

Python Language Companion for
Starting Out with Programming Logic and Design, 4th Edition
By Tony Gaddis

Copyright © 2016 Pearson Education, Inc.

Table of Contents

	Introduction	2
Chapter 1	Introduction to Python	3
Chapter 2	Input, Processing, and Output	8
Chapter 3	Modularizing Programs with Functions	19
Chapter 4	Decision Structures and Boolean Logic	29
Chapter 5	Repetition Structures	38
Chapter 6	Value-Returning Functions	47
Chapter 7	Input Validation	59
Chapter 8	Arrays (Lists)	61
Chapter 9	Sorting and Searching	71
Chapter 10	Files	76
Chapter 11	Menu-Driven Programs	93
Chapter 12	Text Processing	95
Chapter 13	Recursion	99
Chapter 14	Object-Oriented Programming	101
Chapter 15	GUI Applications and Event-Driven Programming	110
Appendix A Introduction to IDLE		125

Introduction

Welcome to the Python Language Companion for *Starting Out with Programming Logic and Design, 4th Edition*, by Tony Gaddis. You can use this guide as a reference for the Python Programming Language as you work through the textbook. Each chapter in this guide corresponds to the same numbered chapter in the textbook. As you work through a chapter in the textbook, you can refer to the corresponding chapter in this guide to see how the chapter's topics are implemented in Python. In this book you will also find Python versions of many of the pseudocode programs that are presented in the textbook.

Chapter 1 Introduction to Python

Installing Python

Before you can run Python programs on your computer you will need to install the Python interpreter. You can download the latest version of the Python Windows installer from www.python.org/download. The web site also provides downloadable versions of Python for several other operating systems.

Note: On the download page you likely will see two current versions of Python. One will be named Python 2.x.x, and the other will be named Python 3.x.x. ***This booklet is written for the Python 3.x.x version.***

When you execute the Python Windows installer, it's best to accept all of the default settings by clicking the Next button on each screen. (Answer "Yes" if you are prompted with any Yes/No questions.) As you perform the installation, take note of the directory where Python is being installed. It will be something similar to **C:\Python37**. (The 37 in the path name represents the Python version. The directory name Python37 would indicate Python version 3.7.) You will need to remember this location after finishing the installation.

When the installer is finished, the Python interpreter, the IDLE programming environment, and the Python documentation will be installed on your system. When you click the Start button and look at your All Programs list you should see a program group named something like *Python 3.7*. The program group will contain the following items:

- ➔ **IDLE (Python GUI)** – When you click this item the IDLE programming environment will execute. IDLE is an integrated development environment that you can use to create, edit, and execute Python programs. See Appendix A for a brief introduction to IDLE.
- ➔ **Python Command Line** – Clicking this item launches the Python interpreter in interactive mode.
- ➔ **Python Manuals** – This item launches a utility program that allows you to browse documentation for the **modules in the Python standard library**.
- ➔ **Python Docs Server** – This item opens the Python Manuals in your web browser. The manuals include **tutorials**, a reference section for the Python standard library, an in-depth reference for the Python language, and information on many advanced topics.
- ➔ **Uninstall Python** – This item removes Python from your system.

Adding the Python Directory to the Path Variable

If you plan to execute the Python interpreter from a command prompt window, you will probably want to add the Python directory to the existing contents of your system's `Path` variable. (You saw the name of the Python directory while installing Python. It is something

similar to **C:\Python34.**) Doing this will allow your system to find the Python interpreter from any directory when you run it at the command-line.

Use the following instructions to edit the Path variable under Windows 8 and Windows 7.

Windows 8

- In the Right bottom corner of the screen, click on the Search icon and type **Control Panel**.
- Click on **Control Panel**, then click **System**, then click **Advanced system settings**.
- Click on the **Advanced** tab, then click **Environment Variables**. Under System Variables, find `Path`, click on it, and then click the **Edit** button.
- Add a semicolon to the end of the existing contents and then add the Python directory path. Click the **OK** buttons until all the dialog boxes are closed, and exit the control panel.

Windows 7

- Click the **Start button**
- Right-click **Computer**
- On the pop-up menu select **Properties**.
- In the window that appears next, click **Advanced system settings**. This displays the System Properties window.
- Click the **Environment Variables...** button.
- In the System Variables list, scroll to the `Path` variable. Select the `Path` variable and click the **Edit** button.
- Add a semicolon to the end of the existing contents, and then add the Python directory path. Click the **OK** buttons until all the dialog boxes are closed, and exit the control panel.

The Python Interpreter

Python is an interpreted language. When you install the Python language on your computer, one of the items that is installed is the Python interpreter. The *Python interpreter* is a program that can read Python programming statements and execute them. (Sometimes we will refer to the Python interpreter simply as the interpreter.)

You can use the interpreter in two modes: interactive mode and script mode. In *interactive mode*, the interpreter waits for you to type Python statements on the keyboard. Once you type a statement, the interpreter executes it and then waits for you to type another statement. In *script mode*, the interpreter reads the contents of a file that contains Python statements. Such a file is known as a *Python program* or a *Python script*. The interpreter executes each statement in the Python program as it reads it.

Interactive Mode (Command Line)

Once Python has been installed and set up on your system, you start the interpreter in interactive mode by going to the operating system's command line and typing the following command:

```
python
```

If you are using Windows, you can alternatively click the *Start* button, then *All Programs*. You should see a program group named something like *Python 3.4*. (The "3.4" is the version of Python that is installed.) Inside this program group you should see an item named *Python (command line)*. Clicking this menu item will start the Python interpreter in interactive mode.

When the Python interpreter starts in interactive mode, you will see something like the following displayed in a console window:

```
Python 3.4.3 (v3.4.3:9a73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> that you see is a prompt that indicates the interpreter is waiting for you to type a Python statement. Let's try it out. One of the simplest actions that you can perform in Python is to display a message on the screen. For example, the following statement causes the message Python programming is fun! to be displayed:

```
print('Python programming is fun!')
```

Notice that inside the parentheses that appear after the word `print`, we have written `'Python programming is fun!'`. The quote marks are necessary, but they will not be displayed. They simply mark the beginning and the end of the text that we wish to display. Here is an example of how you would type this statement at the interpreter's prompt:

```
>>> print('Python programming is fun!')
```

After typing the statement you press the Enter key and the Python interpreter executes the statement, as shown here:

```
>>> print('Python programming is fun!') [Enter]
Python programming is fun!
>>>
```

After the message is displayed, the >>> prompt appears again, indicating that the interpreter is waiting for you to enter another statement. Let's look at another example. In the following sample session we have entered two statements.

```
>>> print('To be or not to be') [Enter]
To be or not to be
>>> print('That is the question.') [Enter]
That is the question.
>>>
```

If you incorrectly type a statement in interactive mode, the interpreter will display an error message. This will make interactive mode useful to you while you learn Python. As you learn new parts of the Python language, you can try them out in interactive mode and get immediate feedback from the interpreter.

To quit the Python interpreter in interactive mode on a Windows computer, press Ctrl-Z (pressing both keys together) followed by Enter. On a Mac, Linux, or UNIX computer, press Ctrl-D.

Writing Python Programs and Running Them in **Script Mode**

Although interactive mode is useful for testing code, the statements that you enter in interactive mode are not saved as a program. They are simply executed and their results displayed on the screen. If you want to save a set of Python statements as a program, you save those statements in a file. Then, to execute the program, you use the Python interpreter in script mode.

For example, suppose you want to write a Python program that displays the following three lines of text:

```
Nudge nudge
Wink wink
Know what I mean?
```

To write the program you would use a simple text editor like Notepad (which is installed on all Windows computers) to create a file containing the following statements:

```
print('Nudge nudge')
print('Wink wink')
print('Know what I mean?')
```

Note: It is possible to use a word processor to create a Python program, but you must be sure to save the program as a plain text file. Otherwise the Python interpreter will not be able to read its contents.

When you save a Python program, you give it a name that ends with the **.py extension**, which identifies it as a Python program. For example, you might save the program previously shown

with the name `test.py`. To run the program you would go to the directory in which the file is saved and type the following command at the operating system command line:

```
python test.py (Script Mode)
```

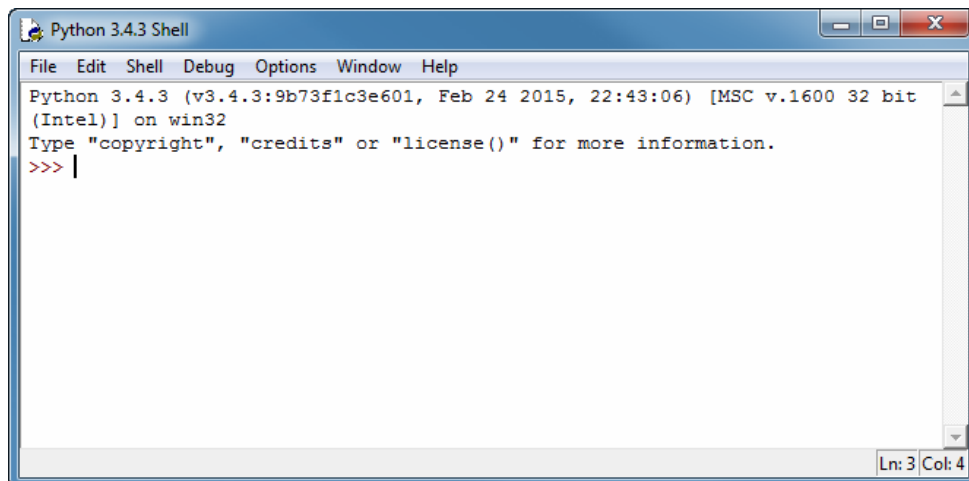
This starts the Python interpreter in script mode and causes it to execute the statements in the file `test.py`. When the program finishes executing, the Python interpreter exits.

The IDLE Programming Environment

The previous sections described how the Python interpreter can be started in interactive mode or script mode at the operating system command line. As an alternative, you can use a program named *IDLE*, which is automatically installed when the Python language is installed. (IDLE stands for **Integrated DeveLopment Environment**.) When you run IDLE, the window shown in Figure 1-1 appears. Notice that the `>>>` prompt appears in the IDLE window, indicating that the interpreter is running in interactive mode. You can type Python statements at this prompt and see them executed in the IDLE window.

IDLE also has a **built-in text editor** with features specifically designed to help you write Python programs. For example, the IDLE editor "colorizes" code so that key words and other parts of a program are displayed in their own distinct colors. This helps make programs easier to read. In IDLE you can write programs, save them to disk, and execute them. Appendix A provides a quick introduction to IDLE, and leads you through the process of creating, saving, and executing a Python program.

Figure 1-1 IDLE



Note: Although IDLE is installed with Python, there are several other Python IDEs available. Your instructor might prefer that you use a specific one in class.

Chapter 2 Input, Processing, and Output

Displaying Screen Output

A *function* is a piece of prewritten code that performs an operation. Python has numerous built-in functions that perform various operations. Perhaps the most fundamental built-in function is the `print` function, which displays output on the screen. Here is an example of a statement that executes the `print` function:

```
print('Hello world')
```

When programmers execute a function, they say that they are *calling* the function. When you call the `print` function, you type the word `print`, followed by a set of parentheses. Inside the parentheses, you type an *argument*, which is the data that you want displayed on the screen. In the previous example, the argument is `'Hello world'`. The quote marks will not be displayed when the statement executes, however. The quote marks simply specify the beginning and the end of the text that we wish to display.

Suppose your instructor tells you to write a program that displays your name and address on the computer screen. Program 2-1 shows an example of such a program, with the output that it will produce when it runs. (The line numbers that appear in a program listing in this book are *not* part of the program. We use the line numbers in our discussion to refer to parts of the program.)

Program 2-1 (*output.py*)

```
1  print('Kate Austen')
2  print('123 Full Circle Drive')
3  print('Asheville, NC 28899')
```

This program is the Python version of **Program 2-1** in your textbook.

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Remember, these line numbers are **NOT** part of the program! Don't type the line numbers when you are entering program code. All of the programs in this booklet will show line numbers for reference purposes only.

It is important to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. When you run this program, the first

statement will execute, followed by the second statement, and followed by the third statement.

Strings and String Literals

In Python code, string literals must be enclosed in quote marks. As mentioned earlier, the quote marks simply mark where the string data begins and ends.

In Python you can enclose string literals in a set of single-quote marks (') or a set of double-quote marks ("). The string literals in Program 2-1 are enclosed in single-quote marks, but the program could also be written as shown here:

```
print("Kate Austen")
print("123 Full Circle Drive")
print("Asheville, NC 28899")
```

If you want a string literal to contain either a single-quote or an apostrophe as part of the string, you can enclose the string literal in double-quote marks. For example, Program 2-2 prints two strings that contain apostrophes.

Program 2-2 (*apostrophe.py*)

```
1 print("Don't fear!")
2 print("I'm here!")
```

Program Output

```
Don't fear!
I'm here!
```

Likewise, you can use single-quote marks to enclose a string literal that contains double-quotes as part of the string. Program 2-3 shows an example.

Program 2-3 (*display_quote.py*)

```
1 print('Your assignment is to read "Hamlet" by tomorrow.')
```

Program Output

```
Your assignment is to read "Hamlet" by tomorrow.
```

Python also allows you to enclose string literals in triple quotes (either " " " or ' ' '). Triple quoted strings can contain both single quotes and double quotes as part of the string. The following statement shows an example:

```
print("""I'm reading "Hamlet" tonight.""")
```

This statement will print:

```
I'm reading "Hamlet" tonight.
```

Triple quotes can also be used to surround multiline strings, which is something that single and double quotes cannot be used for. Here is an example:

```
print( """One  
Two  
Three""" )
```

This statement will print:

```
One  
Two  
Three
```

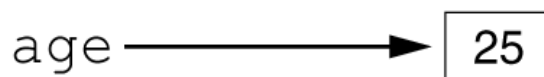
Variables

Variables are not declared in Python. Instead, you use an *assignment statement* to create a variable. Here is an example of an assignment statement:

```
age = 25
```

After this statement executes, a variable named `age` will be created and it will reference the value 25. This concept is shown in Figure 2-1. In the figure, think of the value 25 as being stored somewhere in the computer's memory. The arrow that points from `age` to the value 25 indicates that the name `age` references the value.

Figure 2-1 The `age` variable references the value 25



An assignment statement is written in the following general format:

variable = expression

The equal sign (=) is known as the *assignment operator*. In the general format, *variable* is the name of a variable and *expression* is a value, or any piece of code that results in a value. After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

The code in Program 2-4 demonstrates a variable. Line 1 creates a variable named `room` and assigns it the value 503. The statements in lines 2 and 3 display a message. Notice that line 3 displays the value that is referenced by the `room` variable.

Program 2-4 (*variable_demo.py*)

```
1 room = 503
2 print('I am staying in room number')
3 print(room)
```

Program Output

```
I am staying in room number
503
```

Variable Naming Rules

You may choose your own variable names in Python, as long as you do not use any of the Python key words. The key words make up the core of the language and each has a specific purpose. Table 2-1 shows a list of the Python key words.

Additionally, you must follow these rules when naming variables in Python:

- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (`_`).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means the variable name `ItemsOrdered` is not the same as `itemsordered`.

Table 2-1 The Python key words

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

Displaying Multiple Items with the `print` Function

If you look back at Program 2-4 you will see that we used the following two statements in lines 3 and 4:

```
print('I am staying in room number')
print(room)
```

We used two calls to the `print` function because we needed to display two pieces of data. Line 3 displays the string literal `'I am staying in room number'`, and line 4 displays the value referenced by the `room` variable.

This program can be simplified, however, because Python allows us to display multiple items with one call to the `print` function. We simply have to separate the items with commas as shown in Program 2-5.

Program 2-5 (variable_demo3.py)

```
1 room = 503
2 print('I am staying in room number', room)
```

Program Output

```
I am staying in room number 503
```

The statement in line 2 displays two items: a string literal followed by the value referenced by the `room` variable. Notice that Python automatically printed a space between these two items. When multiple items are printed this way in Python, they will automatically be separated by a space.

Numeric Data Types and Literals

Python uses the `int` data type to store integers, and the `float` data type to store real numbers. Let's look at how Python determines the data type of a number. Many of the programs that you will see will have numeric data written into their code. For example, the following statement, which appears in Program 2-4, has the number 503 written into it.

```
room = 503
```

This statement causes the value 503 to be stored in memory, and it makes the `room` variable reference it. The following statement shows another example. This statement has the number 2.75 written into it.

```
dollars = 2.75
```

This statement causes the value 2.75 to be stored in memory, and it makes the `dollars` variable reference it. A number that is written into a program's code is called a *numeric literal*. When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:

- A numeric literal that is written as a whole number with no decimal point is considered an `int`. Examples are 7, 124, and -9.
- A numeric literal that is written with a decimal point is considered a `float`. Examples are 1.5, 3.14159, and 5.0.

So, the following statement causes the number 503 to be stored in memory as an `int`:

```
room = 503
```

And the following statement causes the number 2.75 to be stored in memory as a `float`:

```
dollars = 2.75
```

Storing Strings with the `str` Data Type

In addition to the `int` and `float` data types, Python also has a data type named `str`, which is used for storing strings in memory. The code in Program 2-6 shows how strings can be assigned to variables.

Program 2-6 (*string_variable.py*)

```
1 first_name = 'Kathryn'
2 last_name = 'Marino'
3 print(first_name, last_name)
```

Program Output

```
Kathryn Marino
```

Reading Input from the Keyboard

In this booklet we will use Python's built-in `input` function to read input from the keyboard. The `input` function reads a piece of data that has been entered at the keyboard and returns that piece of data, as a string, back to the program. You normally use the `input` function in an assignment statement that follows this general format:

`variable = input(prompt)`

In the general format, *prompt* is a string that is displayed on the screen. The string's purpose is to instruct the user to enter a value. *variable* is the name of a variable that will reference the data that was entered on the keyboard. Here is an example of a statement that uses the `input` function to read data from the keyboard:

`name = input('What is your name? ')`

When this statement executes, the following things happen:

- The string 'What is your name? ' is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard, and then press the Enter key.
- When the Enter key is pressed, the data that was typed is returned as a string, and assigned to the name variable.

To demonstrate, look at the following interactive session:

```
>>> name = input('What is your name? ') [Enter]
What is your name? Holly [Enter]
>>> print(name) [Enter]
Holly
>>>
```

When the first statement was entered, the interpreter displayed the prompt 'What is your name? ', and waited for the user to enter some data. The user entered *Holly* and pressed the Enter key. As a result the string 'Holly' was assigned to the name variable. When the second statement was entered, the interpreter displayed the value referenced by the name variable.

Program 2-7 shows a complete program that uses the `input` function to read two strings as input from the keyboard.

Program 2-7 (*string_input.py*)

```
1 first_name = input('Enter your first name: ')
2 last_name = input('Enter your last name: ')
3 print('Hello', first_name, last_name)
```

Program Output (with Input Shown in Bold)

```
Enter your first name: Vinny [Enter]
Enter your last name: Brown [Enter]
Hello Vinny Brown
```

Reading Numbers With The `input` Function

The `input` function always returns the user's input as a string, even if the user enters numeric data. For example, suppose you call the `input` function and the user types the number 72 and pressed the Enter key. The value that is returned from the `input` function is the string '72'. This can be a problem if you want to use the value in a math operation. Math operations can be performed only on numeric values, not strings.

Fortunately, Python has built-in functions that you can use to convert a string to a numeric type. Table 2-2 summarizes two of these functions.

Table 2-2 Data Conversion Functions

Function	Description
<code>int(item)</code>	You pass an argument to the <code>int()</code> function and it returns the argument's value converted to an <code>int</code> .
<code>float(item)</code>	You pass an argument to the <code>float()</code> function and it returns the argument's value converted to a <code>float</code> .

For example, suppose you are writing a payroll program and you want to get the number of hours that the user has worked. Look at the following code:

```
string_value = input('How many hours did you work? ')
hours = int(string_value)
```

The first statement gets the number of hours from the user, and assigns that value as a string to the `string_value` variable. The second statement calls the `int()` function, passing `string_value` as an argument. The value referenced by `string_value` is converted to an `int`, and assigned to the `hours` variable.

This example illustrates how the `int()` function works, but it is inefficient because it creates two variables: one to hold the string that is returned from the `input` function, and another to hold the integer that is returned from the `int()` function. The following code shows a better approach. This one statement does all of the work that the previously shown two statements do, and it creates only one variable:

```
hours = int(input('How many hours did you work? '))
```

This one statement uses *nested function* calls. The value that is returned from the `input` function is passed as an argument to the `int()` function. This is how it works:

- It calls the `input` function to get a value entered at the keyboard.
- The value that is returned from the `input` function (a string), is passed as an argument to the `int()` function.
- The `int` value that is returned from the `int()` function is assigned to the `hours` variable.

After this statement executes, the `hours` variable will be assigned the value entered at the keyboard, converted to an `int`. Program 2-8 shows a sample program that uses the `input` function.

Program 2-8 (input.py)

```
1 age = int(input('What is your age? '))
2 print('Here is the value that you entered:')
3 print(age)
```

Program Output (with Input Shown in Bold)

```
What is your age? 28 [Enter]
Here is the value that you entered:
28
```

This is the Python version
of **Program 2-2** in your
textbook.

You read floating-point numbers as input in a similar fashion. Suppose you want to get the user's hourly pay rate. The following statement prompts the user to enter that value at the keyboard, converts the value to a `float`, and assigns it to the `pay_rate` variable:

```
pay_rate = float(input('What is your hourly pay rate? '))
```

This is how it works:

- It calls the `input` function to get a value entered at the keyboard.
- The value that is returned from the `input` function (a string), is passed as an argument to the `float()` function.
- The `float` value that is returned from the `float()` function is assigned to the `pay_rate` variable.

After this statement executes, the `pay_rate` variable will be assigned the value entered at the keyboard, converted to a `float`.

Performing Calculations

Table 2-3 lists the math operators that are provided by the Python language.

Table 2-3 Python math operators

Symbol	Operation	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the quotient
%	Remainder	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

Here are some examples of statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax
sale = price - discount
population = population * 2
half = number / 2
leftOver = 17 % 3
result = 4**2
```

Program 2-9 shows an example program that performs mathematical calculations (This program is the Python version of pseudocode Program 2-8 in your textbook.)

Program 2-9 (sale_price.py)

```
1 original_price = float(input("Enter the item's original price: "))
2 discount = original_price * 0.2
3 sale_price = original_price - discount
4 print('The sale price is', sale_price)
```

This is the Python version
of **Program 2-9** in your
textbook.

Program Output (With Input Shown in Bold)

```
Enter the item's original price: 100.00 [Enter]
The sale price is 80.0
```

In Python, the order of operations and the use of parentheses as grouping symbols works just as described in the textbook.

Integer Division

In Python, when an integer is divided by an integer the result will also be an integer. This behavior is known as *integer division*. For example, look at the following statement:

```
number = 3 / 2
```

Because the numbers 3 and 2 are both treated as integers, Python will throw away (truncate) the fractional part of the result. So, the statement will assign the value 1 to the number variable, not 1.5.

If you want to make sure that a division operation yields a real number, at least one of the operands must be a number with a decimal point or a variable that references a `float` value. For example, we could rewrite the statement as follows:

```
number = 3.0 / 2.0
```

Documenting a Program with Comments

To write a line comment in Python you simply place the # symbol where you want the comment to begin. The Python interpreter ignores everything from that point to the end of the line. Here is an example:

```
# This program calculates an employee's gross pay.
```

Chapter 3 Modularizing Programs with Functions

Chapter 3 in your textbook discusses modules as named groups of statements that perform specific tasks in a program. You use modules to break a program down into small, manageable units. In Python, we use *functions* for this purpose. (In Python, the term "module" has a slightly different meaning. A Python module is a file that contains a set of related program elements, such as functions.)

In this chapter we will discuss how to define and call Python functions, use local variables in a function, and pass arguments to a function. We also discuss global variables, and the use of global constants.

Defining and Calling a Function

To create a function you write its *definition*. Here is the general format of a function definition in Python:

```
def function_name():  
    statement  
    statement  
    etc.
```

The first line is known as the *function header*. It marks the beginning of the function definition. The function header begins with the key word `def`, followed by the name of the function, followed by a set of parentheses, followed by a colon.

Beginning at the next line is a set of statements known as a block. A *block* is simply a set of statements that belong together as a group. These statements are performed any time the function is executed. Notice in the general format that all of the statements in the block are indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

Let's look at an example of a function. Keep in mind that this is not a complete program. We will show the entire program in a moment.

```
def message():  
    print('I am Arthur,')  
    print('King of the Britons.')
```

This code defines a function named `message`. The `message` function contains a block with two statements. Executing the function will cause these statements to execute.

