Texts in Computer Science

Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises and Solutions

Second Edition

Springer

Texts in Computer Science

More information about this series at http://www.springer.com/series/3191

Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises and Solutions

Second Edition

Springer

Ben Stephenson
Department of Computer Science
University of Calgary
Calgary, AB, Canada

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-030-18872-6 ISBN 978-3-030-18873-3 (eBook)
https://doi.org/10.1007/978-3-030-18873-3

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my wife, Flora, for 16 fantastic years of marriage, and many more to come. To my sons, Jonathan and Andrew, who were both in a hurry to enter the world. I love you all.

Preface

I believe that computer programming is a skill that is best learned through hands-on experience. While it is valuable for you to read about programming in textbooks and watch teachers create programs at the front of classrooms, it is even more important for you to spend time-solving problems that allow you to put programming concepts into practice. With this in mind, the majority of the pages in this book are dedicated to exercises and their solutions while only a few pages are used to briefly introduce the concepts needed to complete them.

This book contains 186 exercises that span a variety of academic disciplines and everyday situations. They can be solved using only the material covered in most introductory Python programming courses. Each exercise that you complete will strengthen your understanding of the Python programming language and enhance your ability to tackle subsequent programming challenges. I also hope that the connections that these exercises make to other academic disciplines and everyday life will maintain your interest as you complete them.

Solutions to approximately half of the exercises are provided in the second half of this book. Most of the solutions include brief annotations that explain the technique used to solve the problem or highlight a specific point of Python syntax. You will find these annotations in shaded boxes, making it easy to distinguish them from the solution itself.

I hope that you will take the time to compare each of your solutions with mine, even when you arrive at your solution without encountering any problems. Performing this comparison may reveal a flaw in your program, or help you become more familiar with a technique that you could have used to solve the problem more easily. In some cases, it could also reveal that you have discovered a faster or easier way to solve the problem than I have. If you become stuck on an exercise, a quick peek at my solution may help you work through your problem and continue to make progress without requiring assistance from someone else. Finally, the solutions that I have provided demonstrate good programming style, including appropriate comments, meaningful variable names, and minimal use of magic numbers. I encourage you to use good programming style when creating your solutions so that they compute the correct result while also being clear, easy to understand, and amenable to being updated in the future.

Exercises that include a solution are clearly marked with (Solved) next to the exercise name. The length of the sample solution is also stated for every exercise in this book. While you shouldn't expect your solution length to match the sample solution length exactly, I hope that providing this information will prevent you from going too far astray before seeking assistance.

This book can be used in a variety of ways. Its concise introductions to major Python programming concepts, which are new in this edition, allow it to be used as the lone textbook for an introductory programming course. It can also be used to supplement another textbook that has a limited selection of exercises. A motivated individual could teach themselves to program in Python using only this book, though there are, perhaps, easier ways to learn the language because the concise introductions to each topic cover only their most important aspects, without examining every special case or unusual circumstance. No matter what other resources you use with this book, if any, reading the chapters, completing the exercises, and studying the provided solutions will enhance your programming ability.

Acknowledgements

I would like to thank Dr. Tom Jenkyns for reviewing this book as it was being created. His helpful comments and suggestions resulted in numerous refinements and corrections that improved the quality of this work.

Calgary, Canada Ben Stephenson
March 2019

Contents

Part I Exercises

1 Introduction to Programming . . . 3
1.1 Storing and Manipulating Values . . . 4
1.2 Calling Functions . . . 5
1.3 Comments . . . 8
1.4 Formatting Values . . . 8
1.5 Working with Strings . . . 11
1.6 Exercises . . . 12

2 Decision Making . . . 23
2.1 If Statements . . . 23
2.2 If-Else Statements . . . 24
2.3 If-Elif-Else Statements . . . 25
2.4 If-Elif Statements . . . 27
2.5 Nested If Statements . . . 27
2.6 Boolean Logic . . . 28
2.7 Exercises . . . 29

3 Repetition . . . 43
3.1 While Loops . . . 43
3.2 For Loops . . . 44
3.3 Nested Loops . . . 46
3.4 Exercises . . . 47

4 Functions . . . 59
4.1 Functions with Parameters . . . 60
4.2 Variables in Functions . . . 62
4.3 Return Values . . . 63
4.4 Importing Functions into Other Programs . . . 65
4.5 Exercises . . . 65

5 Lists . . . 75
5.1 Accessing Individual Elements . . . 75
5.2 Loops and Lists . . . 76

5.3 Additional List Operations 78
5.4 Lists as Return Values and Arguments 82
5.5 Exercises 83

6 Dictionaries 97
6.1 Accessing, Modifying and Adding Values 98
6.2 Removing a Key-Value Pair 99
6.3 Additional Dictionary Operations 99
6.4 Loops and Dictionaries 100
6.5 Dictionaries as Arguments and Return Values 101
6.6 Exercises 102

7 Files and Exceptions 109
7.1 Opening a File 110
7.2 Reading Input from a File 110
7.3 End of Line Characters 111
7.4 Writing Output to a File 113
7.5 Command Line Arguments 113
7.6 Exceptions 115
7.7 Exercises 117

8 Recursion 127
8.1 Summing Integers 127
8.2 Fibonacci Numbers 129
8.3 Counting Characters 131
8.4 Exercises 132

Part II Solutions

9 Solutions to the Introduction to Programming Exercises 143

10 Solutions to the Decision Making Exercises 151

11 Solutions to the Repetition Exercises 161

12 Solutions to the Function Exercises 169

13 Solutions to the List Exercises 183

14 Solutions to the Dictionary Exercises 193

15 Solutions to the File and Exception Exercises 199

16 Solutions to the Recursion Exercises 209

Index 215

Part I

Exercises

1 Introduction to Programming

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*.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_1

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:	`"%d" % x`
Result:	`"12"`
Explanation:	The value stored in `x` is formatted as a decimal (base 10) integer.
Code Segment:	`"%f" % y`
Result:	`"-2.75"`
Explanation:	The value stored in `y` is formatted as a floating-point number.
Code Segment:	`"%d and %f" % (x, y)`
Result:	`"12 and -2.75"`
Explanation:	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:	`"%.4f" % x`
Result:	`"12.0000"`
Explanation:	The value stored in `x` is formatted as a floating-point number with 4 digits to the right of the decimal point.
Code Segment:	`"%.1f" % y`
Result:	`"-2.8"`
Explanation:	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:	`"%10s" % z`
Result:	`"    Andrew"`
Explanation:	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:	`"%4s" % z`
Result:	`"Andrew"`
Explanation:	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:	`"%8i%8i" % (x, y)`
Result:	`"      12      -2"`
Explanation:	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

(*Solved, 9 Lines*)

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

(*9 Lines*)

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

(*Solved, 13 Lines*)

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

(*Solved, 15 Lines*)

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

(*Solved, 15 Lines*)

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

(*Solved, 17 Lines*)

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

(*Solved, 11 Lines*)

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

(*15 Lines*)

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

(*19 Lines*)

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

(*Solved, 22 Lines*)

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
- The result of $\log_{10} a$
- The result of a^b

> Hint: You will probably find the `log10` function in the `math` module helpful for computing the second last item in the list.

Exercise 11: Fuel Efficiency

(*13 Lines*)

In the United States, fuel efficiency for vehicles is normally expressed in miles-per-gallon (MPG). In Canada, fuel efficiency is normally expressed in liters-per-hundred kilometers (L/100 km). Use your research skills to determine how to convert from MPG to L/100 km. Then create a program that reads a value from the user in American units and displays the equivalent fuel efficiency in Canadian units.

Exercise 12: Distance Between Two Points on Earth

(*27 Lines*)

The surface of the Earth is curved, and the distance between degrees of longitude varies with latitude. As a result, finding the distance between two points on the surface of the Earth is more complicated than simply using the Pythagorean theorem.

Let (t_1, g_1) and (t_2, g_2) be the latitude and longitude of two points on the Earth's surface. The distance between these points, following the surface of the Earth, in kilometers is:

$$\text{distance} = 6371.01 \times \arccos(\sin(t_1) \times \sin(t_2) + \cos(t_1) \times \cos(t_2) \times \cos(g_1 - g_2))$$

> The value 6371.01 in the previous equation wasn't selected at random. It is the average radius of the Earth in kilometers.

Create a program that allows the user to enter the latitude and longitude of two points on the Earth in degrees. Your program should display the distance between the points, following the surface of the earth, in kilometers.

> Hint: Python's trigonometric functions operate in radians. As a result, you will need to convert the user's input from degrees to radians before computing the distance with the formula discussed previously. The `math` module contains a function named `radians` which converts from degrees to radians.

Exercise 13: Making Change

(*Solved, 35 Lines*)

Consider the software that runs on a self-checkout machine. One task that it must be able to perform is to determine how much change to provide when the shopper pays for a purchase with cash.

Write a program that begins by reading a number of cents from the user as an integer. Then your program should compute and display the denominations of the coins that should be used to give that amount of change to the shopper. The change should be given using as few coins as possible. Assume that the machine is loaded with pennies, nickels, dimes, quarters, loonies and toonies.

> A one dollar coin was introduced in Canada in 1987. It is referred to as a loonie because one side of the coin has a loon (a type of bird) on it. The two dollar coin, referred to as a toonie, was introduced 9 years later. It's name is derived from the combination of the number two and the name of the loonie.

Exercise 14: Height Units

(*Solved, 16 Lines*)

Many people think about their height in feet and inches, even in some countries that primarily use the metric system. Write a program that reads a number of feet from the user, followed by a number of inches. Once these values are read, your program should compute and display the equivalent number of centimeters.

> Hint: One foot is 12 inches. One inch is 2.54 centimeters.

Exercise 15: Distance Units

(*20 Lines*)

In this exercise, you will create a program that begins by reading a measurement in feet from the user. Then your program should display the equivalent distance in inches, yards and miles. Use the Internet to look up the necessary conversion factors if you don't have them memorized.

Exercise 16: Area and Volume

(*15 Lines*)

Write a program that begins by reading a radius, r, from the user. The program will continue by computing and displaying the area of a circle with radius r and the volume of a sphere with radius r. Use the `pi` constant in the `math` module in your calculations.

> Hint: The area of a circle is computed using the formula $area = \pi r^2$. The volume of a sphere is computed using the formula $volume = \frac{4}{3}\pi r^3$.

Exercise 17: Heat Capacity

(*Solved, 23 Lines*)

The amount of energy required to increase the temperature of one gram of a material by one degree Celsius is the material's specific heat capacity, C. The total amount of energy, q, required to raise m grams of a material by ΔT degrees Celsius can be computed using the formula:

$$q = mC\Delta T$$

Write a program that reads the mass of some water and the temperature change from the user. Your program should display the total amount of energy that must be added or removed to achieve the desired temperature change.

Hint: The specific heat capacity of water is $4.186 \frac{J}{g^\circ C}$. Because water has a density of 1.0 grams per milliliter, you can use grams and milliliters interchangeably in this exercise.

Extend your program so that it also computes the cost of heating the water. Electricity is normally billed using units of kilowatt hours rather than Joules. In this exercise, you should assume that electricity costs 8.9 cents per kilowatt hour. Use your program to compute the cost of boiling the water needed for a cup of coffee.

Hint: You will need to look up the factor for converting between Joules and kilowatt hours to complete the last part of this exercise.

Exercise 18: Volume of a Cylinder

(*15 Lines*)

The volume of a cylinder can be computed by multiplying the area of its circular base by its height. Write a program that reads the radius of the cylinder, along with its height, from the user and computes its volume. Display the result rounded to one decimal place.

Exercise 19: Free Fall

(*Solved, 15 Lines*)

Create a program that determines how quickly an object is travelling when it hits the ground. The user will enter the height from which the object is dropped in meters (m). Because the object is dropped its initial speed is 0 m/s. Assume that the acceleration due to gravity is $9.8\,\text{m/s}^2$. You can use the formula $v_f = \sqrt{v_i^2 + 2ad}$ to compute the final speed, v_f, when the initial speed, v_i, acceleration, a, and distance, d, are known.

Exercise 20: Ideal Gas Law

(*19 Lines*)

The ideal gas law is a mathematical approximation of the behavior of gasses as pressure, volume and temperature change. It is usually stated as:

$$PV = nRT$$

where P is the pressure in Pascals, V is the volume in liters, n is the amount of substance in moles, R is the ideal gas constant, equal to $8.314 \frac{\text{J}}{\text{mol K}}$, and T is the temperature in degrees Kelvin.

Write a program that computes the amount of gas in moles when the user supplies the pressure, volume and temperature. Test your program by determining the number of moles of gas in a SCUBA tank. A typical SCUBA tank holds 12 liters of gas at a pressure of 20,000,000 Pascals (approximately 3,000 PSI). Room temperature is approximately 20 degrees Celsius or 68 degrees Fahrenheit.

> Hint: A temperature is converted from Celsius to Kelvin by adding 273.15 to it. To convert a temperature from Fahrenheit to Kelvin, deduct 32 from it, multiply it by $\frac{5}{9}$ and then add 273.15 to it.

Exercise 21: Area of a Triangle

(*13 Lines*)

The area of a triangle can be computed using the following formula, where b is the length of the base of the triangle, and h is its height:

$$\text{area} = \frac{b \times h}{2}$$

Write a program that allows the user to enter values for b and h. The program should then compute and display the area of a triangle with base length b and height h.

Exercise 22: Area of a Triangle (Again)

(*16 Lines*)

In the previous exercise you created a program that computed the area of a triangle when the length of its base and its height were known. It is also possible to compute the area of a triangle when the lengths of all three sides are known. Let s_1, s_2 and s_3 be the lengths of the sides. Let $s = (s_1 + s_2 + s_3)/2$. Then the area of the triangle can be calculated using the following formula:

$$\text{area} = \sqrt{s \times (s - s_1) \times (s - s_2) \times (s - s_3)}$$

Develop a program that reads the lengths of the sides of a triangle from the user and displays its area.

Exercise 23: Area of a Regular Polygon

(*Solved, 14 Lines*)

A polygon is regular if its sides are all the same length and the angles between all of the adjacent sides are equal. The area of a regular polygon can be computed using the following formula, where s is the length of a side and n is the number of sides:

$$\text{area} = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}$$

Write a program that reads s and n from the user and then displays the area of a regular polygon constructed from these values.

Exercise 24: Units of Time

(*22 Lines*)

Create a program that reads a duration from the user as a number of days, hours, minutes, and seconds. Compute and display the total number of seconds represented by this duration.

Exercise 25: Units of Time (Again)

(*Solved, 24 Lines*)

In this exercise you will reverse the process described in Exercise 24. Develop a program that begins by reading a number of seconds from the user. Then your program should display the equivalent amount of time in the form D:HH:MM:SS, where D, HH, MM, and SS represent days, hours, minutes and seconds respectively. The hours, minutes and seconds should all be formatted so that they occupy exactly two digits. Use your research skills determine what additional character needs to be included in the format specifier so that leading zeros are used instead of leading spaces when a number is formatted to a particular width.

Exercise 26: Current Time

(*10 Lines*)

Python's `time` module includes several time-related functions. One of these is the `asctime` function which reads the current time from the computer's internal clock and returns it in a human-readable format. Use this function to write a program that displays the current time and date. Your program will not require any input from the user.

Exercise 27: When is Easter?

(*33 Lines*)

Easter is celebrated on the Sunday immediately after the first full moon following the spring equinox. Because its date includes a lunar component, Easter does not have a fixed date in the Gregorian calendar. Instead, it can occur on any date between

March 22 and April 25. The month and day for Easter can be computed for a given year using the Anonymous Gregorian Computus algorithm, which is shown below.

Set a equal to the remainder when $year$ is divided by 19
Set b equal to the floor of $year$ divided by 100
Set c equal to the remainder when $year$ is divided by 100
Set d equal to the floor of b divided by 4
Set e equal to the remainder when b is divided by 4
Set f equal to the floor of $\frac{b+8}{25}$
Set g equal to the floor of $\frac{b-f+1}{3}$
Set h equal to the remainder when $19a+b-d-g+15$ is divided by 30
Set i equal to the floor of c divided by 4
Set k equal to the remainder when c is divided by 4
Set l equal to the remainder when $32+2e+2i-h-k$ is divided by 7
Set m equal to the floor of $\frac{a+11h+22l}{451}$

Set month equal to the floor of $\frac{h+l-7m+114}{31}$
Set day equal to one plus the remainder when $h+l-7m+114$ is divided by 31

Write a program that implements the Anonymous Gregorian Computus algorithm to compute the date of Easter. Your program should read the year from the user and then display a appropriate message that includes the date of Easter in that year.

Exercise 28: Body Mass Index

(*14 Lines*)

Write a program that computes the body mass index (BMI) of an individual. Your program should begin by reading a height and weight from the user. Then it should use one of the following two formulas to compute the BMI before displaying it. If you read the height in inches and the weight in pounds then body mass index is computed using the following formula:

$$\text{BMI} = \frac{\text{weight}}{\text{height} \times \text{height}} \times 703$$

If you read the height in meters and the weight in kilograms then body mass index is computed using this slightly simpler formula:

$$\text{BMI} = \frac{\text{weight}}{\text{height} \times \text{height}}$$

Exercise 29: Wind Chill

(*Solved, 22 Lines*)

When the wind blows in cold weather, the air feels even colder than it actually is because the movement of the air increases the rate of cooling for warm objects, like people. This effect is known as wind chill.

In 2001, Canada, the United Kingdom and the United States adopted the following formula for computing the wind chill index. Within the formula T_a is the air temperature in degrees Celsius and V is the wind speed in kilometers per hour. A similar formula with different constant values can be used for temperatures in degrees Fahrenheit and wind speeds in miles per hour.

$$WCI = 13.12 + 0.6215T_a - 11.37V^{0.16} + 0.3965T_a V^{0.16}$$

Write a program that begins by reading the air temperature and wind speed from the user. Once these values have been read your program should display the wind chill index rounded to the closest integer.

> The wind chill index is only considered valid for temperatures less than or equal to 10 degrees Celsius and wind speeds exceeding 4.8 kilometers per hour.

Exercise 30: Celsius to Fahrenheit and Kelvin

(*17 Lines*)

Write a program that begins by reading a temperature from the user in degrees Celsius. Then your program should display the equivalent temperature in degrees Fahrenheit and degrees Kelvin. The calculations needed to convert between different units of temperature can be found on the Internet.

Exercise 31: Units of Pressure

(*20 Lines*)

In this exercise you will create a program that reads a pressure from the user in kilopascals. Once the pressure has been read your program should report the equivalent pressure in pounds per square inch, millimeters of mercury and atmospheres. Use your research skills to determine the conversion factors between these units.

Exercise 32: Sum of the Digits in an Integer

(*18 Lines*)

Develop a program that reads a four-digit integer from the user and displays the sum of its digits. For example, if the user enters 3141 then your program should display $3+1+4+1=9$.

Exercise 33: Sort 3 Integers

(*Solved, 19 Lines*)

Create a program that reads three integers from the user and displays them in sorted order (from smallest to largest). Use the `min` and `max` functions to find the smallest and largest values. The middle value can be found by computing the sum of all three values, and then subtracting the minimum value and the maximum value.

Exercise 34: Day Old Bread

(*Solved, 19 Lines*)

A bakery sells loaves of bread for $3.49 each. Day old bread is discounted by 60 percent. Write a program that begins by reading the number of loaves of day old bread being purchased from the user. Then your program should display the regular price for the bread, the discount because it is a day old, and the total price. Each of these amounts should be displayed on its own line with an appropriate label. All of the values should be displayed using two decimal places, and the decimal points in all of the numbers should be aligned when reasonable values are entered by the user.

2 Decision Making

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 35: Even or Odd?

(*Solved, 13 Lines*)

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 36: Dog Years

(*22 Lines*)

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 37: Vowel or Consonant

(*Solved, 16 Lines*)

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 38: Name That Shape

(*Solved, 31 Lines*)

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 39: Month Name to Number of Days

(*Solved, 18 Lines*)

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 40: Sound Levels

(*30 Lines*)

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 41: Classifying Triangles

(*Solved, 21 Lines*)

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.

Exercise 42: Note to Frequency

(*Solved, 39 Lines*)

The following table lists an octave of music notes, beginning with middle C, along with their frequencies.

Note	Frequency (Hz)
C4	261.63
D4	293.66
E4	329.63
F4	349.23
G4	392.00
A4	440.00
B4	493.88

Begin by writing a program that reads the name of a note from the user and displays the note's frequency. Your program should support all of the notes listed previously.

Once you have your program working correctly for the notes listed previously you should add support for all of the notes from C0 to C8. While this could be done by adding many additional cases to your `if` statement, such a solution is cumbersome, inelegant and unacceptable for the purposes of this exercise. Instead, you should exploit the relationship between notes in adjacent octaves. In particular, the frequency of any note in octave n is half the frequency of the corresponding note in octave $n + 1$. By using this relationship, you should be able to add support for the additional notes without adding additional cases to your `if` statement.

> Hint: You will want to access the characters in the note entered by the user individually when completing this exercise. Begin by separating the letter from the octave. Then compute the frequency for that letter in the fourth octave using the data in the table above. Once you have this frequency you should divide it by 2^{4-x}, where x is the octave number entered by the user. This will halve or double the frequency the correct number of times.

Exercise 43: Frequency to Note

(*Solved, 42 Lines*)

In the previous question you converted from a note's name to its frequency. In this question you will write a program that reverses that process. Begin by reading a frequency from the user. If the frequency is within one Hertz of a value listed in the table in the previous question then report the name of the corresponding note. Otherwise report that the frequency does not correspond to a known note. In this exercise you only need to consider the notes listed in the table. There is no need to consider notes from other octaves.

Exercise 44: Faces on Money

(*31 Lines*)

It is common for images of a country's previous leaders, or other individuals of historical significance, to appear on its money. The individuals that appear on banknotes in the United States are listed in below.

Individual	Amount
George Washington	\$1
Thomas Jefferson	\$2
Abraham Lincoln	\$5
Alexander Hamilton	\$10
Andrew Jackson	\$20
Ulysses S. Grant	\$50
Benjamin Franklin	\$100

Write a program that begins by reading the denomination of a banknote from the user. Then your program should display the name of the individual that appears on the banknote of the entered amount. An appropriate error message should be displayed if no such note exists.

> While two dollar banknotes are rarely seen in circulation in the United States, they are legal tender that can be spent just like any other denomination. The United States has also issued banknotes in denominations of \$500, \$1,000, \$5,000, and \$10,000 for public use. However, high denomination banknotes have not been printed since 1945 and were officially discontinued in 1969. As a result, we will not consider them in this exercise.

Exercise 45: Date to Holiday Name

(*18 Lines*)

Canada has three national holidays which fall on the same dates each year.

Holiday	Date
New Year's Day	January 1
Canada Day	July 1
Christmas Day	December 25

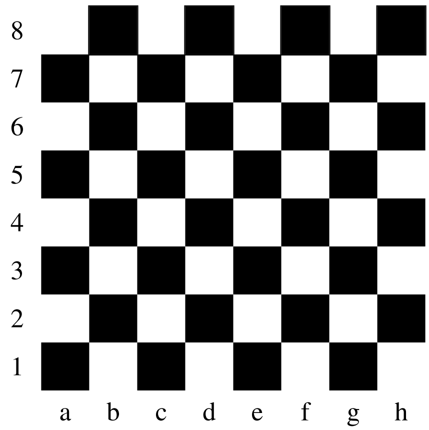
Write a program that reads a month and day from the user. If the month and day match one of the holidays listed previously then your program should display the holiday's name. Otherwise your program should indicate that the entered month and day do not correspond to a fixed-date holiday.

Canada has two additional national holidays, Good Friday and Labour Day, whose dates vary from year to year. There are also numerous provincial and territorial holidays, some of which have fixed dates, and some of which have variable dates. We will not consider any of these additional holidays in this exercise.

Exercise 46: What Color Is That Square?

(*22 Lines*)

Positions on a chess board are identified by a letter and a number. The letter identifies the column, while the number identifies the row, as shown below:



Write a program that reads a position from the user. Use an if statement to determine if the column begins with a black square or a white square. Then use modular arithmetic to report the color of the square in that row. For example, if the user enters `a1` then your program should report that the square is black. If the user enters `d5` then your program should report that the square is white. Your program may assume that a valid position will always be entered. It does not need to perform any error checking.

Exercise 47: Season from Month and Day

(*Solved, 43 Lines*)

The year is divided into four seasons: spring, summer, fall (or autumn) and winter. While the exact dates that the seasons change vary a little bit from year to year because of the way that the calendar is constructed, we will use the following dates for this exercise:

Season	First Day
Spring	March 20
Summer	June 21
Fall	September 22
Winter	December 21

Create a program that reads a month and day from the user. The user will enter the name of the month as a string, followed by the day within the month as an integer. Then your program should display the season associated with the date that was entered.

Exercise 48: Birth Date to Astrological Sign

(*47 Lines*)

The horoscopes commonly reported in newspapers use the position of the sun at the time of one's birth to try and predict the future. This system of astrology divides the year into twelve zodiac signs, as outline in the table below:

Zodiac Sign	Date Range
Capricorn	December 22 to January 19
Aquarius	January 20 to February 18
Pisces	February 19 to March 20
Aries	March 21 to April 19
Taurus	April 20 to May 20
Gemini	May 21 to June 20
Cancer	June 21 to July 22
Leo	July 23 to August 22
Virgo	August 23 to September 22
Libra	September 23 to October 22
Scorpio	October 23 to November 21
Sagittarius	November 22 to December 21

Write a program that asks the user to enter his or her month and day of birth. Then your program should report the user's zodiac sign as part of an appropriate output message.

Exercise 49: Chinese Zodiac

(*Solved, 40 Lines*)

The Chinese zodiac assigns animals to years in a 12 year cycle. One 12 year cycle is shown in the table below. The pattern repeats from there, with 2012 being another year of the dragon, and 1999 being another year of the hare.

Year	Animal
2000	Dragon
2001	Snake
2002	Horse
2003	Sheep
2004	Monkey
2005	Rooster
2006	Dog
2007	Pig
2008	Rat
2009	Ox
2010	Tiger
2011	Hare

Write a program that reads a year from the user and displays the animal associated with that year. Your program should work correctly for any year greater than or equal to zero, not just the ones listed in the table.

Exercise 50: Richter Scale

(*30 Lines*)

The following table contains earthquake magnitude ranges on the Richter scale and their descriptors:

Magnitude	Descriptor
Less than 2.0	Micro
2.0 to less than 3.0	Very Minor
3.0 to less than 4.0	Minor
4.0 to less than 5.0	Light
5.0 to less than 6.0	Moderate
6.0 to less than 7.0	Strong
7.0 to less than 8.0	Major
8.0 to less than 10.0	Great
10.0 or more	Meteoric

Write a program that reads a magnitude from the user and displays the appropriate descriptor as part of a meaningful message. For example, if the user enters 5.5 then your program should indicate that a magnitude 5.5 earthquake is considered to be a moderate earthquake.

Exercise 51: Roots of a Quadratic Function

(*24 Lines*)

A univariate quadratic function has the form $f(x) = ax^2 + bx + c$, where a, b and c are constants, and a is non-zero. Its roots can be identified by finding the values of x that satisfy the quadratic equation $ax^2 + bx + c = 0$. These values can be computed using the quadratic formula, shown below. A quadratic function may have 0, 1 or 2 real roots.

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

The portion of the expression under the square root sign is called the discriminant. If the discriminant is negative then the quadratic equation does not have any real roots. If the discriminant is 0, then the equation has one real root. Otherwise the equation has two real roots, and the expression must be evaluated twice, once using a plus sign, and once using a minus sign, when computing the numerator.

Write a program that computes the real roots of a quadratic function. Your program should begin by prompting the user for the values of a, b and c. Then it should display a message indicating the number of real roots, along with the values of the real roots (if any).

Exercise 52: Letter Grade to Grade Points

(*Solved, 52 Lines*)

At a particular university, letter grades are mapped to grade points in the following manner:

Letter	Grade Points
A+	4.0
A	4.0
A-	3.7
B+	3.3
B	3.0
B-	2.7
C+	2.3
C	2.0
C-	1.7
D+	1.3
D	1.0
F	0

Write a program that begins by reading a letter grade from the user. Then your program should compute and display the equivalent number of grade points. Ensure that your program generates an appropriate error message if the user enters an invalid letter grade.

