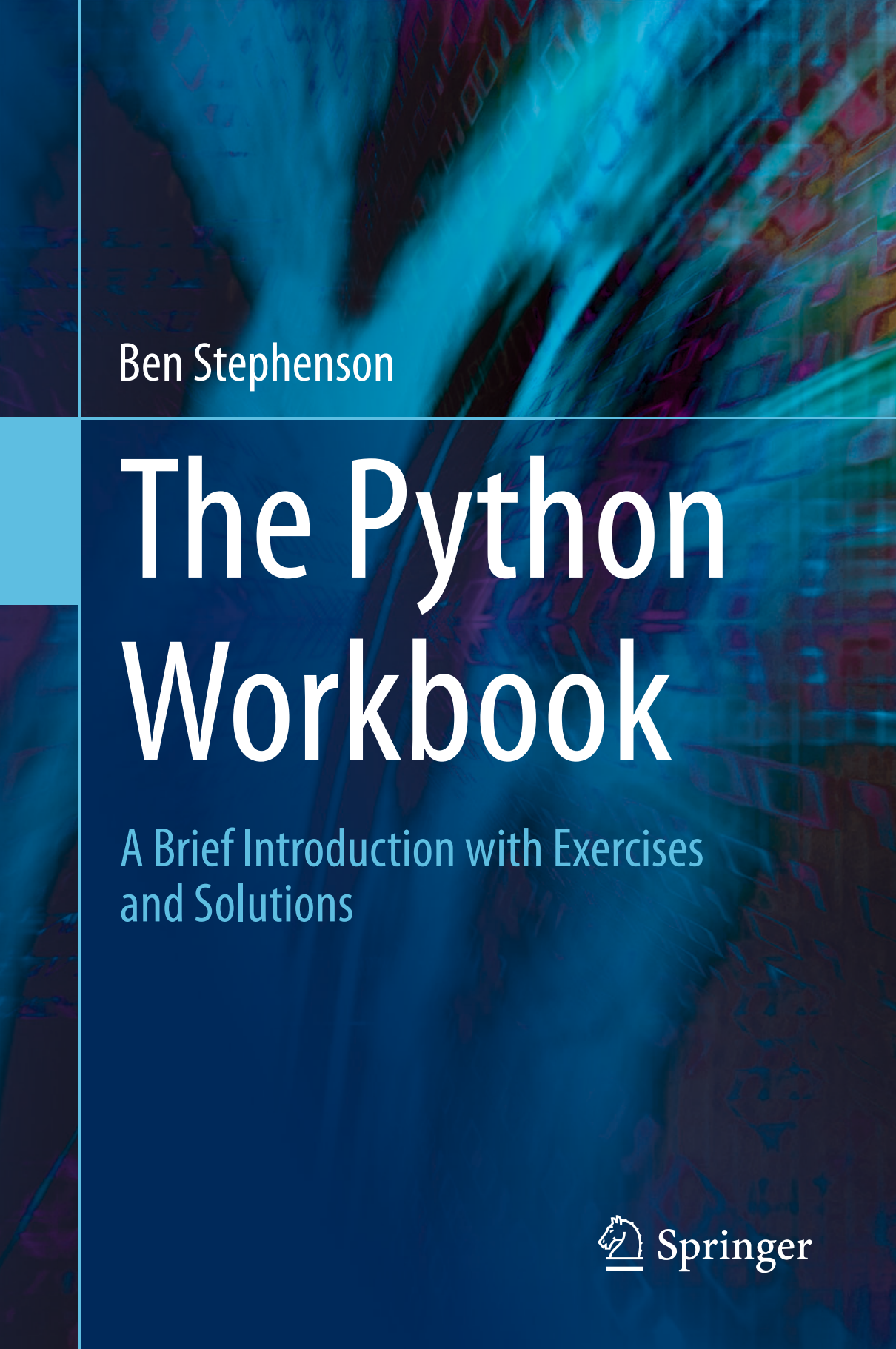

Ben Stephenson
The Python Workbook
A Brief Introduction with Exercises and Solutions
Springer

The Python Workbook

Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises and Solutions

Springer

Ben Stephenson
University of Calgary
Calgary, AB
Canada

ISBN 978-3-319-14239-5 ISBN 978-3-319-14240-1 (eBook)
DOI 10.1007/978-3-319-14240-1

Library of Congress Control Number: 2014957402

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

To my son, Jonathan, who surprised us all by arriving before this book was completed, and to my wife Flora, for 12 fantastic years of marriage, and many more to come.

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 the ideas that you have been introduced to previously into practice.

This book is designed to support and encourage hands-on learning about programming. It contains 174 exercises, spanning a variety of academic disciplines and everyday situations, which you can solve using only the material covered in most introductory Python programming courses. Each exercise that you complete will strengthen your understanding 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 keep you interested 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 form, including appropriate comments, meaningful variable names and minimal use of magic numbers. I encourage you to use good programming form so that your solutions compute the correct result while also being clear, easy to understand and easy to update 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 indicated 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. It can supplement another textbook that has a limited selection of exercises, or it can be used as the sole source of exercises when an instructor has decided not to use another textbook. A motivated individual could also learn Python programming by carefully studying each of the included exercises and solutions, though there are probably easier ways to learn the language. No matter what other resources you use with this book, completing the exercises and studying the provided solutions will enhance your programming ability.

Calgary, Canada, November 2014 Ben Stephenson

Contents

Part I
Exercises

Introduction to Programming Exercises

1

The exercises in this chapter are designed to help you develop your analysis skills by providing you with the opportunity to practice breaking small problems down into sequences of steps. In addition, completing these exercises will help you become familiar with Python's syntax. To complete each exercise you should expect to use some or all of these Python features:

- Generate output with `print` statements
- Read input, including casting that input to the appropriate type
- Perform calculations involving integers and floating point numbers using Python operators like `+`, `-`, `*`, `/`, `//`, `%`, and `**`
- Call functions residing in the `math` module
- Control how output is displayed using format specifiers

Exercise 1: Mailing Address

(*Solved—9 Lines*)

Create a program that displays your name and complete mailing address formatted in the manner that you would usually see it on the outside of an envelope. 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.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_1

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 the 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 always displays exactly two decimal places.

