executing. When a function invocation is complete and control returns to the caller, the function?s variables and parameters go out of scope, and the run-time environment ensures that the memory used by the local variables is freed up for other purposes within the running program. This process of local variable allocation and deallocation happens each time a caller invokes the function.

This call uses the variable val as its ?rst actual parameter and the literal value 24 as its second actual parameter. As with the standard Python functions, we can pass variables, expressions, and literals as actual parameters. The function then computes and returns its result. Here, this result is assigned to the variable factor.

How does the function call and parameter mechanism work? It?s actually quite simple. The executing program binds the actual parameters, in order, to each of the formal parameters in the function de?nition and then passes control to the body of the function. When the function?s body is ?nished executing, control passes back to the point in the program where the function was called. The value returned by the function, if any, replaces the function call expression. The statement

assigns an integer value to factor. The expression on the right is a function call, so the executing program invokes the function to determine what to assign. The value of the variable val is assigned to the formal parameter num1, and the literal value 24 is assigned to the formal parameter num2. The body of the gcd function then executes. When the return statement in the body of gcd executes, program control returns back to where the function was called. The argument of the return statement becomes the value assigned to factor.

The execution of this statement would evaluate x - 2 and bind its value to num1. num2 would be assigned 24. The result of the call is then assigned to x. Since the right side of the assignment statement is evaluated before being assigned to the left side, the original value of x is used when calculating x - 2, and the function return value then updates x.

This example shows two invocations in one statement. Since the function returns an integer value, its result can itself be used as an actual parameter in a function call. Passing the result of one function call as an actual parameter to another function call is called function composition. Function composition is nothing new to us, consider the following statement which prints the square root of 16:

calls the main function which in turn directly calls several other functions (get_int, print, and gcd). The get_int function itself directly calls int and input. In the course of its execution the gcd function calls range. Figure 7.7 contains a diagram that shows the calling relationships among the function executions during a run of Listing 7.12 (gcdwithmain.py).

When a caller invokes a function that expects a parameter, the caller must pass a parameter to the function. The process behind parameter passing in Python is simple: the function call binds to the formal parameter the object referenced by the actual parameter. The kinds of objects we have considered so far?integers, ?oating-point numbers, and strings?are classi?ed as immutable objects. This means a programmer cannot change the value of the object. For example, the assignment

Program Execution

(Time)

The of?cial Python style guide recommends using """ for docstrings rather than '''?see https://www.python.org/dev/peps/pep-0008/. In fact, since the docstring for our gcd function above is only one line of text, the normal ' and " quotation marks are adequate to specify its docstring. We will follow the convention of using """ to delimit our docstrings, even when expressing a single-line documentation.

```
x1 is the x coordinate of the first point
y1 is the y coordinate of the first point
x2 is the x coordinate of the second point
y2 is the y coordinate of the second point
Returns the distance between (x1,y1) and (x2,y2)
"""
...
```

? The complete work of the program is no longer limited to one block of code. The main function is responsible for generating prime candidates and printing the numbers that are prime. main delegates the task of testing for primality to the is_prime function. Both main and is_prime individually are simpler than the original monolithic code. Also, each function is more logically coherent. A function is coherent when it is focused on a single task. Coherence is a desirable property of functions. If a function becomes too complex by trying to do too many different things, it can be more dif?cult to write correctly and debug when problems are detected. A complex function usually can be decomposed into several, smaller, more coherent functions. The original function would then call these new simpler functions to accomplish its task. Here, main is not concerned about how to determine if a given number is prime; main simply delegates the work to is_prime and makes use of the is_prime function?s ?ndings. For is_prime to do its job it does not need to know anything about the history of the number passed to it, nor does it need to know the caller?s intentions with the result it returns.

```
def is_prime(n):
result = True # Provisionally, n is prime
root = round(sqrt(n)) + 1
# Try all potential factors from 2 to the square root of n
trial_factor = 2
while result and trial_factor <= root:
result = (n % trial_factor != 0 ) # Is it a factor?
trial_factor += 1 # Try next candidate
return result
```

uses main?s height as an actual parameter and height happens to be the name as the formal parameter is simply a coincidence. The function call binds the value of main?s height variable to the formal parameter in tree also named height. The interpreter can keep track of which height is which based on the function in which it is being used.

Recall from Listing 3.2 (imprecise.py) that ?oating-point numbers are not mathematical real numbers; a ?oating-point number is ?nite, and is represented internally as a quantity with a binary mantissa and exponent. Just as we cannot represent 1/3 as a ?nite decimal in the base-10 number system, we cannot represent 1/10 exactly in the binary (base 2) number system with a ?xed number of digits. Often, no problems arise from this imprecision, and in fact many software applications have been written using ?oating-point numbers that must perform precise calculations, such as directing a spacecraft to a distant planet. In such cases even small errors can result in complete failures. Floating-point numbers can and are used safely and effectively, but not without appropriate care.

Listing 7.21 (bad?oatcheck.py) demonstrates that the == and != operators are of questionable worth when comparing ?oating-point values. The better approach is to check to see if two ?oating-point values are close enough, which means they differ by only a very small amount. When comparing two ?oating-point numbers x and y, we essentially must determine if the absolute value of their difference is small; for example, |x ?y| < 0.00001. We can construct an equals function and incorporate the fabs function introduced in 6.4. Listing 7.22 (?oatequalsfunction.py) provides such an equals function.

The third parameter, named tolerance, speci?es how close the ?rst two parameters must be in order to be considered equal. The == operator must be used for some special ?oating-point values such as the ?oating-point representation for in?nity, so the function checks for == equality as well. Since Python uses short-circuit evaluation for Boolean expressions involving logical OR (see 4.2), if the == operator indicates equality, the more elaborate check is not performed.

```
turtle.color("green")
for x in range(10):
turtle.penup()
turtle.setposition(-200, y)
turtle.pendown()
turtle.forward(400)
y += 10
```

? Code duplication results in code that is more dif?cult to maintain. Most commercial software is not static; developers constantly work on newer versions. Newer versions typically provide bug ?xes and add features. Upon repairing a logic error or enhancing a section of duplicated code, a programmer must track down all the other code sections within the system that duplicate the modi?ed section. Failure to do so can introduce inconsistency into the program?s behavior.

8. The fact that we found one difference in this small collection of test cases justi?es using the standard math.sqrt function instead of our custom function. Generally speaking, if you have the choice of using a standard library function or writing your own custom function that provides the same functionality, choose to use the standard library routine. The advantages of using the standard library routine include:

? If you write your own custom code, you must thoroughly test it to ensure its correctness; standard library code, while not immune to bugs, generally has been subjected to a complete test suite. Additionally, library code is used by many developers, and thus any lurking errors are usually exposed early; your code is exercised only by the programs you write, and errors may not become apparent immediately. If your programs are not used by a wide audience, bugs may lie dormant for a long time. Standard library routines are well known and trusted; custom code, due to its limited exposure, is suspect until it gains wider exposure and adoption.

Listing 7.25 (squarerootcomparison.py) uses our equals function from Listing 4.7 (?oatequals.py). Observe that the tolerance used within the square root computation is smaller than the tolerance main uses to check the result. The main function, therefore, uses a less strict notion of equality. The output of Listing 7.25 (squarerootcomparison.py) is

Chapter 8

More on Functions

? The same variable name can be used in different functions without any con?ict. The interpreter derives all of its information about a local variable from that variable?s de?nition within the function. If the interpreter attempts to execute a statement that uses a variable that has not been de?ned, the interpreter issues a run-time error. When executing code in one function the interpreter will not look for a variable de?nition in another function. Thus, there is no way a local variable in one function can interfere with a local variable de?ned in another function.

Listing 8.1 (globalcalculator.py) uses global variables result, arg1, and arg2. These names no longer appear in the main function. The program accesses and/or modi?es these global variables in four different functions: get_input, report, add, and subtract. The global keyword within a function?s block of code identi?es the variables which are global variables. Notice that if a function uses a global variable without assigning its value, the global declaration is not necessary. This is because variable assignment is variable de?nition, and a local variable must be de?ned within a function.

A function may be use a global variable without declaring it with the global keyword if the function does not assign a variable of that name anywhere in its body. A function that assigns a global variable must declare that variable as global with the global keyword.

The exclusion of global variables from a function leads to functional independence. A function that depends on information outside of its scope to correctly perform its task is a dependent function. When a function operates on a global variable it depends on that global variable being in the correct state for the function to complete its task correctly. Nontrivial programs that contain many dependent functions are more dif?cult debug and extend. A truly independent function that use no global variables and uses no programmer-de?ned functions to help it out can be tested for correctness in isolation. Additionally, an independent function can be copied from one program, pasted into another program, and work without modi?cation. Functional independence is a desirable quality.

Its behavior is totally predictable. Furthermore, increment does not modify any global variables, meaning its code all by itself cannot in any way in?uence the overall program?s behavior. We say that increment is a pure function. A pure function cannot perform any input or output (for example, use the print or input statements), nor may it use global variables. While increment is pure, the compute function is impure. The following function is impure also, since it performs output:

Default arguments allow programmers to provide a highly tunable function that offer a simpler interface for its typical uses. Recall Listing 7.6 (regularpolygon.py) that uses Turtle graphics to draw a regular polygon with speci?ed number of sides, of a given size, location, and color. Listing 8.2 (enhancedpolygon.py) enhances Listing 7.6 (regularpolygon.py) by adding the ability to optionally ?ll the polygon with a color. Since the number of parameters is becoming unwieldy for a caller to manage, the color and ?ll directives are speci?ed as optional.

Observe that the caller must pass at least four parameters and optionally can pass one or two more. The idea here is that when drawing a polygon it is important to know how many sides it has, the length of each side, and its location. Such a polygon will consist of a black outline of the shape. Callers can enhance the figure by specifying a color for the outline. Finally, a caller can indicate that the polygon should be filled. The color and fill are optional extras that are required just to draw a basic polygon. Figure 8.1 shows the shapes drawn by Listing 8.2 (enhancedpolygon.py).

with color bound to True will generate an exception because True is not one of the color strings like "red", "blue", etc. that the turtle.color function expects. The polygon function therefore will fail. Default parameters always substitute for missing parameters from back to front in a function?s parameter list.

factorial(6) = 6 * factorial(5)
= 6 * 5 * factorial(4)
= 6 * 5 * 4 * factorial(3)
= 6 * 5 * 4 * 3 * factorial(2)
= 6 * 5 * 4 * 3 * 2 * factorial(1)
= 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
= 6 * 5 * 4 * 3 * 2 * 1 * 1
= 6 * 5 * 4 * 3 * 2 * 1
= 6 * 5 * 4 * 3 * 2
= 6 * 5 * 4 * 6
= 6 * 5 * 24
= 6 * 120
= 720

factorial(6) = 6 * factorial(5)
= 6 * 5 * factorial(4)
= 6 * 5 * 4 * factorial(3)
= 6 * 5 * 4 * 3 * factorial(2)
= 6 * 5 * 4 * 3 * 2 * factorial(1)
= 6 * 5 * 4 * 3 * 2 * 1
= 6 * 5 * 4 * 3 * 2
= 6 * 5 * 4 * 6
= 6 * 5 * 24
= 6 * 120
= 720

Figure 8.2 A graphial representation of the chain of recursive factorial invocations when executing the statement print(factorial(6)), where the factorial function is from Listing 8.3 (factorialtest.py) with the condition optimized to n < 2. The vertical bars represent the time a function invocation is active. The shaded area within each bar represents the time that the function, while still active, is idle, waiting for a function it calls to complete. Note that during the process of recursion all earlier function invocations in the call chain remain active (but idle) until all the functions further down the call chain return.

Program Execution (Time)

Each recursive invocation must bring the function?s execution closer to its base case. The factorial function calls itself in the else block of the if/else statement. Its base case is executed if the condition of the if statement is true. Since the factorial is de?ned only for nonnegative integers, the initial invocation of factorial must be passed a value of zero or greater. A zero parameter (the base case) results in no recursive call. Any other positive parameter results in a recursive call with a parameter that is closer to zero than the one before. The nature of the recursive process progresses towards the base case, upon which the recursion terminates.

Which factorial function is better, the recursive or non-recursive version? Generally, if both the recursive and non-recursive functions implement the same basic algorithm, the non-recursive function will be more ef?cient. A function call is a relatively expensive operation compared to a variable assignment or comparison. The body of the non-recursive factorial function invokes no functions, but the recursive version calls a function?it calls itself?during all but the last recursive invocation. The iterative version of factorial is therefore more ef?cient than the recursive version.

Recall the gcd function from Section 7.4. It computed he greatest common divisor (also known as greatest common factor) of two integer values. It works, but it is not very ef?cient. Listing 8.5 (gcd.py) uses a better algorithm. It is based on one of the oldest algorithms known, attributed to Euclid around 300 B.C.

Note that this gcd function is recursive. The algorithm it uses is much different from our original iterative version. Because of the difference in the algorithms, this recursive version is actually much more ef?cient than our original iterative version. A recursive function, therefore, cannot be dismissed as inef?cient just because it is recursive. We will revisit recursion in Section 15.5.

A common problem is computing the nth Fibonacci number. Zero is the 0th, 1 is the 1st, 1 is also the 2nd, 2 is the 3rd, 3 is the 4th, 5 is the 5th, etc.

A recursive Python function to compute the nth Fibonacci number follows easily from the de?nition of the Fibonacci sequence:

In a function de?nition we can package functionality that can be used in many different places within a program. We have yet to see how we can easily reuse our function de?nitions in other programs. For example, our is_prime function in Listing 7.15 (primefunc.py) works well within Listing 7.15 (primefunc.py), and we could put it to good use in other programs that need to test primality (encryption software, for example, makes heavy use of prime numbers). We could use the copy-and-paste feature of our favorite text editor to copy the is_prime function de?nition from Listing 7.15 (primefunc.py) into the new encryption program we are developing. It is possible to reuse a function in this way only if the function de?nition does not use any programmer-de?ned global variables nor any other programmer-de?ned functions. If a function does use any of these programmer-de?ned external entities, we must include these dependencies as well in the new code for the function to viable. Said another way, the code in the function de?nition ideally will use only local variables and parameters. Such a function truly is independent and easily transportable to other programs.

The notion of copying source code from one program to another is not ideal, however. It is too easy for the copy to be incomplete or otherwise incorrect. Furthermore, such code duplication is wasteful. If 100 programs on a particular system all need to use the is_prime function, under this scheme they must all include the is_prime code. This redundancy wastes space. Finally, in perhaps the most compelling demonstration of the weakness of this copy-and-paste approach, what if a bug is discovered in the is_prime function that all 100 programs are built around? When the error is discovered and ?xed in one program, the other 99 programs will still contain the bug. Their source code must be updated, and it may be dif?cult to determine which ?les need to be ?xed. The problem becomes much worse if the code has been released to the general public. It may be impossible to track down and correct all the copies of the faulty function. The

Figure 8.3 The recursive computation of fibonacci(5). Each rectangle represents an invocation of the fibonacci fuunction. The call at the top of the diagram represents the initial call of fibonacci(5). An arrow pointing down indicates the argument being passed into an invocation of fibonacci, and an arrow pointing up represents the value returned by that invocation. An invocation of fibonacci with no arrow pointing down away from the invocation represents a base case; observe that any invocation receiving a 0 or 1 is a base case. We see that the recursive process for fibonacci(5) invokes the function a total of 15 times.

situation would be the same if a correct is_prime function were updated to be made more ef?cient. The problem is this: all the programs using is_prime de?ne their own is_prime function; while the function de?nitions are meant to be identical, there is no mechanism tying all these common de?nitions together. We really would like to reuse the function as is without copying it.

Observe the docstring at the top of the Listing 8.6 (primecode.py) module. This provides external documentation that can be used as an overview to the functions the module makes available. As we saw in Section 7.3, we can use the help function to reveal the information developers have provided in their docstrings. The following interactive sequence demonstrates how the help function accesses information in the primecode module?s docstring:

The ability to pass a function object to another function is a powerful concept. It provides one mechanism that enables programmers to develop code that interoperates with a larger software framework. Python?s Turtle graphics environment is one such framework. Our Turtle graphics examples to this point have involved drawing pictures in the graphics window with no provision for user interaction. With our knowledge of functions as parameters we now have the tools to write graphical programs that respond to mouse clicks and key presses.

To understand the concept of buffering, consider the task of building a wall with bricks. Estimates indicate that the wall will require about 1,350 bricks. Once we are ready to start building the wall we can drive to the building supply store and purchase a brick. We then can drive to the job site and place the brick in its appropriate position using mortar as required. Now we are ready to place the next brick, so we must drive back to the store to get the next brick. We then drive back to the job site and set the brick. We repeat this process about 1,350 times.

In this analogy, the transport vehicle is the buffer. The print function uses a special place in memory called a buffer. Like the vehicle used to transport our bricks, the memory buffer has a ?xed capability. The print statement writes the individual characters to display to the buffer, and when the buffer is full, the function in one operation sends all the characters in the buffer out to the display screen. As with the bricks, this is more ef?cient than sending just one character at a time to the display. This buffering process can speed up signi?cantly the output of programs that display a lot text.

Ordinarily, at the end of its output the print function ?ushes the last characters from its buffer, even if the buffer is not full. The is analogous to the last load of bricks that may not be a full load. In some situations it is necessary to ensure the buffer is ?ushed to the screen even before it is full. Some graphical IDEs such as WingIDE manage their own console windows. The print function executed from an application launched from the IDE sends its output to the IDE?s console. Under certain circumstances programs such as Listing 8.10 (trackmouse.py) may not print a full line of text immediately when executed from a graphical IDE. In the case of Listing 8.10 (trackmouse.py) the full printed output may be a mouse click behind. The interplay between the graphical IDE and the executing graphical application under development can interfere with the normal operation of print. If you experience the print function not displaying its full line of text, adding the flush keyword argument with the value True often can correct print?s behavior.

The turtle.onscreenclick function allows a programmer to register a function with the graphical framework. The registered function (for example, report_position in Listing 8.10 (trackmouse.py) or draw_square in Listing 8.11 (placesquares.py)) is known as a callback function because the graphical framework will call back the function when a mouse click event occurs.

It is important to note that no code within Listing 8.10 (trackmouse.py) or Listing 8.11 (placesquares.py) actually calls the callback function. Neither does the turtle.onscreenclick function call the callback function; the call to onscreenclick merely registers the callback function with the graphical framework. It is the responsibility of the graphical framework to call the callback function. The graphical framework tracks mouse events, and so it is the framework that must call the callback function with the appropriate (x,y) location.

Consider a very simple connect-the-dots game. In such a game two players compete on a two-dimensional rectangular grid of dots, as shown in Figure 8.4. Typically played with pen and paper, the number of rows and columns can vary, sometimes limited only by the size of the paper. The rules of the game are simple. Two players take turns drawing a line between adjacent dots. A player can draw a horizontal or vertical connecting line; diagonal lines are forbidden. Turns alternate until a player?s line completes a square. At that point the player that drew the line becomes the square?s owner and marks the square with an initial. As a bonus for winning the square, the square owner can immediately take another turn. Figure 8.5 shows a game in progress. The player owning the most squares at the end of the game is the winner.

We will implement a simpli?ed version of the connect-the-dots game so that we can exercise some of the concepts introduced in this chapter. We will limit our game to a 3×3 grid because we have yet to cover the concepts required to maintain a large amount of data at one time. (Lists in Chapter 10 provide a way to build connect-the-dot grids of arbitrary size.) Even in its simpli?ed form, our program will be the most complex we have considered so far. This is because instead of writing the application in one large source ?le, we intentionally will divide our implementation into two parts:

Y

X

? The ?nal set of global variables (_lefttop_owner, _righttop_owner, etc.) store the owners of the squares. They begin with the special value None, which means no one owns any squares at the start of a game. When a player completes a square, the game engine assigns "X" or "Y" to the appropriate variable.

? The add_line and initialize_board functions are available to the presentation and can modify the state of the game. The presentation calls add_line when a player attempts to draw a line. The add_line function calls the _update_squares function to check whether a new line completes a square. The initialize_board function resets all the global variables in preparation of a new game. The presentation calls the add_line function when a player provides input representing a move. The presentation calls the initialize_board function to begin a new game. The docstrings within these two functions provide more information about their purpose and use.

? The check_line, square_owner, winner, and current_player functions cannot change the state of the game. They serve to report the state of the game to the presentation. The presentation can call these functions to determine how to change its appearance. The docstrings within these functions provide more information about their purpose and use.

Listing 8.13 (dot3x3turtle.py) uses its own global variables to determine which dots the user selects when making a line. Note that the presentation registers the function mouse_click for the graphical framework to call when the user clicks the mouse button. The mouse_click function must determine if the click occurred over a graphical dot. It uses the initial_dot global to keep track if this is the initial endpoint of a line or the terminal endpoint of a line. The mouse_click function than calls the line_name function to translate the information about the two graphical dots into the proper line string to send to the game engine?s add_line function. If the player attempts an illegal move, the presentation pops up a message box alerting the player.

This separation of concerns, dividing the game control logic from the presentation, requires more effort to implement properly than does putting all the code together in one module. The payoff of this extra work is increased ?exibility. This design allows us to decouple the engine from the presentation. We can ?un-plug? this graphical user interface from the game engine and ?plug in? a completely different presentation without touching the game engine code. Listing 8.14 (dot3x3text.py) uses the exact same game engine as Listing 8.13 (dot3x3turtle.py) but provides a text-output, keyboard-only-input interface to the players.

