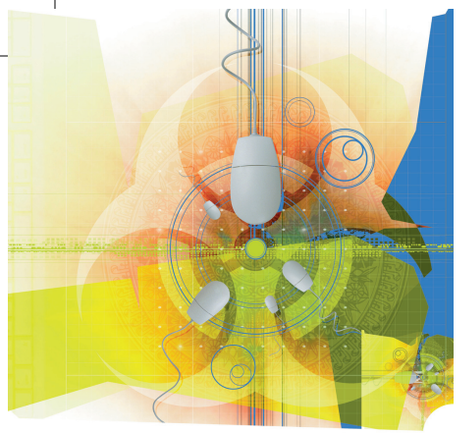
Programming in Python

Online module to accompany *Invitation to Computer Science, 7th Edition,* ISBN-10: 1305075773; ISBN-13: 9781305075771 (Cengage Learning, 2016).



1 Introduction to Python

Hundreds of high-level programming languages have been developed; a fraction of these have become viable, commercially successful languages. There are a half-dozen or so languages that can illustrate some of the concepts of a high-level programming language, but this module uses Python for this purpose.

Our intent here is not to make you an expert Python programmer—any more than our purpose in Chapter 4 was to make you an expert circuit designer. Indeed, there is much about the language that we will not even discuss. You will, however, get a sense of what programming in a high-level language is like, and perhaps see why some people think it is one of the most fascinating of human endeavors.

1.1 *A Simple Python Program*

Figure 1 shows a simple but complete Python program. Even if you know nothing about the Python language, it is not hard to get the general drift of what the program is doing.

FIGURE 1
A Simple Python Program

```
#Computes and outputs travel time
#for a given speed and distance
#Written by J. Q. Programmer, 6/15/16

speed = input("Enter your speed in mph: ")
speed = int(speed)
distance = input("Enter your distance in miles: ")
distance = float(distance)

time = distance/speed

print("At", speed, "mph, it will take")
print(time, "hours to travel", distance, "miles.")

input("\n\nPress the Enter key to exit")
```

Someone running this program (the user) could have the following dialogue with the program, where boldface indicates what the user types:

```
Enter your speed in mph: 58
Enter your distance in miles: 657.5
At 58 mph, it will take
11.3362068966 hours to travel 657.5 miles.
```

To aid our discussion of how the program works, Figure 2 shows the same program with a number in front of each line. The numbers are there for reference purposes only; they are *not* part of the program. Lines 1–3 in the program of Figure 2 are Python **comments**. Anything appearing on a line after the pound symbol (#) is ignored by the compiler, just as anything following the double dash (--) is treated as a comment in the assembly language programs of Chapter 6. Although the computer ignores comments, they are important to include in a program because they give information to the human readers of the code. Every high-level language has some facility for including comments, because understanding code that someone else has written (or understanding your own code after some period of time has passed) is very difficult without the notes and explanations that comments provide. Comments are one way to *document* a computer program to make it more understandable. The comments in lines 1–3 of Figure 2 describe what the program does plus tell who wrote the program and when. These three comment lines together make up the program's **prologue comment** (the introductory comment that comes first). A prologue comment is always a good idea; it's almost like the headline in a newspaper, giving the big picture up front.

Blank lines in Python programs are ignored and are used, like comments, to make the program more readable by human beings. In our example program, we've used blank lines (lines 4, 9, 11, 14) to separate sections of the program, visually indicating groups of statements that are related. Except for blank lines, Python by default considers each line of the program to be an individual program instruction, also called a program statement, so you end a statement by just pressing the Enter key and going to the next line to write the next statement.

FIGURE 2

The Program of Figure 1 (line numbers added for reference)

```
1. #Computes and outputs travel time
2. #for a given speed and distance
3. #Written by J. Q. Programmer, 6/15/16
4.
5. speed = input("Enter your speed in mph: ")
6. speed = int(speed)
7. distance = input("Enter your distance in miles: ")
8. distance = float(distance)
9.
10. time = distance/speed
11.
12. print("At", speed, "mph, it will take")
13. print(time, "hours to travel", distance, "miles.")
14.
15. input("\n\nPress the Enter key to exit")
```

The three quantities involved in this program are the speed of travel and the distance traveled (these are input by the user) and the time to complete that travel (this is computed and output by the program). The program code itself uses descriptive names—*speed*, *distance*, and *time*—for these quantities, to help document their purpose in the program. Lines 5–8 prompt the user to enter values for speed and distance, and store those values in *speed* and *distance*. Later, we'll see more details on how this works. Line 10 computes the time required to travel this distance at this speed. Finally, lines 12 and 13 print the two lines of output to the user's screen. The values of *speed*, *time*, and *distance* are inserted in appropriate places among the strings of text shown in double quotes. The final program statement, line 15, has the effect of holding the output on the user's screen until the user presses the Enter key; otherwise, the output might just flash by and be gone when the program ends.

Python, along with every other programming language, has specific rules of **syntax**—the correct form for each component of the language. Any violation of the syntax rules generates an error message from the compiler because the compiler does not recognize or know how to translate the offending code. Python's rules of syntax are much simpler than those of many other programming languages, which is one reason that Python programs are often shorter and, many would claim, easier to write than programs in C++, Java, Ada, or C#, for example. Nonetheless, a typing error such as

```
printt("Hello World")
```

will produce an error message, as will

```
Print("Hello World")
```

because Python is a **case-sensitive** language, which means that uppercase letters are distinguished from lowercase letters, and the instruction is *print*, not *Print*.

1.2 *Creating and Running a Python Program*

Creating and running a Python program is basically a two-step process. The first step is to type the program into a text editor. When you are finished, you

How About That Snake?

The Python programming language was created in the early 1990s by the Dutch computer scientist and applied mathematician Guido van Rossum as a successor to a language called ABC. It was not actually named for a snake. It was named for the BBC comedy show *Monty Python's Flying Circus*. Make of that what you will!

Since its original release, Python has gone through a number of revisions. Guido van Rossum remains the project leader and final arbiter of new enhancements to the language, although—because it is an open-source language—anyone can tinker with it and propose new features or additions, and many have contributed to its development.

Python is prized for its low-stress (minimalist and intuitive) syntax, which leads to quick development time. While the basics of the language are simple, an extensive library of supporting code makes it a flexible and powerful language.

save the file, giving it a name with the extension *.py*. So the file for Figure 1 could be named

```
TravelPlanner.py
```

The second step is to execute the program. Details of how to run the program depend on your system; you may be able to double-click on the TravelPlanner.py file, or you may have to type the command "TravelPlanner.py" at the operating system prompt. Whenever you run a Python program, the Python compiler (it's actually called an *interpreter*)[1] translates the Python code first into low-level code called **bytecode**, which is not yet object code, then finishes the translation into machine-specific object code and executes it. (Referring to Figure 9.1 in the *Invitation to Computer Science* textbook, there are no explicit linker or loader steps. The program goes quite seamlessly from high-level code to execution.) A Python program will therefore run on any computer that has Python on it.

Another approach is to do all of your work in an **Integrated Development Environment**, or **IDE**. The IDE lets the programmer perform a number of tasks within the shell of a single application program. A modern programming IDE provides a text editor, a file manager, a way to run the program, and tools for debugging, all within this one piece of software. The IDE usually has a graphical user interface (GUI) with menu choices for the different tasks. This can significantly speed up program development. Python comes with its own Integrated Development Environment called IDLE, so you can do all your Python program development with this one tool.

This Python exercise is just a beginning. In the rest of this module, we'll examine the features of the language that will enable you to write your own Python programs to carry out more sophisticated tasks.

PYTHON INTERPRETER

A free Python interpreter is available to download at *www.python.org/download*. Python comes with its own IDE called IDLE. You can do all your Python program development within this one tool. There are versions for Windows, Linux, and Mac OS X.

The graphics software used in Section 7 of this module is courtesy of Dr. John Zelle of Wartburg College, Iowa. This is open-source software released under the terms of the GPL (General Public License; see *www.gnu.org/licenses/gpl.html*) and may be downloaded for free from *http://mcsp.wartburg.edu/zelle/python*. Put the file (graphics.py) in the Python Lib folder after you have installed Python.

[1]The difference between a compiled language and an interpreted language is that a compiler translates source code into object code once. The compiled code is then used over and over, unless the source code is changed, requiring a recompile. An interpreter translates source code into object code each time the program is executed, and no object code is saved for later use.

2 Virtual Data Storage

One of the improvements we seek in a high-level language is freedom from having to manage data movement within memory. Assembly language does not require us to give the actual memory address of the storage location to be used for each item, as in machine language. However, we still have to move values, one by one, back and forth between memory and the arithmetic logic unit (ALU) as simple modifications are made, such as setting the value of A to the sum of the values of B and C. We want the computer to let us use data values by name in any appropriate computation without thinking about where they are stored or what is currently in some register in the ALU. In fact, we do not even want to know that there *is* such a thing as an ALU, where data are moved to be operated on. Instead, we want the virtual machine to manage the details when we request that a computation be performed. A high-level language allows this, and it also allows the names for data items to be more meaningful than in assembly language.

Names in a programming language are called **identifiers**. Each language has its own specific rules for what a legal identifier can look like. In Python an identifier can be any combination of letters, digits, and the underscore symbol (_), as long as it does not begin with a digit. An additional restriction is that an identifier cannot be one of the few words, such as "while", that have a special meaning in Python and that you would not be likely to use anyway. The three integers *B*, *C*, and *A* in our assembly language program can therefore have more descriptive names, such as *subTotal*, *tax*, and *finalTotal*. The use of descriptive identifiers is one of the greatest aids to human understanding of a program. Identifiers can be almost arbitrarily long, so be sure to use a meaningful identifier such as *finalTotal* instead of something like *A*; the improved readability is well worth the extra typing time. Remember that Python is case sensitive; thus, *FinalTotal*, *Finaltotal*, and *finalTotal* are three different identifiers.

CAPITALIZATION OF IDENTIFIERS

There are two standard capitalization patterns for identifiers, particularly "multiple word" identifiers:

camel case: First word begins with a lowercase letter, additional words begin with uppercase letters (*finalTotal*)

Pascal case: All words begin with an uppercase letter (*FinalTotal*)

The code in this chapter uses the following convention for creating identifiers (examples included):

Simple variables – camel case: *speed*, *time*, *finalTotal*

Named constants - all uppercase: *PI, FREEZING_POINT*

Function names – camel case: *myFunction*, *getInput*

Class names – Pascal case: *MyClass*

Object names – camel case: *myObject*

The underscore character is not used except for named constants. Occasionally, however, we'll use single capital letters for identifiers in quick code fragments.

Data that a program uses can come in two varieties. Most quantities used in a program have values that change as the program executes, or values that are not known ahead of time but must be obtained from the computer user (or from a data file previously prepared by the user) as the program runs. These quantities are called **variables**. Some programs may use quantities that are fixed throughout the duration of the program, and their values are known ahead of time. These quantities are called **constants**. The names for both variables and constants must follow the Python identifier syntax rules given previously.

Identifiers for variables and constants serve the same purpose in program statements as pronouns do in ordinary English statements. The English statement "He will be home today" has specific meaning only when we plug in the value for which "He" stands. Similarly, a program statement such as

```
time = distance/speed
```

becomes an actual computation only when numeric values have been stored in the memory locations referenced by the *distance* and *speed* identifiers.

We now know how to name variables, but how do we actually create them in a Python program? The syntax is very simple; variables are created and given values all in one statement of the form

```
identifier = value
```

For example,

```
myNumber = 15
```

associates the identifier *myNumber* with some (unknown to us and we don't care) memory location and stores the integer value 15 in that location. This statement is equivalent to the assembly language statement

```
myNumber: .DATA 15
```

Python recognizes different types of data that can be stored in variables, namely string data or numeric data. A string is just a sequence of characters; the statement

```
print("Hello World")
```

prints the exact sequence of characters within the quote marks. Such a string is sometimes called a **literal string** because it is printed out exactly as is. The first statement below stores that same string in a variable, and the second statement then prints out the contents of that variable:

```
message = "Hello World"
print(message)
```

The effect is the same in either case—the user sees

```
Hello World
```

on the screen.

FIGURE 3

Some of the Python Data Types

int	an integer quantity
float	a real number (a decimal quantity)
string	a sequence of characters

There are also several different types of numeric data, the most common being type *int* and type *float*. Whole numbers with no decimal points, such as –28 and 5231, are of type *int*; numbers with decimal points, such as 25.8 or –52.976, are of type *float*. Figure 3 lists these common Python **data types**.

We know that all data are represented internally in binary form. In Chapter 4 we noted that any one sequence of binary digits can be interpreted as a whole number, a negative number, a real number (one containing a decimal point, such as –17.5 or 28.342), etc. In Python, a variable doesn't have a fixed data type associated with it. Instead, it takes on the data type of whatever value it currently contains. After execution of

```
myNumber = 15
```

the binary string in memory location *myNumber* will be interpreted as an integer. If the statement

```
myNumber = 65.81
```

is executed later in the same program, Python will then interpret the binary string in *myNumber* as a decimal value. Still later, *myNumber* could take on the string data type with the statement

```
myNumber = "This is my number"
```

although at this point the identifier would be somewhat misleading! And this points out a difficulty with the ability of an identifier to take its data type from the value assigned to it at the moment. If the reader of the program has to remember, "let's see, in this section of code the variable *xyz* means something-or-other, but down in that section of code it means something-or-other-else," then there is room for confusion. Good programming practice says that an identifier should represent only one thing in a given program.

Let's consider program constants. An example of a constant is the integer value 2. The integer 2 is a constant that we don't have to name by an identifier, nor do we have to build the value 2 in memory manually by the equivalent of a .DATA pseudo-op. We can just use the symbol "2" in any program statement. When "2" is first encountered in a program statement, the binary representation of the integer 2 is automatically generated and stored in a memory location. In a program that does computations about circles, an approximation to π, say 3.1416, could be used just as easily by simply sticking this number wherever in the program we need it. But if we are really using this number as an approximation to π, it is more informative to use the identifier *PI*. The statement

```
PI = 3.1416
```

stores the desired decimal value in a memory location with the identifier *PI*. The convention among Python programmers is that an identifier of all caps,

like *PI*, represents a constant value in the program. Therefore the value of *PI* should not be changed by a later program statement. However, Python won't prevent you from later making a change in the value of *PI*, so it's up to you as the programmer to treat this value of *PI* as unchangeable. In Python, then, a named constant is really just a variable.

In addition to variables of a primitive data type that hold only one unit of information, we can create a whole collection, called a **list**, of logically related variables at one time. This allows storage to be set aside as needed to contain each of the values in this collection. For example, suppose we want to create a roster of students in the class. We can do this using a Python list that contains string data.[2] The following two statements create a Python list and print its contents.

```
roster = ["Martin", "Susan", "Chaika", "Ted"]
         print(roster)
```

The output is

```
['Martin', 'Susan', 'Chaika', 'Ted']
```

While *roster* refers to the list as a whole, individual list elements can be accessed by giving their position or index in the list. List indices begin at 0, so

```
print(roster[1])
```

produces output of

```
Susan
```

Figure 4 illustrates this list.

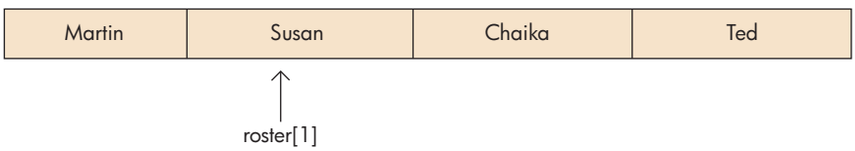
Here is an example of the power of a high-level language. In assembly language, we can name only individual memory locations—that is, individual items of data—but in Python we can also assign a name to an entire collection of related data items. A list thus allows us to talk about an entire table of values, or the individual elements making up that table. If we are writing Python programs to implement the data cleanup algorithms of Chapter 3, we can use a list of integers to store the 10 data items.

A Python list can perform many different actions. For example, the list can be put into sorted order:

```
roster.sort()
print(roster)
```

FIGURE 4

A 4-Element List roster



[2]A Python list serves the same purpose as what is usually called an "array" in other languages.

produces output of

```
['Chaika', 'Martin', 'Susan', 'Ted']
```

PRACTICE PROBLEMS

1. Which of the following are legitimate Python identifiers?

   ```
   martinBradley   C3P_OH   Amy3   3Right   Print
   ```

2. What is the output from the following fragment of Python code?

   ```
   myVariable = 65
   myVariable = 65.0
   myVariable = "Sixty Five"
   print(myVariable)
   ```

3. Using the *roster* list of Figure 4, how do you reference the last item in the list?

3 Statement Types

Now that we understand how to create variables to hold data, we will examine additional kinds of programming instructions (statements) that Python provides. These statements enable us to manipulate the data items and do something useful with them. The instructions in Python, or indeed in any high-level language, are designed as components for algorithmic problem solving, rather than as one-to-one translations of the underlying machine language instruction set of the computer. Thus, they allow the programmer to work at a higher level of abstraction. In this section we examine three types of high-level programming language statements. They are consistent with the pseudocode operations described in Chapter 2 (see Figure 2.9).

Input/output statements make up one type of statement. An **input statement** collects a specific value from the user for a variable within the program. In our TravelPlanner program, we need input statements to get the values of the speed and distance that are to be used in the computation. An **output statement** writes a message or the value of a program variable to the user's screen. Once the TravelPlanner program computes the time required to travel the given distance at the given speed, the output statement displays that value on the screen, along with other information about what that value means.

Another type of statement is the **assignment statement**, which assigns a value to a program variable. This is similar to what an input statement does, except that the value is not collected directly from the user, but is computed by the program. In pseudocode we called this a "computation operation."

Control statements, the third type of statement, affect the order in which instructions are executed. A program executes one instruction or program

statement at a time. Without directions to the contrary, instructions are executed sequentially, from first to last in the program. (In Chapter 2 we called this a straight-line algorithm.) Imagine beside each program statement a light bulb that lights up while that statement is being executed; you would see a ripple of lights from the top to the bottom of the program. Sometimes, however, we want to interrupt this sequential progression and jump around in the program (which is accomplished by the instructions JUMP, JUMPGT, and so on, in assembly language). The progression of lights, which may no longer be sequential, illustrates the **flow of control** in the program—that is, the path through the program that is traced by following the currently executing statement. Control statements direct this flow of control.

