

Texts in Computer Science



Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises
and Solutions

Third Edition

 Springer

Texts in Computer Science

Series Editors

Orit Hazzan, Faculty of Education in Technology and Science, Technion—Israel Institute of Technology, Haifa, Israel

Frank Maurer, Department of Computer Science, University of Calgary, Calgary, Canada

Titles in this series now included in the Thomson Reuters Book Citation Index!

‘Texts in Computer Science’ (TCS) delivers high-quality instructional content for undergraduates and graduates in all areas of computing and information science, including core theoretical/foundational as well as advanced applied topics. TCS books should be reasonably self-contained and aim to provide students with modern and clear accounts of topics ranging across the computing curriculum. As a result, the books are ideal for semester courses or for individual self-study in cases where people need to expand their knowledge. All texts are authored by established experts in their fields, reviewed internally and by the series editors, and provide numerous examples, problems, and other pedagogical tools; many contain fully worked solutions.

The TCS series is comprised of high-quality, self-contained books that have broad and comprehensive coverage and are generally in hardback format and sometimes contain color. For undergraduate textbooks that are likely to be more brief and modular in their approach, Springer offers the flexibly designed *Undergraduate Topics in Computer Science* series, to which we refer potential authors.

Ben Stephenson

The Python Workbook

A Brief Introduction with Exercises and
Solutions

Third Edition

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

ISSN 1868-0941
Texts in Computer Science
ISBN 978-3-031-84559-8
<https://doi.org/10.1007/978-3-031-84560-4>

ISSN 1868-095X (electronic)
ISBN 978-3-031-84560-4 (eBook)

1st edition: © Springer International Publishing Switzerland 2014
2nd and 3rd editions: © Springer Nature Switzerland AG 2019, 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

If disposing of this product, please recycle the paper.

*To my wife, Flora, for more than 20 fantastic
years of marriage, and many more to come.
To my sons, Jonathan and Andrew, who were
both in a hurry to enter the world. I love you
all.*

Preface

I believe that computer programming is a skill that is best learned by doing. While it is valuable to read about programming, and to watch a teacher create a program at the front of a classroom, it is far more important to spend time at the keyboard solving problems and putting the concepts you have been introduced to into practice. This book was written with this principle in mind. The majority of its pages are dedicated to exercises and their solutions, with only a few pages at the beginning of each chapter that concisely introduce the necessary concepts.

There are 212 exercises in this book, which span a variety of academic disciplines and everyday situations. A few of the exercises are classical computer science problems, but most of them offer the opportunity to tackle problems from other fields of study and the world around you. Each exercise that you complete will strengthen your understanding of the Python programming language, improve your programming abilities, and allow you to tackle subsequent programming challenges more effectively.

Solutions to approximately half of the exercises are provided in the second half of this book. If you become stuck on an exercise, a quick peek at my solution may help you work through your problem without requiring assistance from someone else. Many of the solutions include brief annotations that explain the approach 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 solutions themselves.

I hope that you will take the time to compare 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 familiar with a technique that you could have used to solve the problem more easily. The solutions that I have provided also demonstrate good programming style, including appropriate comments, meaningful variable names, and minimal use of magic numbers. I encourage you to use good programming style when creating your solutions, so that they compute the correct result while also being clear, easy to understand, and amenable to future updates.

The length of my solution has been included immediately after each exercise's title. While you shouldn't expect the length of your solution to match mine exactly,

I hope that providing the length of my solution will prevent you from going too far astray before reconsidering your approach or seeking assistance. “Solved” appears immediately ahead of the solution’s length when my solution can be found in the second half of the book.

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

Calgary, Canada
December 2024

Ben Stephenson

Acknowledgements I would like to thank Dr. Tom Jenkyns for reviewing the first edition of this book. His helpful comments and suggestions, which resulted in numerous refinements and improvements, were greatly appreciated.

Competing Interests The author has no competing interests to declare that are relevant to the content of this manuscript.

Contents

Part I Exercises

1	Introduction to Programming	3
1.1	Storing and Manipulating Values	4
1.2	Calling Functions	5
1.2.1	Reading Input	6
1.2.2	Displaying Output	7
1.2.3	Importing Additional Functions	8
1.3	Comments	8
1.4	Formatting Values	9
1.5	Working with Strings	12
1.6	Debugging	13
1.6.1	Syntax Errors	13
1.6.2	Runtime Errors	14
1.6.3	Logic Errors	14
1.7	Exercises	16
2	Decision-Making	29
2.1	If Statements	29
2.2	If-Else Statements	30
2.3	If-Elif-Else Statements	31
2.4	If-Elif Statements	33
2.5	Nested If Statements	33
2.6	Boolean Logic	34
2.7	Debugging	35
2.7.1	Syntax Errors	36
2.7.2	Runtime Errors	36
2.7.3	Logic Errors	37
2.8	Exercises	38
3	Repetition	55
3.1	While Loops	55
3.2	For Loops	56
3.3	Nested Loops	58

3.4	Debugging	59
3.4.1	Syntax Errors	59
3.4.2	Runtime Errors	59
3.4.3	Logic Errors	60
3.5	Exercises	61
4	Functions	75
4.1	Functions with Parameters	76
4.2	Variables in Functions	79
4.3	Return Values	79
4.4	Importing Functions into Other Programs	81
4.5	Debugging	82
4.5.1	Syntax Errors	82
4.5.2	Runtime Errors	83
4.5.3	Logic Errors	84
4.6	Exercises	85
5	Lists	97
5.1	Accessing Individual Elements	98
5.2	Loops and Lists	98
5.3	Additional List Operations	101
5.3.1	Adding Elements to a List	101
5.3.2	Removing Elements from a List	102
5.3.3	Rearranging the Elements in a List	102
5.3.4	Searching a List	103
5.4	Lists as Return Values and Arguments	104
5.5	Debugging	105
5.5.1	Syntax Errors	105
5.5.2	Runtime Errors	106
5.5.3	Logic Errors	106
5.6	Exercises	107
6	Dictionaries	125
6.1	Accessing, Modifying and Adding Values	126
6.2	Removing a Key-Value Pair	127
6.3	Additional Dictionary Operations	127
6.4	Loops and Dictionaries	128
6.5	Dictionaries as Arguments and Return Values	129
6.6	Debugging	130
6.6.1	Syntax Errors	130
6.6.2	Runtime Errors	130
6.6.3	Logic Errors	131
6.7	Exercises	131

7	Files and Exceptions	141
7.1	Opening a File	142
7.2	Reading Input from a File	142
7.3	End of Line Characters	144
7.4	Writing Output to a File	145
7.5	Command Line Arguments	146
7.6	Exceptions	148
7.7	Debugging	150
7.7.1	Syntax Errors	150
7.7.2	Runtime Errors	151
7.7.3	Logic Errors	151
7.8	Exercises	152
8	Recursion	165
8.1	Summing Integers	165
8.2	Fibonacci Numbers	167
8.3	Counting Characters	168
8.4	Debugging	170
8.4.1	Syntax Errors	170
8.4.2	Runtime Errors	171
8.4.3	Logic Errors	171
8.5	Exercises	172
Part II Solutions		
9	Solutions to Selected Introductory Exercises	185
10	Solutions to Selected Decision-Making Exercises	195
11	Solutions to Selected Repetition Exercises	209
12	Solutions to Selected Function Exercises	219
13	Solutions to Selected List Exercises	235
14	Solutions to Selected Dictionary Exercises	249
15	Solutions to Selected File and Exception Exercises	257
16	Solutions to Selected Recursion Exercises	271
Index		279

Part I

Exercises



Introduction to Programming

1

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

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

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

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

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

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

Reading this book, and completing the exercises in it, will help you learn to program in Python. Python was selected because it is relatively easy for new programmers to learn, it can be used to solve a wide variety of problems, and it is used extensively in both academia and industry. Python statements that read keyboard input from the user, perform calculations, and generate text output are described in the sections that follow. Later chapters describe additional concepts that can be used to solve larger and more complex problems.

1.1 Storing and Manipulating Values

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

```
x = 5
```

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

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

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

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

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

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

```
y = y - 6
```

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

1.2 Calling Functions

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

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

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

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

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

Rounding a number reduces or increases its value to the closest integer. But this raises a question: Should the value be reduced or increased if it is exactly halfway between two integers? Many people have been taught that the number should be rounded up in this case. But that is *not* what Python's `round` function does. Instead, it rounds such a value to the closest even number. For example, 1.5 is rounded up to 2, but 4.5 is rounded down to 4. This is referred to as "rounding half to even" or "bankers' rounding". Python uses this type of rounding because it eliminates the bias that is introduced when half values are always rounded up.

1.2.1 Reading Input

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

```
a = input()
```

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

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

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

Notice that an argument was passed to the `input` function each time it was called. This argument, which is optional, is a prompt that tells the user what to enter. The

prompt is enclosed in double quotes so that Python knows to treat the characters as a string, instead of interpreting them as the names of functions or variables.

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

```
total = quantity * price
```

This statement will only execute successfully if `quantity` and `price` have been converted to numbers, using the `int` and `float` functions described previously. Attempting to multiply these values without converting them to numbers will cause your program to terminate immediately and display an error message.

1.2.2 Displaying Output

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

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

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

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

All of these values are printed on the same line. The arguments that are enclosed in double quotes are strings that are displayed exactly as typed. The other arguments are variables. When a variable is printed, Python displays the value that is currently stored in it. Spaces are automatically displayed between the items when `print` is called with two or more arguments.

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

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

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

1.2.3 Importing Additional Functions

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

```
import math
```

Functions in the `math` module include `sqrt`, `ceil`, and `sin`, among many others. A function imported from a module is called by using the module's name, followed by a period, followed by the name of the function and its arguments. For example, the following statement computes the square root of `y` (which must have been initialized previously) by calling the `math` module's `sqrt` function, and stores the result in `z`.

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

Some other commonly used Python modules include `random`, `time`, and `sys`. More information about these modules, and many others, can be found online.

1.3 Comments

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

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

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

1.4 Formatting Values

Sometimes the result of a calculation is a floating-point number that has many digits to the right of the decimal point. While one might want to display all of the digits in some programs, there are other circumstances where the value must be rounded to a particular number of decimal places when it is displayed. Another unrelated program might output a large number of integers that need to be lined up in columns, and also need to center headings over those columns. Python has several formatting constructs that allow these, and many other, formatting tasks to be performed. The use of formatted string literals, commonly referred to as f-strings, will be examined in this section, and used throughout the remainder of this book, but you may encounter other formatting mechanisms when examining code from other sources.²

An f-string consists of the letter `f`, followed by a sequence of characters enclosed in double quotes. The value that will be formatted is enclosed in braces within the f-string. This value is frequently a variable, but it can be any Python expression. It is, optionally, followed by additional characters that control how it will be formatted. Such characters are referred to as a format specifier.

The format specifier begins with a colon, and is followed by additional characters that control the formatting. The `:f` format specifier indicates that a value should be formatted as a floating-point number, while `:d` formats a value as a decimal (base 10) integer, and `:s` formats a value as a string. For example, `f" {num:d}"` is a formatted string literal that formats whatever value is currently stored in the `num` variable as an integer.

Characters can precede the `f`, `d`, or `s` to control additional formatting details. Only a limited number of formatting operations will be considered in this section. Many additional formatting tasks can be performed using f-strings and the format specifiers within them, but these additional tasks are outside the scope of this book.

A floating-point number can be formatted to include a specific number of decimal places by including a decimal point and the desired number of digits immediately ahead of the `f` in the format specifier. For example, `:.2f` is used to format a value as a floating-point number with two digits to the right of the decimal point, while `:.7f` indicates that 7 digits will appear to the right of the decimal point. Rounding is performed when the number of digits to the right of the decimal point is reduced. Zeros are added if the number of digits is increased. If the number of digits to the right of the decimal point is omitted, then Python uses 6 digits when formatting the value.

The number of digits to the right of the decimal point cannot be specified when formatting integers and strings, but integers, floating-point numbers, and strings can all be formatted so that they occupy at least some minimum width. This is useful when generating output that includes columns of values. The minimum number of

² Python has several mechanisms for formatting values. These include the formatting operator denoted by `%`, the `format` function and `format` method, template strings, and most recently, f-strings.

characters to use is placed before the `d`, `f`, or `s`, and before the decimal point (if present). For example, `:8d` formats a value as an integer occupying a minimum of 8 characters, while `:6.2f` formats a value as a floating-point number using a minimum of 6 characters, including the decimal point, and the two digits to its right.

When a minimum width is specified, the default behavior is to insert spaces ahead of the value if it is an integer or floating-point number, causing the digits to be right aligned within the indicated width. The value can be centered within the indicated width by including a `^` character immediately after the colon in the format specifier. Left alignment is achieved using the `<` character. The default behavior for strings is to achieve the minimum width by adding spaces after the characters in the string. This causes them to be left aligned. Right alignment for strings is achieved by including a `>` character immediately after the colon. Like integers and floating-point numbers, strings can be centered by using the `^` character.

When large numbers are written by hand, it's common to use commas to separate the digits into groups of three, so that the number is easier to read and understand. These separators can be included in a formatted value by including a comma in the format specifier. The comma is placed immediately after the minimum width (if specified), and before the decimal point in a format specifier that includes one. For example, `: , d` formats the number as an integer using commas as separators, without specifying a minimum number of characters that will be used, while `:10,.2f` will format a number as a floating-point value, with two digits to the right of the decimal point, occupying at least 10 characters, including the commas used to separate groups of three digits.

Finally, it is worth noting that an f-string can include multiple values that need to be formatted (and their format specifiers), and it can contain text that will be included in the formatted string without modification. This allows complex output messages to be created using only one f-string. Each value being formatted is enclosed in braces together with its format specifier. Any characters that are not within braces in the f-string will be retained without modification. It is also worth emphasizing that format specifiers are optional within an f-string. If one only includes the name of a variable (or some other valid Python expression) within braces in the f-string, then the value of that variable or expression will be included in the string using the default formatting for that type of value.

Formatting is often performed as output is being displayed by passing an f-string to `print`. Two examples of this are shown below. The first displays the value of the variable `x`, with exactly two digits to the right of the decimal point. The second statement formats two values before displaying them as part of a larger output message. The first value that is formatted is a string, while the second is an integer. Both of the format specifiers in the second statement could have been omitted because nothing in those format specifiers has overridden Python's default behavior.

```
print(f"{x:.2f}")
print(f"{name:s} ate {numCookies:d} cookies!")
```

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

Code Segment:	<code>f" {count :d} "</code>
Result:	<code>"12"</code>
Explanation:	The value stored in <code>count</code> is formatted as a decimal (base 10) integer.
Code Segment:	<code>f" {x :f} "</code>
Result:	<code>"-2.750000"</code>
Explanation:	The value stored in <code>x</code> is formatted as a floating-point number.
Code Segment:	<code>f" {count :d} and {x :f} "</code>
Result:	<code>"12 and -2.750000"</code>
Explanation:	The value stored in <code>count</code> is formatted as a decimal (base 10) integer, and the value stored in <code>x</code> is formatted as a floating-point number. The other characters in the string are retained without modification.
Code Segment:	<code>f" {count :.4f} "</code>
Result:	<code>"12.0000"</code>
Explanation:	The value stored in <code>count</code> is formatted as a floating-point number, with 4 digits to the right of the decimal point.
Code Segment:	<code>f" {x :.1f} "</code>
Result:	<code>"-2.8"</code>
Explanation:	The value stored in <code>x</code> is formatted as a floating-point number, with 1 digit to the right of the decimal point. The value was rounded when it was formatted because the number of digits to the right of the decimal point was reduced.
Code Segment:	<code>f" {name :10s} "</code>
Result:	<code>"Andrew "</code>
Explanation:	The value stored in <code>name</code> is formatted as a string so that it occupies at least 10 characters. Because <code>name</code> is only six characters long, four trailing spaces are included in the result.
Code Segment:	<code>f" {name :>10s} "</code>
Result:	<code>" Andrew"</code>
Explanation:	The value stored in <code>name</code> is formatted as a string so that it occupies at least 10 characters. Because <code>name</code> is only six characters long, four spaces are included in the result. These spaces are included ahead of the name because a <code>></code> character was included between the colon and the minimum width.
Code Segment:	<code>f" {name :4s} "</code>
Result:	<code>"Andrew"</code>
Explanation:	The value stored in <code>name</code> is formatted as a string so that it occupies at least four spaces. Because <code>name</code> is longer than the indicated minimum length, the resulting string is equal to <code>name</code> .
Code Segment:	<code>f" {count :8d} {x :8.0f} "</code>
Result:	<code>" 12 -3"</code>
Explanation:	Both <code>count</code> and <code>x</code> are formatted so that each occupies a minimum of eight characters. Spaces are included ahead of the values. Rounding was performed on <code>x</code> because the formatted value has fewer decimal places than the stored value.
Code Segment:	<code>f"Items: {count}, {x}, and {name}."</code>
Result:	<code>"Items: 12, -2.75 and Andrew."</code>
Explanation:	Each of the values is formatted using Python's default behavior because no format specifiers were included in the braces.