One of the primary bene?ts of functions is that we can write a function?s code once and invoke it from many different places within the program (and even invoke it from other programs). Ordinarily, in order to call a function, we must know its name. Almost all the examples we have seen have invoked a function via its name. Listing 8.9 (arithmeticeval.py) in Section 8.5 provided examples of invoking functions without using their names directly. There we saw a function named evaluate that accepts a function as a parameter:

The evaluate function calls f. The question is, what function does evaluate call? The name f refers to one of evaluate?s parameters; there is no separate function named f speci?ed by def within Listing 8.9 (arithmeticeval.py). The answer, of course, is that evaluate invokes the function passed in from the caller. The function named main in Listing 8.9 (arithmeticeval.py) calls evaluate passing the add function on one occasion and the multiply function on another.

The lambda keyword signi?es the de?nition of an unnamed function; thus, the ?rst argument being passed to evaluate is indeed a function. Notice that result of passing the lambda expression here is the same as passing the multiply function from Listing 8.9 (arithmeticeval.py)?both compute the product of the two parameters.

The expression following the colon (:) in a lambda expression cannot be a Python statement. The conditional expression, for example, is acceptable, but an if/else statement is illegal. The expression?s value is what the anonymous function returns, but the keyword return itself may not appear with the expression. Assignments are not possible within lambda expressions, and loops are not allowed. Note that a lambda expression can involve one or more function invocations, so the lambda expression in the following statement is legal:

passes just three parameters to evaluate: a function and two integer values. The main function?s local variable a is not passed as a parameter; instead, it is embedded within the lambda code of the ?rst parameter. The variable a is encoded into the lambda expression. We say the function de?nition (lambda expression) captures the variable a. When evaluate invokes the function sent by the caller, evaluate has no access to a variable named a. The a involved in the conditional expression is captured from main. This is an example of a closure transporting a captured variable into a function call.

make_adder?s loc_var local variable should no longer exist. The function that make_adder returns, however, uses loc_var in its computation. This means the function that make_adder returns forms a closure that captures make_adder?s local variable loc_var. In the output of Listing 8.16 (makeadder.py) we can see that function f still has knowledge of loc_val?s value:

? quad?represents the mathematical quadratic function f(x) = 1

2x2 + 3. A caller can pass this to the plot function for rendering.

Figure 8.7 A screenshot of the plot function rendering the mathematical functions f(x) = 1 2x2 + 3 (red),

Programmers generally do not use the next function directly. Instead, they leave it to the for statement to call next behind the scenes. A generator object is one example of an iterable object. We learned in Section 5.3 that Python?s for statement is built to work with an iterable object. The for statement, therefore, can iterate over the sequence of values produced by a generator object. Listing 8.20 (forgenerator.py) shows that the for statement works naturally with the generator our gen function produces.

The yield statement within the function generates the values of the sequence. It is uncommon to provide separate yield statements for each value the generator is to produce. More likely, one yield statement executes multiple times within a loop. Listing 8.21 (regulargenerator.py) shows a more common scenario.

The expression range(0, 10) does not return a generator object but instead creates and returns a range object. Furthermore, the interative sequence shows that range is not a function at all; it is a class. In reality, the expression range(0, 10) calls the range class constructor. We will not be concerned with such details about objects until Chapter 13. For now we will be content with the understanding that the for statement is designed to work with iterable objects, and generators and range objects are both instances of iterable objects.

Our myrange function may be interesting, but it offers no advantage over the built-in range expression. It is time to use a generator in a more interesting way. Recall Listing 7.15 (primefunc.py) that uses a function named is_prime in the course of printing the prime numbers within a range speci?ed by the user. What if we wish to print only the prime numbers within a range that end with the digit 3? What if wish to add up all the prime numbers within a given range? A generator is ideal for implementing the more modular and ?exible code required to generate prime numbers independent of how a program uses them. Listing 8.23 (generatedprimes.py) uses a generator function to produce the sequence of prime numbers. The caller (main) then can select which values to print and sum the numbers in a sequence. In Listing 8.23 (generatedprimes.py) we further tune the is_prime function from the observations that two is the only even prime number and that no prime number except two may have a factor that is even. Applying these facts allows us to cut by one-half the potential factors to consider within the loop.

In Chapter 7 we introduced functional decomposition?a fancy term that means programmers can use functions to break up a large, complex, monolithic program into smaller, more manageable pieces. The code within a function is somewhat insulated from the rest of the program, in that a caller can in?uence the behavior of a function only via the function?s parameters and any global variables the function may use. Any local variables the function uses are invisible to code outside of the function.

Local functions can access the local variables and other local functions de?ned by enclosing function. As with the global functions we have seen before this section, any variable de?ned within a local function is a local variable of that function. If we need a local function to modify a variable de?ned in its outer scope (its enclosing function), we must declare the variable as nonlocal. The global keyword declares a variable as truly global, so we cannot use the global keyword in place of nonlocal in this situation.

Note that the create_counter function returns a function; it returns its own local function. This returned local function ?remembers? the value of its enclosing function?s local variable count; thus, it represents a closure (see Section 8.7). Since the count variable in the enclosing function is not global, no outside code can modify the count variable except by calling the function that create_counter returns.

It may appear that create_counter is similar to a generator function (see Section 8.8). Unfortunately the create_counter function in Listing 8.25 (localcounter.py) does not create a generator object, as it has no yield statement. This means we cannot use it in a for statement, and it does not work with the global next function.

If you are unfamiliar with calculus, all you need to know is that the derivative of a function is itself a function; the above formula shows how to transform a function into its derivative. The process of computing a derivative is known as differentiation. The limh?0 notation indicates that the answer becomes more precise as the value h gets closer to zero. Letting h be exactly zero would result in division by zero, which is unde?ned. The trick is to make h as small as possible, keeping in mind that the computer?s ?oating-point values have limitations.

Our derivative function allows us to compute the derivative of a function at a given value. This is known as numerical differentiation. Another approach (the one emphasized in calculus courses) uses symbolic differentiation. Symbolic differentiation transforms the formula for a function into a different formula. The details of symbolic differentiation are beyond the scope of this text, but we will use one of its results for a particular function to check our computed numerical results.

$$f(x) = 3x2 +5$$

Listing 8.26 (derivative.py) uses the derivative function on f(x) = 3x2 + 5 and compares its results with the known solution, f ?(x) = 6x.

During testing we would like every call to any of these functions (and any of the possibly hundreds more functions that make up the application) to report the function?s arguments and return value. Ideally we would write this information to a log ?le, but, here we simply will print the information. One approach would be to modify each function temporarily for testing purposes, as Listing 8.28 (poorlydecorated.py) demonstrates

Note that we print information at the beginning of the function?s execution, indicated by the pre?x >>> (for entering the function). We also print some information pre?xed with <<< at the end of the function?s execution (for leaving the function). This allows our output to be readable even for functions that do their own printing like star_rect. It also allows us to better track recursive function invocations like gcd.

2. We must ensure we modify every one of the functions correctly; since each function has different parameter names we simply cannot copy and paste code and use it as is. For functions with return statements that involve an expression with multiple components (as in the original max function), we must introduce a local variable to print and then return that variable. We must be particularly careful, as mistakes here can alter the function?s logic.

Another serious problem involves functions over which we have no control. In our code, for example, we cannot (or, better, should not) modify the randrange function to provide the call details we seek. The randrange function is part of a library function maintained by someone else. If we attempt to change the contents of the random module?s source code, there is a good chance we could introduce an error. Modifying code within a library, therefore, reduces its trustworthiness. Enhancing randrange by modifying its source code is not a viable option.

This show_call_and_return_details accepts a single function as a parameter. It ?wraps? the function in some additional code within a local function. The show_call_and_return_details function ?nally returns this local function to its caller. Listing 8.29 (simpledecorator.py) shows how we can use the show_call_and_return_details function to achieve our goal of printing details about function calls without modifying the code of the individual functions.

In Listing 8.29 (simpledecorator.py), the show_call_and_return_details function is known as a decorator. A decorator does not change the way a function works; it simply adds some ?decoration? to the function, usually to augment the function?s behavior. A decorator ?wraps? a function passed to it. Observe that a decorator cannot modify the inner workings of a function. Our example shows that a decorator can do some preprocessing (activity before calling the function it wraps) and postprocessing (activity after calling the function it wraps). A particular decorator could call a different function instead of the function it wraps.

In Section 8.5 we saw how we can pass functions as arguments to other functions. Section 8.7 showed how a function could serve as a return value from another function. The functools module provides an interesting function named partial that accepts a function as its ?rst parameter and one or more other parameters. The partial function returns a new function that is behaviorally related to the original function passed to it. To see what partial does, consider the function add:

Partial application is a rarely used, advanced Python feature, but we can illustrate its utility in a simple program. Consider a program meant to compare a dice throw simulation to the behavior of a pair of real, physical dice. Suppose a person performs an experiment with a pair of dice, rolling them hundreds of times and recording the result each time. The number recorded is the sum of the numbers on both dice. Since each die face contains one, two, three, four, ?ve, or six spots, the outcome of a roll of two dice can be any number in the range 2?12, inclusive. The person conducting the experiment records the results in a text ?le named dicedata.data.

```
break
# Convert text integer into an actual integer
if int(value) == val:
count += 1
return count
```

The read_file?s parameters consist of the name of the ?le to process (filename), the number of outcomes to consider (n), and the outcome value to count (val). By passing in the ?le name, read_file can ?exibly handle any data ?les of the proper format. The expectation is that the ?le will be very large, and the parameter n limits how much of the data the function processes.

```
Returns the number of times a roll resulted in val. """
count = 0
for i in range(n):
roll = randint(1, 6) + randint(1, 6)
if roll == val:
```

Given the simplicity of Listing 8.31 (comparerolls.py), you may be thinking of ways to rewrite the code to avoid using partial application. Restructuring this code is indeed an option, since we have total control over it. Partial application really shines when we need to interface with library functions over which we have no control. Partial application is a tool that sometimes is handy for quickly adapting an existing function to the requirements of a library developed by others.

```
def sum1(n):
s = 0
while n > 0:
s += 1
n -= 1
return s
```

```
global val
s = 0
while val > 0:
s += 1
val -= 1
return s
```

```
def sum3():
s = 0
for i in range(val, 0, -1):
s += 1
return s
```

```
global val
val = 5
print(sum1(5))
print(sum2())
print(sum3())
```

```
global val
val = 5
print(sum1(5))
print(sum3())
print(sum2())
```

```
global val
val = 5
print(sum2())
print(sum1(5))
print(sum3())
```

```
def main():
print(sum())
print(sum(4))
print(sum(4, 5))
print(sum(5, 4))
print(sum(1, 2, 3))
print(sum(2.6, 1.0, 3))
```

Chapter 9

Objects

In the hardware arena, a personal computer is built by assembling a motherboard (a circuit board containing sockets for a microprocessor and assorted support chips), a processor, memory, a video card, a disk controller, a disk drive, a case, a keyboard, a mouse, and a monitor. The video card by itself is a sophisticated piece of hardware containing a video processor chip, memory, and other electronic components. A technician does not need to assemble the card; the card is used as is off the shelf. The video card provides a substantial amount of functionality in a standard package. One video card can be replaced with another card from a different vendor or with another card with different capabilities. The overall computer will work with either card (subject to availability of drivers for the operating system) because standard interfaces allow the components to work together.

An object is an instance of a class. We have been using objects since the beginning, but we have not taken advantage of all the capabilities that objects provide. Integers, ?oating-point numbers, strings, and functions are all objects in Python. With the exception of function objects, we have treated these objects as passive data. We can assign an integer value to a variable and then use that variable?s value. We can add two ?oating-point numbers and concatenate two strings with the + operator. We can pass objects to functions and functions can return objects.

instance variable comes from the fact that the data is represented by a variable owned by an object, and an object is an instance of a class. Other names for instance variables include attributes and ?elds. Methods are like functions, and they are known also as operations. The instance variables and methods of an object constitutes the object?s members. The code that uses an object is called the object?s client. We say that an object provides a service to its clients. The services provided by an object can be more elaborate that those provided by simple functions because objects make it easy to store persistent data in their instance variables.

We have been using string objects?instamces of class str?for some time. Objects bundle data and functions together, and the data that comprise a string consist of the sequence of characters that make up the string. We now turn our attention to string methods. Listing 9.1 (stringupper.py) shows how a programmer can use the upper method available to string objects.

str Methods
upper

The square brackets when used with an object in the manner shown above invoke that object?s __getitem__ method. In the case of string objects the integer within the square brackets, known as an index, represents the distance from the beginning of the string from which to obtain a character. For string s, s[0] is the ?rst character in the string, s[1] is the second character, as so forth.

Fortunately, Python?s standard library has a ?le class that makes it easy for programmers to make objects that can store data to, and retrieve data from, disk. The formal name of the class of ?le objects we will be using is TextIOWrapper, and it is found in the io module. Since ?le processing is such a common activity, the functions and classes de?ned in the io module are available to any program, and no import statement is required.

The open method opens a ?le for reading or writing, and the read, write, and other such methods enable the program to interact with the ?le. When the executing program is ?nished with its ?le processing it must call the close method to close the ?le properly. Failure to close a ?le can have serious consequences when writing to a ?le, as data meant to be saved could be lost. Every call to the open function should have a corresponding call to the ?le object?s close method.

object

block

The with/as statement can work with classes like TextIOWrapper that provide a particular protocol for initialization and ?nalization. In the case of our ?le object, if the call to open proceeds without an error, the program will execute the code in the with/as block. When the code in the with/as has ?nished executing, the statement executes any ?nalization actions the class requires. In this case, the ?nalization code of the TextIOWrapper class closes the ?le associated with ?le object f. Only certain classes support the initialization/?nalization protocol in a way that is compatible with the with/as statement. Such classes provide a method named __enter__ that performs the initialization and a method named __exit__ that performs the ?nalization.

The literal name of Python?s ?le class is TextIOWrapper from the io module. The kind of ?les processed by this ?le class are known as text ?les. Text ?les store character data, and we can use a simple editor to create and modify text ?les. Many applications prevent the easy modi?cation of data ?les outside of the application by encoding the data in a special way. Depending on the data, the application also may encode the ?les to save space.

TextIOWrapper Methods

open

We say that name, _CHUNK_SIZE, encoding, and line_buffering are all instance variables of the object f. These are just like the variables we have been using, except that we must pre?x their name with their associated object and a dot (.). Since these names refer to data, not methods, no parentheses appear at the end. If we have two different ?le objects, f and g, f.name may be different from g.name. The statement

creates a Fraction object and assigns the variable f1 to the object. The expression Fraction(3, 4) calls a class constructor. Class constructors allow clients to supply data used in the formation of a new object. In this case, the ?rst parameter represents the numerator of the new fraction object, and the second parameter represents the denominator of the object. The Fraction(3, 4) expression returns a reference to the newly created fraction object, and the statement

We say the former statement using the + operator is syntactic sugar for latter statement that uses the explicit __add__ method. Most human readers prefer the version with +, but, in reality, both statements are identical to the Python interpreter. The str class of string objects also provides an __add__ method. If s and t are string objects, the string concatenation expression s + t is syntactic sugar for s.__add__(t).

Calling __add__ instead of using the + operator offers no performance advantages and makes the code less readable for humans. You therefore should use the more readable operator syntax in your code. We mention the alternate syntax at this point more as interesting curiosity, but later in Chapter 13 we can take advantage of these special methods when we are designing our own custom types.

In Section 6.9, we introduced Python?s Turtle graphics functional interface. We saw how it is possible to draw pictures within a graphical window via function calls. Behind the scenes the turtle module creates a global Turtle object which models the pen doing the drawing. The Turtle graphics functions such as left and pencolor manipulate this hidden Turtle object. We can create and use our own Turtle objects. This is useful if we wish to manage multiple pens simultaneously.

The tkinter module provides classes for building graphical user interfaces via the cross-platform Tk toolkit. Tk is available for the Microsoft Windows, Apple Mac, and Linux operating systems. The tkinter module is much larger and more complex than the turtle module. In fact, the turtle module is built from components the tkinter module provides.

? Button: This class represents a graphical button that a user can press. A Button is one many graphical user interface widgets the Tk toolkit provides. (Other Tk widgets include checkboxes, text labels, radio buttons, text entry ?elds, and list boxes, scroll bars, progress bars, and spin boxes.) The statement function can accept up to four optional keyword arguments. We commonly use the end and sep keyword arguments with print. The code here calls the configure method of the Button class with background, text, and command keyword arguments. These names are only three of 35 different keyword argument names that the configure method can accept. A few other characteristics of a Button object that the configure method enables programmers to adjust include text font and color, horizontal and vertical padding, text justi?cation, and border thickness.

Note that in the update function of Listing 9.13 (buttontester.py) we must declare that count is a global variable because the function reassigns it. Without the global declaration update would treat the name color as a local variable. Since the code within update does not assign the b variable, the name b is implicitly global within the function.

creates a Canvas object named canvas placed in the frame widget container. The canvas objects dimensions are 150 pixels wide and 300 pixels tall, as speci?ed by the width and height keyword arguments. The origin of the coordinate system of the canvas, the point (0,0), is found at the left-top corner of the drawing area within the window, and the y axis is inverted, meaning it points down instead of up. This means that while x values increase from left to right as expected, y increase from top to bottom within the canvas. Figure 9.3 illustrates.

invokes the create_rectangle method to add a gray rectangle of the given size to the canvas. The ?rst two parameters set the left-top corner of the rectangle to the location (50,20). The third and fourth parameters locate the rectangle?s right-bottom corner to be exactly one pixel to the left and one pixel above the point (150,280). The fill keyword argument speci?es the rectangle?s background color. This rectangle represents the traf?c light?s frame that holds the lamps. The create_oval method works similarly for creating the circles representing the lamps of the traf?c light. Figure 9.4 shows how the arguments passed to create_oval bounds the oval shape within a rectangular area.

Figure 9.4 The create oval method of the Tk Canvas class adds an oval shape to a canvas object. The statement in the ?gure creates a circle shape on the canvas object referenced by c. The ?rst two parameters specify the left-top corner of a bounding rectangle, and the third and fourth parameters specify the bounding rectangle?s right-bottom corner.

(10, 30)

(60, 80)

? Button: This class represents a graphical button the user can press. Unlike in Listing 9.13 (buttontester.py), this class comes from the tkinter.ttk package. The visual appearance and functionality differs slightly from Button objects from the tkinter package. Buttons from the tkinter.ttk package have a more modern look and feel.

impose a one-row, two-column grid widget layout upon the root graphical window. The Tk toolkit determines this con?guration from the code, and we can too. Note that we add only the button and canvas widgets to the root window. The two calls to the grid method specify only one row (both are 0) but two columns (one is 0 and the other is 1). This means the button and canvas objects will appear side by side (same row 0), and the button will be to the left of the canvas that draws the traf?c light image (0 is the ?rst column, and 1 is the second column).

Note that in the do_button_press function of Listing 9.14 (tkinterlight.py) we must declare that color is a global variable because the function reassigns it. Without the global declaration do_button_press would treat color as a local variable. Since the code within do_button_press does not assign the canvas variable, canvas is implicitly global within the function.

number 1

2, and frac1 is merely a name by which we can access the Fraction object. This informality has not been a problem so far, but as we explore objects more deeply we need to be more careful in how we talk about and use objects.

f1
1
2
f3

1
2
f2

show that the variables f1 and f2 reference two different objects, but f1 and f3 refer to the same object. This proves that f1 and f3 are aliases. Python also has an id function that returns an integer that is unique to a particular object. (For most Python implementations this number is the starting address in memory where the executing program has placed the object.) If a and b are objects, a is b is true exactly when id(a) == id(b).

Prior to this chapter we have restricted our attention to the classes int, float, str, and bool. Object aliasing has no practical consequences for programmers restricted to these data types. Instances of these classes are all immutable objects, which means an object of any these of these types cannot change its state after its creation. The integer 3 always is 3, for example, and the string object 'Fred' cannot change to 'Free'. Instances of the Fraction class are immutable also.

Aliasing can be an issue for mutable objects. In Python?s Turtle graphics library (Section 9.5), Turtle objects are mutable. Programmers can move a turtle object, change its orientation, and change its pen color. Each of these actions changes the state of the turtle and affects the way the turtle draws within the graphics window.

makes t2 a copy of t1 and that they remain distinct objects. If the situation arises where your program is managing what you believe to be similar but separate objects and changing the characteristics of one object unexpectedly changes one or more of the other objects in exactly the same way, you likely have an unintended aliasing problem.

creates a Fraction object representing 2
3 and assigns the variable f to the object. The Python interpreter acts
on this statement by reserving enough space in the computer?s memory to hold the object. It also performs
any initialization that the object requires; in this case it sets f.numerator to 2 and f.denominator to 3.

ences the 2
3 fraction object created earlier. This means the object effectively is cut off from the remainder
of the program?s execution or the remainder of the interactive session. This abandoned object is classi?ed
as garbage. The term garbage is a technical term used in computer science that refers to memory allo-
cated by an executing program that the program no longer can access. The Python interpreter cleans up
garbage through a process called garbage collection. Python uses a reference counting garbage collector
that automatically reclaims the space occupied by abandoned objects.

increments the 2
3 object?s reference count by one. If we make another alias, as in

the reference count of the 2
3 object decreases by one, and the reference count of the new 9
10 object be-
comes 1. If we reassign f:

this leaves only variable h referencing the 2
3 object, so the object?s reference count is 1. If we ?nally reassign
h:

the 2
3 object?s reference count drops to zero. An object with a reference count of zero is garbage, and the garbage collector automatically will reclaim the space held by the object so it can be recycled and used elsewhere.