3.1 *Input/Output Statements*

Remember that the job of an input statement is to collect from the user specific values for variables in the program. In pseudocode, to get the value for *speed* in the TravelPlanner program, we would say something like

Get value for *speed*

In the Python TravelPlanner program, the equivalent program statement is

```
speed = input("Enter your speed in mph: ")
```

This statement accomplishes several things at once, and we should look at it in pieces. The right-hand side of the equals sign,

```
input("Enter your speed in mph: ")
```

is done first, using the built-in Python *input* function ("built-in" means that this function is supplied with the Python language). We pass information to the *input* function in the form of a literal string (enclosed in quote marks). This writes a message to the user requesting information about the speed. Such a message is called a **user prompt**; it alerts the user that the program is waiting for some input. The *input* function also reads the user's response from the keyboard and "returns" this input data. The complete Python statement

```
speed = input("Enter your speed in mph: ")
```

stores the result that *input* returns in the variable *speed*. But *input* always captures the input data as a string of characters. If the user enters 58, then the *input* function captures the string consisting of a 5 followed by an 8; this is just a two-character string, similar to the string "ab" consisting of an *a* followed by a *b*. In other words, the two-length string of characters "58" is not the same as the integer numeric value of 58, and we could not do any numerical computations with it. The next statement of our sample program

```
speed = int(speed)
```

converts the string value "58" into the integer number 58 and again stores the result in *speed*. This statement uses the built-in Python *int* function; we give this function information in the form of the current value of *speed* (which is

string data). The *int* function converts the string into its integer equivalent and "returns" the integer value 58. This in turn is stored in the *speed* variable, which now contains data of type *int*.

The statements

```
distance = input("Enter your distance in miles: ")
distance = float(distance)
```

do the same thing using the built-in *float* function that converts the string returned by the *input* function into a decimal value that then gets stored in *distance*. Here we have assumed that the user might enter a decimal value for distance, as in the sample dialogue, so we used the *float* conversion function rather than the *int* function.

It's possible to chain functions together in one statement. The statement

```
speed = int(input("Enter your speed in mph: "))
```

works by putting the string value returned by the *input* function directly into the *int* function (skipping the intermediate step of storing it in *speed*) and then storing the result returned by the *int* function in *speed*. This single statement has the same effect as the two statements we used earlier, and makes it clear that the real purpose of *speed* in the program is to store an integer value.

This conversion, or **type casting**, from string data to integer data only works if the user entered a string that can be interpreted as an integer. If the user interaction with the TravelPlanner program is

```
Enter your speed in mph: abc
```

an error message will result that says something like "invalid literal for int() with base 10." In other words, Python could not convert the literal string "abc" that the user entered into a base 10 integer. The following

```
Enter your speed in mph: 45.7
```

produces a similar result. However, if the user enters 145 in response to

```
distance = float(input("Enter your distance in miles: "))
```

the *float* function will happily perform automatic type casting and convert the string value "145" to the equivalent decimal value 145.0.

The *int* and *float* functions also perform conversions on numeric data. The result returned by

```
float(145)
```

is, not unexpectedly, 145.0. The result returned by

```
int(45.7)
```

is 45. Notice that while the *int* function could make nothing of the string "45.7", it will truncate the numeric value 45.7.

To complete the picture, there is a Python *str* function that converts numeric data into string data. The result returned by

```
str(56)
```

is the string "56".

Once the value of *time* has been computed in the TravelPlanner program, we want to write it out. A pseudocode operation for producing output would be something like

```
Print the value of time
```

and the Python output statement is almost the same thing:

```
print(time)
```

This statement uses the built-in Python *print* function. The *print* function writes out the information given to it as a string. If the value of *time* is 2.35, the *print* function converts that to the string sequence of four characters "2.35" and that is what is output. Of course the sequence of characters 2.35 and the decimal value 2.35 look exactly the same, so it doesn't matter.

But we don't want the program to simply print a number (or something that looks like a number) with no explanation; we want some words to make the output meaningful. In our TravelPlanner program, we used the *print* function twice, in each case giving the function a mixture of literal strings and numeric variables, separated by commas.

```
print("At", speed, "mph, it will take")
print(time, "hours to travel", distance, "miles.")
```

Each print function produced one line of output. The output was

```
At 58 mph, it will take
11.3362068966 hours to travel 657.5 miles.
```

The *print* function type cast the numeric variables into their string equivalents, and helpfully inserted blanks before each piece (except the first string piece) so that the output looks nice and is not jammed together.

The appearance of the output string can be modified by using an **escape sequence** within a literal string. An escape sequence consists of a backslash (\) followed by a single character; the combination is treated as a unit that results in special output-formatting effects, not as two characters of the literal string itself. Two useful escape sequences are

\n	Insert a new line
\t	insert a tab character

The result of

```
print("This is my favorite \n\t yellow puppydog.")
```

is

```
This is my favorite
        yellow puppydog.
```

The \n forces the second part of the literal string to print on a new line, and the \t indents the second line one tab distance from the left margin. In this way a single *print* function can print more than one line. Conversely, it's possible to make several print functions produce only a single line of output. The result of

```
print(3, end = " ")
print(6, end = " ")
print(7)
```

is

```
3 6 7
```

We mentioned earlier that Python by default considers each line of the program to be an individual program statement, but a single statement can be spread over multiple lines by putting a backslash at the end of a line. This use of the backslash means that the next line is a continuation of this same statement, and the \ is called a **line continuation character**. The following is one statement spread over two lines,

```
print("Oh for a sturdy ship to sail, "\
      "and a star to steer her by.")
```

and the result is

```
Oh for a sturdy ship to sail, and a star to steer her by.
```

The *print* function prints a blank line if no information is passed to it. Therefore

```
print("Hello")
print( )
print("World")
```

would result in

```
Hello

World
```

Another way to get a blank line is to insert the \n escape sequence several times within a literal string. This explains the last line of code in the Travel-Planner program:

```
input("\n\nPress the Enter key to exit")
```

The *input* function prints the user prompt on a new line in any case, so the effect of the \n escape sequences is to add two extra blank lines before the

user prompt. Because the program is waiting to read input, the screen remains visible until the user presses Enter, at which point the program terminates. We've usually stored the string returned by the *input* function in a program variable, but here it just gets ignored.

Literal strings can be joined together by the **concatenation operator**, represented by a + sign. The statement

```
print("First part. " + "Second part.")
```

produces

```
First part. Second part.
```

Here we put a space at the end of the first string in order to separate the printed result from the second string. We could use string concatenation to write the output from the TravelPlanner program, but we'd have to convert the numerical values into strings because concatenation only works on strings. This would look like

```
print("At " + str(speed) + " mph, it will take")
print(str(time) + " hours to travel "\
  + str(distance) + " miles.")
```

where again we had to insert spaces into the literal strings. This is a lot of work compared to the original two statements that let the *print* function do the type casting, spacing, and concatenation for us automatically!

Finally, in our sample execution of the TravelPlanner program, we got the following output:

```
At 58 mph, it will take
11.3362068966 hours to travel 657.5 miles.
```

This is fairly ridiculous output—it does not make sense to display the result to 10 decimal digits. Exercise 11 at the end of this module tells you how decimal output can be formatted to a specified number of decimal places.

PRACTICE PROBLEMS

1. Write a single statement that prompts the user to enter an integer value and stores that value (as an integer) in a variable called *quantity*.
2. A program has computed a value for the variable *average* that represents the average high temperature in San Diego for the month of May. Write an appropriate output statement.
3. What appears on the screen after execution of the following statement?

   ```
   print("This is" + "goodbye" + ", Steve")
   ```

3.2 *The Assignment Statement*

As we said earlier, an assignment statement assigns a value to a program variable. This is accomplished by evaluating some expression and then writing the resulting value in the memory location referenced by the program variable. The general pseudocode operation

Set the value of "variable" to "arithmetic expression"

has as its Python equivalent

```
variable = expression
```

The expression on the right is evaluated, and the result is then written into the memory location named on the left. For example, the assignment statements

```
B = 2
C = 5
```

result in *B* taking on the value 2 and *C* taking on the value 5. After execution of

```
A = B + C
```

A has the value that is the sum of the current values of *B* and *C*. Assignment is a destructive operation, so whatever *A*'s previous value was, it is gone. Note that this one assignment statement says to add the values of *B* and *C* and assign the result to *A*. This one high-level language statement is equivalent to three assembly language statements needed to do this same task (LOAD B, ADD C, STORE A). A high-level language program thus packs more power per line than an assembly language program. To state it another way, whereas a single assembly language instruction is equivalent to a single machine language instruction, a single Python instruction is usually equivalent to many assembly language instructions or machine language instructions, and allows us to think at a higher level of problem solving.

In the assignment statement, the expression on the right is evaluated first. Only then is the value of the variable on the left changed. This means that an assignment statement like

```
A = A + 1
```

makes sense. If *A* has the value 7 before this statement is executed, then the expression evaluates to

7 + 1 or 8

and 8 then becomes the new value of *A*. (Here it becomes obvious that the assignment operator = is not the same as the mathematical equals sign =, because $A = A + 1$ does not make sense mathematically.)

All four basic arithmetic operations can be done in Python, denoted by

```
+ Addition
- Subtraction
* Multiplication
/ Division
```

For the most part, this is standard mathematical notation, rather than the somewhat verbose assembly language op code mnemonics such as SUBTRACT. The reason a special symbol is used for multiplication is that $\times$ would be confused with *x*, an identifier, $\cdot$ (a multiplication dot) doesn't appear on the keyboard, and juxtaposition (writing *AB* for *A*B*) would look like a single identifier named *AB*.

As soon as an arithmetic operation involves one or more real numbers, any integers are converted to their real number equivalent, and the calculations are done with real numbers. Thus the following divisions behave as we expect:

```
7.0/2 7/2.0 7.0/2.0
```

all result in the value 3.5. In Python, the division

```
7/2
```

also results in the value 3.5, even though both the numerator and denominator are integers.[3] But if we think of grade-school long division of integers:

$$\begin{array}{r} 3 \\ 2\overline{)7} \\ \underline{6} \\ 1 \end{array}$$

we see that the division of 7 by 2 results in an integer quotient of 3 and an integer remainder of 1. In other words,

```
7 = 2 * 3 + 1
```

Python provides two operations, symbolized by // and %, that break down integer division into its integer quotient and its integer remainder, respectively. Using these operators,

```
7 // 2
```

results in the value 3, and

```
7 % 2
```

results in the value 1.

[3]This is new beginning with Python 3.0. Earlier versions of Python performed integer division when two integer values were divided.

If the values are stored in variables, the result is the same. For example,

```
numerator = 7
denominator = 2
quotient = numerator/denominator
intQuotient = numerator//denominator
remainder = numerator % denominator
print("The result of", numerator,\
      "/", denominator, "is", quotient)
print("The integer quotient of", numerator, \
      "/", denominator, "is", intQuotient)
print("and the remainder is", remainder)
```

produces the output

```
The result of 7 / 2 is 3.5
The integer quotient of 7 / 2 is 3
and the remainder is 1
```

The expression on the right side of an assignment statement need not evaluate to a numerical value. The input statement

```
speed = input("Enter your speed in mph: ")
```

is also an assignment statement. The expression on the right applies the *input* function to write a user prompt and read the sequence of characters the user types in at the keyboard. The resulting string that the *input* function returns is then assigned to the *speed* variable.

The Python print function does something neat with arithmetic expressions. For example, the result of

```
number = 25
print("New number is", 3 + number)
```

is

```
New number is 28
```

PRACTICE PROBLEMS

1. *newNumber* and *next* contain integer values in a Python program. Write a statement to assign the value of *newNumber* to *next*.
2. What is the value of *Average* after the following statements are executed?

```
Total = 277
Number = 5
Average = Total//Number
```

Recall that the print function type casts numerical values to strings, but in this case the arithmetic expression is evaluated first, and then the numerical result of that expression, 28, is converted to the string "28" for printing.

3.3 *Control Statements*

We mentioned earlier that sequential flow of control is the default; that is, a program executes instructions sequentially from first to last. The flowchart in Figure 5 illustrates this, where S1, S2, . . . , Sk are program instructions (program statements).

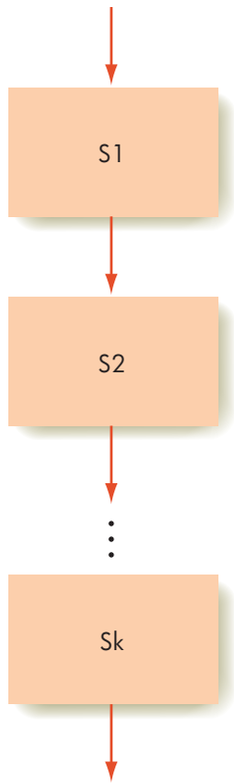
As stated in Chapter 2, no matter how complicated the task to be done, only three types of control mechanisms are needed:

1. **Sequential**: Instructions are executed in order.
2. **Conditional**: Which instruction executes next depends on some condition.
3. **Looping**: A group of instructions may be executed many times.

Sequential flow of control, the default, is what occurs if the program does not contain any instances of the other two control structures. In the TravelPlanner program, for instance, instructions are executed sequentially, beginning with the input statements, next the computation, and finally the output statements.

In Chapter 2 we introduced pseudocode notation for conditional operations and looping. In Chapter 6 we learned how to write somewhat laborious

FIGURE 5
Sequential Flow of Control



assembly language code to implement conditional operations and looping. Now we'll see how Python provides instructions that directly carry out these control structure mechanisms—more evidence of the power of high-level language instructions. We can think in a pseudocode algorithm design mode, as we did in Chapter 2, and then translate that pseudocode directly into Python code.

Conditional flow of control begins with the evaluation of a **Boolean condition**, also called a **Boolean expression**, that can be either true or false. We discussed these "true/false conditions" in Chapter 2, and we also encountered Boolean expressions in Chapter 4, where they were used to design circuits. A Boolean condition often involves comparing the values of two expressions and determining whether they are equal, whether the first is greater than the second, and so on. Again assuming that *A*, *B*, and *C* are variables with integer values in a program, the following are legitimate Boolean conditions:

`A == 0`	(Does *A* currently have the value 0?)
`B < (A + C)`	(Is the current value of *B* less than the sum of the current values of *A* and *C*?)
`A != B`	(Does *A* currently have a different value than *B*?)

If the current values of *A*, *B*, and *C* are 2, 5, and 7, respectively, then the first condition is false (*A* does not have the value zero), the second condition is true (5 is less than 2 plus 7), and the third condition is true (*A* and *B* do not have equal values).

Comparisons need not be numeric. They can also be made between string values, in which the "ordering" is the usual alphabetic ordering. If the current value of *Color* is "Red", then

```
Color > "Blue"
```

is true because "Red" comes after (is greater than) "Blue".

Figure 6 shows the comparison operators available in Python. Note the use of the two equality signs to test whether two expressions have the same value. The single equality sign is used in an assignment statement, the double equality sign in a comparison.

Boolean conditions can be built up using the Boolean operators *and*, *or*, and *not*. Truth tables for these operators were given in Chapter 4 (Figures 4.12–4.14). Python uses the English language connective words for these operators, making them very easy to understand (see Figure 7).

A conditional statement relies on the value of a Boolean condition (true or false) to decide which programming statement to execute next. If the

FIGURE 6

Python Comparison Operators

Comparison	Symbol	Example	Example Result
the same value as	==	2 == 5	false
less than	<	2 < 5	true
less than or equal to	<=	5 <= 5	true
greater than	>	2 > 5	false
greater than or equal to	>=	2 >= 5	false
not the same value as	!=	2 != 5	true

FIGURE 7
Python Boolean Operators

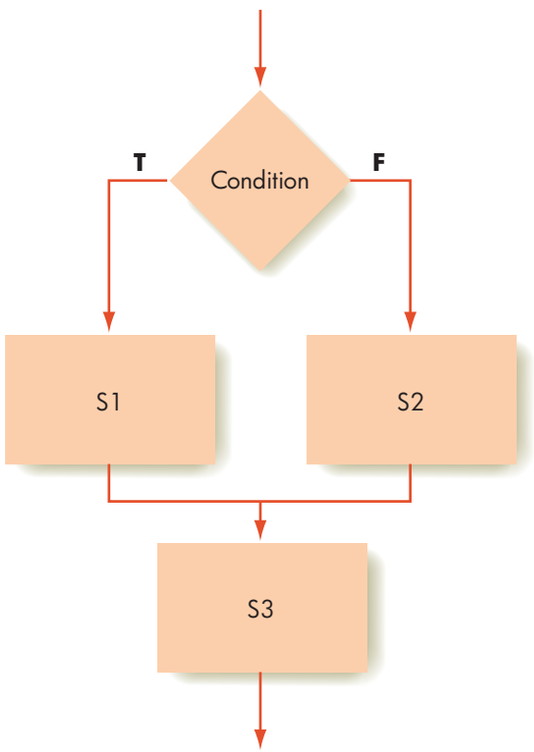
Operator	Symbol	Example	Example Result
AND	and	(2 < 5) and (2 > 7)	false
OR	or	(2 < 5) or (2 > 7)	true
NOT	not	not (2 == 5)	true

condition is true, one statement is executed next, but if the condition is false, a different statement is executed next. Control is therefore no longer in a straight-line (sequential) flow, but may hop to one place or to another. Figure 8 illustrates this situation. If the condition is true, the statement S1 is executed (and statement S2 is not); if the condition is false, the statement S2 is executed (and statement S1 is not). In either case, the flow of control then continues on to statement S3. We saw this same scenario when we discussed pseudocode conditional statements in Chapter 2 (Figure 2.4).