1.5 Working with Strings

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

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

```
# Read the names from the user.  
first = input("Enter the first name: ")  
last = input("Enter the last name: ")  
  
# Concatenate the strings.  
both = last + ", " + first  
  
# Display the result.  
print(both)
```

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

```
# Read the name from the user.  
first = input("Enter your first name: ")  
  
# Compute its length.  
num_chars = len(first)  
  
# Display the result.  
print(f"Your first name contains {num_chars} characters.")
```

Sometimes, it is necessary to access individual characters within a string. For example, one might want to extract the first character from three strings that contain a first name, middle name, and last name, so that a person's initials can be displayed.

Each character in a string has a unique integer *index*. The first character in the string has index 0, while the last character in the string has an index which is equal to the length of the string, minus 1. A single character in a string is accessed by placing its index inside square brackets after the name of the variable containing the string. The following program demonstrates this by displaying a person's initials.

```
# Read the user's name.  
first = input("Enter your first name: ")  
middle = input("Enter your middle name: ")  
last = input("Enter your last name: ")  
  
# Extract the first character from each string and concatenate them.  
initials = first[0] + middle[0] + last[0]  
  
# Display the initials.  
print(f"Your initials are {initials}.")
```

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

1.6 Debugging

Programming is an error prone process, and even experienced programmers make mistakes. Sometimes, these mistakes, which are often referred to as bugs, are typos where a character was missed, or the wrong character was used. In other cases, the mistake results from a misunderstanding of what the program needs to do, or of how a particular Python statement works. You will make these mistakes too. It's expected. It's normal. It's unavoidable. Debugging is the process of correcting errors in a program. Learning how to identify and correct errors in your programs is a critical part of being an effective programmer.

1.6.1 Syntax Errors

There are rules that describe the structure of a valid Python statement. A statement that does not follow these rules contains a syntax error. For example, when calling a function like `print` or `input`, the name of the function must be followed by an open parenthesis, and there must be a subsequent close parenthesis that matches it. Python will not be able to execute your program if one of these parentheses is missing. Other examples of syntax errors include placing a literal value or function call to the left of an assignment operator, and attempting to include a dollar sign in a floating-point value.

Syntax errors are detected and reported by Python as it loads your program. A message is displayed that describes both the nature of the error and the location where the error was detected. The location is indicated by both the line number in the error message and the `^` that marks the location of the error within the line. The error is removed by moving to the indicated line in your editor and modifying it to correct the problem. Syntax errors are normally quite easy to correct because the error message provided by Python includes both the nature of the error and its location.

An error message is shown below. It was displayed when Python attempted to run a program that was missing the open parenthesis between `print` and its argument. Notice that the error message indicates that the error occurred on line 1 in the file, and the `^` identifies the location of the missing open parenthesis within the line.

```
File "ch1-syntax.py", line 1
    print "Hello, World!"
           ^
```

```
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print ("Hello, World!")?
```

1.6.2 Runtime Errors

Some errors cannot be detected by Python until the program begins to run. These errors are referred to as runtime errors. They are characterized by a program that begins to run, but ends with an error message instead of the program's expected output. Whether or not the error occurs may depend on what input is provided by the user. For example, a program that divides by a value entered by the user will perform the division successfully in most cases, but a runtime error will occur when the user enters zero.

When a program terminates unexpectedly (sometimes referred to as a crash) due to a runtime error, the error message includes both the line on which the error occurred and a brief description of the error. This information can be helpful for locating the error. However, the correction may not be confined to that line. For example, correcting the division by zero error described previously might require several lines of new code to be written, so that the program performs a different calculation when the user enters zero than when a non-zero value is entered. Runtime errors can be more difficult to locate and correct than syntax errors because the error may only occur for some input values, and the change needed to correct the error is often larger. The runtime error reported by Python when attempting to divide by zero is shown below. It shows that the error occurred on line 10, and that the error is the result of dividing by zero, but it does not identify where or how the program needs to be changed to correct the error.

```
Traceback (most recent call last):
  File "ch1-runtime.py", line 10, in <module>
    result = a / b
ZeroDivisionError: division by zero
```

1.6.3 Logic Errors

Syntax errors and runtime errors both cause an error message to be displayed when the program is run, but there are other errors that can be present in a program that do not result in an error message. These mistakes, which are commonly referred

to as logic errors, allow the program to run to completion but cause the program to compute and display an incorrect value, because the statements in the program perform a task that is different from what the programmer intended. Python has no way of knowing that the statements differ from what was intended, so it is unable to report that this type of error has occurred. Instead, the programmer must detect that the error is present by examining the results provided by the program and realizing that they are incorrect. One way that this can be accomplished is by calculating the expected output for the program for a particular input by hand. Then the value calculated by hand can be compared to the result produced by the program.

Logic errors are more difficult to detect and correct than syntax and runtime errors, because Python is unable to provide any information to guide the programmer in their search. While one might be able to detect the error by carefully reading their code, this is often ineffective because the programmer is unable to detect the subtle difference between what they intended and what they actually wrote. Instead, one must approach these problems in a more active manner.

One strategy that can be taken is to carefully trace the code by hand. The programmer considers each statement in the program in sequence and, using a pencil and paper, writes down the value of each variable. When a statement changes the value stored in a variable, the old value is crossed out or erased, and the new value is written down. Writing the values down is an essential part of tracing because few people are capable of accurately tracking the values of several variables in their head. Tracing the code helps reveal errors because the programmer performs each step needed to compute the result by hand, and this allows the programmer to identify the point where an incorrect calculation was performed.

A second approach that can be used to debug logic errors is adding extra calls to the `print` function that display the values of variables used to calculate the program's output. Like tracing, the goal is to examine the values of intermediate steps in the calculation and reveal the point where an incorrect calculation was performed. Once the values have been displayed, the programmer identifies the incorrect value and updates the manner in which it is calculated to remove the error.

Printing values to help debug programs is done so frequently that Python's f-strings have a shorthand that can be used to easily display the current value of a variable in a readable manner. Within an f-string, one can enclose the name of the variable of interest, and an equal sign, in braces. The resulting string will consist of the name of the variable, followed by an equal sign and its current value. For example, the name and current value of the `count` variable are display by the following statement:

```
print(f"{count=}")
```

Debugging is an interesting, rewarding, and essential part of programming. While it can be challenging and frustrating, a significant sense of accomplishment can be achieved when the last bug is resolved, and the program produces its intended output.

1.7 Exercises

Completing the exercises in this chapter will allow you to put the concepts discussed previously into practice. While the tasks that they ask you to carry out are generally small, tackling these exercises is an important step toward being able to create larger programs that solve more interesting problems.

Exercise 1: Mailing Address

(Solved, 9 Lines)

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

Exercise 2: Hello

(9 Lines)

Write a program that asks the user to enter their name. The program should greet the user with a message that includes their name.

Exercise 3: Area of a Room

(Solved, 13 Lines)

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

Exercise 4: Area of a Field

(Solved, 15 Lines)

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

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

Exercise 5: Bottle Deposits*(Solved, 15 Lines)*

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

Write a program that reads the number of containers of each size from the user. Your program should continue by computing and displaying the refund that will be received for returning those containers. Format the output so that it includes a dollar sign and two digits to the right of the decimal point. The output should also include commas that separate groups of three digits when the refund amount is \$1,000.00 or more.

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% 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. All of the output values should include a dollar sign, two digits to the right of the decimal point, and appropriate digit grouping in large values.

Exercise 7: Sum of the First n Positive Integers*(Solved, 11 Lines)*

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

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

Exercise 8: Widgets and Gizmos*(15 Lines)*

An online retailer sells two products: widgets and gizmos. Each widget weighs 75 grams. Each gizmo weighs 112 grams. Write a program that reads the number of

widgets and the number of gizmos from the user. Then your program should compute and display the total weight of the parts.

Exercise 9: Compound Interest

(19 Lines)

Pretend that you have just opened a new savings account that earns 4% interest per year. The interest that it earns is paid at the end of the year and is added to the account's balance. 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 with appropriate formatting for a monetary value.

Exercise 10: Pythagorean Theorem

(Solved, 14 Lines)

The Pythagorean Theorem states that, for right triangles,³ the length of the hypotenuse (the longest side) is equal to the square root of the sum of the squares of the lengths of the other sides. This is commonly recited as $a^2 + b^2 = c^2$, which can be rearranged to $c = \sqrt{a^2 + b^2}$, where c is the length of the hypotenuse, and a and b are the lengths of the shorter sides. Use this information to write a program that reads the lengths of the two shorter sides of a right triangle from the user, and displays the length of the triangle's hypotenuse.

There are right triangles where the lengths of all three sides are integers. Some examples of such include triangles with side lengths 3, 4, and 5, side lengths 5, 12, and 13, and side lengths 8, 15, and 17. Knowledge of these triangles can be helpful when performing construction, because one can mark the building materials at the appropriate lengths on the two shorter sides, and then adjust the angle between the materials until the distance between the marks is equal to the expected hypotenuse. Doing this ensures that there is a right angle between the materials.

³ A right triangle is a triangle where one of the interior angles is exactly 90 degrees.

Exercise 11: Arithmetic

(Solved, 22 Lines)

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

- The sum of a and b
- The difference when b is subtracted from a
- The product of a and b
- The quotient when a is divided by b
- The remainder when a is divided by b
- The result of $\log_{10} a$
- The result of a^b .

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

Exercise 12: Pizza Planning

(Solved, 19 Lines)

Imagine that you are ordering pizzas for a group of friends. Each pizza has eight slices, and everyone in the group will eat the same number of slices. Write a program that begins by reading the number of people in the group and the number of slices of pizza that each person will eat from the user. Use these pieces of information to compute and display the number of pizzas that need to be ordered, noting that it is impossible to order part of a pizza. Your program should also display the number of slices that will be left over.

Hint: The `ceil` function, which is located in the `math` module, can be used to identify the next integer greater than or equal to a particular value.

Exercise 13: 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 14: 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 15: Making Change

(Solved, 37 Lines)

Consider the software that runs on a self-checkout machine. One task that it must be able to perform is to compute the amount of change to provide when a 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 will be used to provide that amount of change. The change should be provided 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 was named by combining the number two and the name of the loonie.

Exercise 16: 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 have been 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 17: 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 18: 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 and the volume of a sphere, both 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 19: Heat Capacity

(Solved, 23 Lines)

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

$$q = mC\Delta T$$

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

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

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

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 20: 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 a cylinder, along with its height, from the user, and computes its volume. Display the result rounded to one decimal place.

Exercise 21: Free Fall

(Solved, 15 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 0.0m/s. Assume that the

acceleration due to gravity is 9.8 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 22: 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 kilopascals, V is the volume in liters, n is the amount of substance in moles, R is the ideal gas constant, equal to $8.314 \frac{\text{L kPa}}{\text{mol K}}$, and T is the temperature in 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 kilopascals (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 23: 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 24: 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 of its 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 25: 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 each 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 the regular polygon constructed from those values.

Exercise 26: 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 that duration.

Exercise 27: Units of Time (Again)

(Solved, 23 Lines)

In this exercise, you will reverse the process described in Exercise 26. Develop a program that begins by reading a number of seconds from the user. Then your program should display the equivalent amount of time in the form D:HH:MM:SS, where D, HH, MM, and SS represent days, hours, minutes and seconds, respectively. The hours, minutes and seconds should all be formatted so that they occupy exactly

two digits. Use your research skills to determine which additional character needs to be included in the format specifier so that leading zeros are used instead of leading spaces when a number is formatted to a particular width.

Exercise 28: Current Time

(10 Lines)

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

Exercise 29: When is Easter?

(33 Lines)

Easter is celebrated on the Sunday immediately after the first full moon following the spring equinox. Because its date includes a lunar component, Easter does not have a fixed date in the Gregorian calendar. Instead, it can occur on any date between March 22 and April 25. The month and day for Easter can be computed for a given year using the Anonymous Gregorian algorithm, which is shown below.

Set a equal to the remainder when $year$ is divided by 19

Set b equal to the floor of $year$ divided by 100

Set c equal to the remainder when $year$ is divided by 100

Set d equal to the floor of b divided by 4

Set e equal to the remainder when b is divided by 4

Set f equal to the floor of $\frac{b+8}{25}$

Set g equal to the floor of $\frac{b-f+1}{3}$

Set h equal to the remainder when $19a + b - d - g + 15$ is divided by 30

Set i equal to the floor of c divided by 4

Set k equal to the remainder when c is divided by 4

Set l equal to the remainder when $32 + 2e + 2i - h - k$ is divided by 7

Set m equal to the floor of $\frac{a+11h+22l}{451}$

Set month equal to the floor of $\frac{h+l-7m+114}{31}$

Set day equal to one plus the remainder when $h+l-7m+114$ is divided by 31

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

Exercise 30: 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 31: 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, such as 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.

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

A similar formula, with different constant values, can be used for temperatures in degrees Fahrenheit and wind speeds in miles per hour.

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 32: 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 Kelvin. The calculations needed to convert between different units of temperature can be found on the Internet.

Exercise 33: 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 34: Sum the Digits

(18 Lines)

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

Exercise 35: 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 36: Day Old Bread

(Solved, 19 Lines)

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

Exercise 37: Length of a Spiral

(16 Lines)

A spiral is a curved line that begins at a central point and circles around it at ever increasing distances, forming a series of connected rings. When the distance between the spiral's adjacent rings is constant, the length of a spiral can be computed using the following formula:

$$\text{length} = \pi \times \text{number of rings} \times \frac{\text{first diameter} + \text{last diameter}}{2}$$

Use this information to create a program that reads the number of rings, the average diameter of the first (smallest) ring, and the average diameter of the last (largest) ring, from the user. Then your program should compute and display the length of the spiral. Round the length of the spiral to one decimal place when it is displayed.



Decision-Making

2

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

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

2.1 If Statements

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

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

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

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

The body of an `if` statement consists of one or more statements that must be indented more than the `if` keyword. It ends before the next line that is indented the same amount as (or less than) the `if` keyword. All of the programs in this book use two spaces to indent the body of an `if` statement, but you can use one space, or several spaces, if you prefer.¹

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

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

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

# Display the message.
print(result)
```

2.2 If-Else Statements

The previous example stored one message into `result` when the number entered by the user was zero, and it stored a different message into `result` when the entered number was non-zero. More generally, the conditions on the `if` statements were constructed so that exactly one of the two `if` statement bodies would execute. There

¹ Most programmers choose to use the same number of spaces every time they indent the body of an `if` statement, though Python does not require this consistency. However, consistent indenting is required for the statements within each `if` statement’s body.

is no way for both bodies to execute, and there is no way for neither body to execute. Such conditions are said to be *mutually exclusive*.

An `if-else` statement consists of an `if` part with a condition and a body, and an `else` part with a body (but no condition). When the statement executes, its condition is evaluated. If the condition evaluates to `True`, then the body of the `if` part executes, and the body of the `else` part is skipped. When the condition evaluates to `False`, the body of the `if` part is skipped, and the body of the `else` part executes. It is impossible for both bodies to execute, and it is impossible to skip both bodies.

Two `if` statements can be replaced with an `if-else` statement when one `if` statement immediately follows the other, the first `if` statement's body doesn't modify a value included in the second `if` statement's condition, and the conditions on the `if` statements are mutually exclusive. Using an `if-else` statement is preferable because only one condition needs to be written, only one condition needs to be evaluated when the program executes, and only one condition needs to be corrected if a bug is discovered at some point in the future. The program that reports whether or not a value is zero, rewritten so that it uses an `if-else` statement, is shown below.

```
# Read a number from the user.  
num = float(input("Enter a number: "))  
  
# Store the appropriate message in result.  
if num == 0:  
    result = "The number was zero."  
else:  
    result = "The number was not zero."  
  
# Display the message.  
print(result)
```

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

2.3 If-Elif-Else Statements

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

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

Consider an extension of the previous example where one message is displayed for positive numbers, a different message is displayed for negative numbers, and yet another different message is displayed if the number is zero. While this problem could be solved using a combination of `if` and/or `if-else` statements, it is well suited to an `if-elif-else` statement because exactly one of three alternatives must be executed.

```
# Read a number from the user.  
num = float(input("Enter a number: "))  
  
# Store the appropriate message in result.  
if num > 0:  
    result = "That's a positive number."  
elif num < 0:  
    result = "That's a negative number."  
else:  
    result = "That's zero."  
  
# Display the message.  
print(result)
```

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

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

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

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

2.4 If-Elif Statements

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

2.5 Nested If Statements

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

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

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

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

# Display the message.
print(result)
```

This program begins by reading a number from the user. If the number is greater than zero, then the body of the outer `if` statement is executed. Its body includes a nested `if` statement which determines what adjective, if any, should be displayed when reporting that a positive number was entered. The inner statement stores `really big` in `adjective` if the entered number is at least 1,000,000, and it stores `big` in `adjective` if the entered number is at least 1,000 but less than 1,000,000. Otherwise, the empty string is stored in `adjective`. The final line in the body of the outer `if` part stores the complete message in `result`. Then the bodies of the outer `elif` part and the outer `else` part are skipped because the body of the outer `if` part was executed. Finally, the program completes by executing the `print` statement that reports the result.

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

2.6 Boolean Logic

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

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

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

<code>x</code>	<code>not x</code>
<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>

The `and` and `or` operators combine two Boolean values to compute a Boolean result. The Boolean expression `x and y` evaluates to `True` if `x` is `True` and `y` is

also True. If x is False, or y is False, or both x and y are False, then x and y evaluates to False. The truth table for the and operator is shown below. It has $2^2 = 4$ rows because the and operator is applied to two variables.