Calling a Function

A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must *call* it. This is how we would call the `message` function:

```
message( )
```

When a function is called, the interpreter jumps to that function and executes the statements in its block. Then, when the end of the block is reached, the interpreter jumps back to the part of the program that called the function, and the program resumes execution at that point. When this happens, we say that the function *returns*. To fully demonstrate how function calling works, look at Program 3-1.

Program 3-1 (*function_demo.py*)

```
1 # This program demonstrates a function.
2 # First, we define a function named message.
3 def message():
4     print('I am Arthur,')
5     print('King of the Britons.')
6
7 # Call the message function.
8 message()
```

Program Output

```
I am Arthur,
King of the Britons.
```

When the Python interpreter reads the `def` statement in line 3, a function named `message` is created in memory, containing the block of statements in lines 4 and 5. (A function definition creates a function, but it does not cause the function to execute.) Next, the interpreter encounters the comment in line 7, which is ignored. Then it executes the statement in line 8, which is a function call. This causes the `message` function to execute, which prints the two lines of output.

Program 3-1 has only one function, but it is possible to define many functions in a program. In fact, it is common for a program to have a `main` function that is called when the program starts. The `main` function then calls other functions in the program as they are needed. It is often said that the `main` function contains a program's *mainline logic*, which is the overall logic of the program. Program 3-2 shows an example of a program with two functions: `main` and `show_message`. This is the Python version of Program 3-1 in your textbook, with some extra comments added.

Program 3-2 (function_demo2.py)

```
1  # Define the main function.
2  def main():
3      print("I have a message for you.")
4      show_message()
5      print("That's all folks!")
6
7  # Define the show_message function.
8  def show_message():
9      print("Hello world")
10
11 # Call the main function.
12 main()
```

This is the Python version
of **Program 3-1** in your
textbook.

Program Output

```
I have a message for you.
Hello world
That's all folks!
```

The `main` function is defined in lines 2 through 5, and the `show_message` function is defined in lines 8 through 9. When the program runs, the statement in line 12 calls the `main` function, which then calls the `show_message` function in line 4.

Indentation in Python

In Python, each line in a block must be indented. As shown in Figure 3-1, the last indented line after a function header is the last line in the function's block.

Figure 3-1 All of the statements in a block are indented

The last indented line is
the last line in the block.

```
def greeting():
    print('Good morning!')
    print('Today we will learn about functions.')
```

These statements are not in the block.

```
print('I will call the greeting function.')
greeting()
```

When you indent the lines in a block, make sure each line begins with the same number of spaces. Otherwise an error will occur. For example, the following function definition will cause an error because the lines are all indented with different numbers of spaces.

```
def my_function():  
    print('And now for')  
    print('something completely')  
    print('different.')
```

In an editor there are two ways to indent a line: (1) by pressing the Tab key at the beginning of the line, or (2) by using the spacebar to insert spaces at the beginning of the line. You can use either tabs or spaces when indenting the lines in a block, but don't use both. Doing so may confuse the Python interpreter and cause an error.

IDLE, as well as most other Python editors, automatically indents the lines in a block. When you type the colon at the end of a function header, all of the lines typed afterward will automatically be indented. After you have typed the last line of the block you press the Backspace key to get out of the automatic indentation.

Tip: Python programmers customarily use four spaces to indent the lines in a block. You can use any number of spaces you wish, as long as all the lines in the block are indented by the same amount.

Note: Blank lines that appear in a block are ignored.

Local Variables

Anytime you assign a value to a variable inside a function, you create a *local variable*. A local variable belongs to the function in which it is created, and only statements inside that function can access the variable. (The term *local* is meant to indicate that the variable can be used only locally, within the function in which it is created.) In Chapter 3 of your textbook you learned that a variable's scope is the part of the program in which the variable may be accessed. A local variable's scope is the function in which the variable is created.

Because a function's local variables are hidden from other functions, the other functions may have their own local variables with the same name. For example, look at the Program 3-3. In addition to the `main` function, this program has two other functions: `texas` and `california`. These two functions each have a local variable named `birds`.

Program 3-3 (*birds.py*)

```
1  # This program demonstrates two functions that  
2  # have local variables with the same name.  
3  
4  def main():  
5      # Call the texas function.  
6      texas()  
7      # Call the california function.  
8      california()
```

```
9
10 # Definition of the texas function. It creates
11 # a local variable named birds.
12 def texas():
13     birds = 5000
14     print('texas has', birds, 'birds.')
15
16 # Definition of the california function. It also
17 # creates a local variable named birds.
18 def california():
19     birds = 8000
20     print('california has', birds, 'birds.')
21
22 # Call the main function.
23 main()
```

Program Output

```
texas has 5000 birds.
california has 8000 birds.
```

Although there are two separate variables named `birds` in this program, only one of them is visible at a time because they are in different functions.

Passing Arguments to Functions

If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables. A *parameter variable*, often simply called a *parameter*, is a special variable that is assigned the value of an argument when a function is called. Here is an example of a function that has a parameter variable:

```
def double_number(value):
    result = value * 2
    print(result)
```

This function's name is `double_number`. Its purpose is to accept a number as an argument and display the value of that number doubled. Look at the function header and notice the word `value` that appear inside the parentheses. This is the name of a parameter variable. This variable will be assigned the value of an argument when the function is called. Program 3-4 demonstrates the function in a program.

Program 3-4 (pass_integer.py)

```
1 # Define the main function.
2 def main():
3     double_number(4)
4
5 # Define the double_number function.
6 def double_number(value):
7     result = value * 2
8     print(result)
9
10 # Call the main function.
11 main()
```

This is the Python version of
Program 3-5 in your textbook.

Program Output

8

When this program runs, the main function is called in line 11. Inside the main function, line 3 calls the `double_number` function passing the value 4 as an argument.

The `double_number` function is defined in lines 6 through 8. The function has a parameter variable named `value`. In line 7 a local variable named `result` is assigned the value of the math expression `value * 2`. In line 8 the value of the `result` variable is displayed.

Program 3-5 shows another example using the `double_number` function.

Program 3-5 (pass_arg.py)

```
1 # Define the main function.
2 def main():
3     number = int(input('Enter a number and I will display that number doubled: '))
4     double_number(number)
5
6 # Define the double_number
function.
7 def double_number(value):
8     result = value * 2
9     print(result)
10
11 # Call the main function.
12 main()
```

This is the Python version of
Program 3-6 in your textbook.

Program Output

Enter a number and I will display that number doubled: 20 [Enter]

40

When this program runs, the main function is called in line 12. Inside the main function, line 3 gets a number from the user and assigns it to the `number` variable. Line 4 calls the `double_number` function passing the `number` variable as an argument.

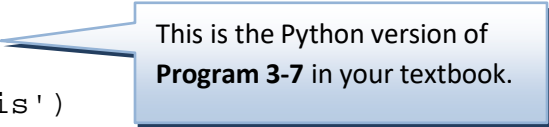
The `double_number` function is defined in lines 7 through 9. The function has a parameter variable named `value`. In line 8 a local variable named `result` is assigned the value of the math expression `value * 2`. In line 9 the value of the `result` variable is displayed.

Passing Multiple Arguments

Often it is useful to pass more than one argument to a function. When you define a function, you must have a parameter variable for each argument that you want passed into the function. Program 3-6 shows an example. This is the Python version of pseudocode Program 3-7 in your textbook.

Program 3-6 (*multiple_args.py*)

```
1  # This program demonstrates a function that accepts
2  # two arguments.
3
4  def main():
5      print('The sum of 12 and 45 is')
6      show_sum(12, 45)
7
8  # The show_sum function accepts two arguments
9  # and displays their sum.
10 def show_sum(num1, num2):
11     result = num1 + num2
12     print(result)
13
14 # Call the main function.
15 main()
```



This is the Python version of
Program 3-7 in your textbook.

Program Output

```
The sum of 12 and 45 is
57
```

Notice that two parameter variable names, `num1` and `num2`, appear inside the parentheses in the `show_sum` function header. This is often referred to as a *parameter list*. Also notice that a comma separates the variable names.

The statement in line 6 calls the `show_sum` function and passes two arguments: 12 and 45. These arguments are passed *by position* to the corresponding parameter variables in the function. In other words, the first argument is passed to the first parameter variable, and the second argument is passed to the second parameter variable. So, this statement causes 12 to be assigned to the `num1` parameter and 45 to be assigned to the `num2` parameter.

Making Changes to Parameters

When an argument is passed to a function in Python, the function parameter variable will reference the argument's value. However, any changes that are made to the parameter variable will not affect the argument. To demonstrate this look at Program 3-7.

Program 3-7 (*change_me.py*)

```
1  # This program demonstrates what happens when you
2  # change the value of a parameter.
3
4  def main():
5      value = 99
6      print('The value is', value)
7      change_me(value)
8      print('Back in main the value is', value)
9
10 def change_me(arg):
11     print('I am changing the value.')
12     arg = 0
13     print('Now the value is', arg)
14
15 # Call the main function.
16 main()
```

Program Output

```
The value is 99
I am changing the value.
Now the value is 0
Back in main the value is 99
```

The `main` function creates a local variable named `value` in line 5, assigned the value 99. The statement in line 6 displays *The value is 99*. The `value` variable is then passed as an argument to the `change_me` function in line 7. This means that in the `change_me` function the `arg` parameter will also reference the value 99.

Global Variables

When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is *global*. A global variable can be accessed by any statement in the program file, including the statements in any function. For example, look at Program 3-8.

Program 3-8 (*global1.py*)

```
1  # Create a global variable.
2  my_value = 10
3
4  # The show_value function prints
```

```

5 # the value of the global variable.
6 def show_value():
7     print(my_value)
8
9 # Call the show_value function.
10 show_value()

```

Program Output

```
10
```

The assignment statement in line 2 creates a variable named `my_value`. Because this statement is outside any function, it is global. When the `show_value` function executes, the statement in line 7 prints the value referenced by `my_value`.

An additional step is required if you want a statement in a function to assign a value to a global variable. In the function you must declare the global variable, as shown in Program 3-9.

Program 3-9 (*global2.py*)

```

1 # Create a global variable.
2 number = 0
3
4 def main():
5     global number
6     number = int(input('Enter a number: '))
7     show_number()
8
9 def show_number():
10    print('The number you entered is', number)
11
12 # Call the main function.
13 main()

```

This is the Python version of
Program 3-12 in your textbook.

Program Output

```

Enter a number: 22 [Enter]
The number you entered is 22

```

The assignment statement in line 2 creates a global variable named `number`. Notice that inside the `main` function, line 5 uses the `global` key word to declare the `number` variable. This statement tells the interpreter that the `main` function intends to assign a value to the global `number` variable. That's just what happens in line 6. The value entered by the user is assigned to `number`.

Global Constants

The Python language does not allow you to create true global constants, but you can simulate them with global variables. If you do not declare a global variable with the `global` key word inside a function, then you cannot change the variable's assignment.

Chapter 4 Decision Structures and Boolean Logic

Relational Operators and the `if` Statement

Python's relational operators, shown in Table 4-1, are exactly the same as those discussed in your textbook.

Table 4-1 Relational Operators

Operator	Meaning
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code>==</code>	Equal to
<code>!=</code>	Not equal to

The relational operators are used to create Boolean expressions, and are commonly used with `if` statements. Here is the general format of the `if` statement in Python:

```
if condition:
    statement
    statement
    etc.
```

For simplicity, we will refer to the first line as the *if clause*. The `if` clause begins with the word `if`, followed by a *condition*, which is an expression that will be evaluated as either true or false. A colon appears after the *condition*. Beginning at the next line is a block of statements. All of the statements in a block must be consistently indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

When the `if` statement executes, the *condition* is tested. If the *condition* is true, the statements that appear in the block following the `if` clause are executed. If the condition is false, the statements in the block are skipped.

Program 4-1 demonstrates the `if` statement. This Python program is similar to pseudocode Program 4-1 in your textbook.

Program 4-1 (test_average.py)

```
1  # This program prompts the user to enter three test
2  # scores. It displays the average of those scores and
3  # and congratulates the user if the average is 95
4  # or greater.
5
6  def main():
7      # Get the three test scores.
8      test1 = float(input('Enter the score for test 1: '))
9      test2 = float(input('Enter the score for test 2: '))
10     test3 = float(input('Enter the score for test 3: '))
11
12     # Calculate the average test score.
13     average = (test1 + test2 + test3) / 3.0
14
15     # Print the average.
16     print('The average score is', average)
17
18     # If the average is 95 or greater,
19     # congratulate the user.
20     if average >= 95:
21         print('Congratulations!')
22         print('That is a great average!')
23
24 # Call the main function.
25 main()
```

This is the Python version of
Program 4-1 in your textbook.

Program Output (with Input Shown in Bold)

```
Enter the score for test 1: 82 [Enter]
Enter the score for test 2: 76 [Enter]
Enter the score for test 3: 91 [Enter]
The average score is 83.0
```

Program Output (with Input Shown in Bold)

```
Enter the score for test 1: 93 [Enter]
Enter the score for test 2: 99 [Enter]
Enter the score for test 3: 96 [Enter]
The average score is 96.0
Congratulations!
That is a great score.
```

Dual Alternative Decision Structures

You use the `if-else` statement in Python to create a dual alternative decision structure. This is the format of the `if-else` statement:

```

if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.

```

When this statement executes, the *condition* is tested. If it is true, the block of indented statements following the *if* clause is executed, and then control of the program jumps to the statement that follows the *if-else* statement. If the condition is false, the block of indented statements following the *else* clause is executed, and then control of the program jumps to the statement that follows the *if-else* statement.

Program 4-2 shows an example. This is the Python version of pseudocode Program 4-2 in your textbook. The program gets the number of hours that the user has worked (line 11) and the user's hourly pay rate (line 12). The *if-else* statement in lines 15 through 18 determines whether the user has worked more than 40 hours. If so, the `calc_pay_with_OT` function is called in line 16. Otherwise the `calc_regular_pay` function is called in line 18.

Program 4-2 (*auto_repair_payroll.py*)

```

1 # Global constants
2 BASE_HOURS = 40          # Base hours per week
3 OT_MULTIPLIER = 1.5      # Overtime multiplier
4
5 # The main function gets the number of hours worked and
6 # the hourly pay rate. It calls either the calc_pay_with_OT
7 # function or the calc_regular_pay function to calculate
8 # and display the gross pay.
9 def main():
10     # Get the hours worked and the hourly pay rate.
11     hours_worked = float(input('Enter the number of hours worked: '))
12     pay_rate = float(input('Enter the hourly pay rate: '))
13
14     # Calculate and display the gross pay.
15     if hours_worked > BASE_HOURS:
16         calc_pay_with_OT(hours_worked, pay_rate)
17     else:
18         calc_regular_pay(hours_worked, pay_rate)
19
20 # The calc_pay_with_OT function calculates pay with
21 # overtime. It accepts the hours worked and the hourly
22 # pay rate as arguments. The gross pay is displayed.
23 def calc_pay_with_OT(hours, rate):
24     # Calculate the number of overtime hours worked.
25     overtime_hours = hours - BASE_HOURS
26
27     # Calculate the amount of overtime pay.
28     overtime_pay = overtime_hours * rate * OT_MULTIPLIER
29

```

This is the Python version of
Program 4-2 in your textbook.


```

30     # Calculate the gross pay.
31     gross_pay = BASE_HOURS * rate + overtime_pay
32
33     # Display the gross pay.
34     print('The gross pay is $', format(gross_pay, '.2f'))
35
36 # The calc_regular_pay function calculates pay with
37 # no overtime. It accepts the hours worked and the hourly
38 # pay rate as arguments. The gross pay is displayed.
39 def calc_regular_pay(hours, rate):
40     # Calculate the gross pay.
41     gross_pay = hours * rate
42
43     # Display the gross pay.
44     print('The gross pay is $', format(gross_pay, '.2f'))
45
46 # Call the main function.
47 main()

```

Program Output (with Input Shown in Bold)

```

Enter the number of hours worked: 40 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $800.00

```

Program Output (with Input Shown in Bold)

```

Enter the number of hours worked: 50 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $1100.00

```

Comparing Strings

The relational operators can be used to compare strings, as shown here:

```

name1 = 'Mary'
name2 = 'Mark'
if name1 == name2:
    print('The names are the same.')
else:
    print('The names are NOT the same.')

```

The == operator compares name1 and name2 to determine whether they are equal. Because the strings 'Mary' and 'Mark' are not equal, the else clause will display the message 'The names are NOT the same.'

Let's look at another example. Assume the month variable references a string. The following code uses the != operator to determine whether the value referenced by month is not equal to 'October'.

```

if month != 'October':
    print('This is the wrong time for Octoberfest!')

```

Program 4-3 is a complete program demonstrating how two strings can be compared. This is the Python version of pseudocode Program 4-3 in your textbook. The program prompts the user to enter a password and then determines whether the string entered is equal to 'prospero'.

Program 4-3 (*password.py*)

```
1 # This program demonstrates how the == operator can
2 # be used to compare strings.
3
4 def main():
5     # Get a password from the user.
6     password = input('Enter the password: ')
7
8     # Determine whether the correct password
9     # was entered.
10    if password == 'prospero':
11        print('Password accepted.')
12    else:
13        print('Sorry, that is the wrong password.')
14
15 # Call the main function.
16 main()
```

This is the Python version of
Program 4-3 in your textbook.

Program Output (with Input Shown in Bold)

Enter the password: **ferdinand** [Enter]
Sorry, that is the wrong password.

Program Output (with Input Shown in Bold)

Enter the password: **prospero** [Enter]
Password accepted.

String comparisons are case sensitive. For example, the strings 'saturday' and 'Saturday' are not equal because the "s" is lowercase in the first string, but uppercase in the second string. The following sample session with Program 4-3 shows what happens when the user enters Prospero as the password (with an uppercase P).

Program Output (with Input Shown in Bold)

Enter the password: **Prospero** [Enter]
Sorry, that is the wrong password.

Nested Decision Structures

Program 4-4 shows an example of nested decision structures. As noted in your textbook, this type of nested decision structure can also be written as an `if-else-if` statement, as shown in Program 4-5.

Program 4-4 (*nested_if.py*)

```
1 def main():
2     # Prompt the user to enter the temperature.
3     temp = float(input('What is the temperature outside? '))
4
5     # Determine the type of weather we're having.
6     if temp < 30:
7         print('Wow. That is cold.')
8     else:
9         if temp < 50:
10            print('A little chilly.')
11        else:
12            if temp < 80:
13                print('Nice and warm.')
14            else:
15                print('Whew! It is hot!')
16
17 # Call the main function.
18 main()
```

Program Output (with Input Shown in Bold)

What is the outside temperature? **20** [Enter]
Wow! That is cold!

Program Output (with Input Shown in Bold)

What is the outside temperature? **45** [Enter]
A little chilly.

Program Output (with Input Shown in Bold)

What is the outside temperature? **70** [Enter]
Nice and warm.

Program Output (with Input Shown in Bold)

What is the outside temperature? **90** [Enter]
Whew! It is hot!

Program 4-5 (if_elif_else.py)

```
1 def main():
2     # Prompt the user to enter the temperature.
3     temp = float(input('What is the temperature outside? '))
4
5     # Determine the type of weather we're having.
6     if temp < 30:
7         print('Wow. That is cold.')
8     elif temp < 50:
9         print('A little chilly.')
10    elif temp < 80:
11        print('Nice and warm.')
12    else:
13        print('Whew! It is hot!')
14
15 # Call the main function.
16 main()
```

Program Output (with Input Shown in Bold)

What is the outside temperature? **20** [Enter]
Wow! That is cold!

Program Output (with Input Shown in Bold)

What is the outside temperature? **45** [Enter]
A little chilly.

Program Output (with Input Shown in Bold)

What is the outside temperature? **70** [Enter]
Nice and warm.

Program Output (with Input Shown in Bold)

What is the outside temperature? **90** [Enter]
Whew! It is hot!

Note: Python does not provide a Case structure, so we can't cover that topic in this booklet.

Logical Operators

Table 4-2 shows Python's logical operators, which work like the ones discussed in your textbook.

Table 4-2 Python's Logical Operators

Operator	Meaning
<code>and</code>	The <code>and</code> operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
<code>or</code>	The <code>or</code> operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
<code>not</code>	The <code>not</code> operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The <code>not</code> operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

For example, the following `if` statement checks the value of `x` to determine if it is in the range of 20 through 40:

```
if x >= 20 and x <= 40:
    print(x, 'is in the acceptable range.')
```

The Boolean expression in the `if` statement will be true only when `x` is greater than or equal to 20 *AND* less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be true. The following statement determines whether `x` is outside the range of 20 through 40:

```
if x < 20 or x > 40:
    print(x, 'is outside the acceptable range.')
```

Here is an `if` statement using the `not` operator:

```
if not (temperature > 100):
    print('This is below the maximum temperature.')
```

First, the expression `(temperature > 100)` is tested and a value of either `true` or `false` is the result. Then the `not` operator is applied to that value. If the expression `(temperature > 100)` is `true`, the `not` operator returns `false`. If the expression `(temperature > 100)` is `false`, the `not` operator returns `true`. The previous code is equivalent to asking: “Is the temperature not greater than 100?”

Boolean Variables

The `bool` data type allows you to create variables that may reference one of two possible values: `True` or `False`. Here are examples of how we assign values to a `bool` variable:

```
hungry = True
sleepy = False
```

Boolean variables are most commonly used as flags that signals when some condition exists in the program. When the flag variable is set to `False`, it indicates the condition does not exist. When the flag variable is set to `True`, it means the condition does exist.

For example, suppose a salesperson has a quota of \$50,000. Assuming `sales` references the amount that the salesperson has sold, the following code determines whether the quota has been met:

```
if sales >= 50000.0:
    sales_quota_met = True
else:
    sales_quota_met = False
```

As a result of this code, the `sales_quota_met` variable can be used as a flag to indicate whether the sales quota has been met. Later in the program we might test the flag in the following way:

```
if sales_quota_met:
    print('You have met your sales quota!')
```

This code displays 'You have met your sales quota!' if the `bool` variable `sales_quota_met` is `True`. Notice that we did not have to use the `==` operator to explicitly compare the `sales_quota_met` variable with the value `True`. This code is equivalent to the following:

```
if sales_quota_met == True:
    print('You have met your sales quota!')
```

Chapter 5 Repetition Structures

The while Loop

In Python, the `while` loop is written in the following general format:

```
while condition:  
    statement  
    statement  
    etc.
```

We will refer to the first line as the *while clause*. The `while` clause begins with the word `while`, followed by a Boolean *condition* that will be evaluated as either true or false. A colon appears after the *condition*. Beginning at the next line is a block of statements. All of the statements in a block must be consistently indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

When the `while` loop executes, the *condition* is tested. If the *condition* is true, the statements that appear in the block following the `while` clause are executed, and then the loop starts over. If the *condition* is false, the program exits the loop.

Program 5-1 shows an example of the `while` loop. This is the Python version of pseudocode Program 5-2 in your textbook.

Program 5-1 (*temperature.py*)

```
1 # This program assists a technician in the process  
2 # of checking a substance's temperature.  
3  
4 # MAX_TEMP is used as a global constant for  
5 # the maximum temperature.  
6 MAX_TEMP = 102.5  
7  
8 # The main function  
9 def main():  
10     # Get the substance's temperature.  
11     temperature = float(input("Enter the substance's Celsius temperature: "))  
12  
13     # As long as necessary, instruct the user to  
14     # adjust the thermostat.  
15     while temperature > MAX_TEMP:  
16         print('The temperature is too high.')  
17         print('Turn the thermostat down and wait')  
18         print('5 minutes. Then take the temperature')  
19         print('again and enter it.')  
20         temperature = float(input('Enter the new Celsius temperature: '))  
21  
22     # Remind the user to check the temperature again  
23     # in 15 minutes.  
24     print('The temperature is acceptable.')  
25     print('Check it again in 15 minutes.')  
26
```

This is the Python version of
Program 5-2 in your textbook.

```
27 # Call the main function.  
28 main()
```

Program Output (with Input Shown in Bold)

```
Enter the substance's temperature: 104.7 [Enter]  
The temperature is too high.  
Turn the thermostat down and wait  
5 minutes. Take the temperature  
again and enter it.  
Enter the new temperature: 103.2 [Enter]  
The temperature is too high.  
Turn the thermostat down and wait  
5 minutes. Take the temperature  
again and enter it.  
Enter the new temperature: 102.1 [Enter]  
The temperature is acceptable.  
Check it again in 15 minutes.
```

Program Output (with Input Shown in Bold)

```
Enter the substance's temperature: 102.1 [Enter]  
The temperature is acceptable.  
Check it again in 15 minutes.
```

The for Loop

In Python, the `for` statement is designed to work with a sequence of data items. When the statement executes, it iterates once for each item in the sequence. We will use the `for` statement in the following general format:

```
for variable in [value1, value2, etc.]:  
    statement  
    statement  
    etc.
```

We will refer to the first line as the *for clause*. In the `for` clause, *variable* is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value. (In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a *list*.) Beginning at the next line is a block of statements that is executed each time the loop iterates.