The Python instruction that carries out conditional flow of control is called an **if-else** statement. It has the following form (note that the words *if* and *else* are lowercase, and that there is a colon at the end of the Boolean condition and at the end of *else*):

```
if Boolean condition:
        S1
else:
        S2
```

FIGURE 8
Conditional Flow of Control (if-else)



Below is a simple if-else statement, where we assume that *A*, *B*, and *C* have integer values:

```
if B < (A + C):
      A = 2*A
else:
      A = 3*A
```

Suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 5, and 7, respectively. As we noted before, the condition $B < (A + C)$ is then true, so the statement

```
A = 2*A
```

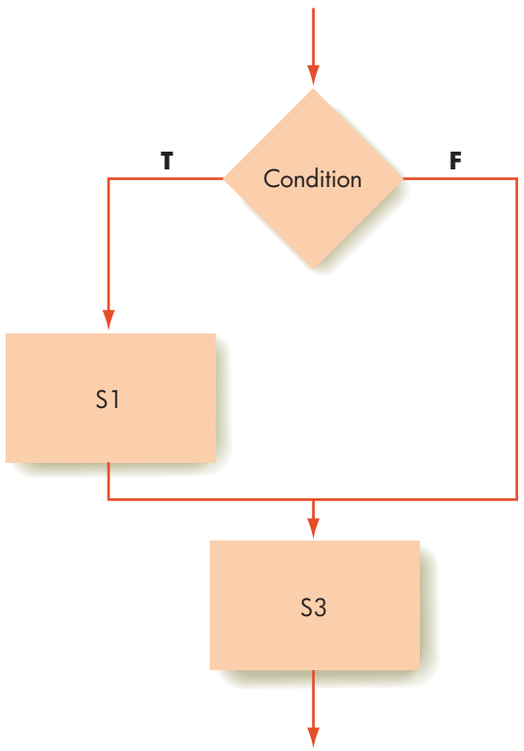
is executed, and the value of *A* is changed to 4. However, suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 10, and 7, respectively. Then the condition $B < (A + C)$ is false, the statement

```
A = 3*A
```

is executed, and the value of *A* is changed to 6.

A variation on the if-else statement is to allow an "empty else" case. Here we want to do something if the condition is true, but if the condition is false, we want to do nothing. Figure 9 illustrates the empty else case. If the condition is true, statement S1 is executed and after that the flow of control continues on to statement S3, but if the condition is false, nothing happens except that the flow of control moves directly on to statement S3.

FIGURE 9
If-Else with Empty Else



This *if* variation of the if-else statement can be accomplished by omitting the word *else*. This form of the instruction therefore looks like

```
if  Boolean condition:
        S1
```

We could write

```
if B < (A + C):
       A = 2*A
```

This has the effect of doubling the value of *A* if the condition is true and of doing nothing if the condition is false.

Multiple statements can be combined into a block. The block is then treated as a single statement, called a **compound statement**. A compound statement can be used anywhere a single statement is allowed. The implication is that in Figure 8, S1 or S2 might be a compound statement. This makes the if-else statement potentially much more powerful, and similar to the pseudocode conditional statement in Figure 2.9. Python recognizes a block by two clues. First, the colon after the Boolean condition or after the *else* indicates that a block is coming next. The extent of the block is given by indentation. For example,

```
if snack == "pb & j":
      print("yummy")
      print("sticky")
      print("gooey")
else:
      print("Must be pizza")
print("That's All, Folks")
```

If the variable *snack* does have the string value "pb & j" at this point, then the block of three print statements is executed, and the output is

```
yummy
sticky
gooey
That's All, Folks
```

If snack has some different value, the one-statement *else* block is executed, and the output is

```
Must be pizza
That's All, Folks
```

Indenting in Python not only makes a program easier for human beings to read, it's also used by the Python interpreter to determine the extent of a block of code. Use the Tab key to get the proper levels of indenting.

Let's expand on our TravelPlanner program and give the user of the program a choice of computing the time either as a decimal number (3.75 hours) or as hours and minutes (3 hours, 45 minutes). This situation is ideal for a conditional statement. Depending on what the user wants to do, the program does one of two tasks. For either task, the program still needs information

about the speed and distance. The program must also collect information to indicate which task the user wishes to perform. We need an additional variable in the program to store this information. Let's use a variable called *choice* to store the user's choice of which task to perform. We also need two new variables to store the (integer) values of hours and minutes.

Figure 10 shows the new program, with the three additional variables. The condition evaluated at the beginning of the if-else statement tests whether *choice* has the value "D" or "d". (Although the program asks the user to enter "D", it's easy to imagine someone typing lowercase "d" instead, and this compound condition works with either value.) If so, then the condition is true, and the first block of statements is executed—that is, the time is output in decimal format as we have been doing all along. If the condition is false, then the second block of statements is executed. In either case, the program then exits normally with the final *input* statement. Note that because of the way the condition is written, if *choice* does not have the value "D" or the value "d", it is assumed that the user wants to compute the time in hours and minutes, even though *choice* may have any other value that the user may have typed in response to the prompt.

To compute hours and minutes (the *else* clause of the if-else statement), time is computed in the usual way, which results in a decimal value. The whole-number part of that decimal is the number of hours needed for the trip. We can get this number by type casting the decimal number to an integer. This is accomplished by

```
hours = int(time)
```

which drops all digits behind the decimal point and stores the resulting integer value in *hours*. To find the fractional part of the hour that we dropped,

FIGURE 10

The TravelPlanner Program with a Conditional Statement

```
#Computes and outputs travel time
#for a given speed and distance
#Written by J. Q. Programmer, 6/28/16

speed = int(input("Enter your speed in mph: "))
distance = float(input("Enter your distance in miles: "))
print("Enter your choice of format for time")
choice = input("decimal hours (D) or hours and minutes (M): ")
print()

if choice == "D" or choice == "d":
    time = distance/speed
    print("At", speed, "mph, it will take")
    print(time, "hours to travel", distance, "miles.")
else:
    time = distance/speed
    hours = int(time)
    minutes = int((time - hours)*60)
    print("At", speed, "mph, it will take")
    print(hours, "hours and", minutes, "minutes to travel",\
          distance, "miles.")

input("\n\nPress the Enter key to exit")
```

we subtract *hours* from *time*. We multiply this by 60 to turn it into some number of minutes, but this is still a decimal number. We do another type cast to truncate this to an integer value for *minutes*:

```
minutes = int((time - hours)*60)
```

For example, if the user enters data of 50 mph and 475 miles and requests output in hours and minutes, the following table shows the computed values.

Quantity	*Value*
speed	50
distance	475.0
time = distance/speed	9.5
hours = int(time)	9
time – hours	0.5
(time – hours) *60	30.0
minutes = int((time – hours)*60)	30

Here is the actual program output for this case:

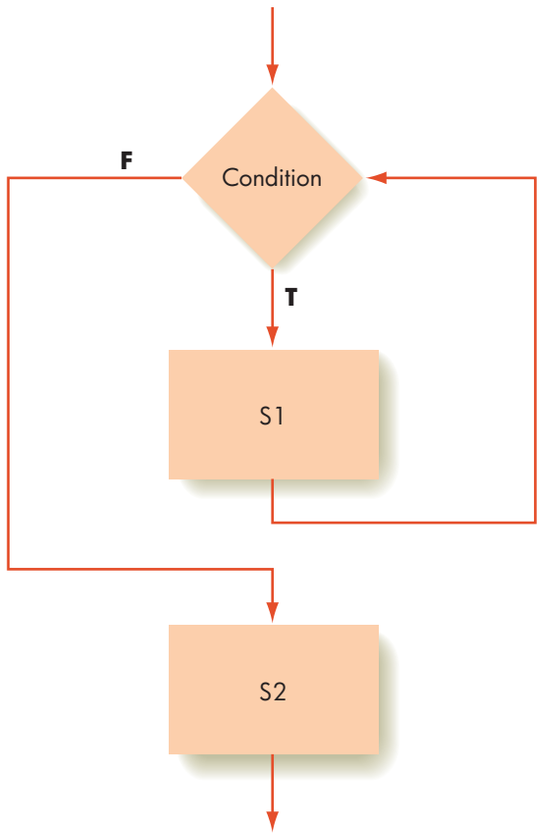
```
Enter your speed in mph: 50
Enter your distance in miles: 475
Enter your choice of format for time
decimal hours (D) or hours and minutes (M): M

At 50 mph, it will take
9 hours and 30 minutes to travel 475.0 miles.
```

Now let's look at the third variation on flow of control, namely looping (iteration). We want to execute the same group of statements (called the **loop body**) repeatedly, depending on the result of a Boolean condition. As long as (while) the condition remains true, the loop body is executed. The condition is tested before each execution of the loop body. When the condition becomes false, the loop body is not executed again, which is usually expressed by saying that the algorithm *exits* the loop. To ensure that the algorithm ultimately exits the loop, the condition must be such that its truth value can be affected by what happens when the loop body is executed. Figure 11 illustrates the while loop. The loop body is statement S1 (which can be a compound statement), and S1 is executed *while* the condition is true. Once the condition is false, the flow of control moves on to statement S2. If the condition is false when it is first evaluated, then the body of the loop is never executed at all. We saw this same scenario when we discussed pseudocode looping statements in Chapter 2 (Figure 2.6).

Python uses a **while** statement to implement this type of looping. The form of the statement is:

```
while Boolean condition:
    S1
```

FIGURE 11
While Loop



The colon after the Boolean condition indicates that a block of code is coming, and the extent of the block is determined by indentation.

For example, suppose we want to write a program to add a sequence of nonnegative integers that the user supplies and write out the total. We need a variable to hold the total; we'll call this variable *sum*. To handle the numbers to be added, we could use a bunch of variables such as *n1*, *n2*, *n3*, . . . , and do a series of input-and-add statements of the form

```
n1 = int(input("next number: "))
sum = sum + n1
n2 = int(input("next number: "))
sum = sum + n2
```

and so on. The problem is that this approach requires too much effort. Suppose we know that the user wants to add 2000 numbers. We could write the above input-and-add statements 2000 times, but it wouldn't be fun. Nor is it necessary—we are doing a very repetitive task here, and we should be able to use a loop mechanism to simplify the job. (We faced a similar situation in the first pass at a sequential search algorithm, Figure 2.11; our solution there was also to use iteration.)

Even if we use a loop mechanism, we are still adding a succession of values to *sum*. Unless we are sure that the value of *sum* is zero to begin with, we cannot be sure that the answer isn't nonsense. We should set the value of *sum* to 0 before we add anything to it.

Now on to the loop mechanism. First, let's note that once a number has been read in and added to *sum*, the program doesn't need to know the value of the number any longer. We can have just one variable called *number* and use it repeatedly to hold the first numerical value, then the second, and so on.

The general idea is then

```
sum = 0

while there are more numbers to add:
      number = int(input("Next number: "))
      sum = sum + number

print("The total is", sum)
```

Now we have to figure out what the condition "there are more numbers to add" really means. Because we are adding nonnegative integers, we could ask the user to enter one extra integer that is not part of the legitimate data, but is instead a signal that there *are* no more data. Such a value is called a **sentinel value**. For this problem, any negative number would be a good sentinel value. Because the numbers to be added are all nonnegative, the appearance of a negative number signals the end of the legitimate data. We don't want to process the sentinel value (because it is not a legitimate data item); we only want to use it to terminate the looping process. This might suggest the following code:

```
sum = 0
while number >= 0:     #but there is a problem here
                       #see following discussion

      number = int(input("Next number: "))
      sum = sum + number

print("The total is", sum)
```

Here's the problem. How can we test whether *number* is greater than or equal to 0 if we haven't read the value of *number* yet? We need to do a preliminary input for the first value of *number* outside of the loop, then test that value in the loop condition. If it is nonnegative, we want to add it to *sum* and then read the next value and test it. Whenever the value of *number* is negative (including the first value), we want to do nothing with it—that is, we want to avoid executing the loop body. The following statements do this; we've also added instructions to the user.

```
sum = 0
print("Enter numbers to add, terminate"\
      " with a negative number.")

number = int(input("Next number: "))
while number >= 0:
    sum = sum + number
    number = int(input("Next number: "))

print("The total is", sum)
```

The value of *number* gets changed within the loop body by reading in a new value. The new value is tested, and if it is nonnegative, the loop body executes again, adding the data value to *sum* and reading in a new value for *number*. The loop terminates when a negative value is read in. Remember the requirement that something within the loop body must be able to affect the truth value of the condition. In this case, it is reading in a new value for *number* that has the potential to change the value of the condition from true to false. Without this requirement, the condition, once true, would remain true forever, and the loop body would be endlessly executed. This results in what is called an **infinite loop**. A program that contains an infinite loop will execute forever (or until the programmer gets tired of waiting and interrupts the program, or until the program exceeds some preset time limit).

Here is a sample of the program output:

```
Enter numbers to add; terminate with a negative number.
Next number: 6
Next number: 8
Next number: 23
Next number: -4
The total is 37
```

The problem we've solved here, adding nonnegative integers until a negative sentinel value occurs, is the same one solved using assembly language in Chapter 6. The Python code above is almost identical to the pseudocode version of the algorithm shown in Figure 6.7. Thanks to the power of the language, the Python code embodies the algorithm directly, at a high level of thinking, whereas in assembly language this same algorithm had to be translated into the lengthy and awkward code of Figure 6.8.

To process data for a number of different trips in the TravelPlanner program, we could use a while loop. During each pass through the loop, the program computes the time for a given speed and distance. The body of the loop is therefore exactly like our previous code. All we are adding here is the framework that provides looping. To terminate the loop, we could use a sentinel value, as we did for the program above. A negative value for *speed*, for example, is not a valid value and could serve as a sentinel value. Instead of that, let's allow the user to control loop termination by having the program ask the user whether he or she wishes to continue. We'll need a variable to hold the user's response to this question. Of course, the user could answer "N" at the first query, the loop body would never be executed at all, and the program would terminate.

Figure 12 shows the complete program. Following the indentation, one can see that the overall structure of the program is

- opening query to the user
- a while loop that contains an if-else statement
- the usual closing statement

FIGURE 12

The TravelPlanner Program with Looping

```
#Computes and outputs travel time
#for a given speed and distance
#Written by J. Q. Programmer, 7/05/16

more = input("Do you want to plan a trip? (Y or N): ")
while more == "Y" or more == "y":
    speed = int(input("Enter your speed in mph: "))
    distance = float(input("Enter your distance in miles: "))
    print("Enter your choice of format for time")
    choice = input("decimal hours (D) or hours "\
       "and minutes (M): ")
    print()

    if choice == "D" or choice == "d":
        time = distance/speed
        print("At", speed, "mph, it will take")
        print(time, "hours to travel", distance, "miles.")
    else:
        time = distance/speed
        hours = int(time)
        minutes = int((time - hours)*60)
        print("At", speed, "mph, it will take")
        print(hours, "hours and", minutes, "minutes to travel",\
              distance, "miles.")
    more = input("\nDo you want to plan another trip? "\
       "(Y or N): ")

input("\n\nPress the Enter key to exit")
```

PRACTICE PROBLEMS

1. What is the output from the following section of code?

```
number1 = 15
number2 = 7
if number1 >= number2:
     print(2*number1)
else:
     print(2*number2)
```

2. What is the output from the following section of code?

```
scores = 1
while scores < 20:
    scores = scores + 2
    print(scores)
```

3. What is the output from the following section of code?

```
quotaThisMonth = 7
quotaLastMonth = quotaThisMonth + 1
if (quotaThisMonth > quotaLastMonth) or \
   (quotaLastMonth >= 8):
```

PRACTICE PROBLEMS

```
        print("Yes")
        quotaLastMonth = quotaLastMonth + 1

    else:

        print("No")
        quotaThisMonth = quotaThisMonth + 1
```

4. How many times is the output statement executed in the following section of code?

```
left = 10
right = 20
while left <= right:
    print(left)
    left = left + 2
```

5. Write a Python statement that outputs "Equal" if the integer values of *night* and *day* are the same, but otherwise does nothing.

4 Another Example

Let's briefly review the types of Python programming statements we've learned. We can do input and output—reading values from the user into memory, writing values out of memory for the user to see, being sure to use meaningful variable identifiers to reference memory locations. We can assign values to variables within the program. And we can direct the flow of control by using conditional statements or looping. Although there are many other statement types available in Python, you can do almost everything using only the modest collection of statements we've described. The power lies in how these statements are combined and nested within groups to produce ever more complex courses of action.

For example, suppose we write a program to assist SportsWorld, a company that installs circular swimming pools. In order to estimate their costs for swimming pool covers or for fencing to surround the pool, SportsWorld needs to know the area or circumference of a pool, given its radius. A pseudocode version of the program is shown in Figure 13.

We can translate this pseudocode fairly directly into Python code. We will also add a prologue comment to explain what the program does (optional but always recommended for program documentation). Also, the computations for circumference and area both involve the constant *pi* (π). We could use some numerical approximation for *pi* each time it occurs in the program. Instead we'll make use of a built-in Python module. A **module** is a collection of useful code that you can make available to your Python program by using the **import**

FIGURE 13

A Pseudocode Version of the SportsWorld Program

```
Get value for user's choice about continuing
While user wants to continue, do the following steps
    Get value for pool radius
    Get value for choice of task
    If task choice is circumference
        Compute pool circumference
        Print output
    Else (task choice is area)
        Compute pool area
        Print output
    Get value for user's choice about continuing
Stop
```

statement. In this case, the value of *pi* is defined in the math module, so we would put

```
import math
```

at the top of our program. Then the expression

```
math.pi
```

has the value assigned to *pi* in the math module, and

```
print(math.pi)
```

would produce

```
3.14159265359
```

As with all Python "constants," however, the value of math.pi can be changed in your program. Thus

```
math.pi = 7
print(math.pi)
```

would produce the value 7 as output. Again, it's up to the programmer to treat a value as a constant (unchangeable) if that's what it's supposed to be. Thankfully, the above assignment statement doesn't change the value of *pi* stored in the Python math module, it just changes it within the program in which it appears.

Figure 14 gives the complete program; the prologue comment notes the use of the math module. Figure 15 shows what actually appears on the screen when this program is executed with some sample data.