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

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

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

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

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

2.7 Debugging

If statements present new opportunities to introduce syntax errors, runtime errors, and logic errors into your programs. Some examples of these errors, and how to correct them, are explored in the remainder of this section.

2.7.1 Syntax Errors

Syntax errors continue to be relatively easy to identify and correct because the error message that is displayed includes the information needed to locate the error. Common syntax errors involving `if` statements include failing to include the colon after an `if` or `elif` part's condition, writing a condition that includes unbalanced parentheses, inadvertently including a condition after an `else`, and inadvertently using `=` instead of `==` when comparing values. Inconsistent indenting within the body of an `if`, `elif` or `else` part is also a syntax error. The error message below was displayed by Python when a program attempted to compare two values using `=` instead of `==`. The error can be corrected by adding the missing `=` character.

```
File "ch2-syntax.py", line 1
  if a = b:
      ^^^^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead
of '='?
```

2.7.2 Runtime Errors

Using `if` statements also provides additional opportunities to inadvertently introduce runtime errors into your programs. For example, attempting to compare two values with incompatible types (such as a string and an integer) using the inequality operators `<`, `<=`, `>`, and `>=` is a type error that will be detected as your program is running. This error is corrected by ensuring that the operands to the left and the right of the inequality operator have compatible types, either by modifying the condition or by converting the value to a different type earlier in the program. For example, sometimes a type error occurs because the programmer forgot to convert a value entered by the user to the correct type when it was initially read. A program that includes such an error is shown below. Notice that the first value read from the user is not converted to a floating-point number, but the second input value is.

```
# Read the input values from the user.
a = input("Enter a number: ")
b = float(input("Enter a second number: "))

# Display the relationship between the values.
if a < b:
    print(f"{a} is less than {b}.")
elif a > b:
    print(f"{a} is greater than {b}.")
else:
    print(f"{a} is equal to {b}.")
```

The type error reported by Python is shown below. It can be corrected by converting the first value entered by the user to a floating-point value by calling the `float` function when it is read.

```
Traceback (most recent call last):
  File "ch2-runtime.py", line 6, in <module>
    if a < b:
TypeError: '<' not supported between instances of 'str' and
'float'
```

A Python program terminates with a `NameError` when the program attempts to access a variable that has not previously been assigned a value. In many cases, this error is the result of a typo in the variable's name, but it can also be caused by assigning a value to a variable along some paths through the program, but not others. Consider the program below:

```
# Read the temperature from the user.
temp = float(input("Enter the temperature in Fahrenheit: "))

# Determine the state.
if temp < 32:
    state = "ice"
elif temp > 32:
    state = "liquid water"

# Display the temperature and the state at that temperature.
print(f"At {temp} degrees it will be {state}.")
```

It is supposed to report whether water is ice or liquid, based on the temperature entered by the user. The program works as intended when the user enters a temperature near the freezing point. However, if the user enters 32 degrees, then the program terminates with a `NameError`, because neither the body of the `if` nor the body of the `else` executes. This causes the `print` statement to attempt to display the value of `state` when it has not previously been assigned a value. The error can be corrected by adjusting the condition on either the `if` or the `elif` statement, so that it includes the case when the temperature is exactly 32 degrees. Alternatively, an `else` could be added so that 32 degrees is handled as a separate case.

2.7.3 Logic Errors

Sometimes the body of an `if` statement executes when the programmer does not want it to. At other times, the body of an `if` statement is skipped when the programmer wants it to execute. Inadvertently reversing the direction of an inequality operator is one common source of these problems. It is remarkably easy to inadvertently write less than, or less than or equal to, when greater than, or greater than or equal to, is intended, and vice versa. Such a reversal is a logic error. This kind of error can be detected by carefully observing the behavior of your program. Its behavior can be made more explicit by adding a `print` statement to the beginning of the body of the `if`

statement that is not working as intended. In fact, adding multiple, even many, print statements to a program is a common tactic for better understanding exactly which parts of it are executing, and which parts are being skipped. These print statements may simply display a message indicating that a particular point in the program has been reached, or they may be expanded to include the values of one or more variables to further increase the programmer’s understanding of why a particular `if` statement body is executing (or not). While the exact message being printed isn’t particularly important, it is important that each statement displays something different, so that it is easy to ascertain which print statements executed and which did not.

Accidental inclusion or omission of an = character adjacent to a greater than or less than character is another common logic error. This error is more subtle than reversal of the greater than and less than symbols, because the body of the `if` statement is executed or skipped as intended in most cases, but the intended behavior does not occur when the values being compared are equal. Adding print statements to the top of the body of each `if`, `elif` and `else` can make this error more explicit and easier to resolve. It also highlights the importance of testing programs with a variety of values, including those that are at the edges of the ranges being compared.

2.8 Exercises

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

Exercise 38: 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 39: Dog Years

(22 Lines)

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

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

Exercise 40: 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 41: 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 42: Month Name to Number of Days

(Solved, 20 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 43: Sound Levels*(30 Lines)*

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

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

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

Exercise 44: Classifying Triangles*(Solved, 21 Lines)*

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

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

Exercise 45: 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 cases to your `if` statement.

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

Exercise 46: Frequency to Note

(Solved, 42 Lines)

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

Exercise 47: 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 below.

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

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

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, they will not be considered in this exercise.

Exercise 48: 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. These additional holidays will not be considered in this exercise.

Exercise 49: Birthstones

(Solved, 48 Lines)

Over time, traditions have arisen which allocate particular precious or semi-precious gems to each month of the year. These gems are referred to as birthstones. The Gemological Institute of America lists the following gems for each month:

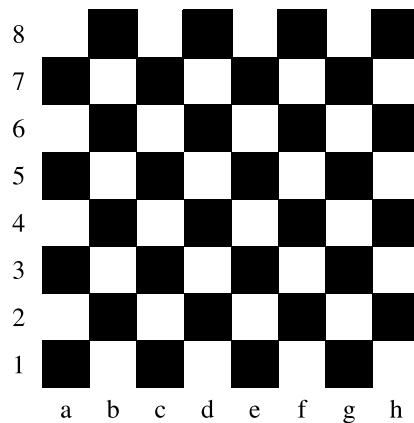
Month	Birthstone(s)
January	Garnet
February	Amethyst
March	Aquamarine and Bloodstone
April	Diamond
May	Emerald
June	Pearl, Alexandrite, and Moonstone
July	Ruby
August	Peridot, Spinel, and Sardonyx
September	Sapphire
October	Opal and Tourmaline
November	Topaz and Citrine
December	Tanzanite, Turquoise, and Zircon

Use the information above to create a program that reads a month from the user and displays the birthstone(s) for that month. Your program should display an appropriate error message if the user does not enter a valid month.

Exercise 50: 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 c5, 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 51: Season from Month and Day

(Solved, 41 Lines)

The year is divided into four seasons: spring, summer, fall (or autumn) and winter. While the exact dates that the seasons change vary a little bit from year to year, the following dates will be used 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 52: Birth Date to Astrological Sign

(47 Lines)

The horoscopes commonly reported in newspapers and online use the position of the sun at the time of one's birth to try to 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–January 19
Aquarius	January 20–February 18
Pisces	February 19–March 20
Aries	March 21–April 19
Taurus	April 20–May 20
Gemini	May 21–June 20
Cancer	June 21–July 22
Leo	July 23–August 22
Virgo	August 23–September 22
Libra	September 23–October 22
Scorpio	October 23–November 21
Sagittarius	November 22–December 21

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

Exercise 53: 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 2036 being another year of the dragon, and 2023 being another year of the hare.

Year	Animal
2024	Dragon
2025	Snake
2026	Horse
2027	Sheep
2028	Monkey
2029	Rooster
2030	Dog
2031	Pig
2032	Rat
2033	Ox
2034	Tiger
2035	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 54: Richter Scale

(30 Lines)

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

Magnitude	Descriptor
Less than 2.0	Micro
2.0–less than 3.0	Very Minor
3.0–less than 4.0	Minor
4.0–less than 5.0	Light
5.0–less than 6.0	Moderate
6.0–less than 7.0	Strong
7.0–less than 8.0	Major
8.0–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 a moderate earthquake.

The Great Chilean Earthquake, which occurred in May of 1960, was the most powerful earthquake ever recorded. Its magnitude was 9.5 on the Richter scale.

Exercise 55: Penalties for Speeding

(Solved, 39 Lines)

In a particular jurisdiction, the penalty for speeding includes both a fine and demerit points applied to the speeder's license. The fine is calculated by multiplying the amount by which the driver was exceeding the speed limit by a penalty value, which increases with the amount of excess speed, as listed below:

Excess Speed	Penalty For Each km/h
1–19 km/h	\$3.00
20–29 km/h	\$4.50
30–49 km/h	\$7.00

Demerit points are assigned for specific excess speed ranges, with more demerit points assigned to greater excesses. The number of demerit points associated with each speed range is listed below:

Excess Speed	Demerit Points
1–15 km/h	0
16–29 km/h	3
30–49 km/h	4

Exceeding the speed limit by 50 km/h or more results in six demerit points and a monetary penalty that is determined by a judge during a mandatory court appearance.

Create a program that begins by reading the amount that the user was traveling in excess of the speed limit. Your program will then report the fine and number of demerit points associated with the offense. Ensure that your program displays appropriate results if the user enters a value less than or equal to 0 (which indicates that they were not speeding), or if the user enters a value of 50 or more.

Exercise 56: Roots of a Quadratic Function

(24 Lines)

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

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

The portion of the expression under the square root sign is called the discriminant. If the discriminant is negative, then the quadratic function does not have any real roots. If the discriminant is 0, then the function has one real root. Otherwise, the function 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 that reports the number of real roots, along with the values of the real roots (if any).

Exercise 57: 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 58: Grade Points to Letter Grade*(47 Lines)*

In the previous exercise, you created a program that converted a letter grade into the equivalent number of grade points. In this exercise, you will create a program that reverses the process and converts 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 appropriately. These should be rounded to the closest letter grade. Your program should report A+ for a grade point value of 4.0 or more.

Exercise 59: Assessing Employees*(Solved, 29 Lines)*

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

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

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

Exercise 60: 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 six colors, as shown below:

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

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 61: Frequency to Name

(31 Lines)

Electromagnetic radiation can be classified into one of seven categories based on its frequency, as shown in the table below:

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

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

Exercise 62: Cell Phone Bill

(44 Lines)

A particular cell phone plan includes 50 minutes of airtime and 50 text messages for \$15.00 a month. Each additional minute of airtime 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% 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), 911 fee, tax, and total. Only display the additional minute and text message charges if the user incurred costs in those categories. Ensure that all of the charges are displayed using 2 decimal places.

Exercise 63: 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 64: 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 18 November 2019, then your program should display a message indicating that the day immediately after 18 November 2019 is 19 November 2019. 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 behaves correctly when the user enters the last day of a month or the last day of a year, and that it behaves correctly for leap years.

Exercise 65: What Day of the Week is January 1?*(32 Lines)*

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

$$\text{day_of_the_week} = (\text{year} + \text{floor}((\text{year} - 1) / 4) - \text{floor}((\text{year} - 1) / 100) + \text{floor}((\text{year} - 1) / 400)) \% 7$$

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

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

Exercise 66: Is a License Plate Valid?*(Solved, 28 Lines)*

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

Write a program that begins by reading a string of characters from the user. Then your program should display a message that indicates 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.

Exercise 67: Roulette Payouts

(Solved, 45 Lines)

A roulette wheel has 38 spaces on it. Of these spaces, 18 are black, 18 are red, and 2 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, and the black spaces are numbered with the remaining integers between 1 and 36.

Many different bets can be placed in roulette. Only the following subset of them will be considered in this exercise:

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

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 paid. 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 only display Pay 0 or Pay 00 after the result of the spin.

Exercise 68: Jersey Numbers

(45 Lines)

Each player in the National Football League has a number on their jersey. This number helps fans identify the players, and also indicates the player's position. The numbers allocated to each position have varied a little bit over the years. During the 2020 season, the numbers that corresponded to each position were:

Offensive positions:

- Quarterbacks: 1–19
- Punters and placekickers: 1–19
- Wide receivers: 10–19 and 80–89
- Running backs: 20–49
- Tight ends: 40–49 and 80–89
- Offensive linemen: 50–79

Defensive positions:

- Defensive backs: 20–49
- Defensive linemen: 50–79 and 90–99
- Linebackers: 40–59 and 90–99

Write a program that reads a player's number and an indication of whether the player is a member of the offense or the defense. Then your program will list all of the positions that the player could be playing. For example, if the user enters number 44 and indicates that it is for a member of the offense, then your program should report that the player is a running back or tight end. If the user enters number 77 and indicates that it is for a member of the defense, then your program should report that the player is a defensive lineman. An error message should be displayed if the user fails to correctly indicate if the player is a member of the offense or the defense. Your program should also display an appropriate error message if a number outside of 1–99 is entered, or if the number entered does not correspond to any position for the offense or defense designation provided.



Repetition

3

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

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

3.1 While Loops

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

Many `while` loop conditions compare a variable holding a value read from the user to some other value. When the value is read from the user within the loop, the user is able to terminate the loop by entering a value that causes the loop's condition

to evaluate to `False`. For example, the following code segment reads values from the user and reports whether each value is positive or negative. The loop terminates when the user enters 0. Neither message is displayed in that case.

```
# Read the first value from the user.  
x = int(input("Enter an integer (0 to quit): "))  
  
# Keep looping while the user enters a non-zero number.  
while x != 0:  
    # Report the nature of the number.  
    if x > 0:  
        print("That's a positive number.")  
    else:  
        print("That's a negative number.")  
  
    # Read the next value from the user.  
    x = int(input("Enter an integer (0 to quit): "))
```

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

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

3.2 For Loops

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

A `for` loop executes once *for* each item in a collection. The collection can be a range of integers, the letters in a string, or as will be described in later chapters, the values stored in a data structure, such as a list. The syntactic structure of a `for`

loop is shown below, where <variable>, <collection>, and <body> are placeholders that must be filled in appropriately.

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

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

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

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

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

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

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

```
# Read the limit from the user.  
limit = int(input("Enter an integer: "))  
  
# Display the positive multiples of 3 up to the limit.  
print(f"The multiples of 3 up to and including {limit} are:")  
for i in range(3, limit + 1, 3):  
    print(i)
```

When this program executes, it begins by reading an integer from the user. Assume that the user entered 11 for this particular run of the program. After the input value is read, execution continues with the print statement that describes the program's output. Then the `for` loop begins to execute.

A range of integers is constructed that begins with 3, and goes up to, but does not include, $11 + 1 = 12$, stepping up by 3 each time. Thus, the range consists of 3, 6, and 9. When the loop executes for the first time, the first integer in the range is assigned to `i`, the body of the loop is executed, and 3 is displayed.

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

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

3.3 Nested Loops

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

```
# Read the first message from the user.  
message = input("Enter a message (blank to quit): ")  
  
# Loop until the message is a blank line.  
while message != "":  
    # Read the number of times the message should be displayed.  
    n = int(input("How many times should it be repeated? "))  
  
    # Display the message the requested number of times.  
    for i in range(n):  
        print(message)  
  
    # Read the next message from the user.  
    message = input("Enter a message (blank to quit): ")
```

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

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

3.4 Debugging

Both `for` loops and `while` loops can execute the statements in the loop's body zero times, one time, or many times. The loop will only solve the intended problem if it executes the correct number of times. When a loop executes the wrong number of times, it may result in a runtime error, or it may result in a logic error. It's also possible to write a program that continues running indefinitely. These errors, and others, are examined in the sections that follow.

3.4.1 Syntax Errors

The syntax errors that arise with `while` loops are similar to those encountered with `if` statements, with an improperly structured condition, a missing colon, or improper indenting being particularly common. In each of these cases, Python will provide an error message that includes the line number where the error occurred, which makes the error relatively easy to locate and correct. Common syntax errors related to `for` loops include omitting `in`, a missing colon, or improper indenting. The error messages reported in these cases also contain information that will help you find and correct the error.

3.4.2 Runtime Errors

When a `for` loop executes, the loop's control variable is created automatically, but other variables accessed in the loop need to be initialized appropriately ahead of the loop. Similarly, the programmer must initialize variables accessed in a `while` loop before the loop begins to execute. Attempting to access a variable inside a loop before the variable has been initialized will result in a `NameError`, as described

previously in Sect. 2.7.2. For example, the program below is supposed to count the number of values entered by the user.

```
line = input("Enter a value (blank line to quit): ")
while line != "":
    count = count + 1
    line = input("Enter a value (blank line to quit): ")

print(f"You entered {count} values!")
```

When it executes, it crashes with the error message shown below because `count = count + 1` can only execute successfully if `count` has previously been assigned a value.

```
Traceback (most recent call last):
  File "ch3-count.py", line 3, in <module>
    count = count + 1
NameError: name 'count' is not defined. Did you mean: 'round'?
```

The error is corrected by setting `count` equal to 0 ahead of the loop.