The `for` statement executes in the following manner: The *variable* is assigned the first value in the list, and then the statements that appear in the block are executed. Then, *variable* is assigned the next value in the list, and the statements in the block are executed again. This

continues until *variable* has been assigned the last value in the list. Program 5-2 shows a simple example that uses a `for` loop to display the numbers 1 through 5.

Program 5-2 (*simple_loop1.py*)

```
1 # This program demonstrates a simple for loop
2 # that uses a list of numbers.
3
4 def main():
5     print('I will display the numbers 1 through 5.')
6     for num in [1, 2, 3, 4, 5]:
7         print(num)
8
9 # Call the main function.
10 main()
```

Program Output

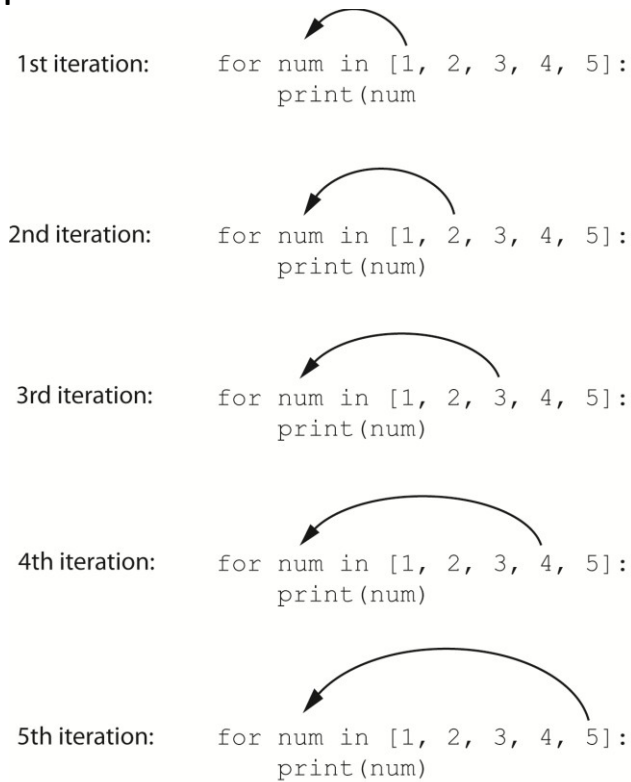
```
I will display the numbers 1 through 5.
1
2
3
4
5
```

The first time the `for` loop iterates, the `num` variable is assigned the value 1 and then the statement in line 7 executes (displaying the value 1). The next time the loop iterates, `num` is assigned the value 2, and the `print` function is called (displaying the value 2). This process continues, as shown in Figure 5-1, until `num` has been assigned the last value in the list. Because the list contains five values, the loop will iterate five times.

Python programmers commonly refer to the variable that is used in the `for` clause as the *target variable* because it is the target of an assignment at the beginning of each loop iteration

The values that appear in the list do not have to be a consecutively ordered series of numbers. For example, Program 5-3 uses a `for` loop to display a list of odd numbers. There are five numbers in the list, so the loop iterates five times.

Figure 5-1 The for loop



Program 5-3 (*simple_loop2.py*)

```
1 # This program also demonstrates a simple for
2 # loop that uses a list of numbers.
3
4 def main():
5     print('I will display the odd numbers 1 through 9.')
6     for num in [1, 3, 5, 7, 9]:
7         print(num)
8
9 # Call the main function.
10 main()
```

Program Output

```
I will display the odd numbers 1 through 9.
1
3
5
7
9
```

Program 5-4 shows another example. In this program the `for` loop iterates over a list of strings. Notice that the list (in line 5) contains the three strings 'Winken', 'Blinken', and 'Nod'. As a result, the loop iterates three times.

Program 5-4 (*simple_loop3.py*)

```
1 # This program also demonstrates a simple for
2 # loop that uses a list of numbers.
3
4 def main():
5     for name in ['Winken', 'Blinken', 'Nod']:
6         print(name)
7
8 # Call the main function.
9 main()
```

Program Output

```
Winken
Blinken
Nod
```

Using the `range` Function with the `for` Loop

Python provides a built-in function named `range` that simplifies the process of writing a count-controlled `for` loop. Here is an example of a `for` loop that uses the `range` function:

```
for num in range(5):
    print(num)
```

Notice that instead of using a list of values, we call to the `range` function passing 5 as an argument. In this statement the `range` function will generate a list of integers in the range of 0 up to (but not including) 5. This code works the same as the following:

```
for num in [0, 1, 2, 3, 4]:
    print(num)
```

As you can see, the list contains five numbers, so the loop will iterate five times. Program 5-5 uses the `range` function with a `for` loop to display "Hello world" five times.

Program 5-5 (*simple_loop4.py*)

```
1 # This program demonstrates how the range
2 # function can be used with a for loop.
3
4 def main():
5     # Print a message five times.
6     for x in range(5):
```

This is the Python version of
Program 5-8 in your textbook.

```
7         print('Hello world!')
8
9 # Call the main function.
10 main()
```

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

If you pass one argument to the `range` function, as demonstrated in Program 5-5, that argument is used as the ending limit of the list. If you pass two arguments to the `range` function, the first argument is used as the starting value of the list and the second argument is used as the ending limit. Here is an example:

```
for num in range(1, 5):
    print(num)
```

This code will display the following:

```
1
2
3
4
```

By default, the `range` function produces a list of numbers that increase by 1 for each successive number in the list. If you pass a third argument to the `range` function, that argument is used as *step value*. Instead of increasing by 1, each successive number in the list will increase by the step value. Program 5-6 shows an example:

Program 5-6 (odd_numbers.py)

```
1 for num in range(1, 12, 2):
2     print(num)
```

This is the Python version of
Program 5-10 in your textbook.

Program Output

```
1
3
5
7
9
11
```

In this `for` statement, three arguments are passed to the `range` function:

- The first argument, 1, is the starting value for the list.
- The second argument, 12, is the ending limit of the list. This means that the last number in the list will be 11.
- The third argument, 2, is the step value. This means that 2 will be added to each successive number in the list.

Using the Target Variable inside the Loop

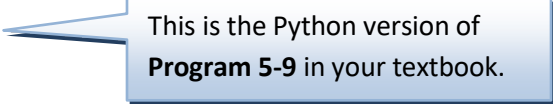
In a `for` loop, the purpose of the target variable is to reference each item in a sequence of items as the loop iterates. In many situations it is helpful to use the target variable in a calculation or other task within the body of the loop. For example, suppose you need to write a program that displays the numbers 1 through 10 and their squares, in a table similar to the following:

<u>Number</u>	<u>Square</u>
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

This can be accomplished by writing a `for` loop that iterates over the values 1 through 10. During the first iteration, the target variable will be assigned the value 1, during the second iteration it will be assigned the value 2, and so forth. Because the target variable will reference the values 1 through 10 during the loop's execution, you can use it in the calculation inside the loop. Program 5-7 shows how this is done.

Program 5-7 (*squares.py*)

```
1 # This program uses a loop to display a
2 # table showing the numbers 1 through 10
3 # and their squares.
4
5 def main():
6     # Print the table headings.
7     print('Number\tSquare')
8     print('-----')
9
10    # Print the numbers 1 through 10
```



This is the Python version of
Program 5-9 in your textbook.

```

11     # and their squares.
12     for number in range(1, 11):
13         square = number**2
14         print(number, '\t', square)
15
16 # Call the main function.
17 main()

```

Program Output

Number	Square
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

First, take a closer look at line 7, which displays the table headings:

```
print('Number\tSquare')
```

Notice that inside the string literal the `\t` escape sequence between the words `Number` and `Square`. These are special formatting characters known as the *tab escape sequence*. The escape sequence works similarly to the word `Tab` that is used in pseudocode in your textbook. As you can see in the program output, the `"\t"` characters are not displayed on the screen, but rather cause the output cursor to "tab over." It is useful for aligning output in columns on the screen.


The `for` loop that begins in line 12 uses the `range` function to produce a list containing the numbers 1 through 10. During the first iteration, `number` will reference 1, during the second iteration `number` will reference 2, and so forth, up to 10. Inside the loop, the statement in line 13 raises `number` to the power of 2 (recall that `**` is the exponent operator), and assigns the result to the `square` variable. The statement in line 14 prints the value referenced by `number`, tabs over, and then prints the value referenced by `square`. (Tabbing over with the `\t` escape sequence causes the numbers to be aligned in two columns in the output.)

Calculating a Running Total

Your textbook discusses the common programming task of calculating the sum of a series of values, also known as calculating a running total. Program 5-8 demonstrates how this can be done in Python. The `total` variable that is declared in line 6 is the accumulator variable. Notice that it is initialized with the value 0. During each loop iteration the user enters a number, and in line 15 this number is added to the value already stored in `total`. The `total` variable accumulates the sum of all the numbers entered by the user. This program is the Python version of pseudocode Program 5-18 in your textbook.

Program 5-14 (*sum_numbers.py*)

```
1  # This program calculates the sum of
2  # five numbers entered by the user.
3
4  def main():
5      # Initialize an accumulator variable.
6      total = 0.0
7
8      # Explain what we are doing.
9      print('This program calculates the sum of')
10     print('five numbers you will enter.')
11
12     # Get five numbers and accumulate them.
13     for counter in range(5):
14         number = int(input('Enter a number: '))
15         total = total + number
16
17     # Display the total of the numbers.
18     print('The total is', total)
19
20 # Call the main function.
21 main()
```



This is the Python version
of Program 5-18 in your
textbook.

Program Output (with Example Input Shown in Bold)

```
This program calculates the sum of
five numbers you will enter.
Enter a number: 1 [Enter]
Enter a number: 2 [Enter]
Enter a number: 3 [Enter]
Enter a number: 4 [Enter]
Enter a number: 5 [Enter]
The total is 15.0
```

Chapter 6 Value-Returning Functions

The functions that you learned about in Chapter 3 of this booklet are simple functions that do not return a value. A function can also be written to return a value to the statement that called the function. We refer to such a function as a value-returning function.

Generating Random Integers

Python provides several library functions for working with random numbers. To use any of these functions you first need to write this `import` statement at the top of your program:

```
import random
```

The first random-number generating function that we will discuss is `random.randint`. The following statement shows an example of how you might call the `random.randint` function.

```
number = random.randint(1, 100)
```

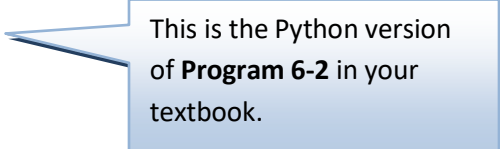
The part of the statement that reads `random.randint(1, 100)` is a call to the function. Notice that two arguments appear inside the parentheses: 1 and 100. These arguments tell the function to give an integer random number in the range of 1 through 100. (The values 1 and 100 are included in the range.)

Notice that the call to the `random.randint` function appears on the right side of an `=` operator. When the function is called, it will generate a random number in the range of 1 through 100 and then *return* that number. The number that is returned will be assigned to the `number` variable.

Program 6-1 shows a complete demonstration. This is the Python version of pseudocode Program 6-2 in your textbook. This program uses a `for` loop that iterates five times. Inside the loop, the statement in line 8 calls the `random.randint` function to generate a random number in the range of 1 through 100.

Program 6-2 (*random_numbers2.py*)

```
1  # This program displays five random
2  # numbers in the range of 1 through 100.
3  import random
4
5  def main():
6      for count in range(5):
7          # Get a random number.
8          number = random.randint(1, 100)
9          # Display the number.
10         print(number)
11
12 # Call the main function.
13 main()
```



This is the Python version
of **Program 6-2** in your
textbook.

Program Output

```
89
7
16
41
12
```

Other Random Number Functions

The standard library contains numerous functions for working with random numbers. In addition to the `random.randint` function, you might find the `random.randrange`, `random`, and `random.uniform` functions useful. (To use any of these functions you need to write `import random` at the top of your program.)

If you remember how to use the `range` function (which we discussed in Chapter 5) then you will immediately be comfortable with the `random.randrange` function. The `random.randrange` function takes the same arguments as the `range` function. The difference is that the `random.randrange` function does not return a list of values. Instead, it returns a randomly selected value from a sequence of values. For example, the following statement assigns a random number in the range of 0 through 9 to the `number` variable:

```
number = random.randrange(10)
```

The argument, in this case 10, specifies the ending limit of the sequence of values. The function will return a randomly-selected number from the sequence of values 0 up to, but not including, the ending limit. The following statement specifies both a starting value and an ending limit for the sequence:

```
number = random.randrange(5, 10)
```

When this statement executes, a random number in the range of 5 through 9 will be assigned to `number`. The following statement specifies a starting value, an ending limit, and a step value:

```
number = random.randrange(0, 101, 10)
```

In this statement the `random.randrange` function returns a randomly selected value from the following sequence of numbers:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Both the `random.randint` and the `random.randrange` functions return an integer number. The `random.random` function returns, however, returns a random floating-point number. You do not pass any arguments to the `random.random` function. When you call it, it returns a random floating point number in the range of 0.0 up to 1.0 (but not including 1.0). Here is an example:

```
number = random.random()
```

The `random.uniform` function also returns a random floating-point number, but allows you to specify the range of values to select from. Here is an example:

```
number = random.uniform(1.0, 10.0)
```

In this statement the `random.uniform` function returns a random floating-point number in the range of 1.0 through 10.0 and assigns it to the `number` variable.

Writing Your Own Value-Returning Methods

Up to now, all of the functions that you have written have been simple functions that do not return a value. You write a value-returning function in the same way that you write a simple function, with one exception: a value-returning function must have a `return` statement. Here is the general format of a value-returning function definition in Python:

```
def function_name( ) :  
    statement  
    statement  
    etc.  
    return expression
```

One of the statements in the function must be a `return` statement, which takes the following form:

```
return expression
```

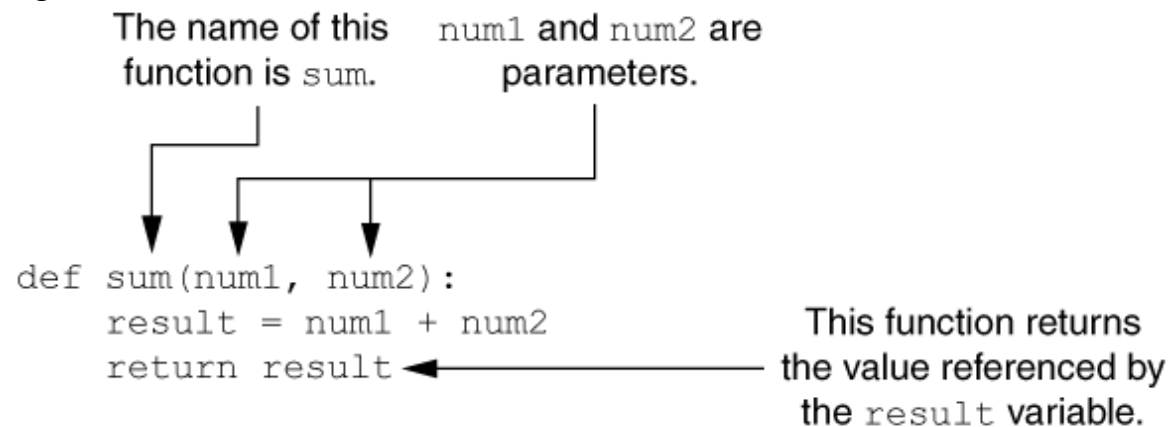
The value of the *expression* that follows the key word `return` will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value (such as a math expression).

Here is a simple example of a value-returning function:

```
def sum(num1, num2):  
    result = num1 + num2  
    return result
```

Figure 6-1 illustrates various parts of the function.

Figure 6-1 Parts of the function



The purpose of this function is to accept two integer values as arguments and return their sum. Let's take a closer look at how it works. The first statement in the function's block assigns the value of `num1 + num2` to the `result` variable. Next, the `return` statement executes, which causes the function to end execution and sends the value referenced by the `result` variable back to the part of the program that called the function. Program 6-3 demonstrates the function.

Program 6-2 (*total_ages.py*)

```
1 # This program uses the return value of a function.  
2  
3 def main():  
4     # Get the user's age.  
5     first_age = int(input('Enter your age: '))  
6  
7     # Get the user's best friend's age.  
8     second_age = int(input("Enter your best friend's age: "))  
9  
10    # Get the sum of both ages.
```

This is the Python version of **Program 6-6** in your textbook.

```

11     total = sum(first_age, second_age)
12
13     # Display the total age.
14     print('Together you are', total, 'years old.')
15
16 # The sum function accepts two numeric arguments and
17 # returns the sum of those arguments.
18 def sum(num1, num2):
19     result = num1 + num2
20     return result
21
22 # Call the main function.
23 main()

```

Program Output (with Input Shown in Bold)

```

Enter your age: 22 [Enter]
Enter your best friend's age: 24 [Enter]
Together you are 46 years old.

```

Returning Strings

So far you've seen examples of functions that return numbers. You can also write functions that return strings. For example, the following function prompts the user to enter his or her name, and then returns the string that the user entered.

```

def get_name():
    # Get the user's name.
    name = input('Enter your name: ')

    # Return the name.
    return name

```

Returning Boolean Values

Python allows you to write *Boolean functions*, which return either `True` or `False`. For example, the following function accepts a number as an argument, and returns `True` if the argument is an even number. Otherwise, it returns `False`.

```
def is_even(number):
    # Determine whether number is even. If it is,
    # set status to true. Otherwise, set status
    # to false.
    if (number % 2) == 0:
        status = True
    else:
        status = False

    # Return the value of the status variable.
    return status
```

The following code gets a number from the user, and then calls the function to determine whether the number is even or odd:

```
value = int(input('Enter a number: '))
if is_even(value):
    print('The number is even.')
else:
    print('The number is odd.')
```

Math Functions

The Python standard library contains several functions that are useful for performing mathematical operations. Table 6-2 lists many of the math functions. To use these functions you first need to write this `import` statement at the top of your program:

```
import math
```

These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result. For example, one of the functions is named `math.sqrt`. The `math.sqrt` function accepts an argument and returns the square root of the argument. Here is an example of how it is used:

```
result = math.sqrt(16)
```

Table 6-2 Many of the functions in the math module

Function	Description
<code>math.acos(x)</code>	Returns the arc cosine of <i>x</i> , <i>in radians</i> .
<code>math.asin(x)</code>	Returns the arc sine of <i>x</i> , <i>in radians</i> .
<code>math.atan(x)</code>	Returns the arc tangent of <i>x</i> , <i>in radians</i> .
<code>math.ceil(x)</code>	Returns the smallest integer that is greater than or equal to <i>x</i> .
<code>math.cos(x)</code>	Returns the cosine of <i>x</i> in radians.
<code>math.degrees(x)</code>	Assuming <i>x</i> is an angle in radians, the function returns the angle converted to degrees.

<code>math.exp(x)</code>	Returns e^x
<code>math.floor(x)</code>	Returns the largest integer that is less than or equal to x .
<code>math.hypot(x, y)</code>	Returns the length of a hypotenuse that extends from (0, 0) to (x, y).
<code>math.log(x)</code>	Returns the natural logarithm of x .
<code>math.log10(x)</code>	Returns the base-10 logarithm of x .
<code>math.radians(x)</code>	Assuming x is an angle in degrees, the function returns the angle converted to radians.
<code>math.sin(x)</code>	Returns the sine of x in radians.
<code>math.sqrt(x)</code>	Returns the square root of x .
<code>math.tan(x)</code>	Returns the tangent of x in radians.

The `math.pi` and `math.e` Values

The Python library also defines two variables, `math.pi` and `math.e`, which are assigned mathematical values for π and e . You can use these variables in equations that require their values. For example, the following statement, which calculates the area of a circle, uses `pi`. (Notice that we use dot notation to refer to the variable.)

```
area = math.pi * radius**2
```

Formatting Numbers

Chapter 6 in your textbook also covers formatting functions, and gives an example for formatting a number to look like a currency amount. In Python 3, we use the built-in `format` function to format numbers for output.

When you call the built-in `format` function, you pass it two arguments: a numeric value, and a format specifier. The *format specifier* is a string that contains special characters specifying how the numeric value should be formatted. Let's look at an example:

```
format(12345.6789, '.2f')
```

The first argument, which is the floating-point number 12345.6789, is the number that we want to format. The second argument, which is the string `'.2f'`, is the format specifier. Here's the meaning of its contents:

- The `.2` specifies the precision. It indicates that we want to round the number to two decimal places.
- The `f` specifies that the data type of the number we are formatting is a floating-point number. (If you are formatting an integer, you cannot use `f` for the type. We will discuss integer formatting momentarily.)

The `format` function returns a string containing the formatted number. The following interactive mode session demonstrates how you use the `format` function along with the `print` function to display a formatted number:

```
>>> print(format(12345.6789, '.2f')) [Enter]
12345.68
>>>
```

Notice that the number is rounded to two decimal places. The following example shows the same number, rounded to one decimal place:

```
>>> print(format(12345.6789, '.1f')) [Enter]
12345.7
>>>
```

Here is an example that prints multiple items, one of which is a formatted number:

```
>>> print('The number is', format(1.234567, '.2f')) [Enter]
The number is 1.23
>>>
```

In Program 6-3 a value is displayed rounded to 1, 2, 3, 4, 5, and 6 decimal places.

Program 6-3 (*decimal_places.py*)

```
1  # This program demonstrates how a value can be
2  # formatted, rounded to different numbers of
3  # decimal places.
4
5  def main():
6      my_value = 1.123456789
7      print(format(my_value, '.1f'))    # Rounded to 1 decimal place
8      print(format(my_value, '.2f'))    # Rounded to 2 decimal places
9      print(format(my_value, '.3f'))    # Rounded to 3 decimal places
10     print(format(my_value, '.4f'))    # Rounded to 4 decimal places
11     print(format(my_value, '.5f'))    # Rounded to 5 decimal places
12     print(format(my_value, '.6f'))    # Rounded to 6 decimal places
13
14 # Call the main function.
15 main()
```

Program Output

```
1.1
1.12
1.123
1.1235
1.12346
1.123457
```

Inserting Comma Separators

If you want the number to be formatted with comma separators, you can insert a comma into the format specifier, as shown here:

```
>>> print(format(12345.6789, ',.2f')) [Enter]
12,345.68
>>>
```

Here is an example that formats an even larger number:

```
>>> print(format(123456789.456, ',.2f')) [Enter]
123,456,789.46
>>>
```

String Functions

Getting a String's length

You use the `len` function to get the length of a string. The following code demonstrates:

```
name = 'Charlemagne'
strlen = len(name)
```

After this code executes, the `strlen` variable will be assigned the value 11, which is the number of characters in the string 'Charlemagne'.

Appending Strings

In Python, you do not use a function to append a string to another string. Instead, you use the `+` operator. When the `+` operator is used with two strings, it performs *string concatenation*. This means that it appends one string to another. For example, look at the following statement:

```
message = 'This is ' + 'one string.'
```

After this statement executes, the `message` variable will reference the string 'This is one string.'

The upper and lower Methods

The `upper` method returns a copy of a string with all of its letters converted to uppercase. Here is an example:

```
littleName = 'herman'
```



```
bigName = littleName.upper()
```

After this code executes, the `bigName` variable will reference the string "HERMAN".

The `lower` method returns a copy of a string with all of its letters converted to lowercase. Here is an example:

```
bigName = 'HERMAN'  
littleName = bigName.lower()
```

After this code executes, the `littleName` variable will reference the string "herman".

The `find` Method

In Python you can use the `find` method to perform a task similar to that of the `contains` function discussed in your textbook. The `find` method searches for a substring within a string. The method returns the index of the first occurrence of the substring, if it is found. (The index is the character position number within the string. The first character in a string is at index 0, the second character in a string is a index 1, and so forth.) If the substring is not found, the method returns -1. Here is an example:

```
string = 'Four score and seven years ago'  
position = string.find('seven')  
if position != -1:  
    print('The word "seven" was found at index', position)  
else:  
    print('The word "seven" was not found.')
```

This code will display:

The word "seven" was found at index 15

String Slicing

A *slice* is a span of items that are taken from a sequence. When you take a slice from a string, you get a span of characters from within the string. String slices are also called *substrings*.