It is inappropriate (and messy) to output the value of the area to 10 or 11 decimal places based on a value of the radius given to one or two decimal places of accuracy. See Exercise 11 at the end of this module for decimal number formatting tips.

FIGURE 14

The SportsWorld Program

```
#This program helps SportsWorld estimate costs
#for pool covers and pool fencing by computing
#the area or circumference of a circle
#with a given radius.
#Any number of circles can be processed.
#Uses module math for pi
#Written by M. Phelps, 10/05/16

import math

print("Do you want to process a pool?")
more = input("Enter Y or N: ")

while (more == "Y") or (more == "y"): #more circles to process
    radius = float(input("Enter the value of the "\
             "radius of the pool: "))

    #See what user wants to compute
    print("\nEnter your choice of task.")
    taskToDo = input("C to compute circumference, "\
               "A to compute area: ")

    if taskToDo == "C": #compute circumference

       circumference = 2*math.pi*radius
       print("\nThe circumference for a pool of radius",\
             radius, "is", circumference)
    else:               #compute area

       area = math.pi * radius * radius
       print("\nThe area for a pool of radius",\
             radius, "is", area)

    print("\nDo you want to process more pools?")
    more = input("Enter Y or N: ")

#finish up
input("\n\nPress the Enter key to exit")
```

FIGURE 15

A Sample Session Using the Program of Figure 14

```
Do you want to process a pool?
Enter Y or N: Y
Enter the value of the radius of the pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius 2.7 is 16.9646003294
Do you want to process more pools?
Enter Y or N: Y
Enter the value of the radius of the pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: A
The area for a pool of radius 2.7 is 22.9022104447
```

FIGURE 15
A Sample Session Using the Program of Figure 14 (continued)

```
Do you want to process more pools?
Enter Y or N: Y
Enter the value of the radius of the pool: 14.53
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius 14.53 is 91.2946825133
Do you want to process more pools?
Enter Y or N: N
Press the Enter key to exit
```

PRACTICE PROBLEMS

1. Write a complete Python program to read in the user's first and last initials and write them out.
2. Write a complete Python program that asks for the price of an item and the quantity purchased, and writes out the total cost.
3. Write a complete Python program that asks for a number. If the number is less than 5, it is written out, but if it is greater than or equal to 5, twice that number is written out.
4. Write a complete Python program that asks the user for a positive integer *n*, and then writes out all the numbers from 1 up to and including *n*.

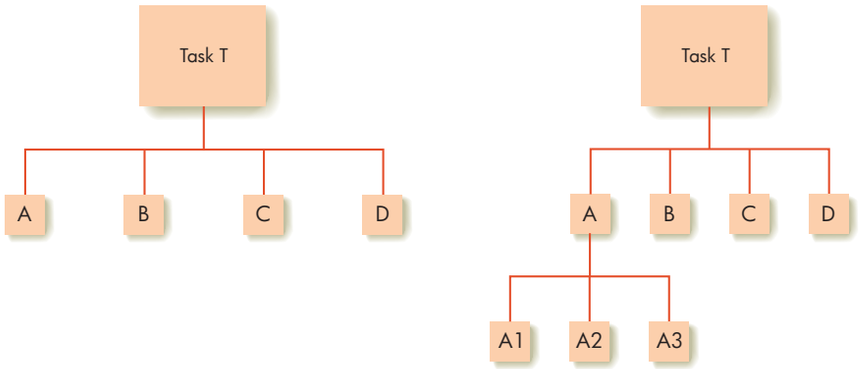
5 Managing Complexity

The programs we have written have been relatively simple. More complex problems require more complex programs to solve them. Although it is fairly easy to understand what is happening in the 30 or so lines of the SportsWorld program, imagine trying to understand a program that is 50,000 lines long. Imagine trying to write such a program! It is not possible to understand—all at once—everything that goes on in a 50,000-line program.

5.1 *Divide and Conquer*

Writing large programs is an exercise in managing complexity. The solution is a problem-solving approach called **divide and conquer**. Suppose a program is to be written to do a certain task; let's call it task T. Suppose further that we can divide this task into smaller tasks, say A, B, C, and D, such that, if we can do those four tasks in the right order, we can do task T. Then our high-level understanding of the problem need only be concerned with *what* A, B, C, and D do and how they must work together to accomplish T. We do not, at this stage, need to understand *how* tasks A, B, C, and D can be done. Figure 16(a), an

FIGURE 16
Structure Charts



(a) Basic decomposition

(b) More detailed decomposition

example of a **structure chart** or **structure diagram**, illustrates this situation. Task T is composed in some way of subtasks A, B, C, and D. Later we can turn our attention to, say, subtask A, and see if it too can be decomposed into smaller subtasks, as in Figure 16(b). In this way, we continue to break the task down into smaller and smaller pieces, finally arriving at subtasks that are simple enough that it is easy to write the code to carry them out. By *dividing* the problem into small pieces, we can *conquer* the complexity that is overwhelming if we look at the problem as a whole.

Divide and conquer is a problem-solving approach, and not just a computer programming technique. Outlining a term paper into major and minor topics is a divide-and-conquer approach to writing the paper. Doing a Form 1040 Individual Tax Return for the Internal Revenue Service can involve subtasks of completing Schedules A, B, C, D, and so on, and then reassembling the results. Designing a house can be broken down into subtasks of designing floor plans, wiring, plumbing, and the like. Large companies organize their management responsibilities using a divide-and-conquer approach; what we have called structure charts become, in the business world, organization charts.

How is the divide-and-conquer problem-solving approach reflected in the resulting computer program? If we think about the problem in terms of subtasks, then the program should show that same structure; that is, part of the code should do subtask A, part should do subtask B, and so on. We divide the code into *subprograms*, each of which does some part of the overall task. Then we empower these subprograms to work together to solve the original problem.

5.2 *Using and Writing Functions*

In Python, subprograms are called **functions**. Each function in a program should do one and only one subtask. Data get passed back and forth between the "main" section of the program and various functions. The main part may pass data to a function, receive new data from a function, or both. Data received from a function could in turn be passed on to another function. You can imagine data flowing along the connecting lines in the structure chart. That's how we "empower these subprograms to work together."

The Force Is with Them

Filmmaker George Lucas started Industrial Light and Magic in 1975 to create the special effects he wanted for the original *Star Wars* movie. Since then, ILM has contributed to the entire *Star Wars* series, *Jurassic Park*, *Raiders of the Lost Ark*, and many other hit films full of amazing special effects. Many of the original special effects were done with miniature models, but by the 1980s computer graphics was taking over the special effects world.

A single frame of computer-generated film can require coordination of perhaps hundreds of software components. ILM turned to Python to create software to manage the flow of the various pieces—including thousands of images—needed for a complex and rapid production process. Over time, Python has assumed an ever-larger role in production process management. It has also been used to create a user interface for computer graphic artists to access various elements at their disposal. And, it supports a network-wide whiteboard and instant messaging system for discussions in daily shot reviews. Python is also integrated with custom-built C and C++ code that supports the in-house lighting tool used to place light sources into a 3-D scene and preview shadings and surfaces.

We've already used some built-in Python functions. The statement

```
print("Hello World")
```

passes a literal string to the *print* function, which then writes it out. The statement

```
speed = input("Enter your speed in mph: ")
```

uses the *input* function. A literal string is passed into this function. The function's job is to print this literal string, pick up the string the user types in response, and return that new string, which then gets assigned to the variable *speed*. We also used the *int* function:

```
speed = int(speed)
```

This statement passes the string value *speed* to the *int* function; the function type casts this string to an integer value and returns that value, which is then assigned to *speed*. Then we got a little fancier:

```
speed = int(input("Enter your speed in mph: "))
```

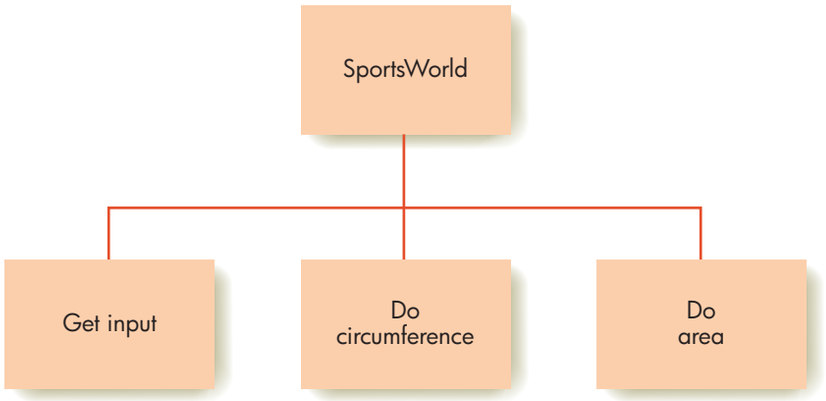
Here the literal string gets passed to the *input* function, and the string value returned by the *input* function is passed directly into the *int* function, whose returned value is then assigned to *speed*.

Let's review the SportsWorld program with an eye to further subdividing the task. There is a loop that does some operations as long as the user wants. What gets done? Input is obtained from the user about the radius of the circle and the choice of task to be done (compute circumference or compute area). Then the circumference or the area gets computed and written out.

We've identified three subtasks, as shown in the structure chart of Figure 17. We can visualize the main section of the program at a pseudocode level, as shown in Figure 18. This divide-and-conquer approach to solving the problem can (and should) be planned first in pseudocode, without regard to the details of the

FIGURE 17

Structure Chart for the SportsWorld Task



programming language to be used. If the three subtasks (input, circumference, area) can all be done, then arranging them within the structure of Figure 18 solves the problem. We can write a function for each of the subtasks. Although we now know what form the main section will take, we have pushed the details of how to do each of the subtasks off into the other functions. Execution of the program begins with the main section. Every time the flow of control reaches the equivalent of a "do subtask" instruction, it transfers execution to the appropriate function code. When execution of the function code is complete, flow of control returns to the main section and picks up where it left off.

We'll start with functions for the circumference and area subtasks. Functions are named using ordinary Python identifiers, so we'll name these functions *doCircumference* and *doArea*. Because we're using meaningful identifiers, it is obvious which subtask is carried out by which function.

A simple function in Python has the following form:

```
def function identifier():
        body of the function
```

The notation "def" says that a function is about to be defined. As we saw with the *if* statement and the *while* statement, the colon and the indentation identify the block of code that is the function body. The *doCircumference* function can be written as

```
def doCircumference():
   circumference = 2*math.pi*radius
   print("\nThe circumference for a pool of radius",\
         radius, "is", circumference)
```

FIGURE 18

A High-Level Modular View of the SportsWorld Program

```
Get value for user's choice about continuing
While the user wants to continue
  Do the input subtask
  If (Task = 'C') then
    do the circumference subtask
  else
    do the area subtask
Get value for user's choice about continuing
```

and the *doArea* function is similar. Figure 19 shows the complete program at this point. The two function bodies are the same statements that previously appeared in the main section of the program. Where these statements used to be, there are now function invocations that transfer control to the function code:

```
doCircumference()
```

and

```
doArea()
```

FIGURE 19

A Modularized SportsWorld Program, Version 1

```
#This program helps SportsWorld estimate costs
#for pool covers and pool fencing by computing
#the area or circumference of a circle
#with a given radius.
#Any number of circles can be processed.
#Uses module math for pi
#Uses simple functions and global variables
#Written by M. Phelps, 10/15/16

import math

def doCircumference():
    circumference = 2*math.pi*radius
    print("\nThe circumference for a pool of radius",\
          radius, "is", circumference)

def doArea():
    area = math.pi * radius * radius
    print("\nThe area for a pool of radius",\
          radius, "is", area)

#main section
print("Do you want to process a pool?")
more = input("Enter Y or N: ")

while(more == "Y") or (more == "y"):  #more circles to process

    radius = float(input("Enter the value of the "\
            "radius of the pool: "))

    #See what user wants to compute
    print("\nEnter your choice of task.")
    taskToDo = input("C to compute circumference, "\
           "A to compute area: ")

    if taskToDo == "C":       #compute circumference
        doCircumference()
    else:                     #compute area
        doArea()

    print("\nDo you want to process more pools?")
    more = input("Enter Y or N: ")

#finish up
input("\n\nPress the Enter key to exit")
```

The *doCircumference* and *doArea* functions obviously need to make use of the *radius* variable. These functions know the value of radius because radius was a variable in the main section of the program, which makes it a **global variable**, known throughout the program, even inside a function body. In general, a function should only have access to the information it needs to do its particular subtask, lest it inadvertently change the value of some variable about which it has no business even knowing. Python solves this problem because the value of a global variable can be used, but it can't easily be changed within a function by a simple assignment statement, as we will see next.

Let's try a function for the third subtask, getting input. We might try

```
#DOES NOT WORK
def getInput():
    radius = float(input("Enter the value of the "\
            "radius of the pool: "))


    #See what user wants to compute
    print("\nEnter your choice of task.")
    taskToDo = input("C to compute circumference, "\
               "A to compute area: ")
```

In the main section, before invoking this function, we would have to create the global variables *radius* and *taskToDo*. We can give them dummy values because they should get their real values within the *getInput* function. Figure 20 shows this version, which DOES NOT WORK. The result of running this program will give 0 for the circumference and the area, no matter what is entered for the radius. Here's why. The statement

```
radius = float(input("Enter the value of the "\
         "radius of the pool: "))
```

creates a new variable called *radius* (and assigns it the value supplied by the user). A variable created within a function is a **local variable** and is not known anywhere else in the program. In other words, this particular *radius* variable has nothing to do with the original *radius* variable, and it's this local variable whose value is being set. After execution of the function is complete, this local variable disappears. The original *radius* variable (whose value has remained 0 all this time) is what the *doCircumference* and *doArea* functions use. In fact, the program doesn't even use the *doCircumference* function because, like the global *radius* variable, *taskToDo* still has its original value ('A') once the *getInput* function exits, so it's always the area that is computed.

We need to find some way to allow the *getInput* function the ability to change the original *radius* and *taskToDo* values. To do this, we make use of the *return* statement, whose syntax is

```
return expression list
```

FIGURE 20

A Modularized SportsWorld Program, Version 2
THIS DOES NOT WORK

```
#This program helps SportsWorld estimate costs
#for pool covers and pool fencing by computing
#the area or circumference of a circle
#with a given radius.
#Any number of circles can be processed.
#Uses module math for pi
#Uses simple functions and global variables
#Written by M. Phelps, 10/15/16
#THIS VERSION DOES NOT WORK

import math

def getInput():
    radius = float(input("Enter the value of the "\
            "radius of the pool: "))

    #See what user wants to compute
    print("\nEnter your choice of task.")
    taskToDo = input("C to compute circumference, "\
               "A to compute area: ")

    #return radius, taskToDo

def doCircumference():
    circumference = 2*math.pi*radius
    print("\nThe circumference for a pool of radius",\
              radius, "is", circumference)

def doArea():
    area = math.pi * radius * radius
    print("\nThe area for a pool of radius",\
              radius, "is", area)

#main section
print("Do you want to process a pool?")
more = input("Enter Y or N: ")

while(more == "Y") or (more == "y"): #more circles to process

    radius = 0
    taskToDo = "A"

    getInput()

    if taskToDo == "C": #compute circumference
        doCircumference()
    else:               #compute area
        doArea()

    print("\nDo you want to process more pools?")
    more = input("Enter Y or N: ")

#finish up
input("\n\nPress the Enter key to exit")
```

The expression list is a list, separated by commas, of expressions for values to be "returned" to the statement that invoked the function. This statement should be an assignment statement, with the function invocation on the right side and a list of variables on the left side that will receive, in order, the values returned. Figure 21 shows a new version of the SportsWorld program where the *getInput* function returns values for the *radius* and *taskToDo* variables. We've used new names within the *getInput* function to emphasize that the values computed there (in this case, the values are simply obtained from the user) are values for local variables. Within the main section, the key statement is

```
radius, taskToDo = getInput( )
```

Here the *getInput* function is invoked on the right side of the assignment statement, and the values returned by *getInput* are assigned, in order, to the *radius* and *taskToDo* variables:

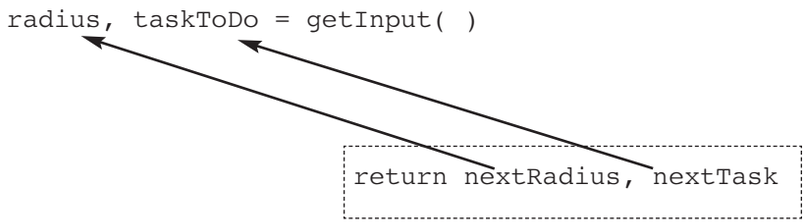


FIGURE 21
A Modularized SportsWorld Program, Version 3

```
#This program helps SportsWorld estimate costs
#for pool covers and pool fencing by computing
#the area or circumference of a circle
#with a given radius.
#Any number of circles can be processed.
#Uses module math for pi
#Uses return statement
#Written by M. Phelps, 10/15/16

import math

def getInput( ):
    nextRadius = float(input("Enter the value of the "\
                 "radius of the pool: "))

    #See what user wants to compute
    print("\nEnter your choice of task.")
    nextTask = input("C to compute circumference, "\
               "A to compute area: ")
    return nextRadius, nextTask

def doCircumference():
    circumference = 2*math.pi*radius
    print("\nThe circumference for a pool of radius",\
              radius, "is", circumference)
```

FIGURE 21

A Modularized SportsWorld Program, Version 3 (continued)

```
def doArea():
    area = math.pi * radius * radius
    print("\nThe area for a pool of radius",\
              radius, "is", area)

#main section
print("Do you want to process a pool?")
more = input("Enter Y or N: ")

while (more == "Y") or (more == "y"): #more circles to process

    radius, taskToDo = getInput( )

    if taskToDo == "C":            #compute circumference
       doCircumference()
    else:                          #compute area
       doArea()

    print("\nDo you want to process more pools?")
    more = input("Enter Y or N: ")

#finish up
input("\n\nPress the Enter key to exit")
```

A return statement with an empty expression list would simply cause an exit from the function in which it appears.