Exercise 53: Grade Points to Letter Grade

(*47 Lines*)

In the previous exercise you created a program that converted a letter grade into the equivalent number of grade points. In this exercise you will create a program that reverses the process and converts from a grade point value entered by the user to a letter grade. Ensure that your program handles grade point values that fall between letter grades. These should be rounded to the closest letter grade. Your program should report A+ if the value entered by the user is 4.0 or more.

Exercise 54: Assessing Employees

(*Solved, 30 Lines*)

At a particular company, employees are rated at the end of each year. The rating scale begins at 0.0, with higher values indicating better performance and resulting in larger raises. The value awarded to an employee is either 0.0, 0.4, or 0.6 or more. Values between 0.0 and 0.4, and between 0.4 and 0.6 are never used. The meaning associated with each rating is shown in the following table. The amount of an employee's raise is $2,400.00 multiplied by their rating.

Rating	Meaning
0.0	Unacceptable Performance
0.4	Acceptable Performance
0.6 or more	Meritorious Performance

Write a program that reads a rating from the user and indicates whether the performance for that rating is unacceptable, acceptable or meritorious. The amount of the employee's raise should also be reported. Your program should display an appropriate error message if an invalid rating is entered.

Exercise 55: Wavelengths of Visible Light

(*38 Lines*)

The wavelength of visible light ranges from 380 to 750 nanometers (nm). While the spectrum is continuous, it is often divided into 6 colors as shown below:

Color	Wavelength (nm)
Violet	380 to less than 450
Blue	450 to less than 495
Green	495 to less than 570
Yellow	570 to less than 590
Orange	590 to less than 620
Red	620 to 750

Write a program that reads a wavelength from the user and reports its color. Display an appropriate error message if the wavelength entered by the user is outside of the visible spectrum.

Exercise 56: Frequency to Name

(*31 Lines*)

Electromagnetic radiation can be classified into one of 7 categories according to its frequency, as shown in the table below:

Name	Frequency Range (Hz)
Radio Waves	Less than 3×10^9
Microwaves	3×10^9 to less than 3×10^{12}
Infrared Light	3×10^{12} to less than 4.3×10^{14}
Visible Light	4.3×10^{14} to less than 7.5×10^{14}
Ultraviolet Light	7.5×10^{14} to less than 3×10^{17}
X-Rays	3×10^{17} to less than 3×10^{19}
Gamma Rays	3×10^{19} or more

Write a program that reads the frequency of some radiation from the user and displays name of the radiation as part of an appropriate message.

Exercise 57: Cell Phone Bill

(*44 Lines*)

A particular cell phone plan includes 50 minutes of air time and 50 text messages for $15.00 a month. Each additional minute of air time costs $0.25, while additional text messages cost $0.15 each. All cell phone bills include an additional charge of $0.44 to support 911 call centers, and the entire bill (including the 911 charge) is subject to 5 percent sales tax.

Write a program that reads the number of minutes and text messages used in a month from the user. Display the base charge, additional minutes charge (if any), additional text message charge (if any), the 911 fee, tax and total bill amount. Only display the additional minute and text message charges if the user incurred costs in these categories. Ensure that all of the charges are displayed using 2 decimal places.

Exercise 58: Is It a Leap Year?

(*Solved, 22 Lines*)

Most years have 365 days. However, the time required for the Earth to orbit the Sun is actually slightly more than that. As a result, an extra day, February 29, is included in some years to correct for this difference. Such years are referred to as leap years. The rules for determining whether or not a year is a leap year follow:

- Any year that is divisible by 400 is a leap year.
- Of the remaining years, any year that is divisible by 100 is **not** a leap year.
- Of the remaining years, any year that is divisible by 4 is a leap year.
- All other years are **not** leap years.

Write a program that reads a year from the user and displays a message indicating whether or not it is a leap year.

Exercise 59: Next Day

(*50 Lines*)

Write a program that reads a date from the user and computes its immediate successor. For example, if the user enters values that represent 2019-11-18 then your program should display a message indicating that the day immediately after 2019-11-18 is 2019-11-19. If the user enters values that represent 2019-11-30 then the program should indicate that the next day is 2019-12-01. If the user enters values that represent 2019-12-31 then the program should indicate that the next day is 2020-01-01. The date will be entered in numeric form with three separate input statements; one for the year, one for the month, and one for the day. Ensure that your program works correctly for leap years.

Exercise 60: What Day of the Week Is January 1?

(*32 Lines*)

The following formula can be used to determine the day of the week for January 1 in a given year:

$$\begin{aligned}\text{day\_of\_the\_week} = {} & (\text{year} + \text{floor}((\text{year} - 1) / 4) - \text{floor}((\text{year} - 1) / 100) + \\ & \text{floor}((\text{year} - 1) / 400))\ \%\ 7\end{aligned}$$

The result calculated by this formula is an integer that represents the day of the week. Sunday is represented by 0. The remaining days of the week following in sequence through to Saturday, which is represented by 6.

Use the formula above to write a program that reads a year from the user and reports the day of the week for January 1 of that year. The output from your program should include the full name of the day of the week, not just the integer returned by the formula.

Exercise 61: Is a License Plate Valid?

(*Solved, 28 Lines*)

In a particular jurisdiction, older license plates consist of three uppercase letters followed by three digits. When all of the license plates following that pattern had been used, the format was changed to four digits followed by three uppercase letters.

Write a program that begins by reading a string of characters from the user. Then your program should display a message indicating whether the characters are valid for an older style license plate or a newer style license plate. Your program should display an appropriate message if the string entered by the user is not valid for either style of license plate.

Exercise 62: Roulette Payouts

(*Solved, 45 Lines*)

A roulette wheel has 38 spaces on it. Of these spaces, 18 are black, 18 are red, and two are green. The green spaces are numbered 0 and 00. The red spaces are numbered 1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30 32, 34 and 36. The remaining integers between 1 and 36 are used to number the black spaces.

Many different bets can be placed in roulette. We will only consider the following subset of them in this exercise:

- Single number (1 to 36, 0, or 00)
- Red versus Black
- Odd versus Even (Note that 0 and 00 do **not** pay out for even)
- 1 to 18 versus 19 to 36

Write a program that simulates a spin of a roulette wheel by using Python's random number generator. Display the number that was selected and all of the bets that must be payed. For example, if 13 is selected then your program should display:

```
The spin resulted in 13...
Pay 13
Pay Black
Pay Odd
Pay 1 to 18
```

If the simulation results in 0 or 00 then your program should display `Pay 0` or `Pay 00` without any further output.

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,

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_3

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 63: Average

(*26 Lines*)

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 64: Discount Table

(*18 Lines*)

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 65: Temperature Conversion Table

(*22 Lines*)

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 66: No More Pennies

(*Solved, 39 Lines*)

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 67: Compute the Perimeter of a Polygon

(*Solved, 42 Lines*)

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 68: Compute a Grade Point Average

(*62 Lines*)

Exercise 52 includes a table that shows the conversion from letter grades to grade points at a particular academic institution. In this exercise you will compute the grade point average of an arbitrary number of letter grades entered by the user. The user will enter a blank line to indicate that all of the grades have been provided. For example, if the user enters `A`, followed by `C+`, followed by `B`, followed by a blank line then your program should report a grade point average of 3.1.

You may find your solution to Exercise 52 helpful when completing this exercise. Your program does not need to do any error checking. It can assume that each value entered by the user will always be a valid letter grade or a blank line.

Exercise 69: Admission Price

(*Solved, 38 Lines*)

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 70: Parity Bits

(*Solved, 27 Lines*)

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.

Exercise 71: Approximate π

(*23 Lines*)

The value of π can be approximated by the following infinite series:

$$\pi \approx 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \frac{4}{10 \times 11 \times 12} - \cdots$$

Write a program that displays 15 approximations of π. The first approximation should make use of only the first term from the infinite series. Each additional approximation displayed by your program should include one more term in the series, making it a better approximation of π than any of the approximations displayed previously.

Exercise 72: Fizz-Buzz

(*17 Lines*)

Fizz-Buzz is a game that is sometimes played by children to help them learn about division. The players are commonly arranged in a circle so that the game can progress from player to player continually. The starting player begins by saying one, and then play passes to the player to the left. Each subsequent player is responsible for the next integer in sequence before play passes to the following player. On a player's turn they must either say their number or one of following substitutions:

- If the player's number is divisible by 3 then the player says fizz instead of their number.
- If the player's number is divisible by 5 then the player says buzz instead of their number.

A player must say both fizz and buzz for numbers that are divisible by both 3 and 5. Any player that fails to perform the correct substitution or hesitates before answering is eliminated from the game. The last player remaining is the winner.

Write a program that displays the answers for the first 100 numbers in the Fizz-Buzz game. Each answer should be displayed on its own line.

Exercise 73: Caesar Cipher

(*Solved, 35 Lines*)

One of the first known examples of encryption was used by Julius Caesar. Caesar needed to provide written instructions to his generals, but he didn't want his enemies to learn his plans if the message slipped into their hands. As a result, he developed what later became known as the Caesar cipher.

The idea behind this cipher is simple (and as such, it provides no protection against modern code breaking techniques). Each letter in the original message is shifted by 3 places. As a result, A becomes D, B becomes E, C becomes F, D becomes G, etc.

The last three letters in the alphabet are wrapped around to the beginning: X becomes A, Y becomes B and Z becomes C. Non-letter characters are not modified by the cipher.

Write a program that implements a Caesar cipher. Allow the user to supply the message and the shift amount, and then display the shifted message. Ensure that your program encodes both uppercase and lowercase letters. Your program should also support negative shift values so that it can be used both to encode messages and decode messages.

Exercise 74: Square Root

(*14 Lines*)

Write a program that implements Newton's method to compute and display the square root of a number, x, entered by the user. The algorithm for Newton's method follows:

Read x from the user
Initialize $guess$ to $x/2$
While $guess$ is not good enough **do**
    Update $guess$ to be the average of $guess$ and $x/guess$

When this algorithm completes, *guess* contains an approximation of the square root of x. The quality of the approximation depends on how you define "good enough". In the author's solution, *guess* was considered good enough when the absolute value of the difference between $guess * guess$ and x was less than or equal to 10^{-12}.

Exercise 75: Is a String a Palindrome?

(*Solved, 26 Lines*)

A string is a palindrome if it is identical forward and backward. For example "anna", "civic", "level" and "hannah" are all examples of palindromic words. Write a program that reads a string from the user and uses a loop to determine whether or not it is a palindrome. Display the result, including a meaningful output message.

> Aibohphobia is the extreme or irrational fear of palindromes. This word was constructed by prepending the -phobia suffix with it's reverse, resulting in a palindrome. Ailihphilia is the fondness for or love of palindromes. It was constructed in the same manner from the -philia suffix.

Exercise 76: Multiple Word Palindromes

(*35 Lines*)

There are numerous phrases that are palindromes when spacing is ignored. Examples include “go dog”, “flee to me remote elf” and “some men interpret nine memos”, among many others. Extend your solution to Exercise 75 so that it ignores spacing while determining whether or not a string is a palindrome. For an additional challenge, further extend your solution so that is also ignores punctuation marks and treats uppercase and lowercase letters as equivalent.

Exercise 77: Multiplication Table

(*Solved, 18 Lines*)

In this exercise you will create a program that displays a multiplication table that shows the products of all combinations of integers from 1 times 1 up to and including 10 times 10. Your multiplication table should include a row of labels across the top of it containing the numbers 1 through 10. It should also include labels down the left side consisting of the numbers 1 through 10. The expected output from the program is shown below:

```
       1    2    3    4    5    6    7    8    9   10
  1    1    2    3    4    5    6    7    8    9   10
  2    2    4    6    8   10   12   14   16   18   20
  3    3    6    9   12   15   18   21   24   27   30
  4    4    8   12   16   20   24   28   32   36   40
  5    5   10   15   20   25   30   35   40   45   50
  6    6   12   18   24   30   36   42   48   54   60
  7    7   14   21   28   35   42   49   56   63   70
  8    8   16   24   32   40   48   56   64   72   80
  9    9   18   27   36   45   54   63   72   81   90
 10   10   20   30   40   50   60   70   80   90  100
```

When completing this exercise you will probably find it helpful to be able to print out a value without moving down to the next line. This can be accomplished by added `end=""` as the last argument to your `print` statement. For example, `print("A")` will display the letter `A` and then move down to the next line. The statement `print("A", end="")` will display the letter `A` without moving down to the next line, causing the next print statement to display its result on the same line as the letter `A`.

Exercise 78: The Collatz Conjecture

(*22 Lines*)

Consider a sequence of integers that is constructed in the following manner:

Start with any positive integer as the only term in the sequence
While the last term in the sequence is not equal to 1 **do**
  If the last term is even **then**
    Add another term to the sequence by dividing the last term by 2 using floor division
  Else
    Add another term to the sequence by multiplying the last term by 3 and adding 1

The Collatz conjecture states that this sequence will eventually end with one when it begins with any positive integer. Although this conjecture has never been proved, it appears to be true.

Create a program that reads an integer, n, from the user and reports all of the values in the sequence starting with n and ending with one. Your program should allow the user to continue entering new n values (and your program should continue displaying the sequences) until the user enters a value for n that is less than or equal to zero.

> The Collatz conjecture is an example of an open problem in mathematics. While many people have tried to prove that it is true, no one has been able to do so. Information on other open problems in mathematics can be found on the Internet.

Exercise 79: Greatest Common Divisor

(*Solved, 17 Lines*)

The greatest common divisor of two positive integers, n and m, is the largest number, d, which divides evenly into both n and m. There are several algorithms that can be used to solve this problem, including:

Initialize d to the smaller of m and n.
While d does not evenly divide m or d does not evenly divide n **do**
  Decrease the value of d by 1
Report d as the greatest common divisor of n and m

Write a program that reads two positive integers from the user and uses this algorithm to determine and report their greatest common divisor.

Exercise 80: Prime Factors

(27 Lines)

The prime factorization of an integer, *n*, can be determined using the following steps:

> Initialize *factor* to 2
> **While** *factor* is less than or equal to *n* **do**
>   **If** *n* is evenly divisible by *factor* **then**
>   Conclude that *factor* is a factor of *n*
>   Divide *n* by *factor* using floor division
>   **Else**
>   Increase *factor* by 1

Write a program that reads an integer from the user. If the value entered by the user is less than 2 then your program should display an appropriate error message. Otherwise your program should display the prime numbers that can be multiplied together to compute *n*, with one factor appearing on each line. For example:

```
Enter an integer (2 or greater): 72
The prime factors of 72 are:
2
2
2
3
3
```

Exercise 81: Binary to Decimal

(18 Lines)

Write a program that converts a binary (base 2) number to decimal (base 10). Your program should begin by reading the binary number from the user as a string. Then it should compute the equivalent decimal number by processing each digit in the binary number. Finally, your program should display the equivalent decimal number with an appropriate message.

Exercise 82: Decimal to Binary

(Solved, 27 Lines)

Write a program that converts a decimal (base 10) number to binary (base 2). Read the decimal number from the user as an integer and then use the division algorithm shown below to perform the conversion. When the algorithm completes, *result* contains the binary representation of the number. Display the result, along with an appropriate message.

Let *result* be an empty string
Let q represent the number to convert
repeat
  Set r equal to the remainder when q is divided by 2
  Convert r to a string and add it to the beginning of *result*
  Divide q by 2, discarding any remainder, and store the result back into q
until q is 0

Exercise 83: Maximum Integer

(*Solved, 34 Lines*)

This exercise examines the process of identifying the maximum value in a collection of integers. Each of the integers will be randomly selected from the numbers between 1 and 100. The collection of integers may contain duplicate values, and some of the integers between 1 and 100 may not be present.

Take a moment and think about how you would solve this problem on paper. Many people would check each integer in sequence and ask themself if the number that they are currently considering is larger than the largest number that they have seen previously. If it is, then they forget the previous maximum number and remember the current number as the new maximum number. This is a reasonable approach, and will result in the correct answer when the process is performed carefully. If you were performing this task, how many times would you expect to need to update the maximum value and remember a new number?

While we can answer the question posed at the end of the previous paragraph using probability theory, we are going to explore it by simulating the situation. Create a program that begins by selecting a random integer between 1 and 100. Save this integer as the maximum number encountered so far. After the initial integer has been selected, generate 99 additional random integers between 1 and 100. Check each integer as it is generated to see if it is larger than the maximum number encountered so far. If it is then your program should update the maximum number encountered and count the fact that you performed an update. Display each integer after you generate it. Include a notation with those integers which represent a new maximum.

After you have displayed 100 integers your program should display the maximum value encountered, along with the number of times the maximum value was updated during the process. Partial output for the program is shown below, with… representing the remaining integers that your program will display. Run your program several times. Is the number of updates performed on the maximum value what you expected?

```
30
74 <== Update
58
17
40
37
13
34
46
52
80 <== Update
37
97 <== Update
45
55
73
...

The maximum value found was 100
The maximum value was updated 5 times
```

Exercise 84: Coin Flip Simulation

(*47 Lines*)

What's the minimum number of times you have to flip a coin before you can have three consecutive flips that result in the same outcome (either all three are heads or all three are tails)? What's the maximum number of flips that might be needed? How many flips are needed on average? In this exercise we will explore these questions by creating a program that simulates several series of coin flips.

Create a program that uses Python's random number generator to simulate flipping a coin several times. The simulated coin should be fair, meaning that the probability of heads is equal to the probability of tails. Your program should flip simulated coins until either 3 consecutive heads of 3 consecutive tails occur. Display an `H` each time the outcome is heads, and a `T` each time the outcome is tails, with all of the outcomes for one simulation on the same line. Then display the number of flips that were needed to reach 3 consecutive occurrences of the same outcome. When your program is run it should perform the simulation 10 times and report the average number of flips needed. Sample output is shown below:

```
H T T T (4 flips)
H H T T H T H T T H H T H T T H T T T (19 flips)
T T T (3 flips)
T H H H (4 flips)
H H H (3 flips)
T H T T H T H H T T H H T H T H H H (18 flips)
H T T H H H (6 flips)
T H T T T (5 flips)
T T H T T H T H T H H H (12 flips)
T H T T T (5 flips)
On average, 7.9 flips were needed.
```

4 Functions

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.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_4

```
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 85: Compute the Hypotenuse

(*23 Lines*)

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 86: Taxi Fare

(*22 Lines*)

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 87: Shipping Calculator

(*23 Lines*)

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 88: Median of Three Values

(*Solved, 43 Lines*)

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.

Exercise 89: Convert an Integer to Its Ordinal Number

(*47 Lines*)

Words like *first*, *second* and *third* are referred to as ordinal numbers. In this exercise, you will write a function that takes an integer as its only parameter and returns a string containing the appropriate English ordinal number as its only result. Your function must handle the integers between 1 and 12 (inclusive). It should return an empty string if the function is called with an argument outside of this range. Include a main program that demonstrates your function by displaying each integer from 1 to 12 and its ordinal number. Your main program should only run when your file has not been imported into another program.

Exercise 90: The Twelve Days of Christmas

(*Solved, 52 Lines*)

The Twelve Days of Christmas is a repetitive song that describes an increasingly long list of gifts sent to one's true love on each of 12 days. A single gift is sent on the first day. A new gift is added to the collection on each additional day, and then the complete collection is sent. The first three verses of the song are shown below. The complete lyrics are available on the Internet.

> On the first day of Christmas
> my true love sent to me:
> A partridge in a pear tree.
>
> On the second day of Christmas
> my true love sent to me:
> Two turtle doves,
> And a partridge in a pear tree.
>
> On the third day of Christmas
> my true love sent to me:
> Three French hens,
> Two turtle doves,
> And a partridge in a pear tree.

Write a program that displays the complete lyrics for The Twelve Days of Christmas. Your program should include a function that displays one verse of the song. It will take the verse number as its only parameter. Then your program should call this function 12 times with integers that increase from 1 to 12.

Each item that is sent to the recipient in the song should only appear in your program once, with the possible exception of the partridge. It may appear twice if that helps you handle the difference between "A partridge in a pear tree" in the first verse and "And a partridge in a pear tree" in the subsequent verses. Import your solution to Exercise 89 to help you complete this exercise.

Exercise 91: Gregorian Date to Ordinal Date

(*72 Lines*)

An ordinal date consists of a year and a day within it, both of which are integers. The year can be any year in the Gregorian calendar while the day within the year ranges from one, which represents January 1, through to 365 (or 366 if the year is a leap year) which represents December 31. Ordinal dates are convenient when computing differences between dates that are a specific number of days (rather than months). For example, ordinal dates can be used to easily determine whether a customer is within a 90 day return period, the sell-by date for a food-product based on its production date, and the due date for a baby.

Write a function named `ordinalDate` that takes three integers as parameters. These parameters will be a day, month and year respectively. The function should return the day within the year for that date as its only result. Create a main program that reads a day, month and year from the user and displays the day within the year for that date. Your main program should only run when your file has not been imported into another program.

Exercise 92: Ordinal Date to Gregorian Date

(*103 Lines*)

Create a function that takes an ordinal date, consisting of a year and a day within in that year, as its parameters. The function will return the day and month corresponding to that ordinal date as its results. Ensure that your function handles leap years correctly.

Use your function, as well as the `ordinalDate` function that you wrote previously, to create a program that reads a date from the user. Then your program should report a second date that occurs some number of days later. For example, your program could read the date a product was purchased and then report the last date that it can be returned (based on a return period that is a particular number of days), or your program could compute the due date for a baby based on a gestation period of 280 days. Ensure that your program correctly handles cases where the entered date and the computed date occur in different years.

Exercise 93: Center a String in the Terminal Window

(*Solved, 29 Lines*)

Write a function that takes a string, s, as its first parameter, and the width of the window in characters, w, as its second parameter. Your function will return a new string that includes whatever leading spaces are needed so that s will be centered in the window when the new string is printed. The new string should be constructed in the following manner:

- If the length of s is greater than or equal to the width of the window then s should be returned.

- If the length of s is less than the width of the window then a string containing `(len(s) - w) // 2` spaces followed by s should be returned.

Write a main program that demonstrates your function by displaying multiple strings centered in the window.

Exercise 94: Is It a Valid Triangle?

(*33 Lines*)

If you have 3 straws, possibly of differing lengths, it may or may not be possible to lay them down so that they form a triangle when their ends are touching. For example, if all of the straws have a length of 6 inches then one can easily construct an equilateral triangle using them. However, if one straw is 6 inches long, while the other two are each only 2 inches long, then a triangle cannot be formed. More generally, if any one length is greater than or equal to the sum of the other two then the lengths cannot be used to form a triangle. Otherwise they can form a triangle.

Write a function that determines whether or not three lengths can form a triangle. The function will take 3 parameters and return a Boolean result. If any of the lengths are less than or equal to 0 then your function should return `False`. Otherwise it should determine whether or not the lengths can be used to form a triangle using the method described in the previous paragraph, and return the appropriate result. In addition, write a program that reads 3 lengths from the user and demonstrates the behaviour of your function.

Exercise 95: Capitalize It

(*Solved, 68 Lines*)

Many people do not use capital letters correctly, especially when typing on small devices like smart phones. To help address this situation, you will create a function that takes a string as its only parameter and returns a new copy of the string that has been correctly capitalized. In particular, your function must:

- Capitalize the first non-space character in the string,
- Capitalize the first non-space character after a period, exclamation mark or question mark, and
- Capitalize a lowercase "i" if it is preceded by a space and followed by a space, period, exclamation mark, question mark or apostrophe.

Implementing these transformations will correct most capitalization errors. For example, if the function is provided with the string "what time do i have to be there? what's the address? this time i'll try to be on time!" then it should return the string "What time do I have to be there? What's the address? This time I'll try to be on time!". Include a main program that reads a string from the user, capitalizes it using your function, and displays the result.

Exercise 96: Does a String Represent an Integer?

(*Solved, 30 Lines*)

In this exercise you will write a function named `isInteger` that determines whether or not the characters in a string represent a valid integer. When determining if a string represents an integer you should ignore any leading or trailing white space. Once this white space is ignored, a string represents an integer if its length is at least one and it only contains digits, or if its first character is either + or - and the first character is followed by one or more characters, all of which are digits.

Write a main program that reads a string from the user and reports whether or not it represents an integer. Ensure that the main program will not run if the file containing your solution is imported into another program.

> Hint: You may find the `lstrip`, `rstrip` and/or `strip` methods for strings helpful when completing this exercise. Documentation for these methods is available online.

Exercise 97: Operator Precedence

(*30 Lines*)

Write a function named `precedence` that returns an integer representing the precedence of a mathematical operator. A string containing the operator will be passed to the function as its only parameter. Your function should return 1 for + and `-`, 2 for `*` and `/`, and 3 for `^`. If the string passed to the function is not one of these operators then the function should return `-1`. Include a main program that reads an operator from the user and either displays the operator's precedence or an error message indicating that the input was not an operator. Your main program should only run when the file containing your solution has not been imported into another program.

> In this exercise, along with others that appear later in the book, we will use `^` to represent exponentiation. Using `^` instead of Python's choice of `**` will make these exercises easier because an operator will always be a single character.

Exercise 98: Is a Number Prime?

(*Solved, 28 Lines*)

A prime number is an integer greater than one that is only divisible by one and itself. Write a function that determines whether or not its parameter is prime, returning `True` if it is, and `False` otherwise. Write a main program that reads an integer from the user and displays a message indicating whether or not it is prime. Ensure that the main program will not run if the file containing your solution is imported into another program.

Exercise 99: Next Prime

(*27 Lines*)

In this exercise you will create a function named `nextPrime` that finds and returns the first prime number larger than some integer, n. The value of n will be passed to the function as its only parameter. Include a main program that reads an integer from the user and displays the first prime number larger than the entered value. Import and use your solution to Exercise 98 while completing this exercise.

Exercise 100: Random Password

(*Solved, 33 Lines*)

Write a function that generates a random password. The password should have a random length of between 7 and 10 characters. Each character should be randomly selected from positions 33 to 126 in the ASCII table. Your function will not take any parameters. It will return the randomly generated password as its only result. Display the randomly generated password in your file's main program. Your main program should only run when your solution has not been imported into another file.

> Hint: You will probably find the `chr` function helpful when completing this exercise. Detailed information about this function is available online.

Exercise 101: Random License Plate

(*45 Lines*)

In a particular jurisdiction, older license plates consist of three letters followed by three digits. When all of the license plates following that pattern had been used, the format was changed to four digits followed by three letters.

Write a function that generates a random license plate. Your function should have approximately equal odds of generating a sequence of characters for an old license plate or a new license plate. Write a main program that calls your function and displays the randomly generated license plate.

Exercise 102: Check a Password

(*Solved, 40 Lines*)

In this exercise you will write a function that determines whether or not a password is good. We will define a good password to be a one that is at least 8 characters long and contains at least one uppercase letter, at least one lowercase letter, and at least one number. Your function should return `True` if the password passed to it as its only parameter is good. Otherwise it should return `False`. Include a main program that reads a password from the user and reports whether or not it is good. Ensure that your main program only runs when your solution has not been imported into another file.

Exercise 103: Random Good Password

(*22 Lines*)

Using your solutions to Exercises 100 and 102, write a program that generates a random good password and displays it. Count and display the number of attempts that were needed before a good password was generated. Structure your solution so that it imports the functions you wrote previously and then calls them from a function named `main` in the file that you create for this exercise.

Exercise 104: Hexadecimal and Decimal Digits

(*41 Lines*)

Write two functions, `hex2int` and `int2hex`, that convert between hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F) and decimal (base 10) integers. The `hex2int` function is responsible for converting a string containing a single hexadecimal digit to a base 10 integer, while the `int2hex` function is responsible for converting an integer between 0 and 15 to a single hexadecimal digit. Each function will take the value to convert as its only parameter and return the converted value as its only result. Ensure that the `hex2int` function works correctly for both uppercase and lowercase letters. Your functions should display a meaningful error message and end the program if the parameter's value is outside of the expected range.

Exercise 105: Arbitrary Base Conversions

(*Solved, 71 Lines*)

Write a program that allows the user to convert a number from one base to another. Your program should support bases between 2 and 16 for both the input number and the result number. If the user chooses a base outside of this range then an appropriate error message should be displayed and the program should exit. Divide your program into several functions, including a function that converts from an arbitrary base to base 10, a function that converts from base 10 to an arbitrary base, and a main program that reads the bases and input number from the user. You may find your solutions to Exercises 81, 82 and 104 helpful when completing this exercise.

Exercise 106: Days in a Month

(*47 Lines*)

Write a function that determines how many days there are in a particular month. Your function will take two parameters: The month as an integer between 1 and 12, and the year as a four digit integer. Ensure that your function reports the correct number of days in February for leap years. Include a main program that reads a month and year from the user and displays the number of days in that month. You may find your solution to Exercise 58 helpful when solving this problem.

Exercise 107: Reduce a Fraction to Lowest Terms

(*Solved, 46 Lines*)

Write a function that takes two positive integers that represent the numerator and denominator of a fraction as its only parameters. The body of the function should reduce the fraction to lowest terms and then return both the numerator and denominator of the reduced fraction as its result. For example, if the parameters passed to the function are 6 and 63 then the function should return 2 and 21. Include a main program that allows the user to enter a numerator and denominator. Then your program should display the reduced fraction.

> Hint: In Exercise 79 you wrote a program for computing the greatest common divisor of two positive integers. You may find that code useful when completing this exercise.

Exercise 108: Reduce Measures

(*Solved, 87 Lines*)

Many recipe books still use cups, tablespoons and teaspoons to describe the volumes of ingredients used when cooking or baking. While such recipes are easy enough to follow if you have the appropriate measuring cups and spoons, they can be difficult to double, triple or quadruple when cooking Christmas dinner for the entire extended family. For example, a recipe that calls for 4 tablespoons of an ingredient requires 16 tablespoons when quadrupled. However, 16 tablespoons would be better expressed (and easier to measure) as 1 cup.

Write a function that expresses an imperial volume using the largest units possible. The function will take the number of units as its first parameter, and the unit of measure (cup, tablespoon or teaspoon) as its second parameter. It will return a string representing the measure using the largest possible units as its only result. For example, if the function is provided with parameters representing 59 teaspoons then it should return the string “1 cup, 3 tablespoons, 2 teaspoons”.

> Hint: One cup is equivalent to 16 tablespoons. One tablespoon is equivalent to 3 teaspoons.

Exercise 109: Magic Dates

(*Solved, 26 Lines*)

A magic date is a date where the day multiplied by the month is equal to the two digit year. For example, June 10, 1960 is a magic date because June is the sixth month, and 6 times 10 is 60, which is equal to the two digit year. Write a function that determines whether or not a date is a magic date. Use your function to create a main program that finds and displays all of the magic dates in the 20th century. You will probably find your solution to Exercise 106 helpful when completing this exercise.

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.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_5

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 110: Sorted Order

(*Solved, 22 Lines*)

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 111: Reverse Order

(*20 Lines*)

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 112: Remove Outliers

(*Solved, 44 Lines*)

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 113: Avoiding Duplicates

(*Solved, 21 Lines*)

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 114: Negatives, Zeros and Positives

(*Solved, 36 Lines*)

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 115: List of Proper Divisors

(*36 Lines*)

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 116: Perfect Numbers

(*Solved, 35 Lines*)

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. Import your solution to Exercise 115 when completing this task.

Exercise 117: Only the Words

(*38 Lines*)

In this exercise you will create a program that identifies all of the words in a string entered by the user. Begin by writing a function that takes a string as its only parameter. Your function should return a list of the words in the string with the punctuation marks at the edges of the words removed. The punctuation marks that you must consider include commas, periods, question marks, hyphens, apostrophes, exclamation points, colons, and semicolons. Do not remove punctuation marks that appear in the middle of a word, such as the apostrophes used to form a contraction. For example, if your function is provided with the string `"Contractions include: don't, isn't, and wouldn't."` then your function should return the list `["Contractions", "include", "don't", "isn't", "and", "wouldn't"]`.

Write a main program that demonstrates your function. It should read a string from the user and then display all of the words in the string with the punctuation marks removed. You will need to import your solution to this exercise when completing Exercises 118 and 167. As a result, you should ensure that your main program only runs when your file has not been imported into another program.

Exercise 118: Word by Word Palindromes

(*34 Lines*)

Exercises 75 and 76 previously introduced the notion of a palindrome. Such palindromes examined the characters in a string, possibly ignoring spacing and punctuation marks, to see if the string was the same forwards and backwards. While palindromes are most commonly considered character by character, the notion of a palindrome can be extended to larger units. For example, while the sentence "Is it crazy how saying sentences backwards creates backwards sentences saying how crazy it is?" isn't a character by character palindrome, it is a palindrome when examined a word at a time (and when capitalization and punctuation are ignored). Other examples of word by word palindromes include "Herb the sage eats sage, the herb" and "Information school graduate seeks graduate school information".

Create a program that reads a string from the user. Your program should report whether or not the entered string is a word by word palindrome. Ignore spacing and punctuation when determining the result.

Exercise 119: Below and Above Average

(*44 Lines*)

Write a program that reads numbers from the user until a blank line is entered. Your program should display the average of all of the values entered by the user. Then the program should display all of the below average values, followed by all of the average values (if any), followed by all of the above average values. An appropriate label should be displayed before each list of values.

Exercise 120: Formatting a List

(*Solved, 41 Lines*)

When writing out a list of items in English, one normally separates the items with commas. In addition, the word "and" is normally included before the last item, unless the list only contains one item. Consider the following four lists:

```
apples
apples and oranges
apples, oranges and bananas
apples, oranges, bananas and lemons
```

Write a function that takes a list of strings as its only parameter. Your function should return a string that contains all of the items in the list, formatted in the manner described previously, as its only result. While the examples shown previously only include lists containing four elements or less, your function should behave correctly for lists of any length. Include a main program that reads several items from the user, formats them by calling your function, and then displays the result returned by the function.

Exercise 121: Random Lottery Numbers

(*Solved, 28 Lines*)

In order to win the top prize in a particular lottery, one must match all 6 numbers on his or her ticket to the 6 numbers between 1 and 49 that are drawn by the lottery organizer. Write a program that generates a random selection of 6 numbers for a lottery ticket. Ensure that the 6 numbers selected do not contain any duplicates. Display the numbers in ascending order.

Exercise 122: Pig Latin

(*32 Lines*)

Pig Latin is a language constructed by transforming English words. While the origins of the language are unknown, it is mentioned in at least two documents from the nineteenth century, suggesting that it has existed for more than 100 years. The following rules are used to translate English into Pig Latin:

- If the word begins with a consonant (including `y`), then all letters at the beginning of the word, up to the first vowel (excluding `y`), are removed and then added to the end of the word, followed by `ay`. For example, `computer` becomes `omputercay` and `think` becomes `inkthay`.
- If the word begins with a vowel (not including `y`), then `way` is added to the end of the word. For example, `algorithm` becomes `algorithmway` and `office` becomes `officeway`.

Write a program that reads a line of text from the user. Then your program should translate the line into Pig Latin and display the result. You may assume that the string entered by the user only contains lowercase letters and spaces.

Exercise 123: Pig Latin Improved

(*51 Lines*)

Extend your solution to Exercise 122 so that it correctly handles uppercase letters and punctuation marks such as commas, periods, question marks and exclamation marks. If an English word begins with an uppercase letter then its Pig Latin representation should also begin with an uppercase letter and the uppercase letter moved to the end of the word should be changed to lowercase. For example, `Computer` should become `Omputercay`. If a word ends in a punctuation mark then the punctuation mark should remain at the end of the word after the transformation has been performed. For example, `Science!` should become `Iencescay!`.

Exercise 124: Line of Best Fit

(*41 Lines*)

A line of best fit is a straight line that best approximates a collection of n data points. In this exercise, we will assume that each point in the collection has an x coordinate and a y coordinate. The symbols $\bar{x}$ and $\bar{y}$ are used to represent the average x value in the collection and the average y value in the collection respectively. The line of best fit is represented by the equation $y = mx + b$ where m and b are calculated using the following formulas:

$$m = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{\sum x^2 - \frac{(\sum x)^2}{n}}$$

$$b = \bar{y} - m\bar{x}$$

Write a program that reads a collection of points from the user. The user will enter the first x coordinate on its own line, followed by the first y coordinate on its own line. Allow the user to continue entering coordinates, with the x and y values each entered on their own line, until your program reads a blank line for the x coordinate. Display the formula for the line of best fit in the form $y = mx + b$ by replacing m and b with the values calculated by the preceding formulas. For example, if the user inputs the coordinates (1, 1), (2, 2.1) and (3, 2.9) then your program should display $y = 0.95x + 0.1$.

Exercise 125: Shuffling a Deck of Cards

(*Solved, 49 Lines*)

A standard deck of playing cards contains 52 cards. Each card has one of four suits along with a value. The suits are normally spades, hearts, diamonds and clubs while the values are 2 through 10, Jack, Queen, King and Ace.

Each playing card can be represented using two characters. The first character is the value of the card, with the values 2 through 9 being represented directly. The characters “T”, “J”, “Q”, “K” and “A” are used to represent the values 10, Jack, Queen, King and Ace respectively. The second character is used to represent the suit of the card. It is normally a lowercase letter: “s” for spades, “h” for hearts, “d” for diamonds and “c” for clubs. The following table provides several examples of cards and their two-character representations.

Card	Abbreviation
Jack of spades	Js
Two of clubs	2c
Ten of diamonds	Td
Ace of hearts	Ah
Nine of spades	9s

Begin by writing a function named `createDeck`. It will use loops to create a complete deck of cards by storing the two-character abbreviations for all 52 cards into a list. Return the list of cards as the function's only result. Your function will not require any parameters.

Write a second function named `shuffle` that randomizes the order of the cards in a list. One technique that can be used to shuffle the cards is to visit each element in the list and swap it with another random element in the list. You must write your own loop for shuffling the cards. You cannot make use of Python's built-in shuffle function.

Use both of the functions described in the previous paragraphs to create a main program that displays a deck of cards before and after it has been shuffled. Ensure that your main program only runs when your functions have not been imported into another file.

A good shuffling algorithm is unbiased, meaning that every different arrangement of the elements is equally probable when the algorithm completes. While the approach described earlier in this problem suggested visiting each element in sequence and swapping it with an element at a random index, such an algorithm is biased. In particular, elements that appear later in the original list are more likely to end up later in the shuffled list. Counterintuitively, an unbiased shuffle can be achieved by visiting each element in sequence and swapping it to a random index between the position of the current element and the end of the list instead of randomly selecting any index.

Exercise 126: Dealing Hands of Cards

(*44 Lines*)

In many card games each player is dealt a specific number of cards after the deck has been shuffled. Write a function, `deal`, which takes the number of hands, the number of cards per hand, and a deck of cards as its three parameters. Your function should return a list containing all of the hands that were dealt. Each hand will be represented as a list of cards.

When dealing the hands, your function should modify the deck of cards passed to it as a parameter, removing each card from the deck as it is added to a player's hand. When cards are dealt, it is customary to give each player a card before any player receives an additional card. Your function should follow this custom when constructing the hands for the players.

Use your solution to Exercise 125 to help you construct a main program that creates and shuffles a deck of cards, and then deals out four hands of five cards each. Display all of the hands of cards, along with the cards remaining in the deck after the hands have been dealt.

Exercise 127: Is a List already in Sorted Order?

(*41 Lines*)

Write a function that determines whether or not a list of values is in sorted order (either ascending or descending). The function should return `True` if the list is already sorted. Otherwise it should return `False`. Write a main program that reads a list of numbers from the user and then uses your function to report whether or not the list is sorted.

> Make sure you consider these questions when completing this exercise: Is a list that is empty in sorted order? What about a list containing one element?

Exercise 128: Count the Elements

(*Solved, 48 Lines*)

Python's standard library includes a method named `count` that determines how many times a specific value occurs in a list. In this exercise, you will create a new function named `countRange`. It will determine and return the number of elements within a list that are greater than or equal to some minimum value and less than some maximum value. Your function will take three parameters: the list, the minimum value and the maximum value. It will return an integer result greater than or equal to 0. Include a main program that demonstrates your function for several different lists, minimum values and maximum values. Ensure that your program works correctly for both lists of integers and lists of floating-point numbers.

Exercise 129: Tokenizing a String

(*Solved, 47 Lines*)

Tokenizing is the process of converting a string into a list of substrings, known as tokens. In many circumstances, a list of tokens is far easier to work with than the original string because the original string may have irregular spacing. In some cases substantial work is also required to determine where one token ends and the next one begins.

In a mathematical expression, tokens are items such as operators, numbers and parentheses. The operator symbols that we will consider in this (and subsequent) problems are `*`, `/`, `^`, `-` and `+`. Operators and parentheses are easy to identify because the token is always a single character, and the character is never part of another token. Numbers are slightly more challenging to identify because the token may consist of multiple characters. Any sequence of consecutive digits should be treated as one number token.

Write a function that takes a string containing a mathematical expression as its only parameter and breaks it into a list of tokens. Each token should be a parenthesis, an operator, or a number. (For simplicity we will only work with integers in this problem). Return the list of tokens as the function's only result.

You may assume that the string passed to your function always contains a valid mathematical expression consisting of parentheses, operators and integers. However, your function must handle variable amounts of whitespace (including no whitespace) between these elements. Include a main program that demonstrates your tokenizing function by reading an expression from the user and printing the list of tokens. Ensure that the main program will not run when the file containing your solution is imported into another program.

Exercise 130: Unary and Binary Operators

(*Solved, 45 Lines*)

Some mathematical operators are unary while others are binary. Unary operators act on one value while binary operators act on two. For example, in the expression `2 * -3` the `*` is a binary operator because it acts on both the 2 and the −3 while the `-` is a unary operator because it only acts on the 3.

An operator's symbol is not always sufficient to determine whether it is unary or binary. For example, while the `-` operator was unary in the previous expression, the same character is used to represent the binary `-` operator in an expression such as `2 - 3`. This ambiguity, which is also present for the `+` operator, must be removed before other interesting operations can be performed on a list of tokens representing a mathematical expression.

Create a function that identifies unary `+` and `-` operators in a list of tokens and replaces them with `u+` and `u-` respectively. Your function will take a list of tokens for a mathematical expression as its only parameter and return a new list of tokens where the unary `+` and `-` operators have been substituted as its only result. A `+` or `-` operator is unary if it is the first token in the list, or if the token that immediately precedes it is an operator or open parenthesis. Otherwise the operator is binary.

Write a main program that demonstrates that your function works correctly by reading, tokenizing, and marking the unary operators in an expression entered by the user. Your main program should not execute when your function is imported into another program.

Exercise 131: Infix to Postfix

(*63 Lines*)

Mathematical expressions are often written in infix form, where operators appear between the operands on which they act. While this is a common form, it is also possible to express mathematical expressions in postfix form, where the operator appears after all of its operands. For example, the infix expression `3 + 4` is written as `3 4 +` in postfix form. One can convert an infix expression to postfix form using the following algorithm:

Create a new empty list, *operators*
Create a new empty list, *postfix*

For each token in the infix expression
    If the token is an integer **then**
        Append the token to *postfix*
    If the token is an operator **then**
        While *operators* is not empty and
                    the last item in *operators* is not an open parenthesis and
                    precedence(token) < precedence(last item in *operators*) **do**
            Remove the last item from *operators* and append it to *postfix*
        Append the token to *operators*
    If the token is an open parenthesis **then**
        Append the token to *operators*
    If the token is a close parenthesis **then**
        While the last item in *operators* is not an open parenthesis **do**
            Remove the last item from *operators* and append it to *postfix*
        Remove the open parenthesis from *operators*

While *operators* is not the empty list **do**
    Remove the last item from *operators* and append it to *postfix*

Return *postfix* as the result of the algorithm

Use your solutions to Exercises 129 and 130 to tokenize a mathematical expression and identify any unary operators in it. Then use the algorithm above to transform the expression from infix form to postfix form. Your code that implements the preceding algorithm should reside in a function that takes a list of tokens representing an infix expression (with the unary operators marked) as its only parameter. It should return a list of tokens representing the equivalent postfix expression as its only result. Include a main program that demonstrates your infix to postfix function by reading an expression from the user in infix form and displaying it in postfix form.

The purpose of converting from infix form to postfix form will become apparent when you read Exercise 132. You may find your solutions to Exercises 96 and 97 helpful when completing this problem. While you should be able to use your solution to Exercise 96 without any modifications, your solution to Exercise 97 will need to be extended so that it returns the correct precedence for the unary operators. The unary operators should have higher precedence than multiplication and division, but lower precedence than exponentiation.

Exercise 132: Evaluate Postfix

(*63 Lines*)

Evaluating a postfix expression is easier than evaluating an infix expression because it does not contain any parentheses and there are no operator precedence rules to consider. A postfix expression can be evaluated using the following algorithm:

Create a new empty list, *values*

For each token in the postfix expression
  If the token is a number **then**
    Convert it to an integer and append it to *values*
  Else if the token is a unary minus **then**
    Remove an item from the end of *values*
    Negate the item and append the result of the negation to *values*
  Else if the token is a binary operator **then**
    Remove an item from the end of *values* and call it *right*
    Remove an item from the end of *values* and call it *left*
    Compute the result of applying the operator to *left* and *right*
    Append the result to *values*

Return the first item in *values* as the value of the expression

Write a program that reads a mathematical expression in infix form from the user, converts it to postfix form, evaluates it, and displays its value. Use your solutions to Exercises 129, 130 and 131, along with the algorithm shown above, to solve this problem.

> The algorithms provided in Exercises 131 and 132 do not perform any error checking. As a result, your programs may crash or generate incorrect results if you provide them with invalid input. The algorithms presented in these exercises can be extended to detect invalid input and respond to it in a reasonable manner. Doing so is left as an independent study exercise for the interested student.

Exercise 133: Does a List Contain a Sublist?

(*44 Lines*)

A sublist is a list that is part of a larger list. A sublist may be a list containing a single element, multiple elements, or even no elements at all. For example, `[1]`, `[2]`, `[3]` and `[4]` are all sublists of `[1, 2, 3, 4]`. The list `[2, 3]` is also a sublist of `[1, 2, 3, 4]`, but `[2, 4]` is not a sublist `[1, 2, 3, 4]` because the elements `2` and `4` are not adjacent in the longer list. The empty list is a sublist of any list. As a result, `[]` is a sublist of `[1, 2, 3, 4]`. A list is a sublist of itself, meaning that `[1, 2, 3, 4]` is also a sublist of `[1, 2, 3, 4]`.

In this exercise you will create a function, `isSublist`, that determines whether or not one list is a sublist of another. Your function should take two lists, `larger` and `smaller`, as its only parameters. It should return `True` if and only if `smaller` is a sublist of `larger`. Write a main program that demonstrates your function.

Exercise 134: Generate All Sublists of a List

(*Solved, 41 Lines*)

Using the definition of a sublist from Exercise 133, write a function that returns a list containing every possible sublist of a list. For example, the sublists of `[1, 2, 3]` are `[]`, `[1]`, `[2]`, `[3]`, `[1, 2]`, `[2, 3]` and `[1, 2, 3]`. Note that your function will always return a list containing at least the empty list because the empty list is a sublist of every list. Include a main program that demonstrate your function by displaying all of the sublists of several different lists.

Exercise 135: The Sieve of Eratosthenes

(*Solved, 33 Lines*)

The Sieve of Eratosthenes is a technique that was developed more than 2,000 years ago to easily find all of the prime numbers between 2 and some limit, say 100. A description of the algorithm follows:

Write down all of the numbers from 0 to the limit
Cross out 0 and 1 because they are not prime

Set p equal to 2
While p is less than the limit **do**
    Cross out all multiples of p (but not p itself)
    Set p equal to the next number in the list that is not crossed out

Report all of the numbers that have not been crossed out as prime

The key to this algorithm is that it is relatively easy to cross out every nth number on a piece of paper. This is also an easy task for a computer—a for loop can simulate this behavior when a third parameter is provided to the `range` function. When a number is crossed out, we know that it is no longer prime, but it still occupies space on the piece of paper, and must still be considered when computing later prime numbers. As a result, you should **not** simulate crossing out a number by removing it from the list. Instead, you should simulate crossing out a number by replacing it with 0. Then, once the algorithm completes, all of the non-zero values in the list are prime.

Create a Python program that uses this algorithm to display all of the prime numbers between 2 and a limit entered by the user. If you implement the algorithm correctly you should be able to display all of the prime numbers less than 1,000,000 in a few seconds.

> This algorithm for finding prime numbers is not Eratosthenes' only claim to fame. His other noteworthy accomplishments include calculating the circumference of the Earth and the tilt of the Earth's axis. He also served as the Chief Librarian at the Library of Alexandria.

6 Dictionaries

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.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_6

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 136: Reverse Lookup

(*Solved, 45 Lines*)

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 137: Two Dice Simulation

(*Solved, 43 Lines*)

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 138: Text Messaging

(*21 Lines*)

On some basic cell phones, text messages can be sent using the numeric keypad. Because each key has multiple letters associated with it, multiple key presses are needed for most letters. Pressing the number once generates the first character listed for that key. Pressing the number 2, 3, 4 or 5 times generates the second, third, fourth or fifth character.

Key	Symbols
1	. , ? ! :
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P Q R S
8	T U V
9	W X Y Z
0	*space*

Write a program that displays the key presses needed for a message entered by the user. Construct a dictionary that maps from each letter or symbol to the key presses needed to generate it. Then use the dictionary to create and display the presses needed for the user's message. For example, if the user enters `Hello, World!` then your program should output `4433555555666110966677755531111`. Ensure that your program handles both uppercase and lowercase letters. Ignore any characters that aren't listed in the table above such as semicolons and parentheses.

Exercise 139: Morse Code

(*15 Lines*)

Morse code is an encoding scheme that uses dashes and dots to represent digits and letters. In this exercise, you will write a program that uses a dictionary to store the mapping from these symbols to Morse code. Use a period to represent a dot, and a hyphen to represent a dash. The mapping from characters to dashes and dots is shown in Table 6.1.

Your program should read a message from the user. Then it should translate all of the letters and digits in the message to Morse code, leaving a space between each sequence of dashes and dots. Your program should ignore any characters that are not listed in the previous table. The Morse code for `Hello, World!` is shown below:

```
.... . .-.. .-.. --- .-- --- .-. .-.. -..
```

Table 6.1 Morse code for letters and numerals

Character	Code	Character	Code	Character	Code	Character	Code
A	.–	J	.–––	S	...	1	.––––
B	–...	K	–.–	T	–	2	..–––
C	–.–.	L	.–..	U	..–	3	...––
D	–..	M	––	V	...–	4	....–
E	.	N	–.	W	.––	5	.....
F	..–.	O	–––	X	–..–	6	–....
G	––.	P	.––.	Y	–.––	7	––...
H	....	Q	––.–	Z	––..	8	–––..
I	..	R	.–.	0	–––––	9	––––.

Morse code was originally developed in the nineteenth century for use over telegraph wires. It is still used today, more than 160 years after it was first created.

Exercise 140: Postal Codes

(24 Lines)

The first, third and fifth characters in a Canadian postal code are letters while the second, fourth and sixth characters are digits. The province or territory in which an address resides can be determined from the first character of its postal code, as shown in the following table. No valid postal codes currently begin with D, F, I, O, Q, U, W, or Z.

Province / Territory	First Character(s)
Newfoundland	A
Nova Scotia	B
Prince Edward Island	C
New Brunswick	E
Quebec	G, H and J
Ontario	K, L, M, N and P
Manitoba	R
Saskatchewan	S
Alberta	T
British Columbia	V
Nunavut	X
Northwest Territories	X
Yukon	Y

The second character in a postal code identifies whether the address is rural or urban. If that character is a 0 then the address is rural. Otherwise it is urban.

Create a program that reads a postal code from the user and displays the province or territory associated with it, along with whether the address is urban or rural. For example, if the user enters `T2N 1N4` then your program should indicate that the postal code is for an urban address in Alberta. If the user enters `X0A 1B2` then your program should indicate that the postal code is for a rural address in Nunavut or Northwest Territories. Use a dictionary to map from the first character of the postal code to the province or territory name. Display a meaningful error message if the postal code begins with an invalid character, or if the second character in the postal code is not a digit.

Exercise 141: Write out Numbers in English

(*65 Lines*)

While the popularity of cheques as a payment method has diminished in recent years, some companies still issue them to pay employees or vendors. The amount being paid normally appears on a cheque twice, with one occurrence written using digits, and the other occurrence written using English words. Repeating the amount in two different forms makes it much more difficult for an unscrupulous employee or vendor to modify the amount on the cheque before depositing it.

In this exercise, your task is to create a function that takes an integer between 0 and 999 as its only parameter, and returns a string containing the English words for that number. For example, if the parameter to the function is 142 then your function should return "`one hundred forty two`". Use one or more dictionaries to implement your solution rather than large if/elif/else constructs. Include a main program that reads an integer from the user and displays its value in English words.

Exercise 142: Unique Characters

(*Solved, 16 Lines*)

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 143: Anagrams

(*Solved, 39 Lines*)

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.

Exercise 144: Anagrams Again

(*48 Lines*)

The notion of anagrams can be extended to multiple words. For example, "William Shakespeare" and "I am a weakish speller" are anagrams when capitalization and spacing are ignored.

Extend your program from Exercise 143 so that it is able to check if two phrases are anagrams. Your program should ignore capitalization, punctuation marks and spacing when making the determination.

Exercise 145: Scrabble™ Score

(*Solved, 18 Lines*)

In the game of Scrabble™, each letter has points associated with it. The total score of a word is the sum of the scores of its letters. More common letters are worth fewer points while less common letters are worth more points. The points associated with each letter are shown below:

Points	Letters
1	A, E, I, L, N, O, R, S, T and U
2	D and G
3	B, C, M and P
4	F, H, V, W and Y
5	K
8	J and X
10	Q and Z

Write a program that computes and displays the Scrabble™ score for a word. Create a dictionary that maps from letters to point values. Then use the dictionary to compute the score.

A Scrabble™ board includes some squares that multiply the value of a letter or the value of an entire word. We will ignore these squares in this exercise.

Exercise 146: Create a Bingo Card

(*Solved, 58 Lines*)

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.

> You may be aware that Bingo cards often have a "free" space in the middle of the card. We won't consider the free space in this exercise.

Exercise 147: Checking for a Winning Card

(*102 Lines*)

A winning Bingo card contains a line of 5 numbers that have all been called. Players normally record the numbers that have been called by crossing them out or marking them with a Bingo dauber. In this exercise we will mark that a number has been called by replacing it with a 0 in the Bingo card dictionary.

Write a function that takes a dictionary representing a Bingo card as its only parameter. If the card contains a line of five zeros (vertical, horizontal or diagonal) then your function should return `True`, indicating that the card has won. Otherwise the function should return `False`.

Create a main program that demonstrates your function by creating several Bingo cards, displaying them, and indicating whether or not they contain a winning line. You should demonstrate your function with at least one card with a horizontal line, at least one card with a vertical line, at least one card with a diagonal line, and at least one card that has some numbers crossed out but does not contain a winning line. You will probably want to import your solution to Exercise 146 when completing this exercise.

> Hint: Because there are no negative numbers on a Bingo card, finding a line of 5 zeros is equivalent to finding a line of 5 entries that sum to zero. You may find the summation problem easier to solve.

Exercise 148: Play Bingo

(*88 Lines*)

In this exercise you will write a program that simulates a game of Bingo for a single card. Begin by generating a list of all of the valid Bingo calls (B1 through O75). Once the list has been created you can randomize the order of its elements by calling the `shuffle` function in the `random` module. Then your program should consume calls out of the list and cross out numbers on the card until the card contains a winning line. Simulate 1,000 games and report the minimum, maximum and average number of calls that must be made before the card wins. You may find it helpful to import your solutions to Exercises 146 and 147 when completing this exercise.

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.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_7

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:

http://www.cpsc.ucalgary.ca/~bdstephe/PythonWorkbook

Exercise 149: Display the Head of a File

(*Solved, 40 Lines*)

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 150: Display the Tail of a File

(*Solved, 35 Lines*)

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 151: Concatenate Multiple Files

(*Solved, 28 Lines*)

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.

Exercise 152: Number the Lines in a File

(*23 Lines*)

Create a program that reads lines from a file, adds line numbers to them, and then stores the numbered lines into a new file. The name of the input file will be read from the user, as will the name of the new file that your program will create. Each line in the output file should begin with the line number, followed by a colon and a space, followed by the line from the input file.

Exercise 153: Find the Longest Word in a File

(*39 Lines*)

In this exercise you will create a Python program that identifies the longest word(s) in a file. Your program should output an appropriate message that includes the length of the longest word, along with all of the words of that length that occurred in the file. Treat any group of non-white space characters as a word, even if it includes digits or punctuation marks.

Exercise 154: Letter Frequencies

(*43 Lines*)

One technique that can be used to help break some simple forms of encryption is frequency analysis. This analysis examines the encrypted text to determine which characters are most common. Then it tries to map the most common letters in English, such as `E` and `T`, to the most commonly occurring characters in the encrypted text.

Write a program that initiates this process by determining and displaying the frequencies of all of the letters in a file. Ignore spaces, punctuation marks, and digits as you perform this analysis. Your program should be case insensitive, treating `a` and `A` as equivalent. The user will provide the name of the file to analyze as a command line argument. Your program should display a meaningful error message if the user provides the wrong number of command line arguments, or if the program is unable to open the file indicated by the user.

Exercise 155: Words that Occur Most

(*37 Lines*)

Write a program that displays the word (or words) that occur most frequently in a file. Your program should begin by reading the name of the file from the user. Then it should process every line in the file. Each line will need to be split into words, and any leading or trailing punctuation marks will need to be removed from each word. Your program should also ignore capitalization when counting how many times each word occurs.

> Hint: You will probably find your solution to Exercise 117 helpful when completing this task.

Exercise 156: Sum a Collection of Numbers

(*Solved, 26 Lines*)

Create a program that sums all of the numbers entered by the user while ignoring any input that is not a valid number. Your program should display the current sum after each number is entered. It should display an appropriate message after each non-numeric input, and then continue to sum any additional numbers entered by the user. Exit the program when the user enters a blank line. Ensure that your program works correctly for both integers and floating-point numbers.

> Hint: This exercise requires you to use exceptions without using files.

Exercise 157: Both Letter Grades and Grade Points

(*106 Lines*)

Write a program that converts from letter grades to grade points and vice-versa. Your program should allow the user to convert multiple values, with one value entered on each line. Begin by attempting to convert each value entered by the user from a number of grade points to a letter grade. If an exception occurs during this process then your program should attempt to convert the value from a letter grade to a number of grade points. If both conversions fail then your program should output a message indicating that the supplied input is invalid. Design your program so that it continues performing conversions (or reporting errors) until the user enters a blank line. Your solutions to Exercises 52 and 53 may be helpful when completing this exercise.

Exercise 158: Remove Comments

(*Solved, 53 Lines*)

Python uses the # character to mark the beginning of a comment. The comment continues from the # character to the end of the line containing it. Python does not provide any mechanism for ending a comment before the end of a line.

In this exercise, you will create a program that removes all of the comments from a Python source file. Check each line in the file to determine if a # character is present. If it is then your program should remove all of the characters from the # character to the end of the line (we will ignore the situation where the comment character occurs inside of a string). Save the modified file using a new name. Both the name of the input file and the name of the output file should be read from the user. Ensure that an appropriate error message is displayed if a problem is encountered while accessing either of the files.

Exercise 159: Two Word Random Password

(*Solved, 39 Lines*)

While generating a password by selecting random characters usually creates one that is relatively secure, it also generally gives a password that is difficult to memorize. As an alternative, some systems construct a password by taking two English words and concatenating them. While this password may not be as secure, it is normally much easier to memorize.

Write a program that reads a file containing a list of words, randomly selects two of them, and concatenates them to produce a new password. When producing the password ensure that the total length is between 8 and 10 characters, and that each word used is at least three letters long. Capitalize each word in the password so that the user can easily see where one word ends and the next one begins. Finally, your program should display the password for the user.

Exercise 160: Weird Words

(*67 Lines*)

Students learning to spell in English are often taught the rhyme "I before E except after C". This rule of thumb advises that when an I and an E are adjacent in a word, the I will precede the E, unless they are immediately preceded by a C. When preceded by a C the E will appear ahead of the I. This advice holds true for words without an immediately preceding C such as believe, chief, fierce and friend, and is similarly true for words with an immediately preceding C such as ceiling and receipt. However, there are exceptions to this rule, such as weird.

Create a program that processes a file containing lines of text. Each line in the file may contain many words (or no words at all). Any words that do not contain an E adjacent to an I should be ignored. Words that contain an adjacent E and I (in either order) should be examined to determine whether or not they follow the "I before E except after C" rule. Construct and report two lists: One that contains all of the words that follow the rule, and one that contains all of the words that violate the rule. Neither of your lists should contain any repeated values. Report the lengths of the lists at the end of your program so that one can easily determine the proportion of the words in the file that respect the "I before E except after C" rule.

Exercise 161: What's that Element Again?

(*59 Lines*)

Write a program that reads a file containing information about chemical elements and stores it in one or more appropriate data structures. Then your program should read and process input from the user. If the user enters an integer then your program should display the symbol and name of the element with the number of protons entered. If the user enters a non-integer value then your program should display the number of protons for the element with that name or symbol. Your program should display an appropriate error message if no element exists for the name, symbol or number of protons entered. Continue to read input from the user until a blank line is entered.

Exercise 162: A Book with No E...

(*Solved, 50 Lines*)

The novel "Gadsby" is over 50,000 words in length. While 50,000 words is not normally remarkable for a novel, it is in this case because none of the words in the book use the letter E. This is particularly noteworthy when one considers that E is the most common letter in English.

Write a program that reads a list of words from a file and determines what proportion of the words use each letter of the alphabet. Display this result for all 26 letters and include an additional message that identifies the letter that is used in the smallest proportion of the words. Your program should ignore any punctuation marks that are present in the file and it should treat uppercase and lowercase letters as equivalent.

A lipogram is a written work that does not use a particular letter (or group of letters). The letter that is avoided is often a common vowel, though it does not have to be. For example, The Raven by Edgar Allan Poe is a poem of more than 1,000 words that does not use the letter Z, and as such, is a lipogram. "La Disparition" is another example of a lipogrammatic novel. Both the original novel (written in French), and its English translation, "A Void", occupy approximately 300 pages without using the letter E, other than in the author's name.

Exercise 163: Names that Reached Number One

(*Solved, 54 Lines*)

The baby names data set consists of over 200 files. Each file contains a list of 100 names, along with the number of times each name was used. Entries in the files are ordered from most frequently used to least frequently used. There are two files for each year: one containing names used for girls and the other containing names used for boys. The data set includes data for every year from 1900 to 2012.

Write a program that reads every file in the data set and identifies all of the names that were most popular in at least one year. Your program should output two lists: one containing the most popular names for boys and the other containing the most popular names for girls. Neither of your lists should include any repeated values.

Exercise 164: Gender Neutral Names

(*56 Lines*)

Some names, like Ben, Jonathan and Andrew are normally only used for boys while names like Rebecca and Flora are normally only used for girls. Other names, like Chris and Alex, may be used for both boys and girls.

Write a program that determines and displays all of the baby names that were used for both boys and girls in a year specified by the user. Your program should generate an appropriate message if there were no gender neutral names in the selected year. Display an appropriate error message if you do not have data for the year requested by the user. Additional details about the baby names data set are included in Exercise 163.

Exercise 165: Most Births in a given Time Period

(*76 Lines*)

Write a program that uses the baby names data set described in Exercise 163 to determine which names were used most often within a time period. Have the user supply the first and last years of the range to analyze. Display the boy's name and the girl's name given to the most children during the indicated years.

Exercise 166: Distinct Names

(*41 Lines*)

In this exercise, you will create a program that reads every file in the baby names data set described in Exercise 163. As your program reads the files, it should keep track of every distinct name used for a boy and every distinct name used for a girl. Then your program should output each of these lists of names. Neither of the lists should contain any repeated values.

Exercise 167: Spell Checker

(*Solved, 58 Lines*)

A spell checker can be a helpful tool for people who struggle to spell words correctly. In this exercise, you will write a program that reads a file and displays all of the words in it that are misspelled. Misspelled words will be identified by checking each word in the file against a list of known words. Any words in the user's file that do not appear in the list of known words will be reported as spelling mistakes.

The user will provide the name of the file to check for spelling mistakes as a command line argument. Your program should display an appropriate error message if the command line argument is missing. An error message should also be displayed if your program is unable to open the user's file. Use your solution to Exercise 117 when creating your solution to this exercise so that words followed by a comma, period or other punctuation mark are not reported as spelling mistakes. Ignore the capitalization of the words when checking their spelling.

> Hint: While you could load all of the English words from the words data set into a list, searching a list is slow if you use Python's `in` operator. It is much faster to check if a key is present in a dictionary, or if a value is present in a set. If you use a dictionary, the words will be the keys. The values can be the integer 0 (or any other value) because the values will never be used.

Exercise 168: Repeated Words

(*61 Lines*)

Spelling mistakes are only one of many different kinds of errors that might appear in a written work. Another error that is common for some writers is a repeated word. For example, an author might inadvertently duplicate a word, as shown in the following sentence:

```
At least one value must be entered
entered in order to compute the average.
```

Some word processors will detect this error and identify it when a spelling or grammar check is performed.

In this exercise you will write a program that detects repeated words in a text file. When a repeated word is found your program should display a message that contains the line number and the repeated word. Ensure that your program correctly handles the case where the same word appears at the end of one line and the beginning of the following line, as shown in the previous example. The name of the file to examine will be provided as the program's only command line argument. Display an appropriate error message if the user fails to provide a command line argument, or if an error occurs while processing the file.

Exercise 169: Redacting Text in a File

(*Solved, 52 Lines*)

Sensitive information is often removed, or redacted, from documents before they are released to the public. When the documents are released it is common for the redacted text to be replaced with black bars.

In this exercise you will write a program that redacts all occurrences of sensitive words in a text file by replacing them with asterisks. Your program should redact sensitive words wherever they occur, even if they occur in the middle of another word. The list of sensitive words will be provided in a separate text file. Save the redacted version of the original text in a new file. The names of the original text file, sensitive words file, and redacted file will all be provided by the user.

> You may find the `replace` method for strings helpful when completing this exercise. Information about the `replace` method can be found on the Internet.

For an added challenge, extend your program so that it redacts words in a case insensitive manner. For example, if `exam` appears in the list of sensitive words then redact `exam`, `Exam`, `ExaM` and `EXAM`, among other possible capitalizations.

Exercise 170: Missing Comments

(*Solved, 48 Lines*)

When one writes a function, it is generally a good idea to include a comment that outlines the function's purpose, its parameters and its return value. However, sometimes comments are forgotten, or left out by well-intentioned programmers that plan to write them later but then never get around to it.

Create a Python program that reads one or more Python source files and identifies functions that are not immediately preceded by a comment. For the purposes of this exercise, assume that any line that begins with `def`, followed by a space, is the beginning of a function definition. Assume that the comment character, `#`, will be the first character on the previous line when the function has a comment. Display the names of all of the functions that are missing comments, along with the file name and line number where the function definition is located.

The user will provide the names of one or more Python files as command line arguments, all of which should be analyzed by your program. An appropriate error message should be displayed for any files that do not exist or cannot be opened. Then your program should process the remaining files.

Exercise 171: Consistent Line Lengths

(*45 Lines*)

While 80 characters is a common width for a terminal window, some terminals are narrow or wider. This can present challenges when displaying documents containing paragraphs of text. The lines might be too long and wrap, making them difficult to read, or they might be too short and fail to make good use of the available space.

Write a program that opens a file and displays it so that each line is as full as possible. If you read a line that is too long then your program should break it up into words and add them to the current line until it is full. Then your program should start a new line and display the remaining words. Similarly, if you read a line that is too short then you will need to use words from the next line of the file to finish filling the current line of output. For example, consider a file containing the following lines from "Alice's Adventures in Wonderland":

```
Alice was
beginning to get very tired of sitting by her
sister
on the bank, and of having nothing to do: once
or twice she had peeped into the book her sister
was reading, but it had
no
pictures or conversations in it,"and what is
the use of a book," thought Alice, "without
pictures or conversations?"
```

When formatted for a line length of 50 characters, it should be displayed as:

```
Alice was beginning to get very tired of sitting
by her sister on the bank, and of having nothing
to do: once or twice she had peeped into the book
her sister was reading, but it had no pictures or
conversations in it, "and what is the use of a
book," thought Alice, "without pictures or
conversations?"
```

Ensure that your program works correctly for files containing multiple paragraphs of text. You can detect the end of one paragraph and the beginning of the next by looking for lines that are empty once the end of line marker has been removed.

> Hint: Use a constant to represent the maximum line length. This will make it easier to update the program when a different line length is needed.

Exercise 172: Words with Six Vowels in Order

(*56 Lines*)

There is at least one word in the English language that contains each of the vowels A, E, I, O, U and Y exactly once and in order. Write a program that searches a file containing a list of words and displays all of the words that meet this constraint. The user will provide the name of the file that will be searched. Display an appropriate error message and exit the program if the user provides an invalid file name, or if something else goes wrong while your program is searching for words with six vowels in order.

8 Recursion

In Chap. 4 we explored many aspects of functions, including functions that call other functions. A closely related topic that we did not consider previously is whether or not a function can call itself. It turns out that this is, in fact, possible, and that it is a powerful technique for solving some problems.

A definition that describes something in terms of itself is *recursive*. To be useful a recursive definition must describe whatever is being defined in terms of a different (typically a smaller or simpler) version of itself. A definition that defines something in terms of the same version of itself, while recursive, is not particularly useful because the definition is circular. A useful recursive definition must make progress toward a version of the problem with a known solution.

Any function that calls itself is recursive because the function's body (its definition) includes a call to the function that is being defined. In order to reach a solution a recursive function must have at least one case where it is able to produce the required result without calling itself. This is referred to as the *base case*. Cases where the function calls itself are referred to as *recursive cases*. Our examination of recursion will continue with three examples.

8.1 Summing Integers

Consider the problem of computing the sum of all the integers from 0 up to and including some positive integer, `n`. This can be accomplished using a loop or a formula. It can also be performed recursively. The simplest case is when `n` is 0. In this case the answer is known to be 0 and that answer can be returned without using another version of the problem. As a result, this is the base case.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_8

For any positive integer, n, the sum of the values from 0 up to and including n can be computed by adding n to the sum of the numbers from 0 up to and including n - 1. This description is recursive because the sum of n integers is expressed as a smaller version of the same problem (summing the numbers from 0 up to and including n - 1), plus a small amount of additional work (adding n to that sum). Each time this recursive definition is applied it makes progress toward the base case (when n is 0). When the base case is reached no further recursion is performed. This allows the calculation to complete and the result to be returned.

The following program implements the recursive algorithm described in the previous paragraphs to compute the sum of the integers from 0 up to and including a positive integer entered by the user. An if statement is used to determine whether to execute the base case or the recursive case. When the base case executes 0 is returned immediately, without making a recursive call. When the recursive case executes the function is called again with a smaller argument (n - 1). Once the recursive call returns, n is added to the returned value. This successfully computes the total of the values from 0 up to and including n. Then this total is returned as the function's result.

```
# Compute the sum of the integers from 0 up to and including n using recursion
# @param n the maximum value to include in the sum
# @return the sum of the integers from 0 up to and including n
def sum_to(n):
  if n <= 0:
    return 0                  # Base case
  else:
    return n + sum_to(n - 1)  # Recursive case

# Compute the sum of the integers from 0 up to and including a value entered by the user
num = int(input("Enter a non-negative integer: "))
total = sum_to(num)
print("The total of the integers from 0 up to and including", \
      num, "is", total)
```

Consider what happens if the user enters 2 when the program is run. This value is read from the keyboard and converted into an integer. Then sum_to is called with 2 as its argument. The if statement's condition evaluates to True so its body executes and sum_to is called recursively with its argument equal to 1. This recursive call must complete before the the copy of sum_to where n is equal to 2 can compute and return its result.

Executing the recursive call to sum_to where n is equal to 1 causes another recursive call to sum_to with n equal to 0. Once that recursive call begins to execute there are three copies of sum_to executing with argument values 2, 1 and 0. The copy of sum_to where n is equal to 2 is waiting for the copy where n is equal to 1 to complete before it can return its result, and the copy where n is equal to 1 is waiting for the copy where n is equal to 0 to complete before it can return its result. While all of the functions have the same name, each copy of the function is entirely separate from all of the other copies.

When sum_to executes with n equal to 0 the base case is executed. It immediately returns 0. This allows the version of sum_to where n is equal to 1 to progress by adding 1 to the 0 returned by the recursive call. Then the total, which is 1, is returned by the call where `n` is equal to 1. Execution continues with the version of `sum_to` where `n` is equal to 2. It adds `n`, which is 2, to the 1 returned by the recursive call, and 3 is returned and stored into `total`. Finally, the total is displayed and the program terminates.

8.2 Fibonacci Numbers

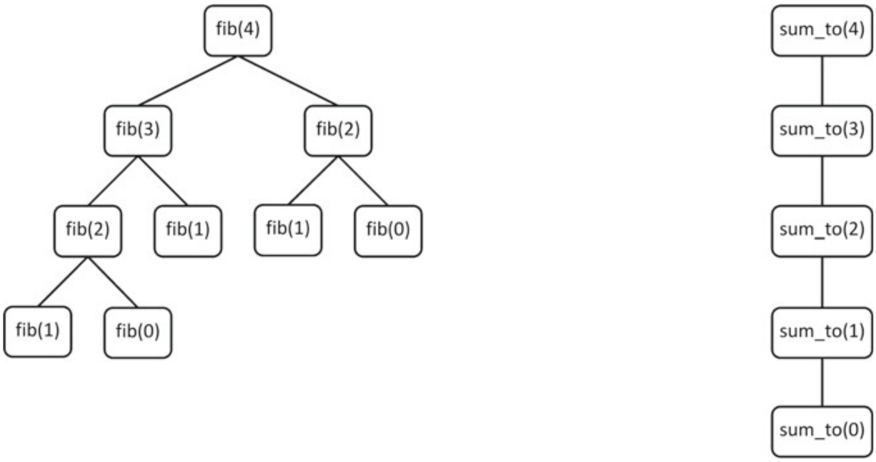
The Fibonacci numbers are a sequence of integers that begin with 0 and 1. Each subsequent number in the sequence is the sum of its two immediate predecessors. As a result, the first 10 numbers in the Fibonacci sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21 and 34. Numbers in the Fibonacci sequence are commonly denoted by F_n, where n is a non-negative integer identifying the number's index within the sequence (starting from 0).

Numbers in the Fibonacci sequence, beyond the first two, can be computed using the formula $F_n = F_{n-1} + F_{n-2}$. This definition is recursive because a larger Fibonacci number is computed using two smaller Fibonacci numbers. The first two numbers in the sequence, F_0 and F_1, are the base cases because they have known values that are not computed recursively. A program that implements this recursive formula for computing Fibonacci numbers is shown below. It computes and displays the value of F_n for some value of n entered by the user.

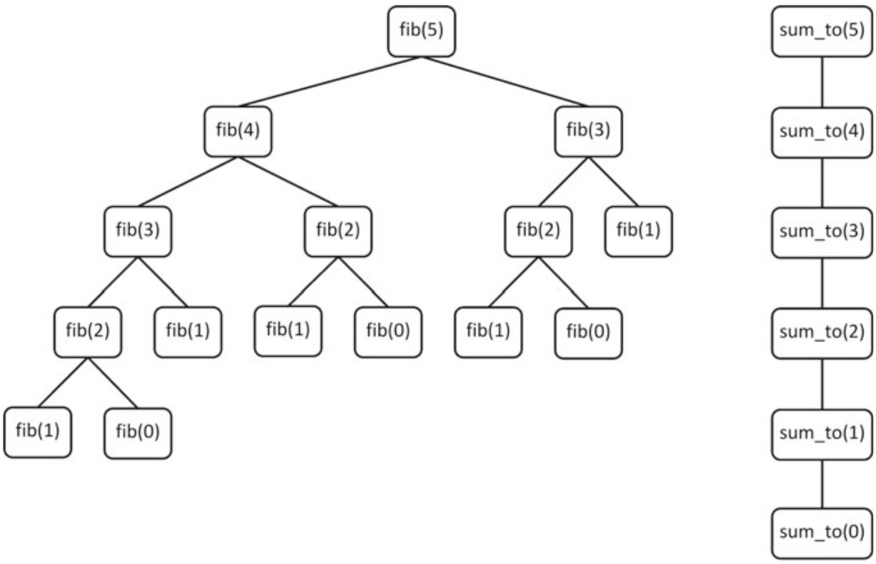
```
# Compute the nth Fibonacci number using recursion
# @param n the index of the Fibonacci number to compute
# @return the nth Fibonacci number
def fib(n):
  # Base cases
  if n == 0:
    return 0
  if n == 1:
    return 1

  # Recursive case
  return fib(n-1) + fib(n-2)

# Compute the Fibonacci number requested by the user
n = int(input("Enter a non-negative integer: "))
print("fib(%d) is %d." % (n, fib(n)))
```

This recursive algorithm for computing Fibonacci numbers is compact, but it is slow, even when working with fairly modest values. While computing `fib(35)` will return quickly on a modern machine, computing `fib(70)` will take years to complete. As a result, larger Fibonacci numbers are normally computed using a loop or a formula.

Based on the performance of our Fibonacci numbers program you might be tempted to conclude that recursive solutions are too slow to be useful. While that is true in this particular situation, it is not true in general. Our previous program that



(a) The Function Calls Used to Compute `fib(4)` and `sum_to(4)`



(b) The Function Calls Used to Compute `fib(5)` and `sum_to(5)`

Fig. 8.1 A comparison of the function calls for `fib` and `sum_to`

summed integers ran quickly even for larger values, and there are some problems that have very efficient recursive algorithms, such as Euclid's algorithm for computing the greatest common divisor of two integers which is described in Exercise 174.

Figure 8.1 illustrates the recursive calls made when computing F_4 and F_5, as well as the recursive calls made to evaluate `sum_to(4)` and `sum_to(5)`. Comparing the function calls made to compute these results for different input values illustrates the difference in efficiency for these problems.

When the argument passed to `sum_to` increases from 4 to 5 the number of function calls also increases from 4 to 5. More generally, when the argument passed to `sum_to` increases by 1 the number of function calls also increases by 1. This is referred to as linear growth because the number of recursive calls is directly proportional to the value of the argument provided when the function is first called.

In contrast, when the argument passed to `fib` increases from 4 to 5 the number of function calls increases from 9 to 15. More generally, when the position of the Fibonacci number being computed increases by 1, the number of recursive calls (nearly) doubles. This is referred to as exponential growth. Exponential growth makes it impossible (in any practical sense) to calculate the result for large values because repeatedly doubling the time needed for the computation quickly results in a running time that is simply too long to be useful.

8.3 Counting Characters

Recursion can be used to solve any problem that can be expressed in terms of itself. It is not restricted to problems that operate on integers. For example, consider the problem of counting the number of occurrences of a particular character, `ch`, within a string, `s`. A recursive function for solving this problem can be written that takes `s` and `ch` as arguments and returns the number of times that `ch` occurs in `s` as its result.

The base case for this problem is `s` being the empty string. Since an empty string does not contain any characters it must contain 0 occurrences of `ch`. As a result, the function can return 0 in this case without making a recursive call.

The number of occurrences of `ch` in a longer string can be determined in the following recursive manner. To help simplify the description of the recursive case we will define the tail of `s` to be all of the characters in `s` except for the first character. The tail of a string containing only one character is the empty string.

If the first character in `s` is `ch` then the number of occurrences of `ch` in `s` is one plus the number of occurrences of `ch` in the tail of `s`. Otherwise the number of occurrences of `ch` in `s` is the number of occurrences of `ch` in the tail of `s`. This definition makes progress toward the base case (when `s` is the empty string) because the tail of `s` is always shorter than `s`. A program that implements this recursive algorithm is shown below.

```
# Count the number of times a particular character is present in a string
# @param s the string in which the characters are counted
# @param ch the character to count
# @return the number of occurrences of ch in s
def count(s, ch):
  if s == "":
    return 0   # Base case

  # Compute the tail of s
  tail = s[1 : len(s)]
```

```
  # Recursive cases
  if ch == s[0]:
    return 1 + count(tail, ch)
  else:
    return count(tail, ch)

# Count the number of times a character entered by the user occurs in a string entered
# by the user
s = input("Enter a string: ")
ch = input("Enter the character to count: ")
print("'%s' occurs %d times in '%s'" % (ch, count(s, ch), s))
```

8.4 Exercises

A recursive function is a function that calls itself. Such functions normally include one or more base cases and one or more recursive cases. When the base case executes the result for the function is computed without making a recursive call. Recursive cases compute their result by making one or more recursive calls, typically to a smaller or simpler version of the problem. All of the exercises in this chapter should be solved by writing one or more recursive functions. Each of these functions will call itself, and may also make use of any of the Python features that were discussed in the previous chapters.

Exercise 173: Total the Values

(*Solved, 29 Lines*)

Write a program that reads values from the user until a blank line is entered. Display the total of all of the values entered by the user (or 0.0 if the first value entered is a blank line). Complete this task using recursion. Your program may not use any loops.

> Hint: The body of your recursive function will need to read one value from the user, and then determine whether or not to make a recursive call. Your function does not need to take any arguments, but it will need to return a numeric result.

Exercise 174: Greatest Common Divisor

(*24 Lines*)

Euclid was a Greek mathematician who lived approximately 2,300 years ago. His algorithm for computing the greatest common divisor of two positive integers, `a` and `b`, is both efficient and recursive. It is outlined below:

If b is 0 **then**
 Return a
Else
 Set c equal to the remainder when a is divided by b
 Return the greatest common divisor of b and c

Write a program that implements Euclid's algorithm and uses it to determine the greatest common divisor of two integers entered by the user. Test your program with some very large integers. The result will be computed quickly, even for huge numbers consisting of hundreds of digits, because Euclid's algorithm is extremely efficient.

Exercise 175: Recursive Decimal to Binary

(*34 Lines*)

In Exercise 82 you wrote a program that used a loop to convert a decimal number to its binary representation. In this exercise you will perform the same task using recursion.

Write a recursive function that converts a non-negative decimal number to binary. Treat 0 and 1 as base cases which return a string containing the appropriate digit. For all other positive integers, n, you should compute the next digit using the remainder operator and then make a recursive call to compute the digits of n // 2. Finally, you should concatenate the result of the recursive call (which will be a string) and the next digit (which you will need to convert to a string) and return this string as the result of the function.

Write a main program that uses your recursive function to convert a non-negative integer entered by the user from decimal to binary. Your program should display an appropriate error message if the user enters a negative value.

Exercise 176: The NATO Phonetic Alphabet

(*33 Lines*)

A spelling alphabet is a set of words, each of which stands for one of the 26 letters in the alphabet. While many letters are easily misheard over a low quality or noisy communication channel, the words used to represent the letters in a spelling alphabet are generally chosen so that each sounds distinct and is difficult to confuse with any other. The NATO phonetic alphabet is a widely used spelling alphabet. Each letter and its associated word is shown in Table 8.1.

Write a program that reads a word from the user and then displays its phonetic spelling. For example, if the user enters `Hello` then the program should output `Hotel Echo Lima Lima Oscar`. Your program should use a recursive function to perform this task. Do not use a loop anywhere in your solution. Any non-letter characters entered by the user should be ignored.

Table 8.1 NATO phonetic alphabet

Letter	Word	Letter	Word	Letter	Word
A	Alpha	J	Juliet	S	Sierra
B	Bravo	K	Kilo	T	Tango
C	Charlie	L	Lima	U	Uniform
D	Delta	M	Mike	V	Victor
E	Echo	N	November	W	Whiskey
F	Foxtrot	O	Oscar	X	Xray
G	Golf	P	Papa	Y	Yankee
H	Hotel	Q	Quebec	Z	Zulu
I	India	R	Romeo		

Exercise 177: Roman Numerals

(*25 Lines*)

As the name implies, Roman numerals were developed in ancient Rome. Even after the Roman empire fell, its numerals continued to be widely used in Europe until the late middle ages, and its numerals are still used in limited circumstances today.

Roman numerals are constructed from the letters M, D, C, L, X, V and I which represent 1000, 500, 100, 50, 10, 5 and 1 respectively. The numerals are generally written from largest value to smallest value. When this occurs the value of the number is the sum of the values of all of its numerals. If a smaller value precedes a larger value then the smaller value is subtracted from the larger value that it immediately precedes, and that difference is added to the value of the number.[1]

Create a recursive function that converts a Roman numeral to an integer. Your function should process one or two characters at the beginning of the string, and then call itself recursively on all of the unprocessed characters. Use an empty string, which has the value 0, for the base case. In addition, write a main program that reads a Roman numeral from the user and displays its value. You can assume that the value entered by the user is valid. Your program does not need to do any error checking.

Exercise 178: Recursive Palindrome

(*Solved, 30 Lines*)

The notion of a palindrome was introduced previously in Exercise 75. In this exercise you will write a recursive function that determines whether or not a string is a palindrome. The empty string is a palindrome, as is any string containing only one

[1]Only C, X and I are used in a subtractive manner. The numeral that a C, X or I precedes must have a value that is no more than 10 times the value being subtracted. As such, I can precede V or X, but it cannot precede L, C, D or M. This means, for example, that 99 must be represented by XCIX rather than by IC.

character. Any longer string is a palindrome if its first and last characters match, and if the string formed by removing the first and last characters is also a palindrome.

Write a main program that reads a string from the user and uses your recursive function to determine whether or not it is a palindrome. Then your program should display an appropriate message for the user.

Exercise 179: Recursive Square Root

(*20 Lines*)

Exercise 74 explored how iteration can be used to compute the square root of a number. In that exercise a better approximation of the square root was generated with each additional loop iteration. In this exercise you will use the same approximation strategy, but you will use recursion instead of a loop.

Create a square root function with two parameters. The first parameter, n, will be the number for which the square root is being computed. The second parameter, *guess*, will be the current guess for the square root. The *guess* parameter should have a default value of 1.0. Do not provide a default value for the first parameter.

Your square root function will be recursive. The base case occurs when $guess^2$ is within 10^{-12} of n. In this case your function should return $guess$ because it is close enough to the square root of n. Otherwise your function should return the result of calling itself recursively with n as the first parameter and $\frac{guess+\frac{n}{guess}}{2}$ as the second parameter.

Write a main program that demonstrate your square root function by computing the square root of several different values. When you call your square root function from the main program you should only pass one parameter to it so that the default value is used for $guess$.

Exercise 180: String Edit Distance

(*Solved, 43 Lines*)

The edit distance between two strings is a measure of their similarity. The smaller the edit distance, the more similar the strings are with regard to the minimum number of insert, delete and substitute operations needed to transform one string into the other.

Consider the strings `kitten` and `sitting`. The first string can be transformed into the second string with the following operations: Substitute the `k` with an `s`, substitute the `e` with an `i`, and insert a `g` at the end of the string. This is the smallest number of operations that can be performed to transform `kitten` into `sitting`. As a result, the edit distance is 3.

Write a recursive function that computes the edit distance between two strings. Use the following algorithm:

Let *s* and *t* be the strings
If the length of *s* is 0 **then**
  Return the length of *t*
Else if the length of *t* is 0 **then**
  Return the length of *s*
Else
  Set *cost* to 0
  If the last character in *s* does not equal the last character in *t* **then**
    Set *cost* to 1
  Set *d1* equal to the edit distance between all characters except the last one in *s*, and all characters in *t*, plus 1
  Set *d2* equal to the edit distance between all characters in *s*, and all characters except the last one in *t*, plus 1
  Set *d3* equal to the edit distance between all characters except the last one in *s*, and all characters except the last one in *t*, plus *cost*
  Return the minimum of *d1*, *d2* and *d3*

Use your recursive function to write a program that reads two strings from the user and displays the edit distance between them.

Exercise 181: Possible Change

(*41 Lines*)

Create a program that determines whether or not it is possible to construct a particular total using a specific number of coins. For example, it is possible to have a total of \$1.00 using four coins if they are all quarters. However, there is no way to have a total of \$1.00 using 5 coins. Yet it is possible to have \$1.00 using 6 coins by using 3 quarters, 2 dimes and a nickel. Similarly, a total of \$1.25 can be formed using 5 coins or 8 coins, but a total of \$1.25 cannot be formed using 4, 6 or 7 coins.

Your program should read both the dollar amount and the number of coins from the user. Then it should display a clear message indicating whether or not the entered dollar amount can be formed using the number of coins indicated. Assume the existence of quarters, dimes, nickels and pennies when completing this problem. Your solution must use recursion. It cannot contain any loops.

Exercise 182: Spelling with Element Symbols

(*67 Lines*)

Each chemical element has a standard symbol that is one, two or three letters long. One game that some people like to play is to determine whether or not a word can be spelled using only element symbols. For example, silicon can be spelled using the symbols Si, Li, C, O and N. However, hydrogen cannot be spelled with any combination of element symbols.

Write a recursive function that determines whether or not a word can be spelled using only element symbols. Your function will require two parameters: the word that you are trying to spell and a list of the symbols that can be used. It will return a string containing the symbols used to achieve the spelling as its result, or an empty string if no spelling exists. Capitalization should be ignored when your function searches for a spelling.

Create a program that uses your function to find and display all of the element names that can be spelled using only element symbols. Display the names of the elements along with the sequences of symbols. For example, one line of your output will be:

```
Silver can be spelled as SiLvEr
```

Your program will use the elements data set, which can be downloaded from the author's website. This data set includes the names and symbols of all 118 chemical elements.

Exercise 183: Element Sequences

(*Solved, 81 Lines*)

Some people like to play a game that constructs a sequence of chemical elements where each element in the sequence begins with the last letter of its predecessor. For example, if a sequence begins with Hydrogen, then the next element must be an element that begins with N, such as Nickel. The element following Nickel must begin with L, such as Lithium. The element sequence that is constructed cannot contain any duplicates. When played alone the goal of the game is to constructed the longest possible sequence of elements. When played with two players, the goal is to select an element that leaves your opponent without an option to add to the sequence.

Write a program that reads the name of an element from the user and uses a recursive function to find the longest sequence of elements that begins with that value. Display the sequence once it has been computed. Ensure that your program responds in a reasonable way if the user does not enter a valid element name.

> Hint: It may take your program up to two minutes to find the longest sequence for some elements. As a result, you might want to use elements like Molybdenum and Magnesium as your first test cases. Each has a longest sequence that is only 8 elements long which your program should find in a fraction of a second.

Exercise 184: Flatten a List

(*Solved, 33 Lines*)

Python's lists can contain other lists. When one list occurs inside another the inner list is said to be nested inside the outer list. Each of the inner lists nested within the outer list may also contain nested lists, and those lists may contain additional nested lists to any depth. For example, the following list includes elements that are nested at several different depths: `[1, [2, 3], [4, [5, [6, 7]]], [[[8], 9], [10]]]`.

Lists that contain multiple levels of nesting can be useful when describing complex relationships between values, but such lists can also make performing some operations on those values difficult because the values are nested at different levels. Flattening a list is the process of converting a list that may contain multiple levels of nested lists into a list that contains all of the same elements without any nesting. For example, flattening the list from the previous paragraph results in `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. The following recursive algorithm can be used to flatten a list named *data*:

If *data* is empty **then**
  Return the empty list
If the first element in *data* is a list **then**
  Set *l*1 to the result of flattening the first element in *data*
  Set *l*2 to the result of flattening all of the elements in *data*, except the first
  Return the concatenation of *l*1 and *l*2
If the first element in *data* is not a list **then**
  Set *l*1 to a list containing only the first element in *data*
  Set *l*2 to the result of flattening all of the elements in *data*, except the first
  Return the concatenation of *l*1 and *l*2

Write a function that implements the recursive flattening algorithm described previously. Your function will take one argument, which is the list to flatten, and it will return one result, which is the flattened list. Include a main program that demonstrates that your function successfully flattens the list shown earlier in this problem, as well as several others.

> Hint: Python includes a function named `type` which returns the type of its only argument. Information about using this function to determine whether or not a variable is a list can be found online.

Exercise 185: Run-Length Decoding

(*33 Lines*)

Run-length encoding is a simple data compression technique that can be effective when repeated values occur at adjacent positions within a list. Compression is achieved by replacing groups of repeated values with one copy of the value, followed by the number of times that it should be repeated. For example, the list `["A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "B", "B", "B", "B", "A", "A", "A", "A", "A", "A", "B"]` would be compressed as `["A", 12, "B", 4, "A", 6, "B", 1]`. Decompression is performed by replicating each value in the list the number of times indicated.

Write a recursive function that decompresses a run-length encoded list. Your recursive function will take a run-length compressed list as its only argument. It will return the decompressed list as its only result. Create a main program that displays a run-length encoded list and the result of decoding it.

Exercise 186: Run-Length Encoding

(*Solved, 38 Lines*)

Write a recursive function that implements the run-length compression technique described in Exercise 185. Your function will take a list or a string as its only argument. It should return the run-length compressed list as its only result. Include a main program that reads a string from the user, compresses it, and displays the run-length encoded result.

> Hint: You may want to include a loop inside the body of your recursive function.

Part II

Solutions

9 Solutions to the Introduction to Programming Exercises

Solution to Exercise 1: Mailing Address

```
##
#  Display a person's complete mailing address.
#
print("Ben Stephenson")
print("Department of Computer Science")
print("University of Calgary")
print("2500 University Drive NW")
print("Calgary, Alberta T2N 1N4")
print("Canada")
```

Solution to Exercise 3: Area of a Room

```
##
#  Compute the area of a room.
#

# Read the dimensions from the user
length = float(input("Enter the length of the room in feet: "))
width = float(input("Enter the width of the room in feet: "))

# Compute the area of the room
area = length * width

# Display the result
print("The area of the room is", area, "square feet")
```

> The `float` function is used to convert the user's input into a number.

> In Python, multiplication is performed using the $*$ operator.

Solution to Exercise 4: Area of a Field

```
##
#  Compute the area of a field, reporting the result in acres.
#
SQFT_PER_ACRE = 43560
```


B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_9

```
# Read the dimensions from the user
length = float(input("Enter the length of the field in feet: "))
width = float(input("Enter the width of the field in feet: "))

# Compute the area in acres
acres = length * width / SQFT_PER_ACRE

# Display the result
print("The area of the field is", acres, "acres")
```

Solution to Exercise 5: Bottle Deposits

```
##
#  Compute the refund amount for a collection of bottles.
#
LESS_DEPOSIT = 0.10
MORE_DEPOSIT = 0.25

# Read the number of containers of each size from the user
less = int(input("How many containers 1 litre or less? "))
more = int(input("How many containers more than 1 litre? "))

# Compute the refund amount
refund = less * LESS_DEPOSIT + more * MORE_DEPOSIT

# Display the result
print("Your total refund will be $%.2f." % refund)
```

The `%.2f` format specifier indicates that a value should be formatted as a floating-point number with 2 digits to the right of the decimal point.

Solution to Exercise 6: Tax and Tip

```
##
#  Compute the tax and tip for a restaurant meal.
#
TAX_RATE = 0.05
TIP_RATE = 0.18
```

My local tax rate is 5%. In Python we represent 5% and 18% as 0.05 and 0.18 respectively.

```
# Read the cost of the meal from the user
cost = float(input("Enter the cost of the meal: "))

# Compute the tax and the tip
tax = cost * TAX_RATE
tip = cost * TIP_RATE
total = cost + tax + tip
```

```
# Display the result
print("The tax is %.2f and the tip is %.2f, making the", \
      "total %.2f" % (tax, tip, total))
```

The \ at the end of the line is called the line continuation character. It tells Python that the statement continues on the next line. Do not include any spaces or tabs after the \ character.

Solution to Exercise 7: Sum of the First *n* Positive Integers

```
##
# Compute the sum of the first n positive integers.
#

# Read the value of n from the user
n = int(input("Enter a positive integer: "))

# Compute the sum
sm = n * (n + 1) / 2
```

Python includes a built-in function named `sum`. As a result, we will use a different name for our variable.

```
# Display the result
print("The sum of the first", n, "positive integers is", sm)
```

Solution to Exercise 10: Arithmetic

```
##
# Demonstrate Python's mathematical operators and its math module.
#
from math import log10
```

We must import the `log10` function from the `math` module before we call it. Import statements normally appear at the top of the file.

```
# Read the input values from the user
a = int(input("Enter the value of a: "))
b = int(input("Enter the value of b: "))

# Compute and display the sum, difference, product, quotient and remainder
print(a, "+", b, "is", a + b)
print(a, "-", b, "is", a - b)
print(a, "*", b, "is", a * b)
print(a, "/", b, "is", a / b)
print(a, "%", b, "is", a % b)
```

The remainder is computed using the `%` operator.

```
# Compute the logarithm and the power
print("The base 10 logarithm of", a, "is", log10(a))
print(a, "^", b, "is", a**b)
```

Solution to Exercise 13: Making Change

```
##
#  Compute the minimum collection of coins needed to represent a number of cents.
#
CENTS_PER_TOONIE = 200
CENTS_PER_LOONIE = 100
CENTS_PER_QUARTER = 25
CENTS_PER_DIME =    10
CENTS_PER_NICKEL =   5

# Read the number of cents from the user
cents = int(input("Enter the number of cents: "))

# Determine the number of toonies by performing an integer division by 200.  Then compute
# the amount of change that still needs to be considered by computing the remainder after
# dividing by 200.
print(" ", cents // CENTS_PER_TOONIE, "toonies")
cents = cents % CENTS_PER_TOONIE
```

Floor division, which ensures that the result of the division is an integer by rounding down, is performed using the // operator.

```
# Repeat the process for loonies, quarters, dimes, and nickels
print(" ", cents // CENTS_PER_LOONIE, "loonies")
cents = cents % CENTS_PER_LOONIE

print(" ", cents // CENTS_PER_QUARTER, "quarters")
cents = cents % CENTS_PER_QUARTER

print(" ", cents // CENTS_PER_DIME, "dimes")
cents = cents % CENTS_PER_DIME

print(" ", cents // CENTS_PER_NICKEL, "nickels")
cents = cents % CENTS_PER_NICKEL

# Display the number of pennies
print(" ", cents, "pennies")
```

Solution to Exercise 14: Height Units

```
##
#  Convert a height in feet and inches to centimeters.
#
IN_PER_FT = 12
CM_PER_IN = 2.54
```

```
# Read the height from the user
print("Enter your height:")
feet = int(input("  Number of feet: "))
inches = int(input("  Number of inches: "))

# Compute the equivalent number of centimeters
cm = (feet * IN_PER_FT + inches) * CM_PER_IN

# Display the result
print("Your height in centimeters is:", cm)
```

Solution to Exercise 17: Heat Capacity

```
##
#  Compute the amount of energy needed to heat a volume of water, and the cost of doing so.
#

# Define constants for the specific heat capacity of water and the price of electricity
WATER_HEAT_CAPACITY = 4.186
ELECTRICITY_PRICE = 8.9
J_TO_KWH = 2.777e-7
```

Python allows numbers to be written in scientific notation by placing the coefficient to the left of an e and the exponent to its right. As a result, $2.777 * 10^{-7}$ is written as 2.777e-7.

```
# Read the volume and temperature increase from the user
volume = float(input("Amount of water in milliliters: "))
d_temp = float(input("Temperature increase (degrees Celsius): "))
```

Because water has a density of 1 gram per milliliter grams and milliliters can be used interchangeably. Prompting the user for milliliters makes the program easier to use because most people think about the volume of water in a coffee cup, not its mass.

```
# Compute the energy in Joules
q = volume * d_temp * WATER_HEAT_CAPACITY

# Display the result in Joules
print("That will require %d Joules of energy." % q)

# Compute the cost
kwh = q * J_TO_KWH
cost = kwh * ELECTRICITY_PRICE

# Display the cost
print("That much energy will cost %.2f cents." % cost)
```

Solution to Exercise 19: Free Fall

```
##
#  Compute the speed of an object when it hits the ground after being dropped.
#
from math import sqrt
```

```
# Define a constant for the acceleration due to gravity in m/s**2
GRAVITY = 9.8

# Read the height from which the object is dropped
d = float(input("Height (in meters): "))

# Compute the final velocity
vf = sqrt(2 * GRAVITY * d)
```

The v_i^2 term has not been included in the calculation of v_f because v_i is 0.

```
# Display the result
print("It will hit the ground at %.2f m/s." % vf)
```

Solution to Exercise 23: Area of a Regular Polygon

```
##
# Compute the area of a regular polygon.
#
from math import tan, pi

# Read input from the user
s = float(input("Enter the length of each side: "))
n = int(input("Enter the number of sides: "))
```

We have chosen to convert `n` to an integer (rather than a floating-point number) because a polygon cannot have a fractional number of sides.

```
# Compute the area of the polygon
area = (n * s ** 2) / (4 * tan(pi / n))

# Display the result
print("The area of the polygon is", area)
```

Solution to Exercise 25: Units of Time (Again)

```
##
# Convert a number of seconds to days, hours, minutes and seconds.
#
SECONDS_PER_DAY = 86400
SECONDS_PER_HOUR = 3600
SECONDS_PER_MINUTE = 60

# Read the duration from the user in seconds
seconds = int(input("Enter a number of seconds: "))

# Compute the days, hours, minutes and seconds
days = seconds / SECONDS_PER_DAY
seconds = seconds % SECONDS_PER_DAY
```

```
hours = seconds / SECONDS_PER_HOUR
seconds = seconds % SECONDS_PER_HOUR

minutes = seconds / SECONDS_PER_MINUTE
seconds = seconds % SECONDS_PER_MINUTE

# Display the result with the desired formatting
print("The equivalent duration is", \
      "%d:%02d:%02d:%02d." % (days, hours, minutes, seconds))
```

The `%02d` format specifier tells Python to format an integer using two digits by adding a leading 0 if necessary.

Solution to Exercise 29: Wind Chill

```
##
#  Compute the wind chill index for a given air temperature and wind speed.
#
WC_OFFSET = 13.12
WC_FACTOR1 = 0.6215
WC_FACTOR2 = -11.37
WC_FACTOR3 = 0.3965
WC_EXPONENT = 0.16
```

Computing wind chill requires several numeric constants that were determined by scientists and medical experts.

```
# Read the air temperature and wind speed from the user
temp = float(input("Air temperature (degrees Celsius): "))
speed = float(input("Wind speed (kilometers per hour): "))

# Compute the wind chill index
wci = WC_OFFSET + \
      WC_FACTOR1 * temp + \
      WC_FACTOR2 * speed ** WC_EXPONENT + \
      WC_FACTOR3 * temp * speed ** WC_EXPONENT

# Display the result rounded to the closest integer
print("The wind chill index is", round(wci))
```

Solution to Exercise 33: Sort 3 Integers

```
##
#  Sort 3 values entered by the user into increasing order.
#

#  Read the numbers from the user, naming them a, b and c
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
c = int(input("Enter the third number: "))
```

```
mn = min(a, b, c)          # the minimum value
mx = max(a, b, c)          # the maximum value
md = a + b + c - mn - mx   # the middle value

# Display the result
print("The numbers in sorted order are:")
print("  ", mn)
print("  ", md)
print("  ", mx)
```

Since min and max are the names of functions in Python we should not use those names for variables. Instead, we use variables named mn and mx to hold the minimum and maximum values respectively.

Solution to Exercise 34: Day Old Bread

```
##
# Compute the price of a day old bread order.
#
BREAD_PRICE = 3.49
DISCOUNT_RATE = 0.60

# Read the number of loaves from the user
num_loaves = int(input("Enter the number of day old loaves: "))

# Compute the discount and total price
regular_price = num_loaves * BREAD_PRICE
discount = regular_price * DISCOUNT_RATE
total = regular_price - discount

# Display the result
print("Regular price: %5.2f" % regular_price)
print("Discount:      %5.2f" % discount)
print("Total:         %5.2f" % total)
```

The %5.2f format tells Python that a total of at least 5 spaces should be used to display the number, with 2 digits to the right of the decimal point. This will help keep the columns lined up when the number of digits needed for the regular price, discount and/or total are different.

Solutions to the Decision Making Exercises

10

Solution to Exercise 35: Even or Odd?

```
##
# Determine and display whether an integer entered by the user is even or odd.
#

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

# Determine whether it is even or odd by using the
# modulus (remainder) operator
if num % 2 == 1:
  print(num, "is odd.")
else:
  print(num, "is even.")
```

Dividing an even number by 2 always results in a remainder of 0. Dividing an odd number by 2 always results in a remainder of 1.

Solution to Exercise 37: Vowel or Consonant

```
##
# Determine if a letter is a vowel or a consonant.
#

# Read a letter from the user
letter = input("Enter a letter: ")

# Classify the letter and report the result
if letter == "a" or letter == "e" or \
   letter == "i" or letter == "o" or \
   letter == "u":
  print("It's a vowel.")
elif letter == "y":
  print("Sometimes it's a vowel... Sometimes it's a consonant.")
else:
  print("It's a consonant.")
```

This version of the program only works for lowercase letters. You can add support for uppercase letters by including additional comparisons that follow the same pattern.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_10

Solution to Exercise 38: Name that Shape

```
##
#  Report the name of a shape from its number of sides.
#

# Read the number of sides from the user
nsides = int(input("Enter the number of sides: "))

# Determine the name, leaving it empty if an unsupported number of sides was entered
name = ""
if nsides == 3:
  name = "triangle"
elif nsides == 4:
  name = "quadrilateral"
elif nsides == 5:
  name = "pentagon"
elif nsides == 6:
  name = "hexagon"
elif nsides == 7:
  name = "heptagon"
elif nsides == 8:
  name = "octagon"
elif nsides == 9:
  name = "nonagon"
elif nsides == 10:
  name = "decagon"
```

The empty string is being used as a sentinel value. If the number of sides entered by the user is outside of the supported range then `name` will remain empty, causing an error message to be displayed later in the program.

```
# Display an error message or the name of the polygon
if name == "":
  print("That number of sides is not supported by this program.")
else:
  print("That's a", name)
```

Solution to Exercise 39: Month Name to Number of Days

```
##
#  Display the number of days in a month.
#

# Read the month name from the user
month = input("Enter the name of a month: ")

# Compute the number of days in the month
days = 31
```

Start by assuming that the number of days is 31. Then update the number of days if necessary.

```
if month == "April" or month == "June" or \
   month == "September" or month == "November":
  days = 30
elif month == "February":
  days = "28 or 29"
```

When `month` is February, the value assigned to `days` is a string. This allows us to indicate that February can have either 28 or 29 days.

```
# Display the result
print(month, "has", days, "days in it.")
```

Solution to Exercise 41: Classifying Triangles

```
##
#  Classify a triangle based on the lengths of its sides.
#

# Read the side lengths from the user
side1 = float(input("Enter the length of side 1: "))
side2 = float(input("Enter the length of side 2: "))
side3 = float(input("Enter the length of side 3: "))

# Determine the triangle's type
if side1 == side2 and side2 == side3:
  tri_type = "equilateral"
elif side1 == side2 or side2 == side3 or \
     side3 == side1:
  tri_type = "isosceles"
else:
  tri_type = "scalene"

# Display the triangle's type
print("That's a", tri_type, "triangle")
```

We could also check that `side1` is equal to `side3` as part of the condition for an equilateral triangle. However, that comparison isn't necessary because the == operator is transitive.

Solution to Exercise 42: Note to Frequency

```
##
#  Convert the name of a note to its frequency.
#
C4_FREQ = 261.63
D4_FREQ = 293.66
E4_FREQ = 329.63
F4_FREQ = 349.23
G4_FREQ = 392.00
A4_FREQ = 440.00
B4_FREQ = 493.88

# Read the note name from the user
name = input("Enter the two character note name, such as C4: ")
```

```
# Store the note and its octave in separate variables
note = name[0]
octave = int(name[1])

# Get the frequency of the note, assuming it is in the fourth octave
if note == "C":
  freq = C4_FREQ
elif note == "D":
  freq = D4_FREQ
elif note == "E":
  freq = E4_FREQ
elif note == "F":
  freq = F4_FREQ
elif note == "G":
  freq = G4_FREQ
elif note == "A":
  freq = A4_FREQ
elif note == "B":
  freq = B4_FREQ

# Now adjust the frequency to bring it into the correct octave
freq = freq / 2 ** (4 - octave)

# Display the result
print("The frequency of", name, "is", freq)
```

Solution to Exercise 43: Frequency to Note

```
##
#  Read a frequency from the user and display the note (if any) that it corresponds to.
#
C4_FREQ = 261.63
D4_FREQ = 293.66
E4_FREQ = 329.63
F4_FREQ = 349.23
G4_FREQ = 392.00
A4_FREQ = 440.00
B4_FREQ = 493.88
LIMIT = 1

# Read the frequency from the user
freq = float(input("Enter a frequency (Hz): "))

# Determine the note that corresponds to the entered frequency.  Set note equal to the empty
# string if there isn't a match.
if freq >= C4_FREQ - LIMIT and freq <= C4_FREQ + LIMIT:
  note = "C4"
elif freq >= D4_FREQ - LIMIT and freq <= D4_FREQ + LIMIT:
  note = "D4"
elif freq >= E4_FREQ - LIMIT and freq <= E4_FREQ + LIMIT:
  note = "E4"
elif freq >= F4_FREQ - LIMIT and freq <= F4_FREQ + LIMIT:
  note = "F4"
```

```
elif freq >= G4_FREQ - LIMIT and freq <= G4_FREQ + LIMIT:
  note = "G4"
elif freq >= A4_FREQ - LIMIT and freq <= A4_FREQ + LIMIT:
  note = "A4"
elif freq >= B4_FREQ - LIMIT and freq <= B4_FREQ + LIMIT:
  note = "B4"
else:
  note = ""

# Display the result, or an appropriate error message
if note == "":
  print("There is no note that corresponds to that frequency.")
else:
  print("That frequency is", note)
```

Solution to Exercise 47: Season from Month and Day

```
##
#  Determine and display the season associated with a date.
#

# Read the date from the user
month = input("Enter the name of the month: ")
day = int(input("Enter the day number: "))
```

> This solution to the season identification problem uses several `elif` parts so that the conditions remain as simple as possible. Another way of approaching this problem is to minimize the number of `elif` parts by making the conditions more complex.

```
# Determine the season
if month == "January" or month == "February":
  season = "Winter"
elif month == "March":
  if day < 20:
    season = "Winter"
  else:
    season = "Spring"
elif month == "April" or month == "May":
  season = "Spring"
elif month == "June":
  if day < 21:
    season = "Spring"
  else:
    season = "Summer"
elif month == "July" or month == "August":
  season = "Summer"
elif month == "September":
  if day < 22:
    season = "Summer"
  else:
    season = "Fall"
```

```
elif month == "October" or month == "November":
  season = "Fall"
elif month == "December":
  if day < 21:
    season = "Fall"
  else:
    season = "Winter"

# Display the result
print(month, day, "is in", season)
```

Solution to Exercise 49: Chinese Zodiac

```
##
# Determine the animal associated with a year according to the Chinese zodiac.
#

# Read a year from the user
year = int(input("Enter a year: "))

# Determine the animal associated with that year
if year % 12 == 8:
  animal = "Dragon"
elif year % 12 == 9:
  animal = "Snake"
elif year % 12 == 10:
  animal = "Horse"
elif year % 12 == 11:
  animal = "Sheep"
elif year % 12 == 0:
  animal = "Monkey"
elif year % 12 == 1:
  animal = "Rooster"
elif year % 12 == 2:
  animal = "Dog"
elif year % 12 == 3:
  animal = "Pig"
elif year % 12 == 4:
  animal = "Rat"
elif year % 12 == 5:
  animal = "Ox"
elif year % 12 == 6:
  animal = "Tiger"
elif year % 12 == 7:
  animal = "Hare"

# Report the result
print("%d is the year of the %s." % (year, animal))
```

When multiple items are formatted all of the values are placed inside parentheses on the right side of the `%` operator.

Solution to Exercise 52: Letter Grade to Grade Points

```
##
#  Convert from a letter grade to a number of grade points.
#
A         = 4.0
A_MINUS = 3.7
B_PLUS   = 3.3
B         = 3.0
B_MINUS = 2.7
C_PLUS   = 2.3
C         = 2.0
C_MINUS = 1.7
D_PLUS   = 1.3
D         = 1.0
F         = 0
INVALID = -1

# Read the letter grade from the user
letter = input("Enter a letter grade: ")
letter = letter.upper()
```

The statement `letter = letter.upper()` converts any lowercase letters entered by the user into uppercase letters, storing the result back into the same variable. Including this statement allows the program to work with lowercase letters without including them in the conditions for the `if` and `elif` parts.