To get a slice of a string, you write an expression in the following general format:

```
string[start : end]
```

In the general format, *start* is the index of the first character in the slice, and *end* is the index marking the end of the slice. The expression will return a string containing a copy of the

characters from *start* up to (but not including) *end*. For example, suppose we have the following:

```
full_name = 'Patty Lynn Smith'
middle_name = full_name[6:10]
```

The second statement assigns the string 'Lynn' to the `middle_name` variable. If you leave out the *start* index in a slicing expression, Python uses 0 as the starting index. Here is an example:

```
full_name = 'Patty Lynn Smith'
first_name = full_name[:5]
```

The second statement assigns the string 'Lynn' to `first_name`. If you leave out the *end* index in a slicing expression, Python uses the length of the string as the *end* index. Here is an example:

```
full_name = 'Patty Lynn Smith'
last_name = full_name[11:]
```

The second statement assigns the string 'Smith' to `first_name`. What do you think the following code will assign to the `my_string` variable?

```
full_name = 'Patty Lynn Smith'
my_string = full_name[:]
```

The second statement assigns the entire string 'Patty Lynn Smith' to `my_string`. The statement is equivalent to:

```
my_string = full_name[0 : len(full_name)]
```

The slicing examples we have seen so far get slices of consecutive characters from strings. Slicing expressions can also have step value, which can cause characters to be skipped in the string. Here is an example of code that uses a slicing expression with a step value:

```
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(letters[0:26:2])
```

The third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every 2nd character from the specified range in the string. The code will print the following:

```
ACEGIKMOQSUY
```

You can also use negative numbers as indexes in slicing expressions, to reference positions relative to the end of the string. Here is an example:

```
full_name = 'Patty Lynn Smith'
last_name = full_name[-5:]
```

Recall that Python adds a negative index to the length of a string to get the position referenced by that index. The second statement in this code assigns the string 'Smith' to the `last_name` variable.

Note: Invalid indexes do not cause an error. For example:

- If the *end* index specifies a position beyond the end of the string, Python will use the length of the string instead.
- If the *start* index specifies a position before the beginning of the string, Python will use 0 instead.
- If the *start* index is greater than the *end* index, the slicing expression will return an empty string.

Chapter 7 Input Validation

Chapter 7 in your textbook discusses the process of input validation in detail. There are no new language features introduced in the chapter, so here we will simply show you a Python version of the pseudocode Program 7-2. This program uses an input validation loop in lines 28 through 30 to validate that the value entered by the user is not negative.

Program 7-2 (retail.py)

```
1 # This program calculates retail prices.
2
3 # MARK_UP is used as a global constant for
4 # the markup up percentage.
5 MARK_UP = 2.5
6
7 # The main function
8 def main():
9     # Variable to control the loop.
10    another = 'y'
11
12    # Process one or more items.
13    while another == 'y' or another == 'Y':
14        # Display an item's retail price.
15        show_retail()
16
17        # Do this again?
18        another = input('Do you have another item? ' +
19                        '(Enter y for yes): ')
20
21 # The show_retail function gets an item's wholesale
22 # cost and displays the item's retail price.
23 def show_retail():
24     # Get the item's wholesale cost.
25     wholesale = float(input("Enter the item's wholesale cost: "))
26
27     # Validate the wholesale cost.
28     while wholesale < 0:
29         print('ERROR: the cost cannot be negative.')
30         wholesale = float(input('Enter the correct wholesale cost: '))
31
32     # Calculate the retail price.
33     retail = wholesale * MARK_UP
34
35     # Display the retail price.
36     print('The retail price is $', format(retail, '.2f'))
37
38 # Call the main function.
39 main()
```

This is the Python version of
Program 7-2 in your textbook.

Program Output

```
Enter the item's wholesale cost: -.50 [Enter]
ERROR: The cost cannot be negative.
```

Enter the correct wholesale cost: **0.50** [*Enter*]

The retail price is \$ 1.25.

Do you have another item? (Enter y for yes): **n** [*Enter*]

Chapter 8 Arrays (Lists)

In Python, you create lists instead of arrays. A *list* is similar to an array, but provides many more capabilities than a traditional array. A list is an object that contains multiple data items. Each item that is stored in a list is called an *element*. Here is a statement that creates a list of integers:

```
even_numbers = [2, 4, 6, 8, 10]
```

The items that are enclosed in brackets and separated by commas are the values of the list elements. The following is another example:

```
names = ['Molly', 'Steven', 'Will', 'Alicia', 'Adriana']
```

This statement creates a list of 5 strings.

You can use the `print` function to display an entire list, as shown here:

```
numbers = [5, 10, 15, 20]
print(numbers)
```

When the `print` function is called in the second statement, it will display the elements of the list like this:

```
[5, 10, 15, 20]
```

List Elements and Subscripts

You access each element of a list with a subscript. As discussed in your textbook, the first element's subscript is 0, the second element's subscript is 1, and so forth. The last element's subscript is the array size minus 1. Program 8-1 shows an example of a list being used to hold values entered by the user. This is the Python version of pseudocode Program 8-1 in your textbook.

Program 8-1 (*list_input.py*)

```
1 # Create a list to hold the number of hours
2 # worked by each employee.
3 hours = [0, 0, 0]
4
5 # Get the hours worked by employee 1.
6 hours[0] = int(input('Enter the hours worked by employee 1: '))
7
8 # Get the hours worked by employee 2.
9 hours[1] = int(input('Enter the hours worked by employee 2: '))
10
```

This is the Python version of
Program 8-1 in your textbook.

```
11 # Get the hours worked by employee 3.
12 hours[2] = int(input('Enter the hours worked by employee 3: '))
13
14 # Display the values entered.
15 print('The hours you entered are:')
16 print(hours[0])
17 print(hours[1])
18 print(hours[2])
```

Program Output

```
Enter the hours worked by employee 1: 40 [Enter]
Enter the hours worked by employee 2: 20 [Enter]
Enter the hours worked by employee 3: 15 [Enter]
The hours you entered are:
40
20
15
```

Using the len Function with Lists

An error will occur if you use an invalid index with a list. For example, look at the following code:

```
# This code will cause an error.
my_list = [10, 20, 30, 40]
index = 0
while index < 5:
    print(my_list[index])
    index += 1
```

The last time that this loop iterates, the `index` variable will be assigned the value 4, which is an invalid index for the list. As a result, the statement that calls the `print` function will cause an error.

The `len` function, that you learned about in Chapter 6 of this language companion, can be used with lists as well as strings. When you pass a list as an argument, the `len` function returns the number of elements in the list. The previously shown code, which causes an error, can be modified as follows to prevent the error:

```
my_list = [10, 20, 30, 40]
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

Iterating Over a List with the `for` Loop

You can easily iterate over the contents of a list with the `for` loop, as shown here:

```
numbers = [99, 100, 101, 102]
for n in numbers:
    print(n)
```

The `for` statement executes in the following manner: The variable `n` is assigned a copy of the first value in the list, and then the statements that appear in the block are executed. Then, the variable `n` is assigned a copy of the next value in the list, and the statements in the block are executed again. This continues until the variable has been assigned the last value in the list. If we run this code, it will print:

```
99
100
101
102
```

Keep in mind that as the `for` loop executes, the `n` variable is assigned a copy of the list elements, and any changes made to the `n` variable do not affect the list. To demonstrate, look at the following:

```
1 numbers = [99, 100, 101, 102]
2 for n in my_list:
3     n = 0
4
5 print(my_list)
```

The statement in line 3 merely reassigns the `n` variable to a different value (0). It does not change the list element that `n` referred to before the statement executed. When this code executes, the statement in line 5 will print:

```
99
100
101
102
```

Program 8-2 further demonstrates the use of loops with a list. This is the Python version of pseudocode Program 8-3 in your textbook.

Program 8-2 (list_loop.py)

```
1 # Create a list to hold the number of hours
2 # worked by each employee.
3 hours = [0, 0, 0]
4
5 # Create a variable to use in the loop, initialized at 0.
6 index = 0
```

This is the Python version of
Program 8-3 in your textbook.


```

7
8 # Get the hours for each employee.
9 while index < len(hours):
10     hours[index] = int(input('Enter the hours worked by employee ' +
11                             str(index + 1) + ': '))
12     index += 1
13
14 # Display the values entered.
15 for n in hours:
16     print(n)

```

Program Output

```

Enter the hours worked by employee 1: 40 [Enter]
Enter the hours worked by employee 2: 20 [Enter]
Enter the hours worked by employee 3: 15 [Enter]
40
20
15

```

Sequentially Searching an Array

Section 8.2 in your textbook discusses the sequential search algorithm, in which a program steps through each of an array's elements searching for a specific value. Program 8-3 shows an example of the sequential search algorithm. This is the Python version of pseudocode Program 8-6 in the textbook.

Program 8-3 (sequential_search.py)

```

1 # Create a list of test scores.
2 scores = [87, 75, 98, 100, 82,
3           72, 88, 92, 60, 78]
4
5 # Create a Boolean variable to act as a flag.
6 found = False
7
8 # Create a variable to use as a loop counter.
9 index = 0
10
11 # Step through the list searching for a
12 # score equal to 100.
13 while found == False and index < len(scores):
14     if scores[index] == 100:
15         found = True
16     else:
17         index = index + 1
18
19 # Display the search results.
20 if found:
21     print('You earned 100 on test number ' + str(index + 1))

```

This is the Python version of
Program 8-6 in your textbook.

```
22 else:
23     print('You did not earn 100 on any test.')
```

Program Output

You earned 100 on test number 4

Searching a String Array

Program 8-4 demonstrates how to find a string in a list of strings. This is the Python version of pseudocode Program 8-7 in the textbook.

Program 8-4 (*string_list_search.py*)

```
1 # Create a list of strings.
2 names = ['Ava Fischer', 'Chris Rich', 'Geri Rose',
3         'Matt Hoyle', 'Rose Harrison', 'Giovanni Ricci']
4
5 # Create a Boolean variable to act as a flag.
6 found = False
7
8 # Create a variable to use as a loop counter.
9 index = 0
10
11 # Get the string to search for.
12 searchValue = input('Enter a name to search for in the list: ')
13
14 # Step through the list searching for the
15 # specified name.
16 while found == False and index < len(names):
17     if names[index] == searchValue:
18         found = True
19     else:
20         index = index + 1
21
22 # Display the search results.
23 if found:
24     print('That name was found in element ' + str(index + 1))
25 else:
26     print('That name was not found in the list.')
```

This is the Python version of
Program 8-7 in your textbook.

Program Output

Enter a name to search for in the list: **Matt Hoyle** [Enter]
That name was found in element 3

Program Output

Enter a name to search for in the list: **Terry Thompson** [Enter]
That name was not found in the array.

Passing List as an Argument to a Function

When passing a list as an argument to a function in Python, it is not necessary to pass a separate argument indicating the list's size. This is because Python provides the `len` function that reports the list's size. The following code shows a function that has been written to accept a list as an argument:

```
def set_to_zero(numbers):
    index = 0
    while index < len(numbers):
        numbers[index] = 0
        index = index + 1
```

The function's parameter, `numbers`, is used to refer to a list. When you call this function and pass a list as an argument, the loop sets each element to 0. Here is an example of a code that calls the function:

```
my_list = [1, 2, 3, 4, 5]
set_to_zero(my_list)
print(my_list)
```

The last statement will print:

```
0
0
0
0
0
```

Program 8-5 gives a complete demonstration of passing an array to a method. This is the Python version of pseudocode Program 8-13 in your textbook.

Program 8-5 (*pass_list.py*)

```
1 def main():
2     # Create a list.
3     numbers = [2, 4, 6, 8, 10]
4
5     # Get the sum of the elements.
6     sum = get_total(numbers)
7
8     # Display the sum of the list elements.
9     print('The total is', sum)
10
11 # The get_total function accepts a list as an
12 # argument returns the total of the values in
13 # the list.
```

This is the Python version of
Program 8-13 in your textbook.

```
14 def get_total(value_list):
15     # Create a variable to use as an accumulator.
16     total = 0
17
18     # Calculate the total of the list elements.
19     for num in value_list:
20         total += num
21
22     # Return the total.
23     return total
24
25 # Call the main function.
26 main()
```

Program Output

The sum of the array elements is 30

Two-Dimensional Arrays

In Python you can create a list of lists, which acts much like a two-dimensional array. Here's an example:

```
numbers = [ [1, 2, 3], [10, 20, 30] ]
```

This creates a list named `numbers`, with two elements. The first element is the following list:

```
[1, 2, 3]
```

The second element is the following list:

```
[10, 20, 30]
```

The following statement prints the contents of `numbers[0]`, which is the first element:

```
print(numbers[0])
```

If we execute this statement, the following be displayed:

```
[1, 2, 3]
```

The following statement prints the contents of `numbers[1]`, which is the second element:

```
print(numbers[1])
```

If we execute this statement, the following be displayed:

```
[10, 20, 30]
```

Rows and Columns

As discussed in your textbook, we normally think of two-dimensional arrays as having rows and columns. We can use this metaphor with two-dimensional lists as well. For example, let's say the following two-dimensional list contains sets of test scores:

```
scores = [ [70, 80, 90],  
           [80, 60, 75],  
           [85, 75, 95] ]
```

By declaring the list this way (with each list that is an element shown on a separate line) it's easy to see how we can think of the list as a set of rows and columns.

When processing the data in a two-dimensional list, we use two subscripts: one for the row and another for the column. In the `scores` list, the elements in row 0 are referenced as follows:

```
scores[0][0]  
scores[0][1]  
scores[0][2]
```

The elements in row 1 are as follows:

```
scores[1][0]  
scores[1][1]  
scores[1][2]
```

And the elements in row 2 are as follows:

```
scores[2][0]  
scores[2][1]  
scores[2][2]
```

To access one of the elements in a two-dimensional list, you use both subscripts. For example, the following statement prints the number in `scores[0][2]`:

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

And the following statement assigns the number 95 to `scores[2][1]`:

```
scores[2][1] = 95
```

Programs that process two-dimensional lists can do so with nested loops. For example, the following code displays all of the elements in the `scores` list:

```
NUM_ROWS = 3
NUM_COLS = 3

row = 0
while row < NUM_ROWS:
    col = 0
    while col < NUM_COLS:
        print(scores[row][col])
        col = col + 1
    row = row + 1
```

And the following code prompts the user to enter a score, once for each element in the list:

```
NUM_ROWS = 3
NUM_COLS = 3

row = 0
while row < NUM_ROWS:
    col = 0
    while col < NUM_COLS:
        scores[row][col] = int(input('Enter a score: '))
        col = col + 1
    row = row + 1
```

Program 8-6 shows a complete example. It creates a list with three rows and four columns, prompts the user for values to store in each element, and then displays the values in each element. This is the Python example of pseudocode Program 8-16 in your textbook.

Program 8-5 (*twoD_list.py*)

```
1 ROWS = 3
2 COLS = 4
3
4 # Create a 2D list.
5 values = [ [0, 0, 0, 0],
6            [0, 0, 0, 0],
7            [0, 0, 0, 0] ]
8
9 # Get values to store in the list.
10 row = 0
11 while row < ROWS:
12     col = 0
13     while col < COLS:
14         values[row][col] = int(input('Enter a number: '))
15         col = col + 1
```

This is the Python version of
Program 8-16 in your textbook.

```
16         row = row + 1
17
18 # Display the vlues in the list.
19 print('Here are the values you entered.')
20 row = 0
21 while row < ROWS:
22     col = 0
23     while col < COLS:
24         print(values[row][col])
25         col = col + 1
26     row = row + 1
```

Program Output

```
Enter a number: 1 [Enter]
Enter a number: 2 [Enter]
Enter a number: 3 [Enter]
Enter a number: 4 [Enter]
Enter a number: 5 [Enter]
Enter a number: 6 [Enter]
Enter a number: 7 [Enter]
Enter a number: 8 [Enter]
Enter a number: 9 [Enter]
Enter a number: 10 [Enter]
Enter a number: 11 [Enter]
Enter a number: 12 [Enter]
Here are the values you entered.
1
2
3
4
5
6
7
8
9
10
11
12
```

Chapter 9 Sorting and Searching

Chapter 9 in your textbook discusses the following sorting algorithms:

- Bubble Sort
- Selection Sort
- Insertion Sort

The Binary Search algorithm is also discussed. The textbook chapter examines these algorithms in detail, and no new language features are introduced. For these reasons we will simply present the Python code for the algorithms in this chapter. For more in-depth coverage of the logic involved, consult the textbook.

Bubble Sort

Program 9-1 is only a partial program. It shows the Python version of pseudocode Program 9-1, which is the Bubble Sort algorithm.

Program 9-1 (*bubble_sort.py*)

```
1 # Note: This is not a complete program.
2 #
3 # The bubble_sort function uses the bubble sort algorithm
4 # to sort a list of integers.
5 # Note the following:
6 # (1) We do not have to pass the array size because we
7 #     can use the len function.
8 # (2) We do not have a separate method to swap values.
9 #     The swap is performed inside this method.
10
11 def bubble_sort(arr):
12     # Set max_element to the length of the arr list, minus
13     # one. This is necessary for the outer loop.
14     max_element = len(arr) - 1
15
16     # The outer loop positions max_element at the last element
17     # to compare during each pass through the list. Initially
18     # max_element is the index of the last element in the array.
19     # During each iteration, it is decreased by one.
20     while max_element >= 0:
21         # Set index to 0, necessary for the inner loop.
22         index = 0
23
24         # The inner loop steps through the list, comparing
25         # each element with its neighbor. All of the elements
26         # from index 0 through max_element are involved in the
27         # comparison. If two elements are out of order, they
28         # are swapped.
29         while index <= max_element - 1:
30             # Compare an element with its neighbor.
31             if arr[index] > arr[index + 1]:
```

This is the Python version of
Program 9-1 in your textbook.


```

32         # Swap the two elements.
33         temp = arr[index]
34         arr[index] = arr[index + 1]
35         arr[index + 1] = temp
36     # Increment index.
37     index = index + 1
38     # Decrement max_element.
39     max_element = max_element - 1

```

Selection Sort

Program 9-2 is also a partial program. It shows the Python version of the selectionSort pseudocode module that is shown in Program 9-5 in your textbook.

This is the Python version of the selectionSort Module shown in **Program 9-5** in your textbook.

Program 9-2 (*selection_sort.py*)

```

1  # Note: This is not a complete program.
2  #
3  # The selection_sort function performs the selection sort
4  # algorithm on a list of integers.
5
6  def selection_sort(arr):
7      # Set start_scan to 0. This is necessary for
8      # the outer loop. It is the starting position
9      # of the scan.
10     start_scan = 0
11
12     # The outer loop iterates once for each element in the
13     # list. The start_scan variable marks the position where
14     # the scan should begin.
15     while start_scan < len(arr) - 1:
16         # Assume the first element in the scannable area
17         # is the smallest value.
18         min_index = start_scan
19         min_value = arr[start_scan]
20
21         # Initialize index for the inner loop.
22         index = start_scan + 1
23
24         # Scan the list, starting at the 2nd element in
25         # the scannable area. We are looking for the smallest
26         # value in the scannable area.
27         while index < len(arr):
28             if arr[index] < min_value:
29                 min_value = arr[index]
30                 min_index = index
31             # Increment index.
32             index = index + 1

```

```

33
34     # Swap the element with the smallest value
35     # with the first element in the scannable area.
36     arr[min_index] = arr[start_scan]
37     arr[start_scan] = min_value
38
39     # Increment start_scan.
40     start_scan = start_scan + 1

```

Insertion Sort

Program 9-3 is also a partial program. It shows the Python version of the insertionSort pseudocode module that is shown in Program 9-6 in your textbook.

Program 9-3 (*insertion_sort.py*)

```

1  # Note: this is not a complete program.
2  #
3  # The insertion_sort function performs an insertion sort
4  # algorithm on a list of integers.
5
6  def insertion_sort(arr):
7      # Set index to 1 for the outer loop.
8      index = 1
9
10     # The outer loop steps the index variable through
11     # each subscript in the list, starting at 1. This
12     # is because element 0 is considered already sorted.
13     while index < len(arr):
14         # The first element outside the sorted subset is
15         # arr[index]. Assign the value of this element
16         # to unsorted_value.
17         unsorted_value = arr[index]
18
19         # Start the scan variable at the subscript of the
20         # first element outside the sorted subset.
21         scan = index
22
23         # Move the first element outside the sorted subset
24         # into its proper position within the sorted subset.
25         while scan > 0 and arr[scan - 1] > unsorted_value:
26             arr[scan] = arr[scan - 1]
27             scan = scan - 1
28
29     # Insert the unsorted value in its proper position

```

This is the Python version of the insertionSort Module shown in **Program 9-6** in your textbook.

```

30         # within the sorted subset.
31         arr[scan] = unsorted_value
32
33         # Increment index.
34         index = index + 1

```

Binary Search

Program 9-4 is also a partial program. It shows the Python version of the `binarySearch` pseudocode module that is shown in Program 9-7 in your textbook.

Program 9-4 (*binary_search.py*)

```

1  # Note: This is not a complete program.
2  #
3  # The binary_search function performs a binary search on a
4  # String list. The list is searched for the string passed to
5  # the value parameter. If the string is found, its subscript
6  # is returned. Otherwise, -1 is returned indicating the value
7  # was not found in the list.
8
9  def binary_search(arr, value):
10     # Set the initial values.
11     first = 0
12     last = len(arr) - 1
13     position = -1
14     found = False
15
16     # Search for the value
17     while not found and first <= last:
18         # Calculate the mid point.
19         middle = (first + last) / 2
20
21         # If the value is found at the mid point...
22         if arr[middle] == value:
23             found = True
24             position = middle
25         # else if value is in the lower half...
26         elif arr[middle] > value:
27             last = middle - 1
28         # else if value is in the upper half...
29         else:
30             first = middle + 1
31
32     # Return the position of the item, or -1

```

This is the Python version of the `binarySearch` Module shown in **Program 9-7** in your textbook.

