Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises

Second Edition

Springer

Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises

Second Edition

Springer

Introduction to Programming

1

Computers help us perform many different tasks. They allow us to read the news, watch videos, play games, write books, purchase goods and services, perform complex mathematical analyses, communicate with friends and family, and so much more. All of these tasks require the user to provide input, such as clicking on a video to watch, or typing the sentences that should be included in a book. In response, the computer generates output, such as printing a book, playing sounds, or displaying text and images on the screen.

Consider the examples in the previous paragraph. How did the computer know what input to request? How did it know what actions to take in response to the input? How did it know what output to generate, and in what form it should be presented? The answer to all of these questions is "a person gave the computer instructions and the computer carried them out".

An *algorithm* is a finite sequence of effective steps that solve a problem. A step is effective if it is unambiguous and possible to perform. The number of steps must be finite (rather than infinite) so that all of the steps can be completed. Recipes, assembly instructions for furniture or toys, and the steps needed to open a combination lock are examples of algorithms that we encounter in everyday life.

The form in which an algorithm is presented is flexible and can be tailored to the problem that the algorithm solves. Words, numbers, lines, arrows, pictures and other symbols can all be used to convey the steps that must be performed. While the forms that algorithms take vary, all algorithms describe steps that can be followed to complete a task successfully.

A *computer program* is a sequence of instructions that control the behaviour of a computer. The instructions tell the computer when to perform tasks like reading input and displaying results, and how to transform and manipulate values to achieve a desired outcome. An algorithm must be translated into a computer program before a computer can be used to solve a problem. The translation process is called *programming* and the person who performs the translation is referred to as a *programmer*.

Computer programs are written in computer programming languages. Programming languages have precise syntax rules that must be followed carefully. Failing to do so will cause the computer to report an error instead of executing the programmer's instructions. A wide variety of different languages have been created, each of which has its own strengths and weaknesses. Popular programming languages currently include Java, C++, JavaScript, PHP, C# and Python, among others. While there are significant differences between these languages all of them allow a programmer to control the computer's behaviour.

This book uses the Python programming language because it is relatively easy for new programmers to learn, and it can be used to solve a wide variety of problems. Python statements that read keyboard input from the user, perform calculations, and generate text output are described in the sections that follow. Later chapters describe additional programming language constructs that can be used to solve larger and more complex problems.

1.1 Storing and Manipulating Values

A *variable* is a named location in a computer's memory that holds a value. In Python, variable names must begin with a letter or an underscore, followed by any combination of letters, underscores and numbers.[1] Variables are created using assignment statements. The name of the variable that we want to create appears to the left of the assignment operator, which is denoted by =, and the value that will be stored in the variable appears to the right of the assignment operator. For example, the following statement creates a variable named `x` and stores `5` in it:

```
x = 5
```

The right side of an assignment statement can be an arbitrarily complex calculation that includes parentheses, mathematical operators, numbers, and variables that were created by earlier assignment statements (among other things). Familiar mathematical operators that Python provides include addition (+), subtraction (−), multiplication (*), division (/), and exponentiation (**). Operators are also provided for floor division (//) and modulo (%). The floor division operator computes the floor of the quotient that results when one number is divided by another while the modulo operator computes the remainder when one number is divided by another.

The following assignment statement computes the value of one plus `x` squared and stores it in a new variable named `y`.

```
y = 1 + x ** 2
```

[1] Variable names are case sensitive. As a result, `count`, `Count` and `COUNT` are distinct variable names, despite their similarity.

Python respects the usual order of operations rules for mathematical operators. Since x is 5 (from the previous assignment statement) and exponentiation has higher precedence than addition, the expression to the right of the assignment operator evaluates to 26. Then this value is stored in y.

The same variable can appear on both sides of an assignment operator. For example:

```
y = y - 6
```

While your initial reaction might be that such a statement is unreasonable, it is, in fact, a valid Python statement that is evaluated just like the assignment statements we examined previously. Specifically, the expression to the right of the assignment operator is evaluated and then the result is stored into the variable to the left of the assignment operator. In this particular case y is 26 when the statement starts executing, so 6 is subtracted from y resulting in 20. Then 20 is stored into y, replacing the 26 that was stored there previously. Subsequent uses of y will evaluate to the newly stored value of 20 (until it is changed with another assignment statement).

1.2 Calling Functions

There are some tasks that many programs have to perform such as reading input values from the keyboard, sorting a list, and computing the square root of a number. Python provides functions that perform these common tasks, as well as many others. The programs that we create will call these functions so that we don't have to solve these problems ourselves.

A function is called by using its name, followed by parentheses. Many functions require values when they are called, such as a list of names to sort or the number for which the square root will be computed. These values, called *arguments*, are placed inside the parentheses when the function is called. When a function call has multiple arguments they are separated by commas.

Many functions compute a result. This result can be stored in a variable using an assignment statement. The name of the variable appears to the left of the assignment operator and the function call appears to the right of the assignment operator. For example, the following assignment statement calls the round function, which rounds a number to the closest integer.

```
r = round(q)
```

The variable q (which must have been assigned a value previously) is passed as an argument to the round function. When the round function executes it identifies the integer that is closest to q and returns it. Then the returned integer is stored in r.

1.2.1 Reading Input

Python programs can read input from the keyboard by calling the `input` function. This function causes the program to stop and wait for the user to type something. When the user presses the `enter` key the characters typed by the user are returned by the `input` function. Then the program continues executing. Input values are normally stored in a variable using an assignment statement so that they can be used later in the program. For example, the following statement reads a value typed by the user and stores it in a variable named `a`.

```
a = input()
```

The `input` function always returns a *string*, which is computer science terminology for a sequence of characters. If the value being read is a person's name, the title of a book, or the name of a street, then storing the value as a string is appropriate. But if the value is numeric, such as an age, a temperature, or the cost of a meal at a restaurant, then the string entered by the user is normally converted to a number. The programmer must decide whether the result of the conversion should be an integer or a floating-point number (a number that can include digits to the right of the decimal point). Conversion to an integer is performed by calling the `int` function while conversion to a floating-point number is performed by calling the `float` function.

It is common to call the `int` and `float` functions in the same assignment statement that reads an input value from the user. For example, the following statements read a customer's name, the quantity of an item that they would like to purchase, and the item's price. Each of these values is stored in its own variable with an assignment statement. The name is stored as a string, the quantity is stored as an integer, and the price is stored as a floating-point number.

```
name = input("Enter your name: ")
quantity = int(input("How many items? "))
price = float(input("Cost per item? "))
```

Notice that an argument was provided to the `input` function each time it was called. This argument, which is optional, is a prompt that tells the user what to enter. The prompt must be string. It is enclosed in double quotes so that Python knows to treat the characters as a string instead of interpreting them as the names of functions or variables.

Mathematical calculations can be performed on both integers and floating-point numbers. For example, another variable can be created that holds the total cost of the items with the following assignment statement:

```
total = quantity * price
```

This statement will only execute successfully if `quantity` and `price` have been converted to numbers using the `int` and `float` functions described previously. Attempting to multiply these values without converting them to numbers will cause your Python program to crash.

1.2.2 Displaying Output

Text output is generated using the `print` function. It can be called with one argument, which is the value that will be displayed. For example, the following statements print the number `1`, the string `Hello!`, and whatever is currently stored in the variable `x`. The value in `x` could be an integer, a floating-point number, a string, or a value of some other type that we have not yet discussed. Each item is displayed on its own line.

```
print(1)
print("Hello!")
print(x)
```

Multiple values can be printed with one function call by providing several arguments to the `print` function. The additional arguments are separated by commas. For example:

```
print("When x is", x, "the value of y is", y)
```

All of these values are printed on the same line. The arguments that are enclosed in double quotes are strings that are displayed exactly as typed. The other arguments are variables. When a variable is printed, Python displays the value that is currently stored in it. A space is automatically included between each item when multiple items are printed.

The arguments to a function call can be values and variables, as shown previously. They can also be arbitrarily complex expressions involving parentheses, mathematical operators and other function calls. Consider the following statement:

```
print("The product of", x, "and", y, "is", x * y)
```

When it executes, the product, `x * y`, is computed and then displayed along with all of the other arguments to the `print` function.

1.2.3 Importing Additional Functions

Some functions, like `input` and `print` are used in many programs while others are not used as broadly. The most commonly used functions are available in all programs, while other less commonly used functions are stored in *modules* that the programmer can import when they are needed. For example, additional mathematical functions are located in the `math` module. It can be imported by including the following statement at the beginning of your program:

```
import math
```

Functions in the `math` module include `sqrt`, `ceil` and `sin`, among many others. A function imported from a module is called by using the module name,

followed by a period, followed by the name of the function and its arguments. For example, the following statement computes the square root of `y` (which must have been initialized previously) and stores the result in `z` by calling the `math` module's `sqrt` function.

```
z = math.sqrt(y)
```

Other commonly used Python modules include `random`, `time` and `sys`, among others. More information about all of these modules can be found online.

1.3 Comments

Comments give programmers the opportunity to explain what, how or why they are doing something in their program. This information can be very helpful when returning to a project after being away from it for a period of time, or when working on a program that was initially created by someone else. The computer ignores all of the comments in the program. They are only included to benefit people.

In Python, the beginning of a comment is denoted by the # character. The comment continues from the # character to the end of the line. A comment can occupy an entire line, or just part of it, with the comment appearing to the right of a Python statement.

Python files commonly begin with a comment that briefly describes the program's purpose. This allows anyone looking at the file to quickly determine what the program does without carefully examining its code. Commenting your code also makes it much easier to identify which lines perform each of the tasks needed to compute the program's results. You are strongly encouraged to write thorough comments when completing all of the exercises in this book.

1.4 Formatting Values

Sometimes the result of a mathematical calculation will be a floating-point number that has many digits to the right of the decimal point. While one might want to display all of the digits in some programs, there are other circumstances where the value must be rounded to a particular number of decimal places. Another unrelated program might output a large number of integers that need to be lined up in columns. Python's formatting constructs allow us to accomplish these, and many other, tasks.

A programmer tells Python how to format a value using a *format specifier*. The specifier is a sequence of characters that describe a variety of formatting details. It uses one character to indicate what type of formatting should be performed. For example, an `f` indicates that a value should be formatted as a floating-point number while a `d` or an `i` indicates that a value should be formatted as a decimal (base-10) integer and an `s` indicates that a value should be formatted as a string. Characters

can precede the `f`, `d`, `i` or `s` to control additional formatting details. We will only consider the problems of formatting a floating-point number so that it includes a specific number of digits to the right of the decimal point and formatting values so that they occupy some minimum number of characters (which allows values to be printed in columns that line up nicely). Many additional formatting tasks can be performed using format specifiers, but these tasks are outside the scope of this book.

A floating-point number can be formatted to include a specific number of decimal places by including a decimal point and the desired number of digits immediately ahead of the `f` in the format specifier. For example, `.2f` is used to indicate that a value should be formatted as a floating-point number with two digits to the right of the decimal point while `.7f` indicates that 7 digits should appear to the right of the decimal point. Rounding is performed when the number of digits to the right of the decimal point is reduced. Zeros are added if the number of digits is increased. The number of digits to the right of the decimal point cannot be specified when formatting integers and strings.

Integers, floating-point numbers and strings can all be formatted so that they occupy at least some minimum width. Specifying a minimum width is useful when generating output that includes columns of values that need to be lined up. The minimum number of characters to use is placed before the `d`, `i`, `f` or `s`, and before the decimal point and number of digits to the right of the decimal point (if present). For example, `8d` indicates that a value should be formatted as a decimal integer occupying a minimum of 8 characters while `6.2f` indicates that a value should be formatted as a floating-point number using a minimum of 6 characters, including the decimal point and the two digits to its right. Leading spaces are added to the formatted value, when needed, to reach the minimum number of characters.

Finally, once the correct formatting characters have been identified, a percent sign (`%`) is prepended to them. A format specifier normally appears in a string. It can be the only characters in the string, or it can be part of a longer message. Examples of complete format specifier strings include `"%8d"`, `"The amount owing is %.2f"` and `"Hello %s! Welcome aboard!"`.

Once the format specifier has been created the formatting operator, denoted by `%`, is used to format a value.[2] The string containing the format specifier appears to the left of the formatting operator. The value being formatted appears to its right. When the formatting operator is evaluated, the value on the right is inserted into the string on the left (at the location of the format specifier using the indicated formatting) to compute the operator's result. Any characters in the string that are not part of a format specifier are retained without modification. Multiple values can be formatted simultaneously by including multiple format specifiers in the string to the left of the formatting operator, and by comma separating all of the values to be formatted inside parentheses to the right of the formatting operator.

[2]Python provides several different mechanisms for formatting strings including the formatting operator, the `format` function and `format` method, template strings and, most recently, f-strings. We will use the formatting operator for all of the examples and exercises in this book but the other techniques can also be used to achieve the same results.

String formatting is often performed as part of a `print` statement. The first `print` statement in the following code segment displays the value of the variable `x`, with exactly two digits to the right of the decimal point. The second `print` statement formats two values before displaying them as part of a larger output message.

```
print("%.2f" % x)
print("%s ate %d cookies!" % (name, numCookies))
```

Several additional formatting examples are shown in the following table. The variables x, y and z have previously been assigned `12`, `-2.75` and `"Andrew"` respectively.