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—12 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 in an order from the user. Then your program should compute and display the total weight of the order.

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—20 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—33 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—25 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 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 $4.186 \frac{J}{g°C}$. Because water has a density of 1.0 gram per millilitre, you can use grams and millilitres 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 water 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—16 Lines*)

Create a program that determines how quickly an object is traveling 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 $9.8\,\mathrm{m/s^2}$. 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 $8.314\frac{\mathrm{J}}{\mathrm{mol\ K}}$, 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 the previous exercise. 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, with a leading 0 displayed if necessary.

Exercise 26: Current Time

(*10 Lines*)

Python includes a library of functions for working with time, including a function called `asctime` in the `time` module. It reads the current time from the computer's internal clock and returns it in a human-readable format. Write a program that displays the current time and date. Your program will not require any input from the user.

Exercise 27: 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 28: 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 with temperatures in degrees Fahrenheit and wind speeds in miles per hour.

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

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 29: 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 30: 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 31: 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 the digits in the number. For example, if the user enters 3141 then your program should display $3+1+4+1=9$.

Exercise 32: 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 33: 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. 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.

If Statement Exercises

2

The programming constructs that you used to solve the exercises in the previous chapter will continue to be useful as you tackle these problems. In addition, the exercises in this chapter will require you to use decision making constructs so that your programs can handle a variety of different situations that might arise. You should expect to use some or all of these Python features when completing these problems:

- Make a decision with an if statement
- Select one of two alternatives with an if-else statement
- Select from one of several alternatives by using an if-elif or if-elif-else statement
- Construct a complex condition for an if statement that includes the Boolean operators `and`, `or` and `not`
- Nest an if statement within the body of another if statement

Exercise 34: 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 35: 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.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_2

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 36: 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 37: 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 38: 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 39: Sound Levels

(*30 Lines*)

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

Noise	Decibel level (dB)
Jackhammer	130
Gas lawnmower	106
Alarm clock	70
Quiet room	40

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 level 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 40: Name that Triangle

(*Solved—20 Lines*)

A triangle can be classified based on the lengths of its sides as equilateral, isosceles or scalene. All 3 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 3 sides of a triangle from the user. Display a message indicating the type of the triangle.

Exercise 41: 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: To complete this exercise you will need to extract individual characters from the two-character note name so that you can work with the letter and the octave number separately. Once you have separated the parts, compute the frequency of the note in the fourth octave using the data in the table above. Then divide the frequency 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 42: Frequency To Note

(*Solved—40 Lines*)

In the previous question you converted from note name to 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 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 43: 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 Table 2.1.

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

Table 2.1 Individuals that appear on Banknotes

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

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 44: 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

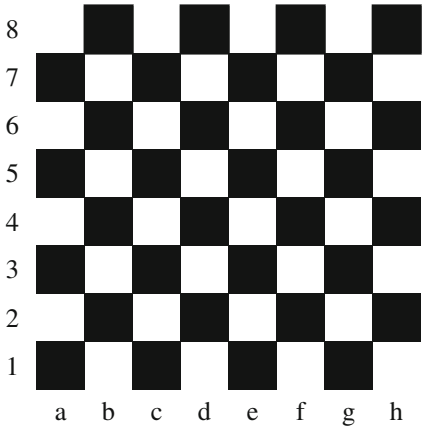
Write a program that reads a month and day from the user. If the month and day match one of the holidays listed previously then your program should display the holiday's name. Otherwise your program should indicate that the entered month and day do not correspond to a fixed-date holiday.

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 45: 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 46: Season from Month and Day

(*Solved—40 Lines*)

The year is divided into four seasons: spring, summer, fall 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 47: 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 48: Chinese Zodiac

(*Solved—35 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 49: 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 50: 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. The roots of a quadratic function can be found by finding the values of x that satisfy the quadratic equation $ax^2 + bx + c = 0$. A quadratic function may have 0, 1 or 2 real roots. These roots can be computed using the quadratic formula, shown below:

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

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 51: 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 52: Grade Points to Letter Grade

(*47 Lines*)

In the previous exercise you created a program that converts 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+ for a 4.0 (or greater) grade point average.

Exercise 53: Assessing Employees

(*Solved—28 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 $2400.00 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 was 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 54: 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 55: 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 the radiation from the user and displays the appropriate name.

Exercise 56: 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 57: 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 58: 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 2013-11-18 then your program should display a message indicating that the day immediately after 2013-11-18 is 2013-11-19. If the user enters values that represent 2013-11-30 then the program should indicate that the next day is 2013-12-01. If the user enters values that represent 2013-12-31 then the program should indicate that the next day is 2014-01-01. 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 59: Is a License Plate Valid?

(*Solved—28 Lines*)

In a particular jurisdiction, older license plates consist of three uppercase letters followed by three numbers. When all of the license plates following that pattern had

been used, the format was changed to four numbers 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 60: 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.

Loop Exercises

3

The exercises that appear in this chapter should all be completed using 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 61: 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 62: 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

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_3

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 63: 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 64: No More Pennies

(*Solved—38 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 65: 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 values for the first point on the perimeter of the polygon from the user. Then continue reading pairs of x and y 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 it should display the total perimeter. Sample input and output is shown below, with user input shown in bold:

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

Exercise 66: Compute a Grade Point Average

(*62 Lines*)

Exercise 51 included 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 51 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 67: 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 68: Parity Bits

(*Solved—25 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 69: 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 70: 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 result, he developed what later became known as the Caesar Cipher.

The idea behind this cipher is simple (and as a result, 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 71: Square Root

(*14 Lines*)

Write a program that implements Newton's method to compute and display the square root of a number 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. 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 72: Is a String a Palindrome?