Chapter 10

Lists

We can modify Listing 10.2 (averagenumbers2.py) to average 25 values much more easily than Listing 10.1 (averagenumbers.py) that must use 25 separate variables?just change the value of NUMBER_OF_ENTRIES. In fact, the coding change to average 1,000 numbers is no more dif?cult. However, unlike the original average program, this new version does not at the end display all the numbers entered. This is a signi?cant difference; it may be necessary to retain all the values entered for various reasons:

The number within the square brackets is called the index. A nonnegative index indicates the distance from the beginning of the list. The expression lst[0] therefore indicates the element at the very beginning (a distance of zero from the beginning) of lst, and lst[1] is the second element (a distance of one away from the beginning). We can read aloud the expression a[3] as ?a sub three,? where the index 3 represents a subscript. The subscript terminology is borrowed from mathematicians who use subscripts to reference elements in a mathematical vector or matrix; for example, V2 represents the second element in vector V. Unlike the convention often used in mathematics, however, the ?rst element in a list is at position zero, not one. As mentioned above, the index indicates the distance from the beginning; thus, the very ?rst element is at a distance of zero from the beginning of the list. The ?rst element of list a is a[0]. As a consequence of a zero beginning index, if list a holds n elements, the last element in a is a[n?1], not a[n].

?

Ist

Not only is this version shorter, it actually is more ef?cient than the version that uses range and len. The reversed expression creates an iterable object that enables the for statement to traverse the elements of the list in reverse. The expression reversed(nums) does not affect the contents of the list nums; it simply enables a backwards traversal of the elements. The reversed function returns a generator that works like the following:

Within the range expression, the ?rst argument, len(lst) - 1, is the index of the last element in the list lst. The second argument, -1, indicates ?1 terminates, but is not included in, the range. The last argument, -1 indicates the range counts backwards. Taken all together we see that the range spans the indices of all the elements in the list, from the last to the ?rst.

Unlike the original program, however, we now conveniently can extend this program to handle as many values as we wish. We need only change the de?nition of the NUMBER_OF_ENTRIES variable to allow the program to handle any number of values. This centralization of the de?nition of the list?s size eliminates duplicating a literal numeric value and leads to a program that is more maintainable. Suppose every occurrence of NUMBER_OF_ENTRIES were replaced with the literal value 5. The program would work exactly the same way, but changing the size would require touching many places within the program. When duplicate information is scattered throughout a program, it is a common mistake to update some but not all of the information when a change is to be made. If all of the duplicate information is not updated to agree, the inconsistencies result in logic errors within the program. By faithfully using the NUMBER_OF_ENTRIES variable throughout the program instead of the literal numeric value, we can avoid the problems of these potential inconsistencies.

does not make a copy of a?s list. Instead it makes a and b aliases to the same list. Lists are mutable data structures. We may reassign individual list elements via []. If more than one variable is bound to the same list, any element modi?cation through one of the variables will affect the list from the point of view of all the aliased variables.

list [begin : end : step]

Suppose variable a refers to a nonempty list. Note the difference between the expression a[0:1] and the expression a[0]. The expression a[0:1] represents a new list that contains only the ?rst element of the original list a. The expression a[0] refers to the ?rst element in a, which is not necessarily a list. The following interactive sequence illustrates the difference:

Slicing is the easiest way to make a copy of a list. The expression lst[:] evaluates to a copy of the entire list lst. The list_copy function we saw in Listing 10.16 (listcopy.py) made for an interesting exercise, but list slicing is shorter, simpler way to achieve the same result. The last statement in Listing 10.19 (listslice.py) shows the expression lst[::-1] makes a copy of list lst with all of its elements appearing in reverse order.

In Listing 10.22 (listfunc.py) the functions sum and make_zero accept a parameter of type list. Section 7.2 addressed the consequences of passing immutable types like integers and strings to functions. Since list objects are mutable, passing to a function a reference to a list object binds the formal parameter to the list object. This means the formal parameter becomes an alias of the actual parameter. The sum method does not attempt to modify its parameter, but the make_zero method changes every element in the list to zero. This means the make_zero function will modify the a list object in main.

list Methods

count

Listing 10.24 (fasterprimes.py) uses an algorithm developed by the Greek mathematician Eratosthenes who lived from 274 B.C. to 195 B.C. Called the Sieve of Eratosthenes, the principle behind the algorithm is simple: Make a list of all the integers two and larger. Two is a prime number, but any multiple of two cannot be a prime number (since a multiple of two has two as a factor). Go through the rest of the list and mark out all multiples of two (4, 6, 8, ...). Move to the next number in the list (in this case, three). If it is not marked out, it must be prime, so go through the rest of the list and mark out all multiples of that number (6, 9, 12, ...). Continue this process until you have listed all the primes you want.

The sys module provides a global variable named argv that is a list of extra text that the user can supply when launching an application from the operating system shell (normally called the command prompt in Windows and terminal in OS X and Linux). To run a program stored in the ?le myprog.py, the user would type the command

2. set builder notation: P = {x2 | x ?{0,1,2,3,4,5,6,7}}

The set roster notation example is obvious?it just lists the elements of the set. We read the set builder notation example as ?P is the set of all squares of x, such that x is taken from the set {0,1,2,3,4,5,6,7}. Set builder notation in mathematics is essential for representing very large sets and in?nite sets; for example, consider S = {x2 | x ?Z}, the set of all perfect squares (Z is the in?nite set of integers). Listing all the elements is impossible. We could try to list the ?rst few elements followed by an ellipsis (...), but the pattern may not be obvious to all readers.

One limitation of range is that its arguments all must be integers. Suppose we wish to create succinctly a list of ?oating-point numbers in a regular sequence. We cannot use the range expression by itself to express such a list, but a list comprehension is ideal for the task. The following interactive sequence creates a list containing the ?rst ten multiples of one-half:

Observe that the program printed all the factors of 100. We want our list to contain factor pairs; that is, we want to pair each factor with its mate so that the two values multiplied together equal the number the user entered. We already have the ?rst elements of the pairs, and we can compute their mates easily with integer division. If x is a factor of n, then n//x will be its mate because x * n//x will equal n. The factor pair is thus (x, n//x). Listing 10.29 (factorpairs.py) shows the resulting program which produces a list of factor pairs.

This is ?ne, but it would be nice to avoid adding redundant pairs pairs to the list; that is, we want just one of the pairs (2, 50) and (50, 2) to appear in our list. Notice that once the ?rst element reaches the square root of the value supplied by the user, the remaining pairs are mirror images of earlier pairs. We can use this fact to limit the range expression; we will choose xs in the range 1...?n, inclusive. The Python implementation of this range is range(1, round(math.sqrt(n)) + 1). Listing 10.30 (uniquefactorpairs.py) adds this ?nishing touch to avoid the redundant pairs.

Python list comprehensions are powerful and can be quite complex. When programming it sometimes is easier to build the list without list comprehension and later discover a way to transform the code to use list comprehension. As an example, consider the task of building a list that contains all the prime numbers less than 100. For all n ?2, the following list comprehension creates a list of all the factors of n, not including 1 and n itself:

List comprehensions are a valuable, powerful tool for creating lists. While the prime numbers example demonstrates the power of Python?s list comprehensions, many programmers would consider the resulting list comprehension expression above to be a bit too complex and tricky. Additionally, it is not very ef?cient. It creates an internal, temporary list of factors for each number it must consider. Simpler list comprehension expressions that avoid nested lists generally are ef?cient and readable. As a rule, you should use list comprehensions when they make your code simpler and more readable, but do not go out of your way to construct arcane list comprehension expressions just because you can.

If you need to keep around the values of a sequence for additional processing, store them in a list and possibly use a list comprehension to make the list. If you simply need to visit the elements in a sequence once, building a list is overkill; use a generator to produce the sequence?s elements as needed. The list has to store all of its elements in memory for the life of the list, but a generator produces only one element at a time. This means the list [x for x in range(n)] will consume a large amount of the computer?s memory if n is large. The generator (x for x in range(n)) uses a relatively small, constant amount of memory regardless of n?s value.

Such a two-dimensional (2D) matrix has a particular number of rows and columns. The values in rows are arranged horizontally, and the elements in columns are arranged vertically. The above matrix, therefore, has four rows and ?ve columns. We say its dimensions are 4×5. The ?rst row of the example matrix above consists of the elements 100, 14, 8, 22, and 71; the ?rst column contains 100, 0, 90, and 115. We can locate an element uniquely within the matrix with two integer indices?the ?rst index represents the element?s offset from the ?rst row, and the second index represents the element?s offset from the ?rst column. As with 1D lists, the index of the ?rst row is zero, and the index of the ?rst column is zero. The element 67 above has a row index of 2 and a column index of 3.

How are 2D matrices used? Mathematicians can represent a system of equations in a matrix and use techniques from linear algebra to solve the system. Computer scientists use 2D matrices to model the articulation of robotic arms. Computer graphics programmers mathematically manipulate 2D matrices to transform data in 3D space and project it onto a 2D screen giving users the illusion of depth, motion, location. Many classic games such as chess, checkers, Go, and Scrabble involve a 2D grid of board positions that naturally maps to a 2D matrix. A word search puzzle is a rectangular array of letters, and a maze is simply a 2D arrangement of adjoining rooms, some of which are connected and others that are not.

Note the double square brackets. The ?rst index (2) selects the row, and the second index (3) speci?es the column. The expression matrix represents the whole 2D list. If we use just one index with a 2D list, it selects an entire row. The expression matrix[x] represents the 1D list that constitutes row x in matrix; for example, matrix[2] is the list [90, 21, 7, 67, 112]. That means we can interpret the expression matrix[2][3] as (matrix[2])[3]; that is, the element at index 3 within the row at index 2. The expression len(matrix) is the number of rows in matrix. The expression len(matrix[2]) represents the number of elements in the row at index 2. For many applications, every row in the 2D list will have the same length; we call such a 2D list a table. Python does not require that all rows have the same number of elements. The following code creates what is called a ragged array, where each row potentially has a different length:

Conceptual View

Internal Representation

builds a 1D list named a that contains four zeros. This list element multiplier technique is straightforward when the elements are immutable types (see Section 9.8). All the elements refer to the same object, but since the object cannot be mutated, any reassignment of a list element will not effect any of the other elements in the list. The following code produces no surprises:

Figure 10.6 illustrates the attempt at 2D list creation and the subsequent element reassignment. As in the 1D case from before, the list multiplication creates a list of elements that all refer to the same object. In this case, however, instead of the integer zero, the object to which all the elements refer is a 1D list containing zeros. Since all the references in the list of lists a refer to the same list, all the rows in our 2D list alias the same list of zeros. Attempting the change the element at index 2 in row 1 changes the element at index 2 in all the rows.

The for expression within the list comprehension requires a variable to control range?s iteration. Here we gave it the name x. This variable x is local to the list comprehension. Section 2.3 emphasized that all variables should have meaningful, descriptive names. Assignment attaches a name to an object so we can access that object in the future via its name. The problem with x is this: we do not really do any thing with x. It is dif?cult to provide a good name for a variable that we never actually use. Some programmers address

```
a = [[0] * 4] * 3
```

this situation by using a special variable name consisting of the single underscore (_). There is nothing magical about the single underscore symbol?it is a legal identi?er, and, therefore, is a valid variable name. Its primary value lies in the fact that its name is ultimately non-descriptive! The name x may have a special meaning in some contexts, like the ?rst element in an (x,y) ordered pair, but it is dif?cult to attach meaning to the name _.

```
>>> 10 + 4
14
>>> _
14
>>> 100 - 60
40
>>> 2 + _
42
>>>
```

a

```
a = [[0 for _ in range(4)] for _ in range(3)]
```

```
a[1][2] = 5
```

Consider the children?s game of Tic-Tac-Toe, sometimes called Noughts and Crosses (see https://en.wikipedia.org/wiki/Tic-tac-toe for more information about the game). It consists of a 3 × 3 playing grid in which two opposing players alternately place Xs and Os. A 3D variation uses a 4 × 4 × 4 cube for placing Xs and Os (see https://en.wikipedia.org/wiki/3-D Tic-Tac-Toe for more information).
If we were to implement 3D Tic-Tac-Toe in Python, we could use a 4 × 4 × 4 list. As a visual proof of concept, Listing 10.33 (threedlist.py) prints an intermediate state of a 3D Tic-Tac-Toe game in progress.

If you need to create a sequence of values and random access is unnecessary, a generator may be the better choice. The generator?s sequence behaves like a lazy list; that is, the sequence element exists only at the time it is needed. If, on the other hand, your program needs to have all the values of a sequence available at any time during the program?s execution, a list is the necessary choice. Unlike generators, a list usually must be fully populated with all of its elements before it truly is useful to the program.

18. Write a function named next_number that accepts a list of integer values. All the elements in the list are unique, and all elements in the list are greater than or equal to one. (The caller must ensure that these conditions are met before passing the list to next_number.) The next_number function should return the smallest positive integer not in the list. (Note that 1 is the smallest positive integer.)

23. We can represent a Tic-Tac-Toe board as a 3 × 3 grid in which each position can hold one of the following three strings: "X", "O", or " ". Write a function named check_winner that accepts a 3 × 3 list as a parameter. If "X" appears in a winning Tic-Tac-Toe pattern, the function should return the string "X". If "O" appears in a winning Tic-Tac-Toe pattern, the function should return the string "O". If no winning pattern exists, the function should return the string " ".

Chapter 11

Tuples, Dictionaries, and Sets

Neither the list nor tuple function actually modi?es its argument; that is, tuple(lst) does not modify lst, and list(tpl) does not modify tuple (since tuples are immutable, any modi?cation would be impossible anyway). The list function makes a new list out of the contents of a tuple, and the tuple function makes a new tuple out of the elements in a list.

You can think of the zip function working like a physical zipper. A physical zipper pairs up two sets of interlocking (usually metal) teeth, closing an opening in a garment or bag. The Python zip function pairs up elements from two different sequences. The paired-up elements are tuples, and the sequences can be lists or sequences constructed from generators (see Section 8.8). If one of the sequences is shorter than the other, the zip function stops at the shorter sequence and ignores the remainder of the longer sequence.

We can use the zip function and list comprehension to build elaborate lists. Suppose we wish to make a new list from two existing lists. The ?rst element in our new list will be the sum of the ?rst elements from the two original lists. Similarly, the second element in our new list will be the sum of the second elements in the two original lists, and so forth. We can use zip to pair up the elements, as the following interactive sequence illustrates:

Since they are so similar, why does Python have both lists and tuples? Under some circumstances an executing program can perform optimizations on immutable objects that would be impossible with mutable objects. These optimizations can increase the program?s performance. Also, it is easier in general to reason about the behavior of programs that use immutable objects. The fact that some objects cannot change makes it easier to understand how a section of code works, or, during debugging, why the section of code does not work.

When we de?ne a function we specify the individual parameters it accepts, providing default values as needed. In the function de?nitions we have seen to this point the number of parameters is ?xed. We need a way to de?ne a function in such a way so that it can accept an arbitrary number of parameters. Fortunately Python has a mechanism for specifying that a function can accept an arbitrary number of parameters. Listing 11.4 (addmany.py) illustrates how write such a function.

```
def sum(num1, num2, *extranums):
s = num1 + num2
for n in extranums:
s += n
return s
```

Notice that, unlike a list which uses square brackets ([]), the contents of a dictionary appear within curly braces ({}). To access an element within a dictionary, however, we use square brackets exactly as we would with a list. In a dictionary every key has an associated value. The dictionary d from the interactive sequence above pairs the key 'Fred' with the value 44. It also pairs the key 'Ella' with the value 31.

The string 'Fred' is the key, and 44 is its associated value. If the key within the square brackets does not exist in the dictionary, the statement adds the key and pairs it with the value on the right of the assignment operator. If the key already exists in the dictionary, the statement replaces the value previously associated with the key with the new value on the right of the assignment operator.

'Fred' must be a valid key in dictionary d, or the program will raise an exception. A valid key is a key that is present in the dictionary. At the end of the interaction sequence above 'Fred' is a valid key but 'Zach' is not. We see the interpreter?s reaction when we attempt to use an invalid key: the interpreter generates a KeyError exception.

Observe that the print function neither lists the keys in lexicographical order nor lists the values in numerical order. While an executing program must store a dictionary?s contents in memory in some particular order, the exact internal ordering of the elements within a dictionary can vary from one program execution to the next. This example further demonstrates that programmers cannot depend on a speci?c ordering of the elements within a dictionary. Unlike in a list or other sequence type, the notions of order and position have no meaning within a dictionary.

Suppose we have the list ['Fred', 'Ella', 'Owen', 'Zoe'] and the list [4174, 2287, 5003, 2012]. We know we can zip them together into a sequence of tuples using zip (Section 11.1). We can use the dict function to create a dictionary of key:value pairs formed from the tuples, as the following interactive sequence shows:

The elements of the list speci?ed as the ?rst actual parameter to zip become the dictionary keys, and the elements of the list speci?ed as the second argument to zip form the values in the dictionary. As we noted earlier, the ordering of the key:value pairs is different from their order in the original lists, but the ?rst element from the names list is paired with the ?rst element of the numbers list, the second element from names is paired with the second element of numbers, and so forth.

A dictionary is sometimes called an associative array because its elements (values) are associated with keys instead of indices. The placement and lookup of an element within a dictionary uses a process known as hashing. A hash function maps a key to a location within the dictionary where the key?s associated value resides. Python dictionaries are related to hash tables in computer science. See http://en.wikipedia.org/wiki/Hash table for more information about hash functions and hash tables. The important thing to know about the hashing process is that it makes value lookup via a key very fast.

It would be inappropriate to place the names in a list and locate a name using the associated phone number as an index into the list. This look-up method is backwards?we do not want to ?nd a name given a phone number; we want to look up a number based on a name. Besides, each phone number contains many digits, and we would not need or want to have a list with indices with values that large?most of the space in the list would be unused.

In our situation a person or company?s name is a unique identi?er for that contact. In this case the name is a key to that contact. A Python dictionary is the ideal data structure for mapping keys to values. A dictionary allows for the fast retrieval of a value given its associated key. Listing 11.7 (phonelist.py) uses a Python dictionary to implement a simple telephone contact database with a rudimentary command line interface.

Dictionaries are useful for counting things. We have experience using variables to count; recall Listing 5.3 (countup.py), Listing 5.15 (countvowels.py), Listing 5.34 (startree.py), Listing 6.7 (measureprimespeed.py), Listing 7.19 (treefunc.py), or Listing 10.25 (timeprimes.py). These programs all have counted one thing at a time, so they each use just one counter variable. In general, we need to use a separate variable for each count we manage. The following code counts the number of negative and nonnegative numbers in a list of numbers and returns a tuple with the results:

The answer is this: We cannot know how many counter variables we will need, so we must use a different approach. If all the things we need to count are immutable objects, like numbers or strings, we can use the objects as keys in a dictionary and associate with each key a count. As a concrete example, Listing 11.10 (wordcount.py) reads the content of a text ?le containing words. After reading the ?le the program prints a count of each word. To simplify things, the text ?le contains only words with no punctuation. The user supplies the ?le name on the command line when launching the program (see Section 10.12).

exercises a method of the str class that separates the very long string composed of all the words in the ?le into separate strings. The split method divides the string based on whitespace (spaces, tabs, and newlines) and returns the individual words in a list. The following interactive sequence shows how the split method works:

This ?le contains ?ve lines of text. Each line may have trailing spaces that are not visible in the listing above, and each line certainly has a newline ("\n") at its end. We must strip the newline from each line and strip the ?nal comma and trailing spaces. The string rstrip will accomplish this end-of-line clean up with the match string " ,\n" (space, comma, newline). Listing 11.11 (readtext?le.py) provides the complete code.

program could use special processing for this set of long words if it ever becomes nonempty. This approach under typical circumstances would result in a number of empty sets at higher indices because most English text contains words of at most about 20 letters. The advantage of a dictionary is that it stores only what it needs. Listing 11.13 (groupwordslist.py) is close transliteration of Listing 11.12 (groupwords.py) that uses a list in place of a dictionary.

The calling code assigns the value of the ?rst actual parameter to the ?rst formal parameter. It assigns the value of the second parameter to the second formal parameter. Finally, it assigns the value of the third actual parameter to the third formal parameter. By default, the association of actual parameter to formal parameter during a function invocation is strictly positional. This is the shortest, simplest way for the caller to pass parameters.

shows that keywords arguments may appear in the same call as non-keyword arguments, but in such mixed-parameter calls all non-keyword arguments must appear before any keyword arguments. The function invocation mechanism assigns the non-keyword arguments as usual: the ?rst actual parameter to the ?rst formal parameter, second actual parameter to the second formal parameter, etc. It assigns the keyword arguments that follow to the formal parameters of the same name.

```
def f(**args):
a = args['a']
b = args['b']
c = args['c']
return 2*a*a + 3*b + c
```

In this function x, y, and z are regular positional arguments, a is the arbitrary arguments tuple, and b is the keywords arguments dictionary. The positional arguments, if any, must appear before any arbitrary arguments and keyword arguments. The arbitrary arguments, if any, must appear after the positional arguments and before the keyword arguments. The keyword arguments, if any, must appear after the positional and arbitrary argument list parameters.

Keyword arguments are very convenient for functions and methods that can accept a large number of arguments. The configure method in the Button class is de?ned to accept a dictionary via the ** notation. It accepts up to 35 keyword arguments. Keyword arguments in general enable a caller to supply just a few arguments, in any order. There is no need for the programmer to remember which argument comes ?rst, then second, etc. as with positional arguments. The function or method can accept the caller-supplied arguments and use predetermined default values for any optional arguments the caller omitted.