Runtime errors can also occur when a `for` loop iterates over the wrong collection of values. For example, one might want to create a table that shows the decimal representations of fractions from $\frac{1}{1}$ to $\frac{1}{10}$. The program below attempts to solve this problem, but it crashes with a zero division error before it displays any of its expected output.

```
for i in range(10):
    print(f"1 / {i} is {1/i:.4f}")
```

The zero division error is corrected by calling `range` with two arguments so that the loop begins counting at 1 instead of 0.

In both of these cases, the line number provided in the error message was the location where Python detected the error, but it was not the location that needed to be changed. Both the correction that was needed, and where it was needed, had to be identified by the programmer. This requires a thorough understanding of both the problem that is being solved and the code that is attempting to solve it.

3.4.3 Logic Errors

The fraction printing program presented in Section 3.4.2 contained a logic error, in addition to the runtime error discussed previously. Once the zero division error is corrected by calling `range` with two arguments, the program will display the decimal values of fractions from $\frac{1}{1}$ to $\frac{1}{9}$, but it fails to display $\frac{1}{10}$ because the loop's

body runs one fewer time than intended. This is referred to as an off-by-one error¹. Like the logic errors that have been described in previous chapters, this error can only be detected by examining the output from the program and comparing it to the expected result. The error is corrected by increasing the second argument to `range` from 10 to 11, because the second argument provided to `range` is exclusive.

Another common logic error in programs that include loops is initializing a variable to the wrong value before the loop. Adding print statements that display the relevant variables ahead of the loop can help the programmer determine whether the problem is before the loop or within it. Additional print statements can also be added to the loop's body to better understand how the values are changing from one loop iteration to the next. Once these print statements have been added, you will want to choose input values that only cause the loop's body to run a small number of times. Otherwise, the output that is generated will be too large to analyze by hand.

3.4.3.1 Infinite Loops

The logic errors that have been discussed previously cause the program to display incorrect results. Once a program contains loops, it is also possible that the program will not produce any output because it is stuck in a loop that never ends. When this occurs, the loop's body runs over and over again, and execution is unable to move on to the later parts of the program. Such a never-ending loop is referred to as an *infinite loop*.

Two common causes of an infinite loop are an incorrect `while` loop condition that always evaluates to `True`, and failing to include a statement in the body of a `while` loop that changes one of the variables in the loop's condition. For example, failing to read the next input value from the user in the loop's body can result in an infinite loop. Infinite loops are easily detected by printing a value inside the loop's body. If that value continues to be printed indefinitely, then the program contains an infinite loop that must be corrected. Unfortunately, like other logic errors, the programmer must find and correct the infinite loop without an error message to guide their search.

3.5 Exercises

The following exercises should all be completed with loops. In some cases, the exercise specifies what type of loop to use. In other cases, you must make this decision yourself. Some of the exercises can be completed easily with both `for` loops and `while` loops. Other exercises are much better suited to one type of loop than the other. In addition, some of the exercises require multiple loops. When multiple loops

¹ An error that causes a loop's body to run one more time than intended is also referred to as an off-by-one error.

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 69: 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 70: Discount Table

(18 Lines)

A particular retailer is having a 60 percent off sale on a variety of discontinued products. The retailer would like to help its customers determine the reduced price of the merchandise by having a printed discount table on the shelf that shows the original prices and the prices after the discount has been applied. Write a program that uses a loop to generate a table that shows 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 71: 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 the columns. The formula for converting between degrees Celsius and degrees Fahrenheit can be found on the Internet.

Exercise 72: No More Pennies

(Solved, 36 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 purchases 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 73: Compute the Perimeter of a Polygon

(Solved, 47 Lines)

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

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

Exercise 74: Compute a Grade Point Average

(62 Lines)

Exercise 57 includes a table that shows the conversion from letter grades to grade points at a particular academic institution. In this exercise, you will use the data in that table to 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 and the average should be displayed. 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 $\frac{4.0+2.3+3.0}{3} = 3.1$.

You may find your solution to Exercise 57 helpful when completing this exercise. No error checking needs to be performed by your program. It can assume that each value entered by the user will be a valid letter grade or a blank line.

Exercise 75: 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 76: Parity Bits

(Solved, 27 Lines)

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

Parity bits can be computed for either even parity or odd parity. If even parity is selected, then the parity bit that is transmitted is chosen so that the total number of one bits transmitted (8 bits of data plus the parity bit) is even. When odd parity is selected, the parity bit is chosen so that the total number of one bits transmitted is odd. Errors are detected by the receiver of the data by counting the number of one bits. If even parity is being used, but the number of one bits received was odd, then the receiver knows that an error occurred. Similarly, the receiver knows that an error occurred if odd parity is being used, but the number of one bits received was even.

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 either use a loop or the `count` method to determine how many zeros and ones are in the string. Information about the `count` method is available online.

Exercise 77: Approximate π

(23 Lines)

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

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

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

Exercise 78: Fizz Buzz

(17 Lines)

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

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

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

Write a program that displays the answers for the first 100 numbers in fizz buzz. Each answer should be displayed on its own line.

Exercise 79: Universal Product Codes

(Solved, 47 Lines)

Many products available for sale are marked with a Universal Product Code, commonly referred to as a UPC code or barcode. This code consists of alternating black and white lines of various widths that encode a 12-digit number. Scanners are able to read the barcode to identify the product. Then the price of the product is retrieved from a database and added to the customer's total.

The final digit in a universal product code is a check digit. This digit helps ensure that the code was read correctly. A computer scanning the code can compare the check digit computed from the first 11 digits of the code to the check digit that was scanned. If the digits are different, then an error occurred when the code was read, and it needs to be scanned again. The check digit is calculated using the algorithm below.

```
Initialize total to 0
For each digit at even position (starting from 0)
    Add three times the value of the digit to total
For each digit at odd position (up to and including 9)
    Add the value of the digit to total
If total is evenly divisible by 10 then
    The check digit is equal to 0
Else
    The check digit is 10 minus the remainder when total is divided by 10
```

Write a program that reads a 12-digit number from the user. Determine whether or not the entered number is a valid universal product code by computing the check digit using the algorithm above and comparing it to the final digit entered by the user. If the computed check digit matches the final digit in the provided number, then your program should report that the entered value is a valid universal product code. Otherwise, your program should report that the number is not a valid universal product code, along with the correct check digit. Ensure that your program displays an appropriate error message if the user enters something other than a 12-digit number.

Exercise 80: Caesar Cipher

(Solved, 35 Lines)

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

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

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

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

Exercise 81: Square Root

(14 Lines)

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

Read x from the user

Initialize $guess$ to $x/2$

While $guess$ is not good enough **do**

 Update $guess$ to be the average of $guess$ and $x/guess$

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

Exercise 82: Is a String a Palindrome?

(Solved, 26 Lines)

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

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

Exercise 83: 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 82 so that it ignores spacing while determining whether or not a string is a palindrome. For an additional challenge, further extend your solution so that it also ignores punctuation marks and treats uppercase and lowercase letters as equivalent.

Exercise 84: Multiplication Table

(Solved, 21 Lines)

In this exercise, you will create a program that displays a multiplication table. It will include 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 its top consisting of the numbers 1 through 10. It should also include labels down its left side. The expected output from your 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 including `end=""` as the last argument to `print`. 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 A.

Exercise 85: Rolling Dice

(37 Lines)

Dice are available with different numbers of sides. While 6-sided dice are, perhaps, the most common, dice with 4, 8, 10, 12, and 20 sides are also readily available. When a game uses several dice with different numbers of sides, a mechanism is needed to describe the number and type of dice that should be rolled. One notation that is used specifies the number of dice, immediately followed by a lowercase d, immediately followed by the number of sides on the dice. For example, 3d8 indicates that three 8-sided dice should be rolled and their total should be computed, while 5d6 indicates that five 6-sided dice should be rolled and totaled. When only one die needs to be rolled, the number of dice can be omitted. As a result, both d20 and 1d20 indicate that a single 20-sided die should be rolled.

Create a program that reads rolls from the user expressed using the notation described previously. Once the roll has been read, your program should use Python's random number generator to simulate the roll. Print a message that shows the number rolled on each die, followed by the total for all of the dice that were rolled. Allow the user to continue entering rolls until a blank line is entered. You may assume that the user will always enter a valid roll. Ensure that your program behaves correctly when the user does not include the number of dice to the left of the d.

Exercise 86: The Collatz Conjecture

(22 Lines)

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

Start with any positive integer as the only term in the sequence

While the last term in the sequence is not equal to 1 **do**

If the last term is even **then**

 Add another term to the sequence by dividing the last term by 2 (using floor division so the result is an integer)

Else

 Add another term to the sequence by multiplying the last term by 3 and adding 1

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

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

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

Exercise 87: Greatest Common Divisor

(Solved, 17 Lines)

The greatest common divisor of two positive integers, n and m , is the largest integer, 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 compute and report their greatest common divisor.

Exercise 88: Prime Factors

(27 Lines)

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

Initialize *factor* to 2

While *factor* is less than or equal to n **do**

If n is evenly divisible by *factor* **then**

 Conclude that *factor* is a factor of n

 Divide n by *factor* using floor division

Else

 Increase *factor* by 1

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

Enter an integer (2 or greater): 72

The prime factors of 72 are:

2

2

2

3

3

Exercise 89: 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 90: Decimal to Binary

(Solved, 27 Lines)

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

Let *result* be an empty string

Let *q* represent the number to convert

Repeat

 Set *r* equal to the remainder when *q* is divided by 2

 Convert *r* to a string and add it to the beginning of *result*

 Divide *q* by 2, discarding any remainder, and store the result back into *q*

Until *q* is 0

Exercise 91: Armstrong Numbers

(30 Lines)

A non-negative integer is an Armstrong number if the sum of all its digits, each raised to the number of digits in the number, is equal to the number itself. For example, 371 is an Armstrong number because $3^3 + 7^3 + 1^3 = 371$. Similarly, 92,727 is an

Armstrong number because $9^5 + 2^5 + 7^5 + 2^5 + 7^5 = 92,727$. The integers 0 through 9 are also Armstrong numbers.

Create a program that reads an integer from the user and reports whether or not it is an Armstrong number. Your program should display an appropriate error message if the integer entered by the user is negative.

It has been proved that there are only 88 Armstrong numbers, the largest of which is 115,132,219,018,763,992,565,095,597,973,971,522,401. Ensure that your program correctly reports that it is an Armstrong number.

Exercise 92: Maximum Integer

(Solved, 39 Lines)

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

Take a moment and think about how you would solve this problem on paper. Many people would check each integer in sequence and ask themselves 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 the question posed at the end of the previous paragraph can be answered using probability theory, this exercise is 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, and record that update was performed. Display each integer after you generate it. Include a notation with the integers that are a new maximum value.

After it has displayed 100 integers, your program should display the maximum value encountered, along with the number of times the maximum value was updated. 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 93: 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, these questions will be explored 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 or 3 consecutive tails occur. Display an H each time the outcome is heads, and a T each time the outcome is tails, with all of the outcomes for one simulation on the same line. Then display the number of flips that were needed to reach 3 consecutive occurrences of the same outcome. When your program is run, it should perform the simulation 10 times and report the average number of flips needed. Sample output is shown below:

```
H T T T (4 flips)
H H T T H T H T 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.
```

Exercise 94: Monty Hall Problem

(Solved, 35 Lines)

The Monty Hall problem provides a contestant with the opportunity to choose one of three doors (numbered 1, 2, and 3). There is a car behind one of the doors, which is the prize the contestant is trying to win. The other doors have something undesirable behind them, often represented by a goat. After the contestant chooses their door, the game's host opens one of the two doors that was not selected, always revealing an undesirable item. The contestant is then given the opportunity to either keep the door that they initially selected, or switch their selection to the other closed door. After the contestant makes their choice, their selected door is opened and their prize is revealed. Whether or not switching doors impacts the contestant's chance of winning the car was debated when this problem was initially posed, with some arguing that choosing to switch doors benefited the contestant, while others argued that switching doors had no impact.

Computer simulation is one technique that can be used to help identify the best course of action for the contestant. Create a program that simulates at least 100,000 instances of the Monty Hall problem. In each simulation, your program should begin by randomly selecting the door hiding the car. Then your program should randomly select one of the doors as the player's choice. Once these values have been selected, the program should select one door for the host to open. This door cannot be the door that the player selected, and it cannot be the door hiding the car. If there are two doors that meet these criteria, then your program should randomly select one of them. Finally, your simulation should determine whether or not it would have been beneficial for the player to switch their selection, and update a counter accordingly. Once all of the games have been simulated, your program should display a message showing what percentage of the time switching was beneficial. Did your simulation show that switching doors was beneficial? Did the outcome of your simulation surprise you?



Functions

4

As programs grow, steps need to be taken to make them easier to develop and debug. One approach that can be used is breaking the program's code into sections called *functions*.

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

The programs that you have written previously called functions like `print`, `input`, `int`, and `float`. These functions were defined by Python's creators, and they can be called in any Python program. In this chapter, you will learn how to define and call your own functions.

A function definition begins with a line that consists of `def`, followed by the name of the function that is being defined, followed by an open parenthesis, a close parenthesis, and a colon. This line is followed by the body of the function, which is the collection of statements that will execute when the function is called. Like the bodies of `if` statements and loops, the bodies of functions are indented. A function's body ends before the next line that is indented the same amount as (or less than) the

line that begins with `def`. For example, the following lines of code define a function that draws a box constructed from asterisk characters.

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

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

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

```
drawBox()
```

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

4.1 Functions with Parameters

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

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

An improved version of the `drawBox` function is shown below. The new definition includes two parameter variables, which are separated by a comma and hold the width and height of the box, respectively. The values stored in these variables control how many lines of output are displayed and how many characters are displayed on each line. The function's body also includes an `if` statement that verifies that the argument

values are reasonable. If the arguments are invalid, then the `quit` function is called and the program ends immediately.

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

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

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

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

which outlines the box with at symbols and fills the box with periods, is included at the end of the program.

```
## Draw a box.  
# @param width the width of the box  
# @param height the height of the box  
# @param outline the character used for the outline of the box  
# @param fill the character used to fill the box  
def drawBox(width, height, outline, fill):  
    # A box that is smaller than 2x2 cannot be drawn by this function.  
    if width < 2 or height < 2:  
        print("Error: The width or height is too small.")  
        quit()  
  
    # Draw the top of the box.  
    print(outline * width)  
  
    # Draw the sides of the box.  
    for i in range(height - 2):  
        print(outline + fill * (width - 2) + outline)  
  
    # Draw the bottom of the box.  
    print(outline * width)  
  
# Demonstrate the drawBox function.  
drawBox(14, 5, "@", ".")
```

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

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

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

Now consider the following call to `drawBox`:

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

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

ଓ ও ও ও ও ও ও ও ও ও ও ও ও

4.2 Variables in Functions

When a variable is created inside a function, the variable is *local* to that function. This means that the variable only exists when the function is executing, and that it can only be accessed within the body of that function. The variable ceases to exist when the function returns, and as such, it cannot be accessed after that time. The `drawBox` function uses several variables to perform its task. These include parameter variables, such as `width` and `fill`, that are created when the function is called, as well as the `for` loop control variable, `i`, that is created when the loop begins to execute. All of these are local variables that can only be accessed within this function. Variables created with assignment statements in the body of a function are also local variables.

4.3 Return Values

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

A function returns a value using the `return` keyword, followed by the value that will be returned. When the `return` keyword executes, the function ends immediately, and control returns to the location where the function was called. For example,

the following statement immediately ends the function's execution and returns 5 to the location from which it was called.

```
return 5
```

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

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

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

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

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

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

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

    # Return the sum.
    return s
```

The function begins by using an `if` statement to determine whether or not `r` is one. If it is, the sum is computed as `a * n`, and the function immediately returns this value without executing the remaining lines in the function's body. When `r` is

not equal to one, the body of the `if` statement is skipped, and the sum of the first n terms is computed and stored in `s`. Then the value stored in `s` is returned to the location from which the function was called.

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

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

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

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

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

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

The preceding program included a function named `main` which read values from the user, computed the desired results, and displayed them. Then the final line in the program called the `main` function. Neither this structure, nor the name `main`, is required by Python, but both are widely used by Python programmers. Following these conventions will make it easier for other people to read and modify your programs, and also allows functions to be imported into other programs, as described below.

4.4 Importing Functions into Other Programs

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

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

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

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

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

4.5 Debugging

Dividing larger programs into functions makes them easier to develop, because the programmer can focus on only one part of the problem at a time. It also makes them easier to debug for the same reason. But functions also provide new opportunities for you to introduce errors into your programs. The following sections examine some of the errors that can arise when a program is separated into functions, and how to resolve them.

4.5.1 Syntax Errors

Some of the syntax errors that are common with functions are similar to those that you encountered when writing `if` statements and loops, such as omitting the colon, or failing to indent the body correctly. The error messages generated by Python in these cases include the location of the error. This generally makes these errors reasonably straightforward to resolve.

Omitting the parentheses when defining a function that does not include any parameters is also a syntax error, as is failing to separate any parameter variables in the function's definition with commas. The error messages displayed in these cases don't explicitly tell the programmer that the parentheses or commas are missing, but they do identify the locations where they are needed. An example of such an error is shown below. The ^ character identifies the location where the missing parentheses need to be inserted.