```
33     # if it was not found.  
34     return position
```

Chapter 10 Files

Opening a File

You use the `open` function in Python to open a file. The `open` function creates a file object and associates it with a file on the disk. Here is the general format of how the `open` function is used:

```
file_variable = open(filename, mode)
```

In the general format:

- *file_variable* is the name of the variable that will reference the file object.
- *filename* is a string specifying the name of the file.
- *mode* is a string specifying the mode (reading, writing, etc.) in which the file will be opened. Table 10-1 shows three of the strings that you can use to specify a mode. (There are other, more complex modes. The modes shown in Table 10-1 are the ones we will use in this book.)

Table 10-1 Some of the Python file modes

Mode	Description
'r'	Open a file for reading only. The file cannot be changed or written to.
'w'	Open a file for writing. If the file already exists, erase its contents. If it does not exist, create it.
'a'	Open a file to be written to. All data written to the file will be appended to its end. If the file does not exist, create it.

For example, suppose the file `customers.txt` contains customer data, and we want to open for reading. Here is an example of how we would call the `open` function:

```
customer_file = open('cusomters.txt', 'r')
```

After this statement executes, the file named `customers.txt` will be opened, and the variable `customer_file` will reference a file object that we can use to read data from the file.

Suppose we want to create a file named `sales.txt` and write data to it. Here is an example of how we would call the `open` function:

```
sales_file = open('sales.txt', 'w')
```

After this statement executes, the file named `sales.txt` will be created, and the variable `sales_file` will reference a file object that we can use to write data to the file.

Warning: Remember, when you use the 'w' mode you are creating the file on the disk. If a file with the specified name already exists when the file is opened, the contents of the existing file will be erased.
--

Writing Data to a File

Once you have opened a file for writing, you use the file object's `write` method to write data to a file. Here is the general format of how you call the write method:

```
file_variable.write(string)
```

In the format, *file_variable* is a variable that references a file object, and *string* is a string that will be written to the file. The file must be opened for writing (using the 'w' or 'a' mode) or an error will occur.

Let's assume that `customer_file` references a file object, and the file was opened for writing with the 'w' mode. Here is an example of how we would write the string 'Charles Pace' to the file:

```
customer_file.write('Charles Pace')
```

The following code shows another example:

```
name = 'Charles Pace'  
customer_file.write(name)
```

The second statement writes the value referenced by the `name` variable to the file associated with `customer_file`. In this case, it would write the string 'Charles Pace' to the file. (These examples show a string being written to a file, but you can also write numeric values.)

Closing a File

Once a program is finished working with a file, it should close the file. Closing a file disconnects the program from the file. In Python you use the file object's `close` method to close a file. For example, the following statement closes the file that is associated with `customer_file`:

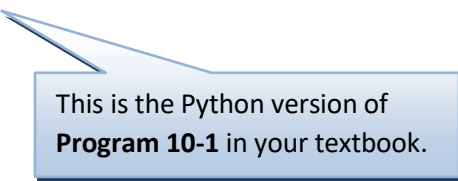
```
customer_file.close()
```

Once a file is closed, the connection between it and the file object is removed. In order to perform further operations on the file, it must be opened again.

Program 10-1 shows a complete Python program that that opens an output file, writes data to it, and then closes it. This is the Python version of pseudocode Program 10-1 in your textbook.

Program 10-1 (file_write_demo.py)

```
1  # This program writes three lines of data
2  # to a file.
3  def main():
4      # Open a file named philosophers.txt.
5      outfile = open('philosophers.txt', 'w')
6
7      # Write the names of three philosophers
8      # to the file.
9      outfile.write('John Locke\n')
10     outfile.write('David Hume\n')
11     outfile.write('Edmund Burke\n')
12
13     # Close the file.
14     outfile.close()
15
16 # Call the main function.
17 main()
```



This is the Python version of
Program 10-1 in your textbook.

When this program executes, line 5 creates a file named `philosophers.dat` on the disk, and lines 9 through 11 write the strings `'John Locke\n'`, `'David Hume\n'`, and `'Edmund Burke\n'` to the file. Line 14 closes the file.

Writing Newlines at the End of Each Line

Notice the use of the `\n` that appears inside the strings that are written to the file in lines 9, 10, and 11. The `\n` sequence is known as an escape character. An *escape character* is a special character that is preceded with a backslash (`\`), appearing inside a string literal. When a string literal that contains escape characters is printed on the screen or written to a file, the escape characters are treated as special commands that are embedded in the string.

The `\n` sequence is the *newline escape character*. When the `\n` escape character is printed by the `print` function, it isn't displayed on the screen. Instead, it causes output to advance to the next line. For example, look at the following statement:

```
print('One\nTwo\nThree')
```

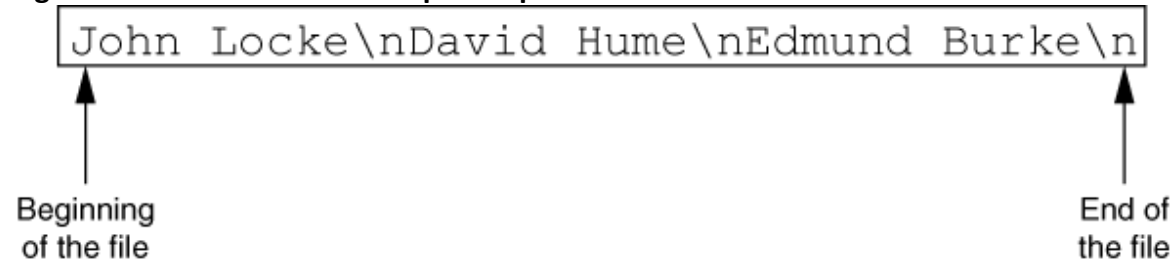
When this statement executes it displays:

```
One
Two
Three
```

So why did we include a `\n` at the end of each item that is written to the file in Program 10-1? Let's take a closer look. The statements in lines 9 through 11 write three strings to the file. Line

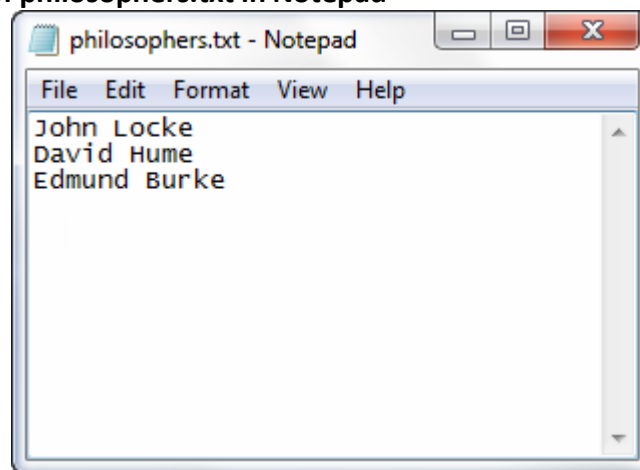
9 writes the string 'John Locke\n', line 10 writes the string 'David Hume\n', and line 11 writes the string 'Edmund Burke\n'. Line 14 closes the file. After this program runs, the three items shown in Figure 10-1 will be written to the philosophers.txt file.

Figure 10-1 Contents of the file philosophers.txt



Notice that each of the strings written to the file end with `\n`. The `\n` marks the location where a new line begins in the file. We can see how this works if we open the file in a text editor. For example, Figure 10-2 shows the philosophers.txt file as it appears in Notepad.

Figure 10-2 Contents of philosophers.txt in Notepad



Opening a File and Reading Data from It

If a file has been opened for reading (using the 'r' mode) you can use the file object's `readline` method to read a line from the file. The method returns the line as a string, including the `\n`. Program 10-2 shows how we can use the `readline` method to read the contents of the philosophers.txt file, one line at a time. (This is the Python version of pseudocode Program 10-2 in your textbook.)

Program 10-2 (*file_read_demo.py*)

```
1 # This program reads the contents of the
2 # philosophers.txt file one line at a time.
3 def main():
4     # Open a file named philosophers.txt.
5     infile = open('philosophers.txt', 'r')
```

This is the Python version
of **Program 10-2** in your
textbook.


```

6
7     # Read three lines from the file
8     line1 = infile.readline()
9     line2 = infile.readline()
10    line3 = infile.readline()
11
12    # Close the file.
13    infile.close()
14
15    # Print the names that were read.
16    print('Here are the names of three philosophers:')
17    print(line1)
18    print(line2)
19    print(line3)
20
21 # Call the main function.
22 main()

```

Program Output

```

Here are the names of three philosophers:
John Locke

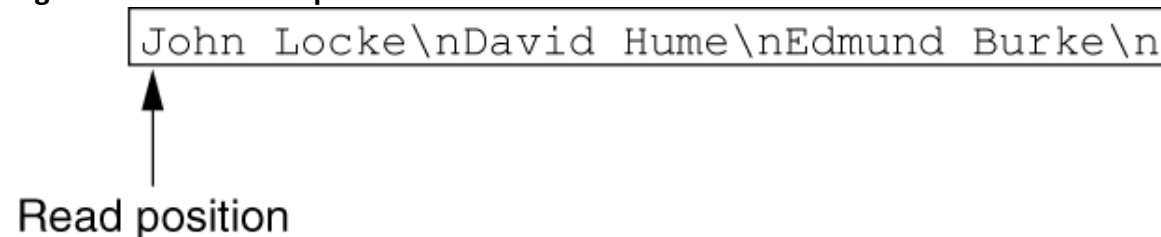
David Hume

Edmund Burke

```

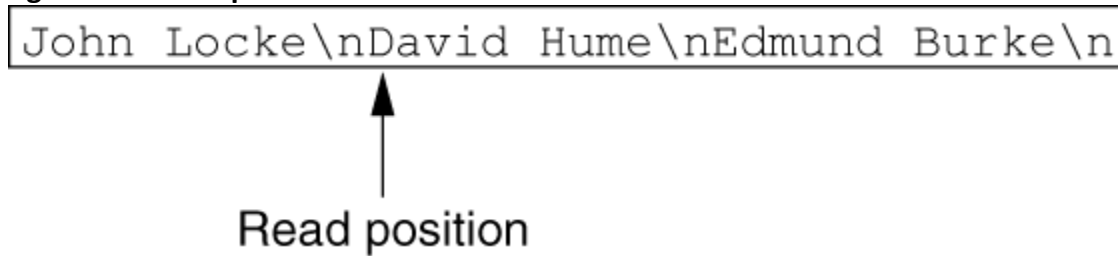
The statement in line 5 opens the philosophers.txt file for reading, using the 'r' mode. It also creates a file object and assigns the object to the infile variable. When a file is opened for reading, a special value known as a *read position* is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. Initially, the read position is set to the beginning of the file. After the statement in line 5 executes, the read position for the philosophers.txt file will be positioned as shown in Figure 10-3.

Figure 10-3 Initial read position



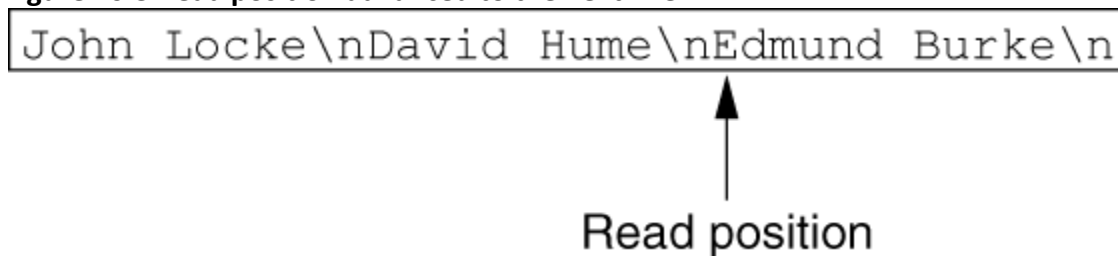
The statement in line 8 calls the `infile.readline` method to read the first line from the file. The line, which is returned as a string, is assigned to the `line1` variable. After this statement executes the `line1` variable will be assigned the string 'John Locke\n'. In addition, the file's read position will be advanced to the next line in the file, as shown in Figure 10-4.

Figure 10-4 Read position advanced to the next line



Then the statement in line 9 reads the next line from the file and assigns it to the `line2` variable. After this statement executes the `line2` variable will be assigned the string `'David Hume\n'`. The file's read position will be advanced to the next line in the file, as shown in Figure 10-5.

Figure 10-5 Read position advanced to the next line



Then the statement in line 10 reads the next line from the file and assigns it to the `line3` variable. After this statement executes the `line3` variable will be assigned the string `'Edmund Burke\n'`. After this statement executes, the read position will be advanced to the end of the file, as shown in Figure 10-6.

10-7 shows the `line1`, `line2`, and `line3` variables and the strings they are assigned after these statements have executed.

Figure 10-6 Read position advanced to the end of the file

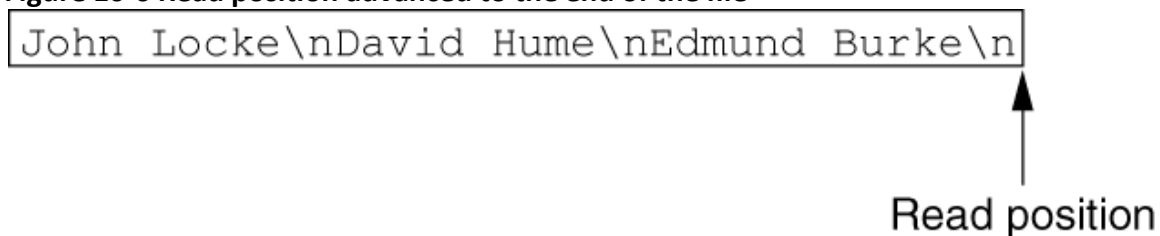
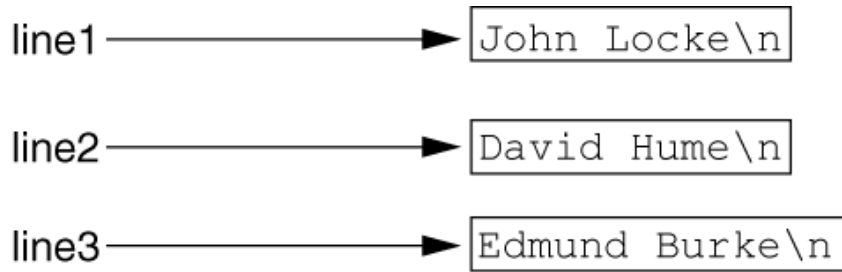


Figure 10-7 The strings referenced by the `line1`, `line2`, and `line3` variables



The statement in line 13 closes the file. The statements in lines 17 through 19 display the contents of the `line1`, `line2`, and `line3` variables.

Note: If the last line in a file is not terminated with a `\n`, the `readline` method will return the line without a `\n`.

Reading a String and Stripping the Newline from It

Sometimes complications are caused by the `\n` that appears at the end of the strings that are returned from the `readline` method. For example, did you notice in the sample output of Program 10-2 that a blank line is printed after each line of output? This is because each of the strings that are printed in lines 17 through 19 end with a `\n` escape sequence. When the strings are printed, the `\n` causes an extra blank line to appear.

The `\n` serves a necessary purpose inside a file: it separates the items that are stored in the file. However, in most cases you want to remove the `\n` from a string after it is read from a file. Each string in Python has a method named `rstrip` that removes, or "strips", specific characters from the end of a string. The following code shows an example of how the `rstrip` method can be used.

```
name = 'Joanne Manchester\n'
name = name.rstrip('\n')
```

The first statement assigns the string `'Joanne Manchester\n'` to the `name` variable. (Notice that the string ends with the `\n` escape sequence.) The second statement calls the `name.rstrip('\n')` method. The method returns a copy of the `name` string without the trailing `\n`. This string is assigned back to the `name` variable. The result is that the trailing `\n` is stripped away from the `name` string.

Program 10-3 is another program that reads and displays the contents of the `philosophers.txt` file. This program uses the `rstrip` method to strip the `\n` from the strings that are read from the file before they are displayed on the screen. As a result, the extra blank lines do not appear in the output.

Program 10-3 (*strip_newline.py*)

```
1  # This program reads the contents of the
2  # philosophers.txt file one line at a time.
3  def main():
4      # Open a file named philosophers.txt.
5      infile = open('philosophers.txt', 'r')
6
7      # Read three lines from the file
8      line1 = infile.readline()
9      line2 = infile.readline()
10     line3 = infile.readline()
11
12     # Strip the \n from each string.
13     line1 = line1.rstrip('\n')
14     line2 = line2.rstrip('\n')
15     line3 = line3.rstrip('\n')
16
17     # Close the file.
18     infile.close()
19
20     # Print the names that were read.
21     print('Here are the names of three philosophers:')
22     print(line1)
23     print(line2)
24     print(line3)
25
26     # Call the main function.
27     main()
```

Program Output

```
Here are the names of three philosophers:
John Locke
David Hume
Edmund Burke
```

Concatenating a Newline to a String

Program 10-1 wrote three string literals to a file, and each string literal ended with a `\n` escape sequence. In most cases, the data items that are written to a file are not string literals, but values in memory that are referenced by variables. This would be the case in a program that prompts the user to enter data, and then writes that data to a file.

When a program writes data that has been entered by the user to a file, it is usually necessary to concatenate a `\n` escape sequence to the data before writing it. This ensures that each piece of data is written to a separate line in the file. Program 10-4 demonstrates how this is done.

Program 10-4 (write_names.py)

```
1  # This program gets three names from the user
2  # and writes them to a file.
3
4  def main():
5      # Get three names.
6      print('Enter the names of three friends.')
7      name1 = input('Friend #1: ')
8      name2 = input('Friend #2: ')
9      name3 = input('Friend #3: ')
10
11     # Open a file named friends.txt.
12     myfile = open('friends.txt', 'w')
13
14     # Write the names to the file.
15     myfile.write(name1 + '\n')
16     myfile.write(name2 + '\n')
17     myfile.write(name3 + '\n')
18
19     # Close the file.
20     myfile.close()
21     print('The names were written to friends.txt.')
22
23 # Call the main function.
24 main()
```

Program Output (with Input Shown in Bold)

```
Enter the names of three friends.
Friend #1: Joe [Enter]
Friend #2: Rose [Enter]
Friend #3: Geri [Enter]
The names were written to friends.txt.
```

Lines 7 through 9 prompt the user to enter three names, and those names are assigned to the variables `name1`, `name2`, and `name3`. Line 12 opens a file named `friends.txt` for writing. Then, lines 15 through 17 write the names entered by the user, each with `\n` concatenated to it. As a result, each name will have the `\n` escape sequence added to it when written to the file. Figure 10-8 shows the contents of the file with the names entered by the user in the sample run.

Figure 10-8 The friends.txt file

```
Joe\nRose\nGeri\n
```

Appending Data to an Existing File

When you use the 'w' mode to open an output file and a file with the specified filename already exists on the disk, the existing file will be erased and a new empty file with the same name will be created. Sometimes you want to preserve an existing file and append new data to its current contents. Appending data to a file means writing new data to the end of the data that already exists in the file.

In Python you can use the 'a' mode to open an output file in *append mode*, which means the following.

- If the file already exists, it will not be erased. If the file does not exist, it will be created.
- When data is written to the file, it will be written at the end of the file's current contents.

For example, assume the file friends.txt contains the following names, each in a separate line:

```
Joe  
Rose  
Geri
```

The following code opens the file and appends additional data to its existing contents.

```
myfile = open('friends.txt', 'a')  
myfile.write('Matt\n')  
myfile.write('Chris\n')  
myfile.write('Suze\n')  
myfile.close()
```

After this program runs, the file friends.txt will contain the following data:

```
Joe  
Rose  
Geri  
Matt  
Chris  
Suze
```

Writing and Reading Numeric Data

Strings can be written directly to a file with the write method, but numbers must be converted to strings before they can be written. Python has a built-in function named `str` that converts a value to a string. For example, assuming the variable `num` is assigned the value 99, the expression `str(num)` will return the string '99'.

Program 10-5 shows an example of how you can use the `str` function to convert a number to a string, and write the resulting string to a file.

Program 10-5 (*write_numbers.py*)

```
1  # This program demonstrates how numbers
2  # must be converted to strings before they
3  # are written to a text file.
4
5  def main():
6      # Open a file for writing.
7      outfile = open('numbers.txt', 'w')
8
9      # Get three numbers from the user.
10     num1 = int(input('Enter a number: '))
11     num2 = int(input('Enter another number: '))
12     num3 = int(input('Enter another number: '))
13
14     # Write the numbers to the file.
15     outfile.write(str(num1) + '\n')
16     outfile.write(str(num2) + '\n')
17     outfile.write(str(num3) + '\n')
18
19     # Close the file.
20     outfile.close()
21     print('Data written to numbers.txt')
22
23 # Call the main function.
24 main()
```

Program Output (with Input Shown in Bold)

```
Enter a number: 22 [Enter]
Enter another number: 14 [Enter]
Enter another number: -99 [Enter]
Data written to numbers.txt
```

The statement in line 7 opens the file `numbers.txt` for writing. Then the statements in lines 10 through 12 prompt the user to enter three numbers, which are assigned to the variables `num1`, `num2`, and `num3`.

Take a closer look at the statement in line 15, which writes the value referenced by `num1` to the file:

```
outfile.write(str(num1) + '\n')
```

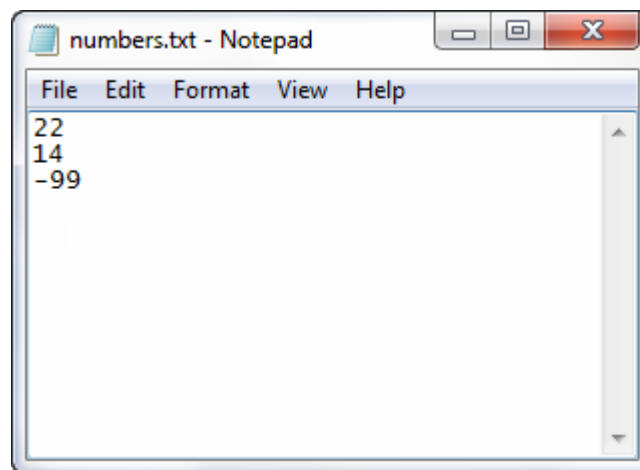
The expression `str(num1) + '\n'` converts the value referenced by `num1` to a string and concatenates the `\n` escape sequence to the string. In the program's sample run, the user entered 22 as the first number, so this expression produces the string `'22\n'`. As a result, the string `'22\n'` is written to the file.

Lines 16 and 17 perform the similar operations, writing the values referenced by `num2` and `num3` to the file. After these statements execute, the values shown in Figure 10-9 will be written to the file. Figure 10-10 shows the file viewed in Notepad.

Figure 10-9 Contents of the numbers.txt file

```
22\n14\n-99\n
```

Figure 10-10 The numbers.txt file viewed in Notepad



When you read numbers from a text file, they are always read as strings. For example, suppose a program uses the following code to read the first line from the `numbers.txt` file that was created by Program 10-5:

```
1  infile = open('numbers.txt', 'r')
2  value = infile.readline()
3  infile.close()
```

The statement in line 2 uses the `readline` method to read a line from the file. After this statement executes, the `value` variable will reference the string `'22\n'`. This can cause a problem if we intend to perform math with the `value` variable, because you cannot perform math on strings. In such a case you must convert the string to a numeric type.

Python provides the built-in function `int` to convert a string to an integer, and the built-in function `float` to convert a string to a floating-point number. For example, we could modify the code previously shown as follows:


```
1  infile = open('numbers.txt', 'r')
2  string_input = infile.readline()
3  value = int(string_input)
4  infile.close()
```

The statement in line 2 reads a line from the file and assigns it to the `string_input` variable. As a result, `string_input` will reference the string `'22\n'`. Then the statement in line 3 uses the `int` function to convert `string_input` to an integer, and assigns the result to `value`. After this statement executes, the `value` variable will reference the integer 22. (Both the `int` and `float` functions ignore the `\n` that appears at the end of the string that is passed as an argument.)

This code demonstrates the steps involved in reading a string from a file with the `readline` method, and then converting that string to an integer with the `int` function. In many situations, however, the code can be simplified. A better way is to read the string from the file and convert it in one statement, as shown here:

```
1  infile = open('numbers.txt', 'r')
2  value = int(infile.readline())
3  infile.close()
```

Notice in line 2 that a call to the `readline` method is used as the argument to the `int` function. Here's how the code works: the `readline` method is called, and it returns a string. That string is passed to the `int` function, which converts it to an integer. The result is assigned to the `value` variable.

Using Loops to Process Files

Program 10-6 demonstrates how a loop can be used to collect items of data to be stored in a file. This is the Python version of pseudocode Program 10-3 in your textbook. Figure 10-11 shows the contents of the `sales.txt` file containing the data entered by the user in the sample run.

Program 10-6 (*write_sales.py*)

```
1  # This program prompts the user for sales amounts
2  # and writes those amounts to the sales.txt file.
3
4  def main():
5      # Get the number of days.
6      num_days = int(input('For how many days do ' +
7                          'you have sales? '))
8
9      # Open a new file named sales.txt.
```

```

10     sales_file = open('sales.txt', 'w')
11
12     # Get the amount of sales for each day and write
13     # it to the file.
14     for count in range(1, num_days + 1):
15         # Get the sales for a day.
16         sales = float(input('Enter the sales for day #' +
17                             str(count) + ': '))
18
19         # Write the sales amount to the file.
20         sales_file.write(str(sales) + '\n')
21
22     # Close the file.
23     sales_file.close()
24     print('Data written to sales.txt.')
25
26 # Call the main function.
27 main()

```

This is the Python version of
Program 10-3 in your textbook.

Program Output (with Input Shown in Bold)

```

For how many days do you have sales? 5 [Enter]
Enter the sales for day #1: 1000.0 [Enter]
Enter the sales for day #2: 2000.0 [Enter]
Enter the sales for day #3: 3000.0 [Enter]
Enter the sales for day #4: 4000.0 [Enter]
Enter the sales for day #5: 5000.0 [Enter]
Data written to sales.txt.

```

Figure 10-11 Contents of the sales.txt file

```
1000.0\n2000.0\n3000.0\n4000.0\n5000.0\n
```

Detecting the End of a File

In Python, the `readline` method returns an empty string (") when it has attempted to read beyond the end of a file. This makes it possible to write a while loop that determines when the end of a file has been reached. Program 10-7 demonstrates how this can be done in code. The program reads and displays all of the values in the `sales.txt` file. (This is the Python version of pseudocode Program 10-3 in your textbook.)

Program 10-7 (*read_sales.py*)

```
1  # This program reads all of the values in
2  # the sales.txt file.
3
4  def main():
5      # Open the sales.txt file for reading.
6      sales_file = open('sales.txt', 'r')
7
8      # Read the first line from the file, but
9      # don't convert to a number yet. We still
10     # need to test for an empty string.
11     line = sales_file.readline()
12
13     print('Here are the sales amounts:')
14
15     # As long as an empty string is not returned
16     # from readline, continue processing.
17     while line != '':
18         # Convert line to a float.
19         amount = float(line)
20
21         # Format and display the amount.
22         print('$', format(amount, '.2f'))
23
24         # Read the next line.
25         line = sales_file.readline()
26
27     # Close the file.
28     sales_file.close()
29
30 # Call the main function.
31 main()
```

This is the Python version of
Program 10-3 in your textbook.

Program Output (with Input Shown in Bold)

```
Here are the sales amounts:
$ 1000.00
$ 2000.00
$ 3000.00
```

```
$ 4000.00
$ 5000.00
```

Using Python's `for` Loop to Read Lines

In the previous example you saw how the `readline` method returns an empty string when the end of the file has been reached. The Python language also allows you to write a `for` loop that automatically reads line in a file without testing for any special condition that signals the end of the file. The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached. When you simply want to read the lines in a file, one after the other, this technique is simpler and more elegant than writing a `while` loop that explicitly tests for an end of the file condition. Here is the general format of the loop:

```
for variable in file_object:
    statement
    statement
    etc.
```

In the general format, *variable* is the name of a variable and *file_object* is a variable that references a file object. The loop will iterate once for each line in the file. The first time the loop iterates, *variable* will reference the first line in the file (as a string), the second time the loop iterates, *variable* will reference the second line, and so forth. Program 10-8 provides a demonstration. It reads and displays all of the items in the `sales.txt` file.

Program 10-8 (`read_sales2.py`)

```
1  # This program uses the for loop to read
2  # all of the values in the sales.txt file.
3
4  def main():
5      # Open the sales.txt file for reading.
6      sales_file = open('sales.txt', 'r')
7
8      # Read all the lines from the file.
9      for line in sales_file:
10         # Convert line to a float.
11         amount = float(line)
12         # Format and display the amount.
13         print('$', format(amount, '.2f'))
14
15     # Close the file.
16     sales_file.close()
17
18 # Call the main function.
19 main()
```

Program Output

\$ 1000.00

\$ 2000.00

\$ 3000.00

\$ 4000.00

\$ 5000.00

Chapter 11 Menu-Driven Programs

Chapter 11 in your textbook discusses menu-driven programs. A menu-driven program presents a list of operations that the user may select from (the menu), and then performs the operation that the user selected. There are no new language features introduced in the chapter, so here we will simply show you a Python program that is menu-driven. Program 11-1 is the Python version of the pseudocode Program 11-3.

Program 11-1 (*menu_driven.py*)

```
1 # Display the menu.
2 print('1. Convert inches to centimeters.')
3 print('2. Convert feet to meters.')
4 print('3. Convert miles to kilometers.')
5 print()
6
7 # Prompt the user for a selection.
8 menu_selection = int(input('Enter your selection: '))
9
10 # Validate the menu selection.
11 while menu_selection < 1 or menu_selection > 3:
12     print('That is an invalid selection.')
13     menu_selection = int(input('Enter 1, 2, or 3: '))
14
15 # Perform the selected operation.
16 if menu_selection == 1:
17     # Convert inches to centimeters.
18     inches = float(input('Enter the number of inches: '))
19     centimeters = inches * 2.54
20     print('That is equal to', centimeters, 'centimeters.')
21 elif menu_selection == 2:
22     # Convert feet to meters.
23     feet = float(input('Enter the number of feet: '))
24     meters = feet * 0.3048
25     print('That is equal to', meters, 'meters.')
26 elif menu_selection == 3:
27     # Convert miles to kilometers.
28     miles = float(input('Enter the number of miles: '))
29     kilometers = miles * 1.609
30     print('That is equal to', kilometers, 'kilometers.')
```

Program Output

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.
```

This is the Python version of
Program 11-3 in your textbook.

```
Enter your selection: 1 [Enter]
```

Enter the number of inches: **10 [Enter]**
That is equal to 25.4 centimeters.