Believe it or not, we are still not quite happy with the modularized version of the SportsWorld program. It's this use of global variables that is troublesome. Within the *doCircumference* and *doArea* functions, the radius variable just seems to pop up "unannounced" by the somewhat back-door route of being a global variable. Even though these functions can't change the value of *radius* by just assigning it a new value (which would be an undesirable **side effect**), it seems that if a function needs to know (that is, use) the value of a variable, it should explicitly "receive" that value. In fact, it would be good practice to get rid of global variables altogether.

To explicitly pass values to a function, we need to use the more general definition of a Python function, which has the form

```
def function identifier(parameter list):
        body of the function
```

The invocation of a function with parameters requires giving the name of the function followed by an **argument list** that will pass values to the function that are pertinent to that function's task. An argument list can contain either variables that have already been assigned a value or literal expressions such as 2 + 3. The **parameter list** is a list of variables local to the function that will receive their values from the corresponding argument list when the function is invoked. The parameters in the parameter list correspond by position to the arguments in the statement that invokes this function; that is, the first parameter in the list matches the first argument given in the statement that

invokes the function, the second parameter matches the second argument, and so on. It is through this correspondence between arguments and parameters that information (data) flows into a function. Parameter names and argument names need not be the same; the name of the parameter is what is used inside the function, but it is the correspondence with the argument list that matters, not the parameter identifier used.

The *doCircumference* function, as noted earlier, needs to know the value of the radius. We'll give the *doCircumference* function a single parameter, and when we invoke this function we'll pass *radius* as the single argument. Of course this is pointless if radius remains a global variable. We'll eliminate global variables by making a *main* function in which we'll put, more or less, the code that has been in the main section. Then the new main section will just invoke the *main* function. Figure 22 shows the final modularized version of SportsWorld. The change in the *main* function is to invoke the *doCircumference* and *doArea* functions by passing the single argument *radius* to each function. In addition, the parameter name for *doArea* has been set to something different from "radius", just to demonstrate that parameter names and argument names need not agree.

We now see that the statement

```
speed = input("Enter your speed in mph: ")
```

passes a literal string argument to the *input* function. Although we haven't seen the code for the built-in *input* function, we can tell that it uses one parameter.

FIGURE 22

A Modularized SportsWorld Program, Version 4

```
#This program helps SportsWorld estimate costs
#for pool covers and pool fencing by computing
#the area or circumference of a circle
#with a given radius.
#Any number of circles can be processed.
#Uses module math for pi
#Uses parameters and return statements
#Written by M. Phelps, 10/16/16

import math

def getInput( ):
    nextRadius = float(input("Enter the value of the "\
                 "radius of the pool: "))

    #See what user wants to compute
    print("\nEnter your choice of task.")
    nextTask = input("C to compute circumference, "\
               "A to compute area: ")
    return nextRadius, nextTask

def doCircumference(radius):
    circumference = 2*math.pi*radius
    print("\nThe circumference for a pool of radius",\
              radius, "is", circumference)
```

FIGURE 22

A Modularized SportsWorld Program, Version 4 (continued)

```
def doArea(theRadius):
    area = math.pi * theRadius * theRadius
    print("\nThe area for a pool of radius",\
             theRadius, "is", area)

def main():

    print("Do you want to process a pool?")
    more = input("Enter Y or N: ")

    while(more == "Y") or (more == "y"):
    #more circles to process
        radius, taskToDo = getInput( )

        if taskToDo == "C":    #compute circumference
           doCircumference(radius)
        else:                  #compute area
           doArea(radius)

        print("\nDo you want to process more pools?")
        more = input("Enter Y or N: ")

#program starts here
main()

#finish up
input("\n\nPress the Enter key to exit")
```

We've now seen several variations of how a function might use data. Assuming that we have isolated a *main* function to avoid any global variables, we can identify four different situations. Figure 23 describes these, and Figure 24 shows a small Python program that illustrates each case. Figure 25 shows the resulting output. A given function could mix and match these cases; for example,

FIGURE 23

Data Flow in and out of Python Functions

Direction of Data Flow	What It Means	How It's Accomplished
None	Function is a "constant" function that does the same thing every time, and needs no data nor does it create new data.	No parameter, no return statement
In only	Function needs to use, but not change, this value.	Pass the value as an argument to a parameter
Out only	Function constructs a new value that the invoking function needs to know.	No parameter; use a local variable to construct the value and send it back via a return statement
In-Out	Function needs to use and also change this value.	Pass the value as an argument to a parameter; send the changed value back via a return statement

FIGURE 24

Parameter Passing and Return Statements

```
#illustrates data flow

def message():
    #message is a "constant" function
    #it always does the same thing
    #it needs no data and returns no data
    print("This program demonstrates data flow "\
          " into and out of functions.")
def inOnly(x):
    #inOnly receives a value through its
    #single parameter and prints that value
    print("the value of x is ", x)

def outOnly():
    #outOnly creates a new value
    #and sends it back via the return statement
    #where it then gets assigned to a variable
    #and printed out
    newValue = 17
    return newValue

def inOut(x):
    #inOut receives a value through its
    #single parameter, changes that value
    #and sends it back via the return statement
    #where it gets used directly as part of an
    #output statement
    x = 2*x;
    return x

def main():
    message()
    inOnly(5)
    y = outOnly()
    print("the value of y is", y)
    y = inOut(y)
    print("the value of y is", y)

#program starts here
main()

input("\n\nPress the Enter key to exit")
```

FIGURE 25

Output from the Program of Figure 24

```
This program demonstrates data flow into and out of functions.
the value of x is  5
the value of y is 17
the value of y is 34
```

it might need to use a value (which would be an "in-only" case requiring a parameter) and also return some new value (which would be an "out-only" case requiring a return statement). It is helpful to plan ahead of time how data should flow in and out of each function.

Because it seems to have been a lot of effort to arrive at the complete, modularized version of our SportsWorld program, shown in Figure 22 (which, after all, does the same thing as the program in Figure 14), let's review why this effort is worthwhile.

The modularized version of the program is compartmentalized in two ways. First, it is compartmentalized with respect to task. The major task is accomplished by a series of subtasks, and the work for each subtask takes place within a separate function. This leaves the main function free of details and consisting primarily of invoking the appropriate function at the appropriate point. As an analogy, think of the president of a company calling on various assistants to carry out tasks as needed. The president does not need to know *how* a task is done, only the name of the person responsible for carrying it out. Second, the program is compartmentalized with respect to data, in the sense that the data values known to the various functions are controlled by parameter lists. In our analogy, the president gives each assistant the information he or she needs to do the assigned task, and expects relevant information to be returned—but not all assistants know all information.

This compartmentalization is useful in many ways. It is useful when we *plan the solution* to a problem, because it allows us to use a divide-and-conquer approach. We can think about the problem in terms of subtasks. This makes it easier for us to understand how to achieve a solution to a large and complex problem. It is also useful when we *code the solution* to a problem, because it allows us to concentrate on writing one section of the code at a time. We can write a function and then fit it into the program, so that the program gradually expands rather than having to be written all at once. Developing a large software project is a team effort, and different parts of the team can be writing different functions at the same time. It is useful when we *test the program*, because we can test one new function at a time as the program grows, and any errors are localized to the function being added. (The *main* function can be tested early by writing appropriate headers but empty or dummy bodies for the remaining functions.) Compartmentalization is useful when we *modify the program*, because changes tend to be within certain subtasks and hence within certain functions in the code. And finally it is useful for anyone (including the programmer) who wants to *read* the resulting program. The overall idea of how the program works, without the details, can be gleaned from reading the *main* function; if and when the details become important, the appropriate code for the other functions can be consulted. In other words, modularizing a program is useful for its

- Planning
- Coding
- Testing
- Modifying
- Reading

Finally, once a function has been developed and tested, it is then available for any application program to use. An application program that does quite different things than SportsWorld, but that needs the value of the area or circumference of a circle computed from the radius, can use our *doCircumference* and *doArea* functions.

PRACTICE PROBLEMS

1. What is the output of the following Python program?

```
def Helper():
    number = 15

number = 10
Helper()
print("number has the value" , number)
```

2. What is the output of the following Python program?

```
def Helper():
    newNumber = 15
    return newNumber

def main():
    number = 10
    print("number has the value" , number)
    number = Helper()
    print("number has the value" , number)

#program starts here
main()
```

3. What is the output of the following Python program?

```
def Helper(number):
    number = number + 15
    return number

def main():
    number = 10
    print("number has the value" , number)
    number = Helper(number)
    print("number has the value" , number)

#program starts here
main()
```

4. **a.** Change the *doCircumference* function from Figure 22 so that instead of computing and printing out the value of the circumference, it computes the value and returns it to the main function.

   b. Change the *main* function in Figure 22 so that it prints out the value of the circumference returned by the function of part (a).

6 Object-Oriented Programming

6.1 *What Is It?*

The divide-and-conquer approach to programming is a "traditional" approach. The focus is on the overall task to be done: how to break it down into subtasks, and how to write algorithms for the various subtasks that are carried out by communicating subprograms (in the case of Python, by functions). The program can be thought of as a giant statement executor designed to carry out the major task, even though the main function may simply call on, in turn, the various other functions that do the subtask work.

Object-oriented programming (OOP) takes a somewhat different approach. A program is considered a simulation of some part of the world that is the domain of interest. "Objects" populate this domain. Objects in a banking system, for example, might be savings accounts, checking accounts, and loans. Objects in a company personnel system might be employees. Objects in a medical office might be patients and doctors. Each object is an example drawn from a class of similar objects. The savings account "class" in a bank has certain properties associated with it, such as name, Social Security number, account type, and account balance. Each individual savings account at the bank is an example of (an object of) the savings account class, and each has specific values for these common properties; that is, each savings account has a specific value for the name of the account holder, a specific value for the account balance, and so forth. Each object of a class therefore has its own data values.

A class also has one or more subtasks associated with it, and all objects from that class can perform those subtasks. In carrying out a subtask, each object can be thought of as providing some service. A savings account, for example, can compute compound interest due on the balance. When an object-oriented program is executed, the program generates requests for services that go to the various objects. The objects respond by performing the requested service—that is, carrying out the subtask. Thus, a program that is using the savings account class might request a particular savings account object to perform the service of computing interest due on the account balance. An object always knows its own data values and may use them in performing the requested service.

There are three terms often associated with object-oriented programming, as illustrated in Figure 26. The first term is **encapsulation**. Each class has its own subprogram to perform each of its subtasks. Any user of the class (which might be some other program) can ask an object of that class to invoke the appropriate subprogram and thereby perform the subtask service. The class user needs to know what services objects of the class can provide and how to request an object to perform any such service. The details of the subprogram code belong to the class itself, and this code may be modified in any manner, as long as the way the user interacts with the class remains unchanged. (In the savings account example, the details of the algorithm used to compute interest due belong only to the class, and need not be known by any user of the class. If the bank wants to change how it computes interest, only the code for the interest method in the savings account class needs to be modified; any programs that use the services of the savings account class can remain

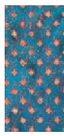
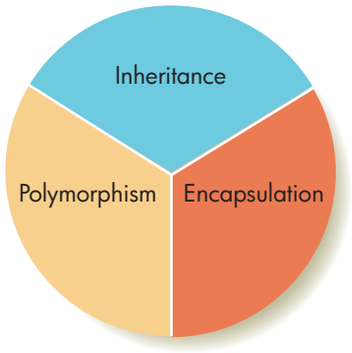


FIGURE 26

Three Key Elements of OOP



unchanged.) Furthermore, the class properties represent data values that will exist as part of each object of the class. A class therefore consists of two components, its properties and its subprograms, and both components are "encapsulated"—bundled—with the class.

A second term associated with object-oriented programming is **inheritance**. Once a class A of objects is defined, a class B of objects can be defined as a "subclass" of A. Every object of class B is also an object of class A; this is sometimes called an "is a" relationship. Objects in the B class "inherit" all of the properties of objects in class A and are able to perform all the services of objects in A, but they may also be given some special property or ability. The benefit is that class B does not have to be built from the ground up, but rather can take advantage of the fact that class A already exists. In the banking example, a senior citizens savings account would be a subclass of the savings account class. Any senior citizens savings account object is also a savings account object, but may have special properties or be able to provide special services.

The third term is **polymorphism**. *Poly* means "many." Objects may provide services that should logically have the same name because they do roughly the same thing, but the details differ. In the banking example, both savings account objects and checking account objects should provide a "compute interest" service, but the details of how interest is computed differ in these two cases. Thus, one name, the name of the service to be performed, has several meanings, depending on the class of the object providing the service. It may even be the case that more than one service with the same name exists for the same class, although there must be some way to tell which service is meant when it is invoked by an object of that class.

Let's change analogies from the banking world to something more fanciful, and consider a football team. Every member of the team's backfield is an "object" of the "backfield" class. The quarterback is the only "object" of the "quarterback" class. Each backfield object can perform the service of carrying the ball if he (or she) receives the ball from the quarterback; ball carrying is a subtask of the backfield class. The quarterback who hands the ball off to a backfield object is requesting that the backfield object perform that subtask because it is "public knowledge" that the backfield class carries the ball and that this service is invoked by handing off the ball to a backfield object. The "program" to carry out this subtask is *encapsulated* within the backfield class, in the sense that it may have evolved over the week's practice and may depend on specific knowledge of the opposing team, but at any rate, its details need

not be known to other players. *Inheritance* can be illustrated by the halfback subclass within the backfield class. A halfback object can do everything a backfield object can but may also be a pass receiver. And *polymorphism* can be illustrated by the fact that the backfield may invoke a different "program" depending on where on the field the ball is handed off. Of course our analogy is imperfect, because not all human "objects" from the same class behave in precisely the same way—fullbacks sometimes receive passes and so on.

6.2 *Python and OOP*

How do these ideas get translated into real programs? The details, of course, vary with the programming language used, and not every language supports object-oriented programming. Python, however, does support object-oriented programming. When we write a class, we specify the properties (called **attributes** or **data attributes** in Python) common to any object of that class. We also specify the services that any object of that class can perform. Services (subtasks) are implemented as subprograms and, as we know, subprograms in Python are functions. Functions associated with a class are called **methods** to distinguish them from the "regular" functions we talked about in the previous section.

Let's rewrite the SportsWorld program one more time, this time as an object-oriented program. What are the objects of interest within the scope of this problem? SportsWorld deals with circular swimming pools, but they are basically just circles. So let's create a *Circle* class, and have the SportsWorld program create objects of (**instances of**) that class. The objects are individual circles. A Circle object has a radius. A Circle object, which knows the value of its own radius, should be able to perform the services of computing its own circumference and its own area. At this point, we are well on the way to answering the two major questions about the *Circle* class:

- What are the attributes common to any object of this class? (In this case, there is a single attribute—the radius.)
- What are the services that any object of the class should be able to perform? (In this case, it should be able to compute its circumference and compute its area, although as we will see shortly, we will need other services as well.)

Figure 27 shows the complete object-oriented version of SportsWorld. There are three major sections to this program. At the top is the class definition for the *Circle* class; below that is the *getInput* function (identical to what it looked like in the previous version) and then the *main* function.

A class definition in Python has the following form:

```
class class_identifier:
        body of the class
```

As usual, the opening : and the indentation define the scope of the class definition. The class identifier can be any Python identifier. The body of the class consists of the definitions of all the class methods—that is, code for the services that any object of this class can perform. This is part of what we expect to see for a class, but what about the attributes? Just like other

FIGURE 27

An Object-Oriented SportsWorld Program

```
#This program helps SportsWorld estimate costs
#for pool covers and pool fencing by computing
#the area or circumference of a circle
#with a given radius.
#Any number of circles can be processed.
#Uses module math for pi
#Uses class Circle
#Written by I. M. Euclid, 10/23/16

import math

class Circle:

    def __init__(self, value = 0):
        #initializes the one attribute of radius to have value 0
        self.__radius = value

    def setRadius(self, value):
        #set the value of radius attribute
        self.__radius = value

    def getRadius(self):
        #return the value of radius attribute
        return self.__radius

    def doCircumference(self):
        #compute and return circumference
        return 2 * math.pi*self.__radius

    def doArea(self):
        #compute and return area
        return math.pi * self.__radius * self.__radius
#end of class definition

def getInput( ):
    nextRadius = float(input("Enter the value of the "\
                 "radius of the pool: "))

    #See what user wants to compute
    print("\nEnter your choice of task.")
    nextTask = input("C to compute circumference, "\
         "A to compute area: ")
    return nextRadius, nextTask

def main():

    print("Do you want to process a pool?")
    more = input("Enter Y or N: ")

    swimmingPool = Circle()              #create a Circle object
    while(more == "Y") or (more == "y"): #more circles to
                                         #process
        radius, taskToDo = getInput( )
        swimmingPool.setRadius(radius)
```

FIGURE 27

An Object-Oriented SportsWorld Program (continued)

```
        if taskToDo == "C":             #compute circumference
            print("\nThe circumference for a pool of radius",\
              swimmingPool.getRadius(), "is",\
              swimmingPool.doCircumference())
        else:                           #compute area
            print("\nThe area for a pool of radius",\
              swimmingPool.getRadius(), "is",\
              swimmingPool.doArea())

        print("\nDo you want to process more pools?")
        more = input("Enter Y or N: ")

#program starts here
main()

#finish up
input("\n\nPress the Enter key to exit")
```

Python variables, data attributes spring into being simply by having a value assigned to them. However, it is customary to initialize attributes all at once in an "initialization" method. This does two things: it creates all the attribute variables in one place so the reader of the code can see what they are, and it assigns initial values to these variables. Fortunately, Python provides an initialization method called *__init__* that is automatically invoked whenever an object of the class is created. We'll discuss its details shortly. Our *Circle* class has five methods, counting the initialization method.