```
File "ch4-syntax.py", line 1
  def main:
  ^
SyntaxError: invalid syntax
```

It is also worth highlighting that function definitions begin with `def` rather than `func` or `function`. Attempting to begin a function definition with an incorrect or mistyped keyword is another syntax error that can occur when creating your own functions.

4.5.2 Runtime Errors

A function may terminate with a runtime error because there is an error in the statements that make up the body of the function. Runtime errors can also occur because the function was called with incorrect arguments and these incorrect values are causing a function that was written correctly to crash. Determining which of these situations has occurred is important so that the search for the error can be focused in the correct area.

Adding `print` statements to the beginning of a function that display the values of the function's parameter variables can help a programmer determine whether or not the values passed to the function are correct. Python's `assert` keyword can also be used to accomplish this goal. The `assert` keyword is immediately followed by a condition that is expected to evaluate to `True`. When this occurs, execution continues with the next statement. If the condition evaluates to `False`, then the program terminates with a runtime error. Using assertions instead of `print` statements is valuable because a message is only displayed when there is a problem. This saves the programmer from having to compare the displayed values to the expected values. It also ensures that the programmer is aware of an error when one occurs because the program terminates with a runtime error.

Assertions at the beginning of functions can be used to ensure that the parameter variables' values are within desired ranges. They can also be used to verify that the provided values have the correct types using Python's `type` function and `is` keyword. For example, the function below includes parameters for a person's name and age. The assertions at the beginning of the function ensure that the person's age is

an integer or floating-point number greater than or equal to zero. These are followed by additional assertions that ensure that the person's name is a non-empty string.

```
def displayPerson(name, age):
    assert type(age) is int or type(age) is float
    assert age >= 0
    assert type(name) is str
    assert name != ""

    print(f"{name} is {age} years old!")
```

Including assertions at the beginning of a function ensures that the program crashes as soon as the function begins executing if the function's arguments are invalid. This indicates that the problem exists in the code preceding the function call, rather than in the body of the function, and allows the programmer to focus on that area. The error message displayed when the `displayPerson` function is called with a negative age is shown below. The assertion that failed (`age >= 0`) is near the bottom of the error message.

```
Traceback (most recent call last):
  File "ch4-name-and-age.py", line 9, in <module>
    displayPerson("Jonathan", -1)
  File "ch4-name-and-age.py", line 3, in displayPerson
    assert age >= 0
AssertionError
```

Other examples of runtime errors that can occur include attempting to call a function with the wrong number of arguments, misspelling the name of a function when it is called, or attempting to access a variable that is local to a function outside of it. In each of these cases, Python will display an error message that identifies the location of the error, allowing the programmer to correct the error relatively easily.

Failing to include a `return` statement in a function that is supposed to return a value often results in a type error because `None`, which Python uses to represent the absence of any value, was returned by the function instead of the expected value. Unfortunately, the line number provided in the error message indicates where the calculation involving `None` occurred, rather than the location where the `return` statement is needed, so knowing that a missing `return` statement is a common cause of this error will help you correct it more quickly.

4.5.3 Logic Errors

Many of the functions that you create will include parameters. When these functions are called, the arguments must be provided in the same order as the parameter variables. Failing to do so will result in either a logic error or a runtime error, depending on the nature of the function. Judicious use of assertions can detect out-of-order arguments in some cases, but there is no way for Python to detect this error when

a function takes two (or more) parameters of the same type with the same range of permissible values. The risk of this error can be minimized by providing meaningful names to parameter variables, including good quality comments ahead of function definitions, and ordering the parameters in a consistent manner when you create multiple functions that take the same (or similar) parameters. Printing the values of the parameter variables at the beginning of the function’s body can also help reveal this kind of error.

Python functions are normally given unique names. However, it is possible to define two functions with the same name. When this occurs, the second definition replaces the first, and all subsequent calls to the function will use the new behavior. This can be a useful feature when comparing different solutions to a problem, but can also result in the inadvertent replacement of one function with another. Adding a `print` statement to the beginning of a function can help you verify that the expected function is the one that is executing. Searching for `def`, followed by the name of the function, in your Python file can also help reveal an inadvertent reuse of a function name.

4.6 Exercises

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

Exercise 95: 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. Reusing parts of your solution to Exercise 10 may help you complete this exercise more quickly.

Exercise 96: 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.

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 97: Scores and Years

(Solved, 28 Lines)

The Gettysburg Address famously opens with the words “Four score and seven years”. These words represent a duration of 87 years because each score represents 20 years, for a total of $4 \times 20 + 7 = 87$. Use this information to write a function that computes the total number of years represented by a number of scores and a number of years. Your function will take two parameters and return the total as its only result. Write a main program that reads input values from the user, provides them to your function, and prints the result.

Exercise 98: Shipping Calculator

(23 Lines)

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

Exercise 99: Median of Three Values

(Solved, 41 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 100: Storage Units

(51 Lines)

Computer files can range in size from a few bytes to trillions of bytes, or more. When the size of a large file is displayed, it may be desirable to use a unit other than bytes to reduce the number of digits that are needed. Simultaneously, units that are appropriate for a large file do not work well for a small file because many digits will be needed to the right of the decimal point to prevent the size from being represented as zero. These challenges can be addressed by using different units for files with different sizes.

The standard metric prefixes kilo, mega, giga, and tera are used ahead of byte to represent one thousand, one million, one billion, and one trillion bytes, respectively. Confusingly, these prefixes have also been used, at times, to represent 2^{10} , 2^{20} , 2^{30} , and 2^{40} bytes. In recent years, this confusion has been lessened by using prefixes kibi, mebi, gibi, and tebi (abbreviated Ki, Mi, Gi, and Ti) to represent the powers of 2.

Write a function that takes an integer number of bytes as its only parameter. The function will return a string which represents that number of bytes using a power-of-two unit that is appropriate for the magnitude of the value. More specifically, the returned string will represent the number of bytes using the largest power-of-two unit that is less than or equal to the provided value. The size should be formatted so that it has two digits to the right of the decimal point, followed by a space, followed by the power-of-two unit in its abbreviated form. For example, passing 2,000,000 to the function should return 1.91 MiB while passing 1024 to it should return 1.00 KiB.

While the largest prefix currently in common use for home computing is tebi, the pebi and exbi prefixes, which represent 2^{50} and 2^{60} , respectively, are used in some commercial and scientific contexts. Consider including support for these prefixes in your function.

Exercise 101: 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). Use an assertion so that your function terminates with a runtime error if the provided value is outside of this range. Include a main program that demonstrates your function by displaying each integer from 1 to 12 and its ordinal number. Your main program should only run when your file has not been imported into another program.

Exercise 102: The Twelve Days of Christmas

(Solved, 52 Lines)

The Twelve Days of Christmas is a repetitive song that describes an increasingly long list of gifts given to one's true love on each of 12 days. A single gift is given on the first day. A new gift is added to the collection on each additional day, and then the complete collection is given. 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 gave to me:
A partridge in a pear tree.

On the second day of Christmas
my true love gave to me:
Two turtle doves,
And a partridge in a pear tree.

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

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

Each item that is given to the recipient in the song should only appear in your program once, with the possible exception of the partridge. It may appear twice if that helps you handle the difference between “A partridge in a pear tree” in the first verse and “And a partridge in a pear tree” in the subsequent verses. Import your solution to Exercise 101 to help you complete this exercise.

Exercise 103: 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 63 helpful when solving this problem.

Exercise 104: Gregorian Date to Ordinal Date*(72 Lines)*

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

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

Exercise 105: Ordinal Date to Gregorian Date*(103 Lines)*

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

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

Exercise 106: Center a String in the Terminal Window

(Solved, 46 Lines)

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

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

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

There are at least two distinct approaches that can be taken to solve this exercise. One option is to implement the steps outlined previously using an `if` statement, string replication and string concatenation. The other option is to achieve the desired result with an f-string. The f-string approach may require you to do a little bit of research because it makes use of a feature that is not discussed in this book.

Exercise 107: Is It a Valid Triangle?

(33 Lines)

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

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

Exercise 108: Capitalize It

(Solved, 70 Lines)

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

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

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

Exercise 109: Does a String Represent an Integer?

(Solved, 33 Lines)

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

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

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

Exercise 110: 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 this book, `^` will be used to represent exponentiation. Using `^` instead of `**` will make these exercises easier because an operator will always be a single character.

Exercise 111: Is a Number Prime?

(Solved, 28 Lines)

A prime number is an integer greater than one that is only divisible by one and itself. Create 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 112: 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 111 while completing this exercise.

Exercise 113: 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 33–126 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 114: Random License Plate

(45 Lines)

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

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

Exercise 115: Check a Password

(Solved, 41 Lines)

In this exercise, you will write a function that determines whether or not a password meets the requirements imposed by a particular system administrator. That administrator requires that all passwords are at least eight characters long and contain at least one uppercase letter, at least one lowercase letter, and at least one digit. Your function should return `True` if the password provided to it as its only parameter meets these requirements. Otherwise, it should return `False`. Include a main program that reads a password from the user and reports whether or not it meets the requirements. Ensure that your main program only runs when your solution has not been imported into another file.

Exercise 116: Random Password with Constraints*(22 Lines)*

Using your solutions to Exercises 113 and 115, write a program that generates and displays a random password that meets the constraints described previously. Count and display the number of attempts that were needed to generate such a password. 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 117: Hexadecimal and Decimal Digits*(41 Lines)*

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

Exercise 118: Arbitrary Base Conversions*(Solved, 71 Lines)*

Write a program that allows the user to convert a number from one base to another. Your program should support bases between 2 and 16 for both the input and the result. 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 89, 90 and 117 helpful when completing this exercise.

Exercise 119: Reduce a Fraction to Lowest Terms*(Solved, 47 Lines)*

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

main program that allows the user to enter a numerator and denominator. Then your program should display the reduced fraction.

Hint: In Exercise 87, you wrote a program that computed the greatest common divisor of two positive integers. You may find that code useful when completing this exercise.

Exercise 120: Reduce Measures

(*Solved, 110 Lines*)

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

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

Exercise 121: 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 twentieth century. You will probably find your solution to Exercise 103 helpful when completing this exercise.

Exercise 122: Formatting a Page Range

(65 Lines)

Sometimes authors refer to pages elsewhere in their own work, or to pages within another work. When only one page is referenced, the author simply uses the number for that page, but things become more complicated when a range of pages is referenced. Should the author use all of the digits in the higher page number, only the digits that have changed in the higher page number, or something between these two extremes? A particular style manual advises authors to format page ranges in the following manner:

- All digits are included for the lower page number.
- If the lower page number is less than 100, then all digits in the higher page number are used.
- If the lower page number is divisible by 100, then all digits in the higher page number are used.
- If the higher page number has more digits than the lower page number, then all digits in the higher page number are used.
- In all other cases, only the changed digits are included in the higher page number, unless the second-to-last digit in the lower number is not 0, in which case two digits are included in the higher page number even if only one digit has changed.

Write a function that formats a page range in the manner outlined previously. The function will take two parameters, which will be the lower and higher page numbers, respectively. It will return a string consisting of the lower page number, followed by a dash, followed by the appropriate digits from the higher page number. Include a main program that tests your function for several page ranges. Some cases that you should consider are shown in the Table 4.1.

Table 4.1 Formatted page ranges

Lower number	Higher number	Formatted range
5	12	5–12
92	94	92–94
202	204	202–4
212	214	212–14
242	357	242–357
300	301	300–301
302	402	302–402
1101	1102	1101–2
1111	1112	1111–12
1111	1211	1111–211



Lists

5

All of the variables used in previous chapters held one value. That value could be an integer, a Boolean, a string, or a value of some other type. While using one variable for each value is practical for small problems, it quickly becomes untenable when working with larger amounts of data. Lists overcome this problem by allowing several, even many, values to be stored in one variable.

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

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

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

5.1 Accessing Individual Elements

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

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

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

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

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

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

5.2 Loops and Lists

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

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

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

# Display the total.
print(f"The total is {total}.")
```

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

Once the body of the loop has executed for the first time, control returns to the top of the loop. Then the second element in `data` is copied into `value`, and the loop body executes again, which adds this new value to the total. This process continues until the loop has executed once for each element in the list, and the total of all of the elements has been computed. Then the result is displayed and the program terminates.

Sometimes loops are constructed which iterate over a list's indices instead of its values. To construct such a loop, one needs to be able to determine how many elements are in a list. This can be accomplished using the `len` function.¹ It takes one argument, which is a list, and it returns the number of elements in the list.²

The `len` function can be used with the `range` function to construct a collection of integers that includes all of the indices for a list. When the `range` function is called with one argument, it returns a collection of sequential integers, starting from 0, and counting up toward the value provided as an argument. As a result, a collection of integers containing only the valid indices for the list can be constructed by calling `range` with the length of the list as its only argument.

A subset of the indices can be constructed by providing a second argument to `range`. When two arguments are provided, the `range` function returns sequential integers, starting with the first value, counting up toward (but not reaching) the second value. The following program demonstrates this by using a `for` loop to iterate through all of `data`'s indices, except the first, to identify the position of the largest element in `data`.

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

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

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

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

The `for` loop begins its execution by storing 1 into `i`. Then the loop body runs for the first time. It compares the value in `data` at index `i`, which is 1.41, to the

¹ The `len` function can also be used to determine how many characters are in a string, as previously described in Sect. 1.5.

² The `len` function returns 0 if the list passed to it is empty.

value in `data` at index `largest_pos`, which is 1.62. Since the element at index `i` is smaller, the `if` statement's condition evaluates to `False`, and the body of the `if` statement is skipped.

Now control returns to the top of the loop. The next value in the range, which is 2, is stored into `i`, and the body of the loop executes for a second time. The value at index `i`, which is 3.14, is compared with the value at index `largest_pos`, which is 1.62. Since the value at index `i` is larger, the body of the `if` statement executes, and `largest_pos` is set equal to `i`, which is 2.

The loop runs one more time with `i` equal to 3. The element at index `i`, which is 2.71, is less than the element at index `largest_pos`, which is 3.14, so the body of the `if` statement is skipped. Then the loop terminates, and the program reports that the largest value is 3.14, which is at index 2.

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

```
# Initialize data to a list of 5 integers.  
data = [0, -1, 4, 1, 0]  
  
# Loop while i is a valid index and the value at index i is not a positive number.  
i = 0  
while i < len(data) and data[i] <= 0:  
    i = i + 1  
  
# If i is less than the length of data, then the loop terminated because a positive number was  
# found. Otherwise, i will be equal to the length of data, indicating that a positive number  
# was not found.  
if i < len(data):  
    print(f"The first positive number is at index {i}.")  
else:  
    print("The list does not contain a positive number.")
```

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

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

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

5.3 Additional List Operations

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

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

A method is applied to a list by using the name of a variable containing a list,³ followed by a period, followed by the method's name. Like a function call, the name of the method is followed by parentheses that surround a comma separated collection of arguments. Some methods return a result. This result can be stored in a variable using an assignment statement, passed as an argument to another method or function call, or used as part of a calculation, just like the result returned by a function.

5.3.1 Adding Elements to a List

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

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

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

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

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

³ Methods can also be applied to a list literal enclosed in square brackets using the same syntax, but there is rarely a need to do so.

5.3.2 Removing Elements from a List

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

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

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

```
data = [2.71, 3.14, 1.41, 1.62]  
  
data.remove(1.62)      # Remove 1.62 from the list.  
last = data.pop()      # Remove the last element from the list.  
  
print(data)  
print(last)
```

5.3.3 Rearranging the Elements in a List

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

```
# Create a list.  
data = [2.71, 3.14, 1.41, 1.62]  
  
# Swap the element at index 1 with the element at index 3.  
temp = data[1]  
data[1] = data[3]  
data[3] = temp  
  
# Display the modified list.  
print(data)
```

When these statements execute, `data` is initialized to `[2.71, 3.14, 1.41, 1.62]`. Then the value at index 1, which is 3.14, is copied into `temp`. This is followed by a line which copies the value at index 3 to index 1. Finally, the value in

`temp` is stored into the list at index 3. When the `print` statement executes, it displays `[2.71, 1.62, 1.41, 3.14]`.

There are two methods that rearrange the elements in a list. The `reverse` method reverses the order of the elements in the list, so that the last element becomes the first and the first element becomes the last. The `sort` method sorts the elements in a list into ascending order.⁴ Both `reverse` and `sort` can be applied to a list without providing any arguments.

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

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

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

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

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

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

5.3.4 Searching a List

Sometimes, one needs to determine whether or not a particular value is in a list. In other situations, it might be necessary to determine the index of a value that is already known to be present. Python's `in` operator and `index` method allow these tasks to be performed.

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

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

⁴ A list can only be sorted if all of its elements can be compared to one another with the less than operator. The less than operator is defined for many Python types including integers, floating-point numbers, strings, and lists, among others.

programmers sometimes use the `in` operator to determine whether or not a value is present in a list, and then use the `index` method to ascertain its location.