Code Segment:
Result:
Explanation:
	`"%d" % x`
	`"12"`
	The value stored in `x` is formatted as a decimal (base 10) integer.

Code Segment:
Result:
Explanation:
	`"%f" % y`
	`"-2.75"`
	The value stored in `y` is formatted as a floating-point number.

Code Segment:
Result:
Explanation:
	`"%d and %f" % (x, y)`
	`"12 and -2.75"`
	The value stored in `x` is formatted as a decimal (base 10) integer and the value stored in `y` is formatted as a floating-point number. The other characters in the string are retained without modification.

Code Segment:
Result:
Explanation:
	`"%.4f" % x`
	`"12.0000"`
	The value stored in `x` is formatted as a floating-point number with 4 digits to the right of the decimal point.

Code Segment:
Result:
Explanation:
	`"%.1f" % y`
	`"-2.8"`
	The value stored in `y` is formatted as a floating-point number with 1 digit to the right of the decimal point. The value was rounded when it was formatted because the number of digits to the right of the decimal point was reduced.

Code Segment:
Result:
Explanation:
	`"%10s" % z`
	`"    Andrew"`
	The value stored in `z` is formatted as a string so that it occupies at least 10 spaces. Because `z` is only 6 characters long, 4 leading spaces are included in the result.

Code Segment:
Result:
Explanation:
	`"%4s" % z`
	`"Andrew"`
	The value stored in `z` is formatted as a string so that it occupies at least 4 spaces. Because `z` is longer than the indicated minimum length, the resulting string is equal to `z`.

Code Segment:
Result:
Explanation:
	`"%8i%8i" % (x, y)`
	`"      12      -2"`
	Both `x` and `y` are formatted as decimal (base 10) integers occupying a minimum of 8 spaces. Leading spaces are added as necessary. The digits to the right of decimal point are truncated (not rounded) when `y` (a floating-point number) is formatted as an integer.

1.5 Working with Strings

Like numbers, strings can be manipulated with operators and passed to functions. Operations that are commonly performed on strings include concatenating two strings, computing the length of a string, and extracting individual characters from a string. These common operations are described in the remainder of this section. Information about other string operations can be found online.

Strings can be concatenated using the + operator. The string to the right of the operator is appended to the string to the left of the operator to form the new string. For example, the following program reads two strings from the user which are a person's first and last names. It then uses string concatenation to construct a new string which is the person's last name, followed by a comma and a space, followed by the person's first name. Then the result of the concatenation is displayed.

```
# Read the names from the user
first = input("Enter the first name: ")
last = input("Enter the last name: ")

# Concatenate the strings
both = last + ", " + first

# Display the result
print(both)
```

The number of characters in a string is referred to as a string's length. This value, which is always a non-negative integer, is computed by calling the `len` function. A string is passed to the function as its only argument and the length of that string is returned as its only result. The following example demonstrates the `len` function by computing the length of a person's name.

```
# Read the name from the user
first = input("Enter your first name: ")

# Compute its length
num_chars = len(first)

# Display the result
print("Your first name contains", num_chars, "characters")
```

Sometimes it is necessary to access individual characters within a string. For example, one might want to extract the first character from each of three strings containing a first name, middle name and last name, in order to display a person's initials.

Each character in a string has a unique integer *index*. The first character in the string has index 0 while the last character in the string has an index which is equal to the length of the string, minus one. A single character in a string is accessed by placing its index inside square brackets after the name of the variable containing the string. The following program demonstrates this by displaying a person's initials.

```
# Read the user's name
first = input("Enter your first name: ")
middle = input("Enter your middle name: ")
last = input("Enter your last name: ")

# Extract the first character from each string and concatenate them
initials = first[0] + middle[0] + last[0]

# Display the initials
print("Your initials are", initials)
```

Several consecutive characters in a string can be accessed by including two indices, separated by a colon, inside the square brackets. This is referred to as slicing a string. String slicing can be used to access multiple characters within a string in an efficient manner.

1.6 Exercises

The exercises in this chapter will allow you to put the concepts discussed previously into practice. While the tasks that they ask you to complete are generally small, solving these exercises is an important step toward the creation of larger programs that solve more interesting problems.

Exercise 1: Mailing Address

Create a program that displays your name and complete mailing address. The address should be printed in the format that is normally used in the area where you live. Your program does not need to read any input from the user.

Exercise 2: Hello

Write a program that asks the user to enter his or her name. The program should respond with a message that says hello to the user, using his or her name.

Exercise 3: Area of a Room

Write a program that asks the user to enter the width and length of a room. Once these values have been read, your program should compute and display the area of the room. The length and the width will be entered as floating-point numbers. Include units in your prompt and output message; either feet or meters, depending on which unit you are more comfortable working with.

Exercise 4: Area of a Field

Create a program that reads the length and width of a farmer's field from the user in feet. Display the area of the field in acres.

> Hint: There are 43,560 square feet in an acre.

Exercise 5: Bottle Deposits

In many jurisdictions a small deposit is added to drink containers to encourage people to recycle them. In one particular jurisdiction, drink containers holding one liter or less have a $0.10 deposit, and drink containers holding more than one liter have a $0.25 deposit.

Write a program that reads the number of containers of each size from the user. Your program should continue by computing and displaying the refund that will be received for returning those containers. Format the output so that it includes a dollar sign and two digits to the right of the decimal point.

Exercise 6: Tax and Tip

The program that you create for this exercise will begin by reading the cost of a meal ordered at a restaurant from the user. Then your program will compute the tax and tip for the meal. Use your local tax rate when computing the amount of tax owing. Compute the tip as 18 percent of the meal amount (without the tax). The output from your program should include the tax amount, the tip amount, and the grand total for the meal including both the tax and the tip. Format the output so that all of the values are displayed using two decimal places.

Exercise 7: Sum of the First n Positive Integers

Write a program that reads a positive integer, n, from the user and then displays the sum of all of the integers from 1 to n. The sum of the first n positive integers can be computed using the formula:

$$\text{sum} = \frac{(n)(n+1)}{2}$$

Exercise 8: Widgets and Gizmos

An online retailer sells two products: widgets and gizmos. Each widget weighs 75 grams. Each gizmo weighs 112 grams. Write a program that reads the number of widgets and the number of gizmos from the user. Then your program should compute and display the total weight of the parts.

Exercise 9: Compound Interest

Pretend that you have just opened a new savings account that earns 4 percent interest per year. The interest that you earn is paid at the end of the year, and is added to the balance of the savings account. Write a program that begins by reading the amount of money deposited into the account from the user. Then your program should compute and display the amount in the savings account after 1, 2, and 3 years. Display each amount so that it is rounded to 2 decimal places.

Exercise 10: Arithmetic

Create a program that reads two integers, a and b, from the user. Your program should compute and display:

- The sum of a and b
- The difference when b is subtracted from a
- The product of a and b
- The quotient when a is divided by b
- The remainder when a is divided by b

Decision Making

2

The programs that you worked with in Chap. 1 were strictly sequential. Each program's statements were executed in sequence, starting from the beginning of the program and continuing, without interruption, to its end. While sequential execution of every statement in a program can be used to solve some small exercises, it is not sufficient to solve most interesting problems.

Decision making constructs allow programs to contain statements that may or may not be executed when the program runs. Execution still begins at the top of the program and progresses toward the bottom, but some statements that are present in the program may be skipped. This allows programs to perform different tasks for different input values and greatly increases the variety of problems that a Python program can solve.

2.1 If Statements

Python programs make decisions using `if` statements. An `if` statement includes a *condition* and one or more statements that form the *body* of the `if` statement. When an `if` statement is executed, its condition is evaluated to determine whether or not the statements in its body will execute. If the condition evaluates to `True` then the body of the `if` statement executes, followed by the rest of the statements in the program. If the `if` statement's condition evaluates to `False` then the body of the `if` statement is skipped and execution continues at the first line after the body of the `if` statement.

The condition on an `if` statement can be an arbitrarily complex expression that evaluates to either `True` or `False`. Such an expression is called a Boolean expression, named after George Boole (1815–1864), who was a pioneer in formal logic. An `if` statement's condition often includes a relational operator that compares two

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_2

values, variables or complex expressions. Python's relational operators are listed below.

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

The body of an `if` statement consists of one or more statements that must be indented more than the `if` keyword. It ends before the next line that is indented the same amount as (or less than) the `if` keyword. You can choose how many spaces to use when indenting the bodies of your `if` statements. All of the programs presented in this book use two spaces for indenting, but you can use one space, or several spaces, if your prefer.[1]

The following program reads a number from the user, uses two `if` statements to store a message describing the number into the `result` variable, and then displays the message. Each `if` statement's condition uses a relational operator to determine whether or not its body, which is indented, will execute. A colon immediately follows each condition to separate the `if` statement's condition from its body.

```
# Read a number from the user
num = float(input("Enter a number: "))

# Store the appropriate message in result
if num == 0:
  result = "The number was zero"
if num != 0:
  result = "The number was not zero"

# Display the message
print(result)
```

2.2 If-Else Statements

The previous example stored one message into `result` when the number entered by the user was zero, and it stored a different message into `result` when the entered

[1] Most programmers choose to use the same number of spaces each time they indent the body of an `if` statement, though Python does not require this consistency.

number was non-zero. More generally, the conditions on the `if` statements were constructed so that exactly one of the two `if` statement bodies would execute. There is no way for both bodies to execute and there is no way for neither body to execute. Such conditions are said to be *mutually exclusive*.

An `if-else` statement consists of an `if` part with a condition and a body, and an `else` part with a body (but no condition). When the statement executes its condition is evaluated. If the condition evaluates to `True` then the body of the `if` part executes and the body of the `else` part is skipped. When the condition evaluates to `False` the body of the `if` part is skipped and the body of the `else` part executes. It is impossible for both bodies to execute, and it is impossible to skip both bodies. As a result, an `if-else` statement can be used instead of two `if` statements when one `if` statement immediately follows the other and the conditions on the `if` statements are mutually exclusive. Using an `if-else` statement is preferable because only one condition needs to be written, only one condition needs to be evaluated when the program executes, and only one condition needs to be corrected if a bug is discovered at some point in the future. The program that reports whether or not a value is zero, rewritten so that it uses an `if-else` statement, is shown below.

```
# Read a number from the user
num = float(input("Enter a number: "))

# Store the appropriate message in result
if num == 0:
  result = "The number was zero"
else:
  result = "The number was not zero"

# Display the message
print(result)
```

When the number entered by the user is zero, the condition on the `if-else` statement evaluates to `True`, so the body of the `if` part of the statement executes and the appropriate message is stored into `result`. Then the body of the `else` part of the statement is skipped. When the number is non-zero, the condition on the `if-else` statement evaluates to `False`, so the body of the `if` part of the statement is skipped. Since the body of the `if` part was skipped, the body of the `else` part is executed, storing a different message into `result`. In either case, Python goes on and runs the rest of the program, which displays the message.

2.3 If-Elif-Else Statements

An `if-elif-else` statement is used to execute exactly one of several alternatives. The statement begins with an `if` part, followed by one or more `elif` parts, followed by an `else` part. All of these parts must include a body that is indented. Each of the `if` and `elif` parts must also include a condition that evaluates to either `True` or `False`.

When an `if-elif-else` statement is executed the condition on the `if` part is evaluated first. If it evaluates to `True` then the body of the `if` part is executed and all of the `elif` and `else` parts are skipped. But if the `if` part's condition evaluates to `False` then its body is skipped and Python goes on and evaluates the condition on the first `elif` part. If this condition evaluates to `True` then the body of the first `elif` part executes and all of the remaining conditions and bodies are skipped. Otherwise Python continues by evaluating the condition on each `elif` part in sequence. This continues until a condition is found that evaluates to `True`. Then the body associated with that condition is executed and the remaining `elif` and `else` parts are skipped. If Python reaches the `else` part of the statement (because all of the conditions on the `if` and `elif` parts evaluated to `False`) then it executes the body of the `else` part.

Let's extend the previous example so that one message is displayed for positive numbers, a different message is displayed for negative numbers, and yet another different message is displayed if the number is zero. While we could solve this problem using a combination of `if` and/or `if-else` statements, this problem is well suited to an `if-elif-else` statement because exactly one of three alternatives must be executed.

```
# Read a number from the user
num = float(input("Enter a number: "))

# Store the appropriate message in result
if num > 0:
  result = "That's a positive number"
elif num < 0:
  result = "That's a negative number"
else:
  result = "That's zero"

# Display the message
print(result)
```

When the user enters a positive number the condition on the `if` part of the statement evaluates to `True` so the body of the if part executes. Once the body of the `if` part has executed, the program continues by executing the print statement on its final line. The bodies of both the `elif` part and the `else` part were skipped without evaluating the condition on the `elif` part of the statement.

When the user enters a negative number the condition on the `if` part of the statement evaluates to `False`. Python skips the body of the `if` part and goes on and evaluates the condition on the `elif` part of the statement. This condition evaluates to `True`, so the body of the `elif` part is executed. Then the `else` part is skipped and the program continues by executing the print statement.