Python provides a data structure that represents a mathematical set. As with mathematical sets, we use curly braces ({}) in Python code to enclose the elements of a literal set. Python distinguishes between set literals and dictionary literals by the fact that all the items in a dictionary are colon-connected (:) key-value pairs, while the elements in a set are simply values. Unlike Python lists, sets are unordered and may contain no duplicate elements. The following interactive sequence demonstrates these set properties:

Python set notation exhibits one important difference with mathematics: the expression {} does not represent the empty set. In order to use the curly braces for a set, the set must contain at least one element. The expression set() produces a set with no elements, and thus represents the empty set. Python reserves the {} notation for empty dictionaries (see Section 11.3).

In most Python programming, sets play a much smaller role than lists and dictionaries. Sets are most similar to lists, and the ordering of data is important in many applications. If order does not matter and all elements are unique, the set type does offer a big advantage over the list type: testing for membership using in is much faster on sets than lists. Listing 11.16 (setvslistaccess.py) creates both a set and a list, each containing the ?rst 1,000 perfect squares. It then searches both data structuures for, and does nothing with, all the integers from 0 to 999,999. It reports the time required for the efforts.

This code requires two function calls in order to manage the indices: one call to len to determine the highest index and another call to the range constructor to produce each index. The __builtins__ module provides a function named enumerate that returns an iterable object that produces tuples. Each tuple pairs an index with its associated element. The following code uses the enumerate function to produce the same results as the above code:

9. Write a function named zero_sum that accepts any number of integer arguments. The function should return True if the sum of its arguments is zero; otherwise, it should return False. The call zero_sum(2, 3, -5), for example, would evaluate to True, since 2 + 3 + ?5 = 0. On the other hand, zero_sum(2, 3, -10, 4) evaluates to False because 2 + 3 + ?10 + 4 = ?1 ?= 0. zero_sum should return True when called with no arguments.

19. Write a graphical, two-player Tic-Tac-Toe game using the tkinter module (see https://en.wikipedia.org/wiki/Tic-tac-toe for more information about the game). You can use nine separate variables to track the contents of the game?s squares. You must be able to draw lines and circles in the appropriate locations.

Chapter 12

Handling Exceptions

In our programming experience so far we have encountered several kinds of run-time exceptions, such as division by zero, accessing a list with an out-of-range index, and attempting to convert a non-number to an integer. We have seen these and other run-time exceptions immediately terminate a running program. Python provides a standard mechanism called exception handling that allows programmers to deal with these kinds of run-time exceptions and many more. Rather than always terminating the program?s execution, an executing program can detect the problem when it arises and possibly execute code to correct the issue or mitigate it in some way. This chapter explores handling exceptions in Python.

Algorithm design can be tricky because the details are crucial. It may be straightforward to write an algorithm to solve a problem in the general case, but the designer may have to address a number of special cases within the problem for the algorithm to be correct. Some of these special cases might occur rarely and only under the most extraordinary circumstances. The algorithm must properly handle these exceptional cases to be truly robust; however, adding the necessary details to the algorithm may render it overly complex and dif?cult to construct correctly. Such an overly complex algorithm would be dif?cult for others to read and understand, and it would be harder to debug and extend.

However, if the code within a function accesses the list in many different places, the large number of conditionals required to ensure the absolute safety of all the list accesses can quickly obscure the overall logic of the function. Fortunately, programmers sometimes can avoid this scenario by checking a list index once for a large number of similar accesses within a block of code or managing the index carefully to ensure it cannot be outside the list?s bounds. Other problems, however, such as the loss of a network connection, may be less straightforward for the algorithm to address directly. Fortunately, speci?c Python exceptions are available to cover problems such as these.

Exceptions represent a standard way to deal with run-time errors. In programming languages that do not support exception handling, programmers must devise their own ways of dealing with exceptional situations. Such ad hoc approaches produce error handling facilities developed by one programmer that can be incompatible with those used by another. Python provides a comprehensive, uniform exception handling framework. Python?s exception framework provides a simple means of communicating errors between functions and short circuiting the normal function return process, effectively bypassing functions up the call chain that are unable to, or have no need to, participate in the error handling activity. The proper use of Python?s exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors. The standard Python library uses exceptions, and programmers can create new exceptions that address issues speci?c to their particular problems. These exceptions all use a common syntax and are completely compatible with each other.

For an English-speaking human, the response ?ve should be just as acceptable as 5. The strings acceptable to the Python int function, however, can contain only numeric characters and an optional leading sign character (+ or -). The user?s input causes the program to produce a run-time exception. As it stands, the program reacts to the exception by printing a message and terminating itself. As shown in the exception error report, the kind of exception that this execution example produces is a ValueError exception.

Unfortunately, any attempt to make Listing 12.3 (enterinteger.py) more robust via the LBYL idiom is not as easy as it is for Listing 12.2 (checkforzero.py). We basically need to determine if the arbitrary string the user enters is acceptable to the int conversion function. The string must contain only the digit characters '0', '1', '2', '3', '4', '5', '6', '7', '8', or '9', and it may contain a leading '-' or '+' character indicating the number?s sign. Python?s regular expression library is ideal for this purpose, but it is somewhat complicatied and deserves an entire chapter devoted to its use. Short of using the regular expression library, the logic to ensure that the string is acceptable to the int function would be relatively complex.

An alternative to LBYL is EAFP, which stands for easier to ask for forgiveness than permission. The EAFP approach attempts to execute the potentially problematic code within a try statement. If the code raises an exception, the program?s execution does not necessarily terminate; instead, the program?s execution jumps imeidately to a different block within the try statement. Listing 12.4 (enterintexcept.py) wraps the code from Listing 12.3 (enterinteger.py) within a try statement to successfully defend again bad user input.

The two statements between try and except constitute the try block. The statement after the except line represents an except block. If the user enters a string unacceptable to the int function, the int function will raise a ValueError exception. At this point the program will not complete the assignment statement nor will it execute the print statement that follows. Instead the program immediately will begin executing the code in the except block. This means if the user enters ?ve, the program will print the message Input not accepted. If the user enters a convertible string like 5, the program will complete the try block and ignore the code in the except block. Figure 12.1 contrasts the possible program execution ?ows within a try/except statement. We say the except block handles the exception raised in the try block. Another common terminology used to describe the excepting handling process uses the throw/catch metaphor: the executing program throws an exception that an except block catches.

Each time through the loop the code within the try block of Listing 12.6 (multiexcept.py) will raise one of three different exceptions based on the generated pseudorandom number. The program offers three except blocks. If the code in the try block raises one of the three types of exceptions, the program will execute the code in the matching except block. Only code in one of the three except blocks will execute as a result of the exception. The following shows a sample run of Listing 12.6 (multiexcept.py):

In addition to raising one of the three exceptions from Listing 12.6 (multiexcept.py), Listing 12.8 (missedexception.py) can raise a KeyError exception. The expression {} represents the empty dictionary, and the expression {}[1] represents the value associated with the key 1 within the empty dictionary (which, of course, does not exist). Unfortunately, Listing 12.8 (missedexception.py) has no handler for the KeyError exception. The following output shows the results for one program run:

If we want our programs not to crash, we need to handle all possible exceptions that can arise. This is particularly important when we use libraries that we did not write. A program may execute code that only under very rare circumstances raises an exception. This situation may be so rare that it evades our thorough testing and appears only after we deploy the application to users. We need a handler that can catch any exception.

Listing 12.9 (catchallexcept.py) offers four except blocks. The except Exception block at the end represents the catch-all handler that can catch any exception not caught by an earlier except block within the try statement. If present, the catch-all except block should be the last except block in the try statement. Since the Exception type matches any exception type, if it appears before another except block, it will intercept a speci?c exception before a later except block has a chance to see it. This is because a program executes at most one except block when executing a try statement.

Now that we have seen how to write a catch-all exception handler, it is important to note that its use should be limited. A catch-all handler, if used properly, is appropriate for some situations, but for beginning programmers it is tempting to provide a catch-all except block all the time, just in case. A good rule of thumb is this: code should handle speci?c kinds of exceptions it expects and ignore (that is, do not attempt to catch) exceptions it does not anticipate. Section 12.7 provides some direction for the proper use the

Python?s exception handling infrastructure is special because it can transcend the usual scoping rules for functions and objects. The exception handling examples we have seen so far have been simple programs where all the code is in the main executing module. The origin of the exception is close to the code we can see. These examples have not demonstrated the true power of Python?s exceptions. To get a better idea of the scope of exceptions, consider Listing 12.12 (makeintegerlist.py).

Listing 12.12 (makeintegerlist.py) provides two handy functions, get_int_in_range and create_list. The get_int_in_range function expects the user to enter an integer value that falls within the range of values speci?ed by its parameters. It ensures the integer the user provides is in the correct range, but if the user enters a non-integer value, the function will raise an exception. The following shows the program?s interaction with a well-behaved user:

An uncaught exception produces a stack trace. Python uses an area of the computer?s memory known as the stack to help it control function and method invocations. The stack stores parameters, return values, and the point in the code where the program?s execution should return when a function completes. An exception stack trace provides a snapshot of the stack that enables developers to reconstruct the chain of function and/or method calls that produced the exception.

We can read the stack trace from the top down. We see that the program (referenced in the stack trace as <module>) called the main function at line 27 in the source ?le makeintegerlist.py. In turn, a statement at line 23 in the main function invoked the create_list function. Code within the create_list function at line 16 called the get_int_in_range function. Finally, line 4 in the get_int_in_range function raised a ValueError exception when it called the int function with an invalid string literal (the value 'eleven' that the user provided).

Figure 12.4 diagrams the behavior of this sample run. When get_int_in_range calls the int conversion function with the argument 'eleven', the int function raises an exception. Since get_int_in_range has no exception handlers, the ValueError exception propagates back up the call chain to the nearest handler. The create_list function has a handler that can catch the exception, so it simply does not add any more elements to the list and returns the list as is to main.

The main function?s reaction to the exception is more general than that of create_list and get_int_in_range, since it is farther away from the exception origin. Since create_list contains signi?cant logic, main is powerless to attempt corrective action. Note that main cannot simply add a loop to allow the user to continue trying to add numbers each time the code within the try block raises an exception; doing so would call create_list again, and, since each call to create_list begins with the empty list, calling it again would lose any values the user entered before the exception appeared. The best main can do is report the program?s inability to create the list.

```
break
# Convert text integer into an actual integer
if int(value) == val:
count += 1
return count
```

Notice that we used read_file in Listing 8.31 (comparerolls.py) to read data from the results of a dice rolling experiment, but read_file is not really dice-speci?c. The function simply reads n integers from any compatible text ?le, counting how many times the value val appears in the ?le. Given the string name of a text ?le, read_file is to open the text ?le and read its contents consisting of integer data.

In Listing 12.14 (comparerollsrobust.py) the try statement wraps the single statement that calls read_file, since the rest of the code should execute safely. The operating system will raise a FileNotFoundError exception on an attempt to open a ?le that does not exist in the current folder. If the OS can ?nd the ?le but the user does not have permission to access the ?le, the OS will raise a PermissionError exception. The OSError covers all ?le related errors, such as attempting to process a corrupted ?le. In fact, just as Exception is the type that matches all routine exception types, the OSError type covers both the FileNotFoundError and PermissionError exceptions, as well as other ?le problems. We include the more speci?c exceptions to provide more helpful messages to the user. Also note that because OSError is more general than FileNotFoundError and PermissionError its except blcok must appear after the except blocks of both FileNotFoundError and PermissionError. If OSError?s except block appears in the source earlier, it will catch the ?le not found and permission error exceptions before the more speci?c handlers get a chance. The OSError exception type is good to use if you need to defend against all ?le processing errors but do not need the ?ner-grained control offered by the more speci?c ?le exception types.

It is important to note the absence of a catch-all handler, introduced in Section 12.5. Any type of exception other than those speci?ed by the except blocks will terminate the program with an error message. We could have added a catch-all exception handler that prints a message such as Some other error occurred, but such a message is no more helper to the user than a cryptic stack trace. To the developers, however, the stack trace printed by the uncaught exception is invaluable for precisely locating the source of the problem so they can address it.

As mentioned in Section 12.5, you should limit your use of catch-all exception handlers. Catch-all handlers have the potential to ?swallow? exceptions; that is, code within a function will catch an exception it did not expect and perhaps attempt some generic remedial action. The caught exception then will not propagate up to its caller, and so the caller (and its callers further up the call chain) will be cut off from the noti?cation of the problem.

What is an appropriate use of a catch-all exception handler? A called function may need to do some sort of local damage control if it encounters any kind of exception, expected or not. Perhaps the function simply needs to log the unexpected error in its own error ?le. In this case the function can use the catch-all exception handler. The last action in the catch-all exception handler should be re-raising the exception. This allows one or more of its callers up the call chain to deal with the exception. In situations that warrant a catch-all handler, prefer to catch the Exception type; use the untyped catch-all handler only as a last resort.

Sometimes it is appropriate for a function (or method) to catch an exception, take some action appropriate to its local context, and then re-raise the same exception so that the function?s caller can take further action if necessary. In essence, the function that catches the exception ?rst administers ??rst aid? and then passes the exception up the call chain for more advanced, application-speci?c treatment and care. In Listing 12.17 (reraise.py), the count_elements function accepts a list, lst, presumed to contain only integers, and a Boolean function predicate. The predicate function parameter accepts a single argument and returns true or false based on whether or not its argument parameter has a certain property. The program de?nes two such predicate functions: is_prime and non_neg. The is_prime function determines if its integer argument is prime, and the non_neg function determines if its argument is a nonnegative integer. Both is_prime and non_neg can raise a TypeError exception if the caller passes a non-integer argument. Neither the is_prime nor the non_neg function attempts to handle the TypeError exception itself. Listing 12.17 (reraise.py) exercises count_elements with the is_prime and non_neg functions.

```
try:
print(count_elements([3, -71, 22, -19, 2, 9], non_neg))
print(count_elements([2, 3, 4, 5, 6, 8, 9], is_prime))
print(count_elements([2, 4, '6', 8, 'x', 7], is_prime))
except TypeError:
print('Error in count_elements')
```

What is the reason for re-raising an exception? After all, the count_elements function could just print the message and continue. If it does so, however, the count that it eventually returns would be meaningless, and its caller would not know that count_elements had a problem. Re-raising the exception enables count_elements?s caller to be informed of the failure so the caller can react to the exception in its own way.

If C raises an exception, functions A and B both may need to know about it to take appropriate action. Function B is closer to C in the call chain. B can catch the exception raised by C, remedy the situation as best it can, and then ensure that its caller (A) receives the same exception. A then can take action appropriate to its own context.

The idea is that B is the caller in the call chain closest the exception origin (C), and B has information unique to its context that its caller (A) would not have. B should handle any exceptions it expects and can handle in some way. If the exception is such that B can repair the situation, continue its execution, and in the end correctly ful?ll A?s expectations, then there is no reason for B to re-raise the exceptions it caught from C. On the other hand, if C?s exception renders B unable to meet A?s expectations, B can do local damage control but must also raise an exception that A can process. Often this means re-raising the same exception, but it can mean raising a different exception that is more B-speci?c.

The Python try statement supports an optional else block. Its behavior is remimiscent of the while statement?s else block (see Section 5.6). If the code within the try block does not produce an exception, no except blocks trigger, and the program?s execution continues with code in the else block. Figure 12.6 contrasts the possible program execution ?ows within a try/else statement. The else block, if present, must appear after all of the except blocks.

Since the code in the else block executes only if the code in the try block does not raise an exception, why not just append the code in the else block to the end of the code within the try block and eliminate the else block altogether? The code restructured in this way may not behave identically to the original code. Consider Listing 12.18 (trynoelse.py) which demonstrates the different behavior.

Listing 12.19 (riskyread.py) does not use a with/as statement as recommended in Section 9.3, and so it will have a problem should an exception arise before the program executes the f.close() statement. The ?le could be corrupted and one or more of the lines could contain text that is not convertible to a number. Either of these problems would raise an exception before the f.close method executes.

Listing 12.20 (try?leread.py) uses two try statements. The ?rst try statement defends against an OSError exception. The operating system will prompt the open function to raise such an exception if it cannot satisfy the request; for example, the ?le may not exist in the current directory or the user may not have suf?cient permissions to access the ?le. The program does not proceed if it cannot open the ?le for reading.

Because of the design of the ?le class, the with/as statement takes care of the details of properly closing a ?le should an exception arise. The with/as statement, however, will not automatically handle any exceptions. We can remedy this with with another pair of nested try statements, as Listing 12.22 (better?leread.py) shows.

Both approaches compute the same result. However, the second approach always raises and handles an exception. The exception de?nitely is not an uncommon occurrence. You should not use exceptions to dictate normal logical ?ow. While very useful for its intended purpose, the exception mechanism adds some overhead to program execution, especially when an exception is raised. This overhead is reasonable when exceptions are rare but not when exceptions are part of the program?s normal execution.

```
lst = [0, 0, 0, 0]
with open('data.txt', 'r') as f:
count = 0
for line in f.readlines():
lst[count] = int(line)
count += 1
```

```
try:
x = int(input())
print(x)
except ValueError:
print('Wrong!')
print('End')
```

```
try:
x = int(input())
print(x)
except IndexError:
print('Wrong!')
print('End')
```

```
try:
x = int(input())
print(x)
except Exception:
print('Wrong!')
print('End')
```

```
try:
x = int(input())
print(x)
except ValueError:
print('Wrong!')
else:
print('Wow')
print('End')
```

```
try:
x = int(input())
print(x)
except ValueError:
print('Wrong!')
finally:
print('Done')
print('End')
```

```
try:
x = int(input())
print(x)
except ValueError:
print('Wrong!')
else:
print('Wow')
finally:
print('Done')
print('End')
```

Chapter 13

Custom Types

We have examined many of Python?s built-in types. Some, like, integers, ?oating-point numbers, and Booleans are relatively simple, while others such as lists, tuples, dictionaries, sets, and exceptions are more complex. Python?s rich collection of built-in types enable us to write a wide variety of programs in diverse problem domains. Python also provides the ability for programmers to design their own custom types by which developers can craft data types that more closely model the problem at hand. This better alignment of software assets with the problem domain can expedite the development process.

A software object generally bundles together data (instance variables) and functionality (methods). The instance variables and methods of an object comprise its members. The class of an object de?nes the object?s basic structure and capabilities. As a simple concrete example, consider the familiar geometric circle, shown in Figure 13.1. Given a circle?s radius (r in Figure 13.1), we can compute the circle?s area and circumference. The circle?s center, (x,y), establishes the circle?s position. We will de?ne a custom Circle class in Python from which we can create Circle instances (objects). The Circle class speci?es what circle objects can do and how clients (that is, code outside of the Circle class needing the services that a Circle object can provide) can interact with them. A center and radius may be good enough for mathematicians, but in a graphical computer program circle objects may need other attributes like a ?ll color, ?ll style, edge thickness, etc. We will keep things simple for now and stick to the abstract mathematical concept of a circle.

What data must each Circle object maintain? Since circles can appear in various places, each circle must keep track of its own position. Just like in mathematics, we can specify the location of a Circle object by its (x,y) center. Also, since some circles are larger or smaller than others, each Circle should have its own radius. It is natural, then, for our Circle object to have a center instance variable and a radius instance variable.

Should our circle objects have an area instance variable and/or a circumference instance variable? Both area and circumference depend solely on a circle?s radius, so if we include a radius instance variable, both area and circumference would be redundant information. Besides, we easily can compute them as needed with simple formulas. We will implement area and circumference as methods in our Circle class.

r

(x, y)

A = ?r2

C = 2?r

block

We see in Listing 13.1 (circle.py) that a class de?nition begins with the reserved word class followed by the name of the class and a colon (:) at the end of the line. Figure 13.2 shows the general form of a class de?nition. As in function de?nitions, the body of the class is indented within a block of code. The code in this block looks like a series of function de?nitions; however, since the de?nitions appear within the block of a class de?nition they are method de?nitions. As with functions, we can (and should) document classes and methods with docstrings.

Notice that each method de?nition has self for its ?rst parameter. The language does not require the ?rst parameter?s name to be self (it could be x, obj, or any valid identi?er), but the universal convention in the Python programming world is to use the name self. During a method?s execution, the self parameter references the object on whose behalf the method is being invoked. In Turtle graphics, for example, the Turtle class de?nition contains a forward method de?nition that begins

```
circ = Circle((2.5, 6), 5)
```

circ

center

(2.5, 6)

radius

5

This statement creates a new Circle object with a center at (10,3.4) and radius 5. It implicitly invokes the __init__ method to do the initialization work. The statement then assigns the variable circ to this newly created Circle object. The constructor ?rst ensures that the radius parameter is nonnegative. An attempt to make a circle with a negative radius produces an exception. Next, the constructor initializes the center and radius instance variables of the objects it is in the process of creating. Within method de?nitions (like __init__), we pre?x instance variable names with self. Any variables not pre?xed with self are treated as normal local or global variables. In the following two statements in the constructor:

The methods get_center and get_radius are sometimes called accessor methods, or getters, as they give clients access to see the state of an object. In this case, clients can obtain a Circle object?s center and radius via these methods. In contrast, the methods move, grow, and shrink are known as mutator methods, or setters, because they allow clients to modify the state of an object. Note that grow and shrink do not allow arbitrary changes; they allow clients to adjust a circle?s radius only by one-unit increments.

The methods get_area and get_circumference are neither accessors not mutators. They do not provide direct access to the data in a Circle object but rather provide indirect access via a computation (you could reverse engineer the result of either method to deduce the radius, but the get_radius method provides direct access).

calls the onclick method of the Screen class. Like functions (see Section 8.5), methods can accept functions and other methods as parameters. The onclick method accepts a function as a parameter. The turtle module provides the Screen.onclick method, and we must provide the function we send to it. The only restriction on the function we send to onclick is that the function must accept two integer parameters. In this case we pass it our do_click function. The purpose of the onclick function is to register a callback function with the Turtle graphics framework. The framework will ?call back? the function when a certain event occurs. If the user clicks the mouse when the pointer is over the graphics window, the framework will call the function registered with onclick (in this case do_click) and pass to it the x and y coordinates of the mouse pointer at the time of the click. That is why this callback function requires a function that expects the two integer parameters. Note that our do_click function packs up into a tuple the x and y coordinates it receives and passes this tuple on to the global Circle object circ?s move method. We must use a global variable in the do_click function because we cannot pass in the circ object?remember, we do not call do_click, the framework does, and the framework only passes in the mouse pointer coordinates.