Consider the following example. It begins by reading integers from the user and storing them in a list. Then one additional integer is read from the user. If it is present in the list, then the position of its first occurrence is reported. Otherwise, a message is displayed which reports its absence.

```
# Read integers from the user and store them in data until a blank line is entered.  
data = []  
line = input("Enter an integer (blank line to finish): ")  
while line != "":  
    n = int(line)  
    data.append(n)  
  
line = input("Enter an integer (blank line to finish): ")  
  
# Read an additional integer from the user.  
x = int(input("Enter one additional integer: "))  
  
# Display the index of the first occurrence of x (if it is present in the list).  
if x in data:  
    print(f"The first {x} is at index {data.index(x)} .")  
else:  
    print(f"{x} is not in the list.")
```

5.4 Lists as Return Values and Arguments

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

Lists can also be passed as arguments to functions. Any lists passed to a function are included inside the parentheses following the function's name when it is called. Each argument, whether it is a list or a value of another type, is assigned to the corresponding parameter variable inside the function.

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

Updates performed on individual list elements (where the name of the list, followed by an index enclosed in square brackets, appears on the left side of an assignment operator) also modify both the parameter variable and the argument that was

provided when the function was called. However, assignments to the entire list (where only the name of the list appears to the left of the assignment operator) only impact the parameter variable. Such assignments do not impact the argument provided when the function was called.

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

5.5 Debugging

All of the errors that were discussed in earlier chapters can occur in programs that use lists, as can some new errors that didn't need to be considered previously. The new errors that you are most likely to encounter are discussed in the sections that follow.

5.5.1 Syntax Errors

Square brackets are used when a list is initially created, and when individual elements in the list are accessed or updated. Like the parentheses used in mathematical expressions and function calls, the square brackets used with lists must be balanced. When a close square bracket is missing, Python will report a syntax error that identifies the unbalanced open square bracket. Similarly, Python's error message highlights the unmatched close square bracket when an open square bracket is omitted, as shown below:

```
File "ch5-missing-open.py", line 1
    odd_digits = 1, 3, 5, 7, 9]
                           ^
SyntaxError: unmatched '['
```

When a list is created, its initial elements are enclosed in square brackets and separated by commas. Failing to include the commas between the elements is a syntax error. The error message reported by Python highlights the location of the missing comma, and indicates that a comma may be needed, making this a particularly easy error to identify and correct.

5.5.2 Runtime Errors

An `IndexError` is perhaps the most common error encountered when working with lists. This runtime error occurs when the program attempts to read or modify a list element with an index that is greater than or equal to the length of the list.⁵

There are two possible causes for this error: either the list contains fewer elements than it was expected to, or the index was not computed correctly. The error message reported by Python will identify the line where the `IndexError` occurred, but Python is unable to provide any information about the underlying cause of the error. As a result, a helpful first step when debugging this error is the addition of two print statements immediately ahead of the line on which the `IndexError` occurred: one that displays the value of the index that was accessed, and one that displays the values in the list (or perhaps just the length of the list if it is expected to have a large number of elements). Displaying these values will help the programmer determine whether the problem is with the number of elements in the list, or the index that was computed. Knowing that will help the programmer continue their search for the error in the correct location.

Another runtime error that programmers sometimes encounter when working with lists is a `TypeError` where the additional information is ‘list’ object is not callable. This message is reported when one attempts to use parentheses to access an element in a list instead of square brackets. The error is corrected by using the proper type of brackets.

5.5.3 Logic Errors

A program that attempts to access an index beyond the end of a list will crash with an `IndexError`. However, such an error is not necessarily reported when a program attempts to access a negative list index. When a negative list index is used, Python accesses an element in the list starting from its end, rather than its beginning, with `-1` representing the last element in the list. Similarly, `-2` represents the second-to-last element, and `-3` represents the third-to-last element. A negative list index will only result in an `IndexError` if it attempts to access an element beyond the beginning of the list.

While negative list indices can be a convenient shorthand for accessing elements near the end of a list, this feature can obscure an incorrect index calculation that results in a negative value. Displaying the list index that is being accessed or updated can help identify this kind of error, as can adding assertions (which ensure that the list indices are greater than or equal to zero) to programs that are not designed to make use of negative list indices.

⁵ Recall that the last valid index for a list is the length of the list, minus one, because list indices begin at zero.

Passing a list as an argument to a function was discussed in Sect. 5.4. In that section, it was noted that some changes made to a parameter variable holding a list modify both the value of the parameter variable and the function’s argument, while other changes only update the parameter variable. Failing to understand which changes impact only the parameter variable, and which changes impact both the parameter variable and the function’s argument, can cause the program to compute and display incorrect results. One can determine whether or not the function is inadvertently modifying the list by displaying the list passed to the function immediately before and after the function call. Similarly, such print statements will reveal cases where the programmer expected the function’s argument to change, but it did not.

5.6 Exercises

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

Exercise 123: Sorted Order

(Solved, 22 Lines)

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

Exercise 124: Reverse Order

(20 Lines)

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

Exercise 125: Remove Outliers

(Solved, 46 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 a non-negative integer, n , as its parameters. The function should create a new copy of the list with the n largest elements and the n smallest elements removed. Then it should return the new copy of the list as the function's only result. The order of the elements in the returned list does not have to match the order of the elements in the original list.

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

Exercise 126: Avoiding Duplicates

(Solved, 21 Lines)

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

```
first  
second  
first  
third  
second
```

then your program should display:

```
first  
second  
third
```

Exercise 127: Negatives, Zeros and Positives

(Solved, 29 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 128: 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 it. 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. Include a main program that demonstrates the function by reading a value from the user and displaying its proper divisors. Ensure that your main program only runs when your solution has not been imported into another file.

Exercise 129: Perfect Numbers

(Solved, 35 Lines)

An integer, n , is said to be *perfect* when the sum of its proper divisors 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 128 when completing this task.

Exercise 130: Only the Words

(38 Lines)

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

Write a main program that demonstrates your function. It should read a string from the user and then display all of the words in the string with the punctuation marks removed. You will need to import your solution to this exercise when completing Exercises 131 and 188. As a result, you should ensure that your main program only runs when your file has not been imported into another program.

Exercise 131: Word by Word Palindromes

(34 Lines)

Exercises 82 and 83 previously introduced the notion of a palindrome. Those exercises considered the characters in a string, possibly ignoring spacing and punctuation marks. While palindromes are most commonly considered character by character, the notion of a palindrome can be extended to larger units. For example, while the sentence “Is it crazy how saying sentences backwards creates backwards sentences saying how crazy it is?” isn’t a character-by-character palindrome, it is a palindrome when examined a word at a time (and when capitalization and punctuation are ignored). Other examples of word-by-word palindromes include “Herb the sage eats sage, the herb” and “Information school graduate seeks graduate school information”.

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

Exercise 132: 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 133: Formatting a List

(Solved, 46 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 with fewer than five elements, 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.

The Oxford comma is an additional comma that some writers choose to include in lists of three or more items. It appears immediately after the second-to-last item in the list, before “and”. Oxford commas have not been included in the examples above, but you are welcome to include them in the output of your function if you prefer that style. If you choose to include an Oxford comma, it should only be present when three or more items are being formatted. It is not appropriate to include a comma after the first item when the list contains only one or two items.

Exercise 134: Random Lottery Numbers

(Solved, 28 Lines)

In order to win the top prize in a particular lottery, one must match all 6 numbers on their 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 selected numbers do not contain any duplicates. Display the numbers in ascending order.

Exercise 135: 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, which suggests 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 of the letters at the beginning of the word, up to the first vowel (excluding y), are moved 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 136: Pig Latin Improved

(51 Lines)

Extend your solution to Exercise 135 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 137: Balanced Parentheses and Square Brackets

(Solved, 67 Lines)

In writing, mathematics and programming, one normally expects each open parenthesis or square bracket to be followed by its corresponding close glyph at a later point in the text. When parentheses and square brackets are nested, one normally expects the latest open glyph to be closed before any earlier open glyph is closed. Failing to close an open glyph, or closing the open glyphs in the wrong order, is an error, as is including more close glyphs than open glyphs. The algorithm below can be used to detect unbalanced parentheses and square brackets in a string, `s`.

Initialize *glyphs* and *indices* to empty lists

Initialize *i* to 0 and *error* to False

While *i* is less than the length of *s* and *error* is False **do**

If the character at position *i* in *s* is an open glyph **then**

 Add the glyph to the end of *glyphs*

 Add its position to the end of *indices*

If the character at position *i* in *s* is a close parenthesis **then**

If *glyphs* is empty, or its last element is not an open parenthesis **then**

 Set *error* to True

Else

 Remove the last element from both *glyphs* and *indices*

If the character at position *i* in *s* is a close square bracket **then**

If *glyphs* is empty, or

 its last element is not an open square bracket **then**

 Set *error* to True

Else

 Remove the last element from both *glyphs* and *indices*

 Increment *i*

When this algorithm completes, *error* is set to True if there is a close glyph that does not have a corresponding open glyph, and the position of the unmatched close glyph is *i* – 1. The string includes at least one open glyph that does not have a corresponding close glyph if the *glyphs* list is not empty when the algorithm completes. The index of the open glyph that needs to be closed first is the last value in *indices*.

Create a program that reads a string from the user and determines whether or not all of its open glyphs are correctly closed. If each open glyph has a corresponding close glyph, and every close glyph has a corresponding open glyph, then your program should report that the glyphs are balanced. Otherwise, your program should report the nature of the first imbalance detected and its location. The location should be reported by displaying the string on one line, and then printing a ^ character on the next line under the unbalanced glyph, similar to what Python does in some of its error messages.

Your program can be extended to handle braces in addition to parentheses and square brackets. Doing so is left as an optional extension for the interested reader. Fewer than 10 additional lines of code are needed to complete this task.

Exercise 138: Page Numbers in an Index

(54 Lines)

The index at the back of a book lists relevant pages for terms that appear in it. Some entries might consist of only a single page, while others may include many pages. When an entry with two or more pages includes consecutive page numbers, each collection of consecutive numbers can be collapsed into a range consisting of the first number, followed by a dash, followed by the last number. For example, if the pages for a particular term include 2, 4, 5, 54, 67, 68, 69, 70 and 101 then the index would list them as 2, 4-5, 54, 67-70, 101.

Create a function that takes a list of page numbers as its only parameter. It should return a list of formatted page numbers, where groups of consecutive page numbers have been collapsed into ranges, as its only result. Use either the `sort` function or the `sorted` method so that your function can handle an unsorted list of page numbers. Include a main program that demonstrates that your function works correctly. Your function may optionally call the function you wrote in Exercise 122 to further abbreviate any groups of consecutive page numbers, if you prefer that style.

Exercise 139: Line of Best Fit

(43 Lines)

A line of best fit is a straight line that best approximates a collection of n points. In this exercise, each point in the collection will have an x coordinate and a y coordinate. The symbols \bar{x} and \bar{y} represent the average x and y values 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 each point on its own line as an x coordinate, followed by a comma, followed by a y coordinate. Allow the user to continue entering points until a blank line is entered. Display the formula for the line of best fit in the form $y = mx + b$ by replacing m and b with the values calculated using the preceding formulas. For example, if the user inputs the points (1, 1), (2, 2.1) and (3, 2.9) then your program should display $y = 0.95x + 0.1$.

Exercise 140: Shuffling a Deck of Cards

(Solved, 49 Lines)

A standard deck of playing cards has 52 unique 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 2–10, 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 2–9 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 in a list. Return the list of cards as the function’s only result. Your function will not require any parameters.

Write a second function, named `shuffle`, that randomizes the order of the cards in a list. One technique that can be used to shuffle the cards is to visit each element in the list and swap it with another randomly selected element in the list. You must write your own loop for shuffling the cards. Do not import the `shuffle` function from the `random` module.

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

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

Exercise 141: 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 140 to create a main program that deals four hands of five cards each from a shuffled deck. Display all of the hands of cards, along with the cards remaining in the deck after the hands have been dealt.

Exercise 142: 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 uses your function to report whether or not it 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 only one element?

Exercise 143: Count the Elements

(Solved, 48 Lines)

Python's standard library includes a method named `count` that determines how many times a specific value occurs in a list. In this exercise, you will create a new function named `countRange`. It will count and return the number of elements within a list that are greater than or equal to some minimum value, and less than some maximum value. Your function will take three parameters: the list, the minimum value and the maximum value. It will return an integer result greater than or equal to 0. Include a main program that demonstrates your function by applying it to 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 144: Tokenizing a String

(Solved, 48 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 easier to work with than the original string because the original string may have irregular spacing. In some cases, substantial work is also required to determine where one token ends and the next one begins.

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

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

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

Exercise 145: Unary and Binary Operators

(Solved, 45 Lines)

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

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

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

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

Exercise 146: Infix to Postfix

(63 Lines)

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

Create a new empty list, *operators*

Create a new empty list, *postfix*

For each token in the infix expression

If the token is an integer **then**

 Append the token to *postfix*

If the token is an operator **then**

While *operators* is not empty and

 the last item in *operators* is not an open parenthesis and

 precedence(token) < precedence(last item in *operators*) **do**

 Remove the last item from *operators* and append it to *postfix*

 Append the token to *operators*

If the token is an open parenthesis **then**

 Append the token to *operators*

If the token is a close parenthesis **then**

While the last item in *operators* is not an open parenthesis **do**

 Remove the last item from *operators* and append it to *postfix*

 Remove the open parenthesis from *operators*

While *operators* is not the empty list **do**

 Remove the last item from *operators* and append it to *postfix*

Return *postfix* as the result of the algorithm

Use your solutions to Exercises 144 and 145 to tokenize a mathematical expression and identify any unary operators in it. Then use the algorithm above to transform the expression from infix form to postfix form. Your code that implements the preceding algorithm should reside in a function that takes a list of tokens representing an infix expression (with the unary operators marked) as its only parameter. It should

return a list of tokens representing the equivalent postfix expression as its only result. Include a main program that demonstrates your infix to postfix function by reading an expression from the user in infix form and displaying it in postfix form.

You may find your solutions to Exercises 109 and 110 helpful when completing this problem. While you should be able to use your solution to Exercise 109 without any modifications, your solution to Exercise 110 will need to be extended so that it returns the correct precedence for the unary operators. The unary operators should have higher precedence than multiplication and division, but lower precedence than exponentiation.

Exercise 147: Evaluate Postfix

(63 Lines)

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

Create a new empty list, *values*

For each token in the postfix expression

If the token is a number **then**

 Convert it to an integer and append it to *values*

Else if the token is a unary minus **then**

 Remove the last integer from *values*

 Negate the integer and append the result of the negation to *values*

Else if the token is a binary operator **then**

 Remove the last integer from *values* and call it *right*

 Remove the last integer from *values* and call it *left*

 Compute the result of applying the operator to *left* and *right*

 Append the result to *values*

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

Write a program that reads a mathematical expression in infix form from the user, converts it to postfix form, evaluates it, and displays its value. Use your solutions to Exercises 144, 145 and 146, along with the algorithm above, to solve this problem.

The algorithms provided in Exercises 146 and 147 do not perform any error checking. As a result, your programs may crash or generate incorrect results if you provide them with invalid input. These algorithms can be extended to detect and respond to invalid input in a reasonable manner. Doing so is left as an independent study exercise for the interested reader.

Exercise 148: Does a List Contain a Sublist?

(44 Lines)

A sublist is a list that is part of a larger list. It 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 of [1, 2, 3, 4] because 2 and 4 are not adjacent in the longer list. The empty list is a sublist of every 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 149: Generate All Sublists of a List

(Solved, 40 Lines)

Using the definition of a sublist from Exercise 148, 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 demonstrates your function by displaying all of the sublists of several different lists.

Exercise 150: 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 feature of 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 `range`. When a number is crossed out, it is known that it is not prime, but it still occupies space on the piece of paper, and must still be considered when identifying later prime numbers. As a result, you should **not** simulate crossing out a number by removing it from the list. Instead, you should replace 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.

Exercise 151: Normal Magic Squares

(Solved, 110 Lines)

A magic square is a table of values where the number of rows and columns is equal, and each row, column and diagonal sums to the same value. A table that meets these requirements can be constructed by placing the same value at every location in it, but such a magic square is not particularly interesting. A normal magic square is a magic square that only includes consecutive integers starting from one. There are many arrangements of consecutive integers that form a normal magic square when the table has three or more rows and columns.

Write a program that reads a table of integers from the user and reports whether or not the entered integers represent a normal magic square. If the values do not represent a normal magic square then your program should report whether or not it represents a magic square without the consecutive integers constraint. The user will provide each row in the table as a single line of input, with the values for the row separated by commas. Once your program reads the first row, it should use the number of values in that row to determine how many additional rows of input should be read. Your program should include appropriate error checking. In particular, it should display a meaningful error message and exit if any of the subsequent lines of input entered by the user do not contain the same number of values as the first line.

There are no arrangements of the integers 1–4 that can be placed in a two-by-two table to form a normal magic square. This can be demonstrated by considering all possible arrangements of those integers within the table and showing that none of them have rows, columns and diagonals that all sum to the same value. A one-by-one table containing only the integer 1 is a normal magic square, though perhaps not a particularly interesting one.

Exercise 152: Creating a Normal Magic Square

(43 Lines)

Several techniques have been developed for creating normal magic squares. The Siamese method will be considered in this exercise. This technique builds an n by n normal magic square in a reasonably straightforward manner, but it only works for odd values of n . Its general approach is to place the values from 1 to n^2 into the table in sequence. The first value is placed in the center of the top row. After each value is placed, the location for the next value is determined, with the movement generally progressing up and to the right (wrapping around the edges of the board when they are encountered). The complete algorithm is shown below.