Finally, when the user enters zero the condition on the `if` part of the statement evaluates to `False`, so the body of the `if` part is skipped and Python goes on and evaluates the condition on the `elif` part. Its condition also evaluates to `False`, so Python goes on and executes the body of the `else` part. Then the final print statement is executed.

Exactly one of an arbitrarily large number of options is executed by an `if-elif-else` statement. The statement begins with an `if` part, followed by as many `elif` parts as needed. The `else` part always appears last and its body only executes when all of the conditions on the `if` and `elif` parts evaluate to `False`.

2.4 If-Elif Statements

The `else` that appears at the end of an `if-elif-else` statement is optional. When the `else` is present, the statement selects *exactly* one of several options. Omitting the `else` selects *at most* one of several options. When an `if-elif` statement is used, none of the bodies execute when all of the conditions evaluate to `False`. Whether one of the bodies executes, or not, the program will continue executing at the first statement after the body of the final `elif` part.

2.5 Nested If Statements

The body of any `if` part, `elif` part or `else` part of any type of `if` statement can contain (almost) any Python statement, including another `if`, `if-else`, `if-elif` or `if-elif-else` statement. When one `if` statement (of any type) appears in the body of another `if` statement (of any type) the `if` statements are said to be *nested*. The following program includes a nested `if` statement.

```
# Read a number from the user
num = float(input("Enter a number: "))

# Store the appropriate message in result
if num > 0:
  # Determine what adjective should be used to describe the number
  adjective = " "
  if num >= 1000000:
    adjective = " really big "
  elif num >= 1000:
    adjective = " big "

  # Store the message for positive numbers including the appropriate adjective
  result = "That's a" + adjective + "positive number"
elif num < 0:
  result = "That's a negative number"
else:
  result = "That's zero"

# Display the message
print(result)
```

This program begins by reading a number from the user. If the number entered by the user is greater than zero then the body of the outer `if` statement is executed. It begins by assigning a string containing one space to `adjective`. Then the inner `if-elif` statement, which is nested inside the outer `if-elif-else` statement, is executed. The inner statement updates `adjective` to `really big` if the entered number is at least 1,000,000 and it updates `adjective` to `big` if the entered number is between 1,000 and 999,999. The final line in the body of the outer `if` part stores the complete message in `result` and then the bodies of the outer `elif` part and the outer `else` part are skipped because the body of the outer `if` part was executed. Finally, the program completes by executing the print statement.

Now consider what happens if the number entered by the user is less than or equal to zero. When this occurs the body of the outer `if` statement is skipped and either the body of the outer `elif` part or the body of the `else` part is executed. Both of these cases store an appropriate message in `result`. Then execution continues with the print statement at the end of the program.

2.6 Boolean Logic

A Boolean expression is an expression that evaluates to either `True` or `False`. The expression can include a wide variety of elements such as the Boolean values `True` and `False`, variables containing Boolean values, relational operators, and calls to functions that return Boolean results. Boolean expressions can also include Boolean operators that combine and manipulate Boolean values. Python includes three Boolean operators: `not`, `and`, and `or`.

The `not` operator reverses the truth of a Boolean expression. If the expression, `x`, which appears to the right of the not operator, evaluates to `True` then `not x` evaluates to `False`. If `x` evaluates to `False` then `not x` evaluates to `True`.

The behavior of any Boolean expression can be described by a *truth table*. A truth table has one column for each distinct variable in the Boolean expression, as well as a column for the expression itself. Each row in the truth table represents one combination of `True` and `False` values for the variables in the expression. A truth table for an expression having n distinct variables has 2^n rows, each of which show the result computed by the expression for a different combination of values. The truth table for the `not` operator, which is applied to a single variable, `x`, has $2^1 = 2$ rows, as shown below.

x	not x
False	True
True	False

The `and` and `or` operators combine two Boolean values to compute a Boolean result. The Boolean expression `x and y` evaluates to `True` if `x` is `True` and `y` is also `True`. If `x` is `False`, or `y` is `False`, or both `x` and `y` are `False` then `x and`

y evaluates to False. The truth table for the and operator is shown below. It has $2^2 = 4$ rows because the and operator is applied to two variables.

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

The Boolean expression x or y evaluates to True if x is True, or if y is True, or if both x and y are True. It only evaluates to False if both x and y are False. The truth table for the or operator is shown below:

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

The following Python program uses the or operator to determine whether or not the value entered by the user is one of the first 5 prime numbers. The and and not operators can be used in a similar manner when constructing a complex condition.

```
# Read an integer from the user
x = int(input("Enter an integer: "))

# Determine if it is one of the first 5 primes and report the result
if x == 2 or x == 3 or x == 5 or x == 7 or x == 11:
  print("That's one of the first 5 primes.")
else:
  print("That is not one of the first 5 primes.")
```

2.7 Exercises

The following exercises should be completed using if, if-else, if-elif, and if-elif-else statements together with the concepts that were introduced in Chap. 1. You may also find it helpful to nest an if statement inside the body of another if statement in some of your solutions.

Exercise 1: Even or Odd?

Write a program that reads an integer from the user. Then your program should display a message indicating whether the integer is even or odd.

Exercise 2: Dog Years

It is commonly said that one human year is equivalent to 7 dog years. However this simple conversion fails to recognize that dogs reach adulthood in approximately two years. As a result, some people believe that it is better to count each of the first two human years as 10.5 dog years, and then count each additional human year as 4 dog years.

Write a program that implements the conversion from human years to dog years described in the previous paragraph. Ensure that your program works correctly for conversions of less than two human years and for conversions of two or more human years. Your program should display an appropriate error message if the user enters a negative number.

Exercise 3: Vowel or Consonant

In this exercise you will create a program that reads a letter of the alphabet from the user. If the user enters `a`, `e`, `i`, `o` or `u` then your program should display a message indicating that the entered letter is a vowel. If the user enters `y` then your program should display a message indicating that sometimes y is a vowel, and sometimes y is a consonant. Otherwise your program should display a message indicating that the letter is a consonant.

Exercise 4: Name That Shape

Write a program that determines the name of a shape from its number of sides. Read the number of sides from the user and then report the appropriate name as part of a meaningful message. Your program should support shapes with anywhere from 3 up to (and including) 10 sides. If a number of sides outside of this range is entered then your program should display an appropriate error message.

Exercise 5: Month Name to Number of Days

The length of a month varies from 28 to 31 days. In this exercise you will create a program that reads the name of a month from the user as a string. Then your program should display the number of days in that month. Display “28 or 29 days” for February so that leap years are addressed.

Exercise 6: Sound Levels

The following table lists the sound level in decibels for several common noises.

Noise	Decibel Level
Jackhammer	130 dB
Gas Lawnmower	106 dB
Alarm Clock	70 dB
Quiet Room	40 dB

Write a program that reads a sound level in decibels from the user. If the user enters a decibel level that matches one of the noises in the table then your program should display a message containing only that noise. If the user enters a number of decibels between the noises listed then your program should display a message indicating which noises the value is between. Ensure that your program also generates reasonable output for a value smaller than the quietest noise in the table, and for a value larger than the loudest noise in the table.

Exercise 7: Classifying Triangles

A triangle can be classified based on the lengths of its sides as equilateral, isosceles or scalene. All three sides of an equilateral triangle have the same length. An isosceles triangle has two sides that are the same length, and a third side that is a different length. If all of the sides have different lengths then the triangle is scalene.

Write a program that reads the lengths of the three sides of a triangle from the user. Then display a message that states the triangle's type.

Repetition

3

How would you write a program that repeats the same task multiple times? You could copy the code and paste it several times, but such a solution is inelegant. It only allows the task to be performed a fixed number of times, and any enhancements or corrections need to be made to every copy of the code.

Python provides two looping constructs that overcome these limitations. Both types of loop allow statements that occur only once in your program to execute multiple times when your program runs. When used effectively, loops can perform a large number of calculations with a small number statements.

3.1 While Loops

A `while` loop causes one or more statements to execute as long as, or *while*, a condition evaluates to `True`. Like an `if` statement, a `while` loop has a condition that is followed by a body which is indented. If the `while` loop's condition evaluates to `True` then the body of the loop is executed. When the bottom of the loop body is reached, execution returns to the top of the loop, and the loop condition is evaluated again. If the condition still evaluates to `True` then the body of the loop executes for a second time. Once the bottom of the loop body is reached for the second time, execution once again returns to the top of the loop. The loop's body continues to execute until the `while` loop condition evaluates to `False`. When this occurs, the loop's body is skipped, and execution continues at the first statement after the body of the `while` loop.

Many `while` loop conditions compare a variable holding a value read from the user to some other value. When the value is read in the body of the loop the user is able to cause the loop to terminate by entering an appropriate value. Specifically,

the value entered by the user must cause the `while` loop's condition to evaluate to `False`. For example, the following code segment reads values from the user and reports whether each value is positive or negative. The loop terminates when the user enters `0`. Neither message is displayed in this case.

```
# Read the first value from the user
x = int(input("Enter an integer (0 to quit): "))

# Keep looping while the user enters a non-zero number
while x != 0:
  # Report the nature of the number
  if x > 0:
    print("That's a positive number.")
  else:
    print("That's a negative number.")

  # Read the next value from the user
  x = int(input("Enter an integer (0 to quit): "))
```

This program begins by reading an integer from the user. If the integer is `0` then the condition on the `while` loop evaluates to `False`. When this occurs, the loop body is skipped and the program terminates without displaying any output (other than the prompt for input). If the condition on the `while` loop evaluates to `True` then the body of the loop executes.

When the loop body executes the value entered by the user is compared to `0` using an `if` statement and the appropriate message is displayed. Then the next input value is read from the user at the bottom of the loop. Since the bottom of the loop has been reached control returns to the top of the loop and its condition is evaluated again. If the most recent value entered by the user is `0` then the condition evaluates to `False`. When this occurs the body of the loop is skipped and the program terminates. Otherwise the body of the loop executes again. Its body continues to execute until the user causes the loop's condition to evaluate to `False` by entering `0`.

3.2 For Loops

Like `while` loops, `for` loops cause statements that only appear in a program once to execute several times when the program runs. However the mechanism used to determine how many times those statements will execute is rather different for a `for` loop.

A `for` loop executes once *for* each item in a collection. The collection can be a range of integers, the letters in a string, or as we'll see in later chapters, the values stored in a data structure, such as a list. The syntactic structure of a `for` loop is shown below, where `<variable>`, `<collection>` and `<body>` are placeholders that must be filled in appropriately.

```
for <variable> in <collection>:
  <body>
```

The body of the loop consists of one or more Python statements that may be executed multiple times. In particular, these statements will execute once for each item in the collection. Like a `while` loop body, the body of a `for` loop is indented.

Each item in the collection is copied into `<variable>` before the loop body executes for that item. This variable is created by the `for` loop when it executes. It is not necessary to create it with an assignment statement, and any value that might have been assigned to this variable previously is overwritten at the beginning of each loop iteration. The variable can be used in the body of the loop in the same ways that any other Python variable can be used.

A collection of integers can be constructed by calling Python's `range` function. Calling `range` with one argument returns a range that starts with 0 and increases up to, but does not include, the value of the argument. For example, `range(4)` returns a range consisting of 0, 1, 2 and 3.

When two arguments are provided to `range` the collection of values returned increases from the first argument up to, but not including, the second argument. For example, `range(4, 7)` returns a range that consists of 4, 5 and 6. An empty range is returned when `range` is called with two arguments and the first argument is greater than or equal to the second. The body of the `for` loop is skipped any time a `for` loop is applied to an empty range. Execution continues with the first statement after the `for` loop's body.

The `range` function can also be called with a third argument, which is the step value used to move from the initial value in the range toward its final value. Using a step value greater than 0 results in a range that begins with the first argument and increases up to, but does not include, the second argument, incrementing by the step value each time. Using a negative step value allows a collection of decreasing values to be constructed. For example, while calling `range(0, -4)` returns an empty range, calling `range(0, -4, -1)` returns a range that consists of 0, -1, -2 and -3. Note that the step value passed to `range` as its third argument must be an integer. Problems which require a non-integer step value are often solved with a `while` loop instead of a `for` loop because of this restriction.

The following program uses a `for` loop and the range `function` to display all of the positive multiples of 3 up to (and including) a value entered by the user.

```
# Read the limit from the user
limit = int(input("Enter an integer: "))

# Display the positive multiples of 3 up to the limit
print("The multiples of 3 up to and including", limit, "are:")
for i in range(3, limit + 1, 3):
  print(i)
```

When this program executes it begins by reading an integer from the user. We will assume that the user entered 11 as we describe the execution of the rest of this program. After the input value is read, execution continues with the `print` statement that describes the program's output. Then the `for` loop begins to execute.

A range of integers is constructed that begins with 3 and goes up to, but does not include, $11 + 1 = 12$, stepping up by 3 each time. Thus the range consists of 3, 6 and

9. When the loop executes for the first time the first integer in the range is assigned to `i`, the body of the loop is executed, and `3` is displayed.