(*Solved—23 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 determines whether or not it is a palindrome. Display the result, including a meaningful output message.

Exercise 73: 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 72 so that it ignores spacing while determining whether or not a string is a palindrome. For an additional challenge, extend your solution so that is also ignores punctuation marks and treats uppercase and lowercase letters as equivalent.

Exercise 74: 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 parameter 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 75: 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 76: Prime Factors

(*27 Lines*)

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

Initialize *factor* to two
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 integer division
    Else
        Increase *factor* by one

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 77: 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 78: Decimal to Binary

(*Solved—26 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 79: 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 handle 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 80: 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 shown on the same line. Then display the number of flips needed to reach 3 consecutive flips with 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.
```

Function Exercises

4

Functions allow a programmer to break a problem into pieces that can be reused. They can also help a programmer focus on one part a larger problem at a time. As a result, writing functions is often an important part of developing larger pieces of software. The exercises in this chapter will help you practice these skills:

- Define a function for later use
- Pass one or more values into a function
- Perform a complex calculation within a function
- Return one or more results from a function
- Call a function that you have defined previously

Exercise 81: 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 82: 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 traveled. Write a function that takes the distance traveled (in kilometers) as its only parameter and returns the total fare as its only result. Write a main program that demonstrates the function.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_4

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 83: 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, and $2.95 for each subsequent item. 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 84: Median of Three Values

(*Solved—42 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 85: 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 a value outside of this range is provided as a parameter. 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 86: The Twelve Days of Christmas

(*Solved—48 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.

Your task is to write a program that displays the complete lyrics for The Twelve Days of Christmas. Write a function that takes the verse number as its only parameter and displays the specified verse of the song. Then call that 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 once in your program, 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 85 to help you complete this exercise.

Exercise 87: Center a String in the Terminal

(*Solved—31 Lines*)

Write a function that takes a string of characters as its first parameter, and the width of the terminal in characters as its second parameter. Your function should return a new string that consists of the original string and the correct number of leading spaces so that the original string will appear centered within the provided width when it is printed. Do not add any characters to the end of the string. Include a main program that demonstrates your function.

Exercise 88: 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. In general, 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. In addition, write a program that reads 3 lengths from the user and demonstrates the behaviour of this function.

Exercise 89: Capitalize It

(*Solved—48 Lines*)

Many people do not use capital letters correctly, especially when typing on small devices like smart phones. In this exercise, you will write a function that capitalizes the appropriate characters in a string. A lowercase "i" should be replaced with an uppercase "I" if it is both preceded and followed by a space. The first character in the string should also be capitalized, as well as the first non-space character after a ".", "!" or "?". For example, if the function is provided with the string "what time do i have to be there? what's the address?" then it should return the string "What time do I have to be there? What's the address?". Include a main program that reads a string from the user, capitalizes it using your function, and displays the result.

Exercise 90: 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 1 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 91: 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 92: Is a Number Prime?

(*Solved—28 Lines*)

A prime number is an integer greater than 1 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 93: 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 92 while completing this exercise.

Exercise 94: 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 95: Random License Plate

(*45 Lines*)

In a particular jurisdiction, older license plates consist of three letters followed by three numbers. When all of the license plates following that pattern had been used, the format was changed to four numbers 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 96: 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 97: Random Good Password

(*22 Lines*)

Using your solutions to Exercises 94 and 96, 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 98: 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 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 the function's only result. Ensure that the `hex2int` function works correctly for both uppercase and lowercase letters. Your functions should end the program with a meaningful error message if an invalid parameter is provided.

Exercise 99: Arbitrary Base Conversions

(*Solved—61 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 77, 78 and 98 helpful when completing this exercise.

Exercise 100: 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 57 helpful when solving this problem.

Exercise 101: Reduce a Fraction to Lowest Terms

(*Solved—47 Lines*)

Write a function that takes two positive integers that represent the numerator and denominator of a fraction as its only two 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 75 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 102: Reduce Measures

(*Solved—83 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. Return a string representing the measure using the largest possible units as the function's 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 103: 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 100 helpful when completing this exercise.

List Exercises

5

Lists help programmers manage larger amounts of data by allowing several (or even many) values to be stored in one variable. This makes it practical to solve larger problems that involve many data values. To solve the exercises in this chapter you should expect to:

- Create a variable that holds a list of values
- Modify a list by appending, inserting, updating and deleting elements
- Search a list for a value
- Display some or all of the values in a list
- Write a function that takes a list as a parameter
- Write a function that returns a list as its result

Exercise 104: Sorted Order

(*Solved—21 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 order from smallest to largest, with one value appearing on each line. Use either the `sort` method or the `sorted` function to sort the list.

Exercise 105: 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.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_5

Exercise 106: Remove Outliers

(*Solved—43 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. Your function should read a list of numbers from the user and remove the two largest and two smallest values from it. 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 107: 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 entered. For example, if the user enters:

```
first
second
first
third
second
```

then your program should display:

```
first
second
third
```

Exercise 108: Negatives, Zeros and Positives

(*Solved—38 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 109: 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 110: 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 109 when completing this task.

Exercise 111: 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 of text 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 remove include commas, periods, question marks, hyphens, apostrophes, exclamation points, colons, and semicolons. Do not remove punctuation marks that appear in the middle of a words, such as the apostrophes used to form a contraction. For example, if your function is provided with the string `"Examples of contractions include: don't, isn't, and wouldn't."` then your function should return the list `["Examples", "of", "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 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 Exercise 158. As a result, you should ensure that your main program only runs when your file has not been imported into another program.

Exercise 112: 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 113: Formatting a List

(*Solved—43 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 114: 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 115: 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 116: Pig Latin Improved

(*51 Lines*)

Extend your solution to Exercise 115 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 117: 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 x part of the first coordinate on its own line, followed by the y part of the first coordinate on its own line. Allow the user to continue entering coordinates, with the x and y parts 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 you calculated using 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 118: Shuffling a Deck of Cards

(*Solved—48 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 take 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.

Exercise 119: 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 118 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 120: 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 121: Count the Elements

(*Solved—49 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` which determines and returns 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 122: Tokenizing a String

(*Solved—64 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. Some tokens, such as `*`, `/`, `^`, `(` and `)` are easy to identify because the token is a single character, and the character is never part of another token. The + and – symbols are a little bit more challenging to handle because they might represent the addition or subtraction operator, or they might be part of a number token.

> Hint: A + or – is an operator if the non-whitespace character immediately before it is part of a number, or if the non-whitespace character immediately before it is a close parenthesis. Otherwise it is part of a number.

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 with an optional leading + or – (for simplicity we will only work with integers in this problem). Return the list of tokens as the function's 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 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 123: Infix to Postfix

(*62 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 both 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**
    Add the token to the end of *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 add it to *postfix*
    Add token to the end of *operators*
  If the token is an open parenthesis **then**
    Add token to the end of *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 add it to *postfix*
    Remove the open parenthesis from *operators*

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

Return *postfix* as the result of the algorithm

Use your solution to Exercise 122 to tokenize a mathematical expression. 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 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 124. You may find your solutions to Exercises 90 and 91 helpful when completing this problem.

> The algorithms provided in Exercises 123 and 124 do not perform any error checking. As a result, you may crash your program or receive incorrect results if you provide them with invalid input. These algorithms 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 124: Evaluate Postfix

(*58 Lines*)

Evaluating a postfix expression is easier than evaluating an infix expression because it does not contain any brackets 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 add it to the end of *values*
  Else
    Remove an item from the end of *values* and call it *right*
    Remove an item from the end of *values* and call it *left*
    Apply the operator to *left* and *right*
    Append the result to the end of *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, evaluates it, and displays its value. Uses your solutions to Exercises 122 and 123 along with the algorithm shown above to solve this problem.

Exercise 125: Does a List contain a Sublist?

(*44 Lines*)

A sublist is a list that makes up 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 126: Generate All Sublists of a List

(*Solved—40 Lines*)

Using the definition of a sublist from Exercise 125, 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 127: 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 *n*th 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.

Dictionary Exercises

6

Dictionaries are another data structure that Python programmers can use to manage larger amounts of data. 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 constructs that you have been introduced to in the previous chapters. Completing the exercises in this chapter will help you learn to:

- Create a new variable that holds a dictionary
- Add a key-value pair to a dictionary
- Update the value associated with a key in a dictionary
- Iterate over all of the keys and/or values in a dictionary
- Write functions that take dictionaries as parameters

Exercise 128: Reverse Lookup

(*Solved—40 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.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_6

Exercise 129: Two Dice Simulation

(*Solved—42 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 total number of rolls. 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 130: 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 letter on the key. Pressing the number 2, 3, 4 or 5 times generates the second, third, fourth or fifth character listed for that key.

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 that must be made to enter a text message read from the user. Construct a dictionary that maps from each letter or symbol to the key presses. Then use the dictionary to generate and display the presses 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 brackets.

Exercise 131: Morse Code

(*15 Lines*)

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

Your program should read a message from the user. Then it should translate each letter and number 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 letters or numbers. The Morse code for `Hello, World!` is shown below:

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

Table 6.1 Morse Code Letters and Numbers

Letter	Code	Letter	Code	Letter	Code	Number	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, over 160 years after it was first created.

Exercise 132: Postal Codes

(*24 Lines*)

In a Canadian postal code, the first, third and fifth characters are letters while the second, fourth and sixth characters are numbers. The province can be determined from the first character of a 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	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 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 name. Display a meaningful error message if the postal code begins with an invalid character.

Exercise 133: 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 134: Unique Characters

(*Solved—14 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 135: 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 136: 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 135 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 137: 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:

One point	A, E, I, L, N, O, R, S, T and U
Two points	D and G
Three points	B, C, M and P
Four points	F, H, V, W and Y
Five points	K
Eight points	J and X
Ten points	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 138: Create a Bingo Card

(*Solved—58 Lines*)

A Bingo card consists of 5 columns of 5 numbers. The columns are labeled 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 labeled 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 139: Checking for a Winning Card

(*102 Lines*)

A winning Bingo card contains a line of 5 numbers that have all been called. Players normally mark the numbers that have been called by crossing them out or marking them with a Bingo dauber. In our implementation 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 138 when completing this exercise.

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

Exercise 140: 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, crossing out numbers on the card, until the card contains a crossed out line (horizontal, vertical or diagonal). Simulate 1,000 games and report the minimum, maximum and average number of calls that must be made before the card wins. Import your solutions to Exercises 138 and 139 when completing this exercise.

File and Exception Exercises

7

Files allow us to work with data, without needing to enter it each time our program runs. Files also allow us to store results from our program in a more permanent manner. These features are often used when creating larger programs. When completing the exercises in this chapter, you should expect to:

- Open a file for reading and/or writing
- Read data from a file
- Write data to a new file
- Use values provided to the program as command line parameters
- Detect and recover from errors such as attempting to open a file that doesn't exist
- Detect and recover from other errors that are not specifically related to files

Some of the exercises in this chapter involve reading from existing files such as a list of words, names or chemical elements. You can download these files from the author's website: http://www.cpsc.ucalgary.ca/~bdstephe/PythonWorkbook.

Exercise 141: 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 parameter. Write a Python program that provides the same behavior. Display an appropriate error message if the file requested by the user does not exist or if the command line parameter is omitted.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_7