An object of a class is created by a statement of the form

object_identifier = class_identifier()

so in the main function of our SportsWorld program we create a Circle object called *swimmingPool* by the statement

```
swimmingPool = Circle()
```

Methods of the Circle class can only be invoked by an object of that class, using the syntax

object_identifier.method_identifier(argument list)

For example, we see in the *main* function that the *swimmingPool* object invokes the *doCircumference* function by

```
swimmingPool.doCircumference()
```

From this invocation, it appears that the *doCircumference* method has no parameters because no arguments are passed. Yes and no—we've lost one parameter that we had in the previous version, but we've gained a "hidden" parameter. What have we lost? We don't have to pass the *radius* value to the *doCircumference* function because, as a method of the class, that function is invoked by an object, and the object carries its data values with it. The only

thing *doCircumference* has to know is what the invoking object is. As we can see from the statement

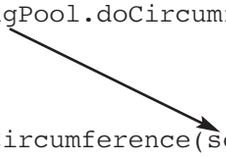
```
swimmingPool.doCircumference()
```

the invoking object is *swimmingPool*. Every method of a class has a parameter called "self" that corresponds to the invoking object. The *doCircumference* function is defined in the *Circle* class as follows:

```
def doCircumference(self):
    #compute and return circumference
    return 2 * math.pi*self.__radius
```

The "self" parameter automatically picks up the calling object as its argument value, even though the argument list of the method invocation is empty.

```
swimmingPool.doCircumference()
```



```
def doCircumference(self):
```

so that

```
self.__radius
```

is actually

```
swimmingPool.__radius
```

(We'll say more momentarily about the double underscore in front of "radius".) If there were additional parameters (besides *self*) in the method, there would be corresponding arguments in the method invocation.

All of the class methods are **public**, meaning they can be invoked from anywhere in the program (by an object of the class, of course). In fact, although we won't go into the details, one could create a module (just a file containing the *Circle* class definition), and then any Python program could use this class with the appropriate *import* statement. Think of the *Circle* class as handing out a business card that advertises these services: Hey, you want a Circle object that can find its own area? Find its own circumference? Set the value of its own radius? I'm your class!

Now consider the class attributes. To refer to an attribute of an object, the syntax is

```
object_identifier.attribute_identifier
```

which would logically be

```
swimmingPool.radius
```

in our case or, if within a method,

```
self.radius
```

Without some further effort, attributes of a class are also public, meaning that they can be used or changed anywhere in the program by statements such as

```
swimmingPool.radius = 17   or   self.radius = 17
```

Recall that we didn't like global variables being available everywhere, even when they couldn't be changed within a function. That's why we made a *main* function. Similarly, we don't like the idea that any function in the program can willy-nilly decide to change the attributes of an object. Python allows us to create attributes that are (almost) **private** by preceding the attribute name with a double underscore. Thus, Circle objects in our program don't have a radius attribute, they have an attribute named *__radius*. Declaring an attribute in this fashion means that within any method of the class, the attribute can be referred to directly by

```
self.attribute_identifier
```

This explains the statement

```
return 2 * math.pi*self.__radius
```

in the *doCircumference* method. But if you try a statement such as

```
print(swimmingPool.__radius) #will produce error message
```

in the *main* function, you will get an error message from the interpreter that says " AttributeError: Circle instance has no attribute '__radius' ". Of course it does have such an attribute, but it's not directly available for use outside of a *Circle* class method.[4]

The rest of the program must rely on the *swimmingPool* object to invoke the *getRadius* and *setRadius* methods to reveal the current value of its radius and change the value of its radius.

This one-and-only attribute variable *__radius* is created within the *__init__* method. Again, this method is automatically invoked whenever a new object is created. The definition is

```
def __init__(self, value = 0):
  #initializes the one attribute of radius
  #to have value 0
  self.__radius = value
```

This method has the usual *self* parameter, plus a second parameter called *value*. If we had created the swimmingPool object within the main function by a statement such as

```
swimmingPool = Circle(25)
```

[4]The attribute is "almost" private because it can actually be referenced anywhere by swimmingPool_Circle__radius. This is considered sufficiently obscure as to provide adequate protection against unintentional misuse of the attribute variable.

then the value of *__radius* would be initialized to 25. Because in Figure 27 we passed no argument, the "default" value set in the parameter list is used, and our *swimmingPool* object has a radius of 0 until the *getInput* function returns another value for the radius from the user.

6.3 *One More Example*

The object-oriented version of our SportsWorld program illustrates encapsulation. All data and calculations concerning circles are encapsulated in the *Circle* class. Let's look at one final example that illustrates the other two watchwords of OOP—polymorphism and inheritance.

In Figure 28 the domain of interest is that of geometric shapes. Four different classes are defined: *Circle*, *Rectangle*, *Square*, and *Square2*.

FIGURE 28
A Python Program with Polymorphism and Inheritance

```
import math

class Circle:
    #class for circles. Area can be computed
    #from radius

    def __init__(self, value = 0):
       #initializes radius attribute to value
       self.__radius = value

    def setRadius(self, value):
       #set the value of radius attribute
       self.__radius = value

    def getRadius(self):
       #return the value of radius attribute
       return self.__radius

    def doArea(self):
       #compute and return area
       return math.pi * self.__radius * self.__radius
#end of Circle class definition

class Rectangle:
    #class for rectangles. Area can be computed
    #from height and width

    def __init__(self, h, w):
       #initializes height and width attributes
       self.__height = h
       self.__width = w

    def setWidth(self, value):
       #set the value of width attribute
       self.__width = value

    def setHeight(self, value):
       #set the value of height attribute
       self.__height = value
```

FIGURE 28

A Python Program with Polymorphism and Inheritance (continued)

```
    def getWidth(self):
        #return the value of width attribute
        return self.__width

    def getHeight(self):
        #return the value of height attribute
        return self.__height

    def doArea(self):
        #compute and return area
        return self.__height * self.__width
#end of Rectangle class definition

class Square:
    #class for squares. Area can be computed
    #from side

    def __init__(self, value):
        #initializes side attribute
        self.__side = value

    def setSide(self, value):
        #set the value of side attribute
        self.__side = value

    def getSide(self):
        #return the value of side attribute
        return self.__side

    def doArea(self):
        #compute and return area
        return self.__side * self.__side
#end of Square class definition

class Square2(Rectangle):
    #Square2 is derived class of Rectangle,
    #uses the inherited height and width
    #attributes and the inherited doArea method

    def __init__(self, value):
        #initializes height and width attributes
        #must explicitly invoke parent class constructor
        Rectangle.__init__(self, value, value)

    def setSide(self, value):
        #set the value of side attribute
        self.__width = value
        self.__height = value
#end of Square2 class definition

def main():
    joe = Circle(23.5)
    print("The area of a circle with radius",\
```

FIGURE 28

A Python Program with Polymorphism and Inheritance (continued)

```
            joe.getRadius(), "is",\
            joe.doArea())

      luis = Rectangle(12.4, 18.1)
      print("The area of a rectangle with height",\
            luis.getHeight(), "and width",\
            luis.getWidth(), "is",\
            luis.doArea())

      anastasia = Square(3)
      print("The area of a square with side",\
            anastasia.getSide(), "is",\
            anastasia.doArea())

      tyler = Square2(4.2)
      print("The area of a square with side",\
            tyler.getWidth(), "is",\
            tyler.doArea())

#program starts here
main()

#finish up
input("\n\nPress the Enter key to exit")
```

Each class includes an initialization method in which the attribute variables for objects of that class are created as private variables. A Circle object has a radius attribute, whereas a Rectangle object has a width attribute and a height attribute. Other methods are defined for the services or subtasks that an object from the class can perform. Any Circle object can set the value of its radius and can compute its area. A Square object has a *side* attribute, as one might expect, but a Square2 object doesn't seem to have any attributes, although it has an initialization method, nor does it seem to have any way to compute its area. We will explain the difference between the *Square* class and the *Square2* class shortly.

The main function uses these classes. It creates objects from the various classes and uses the __init__ function of the appropriate class to set the dimensions of the object. After each object is created, the main function requests the object to compute its area as part of an output statement giving information about the object. For example, the statement

```
joe = Circle(23.5)
```

creates a Circle object named *joe* and automatically invokes the Circle *__init__* method that, in turn, sets the radius of the object to 23.5. Then

```
joe.doArea()
```

invokes the *doArea* method for the *Circle* class and returns the area. Figure 29 shows the output after the program in Figure 28 is run.

Here we see polymorphism at work, because there are lots of *doArea* methods; when the program executes, the correct method is used, on the basis of the class to which the object invoking the function belongs. After all,

FIGURE 29

Output from the Program of Figure 28

```
The area of a circle with radius 23.5 is 1734.94454294
The area of a rectangle with height 12.4 and width 18.1 is 224.44
The area of a square with side 3 is 9
The area of a square with side 4.2 is 17.64
```

computing the area of a circle is quite different from computing the area of a rectangle. The algorithms themselves are straightforward; they employ assignment statements to set the dimensions (if a dimension is to be changed from its initial value, which does not happen in this program) and the usual formulas to compute the area of a circle, rectangle, and square. The methods can use the attributes of the objects that invoke them without having the values of those attributes passed as arguments.

Square is a stand-alone class with a *side* attribute and a *doArea* method. The *Square2* class, however, recognizes the fact that squares are special kinds of rectangles. The *Square2* class is a subclass of the *Rectangle* class, as is indicated by the reference to *Rectangle* in the parentheses after the class name *Square2*. It inherits the *width* and *height* properties from the "parent" *Rectangle* class. But creation of a Square2 object doesn't automatically invoke the Rectangle initialization method. Instead, *Square2* has its own *__init__* method, which has a single parameter *value*. The Square2 *__init__* method itself invokes the Rectangle *__init__* method and passes it two copies of *value* to set the *width* and *height* attributes. (Note the syntax for this invocation: `Rectangle.__init__(self, value, value)`.) Square2 also inherits the *setWidth*, *setHeight*, *getWidth*, *getHeight*, and *doArea* methods. In addition, *Square2* has its own function, *setSide*, because setting the value of the "side" makes sense for a square, but not for an arbitrary rectangle. What the user of the *Square2* class doesn't know is that there really isn't a "side" property; the *setSide* function, like the initialization method, merely sets the inherited *width* and *height* properties to the same value. To compute the area, then, the *doArea* function inherited from the *Rectangle* class can be used, and there's no need to redefine it or even to copy the existing code. Here we see inheritance at work.

Inheritance can be carried through multiple "generations." We might redesign the program so that there is one "superclass" that is a general *Shape* class, of which *Circle* and *Rectangle* are subclasses, *Square2* being a subclass of *Rectangle* (see Figure 30 for a possible class hierarchy).

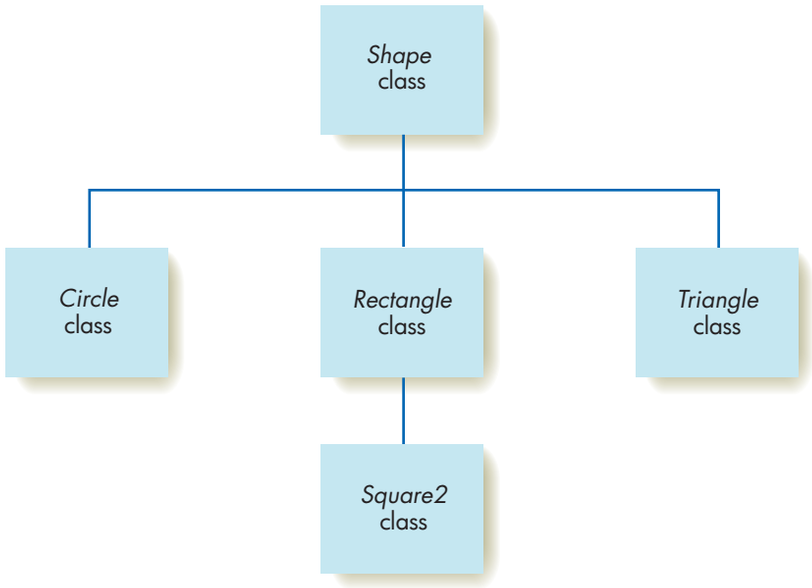
6.4 What Have We Gained?

Now that we have some idea of the flavor of object-oriented programming, we should ask what we gain by this approach. There are two major reasons why OOP is a popular way to program:

- Software reuse
- A more natural "worldview"

SOFTWARE REUSE. Manufacturing productivity took a great leap forward when Henry Ford invented the assembly line. Automobiles could be assembled using identical parts so that each car did not have to be treated as a unique

FIGURE 30
A Hierarchy of Geometric Classes



creation. Computer scientists are striving to make software development more of an assembly-line operation and less of a handcrafted, start-over-each-time process. Object-oriented programming is a step toward this goal: A useful class that has been implemented and tested becomes a component available for use in future software development. Anyone who wants to write an application program involving circles, for example, can use the already written, tried, and tested *Circle* class. As the "parts list" (the class library) grows, it becomes easier and easier to find a "part" that fits, and less and less time has to be devoted to writing original code. If the class doesn't quite fit, perhaps it can be modified to fit by creating a subclass; this is still less work than starting from scratch. Software reuse implies more than just faster code generation. It also means improvements in *reliability*; these classes have already been tested, and if properly used, they will work correctly. And it means improvements in *maintainability*. Thanks to the encapsulation property of object-oriented programming, changes can be made in class implementations without affecting other code, although such change requires retesting the classes.

A MORE NATURAL "WORLDVIEW." The traditional view of programming is procedure-oriented, with a focus on tasks, subtasks, and algorithms. But wait—didn't we talk about subtasks in OOP? Haven't we said that computer science is all about algorithms? Does OOP abandon these ideas? Not at all. It is more a question of *when* these ideas come into play. Object-oriented programming recognizes that in the "real world," tasks are done by entities (objects). Object-oriented program design begins by identifying those objects that are important in the domain of the program because their actions contribute to the mix of activities present in the banking enterprise, the medical office, or wherever. Then it is determined what data should be associated with each object and what subtasks the object contributes to this mix. Finally, an algorithm to carry out each subtask must be designed. We saw in the

modularized version of the SportsWorld program in Figure 22 how the overall algorithm could be broken down into pieces that are isolated within functions. Object-oriented programming repackages those functions by encapsulating them within the appropriate class of objects.

Object-oriented programming is an approach that allows the programmer to come closer to modeling or simulating the world as we see it, rather than to mimic the sequential actions of the Von Neumann machine. It provides another buffer between the real world and the machine, another level of abstraction in which the programmer can create a virtual problem solution that is ultimately translated into electronic signals on hardware circuitry.

Finally, we should mention that a graphical user interface, with its windows, icons, buttons, and menus, is an example of object-oriented programming at work. A general button class, for example, can have properties of height, width, location on the screen, text that may appear on the button, and so forth. Each individual button object has specific values for those properties. The button class can perform certain services by responding to messages, which are generated by events (for example, the user clicking the mouse on a button triggers a "mouse-click" event). Each particular button object individualizes the code to respond to these messages in unique ways. We will not go into details of how to develop graphical user interfaces in Python, but in the next section you will see a bit of the programming mechanics that can be used to draw the graphics items that make up a visual interface.

PRACTICE PROBLEMS

1. What is the output from the following section of code if it is added to the main function of the Python program in Figure 28?

```
one = Square(8)
one.setSide(10)
print("The area of a square with side",\
      one.getSide(), "is",\
      one.doArea())
```

2. In the hierarchy of Figure 30, suppose that the *Triangle* class is able to perform a *doArea* function. What two attributes should any triangle object have?

7 Graphical Programming

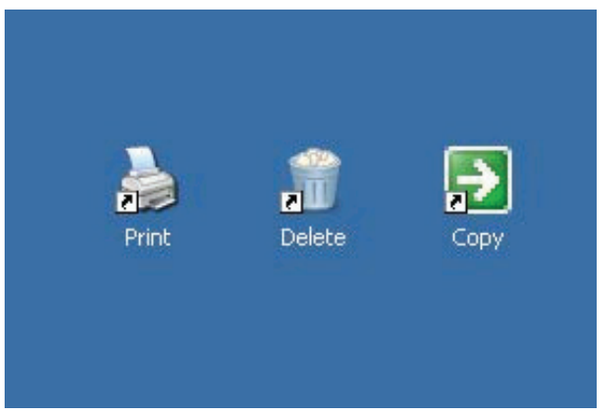
The programs that we have looked at so far all produce *text output*—output composed of the characters {A . . . Z, a . . . z, 0 . . . 9} along with a few punctuation marks. For the first 30 to 35 years of software development, text was

virtually the only method of displaying results in human-readable form, and in those early days it was quite common for programs to produce huge stacks of alphanumeric output. These days an alternative form of output—*graphics*—has become much more widely used. With graphics, we are no longer limited to 100 or so printable characters; instead, programmers are free to construct whatever shapes and images they desire.

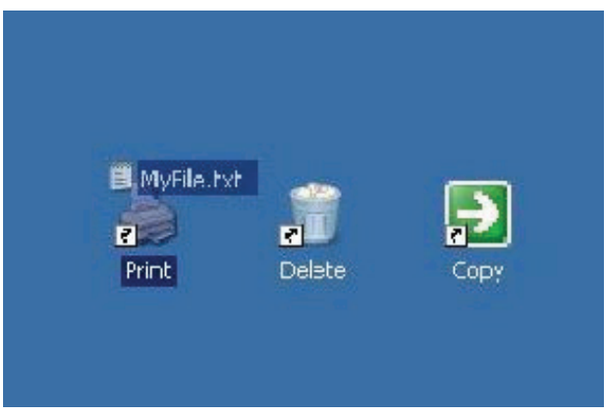
The intelligent and well-planned use of graphical output can produce some phenomenal improvements in software. We discussed this issue in Chapter 6, where we described the move away from the text-oriented operating systems of the 1970s and 1980s, such as MS-DOS and VMS, to operating systems with more powerful and user-friendly graphical user interfaces (GUIs), such as Windows 7, Windows 8, and Mac OS X. Instead of requiring users to learn dozens of complex text-oriented commands for such things as copying, editing, deleting, moving, and printing files, GUIs can present users with simple and easy-to-understand visual metaphors for these operations, such as those shown below. In (a), the operating system presents the user with icons for printing, deleting, and copying a file; in (b), dragging a file to the printer icon prints the file.