Once the loop's body has finished executing for the first time, control returns to the top of the loop and the next value in the range, which is 6, is assigned to `i`. The body of the loop executes again and displays `6`. Then control returns to the top of the loop for a second time.

The next value assigned to `i` is 9. It is displayed the next time the loop body executes. Then the loop terminates because there are no further values in the range. Normally execution would continue with the first statement after the body of the `for` loop. However, there is no such statement in this program, so the program terminates.

3.3 Nested Loops

The statements inside the body of a loop can include another loop. When this happens, the inner loop is said to be *nested* inside the outer loop. Any type of loop can be nested inside of any other type of loop. For example, the following program uses a `for` loop nested inside a `while` loop to repeat messages entered by the user until the user enters a blank message.

```
# Read the first message from the user
message = input("Enter a message (blank to quit): ")

# Loop until the message is a blank line
while message != "":
  # Read the number of times the message should be displayed
  n = int(input("How many times should it be repeated? "))

  # Display the message the number of times requested
  for i in range(n):
    print(message)

  # Read the next message from the user
  message = input("Enter a message (blank to quit): ")
```

When this program executes it begins by reading the first message from the user. If that message is not blank then the body of the `while` loop executes and the program reads the number of times to repeat the message, `n`, from the user. A range of integers is created from 0 up to, but not including, `n`. Then the body of the `for` loop prints the message `n` times because the message is displayed once for each integer in the range.

The next message is read from the user after the `for` loop has executed `n` times. Then execution returns to the top of the `while` loop, and its condition is evaluated. If the condition evaluates to `True` then the body of the `while` loop runs again. Another integer is read from the user, which overwrites the previous value of `n`, and then the `for` loop prints the message `n` times. This continues until the condition on the `while` loop evaluates to `False`. When that occurs, the body of the `while` loop is skipped and the program terminates because there are no statements to execute after the body of the `while` loop.

3.4 Exercises

The following exercises should all be completed with loops. In some cases the exercise specifies what type of loop to use. In other cases you must make this decision yourself. Some of the exercises can be completed easily with both `for` loops and `while` loops. Other exercises are much better suited to one type of loop than the other. In addition, some of the exercises require multiple loops. When multiple loops are involved one loop might need to be nested inside the other. Carefully consider your choice of loops as you design your solution to each problem.

Exercise 1: Average

In this exercise you will create a program that computes the average of a collection of values entered by the user. The user will enter 0 as a sentinel value to indicate that no further values will be provided. Your program should display an appropriate error message if the first value entered by the user is 0.

> Hint: Because the 0 marks the end of the input it should **not** be included in the average.

Exercise 2: Discount Table

A particular retailer is having a 60 percent off sale on a variety of discontinued products. The retailer would like to help its customers determine the reduced price of the merchandise by having a printed discount table on the shelf that shows the original prices and the prices after the discount has been applied. Write a program that uses a loop to generate this table, showing the original price, the discount amount, and the new price for purchases of $4.95, $9.95, $14.95, $19.95 and $24.95. Ensure that the discount amounts and the new prices are rounded to 2 decimal places when they are displayed.

Exercise 3: Temperature Conversion Table

Write a program that displays a temperature conversion table for degrees Celsius and degrees Fahrenheit. The table should include rows for all temperatures between 0 and 100 degrees Celsius that are multiples of 10 degrees Celsius. Include appropriate headings on your columns. The formula for converting between degrees Celsius and degrees Fahrenheit can be found on the Internet.

Exercise 4: No More Pennies

February 4, 2013 was the last day that pennies were distributed by the Royal Canadian Mint. Now that pennies have been phased out retailers must adjust totals so that they are multiples of 5 cents when they are paid for with cash (credit card and debit card transactions continue to be charged to the penny). While retailers have some freedom in how they do this, most choose to round to the closest nickel.

Write a program that reads prices from the user until a blank line is entered. Display the total cost of all the entered items on one line, followed by the amount due if the customer pays with cash on a second line. The amount due for a cash payment should be rounded to the nearest nickel. One way to compute the cash payment amount is to begin by determining how many pennies would be needed to pay the total. Then compute the remainder when this number of pennies is divided by 5. Finally, adjust the total down if the remainder is less than 2.5. Otherwise adjust the total up.

Exercise 5: Compute the Perimeter of a Polygon

Write a program that computes the perimeter of a polygon. Begin by reading the x and y coordinates for the first point on the perimeter of the polygon from the user. Then continue reading pairs of values until the user enters a blank line for the x-coordinate. Each time you read an additional coordinate you should compute the distance to the previous point and add it to the perimeter. When a blank line is entered for the x-coordinate your program should add the distance from the last point back to the first point to the perimeter. Then the perimeter should be displayed. Sample input and output values are shown below. The input values entered by the user are shown in bold.

```
Enter the first x-coordinate: 0
Enter the first y-coordinate: 0
Enter the next x-coordinate (blank to quit): 1
Enter the next y-coordinate: 0
Enter the next x-coordinate (blank to quit): 0
Enter the next y-coordinate: 1
Enter the next x-coordinate (blank to quit):
The perimeter of that polygon is 3.414213562373095
```

Exercise 6: Admission Price

A particular zoo determines the price of admission based on the age of the guest. Guests 2 years of age and less are admitted without charge. Children between 3 and 12 years of age cost $14.00. Seniors aged 65 years and over cost $18.00. Admission for all other guests is $23.00.

Create a program that begins by reading the ages of all of the guests in a group from the user, with one age entered on each line. The user will enter a blank line to indicate that there are no more guests in the group. Then your program should display the admission cost for the group with an appropriate message. The cost should be displayed using two decimal places.

Exercise 7: Parity Bits

A parity bit is a simple mechanism for detecting errors in data transmitted over an unreliable connection such as a telephone line. The basic idea is that an additional bit is transmitted after each group of 8 bits so that a single bit error in the transmission can be detected.

Parity bits can be computed for either even parity or odd parity. If even parity is selected then the parity bit that is transmitted is chosen so that the total number of one bits transmitted (8 bits of data plus the parity bit) is even. When odd parity is selected the parity bit is chosen so that the total number of one bits transmitted is odd.

Write a program that computes the parity bit for groups of 8 bits entered by the user using even parity. Your program should read strings containing 8 bits until the user enters a blank line. After each string is entered by the user your program should display a clear message indicating whether the parity bit should be 0 or 1. Display an appropriate error message if the user enters something other than 8 bits.

Hint: You should read the input from the user as a string. Then you can use the `count` method to help you determine the number of zeros and ones in the string. Information about the `count` method is available online.

Functions

4

As the programs that we write grow, we need to take steps to make them easier to develop and debug. One way that we can do this is by breaking the program's code into sections called *functions*.

Functions serve several important purposes: They let us write code once and then call it from many locations, they allow us to test different parts of our solution individually, and they allow us to hide (or at least set aside) the details once we have completed part of our program. Functions achieve these goals by allowing the programmer to name and set aside a collection of Python statements for later use. Then our program can cause those statements to execute whenever they are needed. The statements are named by *defining* a function. The statements are executed by *calling* a function. When the statements in a function finish executing, control *returns* to the location where the function was called and the program continues to execute from that location.

The programs that you have written previously called functions like `print`, `input`, `int` and `float`. All of these functions have already been defined by the people that created the Python programming language, and these functions can be called in any Python program. In this chapter you will learn how to define and call your own functions, in addition to calling those that have been defined previously.

A function definition begins with a line that consists of `def`, followed by the name of the function that is being defined, followed by an open parenthesis, a close parenthesis and a colon. This line is followed by the body of the function, which is the collection of statements that will execute when the function is called. As with the bodies of `if` statements and loops, the bodies of functions are indented. A function's body ends before the next line that is indented the same amount as (or less than) the line that begins with `def`. For example, the following lines of code define a function that draws a box constructed from asterisk characters.

```
def drawBox():
  print("**********")
  print("*        *")
  print("*        *")
  print("**********")
```

On their own, these lines of code do not produce any output because, while the `drawBox` function has been defined, it is never called. Defining the function sets these statements aside for future use and associates the name `drawBox` with them, but it does not execute them. A Python program that consists of only these lines is a valid program, but it will not generate any output when it is executed.

The `drawBox` function is called by using its name, followed by an open parenthesis and a close parenthesis. Adding the following line to the end of the previous program (without indenting it) will call the function and cause the box to be drawn.

```
drawBox()
```

Adding a second copy of this line will cause a second box to be drawn and adding a third copy of it will cause a third box to be drawn. More generally, a function can be called as many times as needed when solving a problem, and those calls can be made from many different locations within the program. The statements in the body of the function execute every time the function is called. When the function returns execution continues with the statement immediately after the function call.

4.1 Functions with Parameters

The `drawBox` function works correctly. It draws the particular box that it was intended to draw, but it is not flexible, and as a result, it is not as useful as it could be. In particular, our function would be more flexible and useful if it could draw boxes of many different sizes.

Many functions take *arguments* which are values provided inside the parentheses when the function is called. The function receives these argument values in *parameter variables* that are included inside the parentheses when the function is defined. The number of parameter variables in a function's definition indicates the number of arguments that must be supplied when the function is called.

We can make the `drawBox` function more useful by adding two parameters to its definition. These parameters, which are separated by a comma, will hold the width of the box and the height of the box respectively. The body of the function uses the values in the parameter variables to draw the box, as shown below. An `if` statement and the `quit` function are used to end the program immediately if the arguments provided to the function are invalid.

```
## Draw a box outlined with asterisks and filled with spaces.
#  @param width the width of the box
#  @param height the height of the box
def drawBox(width, height):
  # A box that is smaller than 2x2 cannot be drawn by this function
  if width < 2 or height < 2:
    print("Error: The width or height is too small.")
    quit()

  # Draw the top of the box
  print("*" * width)

  # Draw the sides of the box
  for i in range(height - 2):
    print("*" + " " * (width - 2) + "*")

  # Draw the bottom of the box
  print("*" * width)
```

Two arguments must be supplied when the `drawBox` function is called because its definition includes two parameter variables. When the function executes the value of the first argument will be placed in the first parameter variable, and similarly, the value of the second argument will be placed in the second parameter variable. For example, the following function call draws a box with a width of 15 characters and a height of 4 characters. Additional boxes can be drawn with different sizes by calling the function again with different arguments.

```
drawBox(15, 4)
```

In its current form the `drawBox` function always draws the outline of the box with asterisk characters and it always fills the box with spaces. While this may work well in many circumstances there could also be times when the programmer needs a box drawn or filled with different characters. To accommodate this, we are going to update `drawBox` so that it takes two additional parameters which specify the outline and fill characters respectively. The body of the function must also be updated to use these additional parameter variables, as shown below. A call to the `drawBox` function which outlines the box with at symbols and fills the box with periods is included at the end of the program.

```
## Draw a box.
#  @param width the width of the box
#  @param height the height of the box
#  @param outline the character used for the outline of the box
#  @param fill the character used to fill the box
def drawBox(width, height, outline, fill):
  # A box that is smaller than 2x2 cannot be drawn by this function
  if width < 2 or height < 2:
    print("Error: The width or height is too small.")
    quit()

  # Draw the top of the box
  print(outline * width)

  # Draw the sides of the box
  for i in range(height - 2):
    print(outline + fill * (width - 2) + outline)
```

```
    # Draw the bottom of the box
    print(outline * width)

# Demonstrate the drawBox function
drawBox(14, 5, "@", ".")
```

The programmer will have to include the outline and fill values (in addition to the width and height) every time this version of `drawBox` is called. While needing to do so might be fine in some circumstances, it will be frustrating when asterisk and space are used much more frequently than other character combinations because these arguments will have to be repeated every time the function is called. To overcome this, we will add default values for the outline and fill parameters to the function's definition. The default value for a parameter is separated from its name by an equal sign, as shown below.

```
def drawBox(width, height, outline="*", fill=" "):
```

Once this change is made `drawBox` can be called with two, three or four arguments. If `drawBox` is called with two arguments, the first argument will be placed in the `width` parameter variable and the second argument will be placed in the `height` parameter variable. The `outline` and `fill` parameter variables will hold their default values of asterisk and space respectively. These default values are used because no arguments were provided for these parameters when the function was called.

Now consider the following call to drawBox:

```
drawBox(14, 5, "@", ".")
```

This function call includes four arguments. The first two arguments are the width and height, and they are placed into those parameter variables. The third argument is the outline character. Because it has been provided, the default outline value (asterisk) is replaced with the provided value, which is an at symbol. Similarly, because the call includes a fourth argument, the default fill value is replaced with a period. The box that results from the preceding call to `drawBox` is shown below.

```
@@@@@@@@@@@@@@
@............@
@............@
@............@
@@@@@@@@@@@@@@
```

4.2 Variables in Functions

When a variable is created inside a function the variable is *local* to that function. This means that the variable only exists when the function is executing and that it can only be accessed within the body of that function. The variable ceases to exist when the function returns, and as such, it cannot be accessed after that time. The `drawBox`

function uses several variables to perform its task. These include parameter variables such as `width` and `fill` that are created when the function is called, as well as the `for` loop control variable, `i`, that is created when the loop begins to execute. All of these are local variables that can only be accessed within this function. Variables created with assignment statements in the body of a function are also local variables.