Initialize all of the elements in an n -by- n table to 0

Initialize row and col so that they refer to the middle element in the top row

Initialize num to 1

While num is less than or equal to n^2 **do**

 Store num into the table at the position indicated by row and col

 Increment num

 Compute $next_row$ by moving up one row, wrapping around to the bottom of the board if row is currently the top row

 Compute $next_col$ by moving one column to the right, wrapping around to the left edge of the board if col is currently the rightmost column

If the value in the table at new_row and new_col is not 0 **then**

 Set row to one row below its current value, wrapping around to the top of the board if row is currently the bottom row

Else

 Set row equal to $next_row$ and set col equal to $next_col$

Create a program that uses the Siamese method to create a magic square. Your program will begin by reading the value of n from the user. Then it will use the provided algorithm to create the magic square and display it. Your program should only attempt to populate the table if the value entered by the user is an odd positive integer. Otherwise, your program should display an appropriate error message and quit.



Dictionaries

6

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

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

Starting in Python 3.7, the key-value pairs in a dictionary are always stored in the order in which they were added to it.¹ There is no mechanism for inserting a key-value pair into the middle of an existing dictionary. Key-value pairs can be removed from a dictionary. Removing a key-value pair from a dictionary does not change the order of the remaining key-value pairs in it.

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

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

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

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

6.1 Accessing, Modifying and Adding Values

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

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

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

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

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

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

When this program executes, it creates a dictionary named `results` that initially has two keys: `pass` and `fail`. The value associated with each key is 0. A third key,

`withdrawal`, is added to the dictionary with the value 1 using an assignment statement. Then the value associated with `pass` is updated to 3 using a second assignment statement. The line that follows retrieves the current value associated with `fail`, which is 0, adds 1 to it, and then stores this new value back into the dictionary, replacing the previous value. When the values are printed, 1 (the value currently associated with `fail`) is displayed on the first line, 3 (the value currently associated with `pass`) is displayed on the second line, and 1 (the value currently associated with `withdrawal`) is displayed on the third line.

6.2 Removing a Key-Value Pair

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

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

6.3 Additional Dictionary Operations

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

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

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

The `in` operator is used together with the `values` method to determine whether or not a value is present in a dictionary. The value being searched for appears to

the left of the `in` operator, and a dictionary, with the `values` method applied to it, appears to its right. For example, the following code segment determines whether or not any of the values in dictionary `d` are equal to the value stored in `x`.

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

6.4 Loops and Dictionaries

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

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

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

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

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

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

```
# Compute the sum of all the values in the dictionary.  
total = 0  
for v in constants.values():  
    total = total + v  
  
# Display the total.  
print(f"The total is {total}.")
```

Some problems involving dictionaries are better solved with `while` loops than `for` loops. For example, the following program uses a `while` loop to read strings from the user until 5 unique values have been entered. Then all of the strings are displayed with their counts.

```
# Store the number of times each string is entered by the user.  
counts = {}  
  
# Loop until 5 distinct strings have been entered.  
while len(counts) < 5:  
    # Read a string from the user.  
    s = input("Enter a string: ")  
  
    # If s is already a key in the dictionary, then increase its count by 1. Otherwise add s to the  
    # dictionary with a count of 1.  
    if s in counts:  
        counts[s] = counts[s] + 1  
    else:  
        counts[s] = 1  
  
    # Display all of the strings and their counts.  
for k in counts:  
    print(f"{k} occurred {counts[k]} times.")
```

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

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

6.5 Dictionaries as Arguments and Return Values

Dictionaries can be passed as arguments to functions, just like values of other types. As with lists, a change made to a parameter variable that contains a dictionary can modify both the parameter variable and the argument passed to the function. For

example, inserting or deleting a key-value pair will modify both the parameter variable and the argument, as will modifying the value associated with one key in the dictionary using an assignment statement. However, an assignment to the entire dictionary (where only the name of the variable holding the dictionary appears to the left of the assignment operator) only impacts the parameter variable. It does not modify the argument passed to the function. Like other types, dictionaries are returned from a function using the `return` keyword.

6.6 Debugging

While there are many parallels that can be drawn between lists and dictionaries, there also several notable differences. These differences must be carefully considered when locating and correcting errors in programs that make use of dictionaries.

6.6.1 Syntax Errors

Braces are used when a dictionary is first created, but square brackets are used when a new key-value pair is added to the dictionary. Square brackets are also used when the value associated with a key is accessed or updated. Using square brackets instead of braces when a dictionary is created is a syntax error, as is using braces instead of square brackets when a value is accessed or updated at a later time. In both cases, Python correctly reports the line where the error is located, but only a generic invalid syntax message is displayed, leaving the programmer to recognize that the wrong kind of brackets were used. For example, the error message that is displayed when square brackets are used instead of braces during the creation of a dictionary is shown below.

```
File "ch6-syntax.py", line 1
    constants = [{"pi": 3.14, "e": 2.71, "root 2": 1.41}]
                                         ^
SyntaxError: invalid syntax
```

6.6.2 Runtime Errors

Attempting to retrieve the value associated with a key that isn't in the dictionary will cause the program to terminate with a `KeyError`. When this error occurs, Python displays a helpful error message that identifies the portion of the line where the invalid key was used. The error message also displays the key that wasn't in the dictionary. Examining the key allows the programmer to determine whether or not the intended key was accessed. If the correct key is displayed, then the programmer will have to determine why it is not present in the dictionary. If an incorrect key is displayed, then the focus will be on how that key was computed.

Another runtime error that can be encountered when working with dictionaries is a `TypeError` where the explanation provided by Python is ‘`set`’ object is not subscriptable. This error occurs when one inadvertently uses commas, instead of colons, to separate the keys and values when a dictionary is first created. While one might expect this to be reported as a syntax error, it is not, because sets (which are not discussed in this book) can be created by enclosing a collection of values in braces and separating them with commas.

Unfortunately, the runtime error reported by Python in this situation neither identifies the line that needs to be corrected, nor the steps needed to correct the error. The error will be reported for a line which attempts to access or update a value in the dictionary, but the correction is needed on the line where the programmer attempted to create the dictionary with keys and values separated by commas instead of colons. Awareness of this will, hopefully, allow you to track down and correct this error quickly if you make this mistake at some point in the future.

The final runtime error that will be considered in this section is a `NameError`. This error occurs when a dictionary has keys that are strings, but the programmer fails to enclose the string’s characters in double quotes when accessing or updating the value associated with a key. Without the double quotes, the characters are treated as a variable name by Python, and a `NameError` is reported if that variable does not exist. This error is corrected by inserting the missing double quotes.

6.6.3 Logic Errors

When a value is assigned to a key that isn’t present in a dictionary, that key will be added to the dictionary with the provided value. This makes adding a new key-value pair to a dictionary convenient, but can also result in a new pair being added to the dictionary due to a typo or incorrectly computed key. A programmer can detect that a key-value pair has incorrectly been added to the dictionary by displaying either the number of key-value pairs, or the entire dictionary. Assertions can also be added to the program to ensure that the number of key-value pairs is unchanged in functions that are not supposed to add new keys to the dictionary.

6.7 Exercises

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

Exercise 153: Reverse Lookup

(Solved, 46 Lines)

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

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

Exercise 154: Two Dice Simulation

(Solved, 46 Lines)

Write 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.

Continue by writing 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 rolls performed. Your program should also display the percentage expected by probability theory for each total. Sample output is shown below.

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

Exercise 155: Text Messaging

(31 Lines)

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

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

Write a program that displays the key presses needed for a message entered by the user. Construct a dictionary that maps from each letter or symbol to the key presses needed to generate it. Then use the dictionary to create and display the presses needed for the user’s message. For example, if the user enters Hello, World! then your program should output 443355555666110966677755531111. Ensure that your program handles both uppercase and lowercase letters. Ignore any characters that aren’t listed in the table above, such as semicolons and parentheses.

Exercise 156: Morse Code

(23 Lines)

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

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

.... . .-.. . -.. --- . -- - -.. . -.- . -.. -..

Table 6.1 Morse code for letters and digits

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

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

Exercise 157: Postal Codes

(41 Lines)

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

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

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

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

Exercise 158: Write out Numbers in English

(69 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 argument 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 159: Unique Characters

(Solved, 17 Lines)

Create a program that counts 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 160: Anagrams

(Solved, 42 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 161: 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 160 so that it checks if two phrases are anagrams. Your program should ignore capitalization, punctuation marks and spacing when making the determination.

Exercise 162: Capital Quiz

(46 Lines)

Elementary school students are often asked to memorize the capital cities of the states or provinces of their country, or of various countries around the world. In this exercise, you will create a program that tests an individual’s knowledge of these capital cities. Begin by creating a dictionary where the keys are states, provinces or countries, and the values are capital cities. Then your program should create 10 questions by randomly selecting keys from the dictionary, and asking the user to identify the capital city associated with each key. Once the user has answered 10 questions, their score should be displayed. Ensure that your program keeps track of the questions it has already asked so that the quiz does not include any repeated questions.

Exercise 163: Converting from Hexadecimal to Binary

(Solved, 33 Lines)

In Exercise 118, numbers were converted from one base to another by converting the number from the original base to base 10, and then from base 10 to the desired base. That process achieves the desired goal, but it involves quite a bit of multiplication and division. As such, it can be a little bit tedious to complete, especially when performing the calculations by hand.

In some cases, it is possible to convert directly from one base to another, without using base 10 as an intermediate step. This kind of direct conversion is performed using a lookup table instead of multiplication and division, which can make it faster and easier to perform by hand. However, this direct conversion approach can only be used in limited circumstances. Specifically, it can only be performed when one of the bases involved is an integer power of the other. For example, one can perform

a direct conversion from hexadecimal to binary because $2^4 = 16$, but one cannot perform a direct conversion from binary to decimal because $2^{3.3219} = 10$.

Create a program that converts a number directly from hexadecimal to binary. Begin by constructing a dictionary that maps each hexadecimal digit to four binary digits (with leading zeros included as necessary). Then convert the hexadecimal number to binary by converting each of its digits in sequence. Your program should display an appropriate error message and exit if the string entered by the user contains any characters that are not valid hexadecimal digits.

Exercise 164: Scrabble™ Score

(Solved, 22 Lines)

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

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

Write a program that computes and displays the Scrabble™ score for a word. Create a dictionary that maps from letters to point values. Then use the dictionary to look up the value for each letter in a string entered by the user, and compute its score.

A Scrabble™ board includes some spaces that multiply the value of a letter or the value of an entire word. These spaces will be ignored in this exercise.

Exercise 165: Birthstones (Again)

(Solved, 44 Lines)

A program that displayed the birthstone(s) associated with each month was developed in Exercise 49. That process will be reversed in this exercise. Create a program that reads a birthstone from the user, uses a dictionary to determine what month is associated with the entered value, and displays the result. Continue reading birthstones

and displaying results until the user enters a blank line. If the user provides an input value that is not a valid birthstone, then your program should display an appropriate error message before going on to read another input value. Ensure that your program works correctly for months that have multiple birthstones. For example, November should be displayed if the user enters either topaz or citrine.

Exercise 166: Create a Bingo Card

(Solved, 58 Lines)

A bingo card consists of 5 columns of 5 numbers which 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. A bingo card does not contain any repeated values.

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 lists, each of which will hold five numbers. 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. The free space won’t be considered in this exercise.

Exercise 167: Checking for a Winning Card

(102 Lines)

A winning bingo card contains a line of 5 numbers that have all been called. Players normally record the numbers that have been called by crossing them out or marking them with a bingo dauber. In this exercise, numbers will be marked by replacing them 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 to indicate 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 166 when completing this exercise.

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

Exercise 168: 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 random bingo card and a list of all of the valid bingo calls (B1 through O75). Once that list has been created, you can randomize the order of its elements by either using the `shuffle` function you wrote when completing Exercise 140, or by calling the `shuffle` function in the `random` module. Then your program should consume calls from the list and cross out matching numbers on the card until the card contains a winning line. Simulate 1,000 games, and report the minimum, maximum and average number of calls that were made before the card won. You may find it helpful to import your solutions to Exercises 166 and 167 when completing this exercise.

Exercise 169: Resistor Color Bands

(46 Lines)

Resistors are an important part of many electronic circuits. They impede the flow of electricity, with larger values indicating that it is more difficult for electricity to pass through them. When resistors are created, they are often printed with four color bands. The first three indicate the amount of resistance, and the fourth (which will be ignored in this exercise) indicates the tolerance of the resistor. The amount of resistance is calculated by using the values associated with the first two bands to form a two-digit number, and then multiplying that number by the factor indicated by the third band. The value and multiplier associated with each color is shown in Table 6.2.

Write a program that reads three colors from the user. The first two colors will be values, and the third will be the multiplier. Once these values have been read, your program should display the resistance of a resistor baring those colors. For example, if the user enters red, green, and orange, then your program should display 25,000 Ohms because the first two bands, red and green, represent 25, and the orange band indicates that this value must be multiplied by 1,000.

Use one or more dictionaries to look up the value and multiplier associated with each color, rather than using a large collection of `if` statements. Display an appro-

Table 6.2 Values and multipliers for resistor color bands

Color	Value	Multiplier
Black	0	x1
Brown	1	x10
Red	2	x100
Orange	3	x1,000
Yellow	4	x10,000
Green	5	x100,000
Blue	6	x1,000,000
Violet	7	x10,000,000
Gray	8	x100,000,000
White	9	x1,000,000,000

priate error message and exit if any value read from the user is not a valid color. The output from your program should include comma separators in large values so that they are easier to read.



Files and Exceptions

7

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

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

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

Like text files, binary files also contain sequences of bits. But unlike text files, those sequences of bits can represent any kind of data—they are not restricted to characters alone. Files that contain images, sounds, and videos are normally binary files. Only text files are considered in this book because they are easy to create and view with a text editor, but most of the concepts apply to binary files as well.

7.1 Opening a File

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

The `open` function takes two arguments. The first argument is a string containing the name of the file that will be opened. The second argument is also a string. It indicates what *access mode* will be used when the file is opened. The access modes that will be discussed in this book include `read` (denoted by "`r`"), `write` (denoted by "`w`") and `append` (denoted by "`a`").

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

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

Once a file has been opened, methods can be applied to its file object to read data from it. Similarly, data is written to the file by applying appropriate methods to the file object, or by calling the `print` function with appropriate arguments. These operations are described in the sections that follow. Once all of the values have been read or written, the file should be closed. This is accomplished by applying the `close` method to the file object.

7.2 Reading Input from a File

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

The `readline` method reads one line from a file and returns it as a string, much like the `input` function reads one line of text typed on the keyboard. Each subsequent call to `readline` reads another line from the file. The lines are read sequentially, from the beginning of the file to its end. The `readline` method returns an empty string once the end of the file has been reached.

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

```
# Read the file name from the user and open the file.  
fname = input("Enter the file name: ")  
inf = open(fname, "r")  
  
# Initialize the variable that will store the total of all the values.  
total = 0
```

```
# Read the first line from the file.  
line = inf.readline()  
  
# Continue looping until the end of the file is reached.  
while line != "":  
    # Add the value that was read most recently to the total.  
    total = total + float(line)  
  
    # Read the next line from the file.  
    line = inf.readline()  
  
# Close the file.  
inf.close()  
  
# Display the result.  
print(f"The total of the values in {fname} is {total}.")
```

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

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

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

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

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

```
# Close the file.  
inf.close()  
  
# Total the values that were read from the file.  
total = 0  
for line in lines:  
    total = total + float(line)  
  
# Display the result.  
print(f"The total of the values in {fname} is {total}.")
```

7.3 End of Line Characters

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

```
# Read the file name from the user and open the file.  
fname = input("Enter the name of a file to display: ")  
inf = open(fname, "r")  
  
# Initialize the line number.  
num = 1  
  
# Display each line in the file, preceded by its line number.  
line = inf.readline()  
while line != "":  
    print(f"{num}: {line}")  
  
    # Increment the line number and read the next line.  
    num = num + 1  
    line = inf.readline()  
  
# Close the file.  
inf.close()
```

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

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

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

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

```
# Read the file name from the user and open the file.  
fname = input("Enter the name of a file to display: ")  
inf = open(fname, "r")  
  
# Initialize the line number.  
num = 1  
  
# Display each line in the file, preceded by its line number.  
line = inf.readline()  
while line != "":  
    # Remove the end of line marker and display the line preceded by its line number.  
    line = line.rstrip()  
    print(f"{num}: {line}")  
  
    # Increment the line number and read the next line.  
    num = num + 1  
    line = inf.readline()  
  
# Close the file.  
inf.close()
```

7.4 Writing Output to a File

When a file is opened in write mode, a new empty file is created. If the file already exists, then the existing file is destroyed, and any data that it contained is lost. Opening a file that already exists in append mode will add new data to the end of it, without modifying the existing values. If a file opened in append mode does not exist, then a new empty file is created.

The `write` method can be used to store data in a file opened in either write mode or append mode. It takes one argument, which must be a string, that will be written to the file. Values of other types can be converted to a string by calling the `str` function, or by using an f-string to format the value. Multiple values can be written to the file by concatenating all of the items into one long string, by using an f-string to format several values in one string, by calling the `print` function with appropriate arguments, or by calling the `write` method multiple times.