Note that the do_click, do_up, and do_down functions modify the global Circle object circ?s state, but the draw_circle method simply uses the Circle object?s get_center and get_radius methods to be able to draw the circle using the Turtle object?s circle method. The Turtle.circle method draws a circle starting from the turtle?s current position?three o?clock on the circle?and it draws a counter-clockwise curve around the circle.

Listing 13.2 (circlemaker.py) uses global variables to give multiple functions access to the same Turtle and Circle object. Section 8.1 exposed some of the disadvantages of using global variables. Object-oriented techniques allow us to con?ne globals to classes. Listing 13.3 (circlemakerobject.py) is a rewrite of Listing 13.2 (circlemaker.py) that moves the previously global Turtle and Circle objects inside of a new object of type GraphicalCircle.

As an aside, the turtle.TurtleScreen object named screen is not an instance variable; instead, it is a local variable within the GraphicalCircle.__init__ method. Observe that none of the other methods within the GraphicalCircle class need to access screen. We know that local variables disappear when the function in which they are used returns. Does this mean that the screen object may be garbage collected (see Section 9.9) before the user is ?nished running the program?

The developers of the Circle class intend that clients should not directly manipulate a Circle object?s center and radius instance variables. The constructor (__init__), move, grow, and shrink are the only methods that can in?uence the state of a Circle object; however, nothing prevents client code from accessing the instance variables directly via the dot (.) operator. The following interactive sequence illustrates:

As you can see, internally Python changes the name of a class member with a name that begins with double underscores by pre?xing it with a single underscore followed by the class name. Python thus does not provide a way to truly protect object members from the outside world. This sets Python apart from most other mainstream object-oriented languages like C++, Java, and C# which provide language features that can protect the internal details of an object.

The concept of mathematical rational numbers is accessible even to elementary school students, so it presents an ideal platform for further exploration into custom class design in Python. We will de?ne our own simple rational number class. Note that if you are developing an application that requires the use of rational numbers, you ought to use Python?s standard Fraction class that we saw in Section 9.4. We build our own class here because our goal in this section is simply to gain more experience developing custom classes in Python.

```
def __init__(self, num, den):
self.numerator = num
if den != 0:
self.denominator = den
else:
```

Surprisingly, the second statement (assignment of fract.__denominator) does not affect the __denominator ?eld used by the methods in the Rational class; it instead adds a new, unprotected ?eld named __denominator. The client cannot get to the protected ?eld by merely using the dot (.) operator. To avoid such confusion, a client should not attempt to use instance variables of an object with names that begin with two underscores.

We may de?ne a __str__ method for any class. It is one of the special methods like __add__, __sub__, __eq__, etc. introduced in Section 9.4 that provide syntactic sugar to the language. The interpreter calls an object?s __str__ method when executing code that requires a string representation of an object; for example, the print function converts an object into a string so it can display textual output. The str string constructor function also uses the __str__ method of an object behind the scenes. The following simple program demonstrates how this works:

```
# Make some X objects
a = X()
b = X()
c = X()
# Print them
print(a, b, c)
print("--------------")
message = str(b)
print(message)
```

The expression fract3 * fract4 is syntactic sugar for fract3.__mul__(fract4). We leave the implementation of the Rational.__add__ method to the reader; it is a bit more complicated, as rational number addition involves ?nding a common denominator and making adjustments to the numerators to make them compatible for adding together.

This means the code within set_numerator sees self as fract1 and n as the integer object 2. During this particular execution of the set_numerator method the expression self.__numerator within the method?s de?nition refers to the fract1 object?s __numerator ?eld. The method, therefore, reassigns the __numerator member of fract1.

In the Rational class in Listing 13.4 (rational.py) clients can see the values of the instance variables via methods get_numerator and get_denominator. These methods do not change the state of a Rational object. Recalling the terminology from Section 13.1, these methods are often called getter methods because they can get the values of instance variables but cannot change the instance variables. In contrast, set_numerator and set_denominator are mutator methods because they can change the values of instance variables.

When the values of one or more instance variables in an object change, we say the object changes its state; for example, if we use an object to model the behavior of a traf?c light, the object likely will contain an instance variable that represents its current color: red, yellow, or green. When that ?eld changes, the traf?c light?s color changes. In the green to yellow transition, we can say the light goes from the state of being green to the state of being yellow.

Consider a non-programming example. If I deposit $1,000.00 dollars into a bank, the bank then has custody of my money. It is still my money, so I theoretically can reclaim it at any time. The bank stores money in its safe, and my money is in the safe as well. Suppose I wish to withdraw $100 dollars from my account. Since I have $1,000 total in my account, the transaction should be no problem. What is wrong with the following scenario:

This is not the process a normal bank uses to handle withdrawals. In a perfect world where everyone is honest and makes no mistakes, all is well. In reality, many customers might be dishonest and intentionally take more money than they report. Even though I faithfully counted out my funds, perhaps some of the bills were stuck to each other and I made an honest mistake by picking up six $20 bills instead of ?ve. If I place the bills in my wallet with other money that is there already, I may never detect the error. Clearly a bank needs a more controlled procedure for customer withdrawals.

This code using a Stopwatch object is simpler. A programmer writes code using a Stopwatch in a similar way to using an actual stopwatch: push a button to start the clock (call the start method), push a button to stop the clock (call the stop method), and then read the elapsed time (use the result of the elapsed method). Programmers using a Stopwatch object in their code are much less likely to make a mistake because the details that make it work are hidden and inaccessible.

Section 9.6 introduced the Tk graphical user interface toolkit. Listing 9.14 (tkinterlight.py) provides a program that draws three circular lamps of various colors and a rectangular frame. It supports user interaction and simulates a standard traf?c light. What if we wish to build a more sophisticated application that graphically models an intersection containing four traf?c lights? We could follow the same approach, writing code to draw four rectangular frames and 12 circular lamps. Making sure all the rectangular frames and circular lamps are in the correct locations would be tedious work. Next consider an expansion of the application that is to model the scores of intersections and hundreds of traf?c lights within a community. Coding all the parts of the lights in their correct positions within the graphics window would be beyond tedious.

We really are not interested in drawing rectangles and circles in our more sophisticated application; we actually need to be able to draw traf?c lights. Further, such traf?c lights ought to know their current color and automatically be able to become a different color when told to do so. In fact, a traf?c light?s normal operating procedure should be to change from red to green, green to yellow, and yellow to red. Other transitions, such as green to red or red to yellow should not be supported. A single program must be able to use multiple traf?c lights with each light maintaining its own independent operation.

By now you probably have thought of the solution. We need a TrafficLight class that enables us to create TrafficLight objects. Each TrafficLight object should contain instance variables that represents the light?s position and current color. The class constructor could set the light?s position and initial color. A change method could transition the light?s current color to its new color, ensuring no out-of-sequence color changes.

Recall the Tk program in Listing 9.14 (tkinterlight.py) that allows the user to cycle through the colors of a graphical traf?c light by pressing a button. We will use that code as a starting point in designing our traf?c light class. We want clients to be able to specify the position and size of traf?c light objects within the graphics window. Clients also should be able to specify the light?s initial color. To simplify things for this example, we will not provide to clients the ability to reposition or resize a light after its creation. After creating a traf?c light object, a clients can cycle through its color states via a change method.

? The TrafficLight constructor accepts an (x,y) position and width for the graphical traf?c light. The constructor calculates the light?s height and the size and placement of its three lamps from these three parameters. The constructor also accepts a Tk canvas reference from the client. This means the client is responsible for setting up the Tk graphics environment before creating a TrafficLight object. Finally, the constructor can accept an initial color from the client. The color is one of the following strings: "red", "yellow", or "green". The constructor throws a ValueError exception if the client attempts to pass an object as the last argument that is not one of these strings. If the client omits this parameter, the color defaults to "red".

? The event handling code in Listing 13.9 (traf?clightobject.py) does not access any global variables. In
fact, global variables are absent from both the Listing 13.8 (traf?clight.py) and Listing 13.9 (traf?clightobject.py)
modules. This is an important distinction from the code in Listing 9.14 (tkinterlight.py) which required
global variables to achieve the necessary communication with the function that changed the light?s
color. The instance variables within TrafficLight objects assume the roles formerly played by
global variables in Listing 9.14 (tkinterlight.py).

? The widget grid method positions a widget within a grid of rows and columns within its parent. The root widget is the parent of all nine frame widgets. The grid method places every frame widget in row 0 (the ?rst row), but its column depends on i, the loop variable. Within each frame, the grid method places each canvas widget at row 0, column 0 and each button widget at row 1, column 0.

? Note the application of the functools.partial function (see Section 8.11). The command keyword argument in the tkinter.ttk.Button class constructor must be paired with a function or method that expects no parameters. The do_button_press method expects two parameters?self and idx (for index). How do we make these two parameters ?go away?? In the expression partial(self.do_button_press, i - 1) the self. pre?x for do_button_press takes care of the self parameter. That leaves the idx parameter. The expression partial(self.do_button_press, i - 1) makes a new function with the idx parameter hardwired in to be i - 1. The ?rst time through the loop registers a callback to self.do_button_press with the argument 0, the second time through the loop the argument is 1, and so forth. Each call to partial creates a new, independent function, and each function created during a loop iterator becomes the callback of the button object created during that iteration. The net effect is our do_button_press method can ?exibly call the change method of the proper traf?c light object.

Here we use a lambda function (see Section 8.7) with a default argument of the current loop variable?s value. As in the partial function application, this approach creates nine separate independent functions, each associated with a particular button. This approach makes the do_button_press method super?uous; if we use the lambda functions, we can remove the do_button_press method from the class.

We know that just because a program runs to completion without a run-time error it does not imply that the program works correctly. We can detect logic errors in our code as we interact with the executing program. The process of exercising code to reveal errors or demonstrate the lack thereof is called testing. The informal testing that we have done up to this point has been adequate, but serious software development demands a more formal approach. Good testing requires the same skills and creativity as programming itself.

Weaknesses in the standard approach to testing led to a new strategy: test-driven development. In test-driven development the testing is automated, and the design and implementation of good tests is just as important as the design and development of the actual program. In pure test-driven development, test engineers develop tests based on the problem?s requirements before developers write any application code. Testers then can subject all new application code to the pre-existing tests as soon as the code becomes available.

During its execution, the FunctionTester.check method calls the function it receives with the speci?ed parameters and compares the actual result to the expected result. The method will indicate the test?s success or failure and track the number of failed tests. Clients call the report_results method after performing all the tests. The report_results method provides the error statistics for all the tests performed.

In the sum function, the programmer was careless and used 1 as the beginning index for the list. Notice that the ?rst test does not catch the error, since the element in the zeroth position (zero) does not affect the outcome. A tester must be creative and even devious to try and force the code under test to demonstrate its errors.

Notice that even though we have yet to implement the maximum function, we can test it anyway. This is true test-driven development?design the tests ?rst, then write the code. The maximum function here is an example of a stub. A stub is a function or method that does not provide the expected functionality but may be executed without causing a run-time error. A stub ordinarily ignores all parameters passed to it and, if necessary, returns a default value of the type expected by its caller. The yet-to-be implemented maximum function simply returns zero?notice that it accidentally passes one of the tests. We can de?ne a square root function stub as

The FunctionTester class from Listing 13.12 (functiontester.py) has some limitations; for example, testers cannot use it to check the correctness of a function that sorts a list in place. The FunctionTester.check method determines only if a function returns the expected result. It also cannot test that a function does not modify a mutable parameter; for example, in the course of their execution the sum and maximum functions in Listing 13.13 (testliststuff.py) should not modify the list passed by a caller. We could enhance the FunctionTester class to check that functions handle mutable parameters properly.

Section 13.1 showed how we can set up the instance variables of an object in the __init__ method of its class. This technique ensures every object of that class will contain those instance variables. Python allows programmers to add instance variables dynamically to individual objects. Consider the following very simple class:

The setattr function stands for set attribute, and getattr stands for get attribute. The term attribute is another name for instance variable. Note that in the calls to the functions setattr and getattr the instance variable name x is expressed as the string "x". If the name x did not appear in quotes, it would refer to a variable named x; such a variable may or may not exist, but, more importantly, it would not refer to the instance variable named x in the g object.

```
def main():
rect1 = Rectangle((2, 3), 5, 7)
rect2 = Rectangle((5, 13), 1, 3)
rect3 = Rectangle((20, 40), -5, 45)
rect4 = Rectangle((-510, -220), 5, -4)

print(rect1.get_width())
print(rect1.get_height())
print(rect2.get_width())
print(rect2.get_height())
print(rect3.get_width())
print(rect3.get_height())
print(rect4.get_width())
print(rect4.get_height())
print(rect1.get_perimeter())
print(rect1.get_area())
print(rect2.get_perimeter())
print(rect2.get_area())
print(rect3.get_perimeter())
print(rect3.get_area())
print(rect4.get_perimeter())
print(rect4.get_area())
```

9. Develop a Circle class that, like the Rectangle class above, provides methods to compute perimeter and area. The Rectangle instance variables are not appropriate for circles; speci?cally, circles do not have corners, and there is no need to specify a width and height. A center point and a radius more naturally describe a circle. Build your Circle class appropriately.

```
w1 = Widget()
w2 = Widget(5)
print(w1.get())
print(w2.get())
w1.bump()
w2.bump()
print(w1.get())
print(w2.get())
for i in range(20):
w1.bump()
w2.bump()
print(w1.get())
print(w2.get())
```

Chapter 14

Class Design: Composition and Inheritance

The typical Python class contains instance variables, usually established by the class constructor. These instance variables refer to other objects. If these instance variables are essential to the intrinsic meaning of the class, we say objects of the class are composed of other, more fundamental objects. This relationship represents an object-oriented design technique known as composition. Examples of composition abound in the classes we have designed so far:

? Listing 13.8 (traf?clight.py): TrafficLight objects are composed of a string object (color), a Tk object (canvas), and three integer objects (red_lamp, yellow_lamp, and green_lamp?these are integers returned by the Canvas.create_oval method; a canvas object keeps track of its shapes via integer indices known as tags.

We can deduce the composite nature of our classes by examining the instance variables used within their methods. Any expression of within a method of the form self.something, where something refers to an object de?ned outside of the class itself indicates a dependence on an external object and is evidence of composition. Unless it was super?uous or unused, removing the reference to that external object will reduce the functionality of the class or possibly make the class completely inoperable.

Some of the objects involved in Listing 13.10 (multisizelightwindow.py) form a similar but slightly different relationship. Each MultisizeLightWindow object contains a list of nine TrafficLight and Button objects. The quantity nine is arbitrary?we could have had ?ve or 20 lights of different sizes, and the MultisizeLightWindow object would not behave in a fundamentally different manner. In this case we cannot claim that MultisizeLightWindow objects of composed of multiple traf?c light objects, but rather we say that MultisizeLightWindow objects contain an aggregate of traf?c light objects. This concept is called aggregation. In some cases the line between composition and aggregation is not so clear cut and can be open to interpretation. Both aggregation and composition model the has a relationship between objects, but the composition model models a stronger relationship, the needs a relationship. Composition is sometimes called a part-whole relationship.

In Listing 14.1 (comptraf?clight.py), objects of the class CompLamp class form the building blocks for traf?c lights. The CompTrafficLight class stores three of these lamp objects in a dictionary. This may appear to be aggregation, but for a traf?c signal of this kind the quantity three is neither arbitrary nor variable, so in this case we safely can classify the lamps to traf?c light relationship as composition.

The constructor in the CompLamp class accepts four required arguments (self, parent, width, and order), one named default argument (color), and additional optional arguments (*args) and keyword arguments (**kwargs). The ?rst ?ve parameters are meant for the CompLamp class itself, and the *args and **kwargs arguments are meant for the ttk.Frame instance variable on which a CompLamp object relies to handle the graphics. The constructor makes these extra parameters available just in case a client requires special control over the style of the frame object.

A CompLamp object basically is a rectangular Tk frame that contains a Tk canvas widget that draws a circular shape. A lamp has an ?on? color and its ?off? color is black. The turn_on and turn_off methods adjust the lamp?s color accordingly. The resize method allows clients to change the size of a lamp after its creation.

Listing 14.1 (comptraf?clight.py) uses Tk widgets to produce the graphical images and deal with user interaction. These widgets are themselves Python objects. Our custom lamp and traf?c light objects use Tk objects, but our custom objects and the Tk objects live in two different worlds. We developed our custom classes, and someone else developed the Tk classes. We must write extra code to allow our custom objects and Tk objects to interoperate correctly to achieve the effects we desire. This extra code is the price we pay for using a library that we did not write and, therefore, have no control over its contents. Our traf?c light object must have an associated Tk frame object because we want to be able to do things with Tk frames. What if we could design a custom traf?c light class that itself was a Tk Frame class? That could simplify our code considerably. Section 14.2 introduces the concept of inheritance which makes this possible.

We can base a new class on an existing class using a technique known as inheritance. Recall our Stopwatch class we de?ned in Listing 13.6 (stopwatch.py). Clients can start and stop Stopwatch objects as often as necessary without resetting the time. Suppose we need a stopwatch object that records the number of times the watch has been started until it is reset. We can build our enhanced Stopwatch class from scratch, but it would more ef?cient to begin with our existing unadorned Stopwatch class and somehow add on the features we need. We will not merely copy the source code from our existing Stopwatch class and then add code to it. The inheritance mechanism does not touch the source code of our original Stopwatch class, and, at the same time, it allows us to write as little new code as possible. Listing 14.3 (countingstopwatch.py) provides an example of inheritance, de?ning the new class of our enhanced stopwatch objects.

de?nes a new class named CountingStopwatch, but this new class is based on the existing class Stopwatch. This single line enables the CountingStopwatch class to inherit characteristics from the Stopwatch class. CountingStopwatch objects automatically will have start, stop, reset, and elapsed methods because the Stopwatch class has them.

within its constructor de?nition invokes the constructor of its superclass. The expression super() thus refers to code in the superclass. The superclass constructor calls the reset method which establishes the _start_time, _elapsed, and _running instance variables. After executing the superclass constructor code, the subclass constructor de?nes and initializes a new instance variable named _count.

What happens if you do not provide a constructor (__init__ method) for the subclass? In this case when creating an instance of the subclass the interpreter automatically will invoke the superclass constructor. If you do provide an __init__ constructor in a subclass, the interpreter will not call the superclass constructor?you must call the superclass constructor via the super function to ensure the initialization responsibilities of the superclass occur.

rule of thumb is this: you generally should invoke the super() version of the method within a method you are overriding. The super invocation within a method of a class that has exactly one superclass allows the superclass to manage the details known to the superclass. The subclass can take care of the details speci?c to the subclass via the rest of the code in the overridden method.

Notice that the CountingStopwatch class has no apparent stop method. In fact, it does have a stop method because it inherits the stop method as is from the Stopwatch class. The start and reset methods in CountingStopwatch must work with the new __count instance variable and thus must be overridden, but the stop method needs no enhancement at all. Since stop needs no changes or additions, we omit its de?nition within the CountingStopwatch class. A derived class may inherit a method as is from its base class, or it may override the method. An overridden method may invoke the services of the base class version via super and add some additional functionality, or, less commonly, it may not use the base class version of the method at all and do something completely different.

The type function reveals that the sw variable refers to an object of type Stopwatch and csw refers to an object of type CountingStopwatch. It follows that the isinstance function indicates that sw refers to an object that is an instance of Stopwatch. Similarly, csw?s object is an instance of CountingStopwatch. Not surprisingly, isinstance indicates that sw is not an instance of CountingStopwatch. Perhaps surprisingly, however, we see that csw is an instance of both the CountingStopwatch class and Stopwatch class!

Inheritance establishes a special relationship between two classes. An instance of a derived class is also an instance of the base class. In object-oriented software terminology this known as the is a relationship. The major bene?t of this is a relationship is this: an instance of a derived class may safely be used in any context meant to work with an instance of its base class. Suppose we write a function for timing a particular process:

The parameter is supposed to be a reference to a Stopwatch object. As such the time_process function can exercise any methods that a Stopwatch object provides: start, stop, and reset. Any class derived from Stopwatch will automatically inherit these methods and may or may not override their behavior. Regardless, the code within the time_process function legitimately may call the start, stop, or reset method on a timer object if its actual type is Stopwatch or CountingStopwatch (or any other class we may derive from Stopwatch).

Listing 14.6 (restrictedstopwatch.py) derives RestrictedStopwatch from CountingStopwatch. The CountingStopwatch class has Stopwatch as its base class. We say that CountingStopwatch is a direct base class of RestrictedStopwatch and Stopwatch is an indirect base class of RestrictedStopwatch. The inheritance is transitive: since RestrictedStopwatch inherits from CountingStopwatch and CountingStopwatch inherits from Stopwatch, it is the case that RestrictedStopwatch inherits from Stopwatch as well. We can demonstrate this transitivity in the interactive interpreter:

At this point we have several classes derived from Stopwatch. A collection of classes related through inheritance is called a class hierarchy, or inheritance hierarchy. Figure 14.2 illustrates the Stopwatch hierarchy using a standard graphical notation known as the Uni?ed Modeling Language, or UML (see http://www.uml.org). In the UML, a rectangle represents a class, and an arrow with a hollow arrowhead points from a derived class to its immediate base class. A UML class diagram such as the one in Figure 14.2 communicates to developers the relationships amongst the classes more quickly than the textual Python

Section 14.1 alluded to the conceptual mismatch between the custom classes from Listing 14.1 (comptraf?clight.py) and the standard Tk widget classes on which the custom classes depend. Inheritance provides a way to unify these two distinct sets of classes. We can build custom Tk widgets from existing Tk widget classes using inheritance. Listing 14.10 (traf?csignal.py) uses both composition and inheritance in the design of a new traf?c light class called TrafficSignal.