Program Output

1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: **2 [Enter]**
Enter the number of feet: **10 [Enter]**
That is equal to 3.048 meters.

Program Output

1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: **4 [Enter]**
That is an invalid selection.
Enter 1, 2, or 3: **3 [Enter]**
Enter the number of miles: **10 [Enter]**
That is equal to 16.09 kilometers.

Chapter 12 Text Processing

Chapter 12 in your textbook discusses programming techniques for working with the individual characters in a string. Python allows you to retrieve the individual characters in a string using subscript notation, as described in the book. For example, the following code creates the string 'Hello', and then uses subscript notation to print the first character in the string:

```
greeting = 'Hello'
print(greeting[0])
```

Although you can use subscript notation to retrieve the individual characters in a string, you cannot use it to change the value of a character within a string. This is because strings in Python are immutable, which means that once they are created, they cannot be changed.

Because Python strings are immutable, you cannot use an expression in the form *string[index]* on the left side of an assignment operator. For example, the following code will cause an error:

```
# Assign 'Bill' to friend.
friend = 'Bill'
# Can we change the first character to 'J'?
friend[0] = 'J'      # No, this will cause an error!
```

The last statement in this code will cause an error because it attempts to change the value of the first character in the string 'Bill'.

In this chapter we will show you how to do the following in Python:

- Use subscript notation to access the individual characters in a string.
- Use string testing methods.

Because of string immutability, we will not be able to show a simple Python version of pseudocode Program 12-3. Also, there are no string methods for inserting and deleting characters, so we will not discuss that section in this chapter.

Character-By-Character Text Processing

Program 12-1 shows the Python version of pseudocode Program 12-1 in the textbook.

Program 12-1 (*display_characters.py*)

```
1 # Declare and initialize a string.
2 name = 'Jacob'
3
4 # Use subscript notation to display the
5 # individual characters in the string.
```

This is the Python version of
Program 12-1 in your textbook.


```
6 print(name[0])
7 print(name[1])
8 print(name[2])
9 print(name[3])
10 print(name[4])
```

Program Output

```
J
a
c
o
b
```

Program 12-2 is the Python `for` loop version of pseudocode Program 12-2 in your textbook, and Program 12-3 is the `while` loop version. Both of these programs use a loop to step through all of the characters in a string.

Program 12-2 (*loop_display_characters.py*)

```
1 # Declare and initialize a string.
2 name = 'Jacob'
3
4 # Display the characters in the string.
5 for n in name:
6     print(n)
```

Program Output

```
J
a
c
o
b
```

This is the Python `for` loop version of **Program 12-2** in your textbook.

Program 12-3 (*while_display_characters.py*)

```
1 # Declare and initialize a string.
2 name = 'Jacob'
3
4 # Initialize index
5 index = 0
6
7 # Display the characters in the string.
8 while index < len(name):
9     print(name[index])
10    index = index + 1
```

This is the Python `while` loop version of **Program 12-2** in your textbook.

Program Output

J
a
c
o
b

Character Testing Methods

Python provides string testing methods that are similar to the character testing library functions shown in Table 12-2 in your textbook. The Python methods that are similar to those functions are shown here, in Table 12-1.

Table 12-1 Character Testing Methods

Method	Description
<code>isalnum()</code>	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
<code>isalpha()</code>	Returns true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise.
<code>isdigit()</code>	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
<code>islower()</code>	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
<code>isspace()</code>	Returns true if the string contains only whitespace characters, and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (<code>\n</code>), and tabs (<code>\t</code>).
<code>isupper()</code>	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

The difference between these methods and the character testing functions discussed in the textbook is that the Python functions operate on an entire string. For example, the following code determines whether all of the characters in the string referenced by the `my_string` variable are uppercase:

```
my_string = "ABC"
if my_string.isupper():
    print('That string is all uppercase.')
```

This code will print the message *That string is all uppercase* because all of the characters in the string that is assigned to `my_string` are uppercase.

These methods can be applied to an individual character in a string, however. Here is an example:

```
my_string = "Abc"
if my_string[0].isupper():
    print('The first character is uppercase.')
```

This code determines whether the character at subscript 0 in `my_string` is uppercase (and, it is).

Program 12-4 further demonstrates the `isupper()` method. This program is the Python version of Program 12-4 in your textbook.

Program 12-4 (*count_uppercase.py*)

```
1 # Prompt the user to enter a sentence.
2 sentence = input('Enter a sentence: ')
3
4 # Initialize index (loop counter).
5 index = 0
6
7 # Initialize uppercase_count (accumulator)
8 uppercase_count = 0
9
10 # Count the number of uppercase characters.
11 while index < len(sentence):
12     if sentence[index].isupper():
13         uppercase_count = uppercase_count + 1
14     index = index + 1
15
16 # Display the number of uppercase characters.
17 print('That string has', uppercase_count, 'uppercase letters.')
```

This is the Python version of
Program 12-4 in your textbook.

Program Output

```
Enter a sentence: Mr. Jones will arrive TODAY! [Enter]
That string has 7 uppercase letters.
```

Chapter 13 Recursion

A Python function can call itself recursively, allowing you to design algorithms that recursively solve a problem. Chapter 13 in your textbook describes recursion in detail, discusses problem solving with recursion, and provides several pseudocode examples. Other than the technique of a function recursively calling itself, no new language features are introduced. In this chapter we will present Python versions of two of the pseudocode programs that are shown in the textbook. Both of these programs work exactly as the algorithms are described in the textbook. Program 13-1 is the Python version of pseudocode Program 13-2.

Program 13-1 (*recursion_demo.py*)

```
1 # This program has a recursive function.
2
3 def main():
4     # By passing the argument 5 to the message
5     # function we are telling it to display the
6     # message five times.
7     message(5)
8
9 def message(times):
10     if (times > 0):
11         print('This is a recursive function.')
12         message(times - 1)
13
14 # Call the main function.
15 main()
```

This is the Python version of
Program 13-2 in your textbook.

Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

Next, Program 13-2 is the Python version of pseudocode Program 13-3. This program recursively calculates the factorial of a number.

Program 13-2 (*factorial.py*)

```
1 # This program uses recursion to calculate
2 # the factorial of a number.
3
4 def main():
5     # Get a number from the user.
6     number = int(input('Enter a nonnegative integer: '))
```

This is the Python version of
Program 13-3 in your textbook.

```
7
8     # Get the factorial of the number.
9     fact = factorial(number)
10
11     # Display the factorial.
12     print('The factorial of', number, 'is', fact)
13
14 # The factorial function uses recursion to
15 # calculate the factorial of its argument,
16 # which is assumed to be nonnegative.
17 def factorial(num):
18     if num == 0:
19         return 1
20     else:
21         return num * factorial(num - 1)
22
23 # Call the main function.
24 main()
```

Program Output (with Input Shown in Bold)

Enter a nonnegative integer: **7** [**Enter**]
The factorial of 7 is 5040

Chapter 14 Object-Oriented Programming

Python is a powerful object-oriented language. An object is an entity that exists in the computer's memory while the program is running. An object contains data and has the ability to perform operations on its data. An object's data is commonly referred to as the object's fields, and the operations that the object performs are the object's methods.

In addition to the many objects that are provided by the Python language, you can create objects of your own design. The first step is to write a class. A class is like a blueprint. It is a declaration that specifies the methods for a particular type of object. When the program needs an object of that type, it creates an instance of the class. (An object is an instance of a class.)

Here is the general format of a class declaration in Python:

```
class ClassName:  
    Method definitions go here...
```

The first line of a class declaration begins with the keyword `class`, followed by the name of the class, followed by a colon. The class's method definitions follow this line. Method definitions are written much like regular function definitions. Because they belong to the class, method definitions must be indented.

One difference that you will notice between Python class declarations and the pseudocode class declarations in the textbook is that there are no field declarations in a Python class. This is because an object's fields are created by assignment statements that appear inside the class's methods.

Another difference that you will notice is the absence of access specifiers such as `Private` and `Public`. In Python we hide a field or method by starting its name with two underscores. This is similar to making the field or method private.

The following Python program contains a `CellPhone` class like the one shown in your textbook in Class Listing 14-3. It also has a `main` method to demonstrate the class, like that shown in pseudocode Program 14-3 in your textbook.

Program 14-1

```
1 class CellPhone:  
2     def set_manufacturer(self, manufact):  
3         self.manufacturer = manufact  
4  
5     def set_model_number(self, model):  
6         self.model_number = model  
7  
8     def set_retail_price(self, retail):
```

This is the Python version of **Class Listing 14-3** and **Program 14-1** in your textbook.

```

 9         self.  retail_price = retail
10
11     def get_manufacturer(self):
12         return self.  manufacturer
13
14     def get_model_number(self):
15         return self.  model_number
16
17     def get_retail_price(self):
18         return self.  retail_price
19
20 def main():
21     # Create a CellPhone object. The phone
22     # variable will reference the object.
23     phone = CellPhone()
24
25     # Store values in the object's fields.
26     phone.set_manufacturer("Motorola")
27     phone.set_model_number("M1000")
28     phone.set_retail_price(199.99)
29
30     # Display the values stored in the fields.
31     print('The manufacturer is', phone.get_manufacturer())
32     print('The model number is', phone.get_model_number())
33     print('The retail price is', phone.get_retail_price())
34
35 # Call the main function.
36 main()

```

Program Output

```

The manufacturer is Motorola
The model number is M1000
The retail price is 199.99

```

The CellPhone class declaration begins in line 1. It has the following method definitions:

- Lines 2 through 3: the set_manufacturer method
- Lines 5 through 6: the set_model_number method
- Lines 8 through 9: the set_retail_price method
- Lines 11 through 12: the get_manufacturer method
- Lines 14 through 15: the get_model_number method
- Lines 17 through 18: the get_retail_price method

Notice that each of the methods has a parameter named `self`. The `self` parameter is required in every method that a class has. A method operates on a specific object's data attributes. When a method executes, it must have a way of knowing which object's data attributes it is supposed to operate on. That's where the `self` parameter comes in. When a method is called, Python automatically makes its `self` parameter reference the specific object that the method is supposed to operate on.

Now let's look at the `set_manufacturer` method in lines 2 through 3. Notice that in addition to the `self` parameter, it also has a parameter named `manufact`. The statement in line 3 assigns `manufact` to `self.__manufacturer`. What is `self.__manufacturer`? Let's analyze it:

- `self` refers to a specific `CellPhone` object in memory.
- `__manufacturer` is the name of a field. The two underscores at the beginning of the field name make it private to code outside the `CellPhone` class.

So, the statement in line 3 assigns the value of the `manufact` parameter to a `CellPhone` object's `__manufacturer` field.

The `set_model_number` method, in lines 5 through 6 is similar. It has a `model` parameter that is assigned to the object's `__model_number` field.

The `set_retail_price` method, in lines 8 through 9 is also similar. It has a `retail` parameter that is assigned to the object's `__retail_price` field.

The `get_manufacturer` method, in lines 11 through 12, returns the value of the object's `__manufacturer` field. The `get_model_number` method, in lines 14 through 15, returns the value of the object's `model_number` field. The `get_retail_price` method, in lines 17 through 18, returns the object's `__retail_price` field.

Inside the `main` function, line 23 creates an instance of the `CellPhone` class in memory and assigns it to the `phone` variable. We say that the object is referenced by the `phone` variable. (Notice that Python does not require the `New` keyword, as discussed in your textbook.) Lines 26 through 28 call the object's `set_manufacturer`, `set_model_number`, and `set_retail_price` methods, passing arguments to each.

Recall that in the `CellPhone` class, the `set_manufacturer`, `set_model_number`, and `set_retail_price` methods each have two parameters. However, when we call these methods in lines 26 through 28, we pass only one argument. The first parameter in each of these methods is the `self` parameter. When you call a method, you do not pass an argument for the `self` parameter because Python automatically passes a reference to the calling object into the method's first parameter. As a result, the `self` parameter will automatically reference the object that the method is to operate on. This means that:

- In line 26, the argument "Motorola" is being passed into the `set_manufacturer` method's `manufact` parameter.
- In line 27 the argument "M1000" is being passed into the `set_model_number` method's `model` parameter
- In line 28 the 199.99 argument is being passed into the `set_retail_price` method's `retail` parameter.

Lines 31 through 33 call the `print` function to display the values of the object's fields.

Constructors

In Python, classes can have a method named `__init__` which is automatically executed when an instance of the class is created in memory. The `__init__` method is commonly known as an *initializer method* because it initializes the object's data attributes. (The name of the method starts with two underscore characters, followed by the word `init`, followed by two more underscore characters.)

Program 14-2 shows a version of the `CellPhone` class that has an `__init__` method. This is the Python version of Class Listing 14-4 combined with pseudocode Program 14-2 from your textbook.

Program 14-2

```

1 class CellPhone:
2     def __init__(self, manufact, model, retail):
3         self.manufacturer = manufact
4         self.model_number = model
5         self.retail_price = retail
6
7     def set_manufacturer(self, manufact):
8         self.manufacturer = manufact
9
10    def set_model_number(self, model):
11        self.model_number = model
12
13    def set_retail_price(self, retail):
14        self.retail_price = retail
15
16    def get_manufacturer(self):
17        return self.manufacturer
18
19    def get_model_number(self):
20        return self.model_number
21
22    def get_retail_price(self):
23        return self.retail_price

```

This is the Python version of **Class Listing 14-4** and **Program 14-2** in your textbook.

```
24
25 def main():
26     # Create a CellPhone object and initialize its
27     # fields with values passed to the __init__ method.
28     phone = CellPhone("Motorola", "M1000", 199.99)
29
30     # Display the values stored in the fields.
31     print('The manufacturer is', phone.get_manufacturer())
32     print('The model number is', phone.get_model_number())
33     print('The retail price is', phone.get_retail_price())
34
35 # Call the main function.
36 main()
```

Program Output

```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
```

The statement in line 28 creates a `CellPhone` object in memory and assigns it to the `phone` variable. Notice that the values "Motorola", "M1000", and 199.99 appear inside the parentheses after the class name. These values are passed as arguments to the class's `__init__` method.

Inheritance

The inheritance example discussed in your textbook starts with the `GradedActivity` class (see Class Listing 14-8), which is used as a superclass. The `FinalExam` class is then used as a subclass (see Class Listing 14-9). The Python versions of these classes are shown in Program 14-3. This program also has a `main` function that demonstrates how the inheritance works.

Program 14-3 (inheritance_demo.py)

```
1 class GradedActivity:
2     def set_score(self, s):
3         self. score = s
4
5     def get_score(self):
6         return self. score
7
8     def get_grade(self):
9         if self. score >= 90:
10            grade = 'A'
11        elif self. __score >= 80:
12            grade = 'B'
13        elif self. score >= 70:
14            grade = 'C'
15        elif self. score >= 60:
16            grade = 'D'
17        else:
18            grade = 'F'
19        return grade
20
21 class FinalExam(GradedActivity):
22     def  init  (self, questions, missed):
23         # Set the  num_questions and  num_missed fields.
24         self. num_questions = questions
25         self. __num_missed = missed
26
27         # Calculate the points for each question and
28         # the numeric socre for this exam.
29         self. points_each = 100.0 / questions
30         numeric_score = 100.0 - (missed * self. __points_each)
31
32         # Call the inherited set_score method to
33         # set the numeric score.
34         self.set_score(numeric_score)
35 )
36
37 def get_points_each(self):
38     return self. __points_each
39
40 def get_num_missed(self):
41     return self. num_missed
42
43 def main():
44     # Prompt the user for the number of questions
45     # on the exam.
46     questions = int(input('Enter the number of questions on the exam: '))
47
48     # Prompt the user for the number of questions
49     # missed by the student.
50     missed = int(input('Enter the number of questions that the student missed: '))
51
52     # Create a FinalExam object.
53     exam = FinalExam(questions, missed)
54
55     # Display the test results.
56     print('Each question on the exam counts', exam.get_points_each(), 'points.')
57     print('The exam score is', exam.get_score())
58     print('The exam grade is', exam.get_grade())
59
60 # Call the main function.
61 main()
```

This is the Python version of
Class Listing 14-8, Class Listing 14-9, and Program 14-3 in your textbook.

In this method, grade is a local variable.
It is **not** a class field.

Here we are calling the inherited
set_score method.

In this method,
numeric_score is a local
variable. It is **not** a class field.

Program Output

```
Enter the number of questions on the exam: 20 [Enter]
Enter the number of questions that the student missed: 3 [Enter]
Each question on the exam counts 5.0 points.
The exam score is 85.0
The exam grade is B
```

The `GradedActivity` class is declared in lines 1 through 19. Then the `FinalExam` class is declared in lines 21 through 40. Notice the first line of the `FinalExam` class in line 21:

```
class FinalExam(GradedActivity):
```

By writing `GradedActivity` inside parentheses after the class name, we are indicating that the `FinalExam` class extends the `GradedActivity` class. As a result, `GradedActivity` is the superclass and `FinalExam` is the subclass.

Polymorphism

Your textbook presents a polymorphism demonstration that uses the `Animal` class (Class Listing 14-10) as a superclass, and the `Dog` class (Class Listing 14-11) and `Cat` class (Class Listing 14-12) as subclasses of `Animal`. The Python versions of those classes are shown here. The main function and the `show_animal_info` functions are the Python equivalent of Program 14-6 in your textbook.

Program 14-4 (polymorphism.py)

```
1 class Animal:
2     def show_species(self):
3         print('I am just a regular animal.')
4
5     def make_sound(self):
6         print('Grrrrrrr')
7
8 class Dog(Animal):
9     def show_species(self):
10        print('I am a dog.')
11
12    def make_sound(self):
13        print('Woof! Woof!')
14
15 class Cat(Animal):
16    def show_species(self):
17        print('I am a cat.')
18
19    def make_sound(self):
```

```

20         print('Meow')
21
22 # Here is the main function.
23
24 def main():
25     # Create an animal object, a Dog object, and
26     # a Cat object.
27     my_animal = Animal()
28     my_dog = Dog()
29     my_cat = Cat()
30
31     # Show info about an animal.
32     print('Here is info about an animal.')
33     show_animal_info(my_animal)
34     print()
35
36     # Show info about a dog.
37     print('Here is info about a dog.')
38     show_animal_info(my_dog)
39     print()
40
41     # Show info about a cat.
42     print('Here is info about a cat.')
43     show_animal_info(my_cat)
44
45 # The show_animal_info function accepts an Animal
46 # object as an argument and displays information
47 # about it.
48
49 def show_animal_info(creature):
50     creature.show_species()
51     creature.make_sound()
52
53 # Call the main function.
54 main()

```

Program Output

```

Here is info about an animal.
I am just a regular animal.
Grrrrrrrr

```

```

Here is info about a dog.
I am a dog.
Woof! Woof!

```

Here is info about a cat.
I am a cat.
Meow

Chapter 15 GUI Applications and Event-Driven Programming

Python does not have GUI programming features built into the language itself. However, it comes with a module named Tkinter that allows you to create simple GUI programs. The name "Tkinter" is short for "Tk interface." It is named this because it provides a way for Python programmers to use a GUI library named Tk. Many other programming languages use the Tk library as well.

This chapter will give you a brief introduction to GUI programming using Python and Tkinter. We won't go over all of the features, but we will discuss an adequate number of topics to get you started.


Note: There are numerous GUI libraries available for Python. Because the Tkinter module comes with Python, we will use only it in this chapter.

A GUI program presents a window with various graphical *widgets* that the user can interact with and/or display data to the user. The `tkinter` module provides 15 widgets, which are described in Table 15-1. We won't cover all of the Tkinter widgets, but we will demonstrate how to create simple GUI programs that gather input and display data.

Table 15-1 Tkinter Widgets

Widget	Description
Button	A button that can cause an action to occur when it is clicked.
Canvas	A rectangular area that can be used to display graphics.
Checkbutton	A button that may be in either the "on" or "off" position.
Entry	An area in which the user may type a single line of input from the keyboard.
Frame	A container that can hold other widgets.
Label	An area that displays one line of text or an image.
Listbox	A list from which the user may select an item
Menu	A list of menu choices that are displayed when the user clicks a Menubutton widget.
Menubutton	A menu that is displayed on the screen and may be clicked by the user.
Message	Displays multiple lines of text.
Radiobutton	A widget that can be either selected or deselected. Radiobuttons usually appear in groups and allow the user to select one of several options.
Scale	A widget that allows the user to select a value by moving a slider along a track.
Scrollbar	Can be used with some other types of widgets to provide scrolling

	ability.
Text	A widget that allows the user to enter multiple lines of text input.
Toplevel	A container, like a Frame, but displayed in its own window.

The simplest GUI program that we can demonstrate is one that displays an empty window. Program 15-1 shows how we can do this using the `tkinter` module. When the program runs, the window shown in Figure 15-1 is displayed. To exit the program, simply click the standard Windows close button () in the upper right corner of the window.

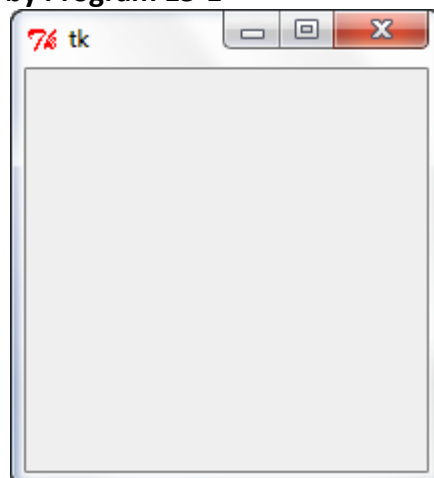
Note: Programs that use Tkinter do not always run reliably under IDLE. This is because IDLE itself uses Tkinter. You can always use IDLE's editor to write GUI programs, but for the best results, run them from your operating systems command prompt.

Program 15-1 (*empty_window1.py*)

```

1 # This program displays an empty window.
2
3 import tkinter
4
5 def main():
6     # Create the main window widget.
7     main_window = tkinter.Tk()
8
9     # Enter the tkinter main loop.
10    tkinter.mainloop()
11
12 # Call the main function.
13 main()
```

Figure 15-1 Window displayed by Program 15-1



In order for the program to use the `tkinter` module, it must have an `import` statement such as the one shown in line 3. Inside the `main` function, line 7 creates an instance of the `tkinter` module's `Tk` class, and assigns it to the `main_window` variable. This object is the root widget, which is the main window in the program. Line 10 calls the `tkinter` module's `mainloop` function. This function runs like an infinite loop until you close the main window.