```
# Convert from a letter grade to a number of grade points using -1 grade points as a sentinel
# value indicating invalid input
if letter == "A+" or letter == "A":
  gp = A
elif letter == "A-":
  gp = A_MINUS
elif letter == "B+":
  gp = B_PLUS
elif letter == "B":
  gp = B
elif letter == "B-":
  gp = B_MINUS
elif letter == "C+":
  gp = C_PLUS
elif letter == "C":
  gp = C
elif letter == "C-":
  gp = C_MINUS
elif letter == "D+":
  gp = D_PLUS
elif letter == "D":
  gp = D
elif letter == "F":
  gp = F
else:
  gp = INVALID
```

```
# Report the result
if gp == INVALID:
  print("That wasn't a valid letter grade.")
else:
  print("A(n)", letter, "is equal to", gp, "grade points.")
```

Solution to Exercise 54: Assessing Employees

```
##
#  Report whether an employee's performance is unacceptable, acceptable or meritorious
#  based on the rating entered by the user.
#
RAISE_FACTOR = 2400.00
UNACCEPTABLE = 0
ACCEPTABLE = 0.4
MERITORIOUS = 0.6

# Read the rating from the user
rating = float(input("Enter the rating: "))

# Classify the performance
if rating == UNACCEPTABLE:
  performance = "Unacceptable"
elif rating == ACCEPTABLE:
  performance = "Acceptable"
elif rating >= MERITORIOUS:
  performance = "Meritorious"
else:
  performance = ""

# Report the result
if performance == "":
  print("That wasn't a valid rating.")
else:
  print("Based on that rating, your performance is %s." % \
        performance)
  print("You will receive a raise of $%.2f." % \
        (rating * RAISE_FACTOR))
```

The parentheses around `rating * RAISE_FACTOR` on the final line are necessary because the `%` and `*` operators have the same precedence. Including the parentheses forces Python to perform the mathematical calculation before formatting its result.

Solution to Exercise 58: Is It a Leap Year?

```
##
#  Determine whether or not a year is a leap year.
#

# Read the year from the user
year = int(input("Enter a year: "))
```

```
# Determine if it is a leap year
if year % 400 == 0:
  isLeapYear = True
elif year % 100 == 0:
  isLeapYear = False
elif year % 4 == 0:
  isLeapYear = True
else:
  isLeapYear = False

# Display the result
if isLeapYear:
  print(year, "is a leap year.")
else:
  print(year, "is not a leap year.")
```

Solution to Exercise 61: Is a License Plate Valid?

```
## Determine whether or not a license plate is valid.  A valid license plate either consists of:
#   1)  3 letters followed by 3 numbers, or
#   2)  4 numbers followed by 3 numbers

# Read the plate from the user
plate = input("Enter the license plate: ")

# Check the status of the plate and display it.  It is necessary to check each of the 6 characters
# for an older style plate, or each of the 7 characters for a newer style plate.
if len(plate) == 6 and \
   plate[0] >= "A" and plate[0] <= "Z" and \
   plate[1] >= "A" and plate[1] <= "Z" and \
   plate[2] >= "A" and plate[2] <= "Z" and \
   plate[3] >= "0" and plate[3] <= "9" and \
   plate[4] >= "0" and plate[4] <= "9" and \
   plate[5] >= "0" and plate[5] <= "9":
  print("The plate is a valid older style plate.")
elif len(plate) == 7 and \
   plate[0] >= "0" and plate[0] <= "9" and \
   plate[1] >= "0" and plate[1] <= "9" and \
   plate[2] >= "0" and plate[2] <= "9" and \
   plate[3] >= "0" and plate[3] <= "9" and \
   plate[4] >= "A" and plate[4] <= "Z" and \
   plate[5] >= "A" and plate[5] <= "Z" and \
   plate[6] >= "A" and plate[6] <= "Z":
  print("The plate is a valid newer style plate.")
else:
  print("The plate is not valid.")
```

Solution to Exercise 62: Roulette Payouts

```
##
#  Display the bets that pay out in a roulette simulation.
#
from random import randrange
```

```
# Simulate spinning the wheel, using 37 to represent 00
value = randrange(0, 38)
if value == 37:
  print("The spin resulted in 00...")
else:
  print("The spin resulted in %d..." % value)

# Display the payout for a single number
if value == 37:
  print("Pay 00")
else:
  print("Pay", value)

# Display the color payout
# The first line in the condition checks for 1, 3, 5, 7 and 9
# The second line in the condition checks for 12, 14, 16 and 18
# The third line in the condition checks for 19, 21, 23, 25 and 27
# The fourth line in the condition checks for 30, 32, 34 and 36
if value % 2 == 1 and value >= 1 and value <= 9 or \
   value % 2 == 0 and value >= 12 and value <= 18 or \
   value % 2 == 1 and value >= 19 and value <= 27 or \
   value % 2 == 0 and value >= 30 and value <= 36:
  print("Pay Red")
elif value == 0 or value == 37:
  pass
else:
  print("Pay Black")

# Display the odd vs. even payout
if value >= 1 and value <= 36:
  if value % 2 == 1:
    print("Pay Odd")
  else:
    print("Pay Even")

# Display the lower numbers vs. upper numbers payout
if value >= 1 and value <= 18:
  print("Pay 1 to 18")
elif value >= 19 and value <= 36:
  print("Pay 19 to 36")
```

The body of an `if`, `elif` or `else` must contain at least one statement. Python includes the `pass` keyword which can be used when a statement is required but there is no work to be performed.

Solutions to the Repetition Exercises

11

Solution to Exercise 66: No More Pennies

```
##
#  Compute the total due when several items are purchased.  The amount payable for cash
#  transactions is rounded to the closest nickel because pennies have been phased out in Canada.
#
PENNIES_PER_NICKEL = 5
NICKEL = 0.05
```

While it is highly unlikely that the number of pennies in a nickel will ever change, it is possible (even likely) that we will need to update our program at some point in the future so that it rounds to the closest dime. Using constants will make it easier to perform that update when it is needed.

```
# Track the total cost for all of the items
total = 0.00

# Read the price of the first item as a string
line = input("Enter the price of the item (blank to quit): ")

# Continue reading items until a blank line is entered
while line != "":
  # Add the cost of the item to the total (after converting it to a floating-point number)
  total = total + float(line)

  # Read the cost of the next item
  line = input("Enter the price of the item (blank to quit): ")

# Display the exact total payable
print("The exact amount payable is %.02f" % total)

# Compute the number of pennies that would be left if the total was paid using nickels
rounding_indicator = total * 100 % PENNIES_PER_NICKEL
```


B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_11

```
if rounding_indicator < PENNIES_PER_NICKEL / 2:
    # If the number of pennies left is less than 2.5 then we round down by subtracting that
    # number of pennies from the total
    cash_total = total - rounding_indicator / 100
else:
    # Otherwise we add a nickel and then subtract that number of pennies
    cash_total = total + NICKEL - rounding_indicator / 100

# Display amount due when paying with cash
print("The cash amount payable is %.02f" % cash_total)
```

Solution to Exercise 67: Compute the Perimeter of a Polygon

```
##
#  Compute the perimeter of a polygon constructed from points entered by the user.  A blank line
#  will be entered for the x-coordinate to indicate that all of the points have been entered.
#
from math import sqrt

# Store the perimeter of the polygon
perimeter = 0

# Read the coordinate of the first point
first_x = float(input("Enter the first x-coordinate: "))
first_y = float(input("Enter the first y-coordinate: "))

# Provide initial values for prev_x and prev_y
prev_x = first_x
prev_y = first_y

# Read the remaining coordinates
line = input("Enter the next x-coordinate (blank to quit): ")
while line != "":
    # Convert the x-coordinate to a number and read the y coordinate
    x = float(line)
    y = float(input("Enter the next y-coordinate: "))

    # Compute the distance to the previous point and add it to the perimeter
    dist = sqrt((prev_x - x) ** 2 + (prev_y - y) ** 2)
    perimeter = perimeter + dist

    # Set up prev_x and prev_y for the next loop iteration
    prev_x = x
    prev_y = y

    # Read the next x-coordinate
    line = input("Enter the next x-coordinate (blank to quit): ")

# Compute the distance from the last point to the first point and add it to the perimeter
dist = sqrt((first_x - x) ** 2 + (first_y - y) ** 2)
perimeter = perimeter + dist

# Display the result
print("The perimeter of that polygon is", perimeter)
```

The distance between the points is computed using Pythagorean theorem.

Solution to Exercise 69: Admission Price

```
##
#  Compute the admission price for a group visiting the zoo.
#

# Store the admission prices as constants
BABY_PRICE = 0.00
CHILD_PRICE = 14.00
ADULT_PRICE = 23.00
SENIOR_PRICE = 18.00

# Store the age limits as constants
BABY_LIMIT = 2
CHILD_LIMIT = 12
ADULT_LIMIT = 64

# Create a variable to hold the total admission cost for all guests
total = 0

# Keep on reading ages until the user enters a blank line
line = input("Enter the age of the guest (blank to finish): ")
while line != "":
  age = int(line)

  # Add the correct amount to the total
  if age <= BABY_LIMIT:
    total = total + BABY_PRICE
  elif age <= CHILD_LIMIT:
    total = total + CHILD_PRICE
  elif age <= ADULT_LIMIT:
    total = total + ADULT_PRICE
  else:
    total = total + SENIOR_PRICE

  # Read the next age from the user
  line = input("Enter the age of the guest (blank to finish): ")

# Display the total due for the group, formatted using two decimal places
print("The total for that group is $%.2f" % total)
```

With the current admission prices the first part of the `if-elif-else` statement could be eliminated. However, including it makes the program easier to update so that it charges for babies in the future.

Solution to Exercise 70: Parity Bits

```
##
#  Compute the parity bit using even parity for sets of 8 bits entered by the user.
#

# Read the first line of input
line = input("Enter 8 bits: ")
```

```
# Continue looping until a blank line is entered
while line != "":
  # Ensure that the line has a total of 8 zeros and ones and exactly 8 characters
  if line.count("0") + line.count("1") != 8 or len(line) != 8:
    # Display an appropriate error message
    print("That wasn't 8 bits... Try again.")
  else:
    # Count the number of ones
    ones = line.count("1")
```

The `count` method returns the number of times that its argument occurs in the string on which it was invoked.

```
    # Display the parity bit
    if ones % 2 == 0:
      print("The parity bit should be 0.")
    else:
      print("The parity bit should be 1.")

  # Read the next line of input
  line = input("Enter 8 bits: ")
```

Solution to Exercise 73: Caesar Cipher

```
##
#  Implement a Caesar cipher that shifts all of the letters in a message by an amount provided
#  by the user.  Use a negative shift value to decode a message.
#

# Read the message and shift amount from the user
message = input("Enter the message: ")
shift = int(input("Enter the shift value: "))

# Process each character to construct the encrypted (or decrypted) message
new_message = ""
for ch in message:
  if ch >= "a" and ch <= "z":
    # Process a lowercase letter by determining its
    # position in the alphabet (0 - 25), computing its
    # new position, and adding it to the new message
    pos = ord(ch) - ord("a")
    pos = (pos + shift) % 26
    new_char = chr(pos + ord("a"))
    new_message = new_message + new_char
  elif ch >= "A" and ch <= "Z":
    # Process an uppercase letter by determining its position in the alphabet (0 - 25),
    # computing its new position, and adding it to the new message
    pos = ord(ch) - ord("A")
    pos = (pos + shift) % 26
    new_char = chr(pos + ord("A"))
    new_message = new_message + new_char
  else:
    # If the character is not a letter then copy it into the new message
    new_message = new_message + ch
```

The `ord` function converts a character to its integer position within the ASCII table. The `chr` function returns the character in the ASCII table at the position provided as an argument.

```
# Display the shifted message
print("The shifted message is", new_message)
```

Solution to Exercise 75: Is a String a Palindrome?

```
##
#  Determine whether or not a string entered by the user is a palindrome.
#

# Read the string from the user
line = input("Enter a string: ")

# Assume that it is a palindrome until we can prove otherwise
is_palindrome = True

# Check the characters, starting from the ends.  Continue until the middle is reached or we have
# determined that the string is not a palindrome.
i = 0
while i < len(line) / 2 and is_palindrome:
  # If the characters do not match then mark that the string is not a palindrome
  if line[i] != line[len(line) - i - 1]:
    is_palindrome = False

  # Move to the next character
  i = i + 1

# Display a meaningful output message
if is_palindrome:
  print(line, "is a palindrome")
else:
  print(line, "is not a palindrome")
```

Solution to Exercise 77: Multiplication Table

```
##
# Display a multiplication table for 1 times 1 through 10 times 10.
#
MIN = 1
MAX = 10

# Display the top row of labels
print("    ", end="")
for i in range(MIN, MAX + 1):
  print("%4d" % i, end="")
print()
```

Including `end=""` as the final argument to `print` prevents it from moving down to the next line after displaying the value.

```
# Display the table
for i in range(MIN, MAX + 1):
  print("%4d" % i, end="")
  for j in range(MIN, MAX + 1):
    print("%4d" % (i * j), end="")
  print()
```

Solution to Exercise 79: Greatest Common Divisor

```
##
# Compute the greatest common divisor of two positive integers using a while loop.
#

# Read two positive integers from the user
n = int(input("Enter a positive integer: "))
m = int(input("Enter a positive integer: "))

# Initialize d to the smaller of n and m
d = min(n, m)

# Use a while loop to find the greatest common divisor of n and m
while n % d != 0 or m % d != 0:
  d = d - 1

# Report the result
print("The greatest common divisor of", n, "and", m, "is", d)
```

Solution to Exercise 82: Decimal to Binary

```
##
# Convert a number from decimal (base 10) to binary (base 2).
#
NEW_BASE = 2

# Read the number to convert from the user
num = int(input("Enter a non-negative integer: "))

# Generate the binary representation of num, storing it in result
result = ""
q = num
```

The algorithm provided for this question is expressed using a repeat-until loop. However, this type of loop isn't available in Python. As a result, the algorithm has to be adjusted so that it generates the same result using a `while` loop. This is achieved by duplicating the loop's body and placing it ahead of the `while` loop.

```
# Perform the body of the loop once
r = q % NEW_BASE
result = str(r) + result
q = q // NEW_BASE

# Keep on looping until q is 0
while q > 0:
  r = q % NEW_BASE
  result = str(r) + result
  q = q // NEW_BASE

# Display the result
print(num, "in decimal is", result, "in binary.")
```

Solution to Exercise 83: Maximum Integer

```
##
# Find the maximum of 100 random integers and count the number of times the maximum value
# is updated during the process.
#
from random import randrange

NUM_ITEMS = 100

# Generate the first number and display it
mx_value = randrange(1, NUM_ITEMS + 1)
print(mx_value)

# Count the number of times the maximum value is updated
num_updates = 0

# For each of the remaining numbers
for i in range(1, NUM_ITEMS):
  # Generate a new random number
  current = randrange(1, NUM_ITEMS + 1)

  # If the generated number is the largest one we have seen so far
  if current > mx_value:
    # Update the maximum and count the update
    mx_value = current
    num_updates = num_updates + 1
    # Display the number, noting that an update occurred
    print(current, "<== Update")
  else:
    # Display the number
    print(current)

# Display the other results
print("The maximum value found was", mx_value)
print("The maximum value was updated", num_updates, "times")
```

Solutions to the Function Exercises

12

Solution to Exercise 88: Median of Three Values

```
##
#  Compute and display the median of three values entered by the user.  This program includes
#  two implementations of the median function that demonstrate different techniques for
#  computing the median of three values.
#

## Compute the median of three values using if statements
#  @param a the first value
#  @param b the second value
#  @param c the third value
#  @return the median of values a, b and c
#
def median(a, b, c):
  if a < b and b < c or a > b and b > c:
    return b
  if b < a and a < c or b > a and a > c:
    return a
  if c < a and b < c or c > a and b > c:
    return c
```

Each function that you write should begin with a comment. Lines beginning with `@param` are used to describe the function's parameters. The value returned by the function is describe by a line that begins with `@return`.

```
## Compute the median of three values using the min and max functions and a little bit of
#  arithmetic
#  @param a the first value
#  @param b the second value
#  @param c the third value
#  @return the median of values a, b and c
#
def alternateMedian(a, b, c):
  return a + b + c - min(a, b, c) - max(a, b, c)
```

The median of three values is the sum of the values, less the smallest, less the largest.

```
# Display the median of 3 values entered by the user
def main():
  x = float(input("Enter the first value: "))
  y = float(input("Enter the second value: "))
  z = float(input("Enter the third value: "))
```


B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_12

```
  print("The median value is:", median(x, y, z))
  print("Using the alternative method, it is:", \
        alternateMedian(x, y, z))

# Call the main function
main()
```

Solution to Exercise 90: The Twelve Days of Christmas

```
##
# Display the complete lyrics for the song The Twelve Days of Christmas.
#
from int_ordinal import intToOrdinal
```

> The function that was written for the previous exercise is imported into this program so that the code for converting from an integer to its ordinal number does not have to be duplicated here.

```
## Display one verse of The Twelve Days of Christmas
# @param n the verse to display
# @return (None)
def displayVerse(n):
  print("On the", intToOrdinal(n), "day of Christmas")
  print("my true love sent to me:")

  if n >= 12:
    print("Twelve drummers drumming,")
  if n >= 11:
    print("Eleven pipers piping,")
  if n >= 10:
    print("Ten lords a-leaping,")
  if n >= 9:
    print("Nine ladies dancing,")
  if n >= 8:
    print("Eight maids a-milking,")
  if n >= 7:
    print("Seven swans a-swimming,")
  if n >= 6:
    print("Six geese a-laying,")
  if n >= 5:
    print("Five golden rings,")
  if n >= 4:
    print("Four calling birds,")
  if n >= 3:
    print("Three French hens,")
```

```
  if n >= 2:
      print("Two turtle doves,")
  if n == 1:
    print("A", end=" ")
  else:
    print("And a", end=" ")
  print("partridge in a pear tree.")
  print()

# Display all 12 verses of the song
def main():
  for verse in range(1, 13):
    displayVerse(verse)

# Call the main function
main()
```

Solution to Exercise 93: Center a String in the Terminal Window

```
##
#  Center a string of characters within a certain width.
#
WIDTH = 80

## Create a new string that will be centered within a given width when it is printed.
#  @param s the string that will be centered
#  @param width the width in which the string will be centered
#  @return a new copy of s that contains the leading spaces needed to center s
def center(s, width):
  # If the string is too long to center, then the original string is returned
  if width < len(s):
    return s

  # Compute the number of spaces needed and generate the result
  spaces = (width - len(s)) // 2
  result = " " * spaces + s
```

The // operator is used so that the result of the division is truncated to an integer. The / operator cannot be used because it returns a floating-point result but a string can only be replicated an integer number of times.

```
  return result

# Demonstrate the center function
def main():
  print(center("A Famous Story", WIDTH))
  print(center("by:", WIDTH))
  print(center("Someone Famous", WIDTH))
  print()
  print("Once upon a time...")

# Call the main function
main()
```

Solution to Exercise 95: Capitalize It

```
##
#  Improve the capitalization of a string.
#

## Capitalize the appropriate characters in a string
#  @param s the string that needs capitalization
#  @return a new string with the capitalization improved
def capitalize(s):
  # Create a new copy of the string to return as the function's result
  result = s

  # Capitalize the first non-space character in the string
  pos = 0
  while pos < len(s) and result[pos] == ' ':
    pos = pos + 1

  if pos < len(s):
    # Replace the character with its uppercase version without changing any other characters
    result = result[0 : pos] + result[pos].upper() + \
             result[pos + 1 : len(result)]
```

Using a colon inside of square brackets retrieves a portion of a string. The characters that are retrieved start at the position that appears to the left of the colon, going up to (but not including) the position that appears to the right of the colon.

```
  # Capitalize the first letter that follows a ".", "!" or "?"
  pos = 0
  while pos < len(s):
    if result[pos] == "." or result[pos] == "!" or \
       result[pos] == "?":
      # Move past the ".", "!" or "?"
      pos = pos + 1

      # Move past any spaces
      while pos < len(s) and result[pos] == " ":
        pos = pos + 1

      # If we haven't reached the end of the string then replace the current character
      # with its uppercase equivalent
      if pos < len(s):
        result = result[0 : pos] + \
                 result[pos].upper() + \
                 result[pos + 1 : len(result)]

    # Move to the next character
    pos = pos + 1
```

```
    # Capitalize i when it is preceded by a space and followed by a space, period, exclamation
    # mark, question mark or apostrophe
    pos = 1
    while pos < len(s) - 1:
      if result[pos - 1] == " " and result[pos] == "i" and \
          (result[pos + 1] == " " or result[pos + 1] == "." or \
           result[pos + 1] == "!" or result[pos + 1] == "?" or \
           result[pos + 1] == "'"):
        # Replace the i with an I without changing any other characters
        result = result[0 : pos] + "I" + \
                 result[pos + 1 : len(result)]
      pos = pos + 1

    return result

# Demonstrate the capitalize function
def main():
  s = input("Enter some text: ")
  capitalized = capitalize(s)
  print("It is capitalized as:", capitalized)

# Call the main function
main()
```

Solution to Exercise 96: Does a String Represent an Integer?

```
##
#  Determine whether or not a string entered by the user is an integer.
#

## Determine if a string contains a valid representation of an integer
#  @param s the string to check
#  @return True if s represents an integer.  False otherwise.
#
def isInteger(s):
  # Remove whitespace from the beginning and end of the string
  s = s.strip()

  # Determine if the remaining characters form a valid integer
  if (s[0] == "+" or s[0] == "-") and s[1:].isdigit():
    return True
  if s.isdigit():
    return True
  return False
```

The `isdigit` method returns `True` if and only if the string is at least one character in length and all of the characters in the string are digits.

```
# Demonstrate the isInteger function
def main():
  s = input("Enter a string: ")
  if isInteger(s):
    print("That string represents an integer.")
  else:
    print("That string does not represent an integer.")
```

```
# Only call the main function when this file has not been imported
if __name__ == "__main__":
  main()
```

The `__name__` variable is automatically assigned a value by Python when the program starts running. It contains `"__main__"` when the file is executed directly by Python. It contains the name of the module when the file is imported into another program.

Solution to Exercise 98: Is a Number Prime?

```
##
#  Determine if a number entered by the user is prime.
#

## Determine whether or not a number is prime
#  @param n the integer to test
#  @return True if the number is prime, False otherwise
def isPrime(n):
  if n <= 1:
    return False

  # Check each number from 2 up to but not including n to see if it divides evenly into n
  for i in range(2, n):
    if n % i == 0:
      return False
  return True
```

If `n % i == 0` then `n` is evenly divisible by `i`, indicating that `n` is not prime.

```
# Determine if a number entered by the user is prime
def main():
  value = int(input("Enter an integer: "))
  if isPrime(value):
    print(value, "is prime.")
  else:
    print(value, "is not prime.")

# Call the main function if the file has not been imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 100: Random Password

```
##
#  Generate and display a random password containing between 7 and 10 characters.
#
from random import randint

SHORTEST = 7
LONGEST = 10
MIN_ASCII = 33
MAX_ASCII = 126
```

```
## Generate a random password
#  @return a string containing a random password
def randomPassword():
  # Select a random length for the password
  randomLength = randint(SHORTEST, LONGEST)

  # Generate an appropriate number of random characters, adding each one to the end of result
  result = ""
  for i in range(randomLength):
    randomChar = chr(randint(MIN_ASCII, MAX_ASCII))
    result = result + randomChar
```

The `chr` function takes an ASCII code as its only parameter. It returns a string containing the character with that ASCII code as its result.

```
  # Return the random password
  return result

# Generate and display a random password
def main():
  print("Your random password is:", randomPassword())

# Call the main function only if the module is not imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 102: Check a Password

```
##
#  Check whether or not a password is good.
#

## Check whether or not a password is good.  A good password is at least 8 characters and
#  contains an uppercase letter, a lowercase letter and a number.
#  @param password the password to check
#  @return True if the password is good, False otherwise
def checkPassword(password):
  has_upper = False
  has_lower = False
  has_num = False

  # Check each character in the password and see which requirement it meets
  for ch in password:
    if ch >= "A" and ch <= "Z":
      has_upper = True
    elif ch >= "a" and ch <= "z":
      has_lower = True
    elif ch >= "0" and ch <= "9":
      has_num = True

  # If the password has all 4 properties
  if len(password) >= 8 and has_upper and has_lower and has_num:
    return True
```

```
    # The password is missing at least one property
    return False

# Demonstrate the password checking function
def main():
    p = input("Enter a password: ")
    if checkPassword(p):
        print("That's a good password.")
    else:
        print("That isn't a good password.")

# Call the main function only if the file has not been imported into another program
if __name__ == "__main__":
    main()
```

Solution to Exercise 105: Arbitrary Base Conversions

```
##
#  Convert a number from one base to another.  Both the source base and the destination base
#  must be between 2 and 16.
#
from hex_digit import *
```

The hex_digit module contains the functions hex2int and int2hex which were developed while solving Exercise 104. Using import * imports all of the functions from that module.

```
## Convert a number from base 10 to base new_base
#  @param num the base 10 number to convert
#  @param new_base the base to convert to
#  @return the string of digits in new_base
def dec2n(num, new_base):
    # Generate the representation of num in base new_base, storing it in result
    result = ""
    q = num

    # Perform the body of the loop once
    r = q % new_base
    result = int2hex(r) + result
    q = q // new_base

    # Continue looping until q is 0
    while q > 0:
        r = q % new_base
        result = int2hex(r) + result
        q = q // new_base

    # Return the result
    return result
```

```
## Convert a number from base b to base 10
#  @param num the base b number, stored in a string
#  @param b the base of the number to convert
#  @return the base 10 number
def n2dec(num, b):
  decimal = 0

  # Process each digit in the base b number
  for i in range(len(num)):
    decimal = decimal * b
    decimal = decimal + hex2int(num[i])

  # Return the result
  return decimal

# Convert a number between two arbitrary bases
def main():
  # Read the base and number from the user
  from_base = int(input("Base to convert from (2-16): "))
  if from_base < 2 or from_base > 16:
    print("Only bases between 2 and 16 are supported.")
    print("Quitting...")
    quit()

  from_num = input("Sequence of digits in that base: ")

  # Convert to base 10 and display the result
  dec = n2dec(from_num, from_base)
  print("That's %d in base 10." % dec)

  # Convert to the new base and display the result
  to_base = int(input("Enter the base to convert to (2-16): "))
  if to_base < 2 or to_base > 16:
    print("Only bases between 2 and 16 are supported.")
    print("Quitting...")
    quit()

  to_num = dec2n(dec, to_base)
  print("That's %s in base %d." % (to_num, to_base))

# Call the main function
main()
```

The base b number must be stored in a string because it may contain letters that represent digits in bases larger than 10.

Solution to Exercise 107: Reduce a Fraction to Lowest Terms

```
##
#  Reduce a fraction to lowest terms.
#

## Compute the greatest common divisor of two integers
#  @param n the first integer under consideration (must be non-zero)
#  @param m the second integer under consideration (must be non-zero)
#  @return the greatest common divisor of the integers
def gcd(n, m):
  # Initialize d to the smaller of n and m
  d = min(n, m)
```

```
    # Use a while loop to find the greatest common divisor of n and m
    while n % d != 0 or m % d != 0:
      d = d - 1

    return d
```

This function uses a loop to achieve its goal. There is also an elegant solution for finding the greatest common divisor of two integers that uses recursion. The recursive solution is explored in Exercise 174.

```
## Reduce a fraction to lowest terms
#  @param num the integer numerator of the fraction
#  @param den the integer denominator of the fraction (must be non-zero)
#  @return the numerator and denominator of the reduced fraction
def reduce(num, den):
    # If the numerator is 0 then the reduced fraction is 0 / 1
    if num == 0:
      return (0, 1)

    # Compute the greatest common divisor of the numerator and denominator
    g = gcd(num, den)

    # Divide both the numerator and denominator by the GCD and return the result
    return (num // g, den // g)
```

We have used the floor division operator, `//`, so that numerator and the denominator are both integers in the result that is returned by the function.

```
# Read a fraction from the user and display the equivalent lowest terms fraction
def main():
    # Read the numerator and denominator from the user
    num = int(input("Enter the numerator: "))
    den = int(input("Enter the denominator: "))

    # Compute the reduced fraction
    (n, d) = reduce(num, den)

    # Display the result
    print("%d/%d can be reduced to %d/%d." % (num, den, n, d))

# Call the main function
main()
```

Solution to Exercise 108: Reduce Measures

```
##
#  Reduce an imperial measurement so that it is expressed using the largest possible units of
#  measure.  For example, 59 teaspoons is reduced to 1 cup, 3 tablespoons, 2 teaspoons.
#
TSP_PER_TBSP = 3
TSP_PER_CUP = 48
```

```
## Reduce an imperial measurement so that it is expressed using the largest possible
#  units of measure
#  @param num the number of units that need to be reduced
#  @param unit the unit of measure (‘‘cup’’, ‘‘tablespoon’’ or ‘‘teaspoon’’)
#  @return a string representing the measurement in reduced form
def reduceMeasure(num, unit):
  # Convert the unit to lowercase
  unit = unit.lower()
```

The unit is converted to lowercase by invoking the `lower` method on `unit` and storing the result into the same variable. This allows the user to use any mixture of uppercase and lowercase letters when specifying the unit.