4.3 Return Values

Our box-drawing function prints characters on the screen. While it takes arguments that specify how the box will be drawn, the function does not compute a result that needs to be stored in a variable and used later in the program. But many functions do compute such a value. For example, the `sqrt` function in the `math` module computes the square root of its argument and returns this value so that it can be used in subsequent calculations. Similarly, the `input` function reads a value typed by the user and then returns it so that it can be used later in the program. Some of the functions that you write will also need to return values.

A function returns a value using the `return` keyword, followed by the value that will be returned. When the `return` executes the function ends immediately and control returns to the location where the function was called. For example, the following statement immediately ends the function's execution and returns 5 to the location from which it was called.

```
return 5
```

Functions that return values are often called on the right side of an assignment statement, but they can also be called in other contexts where a value is needed. Examples of such include an `if` statement or `while` loop condition, or as an argument to another function, such as `print` or `range`.

A function that does not return a result does not need to use the `return` keyword because the function will automatically return after the last statement in the function's body executes. However, a programmer can use the `return` keyword, without a trailing value, to force the function to return at an earlier point in its body. Any function, whether it returns a value or not, can include multiple return statements. Such a function will return as soon as any of the return statements execute.

Consider the following example. A geometric sequence is a sequence of terms that begins with some value, a, followed by an infinite number of additional terms. Each term in the sequence, beyond the first, is computed by multiplying its immediate predecessor by r, which is referred to as the common ratio. As a result, the terms in the sequence are a, ar, ar^2, ar^3, …. When r is 1, the sum of the first n terms of a geometric sequence is $a \times n$. When r is not 1, the sum of the first n terms of a geometric sequence can be computed using the following formula.

$$\text{sum} = \frac{a(1 - r^n)}{1 - r}$$

A function can be written that computes the sum of the first n terms of any geometric sequence. It will require 3 parameters: `a`, `r` and `n`, and it will need to return one result, which is the sum of the first `n` terms. The code for the function is shown below.

```
## Compute the sum of the first n terms of a geometric sequence.
#  @param a the first term in the sequence
#  @param r the common ratio for the sequence
#  @param n the number of terms to include in the sum
#  @return the sum of the first n term of the sequence
def sumGeometric(a, r, n):
  # Compute and return the sum when the common ratio is 1
  if r == 1:
    return a * n

  # Compute and return the sum when the common ratio is not 1
  s = a * (1 - r ** n) / (1 - r)

  return s
```

The function begins by using an `if` statement to determine whether or not `r` is one. If it is, the sum is computed as `a * n` and the function immediately returns this value without executing the remaining lines in the function's body. When `r` is not equal to one, the body of the `if` statement is skipped and the sum of the first `n` terms is computed and stored in `s`. Then the value stored in `s` is returned to the location from which the function was called.

The following program demonstrates the `sumGeometric` function by computing sums until the user enters zero for `a`. Each sum is computed inside the function and then returned to the location where the function was called. Then the returned value is stored in the `total` variable using an assignment statement. A subsequent statement displays `total` before the program goes on and reads the values for another sequence from the user.

```
def main():
  # Read the initial value for the first sequence
  init = float(input("Enter the value of a (0 to quit): "))

  # While the initial value is non-zero
  while init != 0:
    # Read the ratio and number of terms
    ratio = float(input("Enter the ratio, r: "))
    num = int(input("Enter the number of terms, n: "))

    # Compute and display the total
    total = sumGeometric(init, ratio, num)
    print("The sum of the first", num, "terms is", total)

    # Read the initial value for the next sequence
    init = float(input("Enter the value of a (0 to quit): "))

# Call the main function
main()
```

4.4 Importing Functions into Other Programs

One of the benefits of using functions is the ability to write a function once and then call it many times from different locations. This is easily accomplished when the function definition and call locations all reside in the same file. The function is defined and then it is called by using its name, followed by parentheses containing any arguments.

At some point you will find yourself in the situation where you want to call a function that you wrote for a previous program while solving a new problem. New programmers (and even some experienced programmers) are often tempted to copy the function from the file containing the old program into the file containing the new one, but this is an undesirable approach. Copying the function results in the same code residing in two places. As a result, when a bug is identified it will need to be corrected twice. A better approach is to import the function from the old program into the new one, similar to the way that functions are imported from Python's built-in modules.

Functions from an old Python program can be imported into a new one using the `import` keyword, followed by the name of the Python file that contains the functions of interest (without the .py extension). This allows the new program to call all of the functions in the old file, but it also causes the program in the old file to execute. While this may be desirable in some situations, we often want access to the old program's functions without actually running the program. This is normally accomplished by creating a function named `main` that contains the statements needed to solve the problem. Then one line of code at the end of the file calls the `main` function. Finally, an `if` statement is added to ensure that the `main` function does not execute when the file has been imported into another program, as shown below:

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

This structure should be used whenever you create a program that includes functions that you might want to import into another program in the future.

4.5 Exercises

Functions allow us to name sequences of Python statements and call them from multiple locations within our program. This provides several advantages compared to programs that do not define any functions including the ability to write code once and call it from several locations, and the opportunity to test different parts of our solution individually. Functions also allow a programmer to set aside some of the program's details while concentrating on other aspects of the solution. Using functions effectively will help you write better programs, especially as you take on larger problems. Functions should be used when completing all of the exercises in this chapter.

Exercise 1: Compute the Hypotenuse

Write a function that takes the lengths of the two shorter sides of a right triangle as its parameters. Return the hypotenuse of the triangle, computed using Pythagorean theorem, as the function's result. Include a main program that reads the lengths of the shorter sides of a right triangle from the user, uses your function to compute the length of the hypotenuse, and displays the result.

Exercise 2: Taxi Fare

In a particular jurisdiction, taxi fares consist of a base fare of $4.00, plus $0.25 for every 140 meters travelled. Write a function that takes the distance travelled (in kilometers) as its only parameter and returns the total fare as its only result. Write a main program that demonstrates the function.

Hint: Taxi fares change over time. Use constants to represent the base fare and the variable portion of the fare so that the program can be updated easily when the rates increase.

Exercise 3: Shipping Calculator

An online retailer provides express shipping for many of its items at a rate of $10.95 for the first item in an order, and $2.95 for each subsequent item in the same order. Write a function that takes the number of items in the order as its only parameter. Return the shipping charge for the order as the function's result. Include a main program that reads the number of items purchased from the user and displays the shipping charge.

Exercise 4: Median of Three Values

Write a function that takes three numbers as parameters, and returns the median value of those parameters as its result. Include a main program that reads three values from the user and displays their median.

Hint: The median value is the middle of the three values when they are sorted into ascending order. It can be found using if statements, or with a little bit of mathematical creativity.

Lists

5

Up until this point, every variable that we have created has held one value. The value could be a integer, a Boolean, a string, or a value of some other type. While using one variable for each value is practical for small problems it quickly becomes untenable when working with larger amounts of data. Lists help us overcome this problem by allowing several, even many, values to be stored in one variable.

A variable that holds a list is created with an assignment statement, much like the variables that we have created previously. Lists are enclosed in square brackets, and commas are used to separate adjacent values within the list. For example, the following assignment statement creates a list that contains 4 floating-point numbers and stores it in a variable named `data`. Then the values are displayed by calling the `print` function. All 4 values are displayed when the `print` function executes because `data` is the entire list of values.

```
data = [2.71, 3.14, 1.41, 1.62]
print(data)
```

A list can hold zero or more values. The empty list, which has no values in it, is denoted by `[]` (an open square bracket immediately followed by a close square bracket). Much like an integer can be initialized to 0 and then have value added to it at a later point in the program, a list can be initialized to the empty list and then have items added it to it as the program executes.

5.1 Accessing Individual Elements

Each value in a list is referred to as an *element*. The elements in a list are numbered sequentially with integers, starting from 0. Each integer identifies a specific element in the list, and is referred to as the *index* for that element. In the previous code segment the element at index 0 in `data` is 2.71 while the element at index 3 is 1.62.

An individual list element is accessed by using the list's name, immediately followed by the element's index enclosed in square brackets. For example, the following statements use this notation to display `3.14`. Notice that printing the element at index 1 displays the second element in the list because the first element in the list has index 0.

```
data = [2.71, 3.14, 1.41, 1.62]
print(data[1])
```

An individual list element can be updated using an assignment statement. The name of the list, followed by the element's index enclosed in square brackets, appears to the left of the assignment operator. The new value that will be stored at that index appears to the assignment operator's right. When the assignment statement executes, the element previously stored at the indicated index is overwritten with the new value. The other elements in the list are not impacted by this change.

Consider the following example. It creates a list that contains four elements, and then it replaces the element at index 2 with 2.30. When the `print` statement executes it will display all of the values in the list. Those values are 2.71, 3.14, 2.30 and 1.62.

```
data = [2.71, 3.14, 1.41, 1.62]
data[2] = 2.30
print(data)
```

5.2 Loops and Lists

A `for` loop executes once for each item in a collection. The collection can be a range of integers constructed by calling the `range` function. It can also be a list. The following example uses a `for` loop to total the values in `data`.

```
# Initialize data and total
data = [2.71, 3.14, 1.41, 1.62]
total = 0

# Total the values in data
for value in data:
  total = total + value

# Display the total
print("The total is", total)
```

This program begins by initializing `data` and `total` to the values shown. Then the `for` loop begins to execute. The first value in `data` is copied into `value` and then the body of the loop runs. It adds `value` to the total.

Once the body of the loop has executed for the first time control returns to the top of the loop. The second element in `data` is copied into `value`, and the loop body executes again which adds this new value to the total. This process continues until the

loop has executed once for each element in the list and the total of all of the elements has been computed. Then the result is displayed and the program terminates.

Sometimes loops are constructed which iterate over a list's indices instead of its values. To construct such a loop we need to be able to determine how many elements are in a list. This can be accomplished using the `len` function. It takes one argument, which is a list, and it returns the number of elements in the list.[1]

The `len` function can be used with the `range` function to construct a collection of integers that includes all of the indices for a list. This is accomplished by passing the length of the list as the only argument to `range`. A subset of the indices can be constructed by providing a second argument to `range`. The following program demonstrates this by using a `for` loop to iterate through all of `data`'s indices, except the first, to identify the position of the largest element in `data`.

```
# Initialize data and largest_pos
data = [1.62, 1.41, 3.14, 2.71]
largest_pos = 0

# Find the position of the largest element
for i in range(1, len(data)):
  if data[i] > data[largest_pos]:
    largest_pos = i

# Display the result
print("The largest value is", data[largest_pos], \
      "which is at index", largest_pos)
```

This program begins by initializing the `data` and `largest_pos` variables. Then the collection of values that will be used by the `for` loop is constructed using the `range` function. It's first argument is 1, and it second argument is the length of `data`, which is 4. As a result, `range` returns a collection of sequential integers from 1 up to and including 3, which is also the indices for all of the elements in `data`, except the first.

The `for` loop begins to execute by storing 1 into `i`. Then the loop body runs for the first time. It compares the value in `data` at index `i` to the value in `data` at index `largest_pos`. Since the element at index `i` is smaller, the `if` statement's condition evaluates to `False` and the body of the if statement is skipped.

Now control returns to the top of the loop. The next value in the range, which is 2, is stored into `i`, and the body of the loop executes for a second time. The value at index `i` is compared with the value at index `largest_pos`. Since the value at index `i` is larger, the body of the if statement executes, and `largest_pos` is set equal to `i`, which is 2.

The loop runs one more time with `i` equal to 3. The element at index `i` is less than the element at index `largest_pos` so the body of the if statement is skipped. Then the loop terminates and the program reports that the largest value is 3.14, which is at index 2.

[1] The `len` function returns 0 if the list passed to it is empty.

While loops can also be used when working with lists. For example, the following code segment uses a `while` loop to identify the index of the first positive value in a list. The loop uses a variable, `i`, which holds the indices of the elements in the list, starting from 0. The value in `i` increases as the program runs until either the end of the list is reached or a positive element is found.

```
# Initialize data
data = [0, -1, 4, 1, 0]

# Loop while i is a valid index and the value at index i is not a positive value
i = 0
while i < len(data) and data[i] <= 0:
  i = i + 1

# If i is less than the length of data then the loop terminated because a positive number was
# found.  Otherwise i will be equal to the length of data, indicating that a positive number
# was not found.
if i < len(data):
  print("The first positive number is at index", i)
else:
  print("The list does not contain a positive number")
```

When this program executes it begins by initializing `data` and `i`. Then the `while` loop's condition is evaluated. The value of `i`, which is 0, is less than the length of `data`, and the element at position `i` is 0, which is less than or equal to 0. As a result, the condition evaluates to `True`, the body of the loop executes, and the value of `i` increases from 0 to 1.

Control returns to the top of the `while` loop and its condition is evaluated again. The value stored in `i` is still less than the length of `data` and the value at position `i` in the list is still less than or equal to 0. As a result, the loop condition still evaluates to `True`. This causes the body of the loop to execute again, which increases the value of `i` from 1 to 2.