Most programmers prefer to take an object-oriented approach when writing a GUI program. Rather than writing a function to create the on-screen elements of a program, it is a common practice to write a class with an `__init__` method that builds the GUI. When an instance of the class is created, the GUI appears on the screen. To demonstrate, Program 15-2 shows an object-oriented version of our program that displays an empty window. When this program runs it displays the window previously shown in Figure 15-1.

Program 15-2 (*empty_window2.py*)

```
1 # This program displays an empty window.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Create the main window widget.
8         self.main_window = tkinter.Tk()
9
10        # Enter the tkinter main loop.
11        tkinter.mainloop()
12
13 # Create an instance of the MyGUI class.
14 my_gui = MyGUI()
```

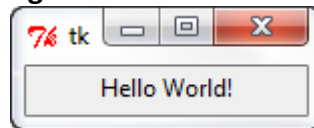
Lines 5 through 11 are the class declaration for the `MyGUI` class. The class's `__init__` method begins in line 6. Line 8 creates the root widget and assigns it to the class field `main_window`. Line 11 executes the `tkinter` module's `mainloop` function. The statement in line 14 creates an instance of the `MyGUI` class. This causes the class's `__init__` method to execute, displaying the empty window on the screen.

You can use a `Label` widget to display a single line of text in a window. To make a `Label` widget you create an instance of the `tkinter` module's `Label` class. Program 15-3 creates a window containing a `Label` widget that displays the text "Hello World!" The window is shown in Figure 15-2.

Program 15-3 (hello_world.py)

```
1 # This program displays a label with text.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Create the main window widget.
8         self.main_window = tkinter.Tk()
9
10        # Create a Label widget containing the
11        # text 'Hello World!'
12        self.label = tkinter.Label(self.main_window, \
13                                   text='Hello World!')
14
15        # Call the Label widget's pack method.
16        self.label.pack()
17
18        # Enter the tkinter main loop.
19        tkinter.mainloop()
20
21 # Create an instance of the MyGUI class.
22 my_gui = MyGUI()
```

This is the line continuation character. It allows a statement to be broken up into multiple lines.

Figure 15-2 Window displayed by Program 15-3

The `MyGUI` class in this program is very similar to the one you saw previously in Program 15-2. Its `__init__` method builds the GUI when an instance of the class is created. Line 8 creates a root widget and assigns it to `self.main_window`. The following statement appears in lines 12 and 13:

```
self.label = tkinter.Label(self.main_window, \
                             text='Hello World!')
```

First, let's explain the `\` character at the end of the first line. This is the line continuation character. In Python, when you want to break a long statement into multiple lines, you type the backslash key (`\`) at the point where you want to break the statement, and then press the Enter key. Now let's look at what the statement does.

This statement creates a `Label` widget and assigns it to `self.label`. The first argument inside the parentheses is `self.main_window`, which is a reference to the root widget. This simply

specifies that we want the Label widget to belong to the root widget. The second argument is `text='Hello World!'`. This specifies the text that we want displayed in the label.

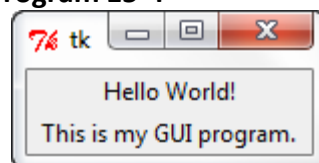
The statement in line 16 calls the Label widget's `pack` method. The `pack` method arranges a widget in its proper position, and it makes the widget visible when the main window is displayed. (You call the `pack` method for each widget in a window.) Line 19 calls the Tkinter module's `mainloop` method which displays the program's main window, shown in Figure 15-2.

Let's look at another example. Program 15-4 displays a window with two Label widgets, shown in Figure 15-3.

Program 15-4 (*hello_world2.py*)

```
1 # This program displays two labels with text.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Create the main window widget.
8         self.main_window = tkinter.Tk()
9
10        # Create two Label widget.
11        self.label1 = tkinter.Label(self.main_window, \
12                                    text='Hello World!')
13        self.label2 = tkinter.Label(self.main_window, \
14                                    text='This is my GUI program.')
15
16        # Call both Label widgets' pack method.
17        self.label1.pack()
18        self.label2.pack()
19
20        # Enter the tkinter main loop.
21        tkinter.mainloop()
22
23 # Create an instance of the MyGUI class.
24 my_gui = MyGUI()
```

Figure 15-3 Window displayed by Program 15-4

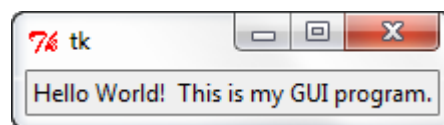


Notice that the two Label widgets are displayed with one stacked on top of the other. We can change this layout by specifying an argument to pack method, as shown in Program 15-5. When the program runs it displays the window shown in Figure 15-4.

Program 15-5 (*hello_world3.py*)

```
1 # This program uses the side='left' argument with
2 # the pack method to change the layout of the widgets.
3
4 import tkinter
5
6 class MyGUI:
7     def __init__(self):
8         # Create the main window widget.
9         self.main_window = tkinter.Tk()
10
11         # Create two Label widgets.
12         self.label1 = tkinter.Label(self.main_window, \
13                                     text='Hello World!')
14         self.label2 = tkinter.Label(self.main_window, \
15                                     text='This is my GUI program.')
16
17         # Call both Label widgets' pack method.
18         self.label1.pack(side='left')
19         self.label2.pack(side='left')
20
21         # Enter the tkinter main loop.
22         tkinter.mainloop()
23
24 # Create an instance of the MyGUI class.
25 my_gui = MyGUI()
```

Figure 15-4 Window displayed by Program 15-5



In lines 18 and 19 we call each Label widget's pack method passing the argument `side='left'`. This specifies that the widget should be positioned as far left as possible inside the parent widget. Because the `label1` widget was added to the `main_window` first, it will appear at the leftmost edge. The `label2` widget was added next, so it appears next to the `label1` widget. As a result, the labels appear side by side. The valid side arguments that you can pass to the pack method are `side='top'`, `side='bottom'`, `side='left'`, and `side='right'`.

Organizing Widgets with Frames

A frame is a container. It is a widget that can hold other widgets. Frames are useful for organizing and arranging groups of widgets in a window. For example, you can place a set of widgets in one frame and arrange them in a particular way, then place a set of widgets in another frame and arrange them in a different way. Program 15-6 demonstrates this. When the program runs it displays the window shown in Figure 15-5.

Program 15-6 (*frame_demo.py*)

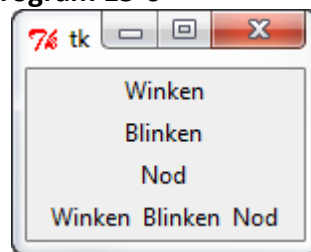
```
1 # This program creates labels in two different frames.
2
3 import tkinter
4
5 class MyGUI:
6     def __init__(self):
7         # Create the main window widget.
8         self.main_window = tkinter.Tk()
9
10        # Create two frames, one for the top of the
11        # window, and one for the bottom.
12        self.top_frame = tkinter.Frame(self.main_window)
13        self.bottom_frame = tkinter.Frame(self.main_window)
14
15        # Create three Label widgets for the
16        # top frame.
17        self.label1 = tkinter.Label(self.top_frame, \
18                                    text='Winken')
19        self.label2 = tkinter.Label(self.top_frame, \
20                                    text='Blinken')
21        self.label3 = tkinter.Label(self.top_frame, \
22                                    text='Nod')
23
24        # Pack the labels that are in the top frame.
25        # Use the side='top' argument to stack them
26        # one on top of the other.
27        self.label1.pack(side='top')
28        self.label2.pack(side='top')
29        self.label3.pack(side='top')
30
31        # Create three Label widgets for the
32        # bottom frame.
33        self.label4 = tkinter.Label(self.top_frame, \
34                                    text='Winken')
35        self.label5 = tkinter.Label(self.top_frame, \
36                                    text='Blinken')
37        self.label6 = tkinter.Label(self.top_frame, \
38                                    text='Nod')
```

```

39
40     # Pack the labels that are in the bottom frame.
41     # Use the side='left' argument to arrange them
42     # horizontally from the left of the frame.
43     self.label4.pack(side='left')
44     self.label5.pack(side='left')
45     self.label6.pack(side='left')
46
47     # Yes, we have to pack the frames too!
48     self.top_frame.pack()
49     self.bottom_frame.pack()
50
51     # Enter the tkinter main loop.
52     tkinter.mainloop()
53
54 # Create an instance of the MyGUI class.
55 my_gui = MyGUI()

```

Figure 15-5 Window displayed by Program 15-6



Take a closer look at lines 12 and 13:

```

self.top_frame = tkinter.Frame(self.main_window)
self.bottom_frame = tkinter.Frame(self.main_window)

```

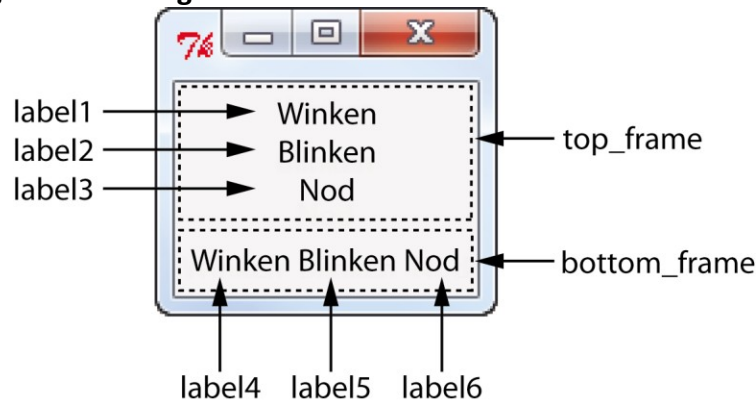
These lines create two Frame objects. The `self.main_window` argument that appears inside the parentheses causes the Frames to be added to the `main_window` widget.

Lines 17 through 22 create three Label widgets. Notice that these widgets are added to the `self.top_frame` widget. Then, lines 27 through 29 call each of the Label widgets' `pack` method, passing `side='top'` as an argument. As shown in Figure 15-6, this causes the three widgets to be stacked one on top of the other inside the Frame.

Lines 33 through 38 create three more Label widgets. These Label widgets are added to the `self.bottom_frame` widget. Then, lines 43 through 45 call each of the Label widgets' `pack` method, passing `side='left'` as an argument. As shown in Figure 15-6, this causes the three widgets to appear horizontally inside the Frame.

Lines 48 and 49 call the Frame widgets' `pack` method, which makes the Frame widgets visible. Line 52 executes the `tkinter` module's `mainloop` function.

Figure 15-6 Arrangement of widgets



Button Widgets and Info Dialog Boxes

A Button is a widget that the user can click to cause an action to take place. When you create a Button widget you can specify the text that is to appear on the face of the button, and the name of a callback function. A *callback function* is a function or method that executes when the user clicks the button.

Note: A callback function is also known as an *event handler* because it handles the event that occurs when the user clicks the button.

To demonstrate, we will look at Program 15-7. This program displays the window shown in Figure 15-7. When the user clicks the button, the program displays a separate *info dialog box*, shown in Figure 15-8. We use a function named `tkinter.messagebox.showinfo` to display the info dialog box. This is the general format of the `showinfo` function call:

```
tkinter.messagebox.showinfo(title, message)
```

In the general format, *title* is a string that is displayed in the dialog box's title bar, and *message* is an informational string that is displayed in the main part of the dialog box.

Program 14-7 (*button_demo.py*)

```
1 # This program demonstrates a Button widget.
2 # When the user clicks the Button, an
3 # info dialog box is displayed.
4
5 import tkinter
6
```

```

7 class MyGUI:
8     def  __init__ (self):
9         # Create the main window widget.
10        self.main_window = tkinter.Tk()
11
12        # Create a Button widget. The text 'Click Me!'
13        # should appear on the face of the Button. The
14        # do_something method should be executed when
15        # the user clicks the Button.
16        self.my_button = tkinter.Button(self.main_window, \
17                                       text='Click Me!', \
18                                       command=self.do_something)
19
20        # Pack the Button.
21        self.my_button.pack()
22
23        # Enter the Tkinter main loop.
24        tkinter.mainloop()
25
26        # The do_something method is a callback function
27        # for the Button widget.
28
29        def do_something(self):
30            # Display an info dialog box.
31            tkinter.messagebox.showinfo('Response', \
32                                       'Thanks for clicking the button.')
33
34 # Create an instance of the MyGUI class.
35 my_gui = MyGUI()

```

Figure 15-7 The main window displayed by Program 15-7

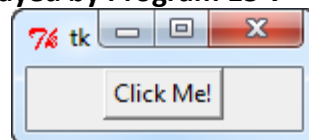
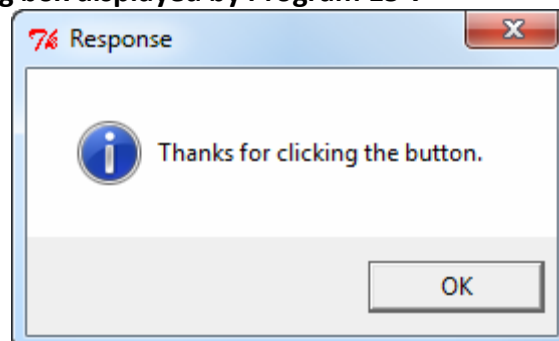


Figure 15-8 The info dialog box displayed by Program 15-7



The statement in lines 16 through 18 creates the Button widget. The first argument inside the parentheses is `self.main_window`, which is the parent widget. The `text='Click Me!'` argument specifies that the string 'Click Me!' should appear on the face of the button. The `command='self.do_something'` argument specifies the class's `do_something` method as the callback function. When the user clicks the button, the `do_something` method will execute.

The `do_something` method appears in lines 29 through 32. The method simply calls the `tkinter.messagebox.showinfo` function to display the info box shown in Figure 15-8. To dismiss the dialog box the user can click the OK button.

Creating a Quit Button

GUI programs usually have a *Quit button* (or an Exit button) that closes the program when the user clicks it. To create a Quit button in a Python program you simply create a Button widget that calls the root widget's `destroy` method as a callback function. Program 15-8 demonstrates how to do this. It is a modified version of Program 15-7, with a second Button widget added as shown in Figure 15-9.

Program 15-8 (*quit_button.py*)

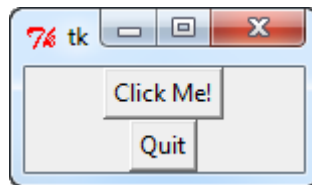
```
1 # This program has a Quit button that calls
2 # the Tk class's destroy method when clicked.
3
4 import tkinter
5
6 class MyGUI:
7     def __init__(self):
8         # Create the main window widget.
9         self.main_window = tkinter.Tk()
10
11         # Create a Button widget. The text 'Click Me!'
12         # should appear on the face of the Button. The
13         # do_something method should be executed when
14         # the user clicks the Button.
15         self.my_button = tkinter.Button(self.main_window, \
16                                         text='Click Me!', command=self.do_something)
17
18         # Create a Quit button. When this button is clicked
19         # the root widget's destroy method is called.
20         # (The main_window variable references the root widget,
21         # so the callback function is self.main_window.destroy.)
22         self.quit_button = tkinter.Button(self.main_window, \
23                                         text='Quit', command=self.main_window.destroy)
24
25         # Pack the Buttons.
26         self.my_button.pack()
27         self.quit_button.pack()
28
```

```

29         # Enter the tkinter main loop.
30         tkinter.mainloop()
31
32     # The do_something method is a callback function
33     # for the Button widget.
34
35     def do_something(self):
36         # Display an info dialog box.
37         tkinter.messagebox.showinfo('Response', \
38                                     'Thanks for clicking the button.')
39
40 # Create an instance of the MyGUI class.
41 my_gui = MyGUI()

```

Figure 15-9 The info dialog box displayed by Program 15-8



The statement in lines 22 through 23 creates the Quit button. Notice that the `self.main_window.destroy` method is used as the callback function. When the user clicks the button, this method is called and the program ends.

Getting Input with the Entry Widget

An Entry widget is a rectangular area that the user can type text into. Entry widgets are used to gather input in a GUI program. Typically, a program will have one or more Entry widgets in a window, along with a button that the user clicks to submit the data that he or she has typed into the Entry widgets. The button's callback function retrieves data from the window's Entry widgets and processes it.

You use an Entry widget's `get` method to retrieve the data that the user has typed into the widget. The `get` method returns a string, so it will have to be converted to the appropriate data type if the Entry widget is used for numeric input.

To demonstrate we will look at a program that allows the user to enter a distance in kilometers into an Entry widget, and then click a button to see that distance converted to miles. The formula for converting kilometers to miles is:

$$\text{Miles} = \text{Kilometers} \times 0.6214$$

Figure 15-10 shows the window that the program displays. To arrange the widgets in the positions shown in the figure, we will organize them in two frames, as shown in Figure 15-11.

The label that displays the prompt and the Entry widget will be stored in the `top_frame`, and their `pack` methods will be called with the `side='left'` argument. This will cause them to appear horizontally in the frame. The Convert button and the Quit button will be stored in the `bottom_frame`, and their `pack` methods will also be called with the `side='left'` argument.

Program 15-9 shows the code for the program. Figure 15-12 shows what happens when the user enters 1000 into the Entry widget and then clicks the Convert button.

Figure 15-10 The `kilo_converter` program's window

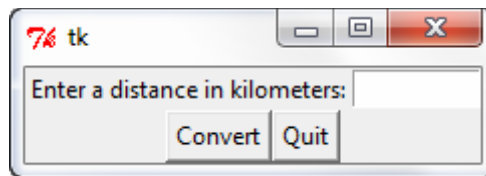
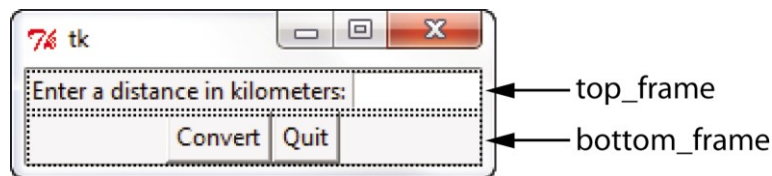


Figure 15-11 The window organized with frames



Program 15-9 (*kilo_converter.py*)

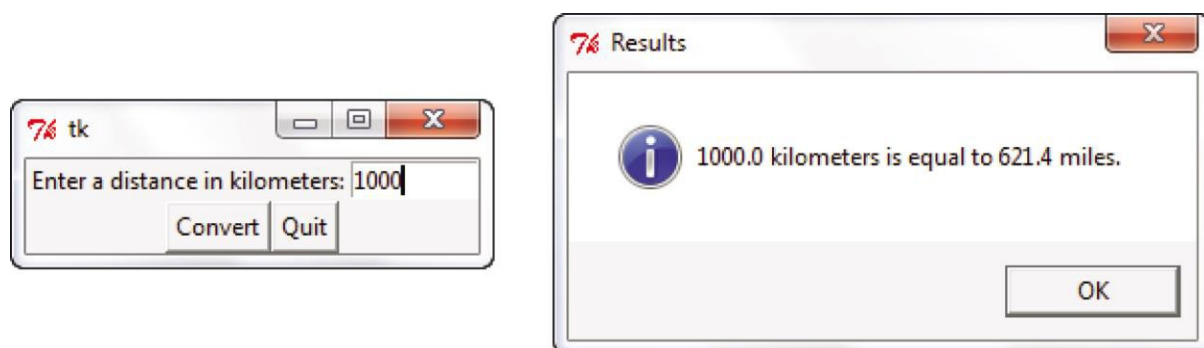
```
1 # This program converts distances in kilometers
2 # to miles. The result is displayed in an info
3 # dialog box.
4
5 import tkinter
6
7 class KiloConverterGUI:
8     def __init__(self):
9
10         # Create the main window.
11         self.main_window = tkinter.Tk()
12
13         # Create two frames to group widgets.
14         self.top_frame = tkinter.Frame(self.main_window)
15         self.bottom_frame = tkinter.Frame(self.main_window)
16
17         # Create the widgets for the top frame.
18         self.prompt_label = tkinter.Label(self.top_frame, \
19             text='Enter a distance in kilometers:')
20         self.kilo_entry = tkinter.Entry(self.top_frame, \
21             width=10)
22
```

```

23     # Pack the top frame's widgets.
24     self.prompt_label.pack(side='left')
25     self.kilo_entry.pack(side='left')
26
27     # Create the button widgets for the bottom frame.
28     self.calc_button = tkinter.Button(self.bottom_frame, \
29                                     text='Convert', \
30                                     command=self.convert)
31     self.quit_button = tkinter.Button(self.bottom_frame, \
32                                     text='Quit', \
33                                     command=self.main_window.destroy)
34
35     # Pack the buttons.
36     self.calc_button.pack(side='left')
37     self.quit_button.pack(side='left')
38
39     # Pack the frames.
40     self.top_frame.pack()
41     self.bottom_frame.pack()
42
43     # Enter the Tkinter main loop.
44     tkinter.mainloop()
45
46     # The convert method is a callback function for
47     # the Calculate button.
48
49     def convert(self):
50         # Get the value entered by the user into the
51         # kilo_entry widget.
52         kilo = float(self.kilo_entry.get())
53
54         # Convert kilometers to miles.
55         miles = kilo * 0.6214
56
57         # Display the results in an info dialog box.
58         tkinter.messagebox.showinfo('Results', \
59                                     str(kilo) + ' kilometers is equal to ' + \
60                                     str(miles) + ' miles.')
61
62 # Create an instance of the KiloConverterGUI class.
63 kilo_conv = KiloConverterGUI()

```

Figure 15-12 The info dialog box



The `convert` method, shown in lines 49 through 60 is the Convert button's callback function. The statement in line 52 calls the `kilo_entry` widget's `get` method to retrieve the data that has been typed into the widget. The value is converted to a `float` and then assigned to the `kilo` variable. The calculation in line 55 performs the conversion and assigns the results to the `miles` variable. Then, the statement in lines 58 through 60 displays the info dialog box with a message that gives the converted value.

Appendix A: Introduction to IDLE

IDLE is an integrated development environment that combines several development tools into one program, including the following:

- A Python shell running in interactive mode. You can type Python statements at the shell prompt and immediately execute them. You can also run complete Python programs.
- A text editor that color codes Python keywords and other parts of programs.
- A "check module" tool that checks a Python program for syntax errors without running the program.
- Search tools that allow you to find text in one or more files.
- Text formatting tools that help you maintain consistent indentation levels in a Python program.
- A debugger that allows you to single-step through each statement in a Python program and watch the values of variables as the statements execute.
- Several other advanced tools for developers.

The IDLE software is bundled with Python. When you install the Python interpreter, IDLE is automatically installed as well. This appendix provides a quick introduction to IDLE, and describes the basic steps of creating, saving, and executing a Python program.

Starting IDLE and Using the Python Shell

After Python is installed on your system a Python program group will appear in your Start menu's program list. One of the items in the program group will be titled *IDLE (Python GUI)*. Click this item to start IDLE and you will see the Python Shell window shown in Figure A-1. Inside this window the Python interpreter is running in interactive mode, and at the top of the window is a menu bar that provides access to all of IDLE's tools.

The `>>>` prompt indicates that the interpreter is waiting for you to type a Python statement. When you type a statement at the `>>>` prompt and press the Enter key, the statement is immediately executed. For example, Figure A-2 shows the Python Shell window after three statements have been entered and executed.