Exercise 142: 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 parameter. Write a Python program that provides the same behavior. Display an appropriate error message if the file requested by the user does not exist or if the command line parameter 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 the 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 the 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 143: Concatenate Multiple Files

(*Solved—27 Lines*)

Unix-based operating systems typically include a tool named `cat`, which is short for concatenate. Its purpose is to concatenate and display one or more files whose names are provided as command line parameters. 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 parameters.

Exercise 144: Number the Lines in a File

(*23 Lines*)

Create a program that adds line numbers to a 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 145: 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 numbers or punctuation marks.

Exercise 146: 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 letters in a file. Ignore spaces, punctuation marks, and numbers as you perform this analysis. Your program should be case insensitive, treating `a` and `A` as equivalent. The user will provide the file name as a command line parameter. Your program should display a meaningful error message if the user provides the wrong number of command line parameters, or if the program is unable to open the file indicated by the user.

Exercise 147: 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 find the word(s) by splitting each line in the file at each space. Finally, any leading or trailing punctuation marks should be removed from each word. In addition, your program should ignore capitalization. As a result, `apple`, `apple!`, `Apple` and `ApPlE` should all be treated as the same word. You will probably find your solution to Exercise 111 helpful when completing this problem.

Exercise 148: Sum a List of Numbers

(*Solved—26 Lines*)

Create a program that sums all of the numbers entered by the user while ignoring any lines entered by the user that are not valid numbers. Your program should display the current sum after each number is entered. It should display an appropriate error message after any invalid input, and then continue to sum any additional numbers entered by the user. Your program should exit 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 149: Both Letter Grades and Grade Points

(*106 Lines*)

Write a program that converts from letter grades to grade points and vice-versa. Your program will convert multiple values entered by the user, 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 the attempt 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 provide a message indicating that the supplied input is invalid. Design your program so that it continues performing conversions until the user enters a blank line. Your solutions to Exercises 51 and 52 may be helpful when completing this exercise.

Exercise 150: Remove Comments

(*Solved—46 Lines*)

Python uses the # character to mark the beginning of a comment. The comment ends at the end of the line containing the # character. 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'll ignore the situation where the comment character occurs inside of a string). Save the modified file using a new name that will be entered by the user. The user will also enter the name of the input file. Ensure that an appropriate error message is displayed if a problem is encountered while accessing the files.

Exercise 151: Two Word Random Password

(*Solved—37 Lines*)

While generating a password by selecting random characters generally gives a relatively secure password, 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 isn't as secure, it is 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. Display the password for the user.

Exercise 152: 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 string 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 153: A Book with No "e" ...

(*Solved—49 Lines*)

The novel "Gadsby" is over 50,000 words in length. While 50,000 words isn't 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 the result for all 26 letters. Include an additional message identifying the letter that is used in the smallest proportion of the words. Your program should ignore any punctuation marks and it should treat uppercase and lowercase letters as equivalent.

Exercise 154: Names that Reached Number One

(*Solved—50 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. 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 155: Gender Neutral Names

(*56 Lines*)

Some names, like Ben and Jonathan, are normally only used for boys while names like Rebbecca 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 154.

Exercise 156: Most Births in a given Time Period

(*76 Lines*)

Write a program that uses the baby names data set described in Exercise 154 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 157: 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 154. As your program reads the files, it should keep track of each name used for a boy and each name used for a girl. Your program should

output two lists. One list will contain all of the names that have been used for girls. The other list will contain all of the names that have been used for boys. Neither of your lists should contain any repeated values.

Exercise 158: Spell Checker

(*Solved—51 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 parameter. Your program should display an appropriate error message if the command line parameter 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 111 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 159: 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 parameter. Display an appropriate error message if the user fails to provide a command line parameter, or if an error occurs while processing the file.

Exercise 160: Redacting Text in a File

(*Solved—49 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 either be found in your textbook or 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 161: Missing Comments

(*Solved—44 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 parameters. If your program encounters a file that doesn't exist or can't be opened then it should display an appropriate error message before moving on and processing the remaining files.

Exercise 162: 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 use of the available space.

Write a program that opens a file and displays it so that each line is filled as full as possible. If you read a line that is too long then your program should break it up into words and add words 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. You may perform error checking if you want to, but it is not required for this exercise.

Hint: Use a constant to represent the maximum line length. This will make it easier to update the program when the window size changes.

Exercise 163: 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 searching for words with six vowels in order.

Recursion Exercises

8

A recursive function is a function that calls itself. In this chapter, you will use recursive functions to solve a variety of problems. The programs that you write will help you learn to:

- Identify the base case(s) for a recursive function
- Identify the recursive case(s) for a recursive function
- Write a non-trivial recursive function
- Use a recursive function that you have written to solve a problem

Exercise 164: Total the Values

(*Solved—28 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 parameters, but it will need to return a numeric result.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_8

Exercise 165: 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.

Exercise 166: Recursive Decimal to Binary

(*34 Lines*)

In Exercise 78 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 167: Recursive Palindrome

(*Solved—29 Lines*)

The notion of a palindrome was introduced previously in Exercise 72. 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 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. Use your recursive function to determine whether or not the string is a palindrome. Then display an appropriate message for the user.

Exercise 168: Recursive Square Root

(*20 Lines*)

Exercise 71 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 iteration of a loop. In this exercise you will use the same approximation strategy, but you will use recursion instead of iteration.

Create a square root function that takes 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 for *guess* is used.

Exercise 169: String Edit Distance

(*Solved—42 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 170: 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 can not be formed using 4, 6 or 7 coins.

Your program should read both the dollar amount and the number of coins from the user. 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 can not contain any loops.

Exercise 171: Spelling with Element Symbols

(*68 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 can not 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 take two parameters: the word that you are trying to spell and a list of the symbols that can be used. Your function will return two results: a Boolean value indicating whether or not a spelling was found, and the string of symbols used to achieve the spelling (or an empty string if no spelling exists). Your function should ignore capitalization when searching 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 172: Element Sequences

(*Solved—83 Lines*)

Another game that some people play with the names of chemical elements involves constructing a sequence of 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 can not contain any duplicates.

Write a program that reads the name of an element from the user. Your program should use a recursive function to find the longest sequence of elements that begins with the entered element. Then it should display the sequence. 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 173: 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 the value 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 parameter. 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 174: Run-Length Encoding

(*Solved—36 Lines*)

Write a recursive function that implements the run-length compression technique described in Exercise 173. Your function will take a list or a string as its only parameter. 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

Introduction to Programming Solutions

9

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 input values 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.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_9

Solution to Exercise 4: Area of a Field

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

# Read input 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 input from the user
less = int(input("How many containers 1 litre or less do you have? "))
more = int(input("How many containers more than 1 litre do you have? "))

# 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

# 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))
```