When `i` is 2 the loop condition evaluates to `False` because the element at position `i` is greater than or equal to 0. The loop body is skipped and execution continues with the `if` statement. It's condition evaluates to `True` because `i` is less than the length of `data`. As a result, the body of the `if` part executes and the index of the first positive number in `data`, which is 2, is displayed.

5.3 Additional List Operations

Lists can grow and shrink as a program runs. A new element can be inserted at any location in the list, and an element can be deleted based on its value or its index. Python also provides mechanisms for determining whether or not an element is present in a list, finding the index of the first occurrence of an element in a list, rearranging the elements in a list, and many other useful tasks.

Tasks like inserting a new element into a list and removing an element from a list are performed by applying a method to a list. Much like a function, a *method* is a collection of statements that can be called upon to perform a task. However, the syntax used to apply a method to a list is slightly different from the syntax used to call a function.

A method is applied to a list by using a statement that consists of a variable containing a list,[2] followed by a period, followed by the method's name. Like a function call, the name of the method is followed by parentheses that surround a comma separated collection of arguments. Some methods return a result. This result can be stored in a variable using an assignment statement, passed as an argument to another method or function call, or used as part of a calculation, just like the result returned by a function.

5.3.1 Adding Elements to a List

Elements can be added to the end of an existing list by calling the `append` method. It takes one argument, which is the element that will be added to the list. For example, consider the following program:

```
data = [2.71, 3.14, 1.41, 1.62]
data.append(2.30)
print(data)
```

The first line creates a new list of 4 elements and stores it in `data`. Then the `append` method is applied to `data` which increases its length from 4 to 5 by adding 2.30 to the end of the list. Finally, the list, which now contains 2.71, 3.14, 1.41, 1.62, and 2.30, is printed.

Elements can be inserted at any location in a list using the `insert` method. It requires two arguments, which are the index at which the element will be inserted and its value. When an element is inserted any elements to the right of the insertion point have their index increased by 1 so that there is an index available for the new element. For example, the following code segment inserts 2.30 in the middle of `data` instead of appending it to the end of the list. When this code segment executes it will display `[2.71, 3.14, 2.30, 1.41, 1.62]`.

```
data = [2.71, 3.14, 1.41, 1.62]
data.insert(2, 2.30)
print(data)
```

[2] Methods can also be applied to a list literal enclosed in square brackets using the same syntax, but there is rarely a need to do so.

5.3.2 Removing Elements from a List

The `pop` method is used to remove an element at a particular index from a list. The index of the element to remove is provided as an optional argument to `pop`. If the argument is omitted then `pop` removes the last element from the list. The `pop` method returns the value that was removed from the list as its only result. When this value is needed for a subsequent calculation it can be stored into a variable by calling `pop` on the right side of an assignment statement. Applying `pop` to an empty list is an error, as is attempting to remove an element from an index that is beyond the end of the list.

A value can also be removed from a list by calling the `remove` method. It's only argument is the value to remove (rather than the index of the value to remove). When the `remove` method executes it removes the first occurrence of its argument from the list. An error will be reported if the value passed to `remove` is not present in the list.

Consider the following example. It creates a list, and then removes two elements from it. When the first print statement executes it displays `[2.71, 3.14]` because 1.62 and 1.41 were removed from the list. The second print statement displays `1.41` because 1.41 was the last element in the list when the `pop` method was applied to it.

```
data = [2.71, 3.14, 1.41, 1.62]

data.remove(1.62)    # Remove 1.62 from the list
last = data.pop()    # Remove the last element from the list

print(data)
print(last)
```

5.3.3 Rearranging the Elements in a List

Sometimes a list has all of the correct elements in it, but they aren't in the order needed to solve a particular problem. Two elements in a list can be swapped using a series of assignment statements that read from and write to individual elements in the list, as shown in the following code segment.

```
# Create a list
data = [2.71, 3.14, 1.41, 1.62]

# Swap the element at index 1 with the element at index 3
temp = data[1]
data[1] = data[3]
data[3] = temp

# Display the modified list
print(data)
```

When these statements execute `data` is initialized to `[2.71, 3.14, 1.41, 1.62]`. Then the value at index 1, which is 3.14, is copied into `temp`. This is

followed by a line which copies the value at index 3 to index 1. Finally, the value in `temp` is copied into the list at index 3. When the print statement executes it displays `[2.71, 1.62, 1.41, 3.14]`.

There are two methods that rearrange the elements in a list. The `reverse` method reverses the order of the elements in the list, and the `sort` method sorts the elements into ascending order. Both `reverse` and `sort` can be applied to a list without providing any arguments.[3]

The following example reads a collection of numbers from the user and stores them in a list. Then it displays all of the values in sorted order.

```
# Create a new, empty list
values = []

# Read values from the user and store them in a list until a blank line is entered
line = input("Enter a number (blank line to quit): ")
while line != "":
  num = float(line)
  values.append(num)

  line = input("Enter a number (blank line to quit): ")

# Sort the values into ascending order
values.sort()

# Display the values
for v in values:
  print(v)
```

5.3.4 Searching a List

Sometimes we need to determine whether or not a particular value is present in a list. In other situations, we might want to determine the index of a value that is already known to be present in a list. Python's `in` operator and `index` method allow us to perform these tasks.

The `in` operator is used to determine whether or not a value is present in a list. The value that is being searched for is placed to the left of the operator. The list that is being searched is placed to the operator's right. Such an expression evaluates to `True` if the value is present in the list. Otherwise it evaluates to `False`.

The `index` method is used to identify the position of a particular value within a list. This value is passed to `index` as its only argument. The index of the first occurrence of the value in the list is returned as the method's result. It is an error to call the `index` method with an argument that is not present in the list. As a result,

[3] A list can only be sorted if all of the elements in it can be compared to one another with the less than operator. The less than operator is defined for many Python types include integers, floating-point numbers, strings, and lists, among others.

programmers sometimes use the `in` operator to determine whether or not a value is present in a list and then use the `index` method to determine it's location.

The following example demonstrates several of the methods and operators introduced in this section. It begins by reading integers from the user and storing them in a list. Then one additional integer is read from the user. The position of the first occurrence of this additional integer in the list of values is reported (if it is present). An appropriate message is displayed if the additional integer is not present in the list of values entered by the user.

```
# Read integers from the user until a blank line is entered and store them all in data
data = []
line = input("Enter an integer (blank line to finish): ")
while line != "":
  n = int(line)
  data.append(n)

  line = input("Enter an integer (blank line to finish): ")

# Read an additional integer from the user
x = int(input("Enter one additional integer: "))

# Display the index of the first occurrence of x (if it is present in the list)
if x in data:
  print("The first", x, "is at index", data.index(x))
else:
  print(x, "is not in the list")
```

5.4 Lists as Return Values and Arguments

Lists can be returned from functions. Like values of other types, a list is returned from a function using the `return` keyword. When the return statement executes, the function terminates and the list is returned to the location where the function was called. Then the list can be stored in a variable or used in a calculation.

Lists can also be passed as arguments to functions. Like values of other types, any lists being passed to a function are included inside the parentheses that follow the function's name when it is called. Each argument, whether it is a list or a value of another type, appears in the corresponding parameter variable inside the function.

Parameter variables that contain lists can be used in the body of a function just like parameter variables that contain values of other types. However, unlike an integer, floating-point number, string or Boolean value, changes made to a list parameter variable can impact the argument passed to the function, in addition to the value stored in the parameter variable. In particular, a change made to a list using a method (such as `append`, `pop` or `sort`) will change the value of both the parameter variable and the argument that was provided when the function was called.

Updates performed on individual list elements (where the name of the list, followed by an index enclosed in square brackets, appears on the left side of an assignment operator) also modify both the parameter variable and the argument that was provided when the function was called. However, assignments to the entire list (where only the name of the list appears to the left of the assignment operator) only impact the parameter variable. Such assignments do not impact the argument provided when the function was called.

The differences in behavior between list parameters and parameters of other types may seem arbitrary, as might the choice to have some changes apply to both the parameter variable and the argument while others only change the parameter variable. However, this is not the case. There are important technical reasons for these differences, but those details are beyond the scope of a brief introduction to Python.

5.5 Exercises

All of the exercises in this chapter should be solved using lists. The programs that you write will need to create lists, modify them, and locate values in them. Some of the exercises also require you to write functions that return lists or that take them as arguments.

Exercise 1: Sorted Order

Write a program that reads integers from the user and stores them in a list. Your program should continue reading values until the user enters 0. Then it should display all of the values entered by the user (except for the 0) in ascending order, with one value appearing on each line. Use either the `sort` method or the `sorted` function to sort the list.

Exercise 2: Reverse Order

Write a program that reads integers from the user and stores them in a list. Use 0 as a sentinel value to mark the end of the input. Once all of the values have been read your program should display them (except for the 0) in reverse order, with one value appearing on each line.

Exercise 3: Remove Outliers

When analysing data collected as part of a science experiment it may be desirable to remove the most extreme values before performing other calculations. Write a function that takes a list of values and an non-negative integer, `n`, as its parameters. The function should create a new copy of the list with the `n` largest elements and the `n` smallest elements removed. Then it should return the new copy of the list as the function's only result. The order of the elements in the returned list does not have to match the order of the elements in the original list.

Write a main program that demonstrates your function. It should read a list of numbers from the user and remove the two largest and two smallest values from it by calling the function described previously. Display the list with the outliers removed, followed by the original list. Your program should generate an appropriate error message if the user enters less than 4 values.

Exercise 4: Avoiding Duplicates

In this exercise, you will create a program that reads words from the user until the user enters a blank line. After the user enters a blank line your program should display each word entered by the user exactly once. The words should be displayed in the same order that they were first entered. For example, if the user enters:

```
first
second
first
third
second
```

then your program should display:

```
first
second
third
```

Exercise 5: Negatives, Zeros and Positives

Create a program that reads integers from the user until a blank line is entered. Once all of the integers have been read your program should display all of the negative numbers, followed by all of the zeros, followed by all of the positive numbers. Within each group the numbers should be displayed in the same order that they were entered by the user. For example, if the user enters the values 3, −4, 1, 0, −1, 0, and −2 then

your program should output the values −4, −1, −2, 0, 0, 3, and 1. Your program should display each value on its own line.

Exercise 6: List of Proper Divisors

A proper divisor of a positive integer, n, is a positive integer less than n which divides evenly into n. Write a function that computes all of the proper divisors of a positive integer. The integer will be passed to the function as its only parameter. The function will return a list containing all of the proper divisors as its only result. Complete this exercise by writing a main program that demonstrates the function by reading a value from the user and displaying the list of its proper divisors. Ensure that your main program only runs when your solution has not been imported into another file.

Exercise 7: Perfect Numbers

An integer, n, is said to be *perfect* when the sum of all of the proper divisors of n is equal to n. For example, 28 is a perfect number because its proper divisors are 1, 2, 4, 7 and 14, and $1 + 2 + 4 + 7 + 14 = 28$.

Write a function that determines whether or not a positive integer is perfect. Your function will take one parameter. If that parameter is a perfect number then your function will return `True`. Otherwise it will return `False`. In addition, write a main program that uses your function to identify and display all of the perfect numbers between 1 and 10,000.

Dictionaries

6

There are many parallels between lists and dictionaries. Like lists, dictionaries allow several, even many, values to be stored in one variable. Each element in a list has a unique integer index associated with it, and these indices must be integers that increase sequentially from zero. Similarly, each value in a dictionary has a unique *key* associated with it, but a dictionary's keys are more flexible than a list's indices. A dictionary's keys can be integers. They can also be floating-point numbers or strings. When the keys are numeric they do not have to start from zero, nor do they have to be sequential. When the keys are strings they can be any combination of characters, including the empty string. All of the keys in a dictionary must be distinct just as all of the indices in a list are distinct.

Every key in a dictionary must have a *value* associated with it. The value associated with a key can be an integer, a floating-point number, a string or a Boolean value. It can also be a list, or even another dictionary. A dictionary key and it's corresponding value are often referred to as a *key-value pair*. While the keys in a dictionary must be distinct there is no parallel restriction on the values. Consequently, the same value can be associated with multiple keys.

Starting in Python 3.7, the key-value pairs in a dictionary are always stored in the order in which they were added to the dictionary.[1] Each time a new key-value pair is added to the dictionary it is added to the end of the existing collection. There is no mechanism for inserting a key-value pair in the middle of an existing dictionary. Removing a key-value pair from the dictionary does not change the order of the remaining key-value pairs in the dictionary.

A variable that holds a dictionary is created using an assignment statement. The empty dictionary, which does not contain any key-value pairs, is denoted by `{}` (an open brace immediately followed by a close brace). A non-empty dictionary can be created by including a comma separated collection of key-value pairs inside the

[1] The order in which the key-value pairs were stored was not guaranteed to be the order in which they were added to the dictionary in earlier versions of Python.

braces. A colon is used to separate the key from its value in each key-value pair. For example, the following program creates a dictionary with three key-value pairs where the keys are strings and the values are floating-point numbers. Each key-value pair associates the name of a common mathematical constant to its value. Then all of the key-value pairs are displayed by calling the `print` function.

```
constants = {"pi": 3.14, "e": 2.71, "root 2": 1.41}
print(constants)
```

6.1 Accessing, Modifying and Adding Values

Accessing a value in a dictionary is similar to accessing a value in a list. When the index of a value in a list is known, we can use the name of the list and the index enclosed in square brackets to access the value at that location. Similarly, when the key associated with a value in a dictionary is known, we can use the name of the dictionary and the key enclosed in square brackets to access the value associated with that key.