The TurnLamp class inherits from SignalLamp. The superclass can draw the circle, but the subclass must provide the code to draw the three lines that comprise a simple arrow pointing to the left indicating the permission to make a left turn. Because TurnLamps present more complicated shapes to draw, we must override the turn_on, turn_off, and resize methods. Note that the resize method invokes the superclass version of resize to take care of the parts of the lamp that do not involve repositioning the lines of the arrow.

Review the code in Listing 14.13 (movablelighttest.py) and Listing 14.15 (turnlighttest.py). Do you see anything troubling in the two applications? The two programs look almost identical. Both programs present the same interface to the user; that is, the same reactions to mouse clicks and button presses. The only difference is the exact type of traf?c light the user can manipulate. Listing 14.13 (movablelighttest.py) hardwires a MovableLight object into the code, while Listing 14.15 (turnlighttest.py) instead uses a TurnLight.

Often we can choose between using composition or inheritance to leverage the functionality of an existing class. The better choice is not always apparent. We can achieve the effects of inheritance using strictly composition and delegation. Our CountingStopwatch class in Listing 14.3 (countingstopwatch.py) uses inheritance to create the CountingStopwatch class from the simpler Stopwatch class. Listing 14.17 (countingstopwatch2.py) de?nes the class CountingStopwatch2 that behaves identically to CountingStopwatch, but it uses composition to reuse the functionality of Stopwatch. Clients would not perceive any functional differences between a CountingStopwatch instance and a CountingStopwatch2 instance.

While the inheritance version and the composition version both accomplish the same goals from the client?s perspective, notice that the composition version requires more code. With inheritance, if a method from the superclass needs no changes, the programmer omits its de?nition in the subclass. With composition, however, every method meant to be used by clients in the original class must have a de?nition in the new class. If the new class does not need to change the method in any way, it simply delegates the work to the contained instance of the original class. We see this in the stop and elapsed methods in CountingStopwatch2. This can make a big difference in the work required to design a new class if the original class has many methods that need no change. We must maintain the interface of the original class if we to simulate the is a relationship without using inheritance.

Good object-oriented designs often combine composition and inheritance to achieve useful results. Suppose we are building a software system that manages a manufacturing process. A physical temperature sensor attached to a piece of equipment relays temperature information to the software via a software object. The software expects to receive an instance of the TemperatureSensor class from the sensor. The TemperatureSensor class provides only two methods: read, which returns the current temperature in degrees Celsius of its attached hardware, and test, which puts the sensor into a self test mode. The software on the physical sensor that sends the TemperatureSensor objects to the system is propriety, and its license forbids reverse engineering to change its behavior. You are stuck with sensors that provide TemperatureSensor objects. Fortunately, since the part of the software system that uses the information sent by the sensors is made by the same company that provides the sensors, everything works well together.

One day your company decides to replace the existing equipment management software with a new system offered by another vendor. The new system provides much greater control and monitoring capabilities, and the cost of the annual licensing fees are lower than the exiting system. There is a problem, however. The new system expects different kinds of sensors on the equipment to monitor. The sensors the new software requires send information via objects of type ThermalValue. The ThermalValue class provides only

The new system receives a ThermalValue object, because a TemperatureSensorAdapter object is a ThermalValue object. Inheritance guarantees the is a relationship. The temperature method delegates most of its work to the contained TemperatureSensor object. The TemperatureSensor object provides the temperature, and the method merely needs to convert the temperature from Celsius to Fahrenheit to conform to the expectations of the new system.

Note that this solution uses both inheritance and composition. It is an example of an object-oriented design pattern. A design pattern provides a solution to a commonly occurring problem in software design (see https://en.wikipedia.org/wiki/Software design pattern). A design pattern does not provide an exact solution to a given problem; it instead shows how techniques such as inheritance and composition may be applied to solve problems of a particular kind. The link above describes over 50 design patterns.

Each design pattern has a name, and the design pattern involved in our TemperatureSensorAdapter class is aptly named adapter. Our TemperatureSensorAdapter class adapts both the interface (the provided method is named read, but the required method name is temperature) and computation (the provided units are degrees Celsius, but the required units are degrees Fahrenheit). The adapter pattern is sometimes called the wrapper pattern because object serves as a wrapper around another object. In our example, a TemperatureSensorAdapter object wraps a ThermalValue object.

Consider a motor vehicle. The drive train of a motor vehicle contains (sounds like composition) axles and wheels, among other essential parts. Each axle assembly consists of an axle and two wheels. Would it make sense to derive an axle assembly class from an axle class and add a left wheel object and a right wheel object as new instance variables? While this would work, we must ask: Is an axle assembly an axle (is a relationship)? No, an axle assembly has an axle and has two wheels (has a relationships), so composition is the better choice in this case.

Our foray into multiple inheritance requires a gentle warning. Python distinguishes itself from most other programming languages due to its relatively simple and straightforward way of allowing programmers to express solutions to problems. Python ordinarily presents a very low barrier for transforming thought into code. Programmers must be diligent at all times, of course, but, in order to implement multiple inheritance in a sound way, Python requires programmers to take extra when designing with multiple inheritance.

Remember the ambiguity in subclasses introduced by multiple inheritance when superclass methods have the same name? We could be careful and make sure the methods in all our classes have unique names. Would this approach eliminate the ambiguity problem at the expense, perhaps, of creating mangled, unnatural names for some of our methods? Potentially convoluted names de?nitely are not a good idea, and, besides, this approach would not eliminate the ambiguity problem. Some methods have names that cannot change; for example, class constructors always are named __init__.

Our intention is that C objects should combine the capabilities of A objects and B objects; we want every C object to have both value_A and value_B instance variables. With single inheritance everything would work as expected. We do not provide a constructor for C, so, as with single inheritance, the creation of a C object would execute the superclass __init__ method, if it exists. Given our earlier experiment with multiple inheritance, what would you expect the following code to print?

The story does not end here, unfortunately. None of the constructors in our A, B, or C classes speci?ed any parameters. What if one superclass contains an __init__ method that expects certain parameters, while another superclass speci?es a different number and/or different types of parameters? If we change class B?s de?nition to be

```
def __init__(self):
print("Making a C object")
A.__init__(self)
B.__init__(self, 5)
self.value_C = 2
```

This quick-and-dirty ?x achieves the desired behavior in this case, but it makes for a far less ?exible design. If we add another superclass to class C?s list of superclasses, we must modify the code in C?s constructor to work properly with the constructor of the additional superclass. Also, by removing the super().__init__() calls in A and B, we render both classes unable to work properly if we want to involve them in a better designed multiple inheritance hierarchy of classes.

(There is nothing special about the name kwargs, and any valid identi?er is acceptable, but the convention is to use kwargs to indicate keyword arguments in a function or method de?nition.) This allows the constructor to accept any number of any types of arguments, but it requires clients to use keyword arguments when creating an instance of the class.

which we can visualize as
object
?
A
?
B
?
C
?
D
?
E
?
F

within any method of class F executes the f method de?ned in class F. Compare that to the same expression appearing in class C. Since class C does not de?ne (override) f, the executing program attempts to execute the f method in the superclass of C; that is, B. Class B does not de?ne (override) f, so the executing program attempts to execute the f method in class A. Class A does provide a de?nition for method f, so the actual code that executes is found in class A, even though it is called from a method appearing within class C.

It is tempting to believe that the word super refers to superclasses. That always is true for single inheritance, but multiple inheritance clouds the meaning of the super function. The order in which an executing program attempts to resolve method calls is known as the method resolution order, or MRO for short. For single inheritance, the MRO is easy to determine: starting with the given class, if you cannot ?nd the method being called within that class, walk up the hierarchy of classes until you ?nd the nearest class that de?nes the method. If the method is inherited, it will be de?ned in a superclass and eventually located. If the method does not exist, the search terminates at class object and the interpreter raises an exception.

10

5

9

2

6

```
class E(C, D): pass
class F(B): pass
class G(B): pass
class H(F, G): pass
class I(E, H): pass
```

The MRO does make sense. The process is this: keep moving up the hierarchy until moving up one more time would mean visiting a class before one of its subclasses. In the example we can go from I to E to C, but we cannot next go to A since we have yet to visit D, and D is a subclass of A. If we cannot go up, we go sideways to the right. The right movement corresponds to the left-to-right ordering of superclasses in a class de?nition.

10

6 9

3 5

2 4

```
class A(object): pass
class B(object): pass
class C(A): pass
class D(A): pass
class F(B): pass
class E(C, D, F): pass
class G(B): pass
class H(D, F, G): pass
class I(E, H): pass
```

Every class in the multiple inheritance hierarchy, except those derived directly from object, must call __init__ via super and pass keyword arguments. This form of the constructor invocation with super makes cooperative multiple inheritance possible. It enables the constructors for other classes in the hierarchy to accept different numbers of parameters. The price to pay for this versatility is this: clients must use keyword arguments for passing all parameters to class constructors.

Suppose we want to combine the functionality of our nongraphical digital stopwatch (Listing 14.8 (digitalstopwatch.py)) with capability of a graphical object. The user should be able to click a location within the graphical window, and the digital stopwatch?s display will appear at that point. The user still should be able to start, stop, and reset the stopwatch as desired. We also would like to leverage existing code as much as possible. One way we can do this is via multiple inheritance.

The Box class is reminiscent of the Dot class from Listing 14.21 (dot.py). The Box class, however, is not a subclass of GraphicalObject. The constructor for Box does not use the expected keyword arguments, and it does not have a draw method. Clearly, the Box class does not cooperate with the classes in our GraphicalObject multiple inheritance hierarchy.

Every BoxAdapter object contains a Box object. The BoxAdapter methods delegate their work to the equivalent methods in Box. The constructor expects keyword arguments and calls super with keyword arguments, exactly as every good GraphicalObject subclass constructor should. The constructor unpacks the keyword arguments for Box?s constructor. The BoxAdapter class inherits run and do_click from GraphicalObject.

We created a rudimentary testing class in Section 13.7. Python provides a considerably more powerful testing framework in its unittest module. Unit testing is a well-established technique for evaluating the correctness of software components (see https://en.wikipedia.org/wiki/Unit testing). Unit testing involves testing individual units of a software system in isolation to determine if they behave as advertised. Examples of individual software units include functions, methods, objects, and modules.

Notice that the class TestFunctionsEtc inherits from unittest.TestCase. This means our TestFunctionsEtc class will have all the capabilities required to work properly in Python?s unit testing framework. All the methods within the TestFunctionsEtc class that begin with the pre?x test represent tests to perform. All the methods in the TestFunctionsEtc class happen to begin with test, but test classes in general can include other, non-test, methods as well. These can be helper methods that the test methods invoke.

tests the value of the ?rst argument of assertEqual to the second argument. The ?rst argument is the actual value, and the second argument is the expected value. The assertEqual method checks to see if they match. If the actual and expected value match, the test passes. If the two values do not match, the test fails. If attempting to evaluate the actual value produces an exception, the test is classi?ed as an error.

This line contains 15 characters?exactly the number of methods named with the pre?x test within our TestFunctionsEtc class. A dot (.) indicates a test that passed, an F represents a failed test, and an E indicates the test produced an exception (run-time error). An important point is that the test framework does not execute the tests in the same order that the test methods appear in the source code. The test execution order is alphabetical (actually lexicographic?the same ordering that the < operator imposes on Python string objects) by method name.

The unittest package includes a family of over 30 methods related to assertEqual. These methods all have the pre?x assert and include such tests as assertTrue, assertFalse, assertRaises, assertNotEqual, assertLess, assertLessEqual, assertListEqual, assertAlmostEqual, assertIs, assertIn, and assertNotIn.

In addition to the large variety of assert methods, the TestCase class provides the setUp and tearDown methods that we chose not to override in our TestFunctionsEtc class. The testing framework automatically will execute the setUp method before each test method runs and automatically execute the tearDown method after each test method executes (even if the test method raises an exception). To demonstrate how this works, see how the addition of these two methods affects the output of Listing 14.29 (testlist2.py):

Even though Python provides 68 standard exception classes, we may not ?nd one that exactly meets our needs, especially now that we can de?ne our own custom types via classes. Inheritance makes it easy to de?ne our own custom exception types. Listing 13.6 (stopwatch.py) de?nes a simple stopwatch timer class. Suppose we wish to consider an attempt to stop a nonrunning stopwatch an error worthy of an exception. We could reuse a standard exception, but which one? The following shows an attempt with ValueError:

(The remainder of help?s output provides information about ValueError?s methods.) Notice that help lists ValueError?s MRO (see Section 14.4), but, more importantly, it gives the meaning of the ValueError exception. ValueError is supposed to indicate an ?inappropriate argument value.? This meaning does not match well with an attempt to stop a stopped stopwatch.

Another problem is this: What if use our Stopwatch class to time code that can produce its own ValueError exception? How can any exception handling code we might write in this situation distinguish between an error with a Stopwatch object and a legitimate ValueError that the other code may raise?

The StopwatchException class inherits from Exception, but its empty class body means that StopwatchException adds nothing to what the Exception class already offers. The value that StopwatchException adds is this: it de?nes a new exception type that can participate as a ?rst-class citizen in Python?s exception handling infrastructure. We can rewrite the Stopwatch.stop method as

Chapter 15

Algorithm Quality

The previous chapters emphasized the mechanics of the Python programming language. We have seen how to manage variables, arithmetic, conditional execution, iteration, functions, parameters, objects, lists, tuples, dictionaries, sets, exceptions, custom types, and inheritance. Our main concern has been using these features to construct programs that correctly implement algorithms to solve problems. Sometimes correct algorithms are subtly dif?cult to get right, and their logic errors can evade even careful testing.

Program correctness always is the primary goal of software construction, but correctness is not the only goal. Two different programs may produce the exact same results in all cases, and yet one objectively may be considered better than the other. This difference in quality has nothing to do with source code style issues, variable names, or the code?s apparent complexity. The different is this: Despite the two programs producing the same results, one program effectively works and the other does not! It turns out that it is not hard to devise and implement an algorithm that correctly solves a problem but takes too much time to complete its work. The program, therefore, does not meet the user?s needs, as the user cannot wait long enough for the result.

This is_ascending function is correct. If the ?rst element is less than all the elements that follow the ?rst element, and the second element is less than all the elements that follow the second element, and the third element is less than all the elements that follow the third element, etc., all the elements must be arranged in non-decreasing order.

The is_ascending2 function uses the mathematical principle of transitivity. The transitive property of inequality for integers is this: If x ?y and y ?z, then x ?z. The is_ascending2 function compares the ?rst element to the second element, the second element to the third, the third to the fourth, etc., until it ?nally compares the next to the last element to the last element. If the function detects any element out of order, it returns False immediately. If it makes it all the way to the end of the list, the function returns True. Because of transitivity, if the loop gets to the end of the list, we know that the ?rst element is no larger than all the follow it; we need not compare the ?rst element directly with each and every element that follows it.

The is_ascending2 function is simpler than the is_ascending function because it uses a single loop rather than a nested loop. Apparent code complexity by itself is not a reliable criterion for judging the quality of an algorithm. We must determine which function computes its result quicker. Sometimes more complex code is faster than simpler code because the more complex code employs special tricks to speed up its processing.

? is ascending. In the case of is_ascending, the outer loop must iterate n?1 times. The inner loop scans n ?1 elements within the ?rst iteration of the outer loop. During each iteration of the outer loop the inner loops scans one fewer element than it did on the previous outer loop iteration. This means the inner loop iterates n ?1 times on the ?rst iteration of the outer loop, n ?2 times on the second iteration of the outer loop, n?3 times on the third iteration of the outer loop, etc. As a result the if statement executes

s = n(n?1)
2
= n2 ?n

2
= 25?5

is_ascending executes its if statement 10
4 = 2.5 times more than does is_ascending2, the comparison
within the if statement is simple and computers are fast, so we will not be able to detect the difference in
execution times.

function will execute its if statement 5002 ?500
2
= 250,000?500

comparison 124,750

function will execute its if statement 50002 ?5000

Listing 15.1 (ascendingplot.py) performs an experiment to test our mathematical analysis. It compares the performance of the two algorithms on lists of growing lengths. The list sizes consist of the the squares of the integers from zero to 200 in increments of 20; that is, $0^2 = 0, 20^2 = 400, 40^2 = 1600, 60^2 = 3600, \ldots, 200^2 = 40{,}000$. Besides printing the performance ?gures to the console, Listing 15.1 (ascendingplot.py) uses the Plotter object from Listing 13.14 (plotobj.py) to plot the performance curves.

The is_ascending2 function consistently outperforms is_ascending, but both functions execute in less than one second for lists of length 3,600 or less. This means if our application deals only with smaller lists, we may not notice the difference. As the list size grows, however, the performance difference grows dramatically. The is_ascending function requires over two minutes to process a list of size 40,000, while

times faster than is_ascending! The left window in Figure 15.1 illustrates the difference graphically. The Plotter object in Listing 15.1 (ascendingplot.py) plots the curves shown in the left window of Figure 15.1. The blue curve shows the growth of is_ascending?s execution time as the number of list elements grow from zero to 20,000. The red line shows the corresponding increase in is_ascending2?s execution time. Given the scale required to plot is_ascending?s curve, the curve for is_ascending2 barely deviates from the x axis.

Figure 15.1 The left window shows the graphical output of Listing 15.1 (ascendingplot.py). The right window shows the graphical output of Listing 15.2 (ascendingtheory.py). The left window plots of the experimental results comparing the performance of is ascending to is ascending2. The blue curve shows the growth of is ascending?s execution time as the number of list elements grow from zero to 20,000. The red line shows the corresponding increase in is ascending2?s execution time. Given the scale required to plot is ascending?s curve, the curve for is ascending2 barely deviates from the x axis. The right window plots the function n2 ?n does not call either Python function but rather plots the f(n) = n2 ?n

The absolute numbers the functions compute will be very different from the numbers that Listing 15.1 (ascendingplot.py) produced. This is because the functions that Listing 15.2 (ascendingtheory.py) plots represent a count of if statement executions while Listing 15.1 (ascendingplot.py) plots execution time in seconds. If our analysis is correct, however, the shape of the curves should be similar. The right window of Figure 15.1 shows the graphical output of Listing 15.2 (ascendingtheory.py). Note that shapes of the curves in the left and right windows match. This means the experimental results con?rm our earlier analysis. As the list size grows, the time difference between the two functions increases. While both is_ascending and is_ascending2 are correct algorithms, is_ascending2 is objectively better than is_ascending.

Examine again the curves in Figure 15.1 that correspond to the execution time of the is_ascending2 Python function (red curve in the left graph) and the f(n) = n?1 mathematical function that represents if statement execution counts (brown curve in the right graph). Both appear to be ?at, but this is due to the extreme large scale of the vertical axis. If both axes used the same scale, these curves would be lines rising

Figure 15.2 Result of plotting f(n) = n2 ?n

does take longer to execute as the list size grows. The function f(n) = n2 ?n

Note that both is_ascending and is_ascending2 scan as few elements as necessary to return a negative result. They both return False immediately upon detecting an element that is out of order. As an even more extreme example of a correct but bad algorithm, is_ascending3 always performs the same amount of work regardless of the list?s ordering:

The is_ascending3 function is just as correct as the is_ascending and is_ascending2 functions because once the is_ascending3 function sets its result variable to False it can never reset it back to True. Either it detects a reason to set result to False or it never changes it from its default value it assigned at the beginning. Given a list with its ?rst element larger than its second element, the is_ascending and is_ascending2 functions both will return False upon the ?rst execution of their if statement. The is_ascending3 function, however, will unnecessarily go through the whole list checking all the elements!

It seems computers are never fast enough or have enough memory to satisfy all our desires for software performance. Faster computers only serve to increase our expectations for applications that perform more complex tasks on larger data sets. As we have seen in the simple example above, the choice of algorithm can make a dramatic difference in the performance of a task. A correct algorithm can be so bad that even the fastest computer cannot enable it to solve a particular problem in an acceptable amount of time. A different algorithm, however, may be able to solve the same problem quickly on even a slow machine.

Lists, introduced in Chapter 10, are convenient structures for storing large amounts of data. Sorting? arranging the elements within a list into a particular order?is a common activity. For example, a list of integers may be arranged in ascending order (that is, from smallest to largest). A list of strings may be arranged in lexicographical (commonly called alphabetical) order. Many sorting algorithms exist, and some perform much better than others. We will consider one sorting algorithm that is relatively easy to describe and implement.

Python lists have a sort method (Section 10.10) that orders the elements in a list. Similar to the reversed function (Section 10.2), the __builtins__ module provides a sorted function the returns an iterable object. If the elements of a list can be ordered, we can use sorted to visit its elements in sorted order. The following code:

Why implement a sorting algorithm when Python provides the standard sorted function and list.sort method? Writing a sort function gives us an opportunity to exercise our problem solving skills. It also gives us more practice using and manipulating lists. The experience we gain though the process better equips us to tackle problems we may encounter that have no solution in the standard library.

This initially does not seem to buy us much?it appears only to make the code a bit more obscure. Notice that to change the way the if statement compares we need to change the name of the function. If we have a greater_than function, for example, we could use it in the place of less_than. Admittedly, changing a function name generally requires more typing than changing a single symbol (< to >); however, we will see that it gives us the ability to build a sort function that can order its elements in many different ways.