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

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 input from the user
n = int(input("Enter a positive integer: "))

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

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

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

Solution to Exercise 10: Arithmetic

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

# Read 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)

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

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

The remainder is computed using the `%` operator.

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

# 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")
```

Integer division, which discards any fractional part of the result, is performed using the // operator.

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 input 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 from the user
volume = float(input("Enter the amount of water in milliliters: "))
d_temp = float(input("Enter the temperature increase (degrees Celsius): "))

# 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)
```

> 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.

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 from which the object is dropped (in meters): "))
```

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

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

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

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 of the polygon: "))
n = int(input("Enter the 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)
```

We have chosen to cast input `n` to an integer because a polygon cannot have a fractional number of sides.

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 input from the user
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 the integer using two digits, adding a leading 0 if necessary.

Solution to Exercise 28: 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("Enter the air temperature (degrees Celsius): "))
speed = float(input("Enter the 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 32: 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 shouldn't use those names for variables. Instead we use variables named `mn` and `mx` to hold the minimum and maximum values respectively.

Solution to Exercise 33: 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 discount and the total are different.

If Statement Solutions

10

Solution to Exercise 34: 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 36: Vowel or Consonant

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

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

# 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*, DOI 10.1007/978-3-319-14240-1_10

Solution to Exercise 37: 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"

# 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)
```

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.

Solution to Exercise 38: Month Name to Number of Days

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

# Read input 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"

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

When `month` is February, the value assigned to `days` is a string so that we can represent 28 or 29 days.

Solution to Exercise 40: Name that Triangle

```
##
# Determine the name of a triangle from 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 name
if side1 == side2 and side2 == side3:
  name = "equilateral"
elif side1 == side2 or side2 == side3 or \
     side3 == side1:
  name = "isoceles"
else:
  name = "scalene"

# Display the triangle's name
print("That's a", name, "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 41: 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 frequncy of", name, "is", freq)
```

Solution to Exercise 42: 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: "))

# 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 46: 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: "))

# 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"
```

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

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

Solution to Exercise 48: 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 51: 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 of the `if` and `elif` statements.

```
# 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

# Display the output
if gp == INVALID:
  print("That wasn't a valid number of grade points.")
else:
  print("That's", gp, "grade points.")
```

Solution to Exercise 53: 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, the 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 the result.

Solution to Exercise 57: 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 59: 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 60: 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, 9
# The second line in the condition checks for 12, 14, 16, 18
# The third line in the condition checks for 19, 21, 23, 25, 27
# The fourth line in the condition checks for 30, 32, 34, 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 number vs. upper number 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. The `pass` statement can be used in situations where a statement is required but there is no work to be performed.

Loop Solutions

11

Solution to Exercise 64: 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 of all 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 float)
  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
# only using nickels
rounding_indicator = total * 100 % PENNIES_PER_NICKEL
```


B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_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 the number of pennies
  cash_total = total + NICKEL - rounding_indicator / 100

# Display the cash amount payable
print("The cash amount payable is %.02f" % cash_total)
```

Solution to Exercise 65: Computer the Perimeter of a Polygon

```
##
# Compute the perimeter of a polygon. The user will enter a blank line
# 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 x part of the coordiante: "))
first_y = float(input("Enter the y part of the 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 x part of the coordinate (blank to quit): ")
while line != "":
  # Convert the x part to a number and read the y part
  x = float(line)
  y = float(input("Enter the y part of the 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 x part of the next coordinate
  line = input("Enter the x part of the 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
```

The distance between the points is computed using Pythagorean theorem.

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

Solution to Exercise 67: 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 line 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)
```

The first condition is not necessary with the current admission prices. However, including it makes it easy to start charging for babbies in the future.

Solution to Exercise 68: Parity Bits

```
##
#  Compute 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")

    # 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: ")
```

The `count` method returns the number of times that the string provided as a parameter occurs in the string on which the method is invoked.

Solution to Exercise 70: 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, constructing a new 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
```

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

```
  else:
    # If the character is not a letter then copy it into the new message
    new_message = new_message + ch

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

Solution to Exercise 72: Is a String a Palindrome?

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

# Read the input 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 until
# the middle is reached
for i in range(0, len(line) // 2):
  # If the characters don't match then mark
  # that the string is not a palindrome
  if line[i] != line[len(line) - i - 1]:
    is_palindrome = False

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

All of the parameters to the range function must be integers. The // operator is used so that the result of the division is truncated to an integer.

Solution to Exercise 74: 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 parameter 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 75: 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 78: 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

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

The algorithm provided for this question is expressed using a repeat-until loop. However, this structure 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 body and placing it ahead of the `while` loop.

```
# Keep on looping until q == 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 79: Maximum Integer

```
##
#  Find the maximum of 100 random integers, counting 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 of the number of updates
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 summary results
print("The maximum value found was", mx_value)
print("The maximum value was updated", num_updates, "times")
```

Function Solutions

12

Solution to Exercise 84: 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

## 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)
```

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`.

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

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_12

```
# 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: "))

  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 86: The Twelve days of Christmas

```
##
# Generate 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.

```
## Generate and display one verse of The Twelve Days of Christmas
# @param n the verse to generate
# @return (none)
def displayVerse(n):
  print("One 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 87: Center a String in the Terminal

```
##
# 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 so that
#       s will appear centered when it is printed.
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

  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()
```

The // operator is used so that the result of the division will be truncated to an integer. This is necessary because a string can only be replicated an integer number of times.