```
  # Compute the number of teaspoons that the parameters represent
  if unit == "teaspoon" or unit == "teaspoons":
    teaspoons = num
  elif unit == "tablespoon" or unit == "tablespoons":
    teaspoons = num * TSP_PER_TBSP
  elif unit == "cup" or unit == "cups":
    teaspoons = num * TSP_PER_CUP

  # Convert the number of teaspoons to the largest possible units of measure
  cups = teaspoons // TSP_PER_CUP
  teaspoons = teaspoons - cups * TSP_PER_CUP
  tablespoons = teaspoons // TSP_PER_TBSP
  teaspoons = teaspoons - tablespoons * TSP_PER_TBSP

  # Create a string to hold the result
  result = ""

  # Add the number of cups to the result string (if any)
  if cups > 0:
    result = result + str(cups) + " cup"
    # Make cup plural if there is more than one
    if cups > 1:
      result = result + "s"

  # Add the number of tablespoons to the result string (if any)
  if tablespoons > 0:
    # Include a comma if there were some cups
    if result != "":
      result = result + ", "

    result = result + str(tablespoons) + " tablespoon"
    # Make tablespoon plural if there is more than one
    if tablespoons > 1:
      result = result + "s"

  # Add the number of teaspoons to the result string (if any)
  if teaspoons > 0:
    # Include a comma if there were some cups and/or tablespoons
    if result != "":
      result = result + ", "
```

```
    result = result + str(teaspoons) + " teaspoon"
    # Make teaspoons plural if there is more than one
    if teaspoons > 1:
      result = result + "s"

  # Handle the case where the number of units was 0
  if result == "":
    result = "0 teaspoons"

  return result
```

Several test cases are included in this program. They exercise a variety of different combinations of zero, one and multiple occurrences of the different units of measure. While these test cases are reasonably thorough, they do not guarantee that the program is bug free.

```
# Demonstrate the reduceMeasure function by performing several reductions
def main():
  print("59 teaspoons is %s." % reduceMeasure(59, "teaspoons"))
  print("59 tablespoons is %s." % \
        reduceMeasure(59, "tablespoons"))
  print("1 teaspoon is %s." % reduceMeasure(1, "teaspoon"))
  print("1 tablespoon is %s." % reduceMeasure(1, "tablespoon"))
  print("1 cup is %s." % reduceMeasure(1, "cup"))
  print("4 cups is %s." % reduceMeasure(4, "cups"))
  print("3 teaspoons is %s." % reduceMeasure(3, "teaspoons"))
  print("6 teaspoons is %s." % reduceMeasure(6, "teaspoons"))
  print("95 teaspoons is %s." % reduceMeasure(95, "teaspoons"))
  print("96 teaspoons is %s." % reduceMeasure(96, "teaspoons"))
  print("97 teaspoons is %s." % reduceMeasure(97, "teaspoons"))
  print("98 teaspoons is %s." % reduceMeasure(98, "teaspoons"))
  print("99 teaspoons is %s." % reduceMeasure(99, "teaspoons"))

# Call the main function
main()
```

Solution to Exercise 109: Magic Dates

```
##
#  Determine all of the magic dates in the 1900s.
#
from days_in_month import daysInMonth

## Determine whether or not a date is ‘‘magic’’
#  @param day the day portion of the date
#  @param month the month portion of the date
#  @param year the year portion of the date
#  @return True if the date is magic, False otherwise
def isMagicDate(day, month, year):
  if day * month == year % 100:
    return True

  return False
```

The expression `year % 100` evaluates to the two digit year.

```
# Find and display all of the magic dates in the 1900s
def main():
  for year in range(1900, 2000):
    for month in range(1, 13):
      for day in range(1, daysInMonth(month, year) + 1):
        if isMagicDate(day, month, year):
          print("%02d/%02d/%04d is a magic date." % (day, month, year))

# Call the main function
main()
```

Solutions to the List Exercises 13

Solution to Exercise 110: Sorted Order

```
##
#  Display a list of integers entered by the user in ascending order.
#

# Start with an empty list
data = []

# Read values and add them to the list until the user enters 0
num = int(input("Enter an integer (0 to quit): "))
while num != 0:
  data.append(num)
  num = int(input("Enter an integer (0 to quit): "))

# Sort the values
data.sort()
```

Invoking the `sort` method on a list rearranges the elements in the list into sorted order. Using the `sort` method is appropriate for this problem because there is no need to retain a copy of the original list. The `sorted` function can be used to create a new copy of the list where the elements are in sorted order. Calling the `sorted` function does not modify the original list. As a result, it can be used in situations where the original list and the sorted list are needed simultaneously.

```
# Display the values in ascending order
print("The values, sorted into ascending order, are:")
for num in data:
  print(num)
```

Solution to Exercise 112: Remove Outliers

```
##
#  Remove the outliers from a data set.
#
```


B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_13

```
## Remove the outliers from a list of values
#  @param data the list of values to process
#  @param num_outliers the number of smallest and largest values to remove
#  @return a new copy of data where the values are sorted into ascending order and the
#          smallest and largest values have been removed
def removeOutliers(data, num_outliers):
  # Create a new copy of the list that is in sorted order
  retval = sorted(data)

  # Remove num_outliers largest values
  for i in range(num_outliers):
    retval.pop()

  # Remove num_outliers smallest values
  for i in range(num_outliers):
    retval.pop(0)

  # Return the result
  return retval
```

The smallest and largest outliers could be removed using the same loop. Two loops are used in this solution to make the steps more clear.

```
# Read data from the user, and remove the two largest and two smallest values
def main():
  # Read values from the user until a blank line is entered
  values = []
  s = input("Enter a value (blank line to quit): ")
  while s != "":
    num = float(s)
    values.append(num)
    s = input("Enter a value (blank line to quit): ")

  # Display the result or an appropriate error message
  if len(values) < 4:
    print("You didn't enter enough values.")
  else:
    print("With the outliers removed: ", \
          removeOutliers(values, 2))
    print("The original data: ", values)

# Call the main function
main()
```

Solution to Exercise 113: Avoiding Duplicates

```
##
#  Read a collection of words entered by the user.  Display each word entered by the user only
#  once, in the same order that the words were entered.
#

# Read words from the user and store them in a list
words = []
word = input("Enter a word (blank line to quit): ")
while word != "":
  # Only add the word to the list if
  # it is not already present in it
  if word not in words:
    words.append(word)

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

The expressions `word not in words` and `not (word in words)` are equivalent.

```
# Display the unique words
for word in words:
  print(word)
```

Solution to Exercise 114: Negatives, Zeros and Positives

```
##
#  Read a collection of integers from the user.  Display all of the negative numbers, followed
#  by all of the zeros, followed by all of the positive numbers.
#

# Create three lists to store the negative, zero and positive values
negatives = []
zeros = []
positives = []
```

> This solution uses a list to keep track of the zeros that are entered. However, because all of the zeros are the same, it is not actually necessary to save them. Instead, one could use an integer variable to count the number of zeros and then display that many zeros later in the program.

```
# Read all of the integers from the user, storing each integer in the correct list
line = input("Enter an integer (blank to quit): ")
while line != "":
  num = int(line)

  if num < 0:
    negatives.append(num)
  elif num > 0:
    positives.append(num)
  else:
    zeros.append(num)

  # Read the next line of input from the user
  line = input("Enter an integer (blank to quit): ")

# Display all of the negative values, then all of the zeros, then all of the positive values
print("The numbers were: ")

for n in negatives:
  print(n)

for n in zeros:
  print(n)

for n in positives:
  print(n)
```

Solution to Exercise 116: Perfect Numbers

```
##
#  A number, n, is a perfect number if the sum of the proper divisors of n is equal to n.  This
#  program displays all of the perfect numbers between 1 and LIMIT.
#
from proper_divisors import properDivisors

LIMIT = 10000
```

```
## Determine whether or not a number is perfect.  A number is perfect if the sum of its proper
#  divisors is equal to the number itself.
#  @param n the number to check for perfection
#  @return True if the number is perfect, False otherwise
def isPerfect(n):
  # Get a list of the proper divisors of n
  divisors = properDivisors(n)

  # Compute the total of all of the divisors
  total = 0
  for d in divisors:
    total = total + d
```

The total could also be computed using Python's built-in `sum` function. This would reduce the calculation of the total to a single line.

```
  # Determine whether or not the number is perfect and return the appropriate result
  if total == n:
    return True
  return False

# Display all of the perfect numbers between 1 and LIMIT
def main():
  print("The perfect numbers between 1 and", LIMIT, "are:")
  for i in range(1, LIMIT + 1):
    if isPerfect(i):
      print(" ", i)

# Call the main function
main()
```

Solution to Exercise 120: Formatting a List

```
##
#  Display a list of items so that they are separated by commas and the word ‘‘and’’ appears
#  between the final two items.
#

## Format a list of items so that they are separated by commas and ‘‘and’’
# @param items the list of items to format
# @return a string containing the items with the desired formatting
def formatList(items):
  # Handle lists of 0 and 1 items as special cases
  if len(items) == 0:
    return "<empty>"
  if len(items) == 1:
    return str(items[0])

  # Loop over all of the items in the list except the last two
  result = ""
  for i in range(0, len(items) - 2):
    result = result + str(items[i]) + ", "
```

Each item is explicitly converted to a string by calling the `str` function before it is concatenated to the result. This allows the `formatList` function to format lists that contain numbers in addition to strings.

```
  # Add the second last and last items to the result, separated by ‘‘and’’
  result = result + str(items[len(items) - 2]) + " and "
  result = result + str(items[len(items) - 1])
```

```
    # Return the result
    return result

# Read several items entered by the user and display them with nice formatting
def main():
    # Read items from the user until a blank line is entered
    items = []
    line = input("Enter an item (blank to quit): ")
    while line != "":
        items.append(line)
        line = input("Enter an item (blank to quit): ")

    # Format and display the items
    print("The items are %s." % formatList(items))

# Call the main function
main()
```

Solution to Exercise 121: Random Lottery Numbers

```
##
#  Compute random but distinct numbers for a lottery ticket.
#
from random import randrange

MIN_NUM = 1
MAX_NUM = 49
NUM_NUMS = 6
```

Using constants makes it easy to reconfigure our program for other lotteries.

```
# Use a list to store the numbers on the ticket
ticket_nums = []

# Generate NUM_NUMS random but distinct numbers
for i in range(NUM_NUMS):
    # Generate a number that isn't already on the ticket
    rand = randrange(MIN_NUM, MAX_NUM + 1)
    while rand in ticket_nums:
        rand = randrange(MIN_NUM, MAX_NUM + 1)

    # Add the number to the ticket
    ticket_nums.append(rand)

# Sort the numbers into ascending order and display them
ticket_nums.sort()
print("Your numbers are: ", end="")
for n in ticket_nums:
    print(n, end=" ")
print()
```

Solution to Exercise 125: Shuffling a Deck of Cards

```
##
#  Create a deck of cards and shuffle it.
#
from random import randrange

## Construct a standard deck of cards with 4 suits and 13 values per suit
#  @return a list of cards, with each card represented by two characters
def createDeck():
    # Create a list to hold the cards
    cards = []
```

```
    # For each suit and each value
    for suit in ["s", "h", "d", "c"]:
        for value in ["2", "3", "4", "5", "6", "7", "8", "9", \
                      "T", "J", "Q", "K", "A"]:
            # Construct the card and add it to the list
            cards.append(value + suit)

    # Return the complete deck of cards
    return cards

## Shuffle a deck of cards by modifying the deck passed to the function
#  @param cards the list of cards to shuffle
#  @return (None)
def shuffle(cards):
    # For each card
    for i in range(0, len(cards)):
        # Pick a random index between the current index and the end of the list
        other_pos = randrange(i, len(cards))

        # Swap the current card with the one at the random position
        temp = cards[i]
        cards[i] = cards[other_pos]
        cards[other_pos] = temp

# Display a deck of cards before and after it has been shuffled
def main():
    cards = createDeck()
    print("The original deck of cards is: ")
    print(cards)
    print()

    shuffle(cards)
    print("The shuffled deck of cards is: ")
    print(cards)

# Call the main function only if this file has not been imported into another program
if __name__ == "__main__":
    main()
```

Solution to Exercise 128: Count the Elements

```
##
#  Count the number of elements in a list that are greater than or equal to some minimum
#  value and less than some maximum value.
#

## Determine how many elements in data are greater than or equal to mn and less than mx
#  @param data the list of values to examine
#  @param mn the minimum acceptable value
#  @param mx the exclusive upper bound on acceptability
#  @return the number of elements, e, such that mn <= e < mx
def countRange(data, mn, mx):
    # Count the number of elements within the acceptable range
    count = 0
    for e in data:
        # Check each element
        if mn <= e and e < mx:
            count = count + 1

    # Return the result
    return count
```

```
# Demonstrate the countRange function
def main():
  data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  # Test a case where some elements are within the range
  print("Counting the elements in [1..10] between 5 and 7...")
  print("Result: %d  Expected: 2" % countRange(data, 5, 7))

  # Test a case where all elements are within the range
  print("Counting the elements in [1..10] between -5 and 77...")
  print("Result: %d  Expected: 10" % countRange(data, -5, 77))

  # Test a case where no elements are within the range
  print("Counting the elements in [1..10] between 12 and 17...")
  print("Result: %d  Expected: 0" % countRange(data, 12, 17))

  # Test a case where the list is empty
  print("Counting the elements in [] between 0 and 100...")
  print("Result: %d  Expected: 0" % countRange([], 0, 100))

  # Test a case with duplicate values
  data = [1, 2, 3, 4, 1, 2, 3, 4]
  print("Counting the elements in", data, "between 2 and 4...")
  print("Result: %d  Expected: 4" % countRange(data, 2, 4))

# Call the main program
main()
```

Solution to Exercise 129: Tokenizing a String

```
##
#  Tokenize a string containing a mathematical expression.
#

## Convert a mathematical expression into a list of tokens
#  @param s the string to tokenize
#  @return a list of the tokens in s, or an empty list if an error occurs
def tokenList(s) :
  # Remove all of the spaces from s
  s = s.replace(" ", "")

  # Loop through all of the characters in the string, identifying the tokens and adding them to
  # the list
  tokens = []
  i = 0
  while i < len(s):
    # Handle the tokens that are always a single character: *, /, ^, ( and )
    if s[i] == "*" or s[i] == "/" or s[i] == "^" or \
       s[i] == "(" or s[i] == ")" or s[i] == "+" or s[i] == "-":
      tokens.append(s[i])
      i = i + 1
```

```
    # Handle a number without a leading + or -
    elif s[i] >= "0" and s[i] <= "9":
      num = ""
      # Keep on adding characters to the token as long as they are digits
      while i < len(s) and s[i] >= "0" and s[i] <= "9":
        num = num + s[i]
        i = i + 1
      tokens.append(num)

    # Any other character means the expression is not valid.  Return an empty list to indicate
    # that an error occurred.
    else:
      return []

  return tokens

# Read an expression from the user, tokenize it, and display the result
def main():
  exp = input("Enter a mathematical expression: ")
  tokens = tokenList(exp)
  print("The tokens are:", tokens)

# Call the main function only if this file has not been imported into another program
if __name__ == "__main__":
  main()
```

Solution to Exercise 130: Unary and Binary Operators

```
##
#  Differentiate between unary and binary + and - operators.
#
from token_list import tokenList

## Identify occurrences of unary + and - operators within a list of tokens and replace them
#  with u+ and u- respectively
#  @param tokens a list of tokens that may include unary + and - operators
#  @return a list of tokens where unary + and - operators have been replaced with u+ and u-
def identifyUnary(tokens):
  retval = []

  # Process each token in the list
  for i in range(len(tokens)):
    # If the first token in the list is + or - then it is a unary operator
    if i == 0 and (tokens[i] == "+" or tokens[i] == "-"):
      retval.append("u" + tokens[i])
    # If the token is a + or - and the previous token is an operator or an open parenthesis
    # then it is a unary operator
    elif i > 0 and (tokens[i] == "+" or tokens[i] == "-") and \
        (tokens[i-1] == "+" or tokens[i-1] == "-" or
         tokens[i-1] == "*" or tokens[i-1] == "/" or
         tokens[i-1] == "("):
      retval.append("u" + tokens[i])
    # Any other token is not a unary operator so it is appended to the result without modification
    else:
      retval.append(tokens[i])

  # Return the new list of tokens where the unary operators have been marked
  return retval
```

```
# Demonstrate that unary operators are marked correctly
def main():
  # Read an expression from the user, tokenize it, and display the result
  exp = input("Enter a mathematical expression: ")
  tokens = tokenList(exp)
  print("The tokens are:", tokens)

  # Identify the unary operators in the list of tokens
  marked = identifyUnary(tokens)
  print("With unary operators marked: ", marked)

# Call the main function only if this file has not been imported into another program
if __name__ == "__main__":
  main()
```

Solution to Exercise 134: Generate All Sublists of a List

```
##
#  Compute all of the sublists of a list.
#

## Generate a list of of all of the sublists of a list
#  @param data the list for which the sublists are generated
#  @return a list containing all of the sublists of data
def allSublists(data):
  # Start out with the empty list as the only sublist of data
  sublists = [[]]

  # Generate all of the sublists of data from length 1 to len(data)
  for length in range(1, len(data) + 1):
    # Generate the sublists starting at each index
    for i in range(0, len(data) - length + 1):
      # Add the current sublist to the list of sublists
      sublists.append(data[i : i + length])

  # Return the result
  return sublists

# Demonstrate the allSublists function
def main():
  print("The sublists of [] are: ")
  print(allSublists([]))

  print("The sublists of [1] are: ")
  print(allSublists([1]))

  print("The sublists of [1, 2] are: ")
  print(allSublists([1, 2]))

  print("The sublists of [1, 2, 3] are: ")
  print(allSublists([1, 2, 3]))

  print("The sublists of [1, 2, 3, 4] are: ")
  print(allSublists([1, 2, 3, 4]))

# Call the main function
main()
```

A list containing an empty list is denoted by `[[]]`.

Solution to Exercise 135: The Sieve of Eratosthenes

```
##
#  Identify all of the prime numbers from 2 to some limit entered by the user using the
#  Sieve of Eratosthenes.
#
# Read the limit from the user
limit = int(input("Identify all primes up to what limit? "))

# Create a list that contains all of the integers from 0 to limit
nums = []
for i in range(0, limit + 1):
  nums.append(i)

# ''Cross out'' 1 by replacing it with a 0
nums[1] = 0

# ''Cross out'' all of the multiples of each prime number that we discover
p = 2
while p < limit:
  # ''Cross out'' all multiples of p (but not p itself)
  for i in range(p*2, limit + 1, p):
    nums[i] = 0

  # Find the next number that is not ''crossed out''
  p = p + 1
  while p < limit and nums[p] == 0:
    p = p + 1

# Display the result
print("The primes up to", limit, "are:")
for i in nums:
  if nums[i] != 0:
    print(i)
```

Solutions to the Dictionary Exercises 14

Solution to Exercise 136: Reverse Lookup

```
##
# Conduct a reverse lookup on a dictionary, finding all of the keys that map to the provided
# value.
#

## Conduct a reverse lookup on a dictionary
# @param data the dictionary on which the reverse lookup is performed
# @param value the value to search for in the dictionary
# @return a list (possibly empty) of keys from data that map to value
def reverseLookup(data, value):
  # Construct a list of the keys that map to value
  keys = []

  # Check each key and add it to keys if the values match
  for key in data:
    if data[key] == value:
      keys.append(key)

  # Return the list of keys
  return keys
```

Each key in a dictionary must be unique. However, values may be repeated. As a result, performing a reverse lookup may identify zero, one or several keys that match the provided value.

```
# Demonstrate the reverseLookup function
def main():
  # A dictionary mapping 4 French words to their English equivalents
  frEn = {"le" : "the", "la" : "the", "livre" : "book", \
          "pomme" : "apple"}

  # Demonstrate the reverseLookup function with 3 cases: One that returns multiple keys,
  # one that returns one key, and one that returns no keys
  print("The french words for 'the' are: ", \
        reverseLookup(frEn, "the"))
  print("Expected: ['le', 'la']")
  print()
```


B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_14

```
  print("The french word for 'apple' is: ", \
        reverseLookup(frEn, "apple"))
  print("Expected: ['pomme']")
  print()

  print("The french word for 'asdf' is: ", \
        reverseLookup(frEn, "asdf"))
  print("Expected: []")

# Call the main function only if this file has not been imported into another program
if __name__ == "__main__":
  main()
```

Solution to Exercise 137: Two Dice Simulation

```
##
#  Simulate rolling two dice many times and compare the simulated results to the results
#  expected by probability theory.
#
from random import randrange

NUM_RUNS = 1000
D_MAX = 6

## Simulate rolling two six-sided dice
#  @return the total from rolling two simulated dice
def twoDice():
  # Simulate two dice
  d1 = randrange(1, D_MAX + 1)
  d2 = randrange(1, D_MAX + 1)

  # Return the total
  return d1 + d2

# Simulate many rolls and display the result
def main():
  # Create a dictionary of expected proportions
  expected = {2: 1/36, 3: 2/36, 4: 3/36, 5: 4/36, 6: 5/36, \
              7: 6/36, 8: 5/36, 9: 4/36,  10: 3/36, \
              11: 2/36, 12: 1/36}

  # Create a dictionary that maps from the total of two dice to the number of occurrences
  counts = {2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, \
            8: 0, 9: 0, 10: 0, 11: 0, 12: 0}
```

Each dictionary is initialized so that it has keys 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12. In the `expected` dictionary, the value is initialized to the probability that each key will occur when two 6-sided dice are rolled. In the `counts` dictionary, each value is initialized to 0. The values in `counts` are increased as the simulation runs.

```
  # Simulate NUM_RUNS rolls, and count each roll
  for i in range(NUM_RUNS):
    t = twoDice()
    counts[t] = counts[t] + 1
```

```
  # Display the simulated proportions and the expected proportions
  print("Total   Simulated   Expected")
  print("          Percent     Percent")
  for i in sorted(counts.keys()):
    print("%5d %11.2f   %8.2f" % \
          (i, counts[i] / NUM_RUNS * 100, expected[i] * 100))

# Call the main function
main()
```

Solution to Exercise 142: Unique Characters

```
##
#  Compute the number of unique characters in a string using a dictionary.
#

# Read the string from the user
s = input("Enter a string: ")

# Add each character to a dictionary with a value of True.  Once we are done the number
# of keys in the dictionary will be the number of unique characters in the string.
characters = {}
for ch in s:
  characters[ch] = True
```

Every key in a dictionary must have a value associated with it. However, in this solution the value is never used. As a result, we have elected to associate `True` with each key, but any other value could have been used instead of `True`.

```
# Display the result
print("That string contained", len(characters), \
      "unique character(s).")
```

The `len` function returns the number of keys in a dictionary.

Solution to Exercise 143: Anagrams

```
##
#  Determine whether or not two strings are anagrams and report the result.
#

## Compute the frequency distribution for the characters in a string
#  @param s the string to process
#  @return a dictionary mapping each character to its count
def characterCounts(s):
  # Create a new, empty dictionary
  counts = {}

  # Update the count for each character in the string
  for ch in s:
    if ch in counts:
      counts[ch] = counts[ch] + 1
    else:
      counts[ch] = 1

  # Return the result
  return counts
```

```
# Determine if two strings entered by the user are anagrams
def main():
  # Read the strings from the user
  s1 = input("Enter the first string: ")
  s2 = input("Enter the second string: ")

  # Get the character counts for each string
  counts1 = characterCounts(s1)
  counts2 = characterCounts(s2)

  # Display the result
  if counts1 == counts2:
    print("Those strings are anagrams.")
  else:
    print("Those strings are not anagrams.")
```

Two dictionaries are equal if and only if both dictionaries have the same keys and for every key, *k*, the value associated with *k* is the same in both dictionaries.

```
# Call the main function
main()
```

Solution to Exercise 145: Scrabble™ Score

```
##
#  Use a dictionary to compute the Scrabble™ score for a word.
#

# Initialize the dictionary so that it maps from letters to points
points = {"A": 1, "B": 3, "C": 3, "D": 2, "E": 1, "F": 4, \
          "G": 2, "H": 4, "I": 1, "J": 2, "K": 5, "L": 1, \
          "M": 3, "N": 1, "O": 1, "P": 3, "Q": 10, "R": 1, \
          "S": 1, "T": 1, "U": 1, "V": 4, "W": 4, "X": 8,
          "Y": 4, "Z": 10}

# Read a word from the user
word = input("Enter a word: ")

# Compute the score for the word
uppercase = word.upper()
score = 0
for ch in uppercase:
  score = score + points[ch]
```

The word is converted to uppercase so that the correct result is computed when the user enters the word in upper, mixed or lowercase. This could also be accomplished by adding all of the lowercase letters to the dictionary.

```
# Display the result
print(word, "is worth", score, "points.")
```

Solution to Exercise 146: Create a Bingo Card

```
##
# Create and display a random Bingo card.
#
from random import randrange

NUMS_PER_LETTER = 15

## Create a Bingo card with randomly generated numbers
# @return a dictionary representing the card where the keys are the strings ‘‘B’’, ‘‘I’’, ‘‘N’’,
#         ‘‘G’’, and ‘‘O’’, and the values are lists of the numbers that appear under each letter
#         from top to bottom
def createCard():
  card = {}

  # The range of integers that can be generated for the current letter
  lower = 1
  upper = 1 + NUMS_PER_LETTER

  # For each of the five letters
  for letter in ["B", "I", "N", "G", "O"]:
    # Start with an empty list for the letter
    card[letter] = []

    # Keep generating random numbers until we have 5 unique ones
    while len(card[letter]) != 5:
      next_num = randrange(lower, upper)
      # Ensure that we do not include any duplicate numbers
      if next_num not in card[letter]:
        card[letter].append(next_num)

    # Update the range of values that will be generated for the next letter
    lower = lower + NUMS_PER_LETTER
    upper = upper + NUMS_PER_LETTER

  # Return the card
  return card

## Display a Bingo card with nice formatting
# @param card the Bingo card to display
# @return (None)
def displayCard(card):
  # Display the headings
  print("B   I   N   G   O")

  # Display the numbers on the card
  for i in range(5):
    for letter in ["B", "I", "N", "G", "O"]:
      print("%2d " % card[letter][i], end="")
    print()

# Create a random Bingo card and display it
def main():
  card = createCard()
  displayCard(card)

# Call the main function only if this file has not been imported into another program
if __name__ == "__main__":
  main()
```

15 Solutions to the File and Exception Exercises

Solution to Exercise 149: Display the Head of a File

```
##
# Display the head (first 10 lines) of a file whose name is provided as a command line argument.
#
import sys

NUM_LINES = 10

# Verify that exactly one command line argument (in addition to the .py file) was supplied
if len(sys.argv) != 2:
  print("Provide the file name as a command line argument.")
  quit()

try:
  # Open the file for reading
  inf = open(sys.argv[1], "r")

  # Read the first line from the file
  line = inf.readline()

  # Keep looping until we have either read and displayed 10 lines or we have reached the end
  # of the file
  count = 0
  while count < NUM_LINES and line != "":
    # Remove the trailing newline character and count the line
    line = line.rstrip()
    count = count + 1

    # Display the line
    print(line)

    # Read the next line from the file
    line = inf.readline()

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

When the `quit` function is called the program ends immediately.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_15

```
except IOError:
  # Display a message if something goes wrong while accessing the file
  print("An error occurred while accessing the file.")
```

Solution to Exercise 150: Display the Tail of a File

```
##
# Display the tail (last lines) of a file whose name is provided as a command line argument.
#
import sys

NUM_LINES = 10

# Verify that exactly one command line argument (in addition to the .py file) was provided
if len(sys.argv) != 2:
  print("Provide the file name as a command line argument.")
  quit()

try:
  # Open the file for reading
  inf = open(sys.argv[1], "r")

  # Read through the file, always saving the NUM_LINES most recent lines
  lines = []
  for line in inf:
    # Add the most recent line to the end of the list
    lines.append(line)
    # If we now have more than NUM_LINES lines then remove the oldest one
    if len(lines) > NUM_LINES:
      lines.pop(0)

  # Close the file
  inf.close()

except:
  print("An error occurred while processing the file.")
  quit()

# Display the last lines of the file
for line in lines:
  print(line, end="")
```

Solution to Exercise 151: Concatenate Multiple Files

```
##
# Concatenate one or more files and display the result.
#
import sys

# Ensure that at least one command line argument (in addition to the .py file) has been provided
if len(sys.argv) == 1:
  print("You must provide at least one file name.")
  quit()
```

```
# Process all of the files provided on the command line
for i in range(1, len(sys.argv)):
  fname = sys.argv[i]
  try:
    # Open the current file for reading
    inf = open(fname, "r")

    # Display the file
    for line in inf:
      print(line, end="")

    # Close the file
    inf.close()

  except:
    # Display a message, but do not quit, so that the program will go on and process any
    # subsequent files
    print("Couldn't open/display", fname)
```

The element at position 0 in `sys.argv` is the Python file that is being executed. As a result, our `for` loop starts processing file names at position 1 in the list.

Solution to Exercise 156: Sum a Collection of Numbers

```
##
#  Compute the sum of numbers entered by the user, ignoring non-numeric input.
#

# Read the first line of input from the user
line = input("Enter a number: ")
total = 0

# Keep reading until the user enters a blank line
while line != "":
  try:
    # Try and convert the line to a number
    num = float(line)
    # If the conversion succeeds then add it to the total and display it
    total = total + num
    print("The total is now", total)

  except ValueError:
    # Display an error message before going on to read the next value
    print("That wasn't a number.")