The comparison function passed to the sort routine customizes the sort?s behavior. The basic structure of the sorting algorithm does not change, but its notion of ordering is adjustable. If the second parameter to selection_sort is less_than, the function arranges the elements ascending order. If the second parameter instead is greater_than, the function sorts the list in descending order. More creative orderings are possible with more elaborate comparison functions.

The key function in Listing 15.5 (linearsearch.py) is locate; all the other functions simply lead to a more interesting display of locate?s results. If locate ?nds a match, the function immediately returns the position of the matching element; otherwise, if after examining all the elements of the list locate cannot ?nd the element sought, the function returns None. Here None indicates the function could not return a valid answer. The calling code, in this example the display function, must ensure that locate?s result is not None before attempting to use the result as an index into a list.

```
lst = [100, 44, 2, 80, 5, 13, 11, 2, 110]
x = locate(lst, 13)
```

5

lst

5

ensure that first is less than or equal to last for a nonempty list. If the list is empty, first is zero, and last is equal to len(lst) - 1 = 0?1 = ?1. So in the case of an empty list the function will skip the loop and return None. This is correct behavior because an empty list cannot possibly contain any item we seek.

? The elif and else blocks ensure that either last decreases or first increases each time through the loop. Thus, if the loop does not terminate for other reasons, eventually first will be larger than last, and the loop will terminate. If the loop terminates for this reason, the function returns None. This is the correct behavior.

```
lst = [10, 14, 20, 28, 29, 33, 34, 45, 48]
x = locate(lst, 33)
```

29

lst

Suppose a list to search contains n elements. In the worst case?looking for an element larger than any currently in the list?the loop in linear search takes n iterations. In the best case?looking for an element smaller than any currently in the list?the function immediately returns without considering any other elements. The number of loop iterations thus ranges from 1 to n, and so on average linear search requires n 2 comparisons before the loop ?nishes and the function returns.

4, then n

8, and so forth. The problem of determining how many times a set of things can be divided in half until only one element remains can be solved with a base-2 logarithm. For binary search, the worst case scenario of not ?nding the sought element requires the loop to make log2 n iterations.

In our situation, both search algorithms process the list with only a few extra local variables, so for large lists they both require essentially the same space. The big difference here is speed. Binary search performs more elaborate computations each time through the loop, and each operation takes time, so perhaps binary search is slower. Linear search is simpler (fewer operations through the loop), but perhaps its loop executes many more times than the loop in binary search, so overall it is slower.

The program assigns each of these lists in turn to the test_list variable. The program also builds lists of random integer values (referenced via seek_list) to be used as search candidates for each test_list. It then passes these lists off to the test_searches function to measure the running times of the two search functions. The test_searches function, in turn, calls the run_search function to test a particular search function. The run_search function uses the elements from main?s seek_list as search candidates. The run_search function searches for all the elements in seek_list a speci?ed number of times and averages the running times. The main function directs test_searches to average 10 runs for each of the list sizes.

In addition to empirical observations, we can judge which algorithm is better by analyzing the source code for each function. Each arithmetic operation, assignment, logical comparison, function call, and list access requires time to execute. We will assume each of these activities requires one unit of processor ?time.? This assumption is not strictly true, but it will give good results for relative comparisons. Since we will follow the same rules when analyzing both search algorithms, the relative results for comparison will be close enough for our purposes.

2 times. The function returns either i or

None, and it may excute at most one return statement during each call. Table 15.1 shows the breakdown for linear search. The results in Table 15.1 indicate the running time of the linear_search function can be expressed as a simple mathematical linear function: f(n) = 3n + 4.

Next, we consider binary search. We determined that in the worst case the loop in binary_search iterates log2 n times if the list contains n elements. The binary_search function performs the two initializations before the loop just once per call. Most of the actions within the loop occur log2 n times, except that only one return statement can be executed per call, and in the if/elif/else statement only one path can be chosen per loop iteration. Table 15.2 shows the complete analysis of binary search. We see that the execution time for binary search can be expressed as the logarithmic function 12log2 n+6.

2 its actual cost.

Figure 15.5 compares the empirical results with the analytical results for lists containing 100 to 1,000 elements. The left side of Figure 15.5 plots the values produced by Listing 15.7 (searchcompare.py), and the right side of Figure 15.5 plots the two functions 3n + 4 and 12log2 n + 6. In these two graphs we can compare the growth rates of the two search techniques by examining the shapes of the curves. Notice how closely the two graphs compare to each other. In both graphs the gap between the linear search curve and binary search curve increasingly widens at the same rate as the list size incrases. The binary search curve appears to be effectively ?at, although it really is growing very slowly, much more slowly than the linear search curve. The bottom line is that binary search is fast even for large lists.

```
lst1 = [21, 19, 31, 22, 14, 31, 22, 6, 31]
print(count(lst1, 31))
lst2 = ['FRED', [2, 3], 44, 'WILMA', 'FRED', 8, 'BARNEY']
print(count(lst2, 'FRED'))
print(count(lst2, 'BETTY'))
print(count([], 16))
```

count([21, 19, 31, 14, 31, 6, 31], 31) = count([19, 31, 14, 31, 6, 31], 31)
= count([31, 14, 31, 6, 31], 31)
= 1 + count([14, 31, 6, 31], 31)
= 1 + count([31, 6, 31], 31)
= 1 + 1 + count([6, 31], 31)
= 1 + 1 + count([31], 31)
= 1 + 1 + 1 + count([], 31)
= 1 + 1 + 1 + 0
= 1 + 1 + 1
= 1 + 2
= 3

While the count function in Listing 15.8 (recursivecount.py) works properly, it has one, potentially big disadvantage. Each time the function?s execution selects the recursive route it slices the list. Slicing a list creates a copy of the list (see Section 10.7). This means every call to count in the recursive call chain makes a complete copy of the list, except for the ?rst element of each successive list. If the list is long, this can unnecessarily consume a large amount of the computer?s memory. How big can it get? Consider an initial call to count that passes a list of 1,000 elements. The ?rst recursive call passes a new list of 999 elements.

Listing 15.9 (inplacecount.py) uses two functions to do the counting. Its count function merely calls count_helper with the proper initial parameters. The count_helper function does all the interesting work. Instead of creating copies of the list, count_helper accepts an additional parameter, an index, for the recursion to keep track of its position within the list. The list parameter is an alias of the original list, not a copy. When a function can process a list without making a copy, we say the function processes the list in place.

This processes the list in place and does not use recursion. This version of count actually is superior to both recursive versions. As we saw in Section 8.3, every function call requires a little extra time and memory. If two functions?one iterative and one recursive?faithfully implement the same algorithm, the iterative version will be more ef?cient.

A recursive function does have one distinct advantage over a non-recursive function, though. A recursive function does not just call itself; the self-call eventually returns back to the site of its invocation. Each recursive invocation has its own parameters and creates its own local variables. When the function returns to itself, it ?remembers? its original parameters and local variables the way they were before the recursive invocation. We say the function unwinds back to its previous state.

Observe that the unwinding of the recursive calls restores the original values of the parameter n and local variable rand. Since rand is assigned pseudo-probablistically, it is not possible to restore its original value without ?rst storing it somewhere for later retrieval. The function-call-and-return process automatically takes care of saving and restoring the previous values of local variables.

The output of Listing 15.12 (nonrecursivememory.py) is identical to the output of Listing 15.11 (recursivememory.py). Listing 15.12 (nonrecursivememory.py) uses two separate, sequential loops and an accessory list to simulate the recursive behavior of Listing 15.11 (recursivememory.py). The purpose of the local history list is to remember the current state of the variables n, depth, and rand so the executing program can restore their values after the simulated recursion ?returns.? The extra space for the history list and extra time spent managing the history list is comparable to the space and time the recursive function requires. It is much easier to allow the magic of recursion to automatically take care of saving and restoring the function?s local variables and parameters.

Sometimes it is useful to consider all the possible arrangements of the elements within a list. A sorting algorithm, for example, must work correctly on any initial arrangement of elements in a list. To test a sort function, a programmer could check to see to see if it produces the correct result for all arrangements of a relatively small list. We saw in Section 5.4 that an arrangement of a sequence of ordered items is called a permutation. Listing 5.20 (permuteabc.py) prints all the permutations of the sequence ABC. We need something more ?exible: a function that generates all the possible permutations of any list. The function will accept a list as a parameter and return a list containing all the permutations of the parameter. (Note that the return value is a list of lists.) Listing 15.13 (listpermutations.py) contains functions that build a new list containing all the permutations of a given list.

The perm function in Listing 15.13 (listpermutations.py) is a recursive function, as it calls itself inside of its de?nition. We have seen how recursion can be an alternative to iteration; however, the perm function here uses both iteration and recursion together to generate all the arrangements of a list. At ?rst glance, the combination of these two algorithm design techniques as used here may be dif?cult to follow, but we actually can understand the process better if we ignore some of the details of the code.

```
print(0)
print(1)
print(2)
print(3)
print(4)
```

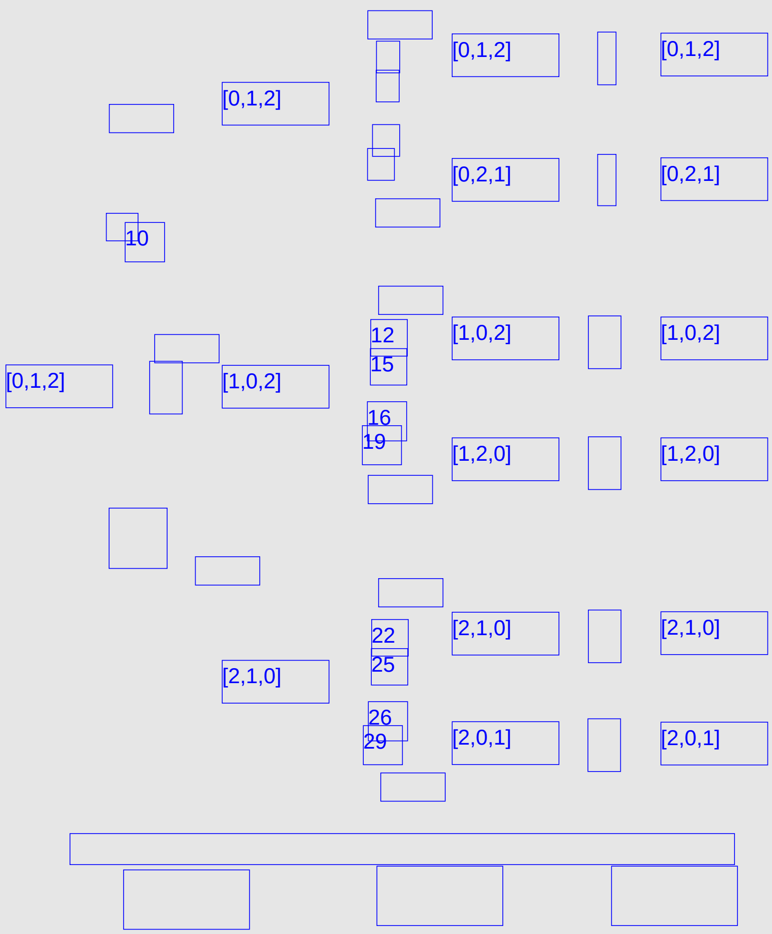
to make the code?s execution faster. After unrolling the loop, the loop control variable (in this case i) is gone, so there is no need to initialize i (done once) and, more importantly, no need to check and update i during each iteration of the loop. Eliminating these tasks, especially the check and update within the loop, reduces the work the program must do, thereby speeding up its execution.

Once the loop is gone, we see we have simply a series of recursive calls of perm sandwiched by element swaps. The ?rst swap interchanges an element in the list with the ?rst element. The second swap reverses the effects of the ?rst swap. This series of swap-call perm-swap back operations allows each element in the list to have its turn being the ?rst element in the permuted list. The perm recursive call generates all the permutations of the rest of the list. Figure 15.6 traces the recursive process of generating all the permutations of the list [0,1,2]. The leftmost third of Figure 15.6 shows the original contents of the list and the initial call of perm. The three branches represent the three iterations of the for loop: i varying from begin (0) to the last index (2). The lists indicate the state of the list after the ?rst swap but before the recursive call to perm.

The middle third of Figure 15.6 shows the state of the list during the ?rst recursive call to perm. The two branches represent the two iterations of the for loop: i varying from begin (1) to the last index (2). The lists indicate the state of the list after the ?rst swap but before the next recursive call to perm. At this level of recursion the element at index zero is ?xed, and the remainder of the processing during this chain of recursion is restricted to indices greater than zero.

The rightmost third of Figure 15.6 shows the state of the list during the second recursive call to perm. At this level of recursion the elements at indices zero and one are ?xed, and the remainder of the processing during this chain of recursion is restricted to indices greater than one. This leaves the element at index two, but this represents the base case of the recursion because begin (2) equals the index of the last element (2). In this case the function makes no more recursive calls to itself. The function merely adds a copy of the current list to the list of permutations.

Figure 15.6 A tree mapping out the recursive process of the perm function operating on the list [0, 1, 2]. The second column from the left shows the original contents of the list after the ?rst swap but before the ?rst recursive call to perm. The swapped elements appear in red. The third column shows the contents of the list at the second level of recursion. In the third column the elements at index zero are ?xed, as this recursion level is using begin with a value of one instead of zero. The for loop within this recursive call swaps the elements highlighted in red. The rightmost column is the point where begin equals the index of the last element, and so the perm function does not call itself, effectively terminating the recursion. The numbers on the arrows trace the order in which the program makes the calls to, and returns from, the perm function.



We can augment the perm function to better illustrate the iterative and recursive processes. With a technique known as code instrumentation, we will add statements that provide insight into the algorithm?s progression. The term instrumentation mirrors its meaning outside the realm of programming. A motor vehicle, for example, has an instrument panel containing several different instruments. The speedometer indicates the vehicle?s current speed, and the tachometer provides the vehicle?s engine?s RPMs. Neither of these devices is absolutely essential for driving the vehicle, but they do give the driver more precise information about the state of the driving experience.

(1 zettabyte = 1 billion terabytes.) 387,780 zettabytes is about 140,000 times greater than 2.7 zettabytes, the estimated total data storage space in all media (hard drives, solid state drives, CDs, DVDs, tape, etc.) found on the planet (see http://en.wikipedia.org/wiki/Zettabyte). It is safe to assume that your laptop or desktop would not have enough RAM to hold the list of all permutations. Besides, if your program could generate one permutation each nanosecond (an unreasonably fast rate even with today?s fastest processors), the program would require

Listing 15.13 (listpermutations.py) is impractical for all but relatively small lists because the perm function does not return until after building the list containing all the permutations. The basic algorithm is sound, however, and fortunately we can salvage it nicely using generators. (We ?rst explored generators in Section 8.8.) Instead of producing the entire list of permutations, our function will yield each permutation one at a time.

We covered the base case perfectly, what happens in the recursive case? Recursive generators are a little different from the iterative generators we saw earlier. Since the if block contains a yield statement, the else block needs one as well. What we want to yield is what the recursive call to perm eventually yields when it reaches its base case. When we need to yield a value from a recursive call we must use the yield from statement. The yield from statement indicates the generator should yield the result that the chain of recursive calls ultimately yields when it reaches its terminal base case. Listing 15.15 (generatepermutations.py) shows how to use yield and yield from in a recursive generator.

The permutations function must yield the result of the call to perm, so the yield from statement appears there as well. This is because the permutations function itself does not create the value; instead, permutations relies on the perm function to create the value. A function that relies on another function to create the yielded value must use yield from. Note that this is is consistent with the way yield from works with recursion.

This does not help the time it takes to produce all the permutations; however, the generator perm function ?returns? a permutation immediately, thus avoiding the problems with the original version that tried to make all the permutations before returning. This means the caller can get into and out of the function quickly. While the program still would require centuries to complete its execution if asked to print all the permutations of a list with 25 elements, it could print the ?rst 100 permutations very quickly:

We have seen that generating all the permutations of a large list is computationally intractable. Often, however, we merely need to produce one permutation chosen at random. For example, we may need to randomly rearrange the contents of an ordered list so that we can test a sort function to see if it will produce the original list. We could generate all the permutations, put each one in a list, and select a permutation at random from that list. This approach is inef?cient, especially as the length of the list to permute grows larger. Fortunately, we can randomly permute the contents of a list easily and quickly. Listing 15.18 (randompermute.py) contains a function named permute that randomly permutes the elements of a list.

Notice that the permute function in Listing 15.18 (randompermute.py) uses a simple un-nested loop and no recursion. The permute function varies the i index variable from 0 to the index of the next to last element in the list. The function pseudorandomly chooses an index greater than i using randrange (see Section 6.6), and the function then exchanges the elements at position i and the random position. At this point all the elements at position i and smaller are ?xed and will not change as the function?s execution continues. The function then increments index i for the next iteration and continues its work until it has considered all acceptable values for i.

To be correct, our permute function must be able to generate any valid permutation of the list. It is important also that our permute function is able produce all possible permutations with equal probability; said another way, we do not want our permute function to generate some permutations more often than others. The permute function in Listing 15.18 (randompermute.py) is ?ne, but consider a slight variation of the algorithm:

Do you see the difference between faulty_permute and permute? In faulty_permute, the random index is chosen from all valid list indices, whereas permute restricts the random index to valid indices greater than or equal to i. This means that any element within lst can be exchanged with the element at position i during any loop iteration. While this approach may super?cially appear to be just as good as permute, it in fact produces an uneven distribution of permutations. Listing 15.19 (comparepermutations.py) exercises each permutation function 1,000,000 times on the list [1, 2, 3] and tallies each permutation. There are exactly six possible permutations of this three-element list.

In one million runs, the permute function provides a fairly even distribution of the six possible permutations of [1, 2, 3]. The distributions are not all exactly the same, but we would expect minor differences due to the variations that randomness introduces. On the other hand, the faulty_permute function generates the permutations [1, 3, 2], [2, 1, 3], and [2, 3, 1] more often than the permutations [1, 2, 3], [3, 1, 2], and [3, 2, 1].

123

123 213 321

list represents the element at index zero swapped with the element at index zero (effectively no change). The second list on the second row represents the interchange of the elements at index 0 and index 1. The third list on the second row results from the interchange of the elements at positions 0 and 2. The underlined elements represent the elements most recently swapped. If only one item in the list is underlined, the function merely swapped the item with itself. The bottom row contains all the possible outcomes of the faulty_permute function given the list [1, 2, 3].

others appear 5
27 = 18.519% of the time. When generating one million permutations we would get 14.815% · 1,000,000 = 148150, and 18.519 · 1,000,000 = 185190. Notice that these numbers agree with our experimental results from Listing 15.19 (comparepermutations.py).

Compare Figure 15.8 to Figure 15.9. The second row of the tree for permute is identical to the second row of the tree for faulty_permute, but the third rows are different. The second time through its loop the permute function does not attempt to exchange the element at index zero with any other elements. We see that none of the ?rst elements in the lists in row three are underlined. The third row contains exactly one instance of each of the possible permutations of [1, 2, 3]. This means that the correct permute function is not biased towards any of the individual permutations, and so the function can generate all the permutations with equal probability. The permute function has a $\frac{1}{6} = 16.667\%$ probability of generating a particular permutation. Over 1,000,000 trials, we would expect each permutation to occur 166667 times. This number agrees with our the experimental results of Listing 15.19 (comparepermutations.py).

Python has a standard function, reversed, that accepts a list parameter. The reversed function does not return a list but instead returns an iterable object that works like a generator or range within a for loop (see Section 5.3). Listing 15.21 (reversedexample.py) shows how reversed can be used to print the contents of a list backwards.

We know a program can use variables to remember values as it executes. A programmer must be able to predict the number of values the program must manage in order to write enough variables in the code. A dictionary provides an opportunity to create an arbitrary amount of new storage during a program?s execution. We will consider a simple problem that demonstrates the value of the dynamic storage provided by dictionaries.

Note the results that Listing 15.22 (?bonacciinstrumented.py) prints agree exactly with the call count shown in Figure 15.10. As we compute larger Fibonacci numbers, the amount of repeated work worsens quickly; for example, the call fibonacci(20) recursively calls fibonacci(1) 6,765 times! For additional emphasis, the call fibonacci(35) recursively calls fibonacci(1), fibonacci(2), fibonacci(3), fibonacci(4), and fibonacci(5) over one million times each! We may be tempted to care less about the program?s repeated work?after all, it is the computer doing the work, not us. Unfortunately, the computer, even though it is very fast, requires some amount of time to perform any task. As we multiply the number of tasks a program must do to solve a problem, the time to compute the solution increases, and, in the case of factorial, the time increases dramatically.

We can improve the performance of our fibonacci function using a technique known as memoization (not to be confused with the word memorization which means to commit something to memory). Memoization is an algorithm design technique that records the result of a speci?c computation so that result can be used as needed at a later time during the algorithm?s execution. It is as if the executing program ?makes a note to itself? or ?stores the result in a memo.? When the program needs the result of an identical computation in the future, if simply reads the memo with the answer it stored earlier. In this way the program avoids repeating the work. Memoization is especially useful for problems that consist of subproblems that overlap and appear to require multiple computations with identical input.

Figure 15.11 shows the recursion tree for our memoized Fibonacci function computing the ?fth Fibonacci number. It assumes the fibonacci2 has not previously been called, and so at the initial call at the top of the tree the global ans dictionary contains only the 0 and 1 keys. The ?gure shows only nine invocations of fibonacci2, compared to 15 invocations of the non-memoized fibonacci function. If you add the function call counting instrumentation used in Listing 15.22 (?bonacciinstrumented.py) to fibonacci2, you will ?nd the numbers it reports agrees with Figure 15.11.