Solution to Exercise 89: Capitalize it

```
##
# Improve the capitalization of a string by replacing " i " with " I " and by
# capitalizing the first letter of each sentence.
#

## 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):
  # Correct the capitalization for i
  result = s.replace(" i ", " I ")

  # Capitalize the first character of the string
  if len(s) > 0:
    result = result[0].upper() + \
             result[1 : len(result)]

  # 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

  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()
```

The `replace` method replaces all occurences of its first parameter with its second parameter in the string on which it is invoked.

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.

Solution to Exercise 90: 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

# 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 `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.

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 92: 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

# 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()
```

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

Solution to Exercise 94: 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

  # 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()
```

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

Solution to Exercise 96: 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
# long 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 on 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 99: 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 98. Using `import *` imports all of the functions from that module.

```
## Convert a number from base 10 to base n
#  @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 == 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
  power = 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
```

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

```
# Convert a number between two arbitrary bases
def main():
  # Read the number from the user
  from_base = int(input("Enter the base to convert from: "))
  from_num = input("Enter a 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 a new base and display the result
to_base = int(input("Enter the base to convert to: "))
to_num = dec2n(dec, to_base)
print("That's %s in base %d." % (to_num, to_base))
```

Solution to Exercise 101: Reduce a Fraction to Lowest Terms

```
##
# Reduce a fraction to its 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 used 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 a later exercise.

```
## Reduce a fraction to lowest terms.
# @param the integer numerator of the fraction
# @param 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)

# Read the fraction from the user and display the reduced fraction
def main():
  # Read the input 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()
```

We have used the integer division operator, `//`, so that we return a result like `(1, 3)` instead of `(1.0, 3.0)`.

Solution to Exercise 102: 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):
  # Compute the number of teaspoons that the parameters represent
  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.

```
  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

# Generate the result string
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 occurences 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 103: 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

# 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()
```

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

List Solutions

13

Solution to Exercise 104: Sorted Order

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

# Start with an empty list
data = []

# Read values, adding 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 keep 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)
```


B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_13

Solution to Exercise 106: Remove Outliers

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

## Remove the outliers from a list of data
# @param data the list of data 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

# 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()
```

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.

Solution to Exercise 107: 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.
#

# Begin reading words into 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): ")

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

The expression `word not in words` is equivalent to `not (word in words)`.

Solution to Exercise 108: 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 = []

# 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): ")
```

This solution uses a list to keep track of the zeros that are entered. However, because all of the zeros are the same, it isn't 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.

```
# 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 110: 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

  # 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()
```

The total could also be computed using the `sum` function. This would reduce the calculation of the total to a single line.

Solution to Exercise 113: 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 cast to a string by calling the `str` function before it is added to the result. This allows `formatList` 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 114: 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

# Keep a list of the numbers for 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 distinct 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()
```

Using constants makes it easy to recongifure our program for other lotteries.

Solution to Exercise 118: 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 store the cards in
  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, modifying the deck of cards passed as a parameter
# @param cards the list of cards to shuffle
def shuffle(cards):
  # For each card
  for i in range(0, len(cards)):
    # Pick a random index
    other_pos = randrange(0, 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 program only if this file has not been imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 121: 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 to process
# @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] that are 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] that are 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] that are 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 [] that are 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 [1, 2, 3, 4, 1, 2, 3, 4] that are", \
        "between 2 and 4...")
  print("Result: %d  Expected: 4" % countRange(data, 2, 4))

# Call the main program
main()
```

Solution to Exercise 122: 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] == ")":
      tokens.append(s[i])
      i = i + 1

    # Handle + and -
    elif s[i] == "+" or s[i] == "-":
      # If there is a previous character and it is a number or close bracket
      # then the + or - is a token on its own
      if i > 0 and (s[i-1] >= "0" and s[i-1] <= "9" or s[i-1] == ")"):
        tokens.append(s[i])
        i = i + 1
      else:
        # The + or - is part of a number
        num = s[i]
        i = i + 1

        # 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)

    # 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 ocurred.
    else:
      return []
  return tokens

# Read an expression from the user and tokenize it, displaying the result
def main():
  exp = input("Enter a mathematical expression: ")
  tokens = tokenList(exp)
  print("The tokens are:", tokens)

# Call the main function only if the file hasn't been imported
if __name__ == "__main__":
  main()
```

Solution to Exercise 126: Generate All Sublists of a List

```
##
# Compute all 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 127: The Sieve of Eratosthenes

```
##
# Determine 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("Generate all primes up to what limit? "))
```

```
# Generate all of the numbers 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)
```

Dictionary Solutions

14

Solution to Exercise 128: 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 to perform the reverse lookup on
# @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, adding 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

# 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()
  print("The french word for 'apple' is: ", reverseLookup(frEn, "apple"))
  print("Expected: ['pomme']")
  print()
```

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.

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_14

```
  print("The french word for 'asdf' is: ", reverseLookup(frEn, "asdf"))
  print("Expected: []")

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

Solution to Exercise 129: 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 of 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}

  # 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))
```

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.

```
# Call the main function
main()
```

Solution to Exercise 134: 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 135: 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.")

# Call the main function
main()
```

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.