Figure A-1 IDLE shell window

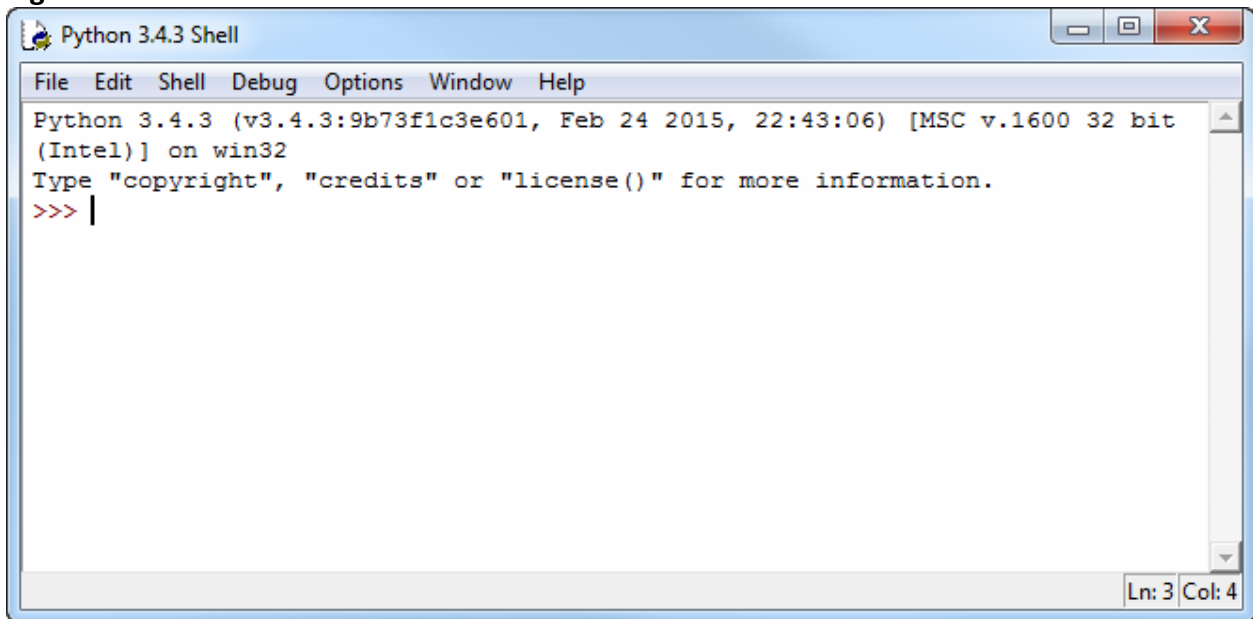
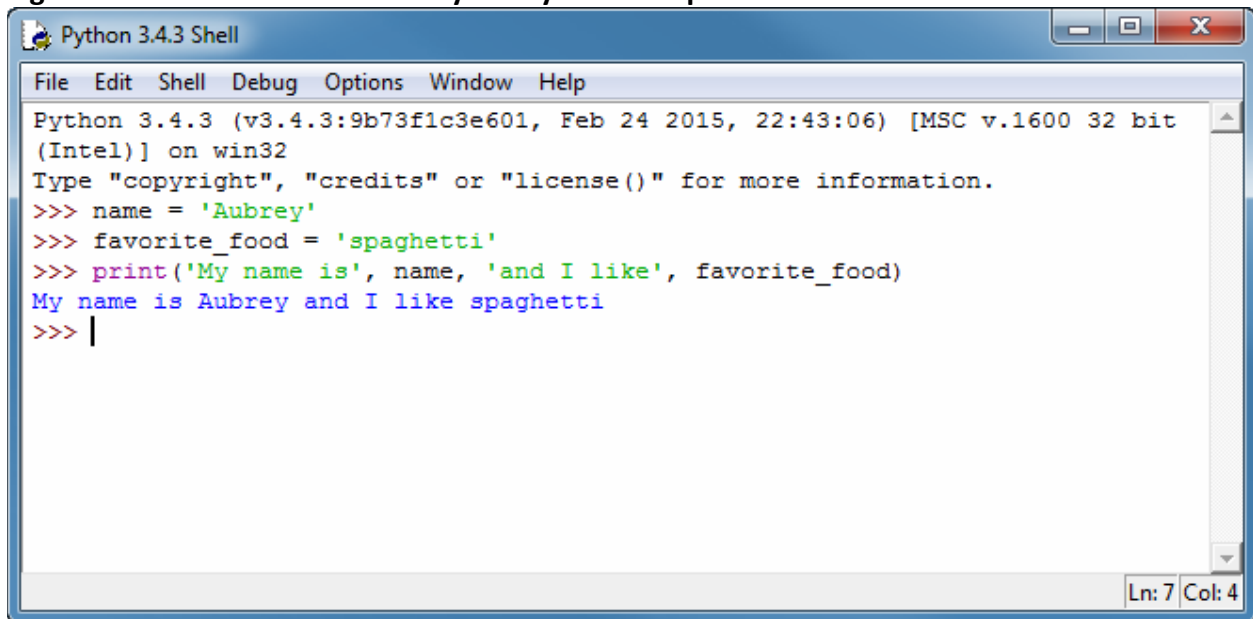
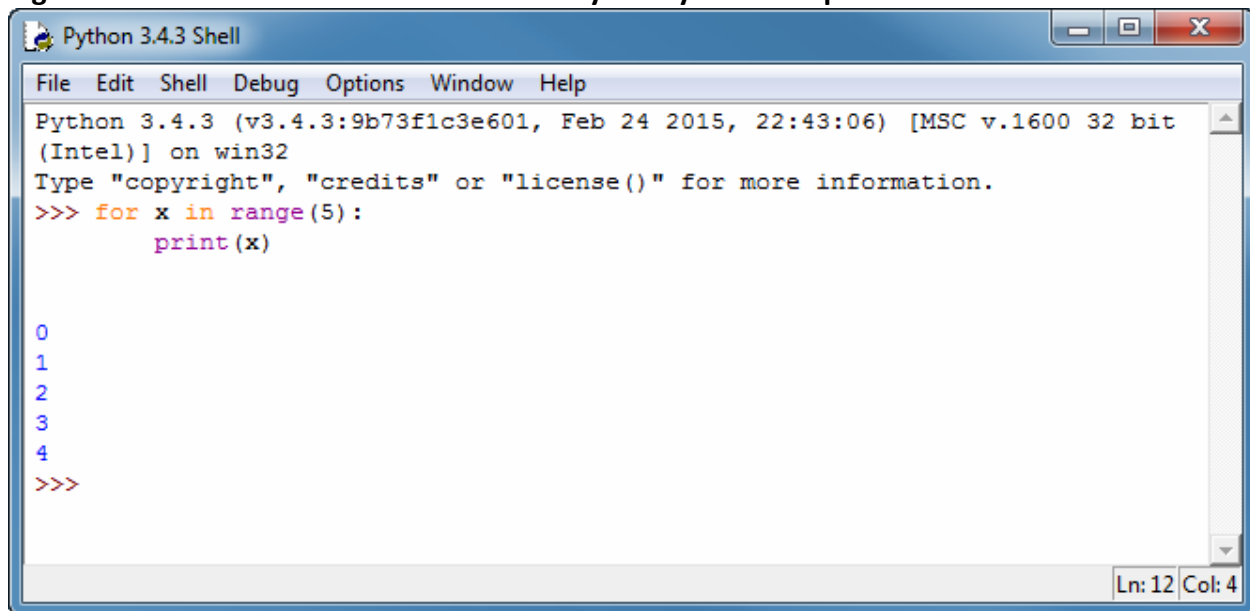


Figure A-2 Statements executed by the Python interpreter



When you type the beginning of a multiline statement, such as an `if` statement or a loop, each subsequent line is automatically indented. Pressing the Enter key on an empty line indicates the end of the multiline statement and causes the interpreter to execute it. Figure A-3 shows the Python Shell window after a `for` loop has been entered and executed.

Figure A-3 A multiline statement executed by the Python interpreter



Writing a Python Program in the IDLE Editor

To write a new Python program in IDLE you open a new editing window. As shown in Figure A-4 you click *File* on the menu bar, then click *New Window*. (Alternatively you can press Ctrl+N.) This opens a text editing window like the one shown in Figure A-5.

To open a program that already exists, click *File* on the menu bar, then *Open*. Simply browse to the file's location and select it, and it will be opened in an editor window.

Figure A-4 The File menu

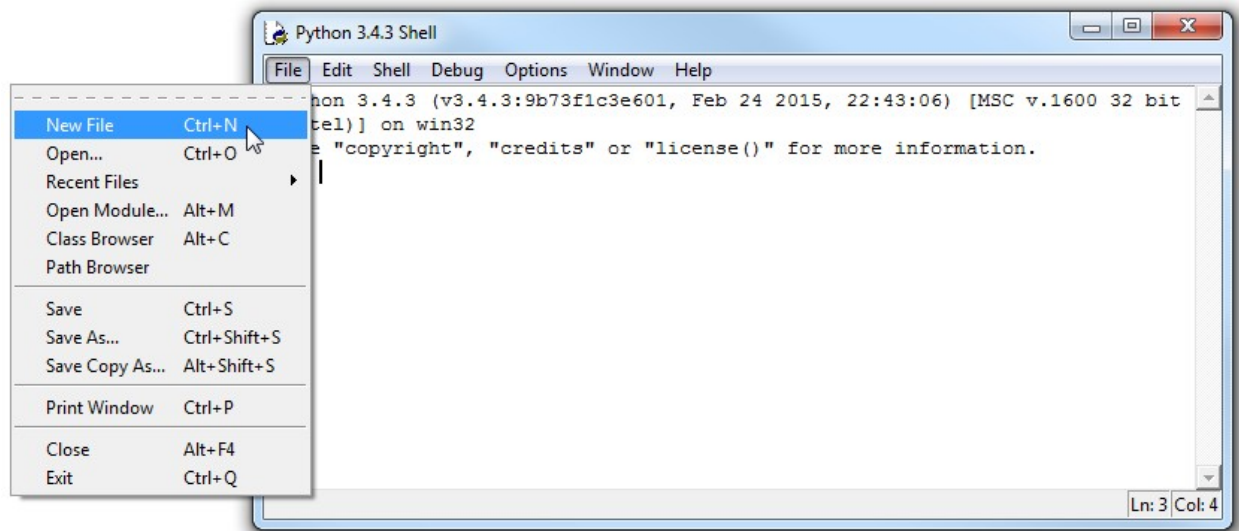
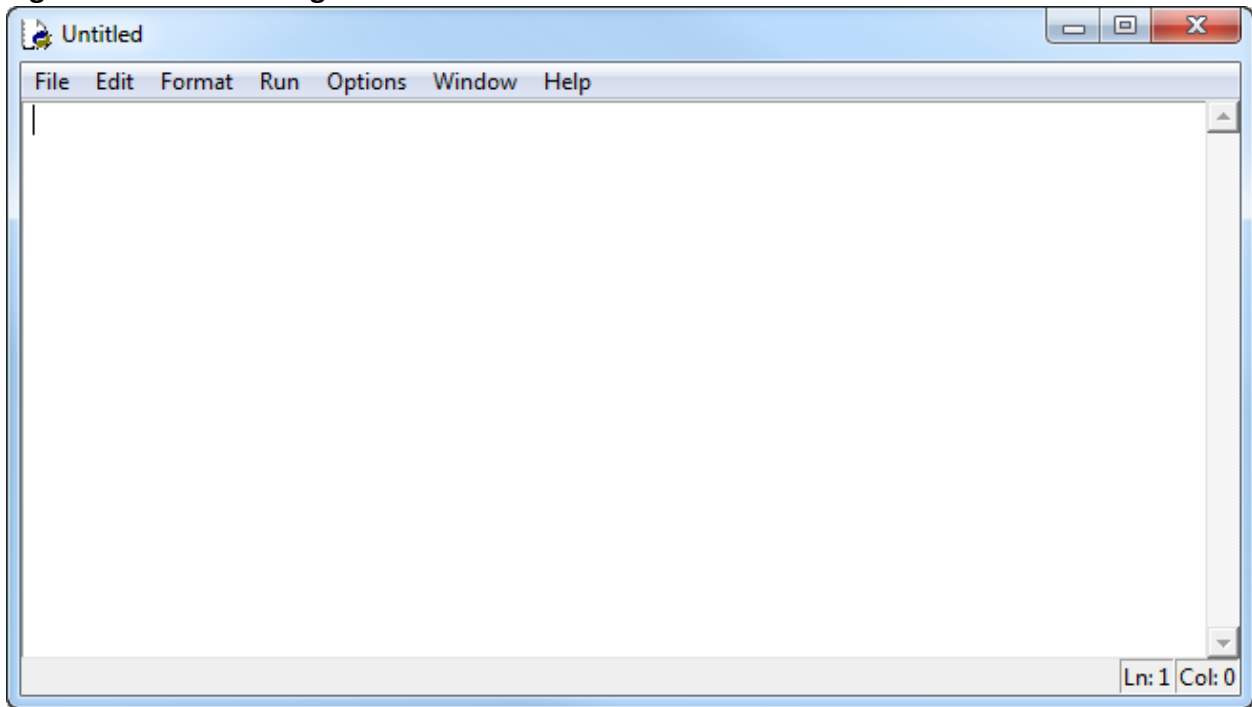


Figure A-5 A text editing window



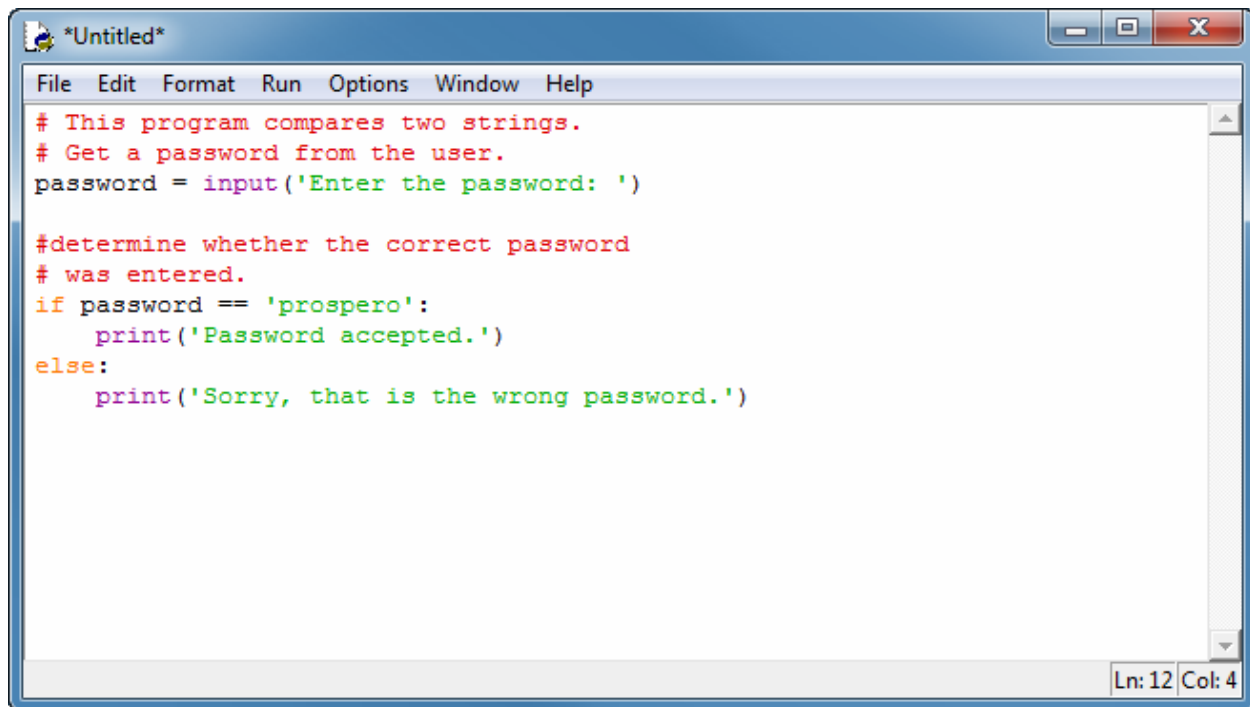
Color Coding

Code that is typed into the editor window, as well as in the Python Shell window, is colorized as follows:

- Python keywords are displayed in orange
- Comments are displayed in red
- String literals are displayed in green
- Defined names, such as the names of functions and classes, are displayed in blue.
- Built-in functions are displayed in purple

Figure A-6 shows an example of the editing window containing colorized Python code.

Figure A-6 Colorized code in the editing window

The image shows a screenshot of the Python IDLE editor window. The window has a title bar with the text '*Untitled*' and standard minimize, maximize, and close buttons. Below the title bar is a menu bar with the following options: File, Edit, Format, Run, Options, Window, and Help. The main text area contains the following Python code:

```
# This program compares two strings.  
# Get a password from the user.  
password = input('Enter the password: ')  
  
#determine whether the correct password  
# was entered.  
if password == 'prospero':  
    print('Password accepted.')  
else:  
    print('Sorry, that is the wrong password.')
```

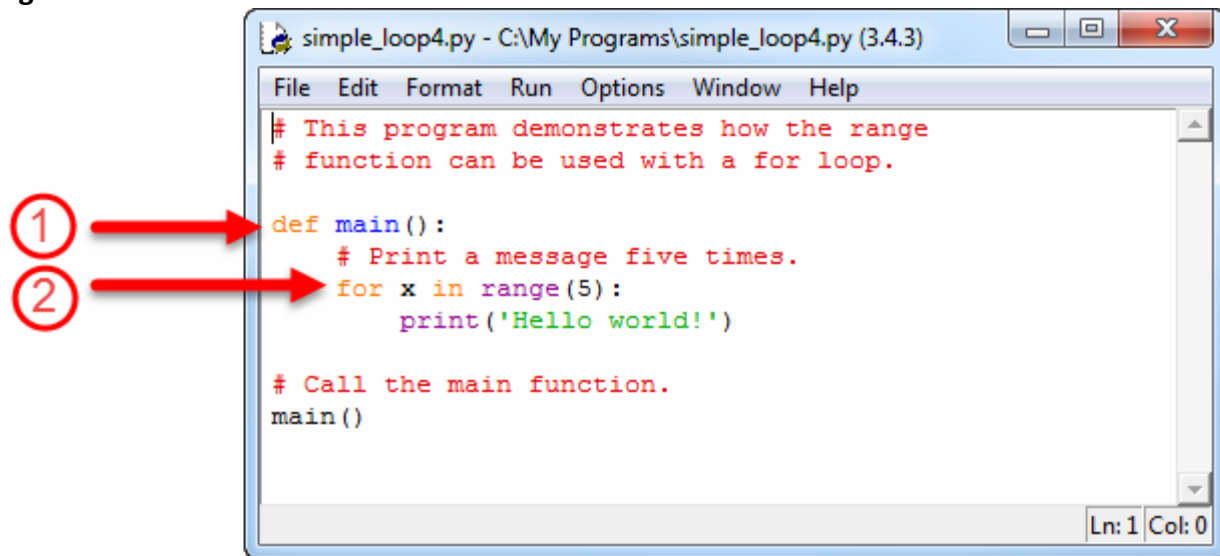
The code is color-coded: comments are in red, strings are in green, and keywords and identifiers are in black. A status bar at the bottom right of the window shows 'Ln: 12 Col: 4'.

Tip: You can change IDLE's color settings by clicking *Options* on the menu bar, then clicking *Configure IDLE*. Select the *Highlighting* tab at the top of the dialog box and you can specify colors for each element of a Python program.

Automatic Indentation

The IDLE editor has features that help you to maintain consistent indentation in your Python programs. Perhaps the most helpful of these features is automatic indentation. When you type a line that ends with a colon, such as an `if` clause, the first line of a loop, or a function header, and then press the Enter key, the editor automatically indents the lines that are entered next. For example, suppose you are typing the code shown in Figure A-7. After you press the Enter key at the end of the line marked ①, the editor will automatically indent the lines that you type next. Then, after you press the Enter key at the end of the line marked ②, the editor indents again. Pressing the Backspace key at the beginning of an indented line cancels one level of indentation.

Figure A-7 Lines that cause automatic indentation



By default, IDLE indents four spaces for each level of indentation. It is possible to change the number of spaces by clicking *Options* on the menu bar, then clicking *Configure IDLE*. Make sure *Fonts/Tabs* is selected at the top of the dialog box, and you will see a slider bar that allows you to change the number of spaces used for indentation width. However, because four spaces is the standard width for indentation in Python, it is recommended that you keep this setting.

Saving a Program

In the editor window you can save the current program by performing any of these operations from the *File* menu:

- Save
- Save As
- Save Copy As

The *Save* and *Save As* operations work just like they do in any Windows application. The *Save Copy As* operation works like *Save As*, but it leaves the original program in the editor window.

Running a Program

Once you have typed a program into the editor, you can run it by pressing the F5 key, or as shown in Figure A-8, by clicking *Run* on the editor window's menu bar, then *Run Module*. If the program has not been saved since the last modification was made, you will see the dialog box shown in Figure A-9. Click OK to save the program. When the program runs you will see its output displayed in IDLE's Python Shell window, as shown in Figure A-10.

Figure A-8 The editor window's Run menu

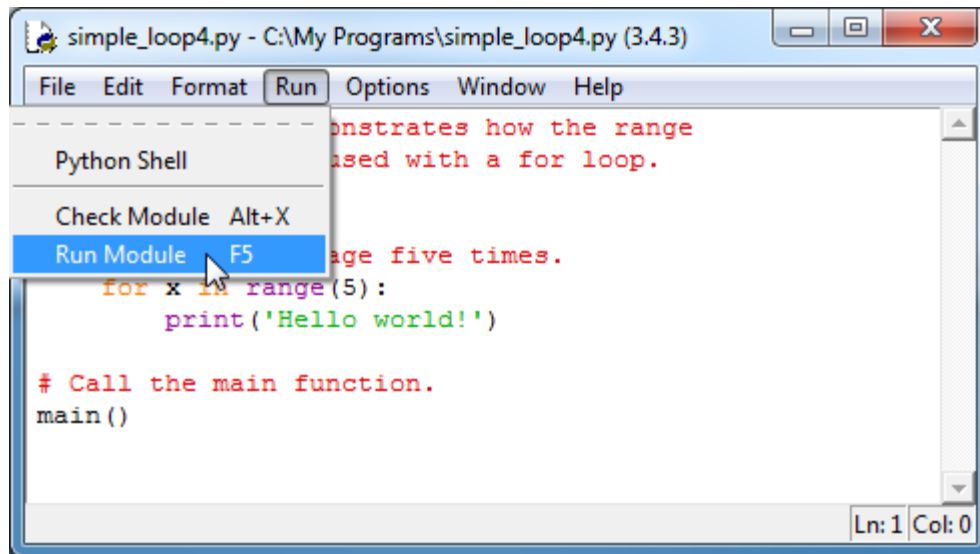


Figure A-9 Save confirmation dialog box

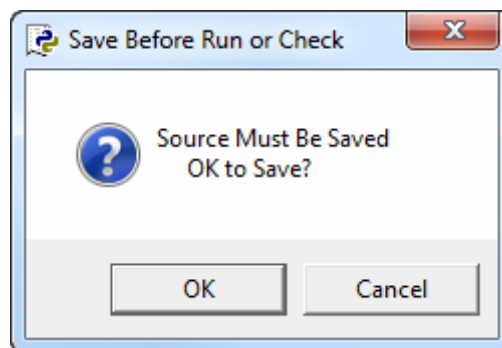
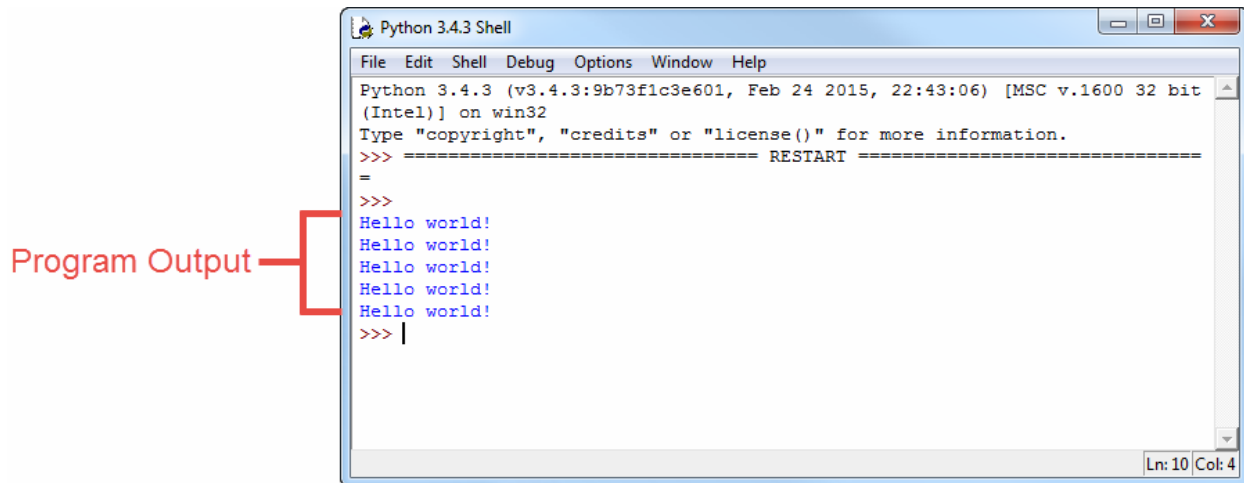
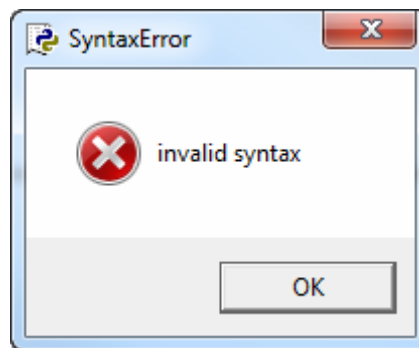


Figure A-10 Output displayed in the Python Shell window



If a program contains a syntax error, when you run the program you will see the dialog box shown in Figure A-11. After you click the OK button the editor will highlight the location of the error in the code. If you want to check the syntax of a program without trying to run it you can click *Run* on the menu bar, then *Check Module*. Any syntax errors that are found will be reported.

Figure A-11 Dialog box reporting a syntax error



Other Resources

This appendix has provided an overview for using IDLE to create, save, and execute programs. IDLE provides many more advanced features. To read about additional capabilities, see the official IDLE documentation at www.python.org/idle.