Not only does graphics make it easier to manage the tasks of the operating system, it can help us visualize and make sense of massive amounts of output produced by programs that model complex physical, social, and mathematical systems. (We discuss modeling and visualization in Chapter 13.) Finally, there are many applications of computers that would simply be impossible without the ability to display output visually. Applications such as virtual reality, computer-aided design/computer-aided manufacturing (CAD/CAM), games and entertainment, medical imaging, and computer mapping would not be anywhere near as important as they are without the enormous improvements that have occurred in the areas of graphics and visualization.

So, we know that graphical programming is important. The question is: What features must be added to a programming language like Python to produce graphical output?



(a)



(b)

7.1 Graphics Hardware

Modern computer terminals use what is called a **bitmapped display**, in which the screen is made up of thousands of individual picture elements, or **pixels**, laid out in a two-dimensional grid. These are the same pixels used in visual images, as discussed in Chapter 4. In fact, the display is simply one large visual image. The number of pixels on the screen varies from system to system; typical values range from 800 × 600 up to 1560 × 1280. Terminals with a high density of pixels are called **high-resolution** terminals. The higher the resolution—that is, the more pixels available in a given amount of space—the sharper the visual image because each individual pixel is smaller. However, if the screen size itself is small, then a high-resolution image can be too tiny to read. A 30" wide-screen monitor might support a resolution of 2560 × 1600, but that would not be suitable for a laptop screen. In Chapter 4 you learned that a color display requires 24 bits per pixel, with 8 bits used to represent the value of each of the three colors red, green, and blue. The memory that stores the actual screen image is called a **frame buffer**. A high-resolution color display might need a frame buffer with (1560 × 1280) pixels × 24 bits/pixel = 47,923,000 bits, or about 6 MB, of memory for a single image. (One of the problems with graphics is that it requires many times the amount of memory needed for storing text.)

The individual pixels in the display are addressed using a two-dimensional coordinate grid system, the pixel in the upper-left corner being (0, 0). The overall pixel-numbering system is summarized in Figure 31. The specific values for *maxX* and *maxY* in Figure 31 are, as mentioned earlier, system-dependent. (Note that this coordinate system is not the usual mathematical one. Here, the origin is in the upper-left corner, and *y* values are measured downward.)

The terminal hardware displays on the screen the frame buffer value of every individual pixel. For example, if the frame buffer value on a color monitor for position (24, 47) is RGB (0, 0, 0), the hardware sets the color of

FIGURE 31
Pixel-Numbering System in a Bitmapped Display

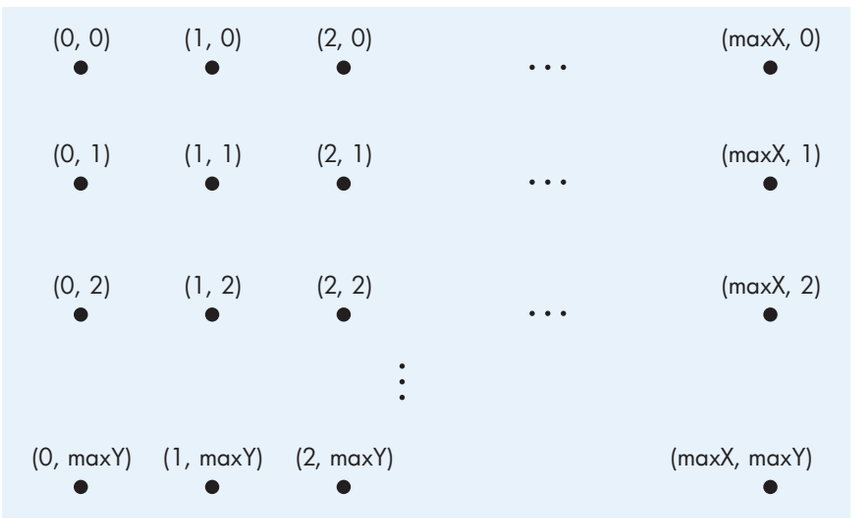
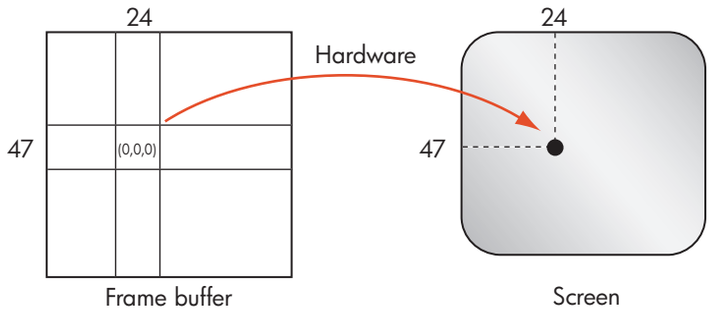


FIGURE 32
Display of Information on the Terminal



the pixel located at column 24, row 47 to black, as shown in Figure 32. The operation diagrammed in Figure 32 must be repeated for all of the 500,000 to 2 million pixels on the screen. However, the setting of a pixel is not permanent; on the contrary, its color and intensity fade quickly. Therefore, each pixel must be "repainted" often enough so that our eyes do not detect any "flicker," or change in intensity. This requires the screen to be completely updated, or refreshed, 30–50 times per second. By setting various sequences of pixels to different colors, the user can have the screen display any desired shape or image. This is the fundamental way in which graphical output is achieved.

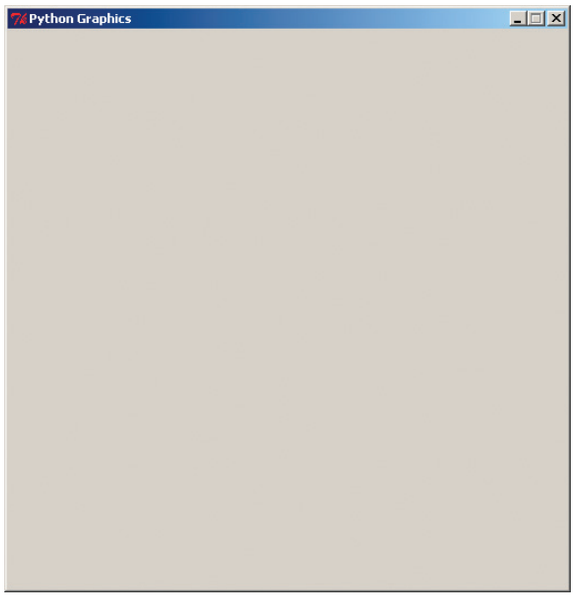
7.2 Graphics Software

To control the setting and clearing of pixels, programmers use a collection of software routines that are part of a special package called a **graphics library**. Typically an "industrial strength" graphics library includes dozens or hundreds of functions for everything from drawing simple geometric shapes like lines and circles, to creating and selecting colors, to more complex operations such as displaying scrolling windows, pull-down menus, and buttons. In this module we'll be using the graphics.py library, written by Dr. John Zelle of Wartburg College, Waverly, Iowa. (See *http://mcsp.wartburg.edu/zelle/python* to download graphics.py, then put the file in the Python Lib folder.) This easy-to-use graphics library allows you to draw basic shapes in a graphics window and will give you a good idea of what visual programming is like.

Because we need to use the graphics library, all graphics programs will begin with the following form of the *import* statement:

```
from graphics import *
```

The graphics library contains a number of classes. To get started, we use a class called *GraphWin* that creates a window where we can do our drawing. Figure 33 shows the complete Python program that brings up the empty window shown on the next page.



The first line in the *main* function in Figure 33 creates a *GraphWin* object called *win*. Three arguments are passed to the GraphWin *__init__* method that represent, respectively, the literal string to appear in the title bar of the window, the width of the window, and the height of the window. If the width and height arguments are omitted, the default dimensions are 200 × 200. The second line has the win object invoke the *getMouse* method of the *GraphWin* class. This method returns information about the point in the graphics window where a mouse click has occurred. In this program we are ignoring the information returned, but the effect is that the program will wait (holding the window on the screen) until the user clicks the mouse somewhere in the window. This causes the second line of the program to be executed, followed by line 3, which closes the graphics window, after which the program closes as well.

The default color of the drawing window is gray, but you can make it white by adding the following line of code after the win object gets created:

```
win.setBackground('white')
```

FIGURE 33

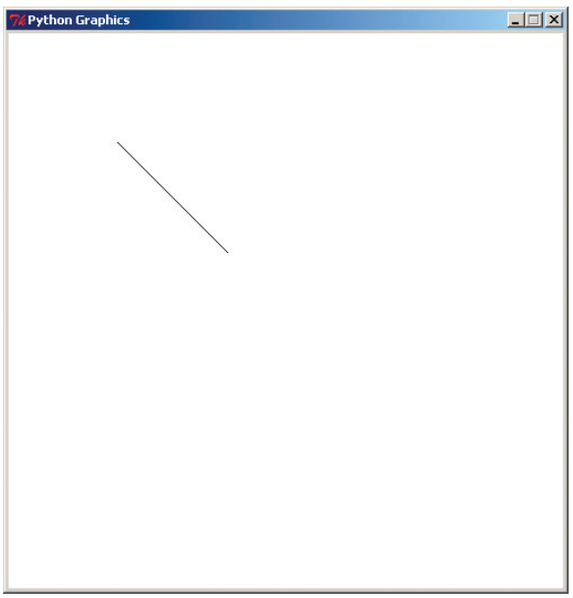
Python Program for Graphics Window

```
from graphics import *

def main():
    win = GraphWin("Python Graphics", 500, 500)
    #add drawing code here
    win.getMouse() # wait for mouse click
    win.close()

main()
```

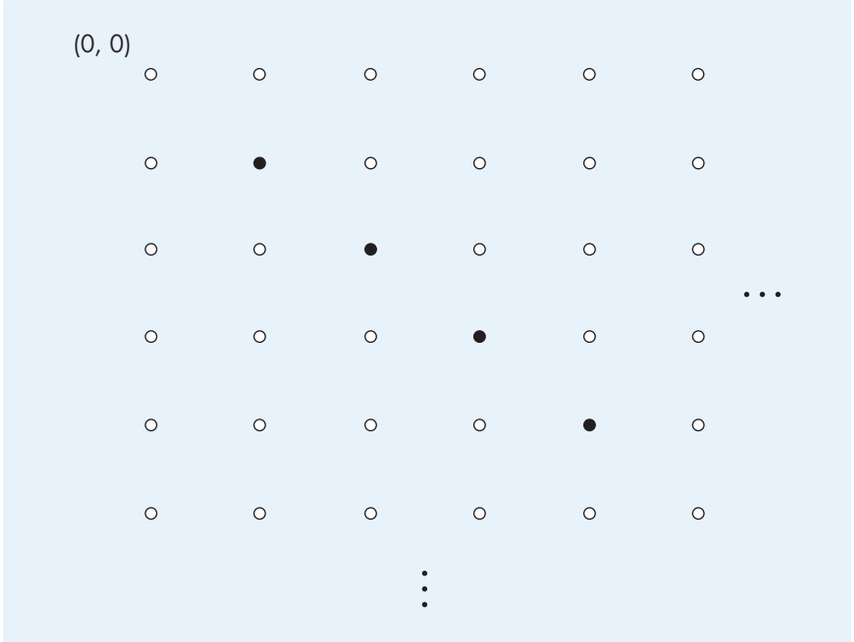
Now that we can get a graphics window to appear, we need the ability to draw in it. We'll be able to draw lines, circles, ovals, and rectangles; fill objects with various colors; and display text. The default drawing color is black. The code snippets shown to create these effects go in the middle of the program, shown in Figure 33, where the boldface comment appears. (We've also set the background color to white in each case to make it easier to see what's being drawn.)

1. **Lines**. The following code creates two Point objects with specific coordinates, then creates an object of the *Line* class from one point to the other. Nothing will appear in the graphics window, however, until the Line object invokes the "draw" method and passes the window on which to draw.

```
start = Point(100, 100)
finish = Point(200, 200)
l = Line(start, finish)
l.draw(win)
```



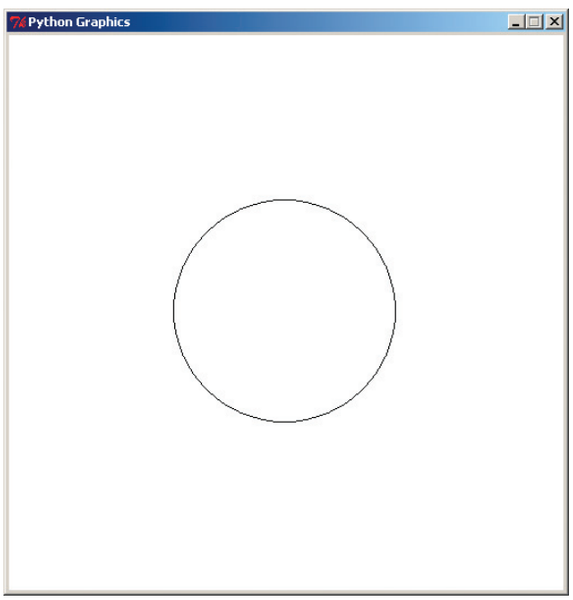
What actually happens internally when the program executes the *l.draw(win)* statement? The terminal hardware determines (using some simple geometry and trigonometry) exactly which pixels on the screen must be "turned on" (i.e., set to the current value of the drawing color) to draw a straight line between the specified coordinates. For example, if the start point is (1,1), the finish point is (4, 4), and the drawing color is black, then the statement *l.draw(win)* causes four pixels in the frame buffer to be set to the RGB value (0, 0, 0) as shown in the next figure.



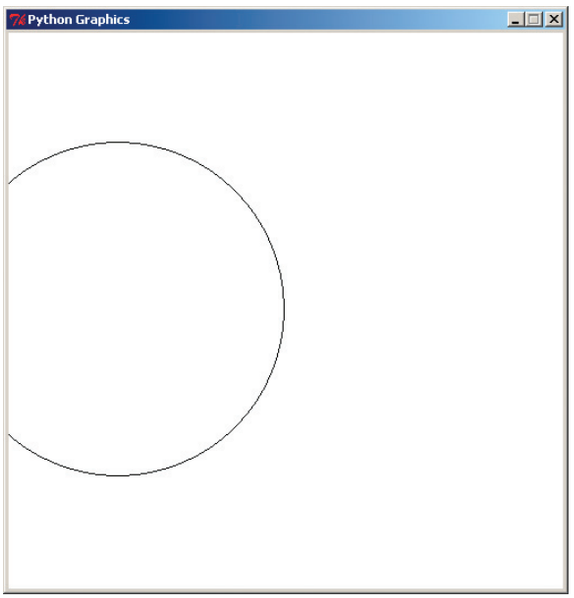
Now, when the hardware draws the frame buffer on the screen, these four pixels are colored black. Because pixels are only about 1/100th of an inch apart, our eyes do not perceive four individual black dots, but an unbroken line segment.

2. **Circles**. The *Circle* class *__init__* method takes two arguments, the center point and the radius. The following code creates a circle with center at (250, 250)—which is the center of our window—and with a radius of 100.

```
center = Point(250, 250)
c = Circle(center, 100)
c.draw(win)
```



If you position the drawing object in such a way that it does not fit within the graphics window, then the image is "clipped" so that only part of it is shown.



3. **Ovals**. The *Oval* class *__init__* method takes two point arguments, the upper-left corner and lower-right corner of the imaginary "bounding box" enclosing the oval.
4. **Rectangles**. The *Rectangle* class *__init__* method also takes two point arguments, the upper-left corner and lower-right corner of the rectangle.
5. **Fill**. Any graphics object can invoke the *setFill* method for its class, which has a single argument of type color. Standard colors such as blue, green, etc., are available—pass the argument in single quotes, as in `c.setFill('black')`.
6. **Text**. The *Text* class *__init__* method takes two arguments, a point and a string. The point is the "center point" of the text string, which will be drawn horizontally. The string is a literal string.

Armed with these objects and methods, we can have a little fun. The program in Figure 34 produces the following window:

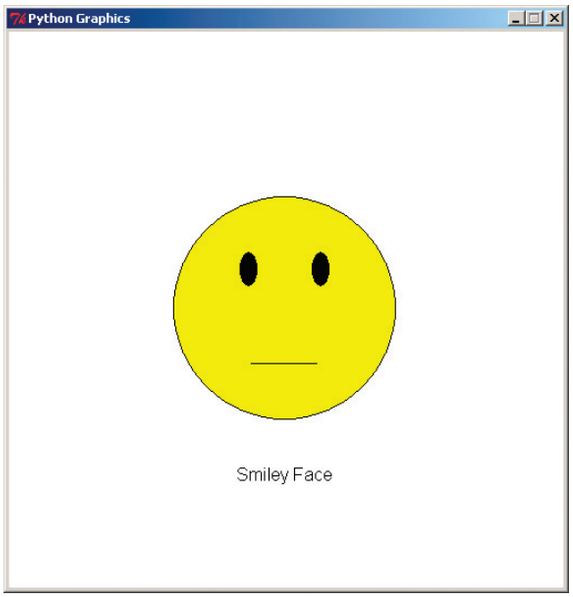


FIGURE 34

Python Program for Smiley Face

```
from graphics import*
def main():  #smiley face
    win = GraphWin("Python Graphics", 500, 500)
    win.setBackground('white')
    center = Point(250, 250)
    c = Circle(center, 100)
    c.draw(win)
    c.setFill('Yellow')
    upper1 = Point(210, 200)
    lower1 = Point(225, 230)
    upper2 = Point(275, 200)
    lower2 = Point(290, 230)
    o1 = Oval(upper1, lower1)
    o1.draw(win)
    o2 = Oval(upper2, lower2)
    o2.draw(win)
    o1.setFill('black')
    o2.setFill('black')
    left = Point(220, 300)
    right = Point(280, 300)
    mouth = Line(left, right)
    mouth.draw(win)
    p = Point(250, 400)
    t = Text(p, "Smiley Face")
    t.draw(win)
    win.getMouse()  # wait for mouse click
    win.close()

main()
```

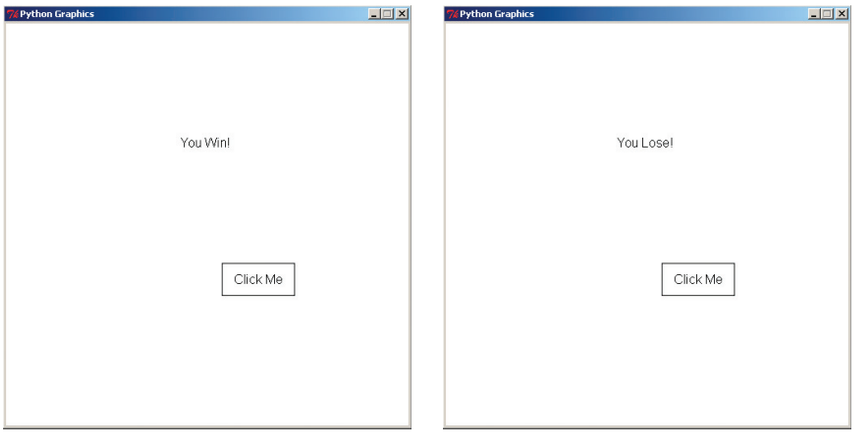
Now that a display window and graphics are available, we seem close to producing elements of a typical GUI. Can we draw a button that acts like a button on a real GUI form—that is, can we write code to sense a mouse click on that button and respond with some action? To make this work, we'll need to again use the *getMouse* method of the *GraphWin* class, but this time we actually want to capture the point information this method returns. In the program of Figure 35, we use the *Rectangle* class to draw a "button" in the graphics window. Then we create a Point object *p* to capture the location returned by the *getMouse* function. The *Point* class has *getX* and *getY* methods that return the coordinates of the calling object. The *if* statement tests whether these coordinates fall within the boundaries of the rectangle and, depending on whether that is true or not, writes a "win" or "lose" message. Figure 36 shows two versions of the result, depending on whether the mouse click occurred on the button (a) or not (b). Of course in a true windows GUI program, we would expect something more interesting to happen in response to clicking on a button.