Solution to Exercise 137: 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]

# Display the result
print(word, "is worth", score, "points.")
```

The word is converted to uppercase so that the user will get a correct response when they enter the word in upper, mixed or lowercase. This could also be accomplished by adding all of the lowercase letters to the dictionary.

Solution to Exercise 138: 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 generated 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()
```

```
# Generate a random Bingo card and display it
def main():
  card = createCard()
  displayCard(card)

# Call the main program only if this file hasn't been imported
if __name__ == "__main__":
  main()
```

File and Exception Solutions

15

Solution to Exercise 141: Display the Head of a File

```
##
# Display the head (first 10 lines) of a file whose name is provided as a command line parameter.
#
import sys

NUM_LINES = 10

# Verify that exactly one command line parameter (in addition to the .py file) was supplied
if len(sys.argv) != 2:
  print("You must provide the file name as a command line parameter.")
  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*, DOI 10.1007/978-3-319-14240-1_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 142: Display the Tail of a File

```
##
# Display the tail (last lines) of a file whose name is provided as a command line parameter.
#
import sys

NUM_LINES = 10

# Verify that exactly one command line parameter (in addition to the .py file) was provided
if len(sys.argv) != 2:
  print("The file name must be provided as a command line parameter.")
  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 143: Concatenate Multiple Files

```
##
# Concatenate one or more files, displaying the result.
#
import sys
```

```
# Ensure that at least one command line parameter (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 to process 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 files at position 1 in the list.

Solution to Exercise 148: Sum a List 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 150: Remove Comments

```
##
#  Remove all of the comments from a python file, ignoring the case where
#  a comment character occurs within a string.
#

# Read and open the input file, ensuring that it was opened successfully
try:
  in_name = input("Enter the name of a Python file: ")
  inf = open(in_name, "r")
except:
  print("A problem was encountered while opening the input file.")
  print("Quitting...")
  quit()

# Read and open the output file, ensuring that it was opened successfully
try:
  out_name = input("Enter the output file name: ")
  outf = open(out_name, "w")
except:
  print("A problem was encountered while opening the output file.")
  print("Quitting...")
  quit()

try:
  # Read all of the lines from the input file, process them to remove
  # comments, and save the 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 form a slice of the
    # string that excludes it, overwriting 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:
  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 151: Two Word Random Password

```
##
#  Generate a password by concatenating two random words.  The password will be
#  between 8 and 10 characters, and each word will be at least three letters long.
#
from random import randrange

WORD_FILE = "../Data/words.txt"

# Read all of the words from the file, only keeping those that are
# between 3 and 7 characters 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 generated password
print("The random password is:", password)
```

The password we are creating will be between 8 and 10 characters in length. Since the shortest acceptable word is 3 letters long, and a password must have 2 words in it, a password can never contain a word longer than 7 letters.

Solution to Exercise 153: A Book with No "e" ...

```
##
#  Determine the proportion of words that include each letter of the alphabet.
#  The letter that is used in the smallest proportion of words is highlighted.
#
WORD_FILE = "../Data/words.txt"

# Maintain a dictionary that counts the number of words containing each letter.
# Initialize the count for each letter to 0.
contains = {}
for ch in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
  contains[ch] = 0
```

```
# Open the file, and process each word, updating the contains 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 remove any hyphens that might be present.
  unique = []
  for ch in word:
    if ch not in unique and ch != "-":
      unique.append(ch)

  # Now increment the counts for all of the letters that are in the list of unique characters
  for ch in unique:
    contains[ch] = contains[ch] + 1

  # Keep count 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 later.
smallest_count = min(contains.values())
for ch in sorted(contains):
  if contains[ch] == smallest_count:
    smallest_letter = ch
  percentage = contains[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 154: 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 the name of the file to read the data from
#  @param 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 158: Spell Checker

```
##
#  Find and list all of the words in a file that are misspelled.
#
from only_words import onlyTheWords
from sys import argv


WORDS_FILE = "../Data/words.txt"

# Ensure that the program has the correct number of command line parameters
if len(argv) != 2:
  print("One command line parameter must be provided.  Quiting...")
  quit()

# Open the file.  Quit if the file is not opened successfully.
try:
  inf = open(argv[1], "r")
except:
  print("Failed to open '%s' for reading.  Quiting..." % 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:
  word = word.lower().rstrip()
  valid[word] = 0
words_file.close()

# 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 111
  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()
```

This solution uses a dictionary, but the values in the dictionary are never used. As a result, a set would be a better choice if you have learned about 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.

```
# 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 160: 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()

sen.close()

# 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.  Use a number of asterisks that
  # matches the number of letters in the word
  for word in words:
    line = line.replace(word, "*" * len(word))
```

The sensitive word file can be closed now because all of the words have been read into a list.

```
  # 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 161: 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 parameter
if len(argv) == 1:
  print("At least one filename must be provided as a", \
        "command line parameter.")
  print("Quiting...")
  quit()

# Process each file provided as a command line parameter
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...")
```

Recursion Solutions

16

Solution to Exercise 164: Total the Values

```
##
#  Compute the total of a collection of numbers entered by the user.  The user
#  will enter a blank line to indicate that no further numbers will be entered.
#

# Compute the total of all 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 next line
    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 result
  print("The total of all those values is", total)

# Call the main function
main()
```

Solution to Exercise 167: Recursive Palindrome

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


B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1_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

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

  # Check the 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 169: String Edit Distance

```
##
# Compute and display the edit distance between two strings.
#

## Compute the edit distance between two strings as a count of 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
#  @param 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, 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 input from the user
  s1 = input("Enter a string: ")
  s2 = input("Enter another string: ")

  # Compute and display the result
  print("The edit distance between %s and %s is %d." % \
        (s1, s2, editDistance(s1, s2)))

# Call the main function
main()
```

Solution to Exercise 172: Element Sequences

```
##
# Determine 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.
#
from sys import *

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 the list of words that can occur in the sequence
# @return the longest list of words beginning with start that meets the
#       letter 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, preceeded 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()

  # Remove the starting element from the list of elements
  names.remove(start)

  # Find a longest sequence starting with start
  sequence = longestSequence(start, names)

  # Display the sequence of elements
  print("A longest sequence that starts with", start, "is:")
  for element in sequence:
    print(" ", element)

# Call the main function
main()
```

Solution to Exercise 174: Run-Length Encoding

```
##
#  Perform run-length encoding on a string of values using recursion.
#

## Perform run-length encoding on a string or list of values
#  @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 before 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:")
  print(encode(s))

# Call the main function
main()
```

Index

B. Stephenson, *The Python Workbook*, DOI 10.1007/978-3-319-14240-1