Modifying an existing value in a dictionary and adding a new key-value pair to a dictionary are both performed using assignment statements. The name of the dictionary, along with the key enclosed in square brackets, is placed to the left of the assignment operator, and the value to associate with the key is placed to the right of the assignment operator. If the key is already present in the dictionary then the assignment statement will replace the key's current value with the value to the right of the assignment operator. If the key is not already present in the dictionary then a new key-value pair is added to it. These operations are demonstrated in the following program.

```
# Create a new dictionary with 2 key-value pairs
results = {"pass": 0, "fail": 0}

# Add a new key-value pair to the dictionary
results["withdrawal"] = 1

# Update two values in the dictionary
results["pass"] = 3
results["fail"] = results["fail"] + 1

# Display the values associated with fail, pass and withdrawal respectively
print(results["fail"])
print(results["pass"])
print(results["withdrawal"])
```

When this program executes it creates a dictionary named `results` that initially has two keys: `pass` and `fail`. The value associated with each key is 0. A third key, `withdrawal`, is added to the dictionary with the value 1 using an assignment statement. Then the value associated with `pass` is updated to 3 using a second assignment statement. The line that follows reads the current value associated with `fail`, which is 0, adds 1 to it, and then stores this new value back into the dictionary,

replacing the previous value. When the values are printed `1` (the value currently associated with `fail`) is displayed on the first line, `3` (the value currently associated with `pass`) is displayed on the second line, and `1` (the value currently associated with `withdrawal`) is displayed on the third line.

6.2 Removing a Key-Value Pair

A key-value pair is removed from a dictionary using the `pop` method. One argument, which is the key to remove, must be supplied when the method is called. When the method executes it removes both the key and the value associated with it from the dictionary. Unlike a list, it is not possible to pop the last key-value pair out of a dictionary by calling `pop` without any arguments.

The `pop` method returns the value associated with the key that is removed from the dictionary. This value can be stored into a variable using an assignment statement, or it can be used anywhere else that a value is needed, such as passing it as an argument to another function or method call, or as part of an arithmetic expression.

6.3 Additional Dictionary Operations

Some programs add key-value pairs to dictionaries where the key or the value were read from the user. Once all of the key-value pairs have been stored in the dictionary it might be necessary to determine how many there are, whether a particular key is present in the dictionary, or whether a particular value is present in the dictionary. Python provides functions, methods and operators that allow us to perform these tasks.

The `len` function, which we previously used to determine the number of elements in a list, can also be used to determine how many key-value pairs are in a dictionary. The dictionary is passed as the only argument to the function, and the number of key-value pairs is returned as the function's result. The `len` function returns 0 if the dictionary passed as an argument is empty.

The `in` operator can be used to determine whether or not a particular key or value is present in a dictionary. When searching for a key, the key appears to the left of the `in` operator and a dictionary appears to its right. The operator evaluates to `True` if the key is present in the dictionary. Otherwise it evaluates to `False`. The result returned by the `in` operator can be used anywhere that a Boolean value is needed, including in the condition of an `if` statement or `while` loop.

The `in` operator is used together with the `values` method to determine whether or not a value is present in a dictionary. The value being searched for appears to the left of the `in` operator and a dictionary, with the `values` method applied to it, appears to its right. For example, the following code segment determines whether or not any of the values in dictionary `d` are equal to the value that is currently stored in variable `x`.

```
if x in d.values():
  print("At least one of the values in d is", x)
else:
  print("None of the values in d are", x)
```

6.4 Loops and Dictionaries

A `for` loop can be used to iterate over all of the keys in a dictionary, as shown below. A different key from the dictionary is stored into the `for` loop's variable, `k`, each time the loop body executes.

```
# Create a dictionary
constants = {"pi": 3.14, "e": 2.71, "root 2": 1.41}

# Print all of the keys and values with nice formatting
for k in constants:
  print("The value associated with", k, "is", constants[k])
```

When this program executes it begins by creating a new dictionary that contains three key-value pairs. Then the `for` loop iterates over the keys in the dictionary. The first key in the dictionary, which is `pi`, is stored into `k`, and the body of the loop executes. It prints out a meaningful message that includes both `pi` and its value, which is `3.14`. Then control returns to the top of the loop and `e` is stored into `k`. The loop body executes for a second time and displays a message indicating that the value of `e` is `2.71`. Finally, the loop executes for a third time with `k` equal to `root 2` and the final message is displayed.

A `for` loop can also be used to iterate over the values in a dictionary (instead of the keys). This is accomplished by applying the `values` method, which does not take an arguments, to a dictionary to create the collection of values used by the `for` loop. For example, the following program computes the sum of all of the values in a dictionary. When it executes, `constants.values()` will be a collection that includes 3.14, 2.71 and 1.41. Each of these values is stored in `v` as the `for` loop runs, and this allows the total to be computed without using any of the dictionary's keys.

```
# Create a dictionary
constants = {"pi": 3.14, "e": 2.71, "root 2": 1.41}

# Compute the sum of all the value values in the dictionary
total = 0
for v in constants.values():
  total = total + v

# Display the total
print("The total is", total)
```

Some problems involving dictionaries are better solved with `while` loops than `for` loops. For example, the following program uses a `while` loop to read strings

from the user until 5 unique values have been entered. Then all of the strings are displayed with their counts.

```
# Count how many times each string is entered by the user
counts = {}

# Loop until 5 distinct strings have been entered
while len(counts) < 5:
  s = input("Enter a string: ")

  # If s is already a key in the dictionary then increase its count by 1.  Otherwise add s to the
  # dictionary with a count of 1.
  if s in counts:
    counts[s] = counts[s] + 1
  else:
    counts[s] = 1

# Displays all of the strings and their counts
for k in counts:
  print(k, "occurred", counts[k], "times")
```

When this program executes it begins by creating an empty dictionary. Then the `while` loop condition is evaluated. It determines how many key-value pairs are in the dictionary using the `len` function. Since the number of key-value pairs is initially 0, the condition evaluates to `True` and the loop body executes.

Each time the loop body executes a string is read from the user. Then the `in` operator is used to determine whether or not the string is already a key in the dictionary. If so, the count associated with the key is increased by one. Otherwise the string is added to the dictionary as a new key with a value of 1. The loop continues executing until the dictionary contains 5 key-value pairs. Once this occurs, all of the strings that were entered by the user are displayed, along with their associated values.

6.5 Dictionaries as Arguments and Return Values

Dictionaries can be passed as arguments to functions, just like values of other types. As with lists, a change made to a parameter variable that contains a dictionary can modify both the parameter variable and the argument passed to the function. For example, inserting or deleting a key-value pair will modify both the parameter variable and the argument, as will modifying the value associated with one key in the dictionary using an assignment statement. However an assignment to the entire dictionary (where only the name of the dictionary appears to the left of the assignment operator) only impacts the parameter variable. It does not modify the argument passed to the function. As with other types, dictionaries are returned from a function using the `return` keyword.

6.6 Exercises

While many of the exercises in this chapter can be solved with lists or `if` statements, most (or even all) of them have solutions that are well suited to dictionaries. As a result, you should use dictionaries to solve all of these exercises instead of (or in addition to) using the Python features that you have been introduced to previously.

Exercise 1: Reverse Lookup

Write a function named `reverseLookup` that finds all of the keys in a dictionary that map to a specific value. The function will take the dictionary and the value to search for as its only parameters. It will return a (possibly empty) list of keys from the dictionary that map to the provided value.

Include a main program that demonstrates the `reverseLookup` function as part of your solution to this exercise. Your program should create a dictionary and then show that the `reverseLookup` function works correctly when it returns multiple keys, a single key, and no keys. Ensure that your main program only runs when the file containing your solution to this exercise has not been imported into another program.

Exercise 2: Two Dice Simulation

In this exercise you will simulate 1,000 rolls of two dice. Begin by writing a function that simulates rolling a pair of six-sided dice. Your function will not take any parameters. It will return the total that was rolled on two dice as its only result.

Write a main program that uses your function to simulate rolling two six-sided dice 1,000 times. As your program runs, it should count the number of times that each total occurs. Then it should display a table that summarizes this data. Express the frequency for each total as a percentage of the number of rolls performed. Your program should also display the percentage expected by probability theory for each total. Sample output is shown below.

```
Total   Simulated   Expected
          Percent    Percent
    2        2.90       2.78
    3        6.90       5.56
    4        9.40       8.33
    5       11.90      11.11
    6       14.20      13.89
    7       14.20      16.67
    8       15.00      13.89
    9       10.50      11.11
   10        7.90       8.33
   11        4.50       5.56
   12        2.60       2.78
```

Exercise 3: Create a Bingo Card

A Bingo card consists of 5 columns of 5 numbers which are labelled with the letters B, I, N, G and O. There are 15 numbers that can appear under each letter. In particular, the numbers that can appear under the B range from 1 to 15, the numbers that can appear under the I range from 16 to 30, the numbers that can appear under the N range from 31 to 45, and so on.
Write a function that creates a random Bingo card and stores it in a dictionary. The keys will be the letters B, I, N, G and O. The values will be the lists of five numbers that appear under each letter. Write a second function that displays the Bingo card with the columns labelled appropriately. Use these functions to write a program that displays a random Bingo card. Ensure that the main program only runs when the file containing your solution has not been imported into another program.

Exercise 4: Unique Characters

Create a program that determines and displays the number of unique characters in a string entered by the user. For example, `Hello, World!` has 10 unique characters while `zzz` has only one unique character. Use a dictionary or set to solve this problem.

Exercise 5: Anagrams

Two words are anagrams if they contain all of the same letters, but in a different order. For example, “evil” and “live” are anagrams because each contains one “e”, one “i”, one “l”, and one “v”. Create a program that reads two strings from the user, determines whether or not they are anagrams, and reports the result.

Files and Exceptions

7

The programs that we have created so far have read all of their input from the keyboard. As a result, it has been necessary to re-type all of the input values each time the program runs. This is inefficient, particularly for programs that require a lot of input. Similarly, our programs have displayed all of their results on the screen. While this works well when only a few lines of output are printed, it is impractical for larger results that move off the screen too quickly to be read, or for output that requires further analysis by other programs. Writing programs that use files effectively will allow us to address all of these concerns.

Files are relatively permanent. The values stored in them are retained after a program completes and when the computer is turned off. This makes them suitable for storing results that are needed for an extended period of time, and for holding input values for a program that will be run several times. You have previously worked with files such as word processor documents, spreadsheets, images, and videos, among others. Your Python programs are also stored in files.

Files are commonly classified as being *text files* or *binary files*. Text files only contain sequences of bits that represent characters using an encoding system such as ASCII or UTF-8. These files can be viewed and modified with any text editor. All of the Python programs that we have created have been saved as text files.

Like text files, binary files also contain sequences of bits. But unlike text files, those sequences of bits can represent any kind of data. They are not restricted to characters alone. Files that contain image, sound and video data are normally binary files. We will restrict ourselves to working with text files in this book because they are easy to create and view with your favourite editor. Most of the principles described for text files can also be applied to binary files.

7.1 Opening a File

A file must be opened before data values can be read from it. It is also necessary to open a file before new data values are written to it. Files are opened by calling the `open` function.

The `open` function takes two arguments. The first argument is a string that contains the name of the file that will be opened. The second argument is also a string. It indicates the *access mode* for the file. The access modes that we will discuss include read (denoted by `"r"`), write (denoted by `"w"`) and append (denoted by `"a"`).

A *file object* is returned by the `open` function. As a result, the `open` function is normally called on the right side of an assignment statement, as shown below:

```
inf = open("input.txt", "r")
```

Once the file has been opened, methods can be applied to the file object to read data from the file. Similarly, data is written to the file by applying appropriate methods to the file object. These methods are described in the sections that follow. The file should be closed once all of the values have been read or written. This is accomplished by applying the `close` method to the file object.

7.2 Reading Input from a File

There are several methods that can be applied to a file object to read data from a file. These methods can only be applied when the file has been opened in read mode. Attempting to read from a file that has been opened in write mode or append mode will cause your program to crash.

The `readline` method reads one line from the file and returns it as a string, much like the input function reads a line of text typed on the keyboard. Each subsequent call to `readline` reads another line from the file sequentially from the top of the file to the bottom of the file. The `readline` method returns an empty string when there is no further data to read from the file.

Consider a data file that contains a long list of numbers, each of which appears on its own line. The following program computes the total of all of the numbers in such a file.

```
# Read the file name from the user and open the file
fname = input("Enter the file name: ")
inf = open(fname, "r")

# Initialize the total
total = 0

# Total the values in the file
line = inf.readline()
while line != "":
  total = total + float(line)
  line = inf.readline()
```

```
# Close the file
inf.close()

# Display the result
print("The total of the values in", fname, "is", total)
```

This program begins by reading the name of the file from the user. Once the name has been read, the file is opened for reading and the file object is stored in `inf`. Then `total` is initialized to 0, and the first line is read from the file.