FIGURE 35

Python Program That Responds to Button Click

```
from graphics import*
def main():
    win = GraphWin("Python Graphics", 500, 500)
    win.setBackground('white')
    upper = Point(270, 300)
    lower = Point(360, 340)
    r = Rectangle(upper, lower)
    r.draw(win)
    rcenter = Point(315, 320)
    t = Text(rcenter, "Click Me")
    t.draw(win)

    p = Point(0,0)
    p = win.getMouse()

    if (p.getX() > 270 and p.getX() < 360)\
      and (p.getY() > 300 and p.getY() < 340):
        center = Point(250, 150)
        t = Text(center, "You Win!")
        t.draw(win)
    else:
        center = Point(250, 150)
        t = Text(center, "You Lose!")
        t.draw(win)

    win.getMouse() # wait for mouse click
    win.close()
main()
```

FIGURE 36
Results of Executing the Program of Figure 35

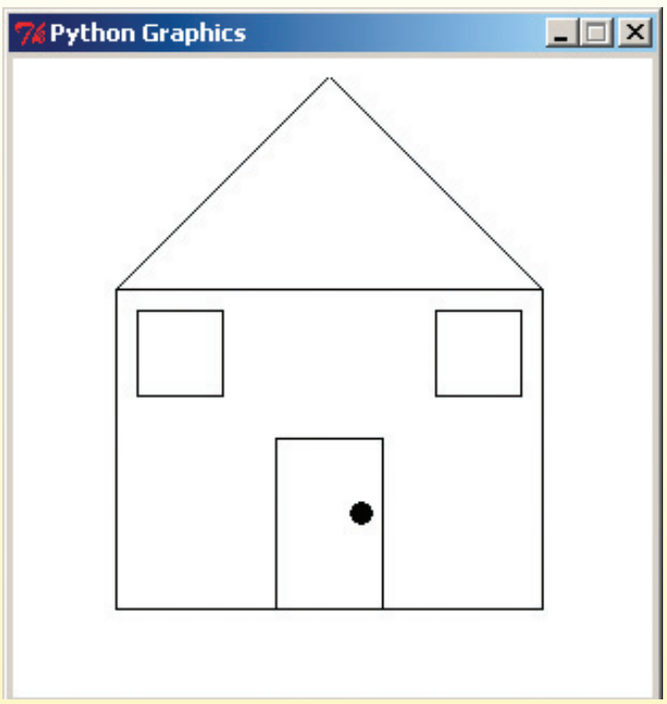


(a) (b)

This brief introduction to graphical programming allows you to produce some interesting images and, even more important, gives you an appreciation for how visually oriented software is developed.

PRACTICE PROBLEM

Write a Python program to draw the following "house" in the graphics window. Create the house using four rectangles (for the base of the house, the door, and the two windows), two line segments (for the roof), and one filled circle (for the doorknob). Locate the house anywhere you want in the graphics window.



8 Conclusion

In this module we looked at one representative high-level programming language, Python. Of course, there is much about this language that has been left unsaid, but we have seen how the use of a high-level language overcomes many of the disadvantages of assembly language programming, creating a more comfortable and useful environment for the programmer. In a high-level language, the programmer need not manage the storage or movement of data values in memory. The programmer can think about the problem at a higher level, can use program instructions that are both more powerful and more natural language–like, and can write a program that is much more portable among various hardware platforms. We also saw how modularization, through the use of functions and parameters, allows the program to be more cleanly structured, and how object orientation allows a more intuitive view of the problem solution and provides the possibility for reuse of helpful classes.

Python is not the only high-level language. You might be interested in looking at the other online language modules for languages similar to Python (Java, C++, C#, and Ada). Some languages have different ways to do assignments, conditional statements, and looping statements. Still other languages take quite a different approach to problem solving. In Chapter 10 of *Invitation to Computer Science*, we look at some other languages and language approaches and also address the question of why there are so many different programming languages.

EXERCISES

1. Write a Python statement to create a variable called *quantity* with initial value 5.0.
2. Write a Python statement to create a string variable called *greeting* with initial value "Top o' the Mornin'".
3. A Python main function needs one variable called *choice*, one variable called *inventory*, and one variable called *sales*. Write Python statements to create these variables; initialize *choice* to the blank character and the other values to zero.
4. Assume that *import math* has been included at the top of your program:
   a. Write a Python statement to print the value of the mathematical constant *e* supplied by the math module.
   b. What will be the result of executing the following two Python statements?

```
math.e = 10
print(math.e)
```

5. You want to write a Python program to compute the average of three integer quiz grades for a single student. Decide what variables your program needs, and create them with appropriate initial values.
6. Given the statement

```
myList = ["eeny", "meeny", "miny",\
"moe"]
```

what Python statement would output "miny"?

7. A Python list can be thought of as representing a 1-D table of values. A 2-D table of values can be represented as a list of lists. For example, the following code

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]
myList = [list1, list2, list3]
```

creates a representation of the 3×3 table $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

Given this code, what would be printed by the following statement?

```
print(myList[2][0])
```

8. Write Python statements to prompt for and collect values for the time in hours and minutes (two integer quantities).
9. Say a program computes two integer quantities *inventoryNumber* and *numberOrdered*. Write a single output statement that prints these two quantities along with appropriate text information.
10. The variables *age* and *weight* currently have the values 32 and 187, respectively. Write the exact output generated by the following statement:

```
print("Your age is" + str(age)\
      + "and your weight is"\
      + str(weight))
```

11. Output that is a decimal number can be formatted so that the number is rounded to a specified number of decimal places. For example, in the TravelPlanner program we might decide that the travel time required should be printed to only two decimal places. We would change the second print statement as follows:

```
print("At", speed, "mph, it will "\
      "take ")
print("%5.2f" % time, "hours to "\
      "travel", distance, "miles.")
```

The "%5.2f" % is a **formatting directive** for printing the numerical value *time* that follows it. The "f" part says the next variable is to be converted (if necessary) to type float. The 5.2 says to print the resulting decimal value using 5 columns (including the decimal point) and rounded to 2 decimal places. The sample result would be

```
Enter your speed in mph: 58
Enter your distance in miles: 657.5
At 58 mph, it will take
11.34 hours to travel 657.5 miles.
```

If this were done in one print statement instead of two, the formatting directive could be included as the end of the literal string preceding the variable time.

Write two Python print statements to generate the following output, assuming that *density* is a type double variable with the value 63.78:

```
The current density is 63.8 to
within one decimal place.
```

12. What is the output after the following sequence of statements is executed?

```
a = 12
b = 20
b = b + 1
a = a + b
print(2*a)
```

13. Write a Python program that gets the length and width of a rectangle from the user and computes and writes out the area.
14. In the SportsWorld program of Figure 14, the user must respond with "C" to choose the circumference task. In such a situation, it is preferable to accept either uppercase or lowercase letters. Rewrite the condition in the program to allow this.
15. Write a Python program that gets a single character from the user and writes out a congratulatory message if the character is a vowel (a, e, i, o, or u), but otherwise writes out a "You lose, better luck next time" message.

EXERCISES

16. Insert the missing line of code so that the following adds the integers from 1 to 10, inclusive.

```
value = 0
top = 10
score = 1
while score <= top:
    value = value + score
    #the missing line
```

17. What is the output after the following code is executed?

```
low = 1
high = 20
while low < high:
    print(low, " ", high)
    low = low + 1
    high = high - 1
```

18. Write a Python program that outputs the even integers from 2 through 30, one per line. Use a while loop.

19. In a while loop, the Boolean condition that tests for loop continuation is done at the top of the loop, before each iteration of the loop body. As a consequence, the loop body might not be executed at all. Our pseudocode language of Chapter 2 contains a do-while loop construction, in which a test for loop termination occurs at the bottom of the loop rather than at the top, so that the loop body always executes at least once. Python contains a *break* statement that causes an exit from the loop, so a do-while effect can be accomplished by the following

```
while True:
    S1
    if condition:
        break
```

where, as usual, S1 can be a compound statement. Write Python code to add up a number of nonnegative integers that the user supplies and to write out the total. Use a negative value as a sentinel, and assume that the first value is nonnegative. Use an *if* and *break* statement.

20. Write a Python program that asks for a duration of time in hours and minutes and writes out the duration only in minutes.

21. Write a Python program that asks for the user's age in years; if the user is under 35, then quote an insurance rate of $2.23 per $100 for life insurance, otherwise, quote a rate of $4.32.

22. Write a Python program that reads integer values until a 0 value is encountered, then writes out the sum of the positive values read and the sum of the negative values read.

23. Write a Python program that reads in a series of positive integers and writes out the product of all the integers less than 25 and the sum of all the integers greater than or equal to 25. Use 0 as a sentinel value.

24. a. Write a Python program that reads in 10 integer quiz grades and computes the average grade.

b. Write a Python program that asks the user for the number of quiz grades, reads them in, and computes the average grade.

c. Redo part (b) so that only the integer part of the average is computed.

25. Write a Python function that receives two integer arguments and writes out their sum and their product. Assume no global variables.

26. Write a Python function that receives a real number argument representing the sales amount for videos rented so far this month. The function asks the user for the number of videos rented today and returns the updated sales figure to the main function. All videos rent for $4.25.

27. Write a Python function that receives three integer arguments and returns the maximum of the three values.

28. Write a Python function that receives miles driven and gallons of gas used and returns miles per gallon.

29. Write a Python program where the main function uses an input function to get the miles driven (a decimal value) and the gallons of gas used (an integer value), then writes out the miles per gallon, using the function from Exercise 28.

30. Write a Python program to balance a checkbook. The main function needs to get the initial balance, the amounts of deposits, and the amounts of checks. Allow the user to process as many transactions as desired; use separate functions to handle deposits and checks. (See Exercise 11 on how to format output to two decimal places, as is usually done with monetary values.)

31. Write a Python program to compute the cost of carpeting three rooms. Carpet cost is $8.95 per square yard. Use four separate functions to collect the dimensions of a room in feet, convert feet into yards, compute the area, and compute the cost per room. The main function should use a loop to process each of the three rooms, then add the three costs, and write out the total cost. (*Hints*: The function to convert feet into yards must be used twice for each room, with two different arguments. Hence, it does not make sense to try to give the parameter the same name as the argument. See Exercise 11 on how to format output to two decimal places, as is usually done with monetary values.)

EXERCISES

32. a. Write a Python *doPerimeter* function for the *Rectangle* class of Figure 28.

b. Write Python code that creates a new Rectangle object called *yuri*, with dimensions 14.4 and 6.5, then writes out information about this object and its perimeter using the *doPerimeter* function from part (a).

33. Draw a class hierarchy diagram similar to Figure 30 for the following classes: *Student*, *Undergraduate_Student*, *Graduate_Student*, *Sophomore*, *Senior*, *PhD_Student*.

34. Imagine that you are writing a program using an object-oriented programming language. Your program will be used to maintain records for a real estate office. Decide on one class in your program and a service that objects of that class might provide.

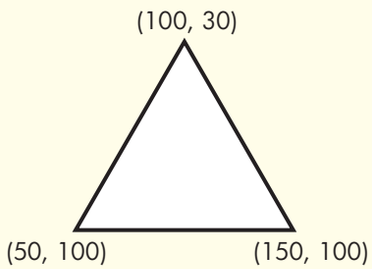
35. Determine the resolution on the screen on your computer (ask your instructor or the local computer center how to do this). Using this information, determine how many bytes of memory are required for the frame buffer to store the following:

a. A black-and-white image (1 bit per pixel)

b. A grayscale image (8 bits per pixel)

c. A color image (24 bits per pixel)

36. Using the *Point* and *Line* classes described in Section 7.2, draw an isosceles triangle with the following configuration:



37. Discuss what problem the display hardware might encounter while attempting to execute the following operations, and describe how this problem could be solved.

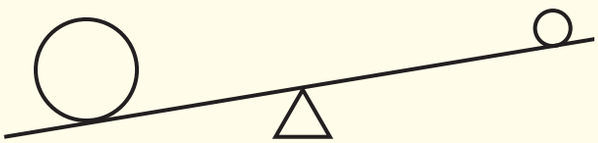
```
start = Point(1, 1)
finish = Point(4, 5)
l = Line(start, finish)
l.draw(win)
```

38. Draw a square with sides 100 pixels in length. Then inscribe a circle of radius 50 inside the square. Position the square and the inscribed circle in the middle of the graphics window.

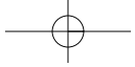
39. Create the following three labeled rectangular buttons:

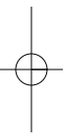
Have the space between the Start and Stop buttons be the same as the space between the Stop and Pause buttons.

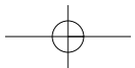
40. Create the following image of a "teeter-totter":



41. Write a program that inputs the coordinates of three mouse clicks from the user and then draws a triangle in the graphics window using those three points.







ANSWERS TO PRACTICE PROBLEMS

Section 2

1. All but number four.

`martinBradley` (camel case)

`C3P_OH` (acceptable, although best not to use underscore character)

`Amy3` (Pascal case)

`3Right` (not acceptable, begins with digit)

`Print` (acceptable, although this could lead to confusing code such as

```
Print = 5
print(Print))
```

2. `Sixty Five`

3. `roster[3]`

Section 3.1

1.
```
quantity = int(input("Enter an integer value: "))
```

2.
```
print("The average high temperature in San Diego "\
      "for the month of May is", average)
```

3. `This isgoodbye, Steve`

Section 3.2

1.
```
next = newNumber
```

2. 55

Section 3.3

1. 30

2. 3
5
7
9
11
13
15
17
19
21

3. Yes

4. 6

5.
```
if night==day:
      print("Equal")
```

Section 4

1.
```
#program to read in and write out
#user's initials
```

```
firstInitial = input("Enter your first initial: ")
lastInitial = input("Enter your last initial: ")
print("Your initials are " + firstInitial + lastInitial)

input("\n\nPress the Enter key to exit")
```

2.

```
#program to compute cost based on
#price per item and quantity purchased

price = float(input("What is the price of the item? "))
quantity = int(input("How many of this item are being "\
  "purchased? "))

cost = price * quantity
print("The total cost for this item is $", cost)

input("\n\nPress the Enter key to exit")
```

3.

```
#program to test a number relative to 5
#and write out the number or its double

number = int(input("Enter a number: "))
if number < 5:
   print("The number is", number)
else:
   print("Twice the number is", 2*number)

input("\n\nPress the Enter key to exit")
```

4.

```
#program to collect a number, then write all
#the values from 1 to that number

counter = 1
number = int(input("Enter a positive number: "))
while counter <= number:
    print(counter)
    counter = counter + 1

input("\n\nPress the Enter key to exit")
```

Section 5.2

1. `number has the value 10`

(The variable number declared in the Helper function is local to that function.)

2.

```
number has the value 10
number has the value 15
```

3.
```
number has the value 10
number has the value 25
```

4. a.
```
def doCircumference(radius):
    circumference = 2*math.pi*radius
    return circumference
```

b. The following is the changed code:

```
if taskToDo == "C":   #compute circumference
    print("\nThe circumference for a pool of radius",\
            radius, "is", doCircumference(radius))
```

Section 6.4

1.
```
The area of a square with side 10 is 100
```

2. height and base

Section 7.2

```
from graphics import*
def main(): #House
    win = GraphWin("Python Graphics", 300, 300)
    win.setBackground('white')
    upper1 = Point(50, 110)
    lower1 = Point (250, 260)
    house = Rectangle(upper1, lower1)
    house.draw(win)

    upper2 = Point(60, 120)
    lower2 = Point(100, 160)
    window1 = Rectangle(upper2, lower2)
    window1.draw(win)

    upper3 = Point(200, 120)
    lower3 = Point(240, 160)
    window2 = Rectangle(upper3, lower3)
    window2.draw(win)

    upper4 = Point(125, 180)
    lower4 = Point(175, 260)
    door = Rectangle(upper4, lower4)
    door.draw(win)

    start1 = Point(50, 110)
    finish1 = Point(150, 10)
    line1 = Line(start1, finish1)
    line1.draw(win)

    start2 = Point(250, 110)
    finish2 = Point(150, 10)
    line2 = Line(start2, finish2)
    line2.draw(win)
```

```
    center = Point(165, 215)
    c = Circle(center, 5)
    c.draw(win)
    c.setFill('black')

    win.getMouse() # wait for mouse click
    win.close()
main()
```