Each call to fibonacci3 starts with a fresh ans dictionary containing only the base case keys 0 and 1. It calls a local helper function, fib, which performs the recursion. The fib function avoids redundant computations using the ans dictionary to which it has access. If a program needs to compute only one Fibonacci number in the course of its execution, fibonacci3 will be just as fast as fibonacci2 when computing the same number. If, however, a program needs to compute Fibonacci numbers multiple times, fibonacci2 may be the better choice. The global dictionary that fibonacci2 uses maintains its contents between function calls, and so it is more likely to have the desired result memoized from an earlier invocation.

Figure 15.11 The hierarchy of recursive function calls that result from the call fibonacci2(5). Note that the calls of fibonacci(2) and fibonacci(3) on the right side of the tree need no recursive calls; this is because their earlier calls on the left side of the tree stored their results in the global ans dictionary for instant retrieval.

B

A

Use value cached at A

Use value cached at B

One disadvantage of the global ans dictionary is that other functions within the program may accidentally or maliciously manipulate its contents causing fibonacci2?s results to be unreliable. To help avoid this problem you can put fibonacci2 and ans into a separate module. Rename ans to _ans to make a point to callers that _ans is off limits to their code.

Another application of memoization is ?nding the longest common subsequence of two sequences. A sequence is simply an ordered list of elements, as in a Python list, tuple, or string. Most typically the elements of a sequence all will have the same type. A subsequence is a sequence formed from another sequence by removing elements without changing the relative order of the remaining elements. As an example, consider the sequence of characters in the Python list ["A", "B", "C", "D"]. The complete set of subsequences is

```
[]
["A"]
["B"]
["A", "B"]
["C"]
["A", "C"]
["B", "C"]
["A", "B", "C"]
["D"]
["A", "D"]
["B", "D"]
["A", "B", "D"]
["C", "D"]
["A", "C", "D"]
["B", "C", "D"]
["A", "B", "C", "D"]
```

The longest common subsequence (LCS) problem is this: given two sequences of symbols, ?nd the longest subsequence that is common to both sequences. This problem is related to genome sequencing problems in computational biology. The LCS problem has a relatively simple recursive solution. We will use a Python string to represent a sequence, where each character is a symbol in the sequence. With this representation, the string "ACD" is a subsequence of "ABCD". The following function computes the LCS of two strings:

The LCS function is correct, but its recursive invocations suffer a similar fate to our original fibonacci function?they unnecessarily recompute multiple times the LCS of identical string pairs. As an example, the call LCS("ABCBDAB", "BDCABA") will recompute recursively the LCS of the subsequences "BDAB" and "ABA" three times. These multiple, super?uous recomputations render this LCS function impractical for all but relatively short sequences.

Note that the memoization dictionary is local to LCS_memoized. Each time a caller invokes LCS_memoized the memoization dictionary begins empty. The memoization dictionary is valid only for the two speci?c strings that the caller passes. It would be unwise to make the memoization dictionary global; otherwise, if an application ran a long time repeatedly calling LCS_memoized with many different strings, a global dictionary would grow continuously. This unbounded growth potentially would exhaust the memory available to the

Figure 15.12 A graph produced by one run of Listing 15.24 (lcsmemo.py). The horizontal axis represents the length of the strings, increasing to the right. The vertical axis represents the program?s running time. The blue line plots the accumulated time consumed by the non-memoized function, and the red line plots the accumulated time of the memoized version of the function. Note that difference in execution times increases dramatically as strings become longer, and the memoized version continues to perform well even with longer strings.

2. Complete the following function that reorders the contents of a list of integers so that all the even numbers appear before any odd number. The even values are sorted in ascending order with respect to themselves, and the odd numbers that follow are also sorted in ascending order with respect to themselves. For example, a list containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 would be transformed into 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 Note that your function must physically rearrange the elements within the list, not just print the elements in the desired order.

5. Complete the following function that shifts all the elements of a list backward one place. The last element that gets shifted off the back end of the list is copied into the ?rst (0th) position. For example, if a list containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 is passed to the function, it would be transformed into 5, 2, 1, 10, 4, 3, 6, 7, 9, 8 Note that your function must physically rearrange the elements within the list, not just print the elements in the shifted order.

6. Complete the following function that determines if the number of even and odd values in an integer list is the same. The function would return true if the list contains 5, 1, 0, 2 (two evens and two odds), but it would return false for the list containing 5, 1, 0, 2, 11 (too many odds). The function should return true if the list is empty, since an empty list contains the same number of evens and odds (0 for both). The function does not affect the contents of the list.

11. Complete the following function that determines if two lists contain the same elements, but not necessarily in the same order. The function would return true if the ?rst list contains 5, 1, 0, 2 and the second list contains 0, 5, 2, 1. The function would return false if one list contains elements the other does not or if the number of elements differ. This function could be used to determine if one list is a permutation of another list. The function does not affect the contents of either list.

Chapter 16

Representing Relationships with Graphs

The circles and lines make up a mathematical graph, or simply graph. A graph represents relationships among things. A circle on the graph is known as a vertex (plural: vertices), or node. A line connecting two vertices is called an edge, or arc. Two vertices connected directly by an edge are adjacent; two vertices not connected directly by an edge are non-adjacent. Two adjacent vertices are related to each other in some way; for example, in the airline routes graph two airports are related if a direct ?ight between them exists.

In a directed graph (or digraph for short) the edges have a direction; that is, vertex a may be adjacent to vertex b with vertex b not being adjacent to vertex a. Figure 16.4 visualizes a directed graph. In the ?gure the arrow pointing from vertex d to vertex f indicates that d is related to f. This means f is adjacent to d, but observe that d is not adjacent to f. Since edges connect c and d both ways, c is adjacent to d, and d is adjacent to c.

Figure 16.1 A map showing the routes between cities for an airline. The circles in the ?gure represent airports, and the associated labels are airport code names (for example, ATL is Harts?eld-Jackson Atlanta International Airport near Atlanta, Georgia). A line connecting two circles indicates a direct nonstop ?ight between two airports. (The image of the United States map is adapted from https://commons.wikimedia.org/wiki/File:Blank US Map.svg.)

b d

a e

? Degree requirements. A university degree for a particular major requires a number of courses. Some of the courses may have prerequisite courses; for example, typically a student wishing to enroll in a second, more advanced computer programming course must ?rst complete an introductory computer programming course. In a prerequisite graph, every course is a vertex, and a directed edge connects a course to its prerequisite course(s).

```
"ATL": {"MIA", "DCA", "ORD", "MCI", "DFW", "DEN"},
"MIA": {"LGA", "DCA", "ATL", "DFW"},
"DFW": {"LAX", "DEN", "MCI", "ORD", "ATL", "MIA"},
"LAX": {"SFO", "DEN", "MCI", "DFW"},
"DEN": {"SFO", "LAX", "MCI", "DFW", "SEA", "ATL"},
"SEA": {"SFO", "DEN", "ORD", "LGA"},
"MCI": {"DEN", "LAX", "DFW", "ATL", "ORD", "LGA"},
"ORD": {"SEA", "MCI", "DFW", "ATL", "DCA", "LGA"},
"DCA": {"ORD", "ATL", "MIA", "LGA"},
"LGA": {"SEA", "MCI", "ORD", "DCA", "MIA"},
"SFO": {"SEA", "DEN", "LAX"}
}
```

Since a Python set holds the collection of adjacent airports, the statement above most likely will print the airports out in a different order than they appear in the source code assigning routes. If in a particular application the ordering of adjacent vertices is important, use a Python list instead of a set. For most problems, however, a set is the preferred adjacency structure. Counterintuitively, the formal name for this kind of graph representation is adjacency list (see https://en.wikipedia.org/wiki/Adjacency list), where list as used here means an unordered collection, or set. Python reserves the name list for an ordered linear collection.

Computer scientists and mathematicians have developed a number of algorithms that solve common problems in graph theory. One problem is ?nding a path from one vertex to another within a graph. From our airline example graph in Figure 16.3, ATL?DEN?SFO?SEA represents a path from ATL to SEA. Another path from ATL to SEA is ATL?MIA?DFW?MCI?LGA?ORD?SEA. Since no direct ?ight exists between ATL and SFO, the sequence ATL?SFO?SEA is not a path in the Figure 16.3 graph.

Generally, shorter paths are desirable, as a shorter path usually translates into some kind of savings, such as shorter travel time or less fuel consumed. Some graphs have values attached to each edge that represent a cost such as time or distance, making some edges more expensive than others. We will keep things simple here and treat all edges equally. This means one path is shorter than another path if it visits fewer vertices. The length of a path is the number of edges used in the path, so the path length of ATL?MIA?DFW?MCI?LGA?ORD?SEA is 6. So, while ATL?MIA?DFW?MCI?LGA?ORD?SEA surely gets a passenger from from ATL to SEA, the path ATL?DEN?SFO?SEA is shorter. (Can you ?nd an even shorter path from ATL to SEA in Figure 16.3?)

In order to ?nd a shortest path in a graph, the algorithm must traverse the graph. Graph traversal is more complicated than list traversal. Traversing a list is easy: begin at the ?rst element (index 0) and visit each successive element (element at current index plus one), in order, until locating the desired element or running out of elements to visit. A graph, however, is not a linear structure; it is a network of interconnected elements, and an element (vertex) can have multiple successors (zero or more adjacent vertices). If our traversal algorithm is not careful, it could, for example, begin at vertex ATL and follow the path ATL?DCA?LGA?ORD?ATL. Such a path that begins and ends at the same vertex is known as a cycle. A na¨?ve algorithm could continue the traversal following the exact same path as shown here:

To avoid getting stuck in a cycle our traversal algorithm must have a way to remember which vertices it has visited so it does not attempt to revisit them. Our path ?nding algorithm will add a visited vertex to a dictionary. The vertex serves as a key, and its associated value will be the vertex that immediately precedes it on a shortest path from the starting vertex to the destination vertex. During its traversal of the graph, the algorithm will check this dictionary before attempting to visit a vertex. If the vertex is not in the dictionary, the algorithm ?visits? the vertex by adding it (with its predecessor on a shortest path) to the dictionary. If the vertex is already present in the dictionary, the algorithm ignores the vertex and continues its traversal through other available vertices in the graph.

The second problem for our path ?nding algorithm is not merely to ?nd a path from the starting vertex to the ending vertex but to ?nd a shortest path. Remember that an algorithm cannot see the graph globally as we can when we look at its visual representation. The algorithm must be clever in its choice of which vertex to visit next as it traverses the graph. This choice is crucial because once the algorithm adds a vertex to the dictionary of visited vertices it can never go back and revisit the vertex to determine if choosing a different adjacent vertex would result in a shorter path. (As noted above, if it attempts to revisit a vertex, the algorithm could get stuck in a cycle, retracing the same path over and over again.)

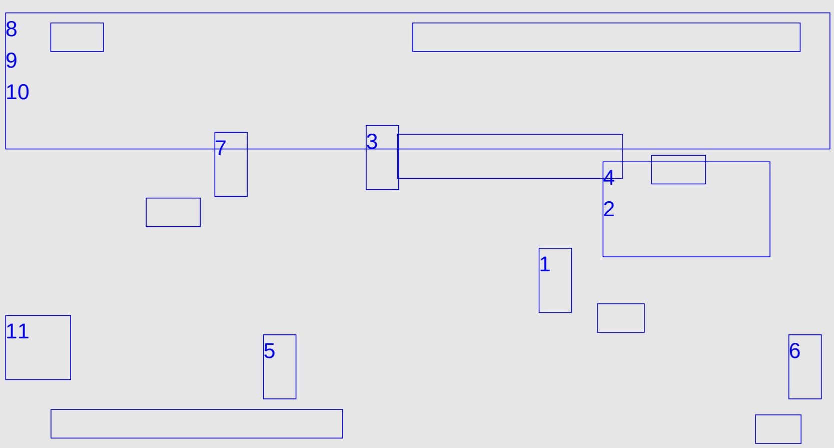
There are a number path ?nding algorithms, and Listing 16.1 (airlineroute.py) uses one such algorithm, breadth-?rst search (BFS), to compute a shortest path from one vertex to another (see https://en.wikipedia.org/wiki/Breadth-first search). The BFS algorithm uses a queue to guide the algorithm?s progress. A queue is a ?rst-come, ?rst-serve data structure. It is similar to a list, in that a queue is a linear sequence of elements, and clients can add elements to, and remove elements from, the sequence. Unlike with a list, however, element addition and removal in a queue is restricted: clients may remove only the most recent element added to the queue. A queue thus works like a line of customers waiting at a checkout counter in a store. The cashier serves the person at the front of the line ?rst, and customers who are ready to check out but not in line must join the end of the line and wait for their turn. Conceptually, we can place items on the back of a queue only, and we can remove items only from the front of a queue.

Inputs: A starting vertex and destination vertex within a graph.
Begin with an empty dictionary and an empty queue.
Add the starting vertex to the queue.
while the queue is not empty and the dictionary does not contain the destination vertex as a key:
Serve the next vertex, v, from the queue.
for each vertex w adjacent to v:
if w is not in the dictionary:
Add w to the queue.
Add key w with its value v to the dictionary.
Output: The dictionary with the keys consisting vertices reachable from the starting vertex;
the value of a key is the vertex that immediately precedes it on a shortest path from the starting vertex
to the destination vertex.

The ?rst time through its while loop the BFS algorithm extracts the starting vertex from the queue. The for loop then considers every vertex adjacent to the starting vertex. These adjacent vertices go into the queue. Note that all of these vertices added to the queue are a distance of 1 away from the starting vertex and record the starting vertex as their predecessor. On the second iteration of its while loop the algorithm removes one vertex from the queue. This has to be one of the vertices it added on its ?rst iteration?a vertex adjacent to the starting vertex. Call this vertex v. The algorithm adds any nonvisited vertices adjacent to v to the queue. Note that all these nonvisited vertices adjacent to v are a distance of 2 from the starting vertex. Because it is using a queue the algorithm cannot visit these newly added vertices until it processes the vertices added to the queue earlier. This means the algorithm visits all the vertices that are a distance of 1 from the starting vertex before it attempts to visit any vertices at a distance of 2 away from the starting vertex. Unless it encounters the destination vertex beforehand, the algorithm eventually will extract from the queue a vertex at a distance of 2 from the starting vertex, adding all of that vertex?s unvisited vertices (distance 3) to the back of the queue. Consequently, all distance 2 vertices will be ahead of distance 3 vertices in the queue, and so the algorithm visits all distance 2 vertices before visiting any distance 3 vertex. The while loop continues until the queue is empty or the algorithm reaches the destination vertex. In this way the algorithm visits all distance 1 vertices before any distance 2 vertex, all distance 2 vertices before any distance 3 vertex, all distance 3 vertices before any distance 4 vertex, and so forth. Figure 16.5 illustrates with an example BFS traveral. The queue ensures that the next vertex visited in the while loop is no farther away from the starting vertex than any other yet-to-be visited vertex. Expanding the search to a more distant vertex only after all nearer vertices have been considered is the trick to discovering a shortest path in the graph from the starting vertex to the destination vertex.

Inputs: A starting vertex and destination vertex within a graph.
Begin with an empty path.
Use the BFS algorithm to compute the dictionary containing vertex predecessors on
a shortest path from the starting vertex to the destination vertex.
if the destination vertex is in the dictionary:
Add the destination vertex to the path.
Set v to be the destination vertex.

Figure 16.5 The numbers indicate a possible order in which BFS visits vertices starting at vertex ATL. Since a Python program uses a dictionary to represent the graph, the exact order of the visitation is unpredictable, even from one execution to the next on the same machine. What is predictable, however, is that BFS will visit all vertices at a distance of n from the starting vertex before it visits any vertex at a distance greater than or equal to n + 1 from the starting vertex. For example, the shortest distance between ATL and DEN is 1 and the shortest distance between ATL and LAX is 2; this means the number associated with DEN must be less than the number associated with LAX. Consequently, BFS from ATL necessarily would visit DEN before visiting LAX.



while the starting vertex is not in the path:
Update v to be v?s predecessor from the dictionary.
Add v to the front of the path.
Output:The path that contains, in order, the vertices found on a shortest path from the starting vertex to the destination vertex; an empty path indicates that the destination vertex is not reachable from the starting vertex.

Listing 16.1 (airlineroute.py) implements the BFS and path building algorithms in Python. The bfs function faithfully implements the breadth-?rst search traversal algorithm, building and returning a dictionary containing vertices, each paired with its immediate predecessor on a shortest path from the starting vertex. The find_path function implements the second algorithm from above that uses the result of bfs to construct a list of vertices, in order, an a shortest path from from the starting vertex to the destination vertex.

The graph processed by Listing 16.1 (airlineroute.py) is shown in Figure 16.6. This graph adds three new airports to Figure 16.3: BNA, CHA, and CLT. These new vertices are interconnected to each other, but they are not connected to any of the vertices from the original graph. (In the real world you can get from CHA [Chattanooga, Tennessee], for example, to DEN, but this contrived graph illustrates the general case where a path may not exist between two vertices in a graph.)

This function essentially attempts to ?nd a path from all vertices to all other vertices (except from a vertex to itself). While this code works, it is very inef?cient. It performs a large amount of unnecessary work. As an example, given the graph in Figure 16.3 the is_connected function will ?nd a path between ATL and SFO and also during its processing ?nd a path between SFO and ATL. Since the graph is undirected, attempting to ?nd the reverse path is super?uous. If the number of vertices in the graph is n, the is_connected function calls the bfs function n2 ?n times. (The n2 is the number of possibilities resulting from pairing every vertex with every other vertex, and we subtract n so we do not count pairing a vertex with itself.) Using this technique on a graph with 50 vertices produces 2,450 calls of bfs.

If the caller omits the end_vertex parameter, the while loop within bfs will compute predecessors for all reachable vertices within the graph. This is because end_vertex defaults to None, and the function never adds the None key to its initially empty predecessor dictionary. If the caller omits both the start_vertex and end_vertex, the function will choose an arbitrary starting vertex from the graph. Due to the nature of default arguments, the caller has no way to omit only the start_vertex parameter (see Section 8.2).

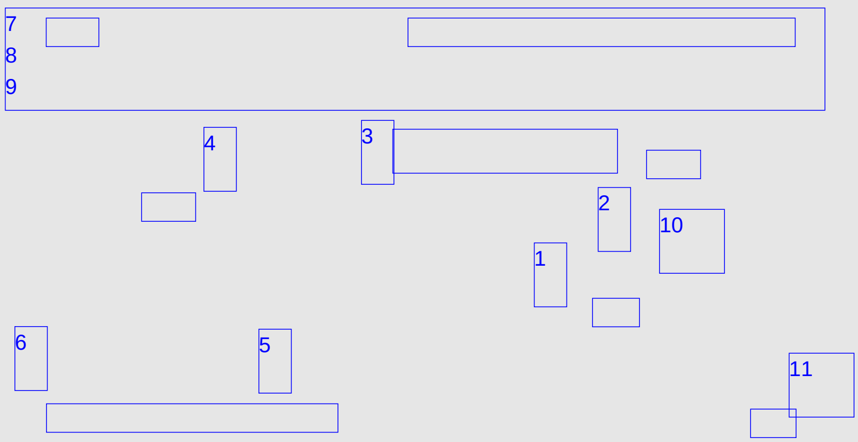
When traversing a graph an alternative to breadth-?rst search is depth-?rst search (DFS). BFS conservatively visits vertices at increasing distances, never venturing farther away until all closer options are exhausted. DFS, on the other hand, traces a path that moves as far away from the starting vertex as possible. DFS backs up to consider other paths only when it can go no further. It can proceed no further when it reaches a vertex connected only to already visited vertices. At this point it must back up to its most recently visited vertex and consider another path. It then continues its effort to move as far as possible away the starting vertex.

The BFS algorithm uses a queue to guide its traversal. DFS uses a different accessory data structure?a stack. A stack is a linear data structure with restricted access like a queue, but, unlike a queue, a stack permits removal of only the most recently added element. A stack, therefore, is a last-in, ?rst out (LIFO) container. Unlike the Queue class, Python does not have a standard stack class. Fortunately, we can use a regular list object in a disciplined way to perfectly model a stack.

Inputs: A starting vertex and destination vertex within a graph.
Begin with an empty set and an empty stack.
Push the starting vertex onto the stack.
while the stack is not empty and the set does not contain the destination vertex:
Pop the top vertex, v, from the stack.
if v in not in the set:
Add v to the set:
for each vertex w adjacent to v:

The ?rst time through its while loop the DFS algorithm extracts the starting vertex from the stack. Since the visited set begins empty, the algorithms adds the starting vertex to the set. It then will add all of the starting vertex?s adjacent vertices to the stack. On the second iteration of its while loop the algorithm removes from the stack the last vertex it added on the previous iteration, one adjacent to the starting vertex. Call this vertex v. The algorithm adds any nonvisited vertices adjacent to v to the stack. Because it uses a stack the algorithm will visit these newly added vertices before it visits any vertices it added to the stack earlier. This allows the algorithm to propagate to vertices farther away from the starting vertex before it considers closer vertices. Figure 16.7 illustrates one such DFS traversal.

Figure 16.7 The numbers indicate a possible order in which DFS visits vertices starting at vertex ATL. Since a Python program uses a dictionary to represent the graph, the exact order of the visitation is unpredictable, even from one execution to the next on the same machine. What is predictable, however, is that the path traced by DFS will tend to move farther from the starting vertex before visiting all vertices closer to the starting vertex. Observe that this traversal visits MIA last, even though MIA is adjacent to the starting vertex.



Index