The condition on the `while` loop is evaluated next. If the first line read from the file is non-empty, then the body of the loop executes. It converts the line read from the file into a floating-point number and adds it to `total`. Then the next line is read from the file. If the file contains more data then the `line` variable will contain the next line in the file, the `while` loop condition will evaluate to `True`, and the loop will execute again causing another value to be added to the total.

At some point all of the data will have been read from the file. When this occurs the `readline` method will return an empty string which will be stored into `line`. This will cause the condition on the `while` loop to evaluate to `False` and cause the loop to terminate. Then the program will go on and display the total.

Sometimes it is helpful to read all of the data from a file at once instead of reading it one line at a time. This can be accomplished using either the `read` method or the `readlines` method. The `read` method returns the entire contents of the file as one (potentially very long) string. Then further processing is typically performed to break the string into smaller pieces. The `readlines` method returns a list where each element is one line from the file. Once all of the lines are read with `readlines` a loop can be used to process each element in the list. The following program uses `readlines` to compute the sum of all of the numbers in a file. It reads all of the data from the file at once instead of adding each number to the total as it is read.

```
# Read the file name from the user and open the file
fname = input("Enter the file name: ")
inf = open(fname, "r")

# Initialize total and read all of the lines from the file
total = 0
lines = inf.readlines()

# Total the values in the file
for line in lines:
  total = total + float(line)

# Close the file
inf.close()

# Display the result
print("The total of the values in", fname, "is", total)
```

7.3 End of Line Characters

The following example uses the `readline` method to read and display all of the lines in a file. Each line is preceded by its line number and a colon when it is printed.

```
# Read the file name from the user and open the file
fname = input("Enter the name of a file to display: ")
inf = open(fname, "r")

# Initialize the line number
num = 1

# Display each line in the file, preceded by its line number
line = inf.readline()
while line != "":
  print("%d: %s" % (i, line))

  # Increment the line number and read the next line
  num = num + 1
  line = inf.readline()

# Close the file
inf.close()
```

When you run this program you might be surprised by its output. In particular, each time a line from the file is printed, a second line, which is blank, is printed immediately after it. This occurs because each line in a text file ends with one or more characters that denote the end of the line.[1] Such characters are needed so that any program reading the file can determine where one line ends and the next one begins. Without them, all of the characters in a text file would appear on the same line when they are read by your program (or when loaded into your favourite text editor).

The end of line marker can be removed from a string that was read from a file by calling the `rstrip` method. This method, which can be applied to any string, removes any whitespace characters (spaces, tabs, and end of line markers) from the right end of a string. A new copy of the string with such characters removed (if any were present) is returned by the method.

An updated version of the line numbering program is shown below. It uses the `rstrip` method to remove the end of line markers, and as a consequence, does not include the blank lines that were incorrectly displayed by the previous version.

```
# Read the file name from the user and open the file
fname = input("Enter the name of a file to display: ")
inf = open(fname, "r")

# Initialize the line number
num = 1

# Display each line in the file, preceded by its line number
line = inf.readline()
while line != "":
  # Remove the end of line marker and display the line preceded by its line number
  line = line.rstrip()
  print("%d: %s" % (i, line))
```

[1] The character or sequence of characters used to denote the end of a line in a text file varies from operating system to operating system. Fortunately, Python automatically handles these differences and allows text files created on any widely used operating system to be loaded by Python programs running on any other widely used operating system.

```
    # Increment the line number and read the next line
    num = num + 1
    line = inf.readline()

# Close the file
inf.close()
```

7.4 Writing Output to a File

When a file is opened in write mode, a new empty file is created. If the file already exists then the existing file is destroyed and any data that it contained is lost. Opening a file that already exists in append mode will cause any data written to the file to be added to the end of it. If a file opened in append mode does not exist then a new empty file is created.

The `write` method can be used to write data to a file opened in either write mode or append mode. It takes one argument, which must be a string, that will be written to the file. Values of other types can be converted to a string by calling the `str` function. Multiple values can be written to the file by concatenating all of the items into one longer string, or by calling the `write` method multiple times.

Unlike the `print` function, the `write` method does not automatically move to the next line after writing a value. As a result, one has to explicitly write an end of line marker to the file between values that are to reside on different lines. Python uses `\n` to denote the end of line marker. This pair of characters, referred to as an *escape sequence*, can appear in a string on its own, or `\n` can appear as part of a longer string.

The following program writes the numbers from 1 up to (and including) a number entered by the user to a file. String concatenation and the `\n` escape sequence are used so that each number is written on its own line.

```
# Read the file name from the user and open the file
fname = input("Where will the numbers will be stored? ")
outf = open(fname, "w")

# Read the maximum value that will be written
limit = int(input("What is the maximum value? "))

# Write the numbers to the file with one number on each line
for num in range(1, limit + 1):
    outf.write(str(num) + "\n")

# Close the file
outf.close()
```

7.5 Command Line Arguments

Computer programs are commonly executed by clicking on an icon or selecting an item from a menu. Programs can also be started from the command line by typing an appropriate command into a terminal or command prompt window. For example, on

many operating systems, the Python program stored in `test.py` can be executed by typing either `test.py` or `python test.py` in such a window.

Starting a program from the command line provides a new opportunity to supply input to it. Values that the program needs to perform its task can be part of the command used to start the program by including them on the command line after the name of the `.py` file. Being able to provide input as part of the command used to start a program is particularly beneficial when writing scripts that use multiple programs to automate some task, and for programs that are scheduled to run periodically.

Any command line arguments provided when the program was executed are stored into a variable named `argv` (argument vector) that resides in the `sys` (system) module. This variable is a list, and each element in the list is a string. Elements in the list can be converted to other types by calling the appropriate type conversion functions like `int` and `float`. The first element in the argument vector is the name of the Python source file that is being executed. The subsequent elements in the list are the values provided on the command line after the name of the Python file (if any).

The following program demonstrates accessing the argument vector. It begins by reporting the number of command line arguments provided to the program and the name of the source file that is being executed. Then it goes on and displays the arguments that appear after the name of the source file if such values were provided. Otherwise a message is displayed that indicates that there were no command line arguments beyond the `.py` file being executed.

```
# The system module must be imported to access the command line arguments
import sys

# Display the number of command line arguments (including the .py file)
print("The program has", len(sys.argv), \
      "command line argument(s).")

# Display the name of the .py file
print("The name of the .py file is", sys.argv[0])

# Determine whether or not there are additional arguments to display
if len(sys.argv) > 1:
  # Display all of the command line arguments beyond the name of the .py file
  print("The remaining arguments are:")
  for i in range(1, len(sys.argv)):
    print(" ", sys.argv[i])
else:
  print("No additional arguments were provided.")
```

Command line arguments can be used to supply any input values to the program that can be typed on the command line, such as integers, floating-point numbers and strings. These values can then be used just like any other values in the program. For example, the following lines of code are a revised version of our program that sums all of the numbers in a file. In this version of the program the name of the file is provided as a command line argument instead of being read from the keyboard.

```
# Import the system module
import sys

# Ensure that the program was started with one command line argument beyond the name
# of the .py file
if len(sys.argv) != 2:
  print("A file name must be provided as a command line", \
        "argument.")
  quit()

# Open the file listed immediately after the .py file on the command line
inf = open(sys.argv[1], "r")

# Initialize the total
total = 0

# Total the values in the file
line = inf.readline()
while line != "":
  total = total + float(line)
  line = inf.readline()

# Close the file
inf.close()

# Display the result
print("The total of the values in", sys.argv[1], "is", total)
```

7.6 Exceptions

There are many things that can go wrong when a program is running: The user can supply a non-numeric value when a numeric value was expected, the user can enter a value that causes the program to divide by 0, or the user can attempt to open a file that does not exist, among many other possibilities. All of these errors are *exceptions*. By default, a Python program crashes when an exception occurs. However, we can prevent our program from crashing by catching the exception and taking appropriate actions to recover from it.

The programmer must indicate where an exception might occur in order to catch it. He or she must also indicate what code to run to handle the exception when it occurs. These tasks are accomplished by using two keywords that we have not yet seen: `try` and `except`. Code that might cause an exception that we want to catch is placed inside a `try` block. The `try` block is immediately followed by one or more `except` blocks. When an exception occurs inside a `try` block, execution immediately jumps to the appropriate `except` block without running any remaining statements in the `try` block.

Each `except` block can specify the particular exception that it catches. This is accomplished by including the exception's type immediately after the `except` keyword. Such a block only executes when an exception of the indicated type occurs. An `except` block that does not specify a particular exception will catch any type

of exception (that is not caught by another `except` block associated to the same `try` block). The `except` blocks only execute when an exception occurs. If the `try` block executes without raising an exception then all of the `except` blocks are skipped and execution continues with the first line of code following the final `except` block.

The programs that we considered in the previous sections all crashed when the user provided the name of a file that did not exist. This crash occurred because a `FileNotFoundError` exception was raised without being caught. The following code segment uses a `try` block and an `except` block to catch this exception and display a meaningful error message when it occurs. This code segment can be followed by whatever additional code is needed to read and process the data in the file.

```
# Read the file name from the user
fname = input("Enter the file name: ")

# Attempt to open the file
try:
  inf = open(fname, "r")
except FileNotFoundError:
  # Display an error message and quit if the file was not opened successfully
  print("'%s' could not be opened.  Quitting...")
  quit()
```

The current version of our program quits when the file requested by the user does not exist. While that might be fine in some situations, there are other times when it is preferable to prompt the user to re-enter the file name. The second file name entered by the user could also cause an exception. As a result, a loop must be used that runs until the user enters the name of a file that is opened successfully. This is demonstrated by the following program. Notice that the `try` block and the `except` block are both inside the `while` loop.

```
# Read the file name from the user
fname = input("Enter the file name: ")

file_opened = False
while file_opened == False:
  # Attempt to open the file
  try:
    inf = open(fname, "r")
    file_opened = True
  except FileNotFoundError:
    # Display an error message and read another file name if the file was not
    # opened successfully
    print("'%s' wasn't found.  Please try again.")
    fname = input("Enter the file name: ")
```

When this program runs it begins by reading the name of a file from the user. Then the `file_opened` variable is set to `False` and the loop runs for the first time. Two lines of code reside in the `try` block inside the loop's body. The first attempts to open

the file specified by the user. If the file does not exist then a `FileNotFoundError` exception is raised and execution immediately jumps to the `except` block, skipping the second line in the `try` block. When the `except` block executes it displays an error message and reads another file name from the user.

Execution continues by returning to the top of the loop and evaluating its condition again. The condition still evaluates to `False` because the `file_opened` variable is still `False`. As a result, the body of the loop executes for a second time, and the program makes another attempt to open the file using the most recently entered file name. If that file does not exist then the program progresses as described in the previous paragraph. But if the file exists, the call to `open` completes successfully, and execution continues with the next line in the `try` block. This line sets `file_opened` to `True`. Then the `except` block is skipped because no exceptions were raised while executing the `try` block. Finally, the loop terminates because `file_opened` was set to `True`, and execution continues with the rest of the program.

The concepts introduced in this section can be used to detect and respond to a wide variety of errors that can occur as a program is running. By creating `try` and `except` blocks your programs can respond to these errors in an appropriate manner instead of crashing.

7.7 Exercises

Many of the exercises in this chapter read data from a file. In some cases any text file can be used as input. In other cases appropriate input files can be created easily in your favourite text editor. There are also some exercises that require specific data sets such as a list of words, names or chemical elements. These data sets can be downloaded from the author's website:

Exercise 1: Display the Head of a File

Unix-based operating systems usually include a tool named `head`. It displays the first 10 lines of a file whose name is provided as a command line argument. Write a Python program that provides the same behaviour. Display an appropriate error message if the file requested by the user does not exist, or if the command line argument is omitted.

Exercise 2: Display the Tail of a File

Unix-based operating systems also typically include a tool named `tail`. It displays the last 10 lines of a file whose name is provided as a command line argument. Write a Python program that provides the same behaviour. Display an appropriate error message if the file requested by the user does not exist, or if the command line argument is omitted.

There are several different approaches that can be taken to solve this problem. One option is to load the entire contents of the file into a list and then display its last 10 elements. Another option is to read the contents of the file twice, once to count the lines, and a second time to display its last 10 lines. However, both of these solutions are undesirable when working with large files. Another solution exists that only requires you to read the file once, and only requires you to store 10 lines from the file at one time. For an added challenge, develop such a solution.

Exercise 3: Concatenate Multiple Files

Unix-based operating systems typically include a tool named `cat`, which is short for concatenate. Its purpose is to display the concatenation of one or more files whose names are provided as command line arguments. The files are displayed in the same order that they appear on the command line.

Create a Python program that performs this task. It should generate an appropriate error message for any file that cannot be displayed, and then proceed to the next file. Display an appropriate error message if your program is started without any command line arguments.