  # Read the next number
  line = input("Enter a number: ")

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

Solution to Exercise 158: Remove Comments

```
##
#  Remove all of the comments from a Python file (ignoring the case where a comment
#  character occurs within a string)
#

# Read the file name and open the input file
try:
  in_name = input("Enter the name of a Python file: ")
  inf = open(in_name, "r")
```

```
except:
    # Display an error message and quit if the file was not opened successfully
    print("A problem was encountered with the input file.")
    print("Quitting...")
    quit()

# Read the file name and open the output file
try:
    out_name = input("Enter the output file name: ")
    outf = open(out_name, "w")

except:
    # Close the input file, display an error message and quit if the file was not opened
    # successfully
    inf.close()
    print("A problem was encountered with the output file.")
    print("Quitting...")
    quit()

try:
    # Read all of the lines from the input file, remove the comments from them, and save the
    # modified lines to a new file
    for line in inf:
        # Find the position of the comment character (-1 if there isn't one)
        pos = line.find("#")

        # If there is a comment then create a slice of the string that excludes it and store it back
        # into line
        if pos > -1:
            line = line[0 : pos]
            line = line + "\n"

        # Write the (potentially modified) line to the file
        outf.write(line)

    # Close the files
    inf.close()
    outf.close()

except:
    # Display an error message if something went wrong while processing the file
    print("A problem was encountered while processing the file.")
    print("Quitting...")
```

The position of the comment character is stored in `pos`. As a result, `line[0 : pos]` is all of the characters up to but not including the comment character.

Solution to Exercise 159: Two Word Random Password

```
##
#  Generate a password by concatenating two random words.  The password will be  between
#  8 and 10 letters, and each word will be at least three letters long.
#
from random import randrange

WORD_FILE = "../Data/words.txt"
```

The password we are creating will be 8, 9 or 10 letters. Since the shortest acceptable word is 3 letters, and a password must have 2 words in it, a password can never contain a word that is longer than 7 letters.

```
# Read all of the words from the file, only keeping those between 3 and 7 letters in length,
# and store them in a list
words = []
inf = open(WORD_FILE, "r")
for line in inf:
  # Remove the newline character
  line = line.rstrip()

  # Keep words that are between 3 and 7 letters long
  if len(line) >= 3 and len(line) <= 7:
    words.append(line)

# Close the file
inf.close()

# Randomly select the first word for the password.  It can be any word.
first = words[randrange(0, len(words))]
first = first.capitalize()

# Keep selecting a second word until we find one that doesn't make the password too short
# or too long
password = first
while len(password) < 8 or len(password) > 10:
  second = words[randrange(0, len(words))]
  second = second.capitalize()
  password = first + second

# Display the random password
print("The random password is:", password)
```

Solution to Exercise 162: A Book with No E...

```
##
#  Determine and display the proportion of words that include each letter of the alphabet.  The
#  letter that is used in the smallest proportion of words is highlighted at the end of the
#  program's output.
#
WORD_FILE = "../Data/words.txt"

# Create a dictionary that counts the number of words containing each letter.  Initialize the
# count for each letter to 0.
counts = {}
for ch in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
  counts[ch] = 0

# Open the file, process each word, and update the counts dictionary
num_words = 0
inf = open(WORD_FILE, "r")
for word in inf:
  # Convert the word to uppercase and remove the newline character
  word = word.upper().rstrip()
```

```
    # Before we can update the dictionary we need to generate a list of the unique letters in the
    # word.  Otherwise we will increase the count multiple times for words that contain repeated
    # letters.  We also need to ignore any non-letter characters that might be present.
    unique = []
    for ch in word:
      if ch not in unique and ch >= "A" and ch <= "Z":
        unique.append(ch)

    # Now increment the counts for all of the letters that are in the list of unique characters
    for ch in unique:
      counts[ch] = counts[ch] + 1

    # Keep track of the number of words that we have processed
    num_words = num_words + 1

# Close the file
inf.close()

# Display the result for each letter.  While displaying the results we will also determine which
# character had the smallest count so that we can display it again at the end of the program.
smallest_count = min(counts.values())
for ch in sorted(counts):
  if counts[ch] == smallest_count:
    smallest_letter = ch
  percentage = counts[ch] / num_words * 100
  print(ch, "occurs in %.2f percent of words" % percentage)

# Display the letter that is easiest to avoid based on the number of words in which it appears
print()
print("The letter that is easiest to avoid is", smallest_letter)
```

Solution to Exercise 163: Names That Reached Number One

```
##
#  Display all of the girls' and boys' names that were the most popular in at least one year
#  between 1900 and 2012.
#
FIRST_YEAR = 1900
LAST_YEAR = 2012

## Load the first line from the file, extract the name, and add it to the names list if it is not
#  already present.
#  @param fname the name of the file from which the data will be read
#  @param names the list to add the name to (if it isn't already present)
#  @return (None)
def LoadAndAdd(fname, names):
  # Open the file, read the first line, and extract the name
  inf = open(fname, "r")
  line = inf.readline()
  inf.close()
  parts = line.split()
  name = parts[0]

  # Add the name to the list if it is not already present
  if name not in names:
    names.append(name)
```

```
# Display the girls' and boys' names that reached #1 in at least one year between 1900 and 2012
def main():
  # Create two lists to store the most popular names
  girls = []
  boys = []

  # Process each year in the range, reading the first line out of the girl file and the boy file
  for year in range(FIRST_YEAR, LAST_YEAR + 1):
    girl_fname = "../Data/BabyNames/" + str(year) + \
                 "_GirlsNames.txt"
    boy_fname = "../Data/BabyNames/" + str(year) + \
                "_BoysNames.txt"
```

My solution assumes that the baby names data files are stored in a different folder than the Python program. If you have the data files in the same folder as your program then `../Data/BabyNames/` should be omitted.

```
    LoadAndAdd(girl_fname, girls)
    LoadAndAdd(boy_fname, boys)

  # Display the lists
  print("Girls' names that reached #1:")
  for name in girls:
    print(" ", name)
  print()

  print("Boys' names that reached #1: ")
  for name in boys:
    print(" ", name)

# Call the main function
main()
```

Solution to Exercise 167: Spell Checker

```
##
#  Find and list all of the words in a file that are misspelled.
#
from only_words import onlyTheWords
import sys

WORDS_FILE = "../Data/words.txt"

# Ensure that the program has the correct number of command line arguments
if len(sys.argv) != 2:
  print("One command line argument must be provided.")
  print("Quitting...")
  quit()

# Open the file.  Quit if the file is not opened successfully.
try:
  inf = open(sys.argv[1], "r")
except:
  print("Failed to open '%s' for reading.  Quitting..." % \
        sys.argv[1])
  quit()
```

```
# Load all of the words into a dictionary of valid words.  The value 0 is associated with each
# word, but it is never used.
valid = {}
words_file = open(WORDS_FILE, "r")

for word in words_file:
    # Convert the word to lowercase and remove the trailing newline character
    word = word.lower().rstrip()

    # Add the word to the dictionary
    valid[word] = 0

words_file.close()
```

This solution uses a dictionary where the keys are the valid words, but the values in the dictionary are never used. As a result, a set would be a better choice if you are familiar with that data structure. A list is not used because checking if a key is in a dictionary is faster than checking if an element is in a list.

```
# Read each line from the file, adding any misspelled words to the list of misspellings
misspelled = []
for line in inf:
    # Discard the punctuation marks by calling the function developed in Exercise 117
    words = onlyTheWords(line)
    for word in words:
        # Only add to the misspelled list if the word is misspelled and not already in the list
        if word.lower() not in valid and word not in misspelled:
            misspelled.append(word)

# Close the file being checked
inf.close()

# Display the misspelled words, or a message indicating that no words are misspelled
if len(misspelled) == 0:
    print("No words were misspelled.")
else:
    print("The following words are misspelled:")
    for word in misspelled:
        print("  ", word)
```

Solution to Exercise 169: Redacting Text in a File

```
##
#  Redact a text file by removing all occurrences of sensitive words.  The redacted version
#  of the text is written to a new file.
#
#  Note that this program does not perform any error checking, and it does not implement
#  case insensitive redaction.
#

# Get the name of the input file and open it
inf_name = input("Enter the name of the text file to redact: ")
inf = open(inf_name, "r")
```

```
# Get the name of the sensitive words file and open it
sen_name = input("Enter the name of the sensitive words file: ")
sen = open(sen_name, "r")

# Load all of the sensitive words into a list
words = []
line = sen.readline()
while line != "":
  line = line.rstrip()
  words.append(line)

  line = sen.readline()

# Close the sensitive words file
sen.close()
```

> The sensitive word file can be closed at this point because all of the words have been read into a list. No further data needs to be read from that file.

```
# Get the name of the output file and open it
outf_name = input("Enter the name for the new redacted file: ")
outf = open(outf_name, "w")

# Read each line from the input file.  Replace all of the sensitive words with asterisks.  Then
# write the line to the output file.
line = inf.readline()
while line != "":
  # Check for and replace each sensitive word.  The number of asterisks matches the number
  # of letters in the word being redacted.
  for word in words:
    line = line.replace(word, "*" * len(word))

  # Write the modified line to the output file
  outf.write(line)

  # Read the next line from the input file
  line = inf.readline()

# Close the input and output files
inf.close()
outf.close()
```

Solution to Exercise 170: Missing Comments

```
##
# Find and display the names of Python functions that are not immediately preceded by a
# comment.
#
from sys import argv
```

```
# Verify that at least one file name has been provided as a command line argument
if len(argv) == 1:
  print("At least one filename must be provided as a", \
        "command line argument.")
  print("Quitting...")
  quit()

# Process each file provided as a command line argument
for fname in argv[1 : len(argv)]:
  # Attempt to process the file
  try:
    inf = open(fname, "r")

    # As we move through the file we need to keep a copy of the previous line so that we
    # can check to see if it starts with a comment character.  We also need to keep track
    # of the line number within the file.
    prev = " "
    lnum = 1
```

The `prev` variable must be initialized to a string that is at least one character in length. Otherwise the program will crash when the first line in the file that is being checked is a function definition.

```
    # Check each line in the current file
    for line in inf:
      # If the line is a function definition and the previous line is not a comment
      if line.startswith("def ") and prev[0] != "#":
        # Find the first ( on the line so that we know where the function name ends
        bracket_pos = line.index("(")
        name = line[4 : bracket_pos]

        # Display information about the function that is missing its comment
        print("%s line %d: %s" % (fname, lnum, name))

      # Save the current line and update the line counter
      prev = line
      lnum = lnum + 1

    # Close the current file
    inf.close()

  except:
    print("A problem was encountered with file '%s'." % fname)
    print("Moving on to the next file...")
```

16 Solutions to the Recursion Exercises

Solution to Exercise 173: Total the Values

```
##
#  Total a collection of numbers entered by the user.  The user will enter a blank line to
#  indicate that no further numbers will be entered and the total should be displayed.
#

## Total all of the numbers entered by the user until the user enters a blank line
#  @return the total of the entered values
def readAndTotal():
    # Read a value from the user
    line = input("Enter a number (blank to quit): ")

    # Base case: The user entered a blank line so the total is 0
    if line == "":
        return 0
    else:
        # Recursive case: Convert the current line to a number and use recursion to read the
        # subsequent lines
        return float(line) + readAndTotal()

# Read a collection of numbers from the user and display the total
def main():
    # Read the values from the user and compute the total
    total = readAndTotal()

    # Display the total
    print("The total of all those values is", total)

# Call the main function
main()
```

Solution to Exercise 178: Recursive Palindrome

```
##
#  Determine whether or not a string entered by the user is a palindrome using recursion.
#
```


B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_16

```
## Determine whether or not a string is a palindrome
#  @param s the string to check
#  @return True if the string is a palindrome, False otherwise
def isPalindrome(s):
  # Base case: The empty string is a palindrome.  So is a string containing only 1 character.
  if len(s) <= 1:
    return True

  # Recursive case: The string is a palindrome only if the first and last characters match, and
  # the rest of the string is a palindrome
  return s[0] == s[len(s) - 1] and \
         isPalindrome(s[1 : len(s) - 1])

# Check whether or not a string entered by the user is a palindrome
def main():
  # Read the string from the user
  line = input("Enter a string: ")

  # Check its status and display the result
  if isPalindrome(line):
    print("That was a palindrome!")
  else:
    print("That is not a palindrome.")

# Call the main function
main()
```

Solution to Exercise 180: String Edit Distance

```
##
#  Compute and display the edit distance between two strings.
#

## Compute the edit distance between two strings.  The edit distance is the minimum number of
#  insert, delete and substitute operations needed to transform one string into the other.
#  @param s the first string
#  @param t the second string
#  @return the edit distance between the strings
def editDistance(s, t):
  # If one string is empty, then the edit distance is one insert operation for each letter in the
  # other string
  if len(s) == 0:
    return len(t)
  elif len(t) == 0:
    return len(s)
  else:
    cost = 0
    # If the last characters are not equal then set cost to 1
    if s[len(s) - 1] != t[len(t) - 1]:
      cost = 1

    # Compute three distances
    d1 = editDistance(s[0 : len(s) - 1], t) + 1
    d2 = editDistance(s, t[0 : len(t) - 1]) + 1
    d3 = editDistance(s[0 : len(s) - 1] , t[0 : len(t) - 1]) + \
         cost

    # Return the minimum distance
    return min(d1, d2, d3)
```

```
# Compute the edit distance between two strings entered by the user
def main():
  # Read two strings from the user
  s1 = input("Enter a string: ")
  s2 = input("Enter another string: ")

  # Compute and display the edit distance
  print("The edit distance between %s and %s is %d." % \
        (s1, s2, editDistance(s1, s2)))

# Call the main function
main()
```

Solution to Exercise 183: Element Sequences

```
##
#  Identify the longest sequence of elements that can follow an element entered by the
#  user where each element in the sequence begins with the same letter as the last letter
#  of its predecessor.
#
ELEMENTS_FNAME = "../Data/Elements.csv"

## Find the longest sequence of words, beginning with start, where each word begins with
#  the last letter of its predecessor
#  @param start the first word in the sequence
#  @param words the list of words that can occur in the sequence
#  @return the longest list of words beginning with start that meets the constraints
#          outlined previously
def longestSequence(start, words):
  # Base case: If start is empty then the list of words is empty
  if start == "":
    return []

  # Find the best (longest) list of words by checking each possible word that could appear
  # next in the sequence
  best = []
  last_letter = start[len(start) - 1].lower()
  for i in range(0, len(words)):
    first_letter = words[i][0].lower()

    # If the first letter of the next word matches the last letter of the previous word
    if first_letter == last_letter:
      # Use recursion to find a candidate sequence of words
      candidate = longestSequence(words[i], \
                  words[0 : i] + words[i + 1 : len(words)])
      # Save the candidate if it is better than the best sequence that we have seen previously
      if len(candidate) > len(best):
        best = candidate

  # Return the best candidate sequence, preceded by the starting word
  return [start] + best

## Load the names of all of the elements from the elements file
#  @return a list of all the element names
def loadNames():
  names = []

  # Open the element data set
  inf = open(ELEMENTS_FNAME, "r")
```

```
    # Load each line, storing the element name in a list
    for line in inf:
        line = line.rstrip()
        parts = line.split(",")
        names.append(parts[2])

    # Close the file and return the list
    inf.close()
    return names

# Display a longest sequence of elements starting with an element entered by the user
def main():
    # Load all of the element names
    names = loadNames()

    # Read the starting element and capitalize it
    start = input("Enter the name of an element: ")
    start = start.capitalize()

    # Verify that the value entered by the user is an element
    if start in names:
        # Remove the starting element from the list of elements
        names.remove(start)

        # Find a longest sequence that begins with the starting element
        sequence = longestSequence(start, names)

        # Display the sequence of elements
        print("A longest sequence that starts with", start, "is:")
        for element in sequence:
            print(" ", element)
    else:
        print("Sorry, that wasn't a valid element name.")

# Call the main function
main()
```

Solution to Exercise 184: Flatten a List

```
##
#  Use recursion to flatten a list that may contain nested lists.
#

## Flatten a list so that all nested lists are removed
#  @param data the list to flatten
#  @return a flattened version of data
def flatten(data):
    # If data is empty then there is no work to do
    if data == []:
        return []

    # If the first element in data is a list
    if type(data[0]) == list:
        # Flatten the first element and flatten the remaining elements in the list
        return flatten(data[0]) + flatten(data[1:])
    else:
        # The first element in data is not a list so only the remaining elements in the list need
        # to be flattened
        return [data[0]] + flatten(data[1:])
```

```
# Demonstrate the flatten function
def main():
  print(flatten([1, [2, 3], [4, [5, [6, 7]]], [[[8], 9], [10]]]))
  print(flatten([1, [2, [3, [4, [5, [6, [7, [8, [9, \
                 [10]]]]]]]]]]))
  print(flatten([[[[[[[[[[1], 2], 3], 4], 5], 6], 7], 8], 9], \
                 10]))
  print(flatten([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
  print(flatten([]))

# Call the main function
main()
```

Solution to Exercise 186: Run-Length Encoding

```
##
# Perform run-length encoding on a string or a list using recursion.
#

## Perform run-length encoding on a string or a list
# @param data the string or list to encode
# @return a list where the elements at even positions are data values and the elements at odd
#         positions are counts of the number of times that the data value ahead of it should be
#         replicated
def encode(data):
  # If data is empty then no encoding work is necessary
  if len(data) == 0:
    return []
```

> If we performed the comparison `data == ""` then this function would only work for strings. If we performed the comparison `data == []` then it would only work for lists. Checking the length of the parameter allows the function to work for both strings and lists.

```
  # Find the index of the first item that is not the same as the item at position 0 in data
  index = 1
  while index < len(data) and data[index] == data[index - 1]:
    index = index + 1

  # Encode the current character group
  current = [data[0], index]

  # Use recursion to process the characters from index to the end of the string
  return current + encode(data[index : len(data)])

# Demonstrate the encode function
def main():
  # Read a string from the user
  s = input("Enter some characters: ")

  # Display the result
  print("When those characters are run-length encoded," \
        "the result is:", encode(s))

# Call the main function
main()
```

Index

Symbols
!=, 24
π, 50
**, 4
+, 4, 11
//, 4
==, 24
#, 8
%, 4, 9, 10
>, 24
>=, 24
<, 24
<=, 24

A
A Void, 122
Acceleration, 17
Access mode, 110
Ace, 88
Admission, 49
Algorithm, 3, 20, 51, 53, 54, 92–94, 133, 135, 138
Alice's Adventures in Wonderland, 125
Alphabet, 30, 51, 121, 133
Anagram, 105, 106
`and` operator, 28, 29
Anonymous Gregorian Computus algorithm, 20
`append` method, 79, 82
Append mode, 110, 113
Approximation, 50, 51, 135
Area, 12, 13, 16, 18
Argument, 5, 60, 62, 79, 82, 101
Argument vector, 114
`argv`, 114
Ascending order, 81
ASCII, 71, 109
Assignment operator, 4, 5, 76, 83, 98
Astrology, 35
Average, 47, 86

B
Baby names, 122, 123
Banknote, 33
Base case, 127, 129, 131
Binary, 54, 133
Binary file, 109
Binary operator, 91
Bingo, 106, 107
Body, 23, 24, 43, 45, 59
Body mass index, 20
Boolean expression, 23, 28
Boolean operator, 28
Boole, George, 23
Bread, 22

C
Caesar cipher, 50
Capitalize, 69, 120
Cards, 88, 89
`cat`, 118
Cell phone, 39, 103
Celsius, 16, 18, 21, 47
Chemical element, 117, 121, 136
Chess, 34
Circle, 16

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

`close` method, 110
Clubs, 88
Coin, 15, 56, 136
Collatz conjecture, 53
Command line argument, 114
Comments, 8, 120, 124
Common ratio, 63
Compression, 138
Computer program, 3
Concatenate, 118
Concatenation, 11, 113
Condition, 23, 25, 43
Consonant, 30, 87
Coordinate, 48, 88
Cup, 73
Cylinder, 17

D
Date, 19, 33, 35, 40, 68, 73
Day, 19, 20, 30, 33, 35, 40, 67, 68, 72, 73
Decibels, 31
Decimal, 8, 54, 72, 133
Decision making constructs, 23
Deck of cards, 88, 89
`def`, 59, 124
Default value, 62, 135
Denominator, 73
Deposit, 13
Diamonds, 88
Dice, 102
Dictionary, 97–99
Difference, 14
Digit, 21, 41, 70, 72, 91, 133
Discount, 22, 47
Discriminant, 37
Distance, 15–17, 48, 66
Division algorithm, 54
Dog years, 30

E
Earth, 15, 40, 95
Earthquake, 36
Easter, 19
Edit distance, 135
Element, 79, 80, 90
Empty dictionary, 97, 99
Empty list, 75
Encode, 51, 103, 138
Encryption, 50, 119
End of line, 112, 113
Equilateral, 31, 69
Eratosthenes, 95
Escape sequence, 113
Euclid's algorithm, 133
Even parity, 49
Exception, 115
`except` keyword, 115, 117
Exponential growth, 131

F
Fahrenheit, 18, 21, 47
`False`, 23, 28
Fibonacci number, 129
File, 109
`FileNotFoundError`, 116
File object, 110
Fizz-Buzz, 50
Flattening a list, 138
`float` function, 6
Floating-point number, 6, 8
Floor division, 4
`for` loop, 44–46, 76, 100
Format specifier, 8, 9
Formatting operator, 9
Frequency, 32, 39
Frequency analysis, 119
Fuel efficiency, 14
Function, 5, 59, 60
Function definition, 59

G
Gadsby, 121
Geometric sequence, 63
Grade points, 37, 38, 48, 120
Greatest common divisor, 53, 73, 132
Gregorian calendar, 19, 68

H
`head`, 117
Heads, 56
Hearts, 88
Heat capacity, 16
Height, 16–18, 20
Hexadecimal, 72
Holiday, 33
Horoscope, 35
Hour, 19
Hypotenuse, 66

I
Ideal gas law, 17
`if` statement, 23, 65

if-elif statement, 27
if-elif-else statement, 25
if-else statement, 25
import keyword, 7, 65
Index, 11, 75, 78
index method, 81
Infinite series, 50
Infix, 92
in operator, 81, 99
Input, 3, 110, 114
input function, 6
insert method, 79
Integer, 5, 6, 8
Interest, 14
int function, 6
Isosceles, 31

J
Jack, 88

K
Kelvin, 17, 21
Key, 97, 98
Key-value pair, 97, 98
King, 88

L
La Disparition, 122
Latitude, 15
Leap year, 30, 40, 68, 72
len function, 11, 77, 99
Letter grade, 37, 38, 48, 120
Library of Alexandria, 95
License plate, 41, 71
Linear growth, 131
Line number, 111, 118
Line of best fit, 88
Lipogram, 122
List, 75
Local variable, 63
Longest word, 118
Longitude, 15
Loop, 43
Lottery, 87

M
Magic date, 73
Mailing address, 12
main function, 65
Mathematical operators, 4
math module, 7
Maximum, 55
Median, 66
Method, 79
Minute, 19
Module, 7, 65
Modulo, 4
Money, 14, 32
Month, 20, 30, 33, 35, 40, 68, 72, 73
Morse code, 103
Multiplication table, 52
Music note, 31
Mutually exclusive, 25

N
NATO phonetic alphabet, 133
Nested if statement, 27
Nested list, 137
Nested loop, 46
Newton's method, 51
Nickel, 15, 48, 136
not operator, 28
Numerator, 73

O
Octave, 31
Odd parity, 49
open function, 110
Open problem, 53
Operator, 4, 9, 70, 90, 91
Order of operations, 5
Ordinal date, 68
Ordinal number, 67
or operator, 28, 29
Output, 7, 8, 109, 113

P
Palindrome, 51, 52, 86, 134
Paragraph, 125, 126
Parameter, 60, 62, 82, 101
Parity, 49
Password, 71, 72, 120
Penny, 15, 48
Perfect number, 85
Pig Latin, 87
Playing cards, 88
Polygon, 18, 48
pop method, 80, 82, 99
Postal code, 104
Postfix, 92, 93
Precedence, 5, 70, 92, 93
Pressure, 17, 21

Prime factorization, 54
Prime number, 54, 70, 71, 94, 95
`print` function, 7, 10, 75, 98
Product, 14, 52
Programmer, 3
Programming, 3
Programming language, 4
Prompt, 6
Proper divisor, 85
Proton, 121
Province, 104
Punctuation mark, 85
Pythagorean theorem, 15, 66
Python, 4

Q
Quadratic equation, 37
Queen, 88
Quotient, 14

R
`radians` function, 15
Radiation, 39
Radius, 15–17
Random, 41, 55, 56, 71, 72, 87, 89, 106, 107, 120
`range` function, 45, 76, 77
`readline` method, 110, 111
`read` method, 111
Read mode, 110
Recursion, 127
Recursive case, 127, 128, 131
Recursive function, 127
Redact, 124
Relational operator, 24
Remainder, 4, 14
`remove` method, 80
Repeated word, 123
`return` keyword, 63, 82, 101
Reverse lookup, 102
`reverse` method, 81
Reverse order, 83
Richter scale, 36
Roman numeral, 134
Roulette, 41
Rounded, 8, 48
`round` function, 5
`rstrip` method, 112
Run-length encoding, 138
Rural, 104

S
Scalene, 31
Scrabble™, 106
Season, 34
Second, 19
Sequence, 53, 63, 129
Shape, 30
Shipping, 66
Sieve of Eratosthenes, 94
Slicing a string, 12
Sorted, 22, 90
`sorted` function, 83
`sort` method, 81–83
Spades, 88
Spell checker, 123
Spelling, 123
Spelling alphabet, 133
Sphere, 16
`sqrt` function, 8, 63
Square root, 8, 51, 135
`str` function, 113
String, 6, 8, 11, 112
String concatenation, 11, 113
String length, 11
String similarity, 135
Sublist, 93, 94
Sum, 13, 14, 21, 119, 127
Swap, 80, 89
Syntax, 4
`sys` module, 114

T
Tablespoon, 73
`tail`, 118
Tail of a string, 131
Tails, 56
Tax, 13, 39
Taxi, 66
Teaspoon, 73
Temperature, 16, 17, 21, 47
Text file, 109, 112
Text message, 39, 103
Time, 19
Tip, 13
Token, 90, 91
Triangle, 18, 31, 66, 69
`True`, 23, 28
Truth table, 28
`try` keyword, 115, 117
The Twelve Days of Christmas, 67

U
Unary operator, 91
Urban, 104

V
Value, 97, 98
`values` method, 99, 100
Variable, 4, 62
Visible light, 38, 39
Volume, 16, 17, 73
Vowel, 30, 87, 126

W
Wavelength, 38
Week, 40
`while` loop, 43, 46, 78, 100, 111
Wind chill, 21
`write` method, 113
Write mode, 110, 113

Y
Year, 20, 30, 36, 40, 68, 72, 73, 122

Z
Zodiac, 35, 36
Zoo, 49