Unlike the `print` function, the `write` method does not automatically move to the next line after writing a value. As a result, one has to explicitly write an end of

line marker to the file between values that are to reside on different lines. Python uses \n to denote the end of a line. This pair of characters, referred to as an *escape sequence*, can appear in a string on its own, or \n can be part of a longer string.

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

```
# Read the file name from the user and open the file.  
fname = input("Where will the numbers be stored? ")  
outf = open(fname, "w")  
  
# Read the maximum value that will be written.  
limit = int(input("What is the maximum value? "))  
  
# Write the numbers to the file with one number on each line.  
for num in range(1, limit + 1):  
    outf.write(str(num) + "\n")  
  
# Close the file.  
outf.close()
```

It is also possible to store values in a file using the `print` function. For example, the call to `write` in the previous example could be replaced with the call to `print` shown below. It stores `num` in the output file instead of displaying it on the screen, because `file=outf` passes the file object that was opened for writing to the `print` function's optional `file` parameter.

```
print(f"{num}", file=outf)
```

7.5 Command Line Arguments

Computer programs are commonly executed by clicking on an icon or selecting an item from a menu. Programs can also be started by typing a command into a terminal or command prompt window. For example, on many operating systems, the Python program stored in `test.py` can be executed by typing either `test.py` or `python test.py` in such a window.

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

Any command line arguments provided when the program was executed are stored into a variable named `argv` (argument vector) that resides in the `sys` (system) module. This variable holds a list, and every element in that list is a string. The

first element is the name of the Python file that is being executed. The subsequent elements are the values that followed the name of the Python file on the command line (if any).

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

```
# The system module must be imported to access the command line arguments.  
import sys  
  
# Display the number of command line arguments (including the .py file).  
print(f"The program has {len(sys.argv)}",  
      "command line argument(s).")  
  
# Display the name of the .py file.  
print(f"The name of the .py file is {sys.argv[0]}")  
  
# Determine whether or not there are additional arguments to display.  
if len(sys.argv) > 1:  
    # Display all of the command line arguments beyond the name of the .py file.  
    print("The remaining arguments are:")  
    for i in range(1, len(sys.argv)):  
        print(f" {sys.argv[i]}")  
else:  
    print("No additional arguments were provided.")
```

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

```
# Import the system module.  
import sys  
  
# Ensure that the program was started with one command line argument beyond the name of the  
#.py file.  
if len(sys.argv) != 2:  
    print("One file name must be provided as a command line",  
          "argument.")  
    quit()  
  
# Open the file listed immediately after the .py file on the command line.  
inf = open(sys.argv[1], "r")
```

```
# Initialize the variable that will store the total of all the values.  
total = 0  
  
# Total the values in the file.  
line = inf.readline()  
while line != "":  
    total = total + float(line)  
    line = inf.readline()  
  
# Close the file.  
inf.close()  
  
# Display the result.  
print(f"The total of the values in {sys.argv[1]} is {total}.")
```

7.6 Exceptions

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

The programmer must indicate where an exception can occur in order to catch it. They must also provide the code that will run to handle the exception when it occurs. This is accomplished using two keywords: `try` and `except`. Code that might cause an exception that the programmer wants to catch is placed inside a `try` block. The `try` block is immediately followed by one or more `except` blocks. When an exception occurs inside a `try` block, execution immediately jumps to the appropriate `except` block, without running any remaining statements in the `try` block.

The type of exception that an `except` block catches immediately follows the `except` keyword. An `except` block that does not specify a particular type of exception will catch any exception (that is not caught by another `except` block associated with the same `try` block). An `except` block only executes when an exception occurs. If the `try` block completes without raising an exception, then all of the `except` blocks are skipped, and execution continues with the first line of code following the final `except` block.

The programs that appeared earlier in this chapter crashed when the user provided the name of a file that did not exist, because a `FileNotFoundException` exception was raised without being caught. A `try` block and an `except` block can be used to catch this exception and display a meaningful error message, instead of crashing the

program, as shown below. This code segment can be followed by whatever additional code is needed to read and process the data in the file.

```
# Read the file name from the user.  
fname = input("Enter the file name: ")  
  
# Attempt to open the file.  
try:  
    inf = open(fname, "r")  
except FileNotFoundError:  
    # Display an error message and quit if the file was not opened successfully.  
    print(f'{fname}' ' could not be opened. Quitting...')  
    quit()
```

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

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

When this program runs, it begins by reading the name of a file from the user. Then the `file_opened` variable is set to `False`, and the loop runs for the first time. Two lines of code reside in the `try` block inside the loop's body. The first attempts to open the file specified by the user. If the file does not exist, then a `FileNotFoundError` exception is raised, and execution immediately jumps to the `except` block, skipping the second line in the `try` block. When the `except` block executes, it displays an error message and reads another file name from the user.

Execution continues by returning to the top of the loop and evaluating its condition again. The condition still evaluates to `False` because the `file_opened` variable is still `False`; the line of code that sets it to `True` was skipped when the exception occurred. As a result, the body of the loop executes for a second time, and the program attempts to open the file again using the most recently entered file name. If that file

does not exist, then the program progresses as described in the previous paragraph. But if the file exists, the call to `open` completes successfully, and execution continues with the next line in the `try` block. This line sets `file_opened` to `True`. Then the `except` block is skipped because no exceptions were raised while executing the `try` block. Finally, the loop terminates because `file_opened` was set to `True`, and execution continues with the rest of the program.

The concepts introduced in this section can be used to catch exceptions beyond `FileNotFoundException`. Some other exceptions that you will need to catch as you complete the exercises in this section include `ValueError` (which is raised when a program fails to convert a string to an integer or floating-point number), and `IOError` (which is raised when a program encounters a problem while reading or writing a file). More generally, `try` and `except` blocks can be used to catch any runtime errors, and respond to them in a meaningful way, instead of allowing them to crash your programs.

7.7 Debugging

Programmers can use `try` and `except` blocks to catch and respond to runtime errors, but accessing files, and working with exceptions, also provides new opportunities for programmers to introduce errors into their programs. Some common syntax, runtime and logic errors that you might encounter when working with files and exceptions are described in the sections that follow.

7.7.1 Syntax Errors

There are several syntax errors that can occur when working with exceptions. Each `try` keyword must be immediately followed by a colon, and each `except` keyword must either be immediately followed by a colon, or followed by the type of exception being caught and a colon. Failing to include the colon is a syntax error, as is failing to indent at least one line of code so that it resides inside the `try` or `except` block. In each of these cases, Python reports a meaningful error message which correctly identifies the source of the error and the location that needs to be corrected.

Each `try` block must be followed by at least one `except` block.² Failing to include an `except` block is a syntax error. When the `except` block is omitted, an error message is provided that indicates that an `except` block is missing. The error message also identifies the location where the `except` block needs to be inserted.

² It is also possible to follow a `try` block with a `finally` block, in addition to (or instead of) an `except` block. The use of `finally` blocks is beyond the scope of this book.

7.7.2 Runtime Errors

Some runtime errors occur because of actions taken by the user. These errors, such as a `FileNotFoundException`, can be caught by including appropriate `try` and `except` blocks in the program. In other cases, a runtime error occurs because the programmer made a mistake when the program was created. Those runtime errors need to be corrected by modifying the program.

The file mode passed to the `open` function must be consistent with the operations that will subsequently be performed on the file object. Attempting to write to a file that has been opened for reading will result in an `io.UnsupportedOperation` error, as will attempting to read from a file that has been opened in write mode or append mode. Python's error message identifies the line in the program that attempted to perform the invalid operation. If the operation was applied to the wrong file object, then this will be the line that needs to be corrected. However, if the file was opened with the wrong mode, then the programmer will need to locate the statement that opened the file and correct it.

Files are opened by calling the `open` function, but files are closed by calling the `close` method. This asymmetry can result in programmers inadvertently attempting to close a file by calling the `close` function instead of the method. Doing so will result in a `NameError` because Python does not have a `close` function. The error message reported by Python identifies the line that failed to properly close the file. This error is corrected by applying the `close` method to the file object, instead of calling the `close` function.

Mistyping the name of an exception at the top of an `except` block is also a runtime error. Such an error can go undetected for a long period of time, because Python doesn't validate `except` blocks' errors until an exception is raised and a matching `except` block is needed. When the exception is not valid, the program crashes with a `NameError`. This error is corrected by updating the name of the exception.

7.7.3 Logic Errors

Many programs read values from a file using a loop. Such programs normally open the file before the loop, read one line from the file each time the loop's body executes, and close the file after the loop terminates. This approach works well, but there are two common errors that can result in an infinite loop: failing to read the next line from the file within the loop's body, and opening the file inside the loop.

Failing to read the next line from the file within a loop's body often causes an infinite loop, because the loop's condition includes the value read from the file. For example, it is common to loop until the value read from the file is the empty string. If the first line is read from the file before the loop begins to execute, but the next line is not read from the file inside the loop's body, then the program will loop infinitely and repeatedly process the first line from the file. This error is corrected by ensuring that a line is read from the file each time the loop's body executes.

Opening the file inside the loop’s body can cause an infinite loop because reading begins at the top of the file each time it is opened. As a result, while a line is read from the file each time the loop body executes, it is always the first line in the file. This error is corrected by ensuring that the file is opened before the loop, not inside it.

Using an `except` block without specifying what type of exception will be caught is a convenient way to catch and respond to any error that occurs within a `try` block. While this may be a reasonable approach when there is a `try` block that is expected to raise several exceptions that all need to be handled in the same way, it can also obscure errors in the program that need to be corrected. For example, such an `except` block will catch the `NameError` that occurs when a programmer mistypes the name of a variable, method or function, instead of crashing the program. Consequently, no error message is displayed to make the programmer aware of the problem, or provide the information needed to correct it. Similar problems arise for errors such as inadvertently providing incompatible operand types to an operator and list indices that are out of range. As a result, `except` blocks that do not catch a specific type of error should be used cautiously. In most circumstances, it is preferable to only catch the particular exceptions that are expected to occur. That way, any unexpected runtime errors are reported, along with the information needed to correct them.

7.8 Exercises

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

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

Exercise 170: Display the Head of a File

(Solved, 40 Lines)

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

Exercise 171: Display the Tail of a File*(Solved, 37 Lines)*

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

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

Exercise 172: Concatenate Multiple Files*(Solved, 28 Lines)*

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

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

Exercise 173: Number the Lines in a File*(23 Lines)*

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

Exercise 174: Find the Longest Word in a File

(39 Lines)

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

Exercise 175: Letter Frequencies

(43 Lines)

Frequency analysis can be used to help break some simple forms of encryption. In the simplest case, this technique examines the encrypted text to determine which characters are most common. Then it tries to map the most commonly occurring letters in English, such as E and T, to the most commonly occurring characters in the encrypted text.

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

Exercise 176: Most Frequently Occurring Word

(37 Lines)

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

Hint: You will probably find your solution to Exercise 130 helpful when completing this task.

Exercise 177: Sum a Collection of Numbers

(*Solved, 28 Lines*)

Create a program that sums all of the numbers entered by the user while ignoring any non-numeric input. Your program should display the current sum after each number is entered. If a non-numeric value is entered, then an appropriate message should be displayed before your program goes on and reads additional numbers. Exit the program when the user enters a blank line. Ensure that your program works correctly for both integers and floating-point numbers.

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

Exercise 178: Both Letter Grades and Grade Points

(*106 Lines*)

Write a program that converts from letter grades to grade points and vice versa. Your program should allow the user to convert multiple values, with one value entered on each line. Begin by attempting to convert each value entered by the user from a number of grade points to a letter grade. If an exception occurs during this process, then your program should attempt to convert the value from a letter grade to a number of grade points. If both conversions fail, then your program should output a message indicating that the supplied input is invalid. Design your program so that it continues performing conversions (or reporting errors) until the user enters a blank line. Your solutions to Exercises 57 and 58 may be helpful when completing this exercise.

Exercise 179: Remove Comments

(*Solved, 53 Lines*)

Python uses the # character to mark the beginning of a comment. The comment continues from the # character to the end of the line containing it. There is no mechanism for ending a comment before the end of a line.

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

Exercise 180: Four Word Random Password

(Solved, 47 Lines)

Generating a password by selecting random characters usually results in one that is relatively secure, but it also tends to create a password that is difficult to memorize. As an alternative, some systems construct a password by taking several English words and concatenating them. Such a password is normally easier to memorize while also being relatively secure.

Write a program that reads a file containing a list of words, randomly selects four of them, and concatenates them to produce a password. When producing the password, ensure that each word selected is between 3 and 7 characters so that the resulting password is between 12 and 28 characters. Capitalize each word in the password so that the user can easily see where one word ends and the next one begins. Finally, your program should display the password for the user.

The advantages and disadvantages of passwords constructed by selecting four words have been extensively analyzed, thanks (at least in part) to an XKCD comic about password strength which used the words correct, horse, battery and staple. Entering those four words into your favorite search engine should turn up the comic and numerous websites discussing the ideas presented in it.

Exercise 181: Weird Words

(67 Lines)

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

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

There are some words that both follow and violate this rule. For example, deified follows the rule because the I is before the E after the F, but it also violates the rule because the E is before the I after the D. This contradiction is not unique to deified. Additional examples of words that both follow and violate the rule include reified, veinier and weightier, among others.

Exercise 182: What's That Element Again?

(59 Lines)

Write a program that reads a file containing information about chemical elements and stores the information it contains 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 chemical element with the number of protons entered. If the user enters a non-integer value, then your program should display the number of protons for the chemical element with that name or symbol. Your program should display an appropriate error message if no chemical 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 183: A Book with No E...

(Solved, 50 Lines)

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

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

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

Exercise 184: Names That Reached Number One

(Solved, 54 Lines)

The baby names data set consists of over 200 files. Each file contains a list of 100 names, along with the number of times each name was used. Entries in the files are ordered from most frequently to least frequently used. There are two files for each year: one containing names used for girls, and the other containing names used for boys. The data set includes files 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 185: Gender-Neutral Names

(56 Lines)

Some names, like Ben, Jonathan and Andrew, are normally only used for boys, while other names, like Esther, Rebecca and Flora, are normally only used for girls. Other names, like Chris, Ryan and Alex, are commonly used for both boys and girls.

Write a program that determines and displays all of the baby names that were used for both girls and boys in a year specified by the user. Your program should display 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 184.

Exercise 186: Most Used Names in a Time Period

(76 Lines)

Write a program that uses the baby names data set, described in Exercise 184, 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 girl's name and the boy's name given to the most children during the indicated years.

Exercise 187: 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 184. As your program reads the files, it should keep track of every distinct name used for a boy and every distinct name used for a girl. Then your program should output each of these lists of names. Neither of the lists should contain any repeated values.

Exercise 188: Spell Checker

(Solved, 61 Lines)

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

The user will provide the name of the file to check for spelling mistakes as a command line argument. Your program should display an appropriate error message if the command line argument is missing. An error message should also be displayed if your program is unable to open the user's file. Use your solution to Exercise 130 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 the words data set into a list, determining whether or not a string is present in a long list is quite slow. 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 189: 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 and report this error when a spelling or grammar check is performed.

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

Exercise 190: Redacting Text in a File

(Solved, 52 Lines)

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

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

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

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

Exercise 191: Missing Comments

(Solved, 49 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 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 line preceding `def` 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 to analyze as command line arguments. An appropriate error message should be displayed for any files that do not exist or cannot be opened. Then your program should process the remaining files.

Exercise 192: Display a Spreadsheet

(71 Lines)

Some text files store multiple values on one line. When this occurs, it is necessary to denote where one value ends, and the next one begins. Commas are commonly used for this purpose. When they are, the file is referred to as a comma separated value (CSV) file.

CSV files are easily read and written by Python programs. There are also many spreadsheet packages that can import and export data in this format. As a result, using CSV files allows Python programs to output results that can be further analyzed or graphed using a spreadsheet package, and also allows data stored in a spreadsheet to be exported for further analysis or manipulation by a Python program.

Write a program that reads a CSV file representing a spreadsheet and displays it on the screen. The widths of the columns should be determined from the values that are read, with each column being wide enough to hold the longest value in it. Numeric values should be right aligned within each column. Non-numeric values should be left aligned. Leave two spaces between adjacent columns so that it is clear where one value ends, and the next one begins. Your program can either read the name of the file from the keyboard, or have it provided as a command line argument.

Exercise 193: Percentage Grades to Letter Grades

(Solved, 71 Lines)

At a particular academic institution, students are awarded percentage grades on all of the pieces of work that they complete during the term. At the end of the term, these grades are combined to compute each student's percentage grade for the course (rounded to the closest integer), and then that percentage grade is converted to a letter grade using the following table.

Percentage	Letter grade
90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

Create a program that reads a collection of percentage grades from a file and converts each of them to a letter grade. Each percentage grade will appear on its own line in the file. The new file, which will replace the original, will have lines that consist of the original percentage grade, followed by a comma, followed by the corresponding letter grade. The name of the file to update will be provided as a command line argument to the program. Report an appropriate error message and exit the program if the command line argument is omitted, or if the file cannot be opened. The program

should also display an error message and exit (without updating any of the values in the file) if any of the lines in the file are not an integer between 0 and 100.

Exercise 194: Consistent Line Lengths

(45 Lines)

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

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

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

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

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

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

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

Exercise 195: Words with Six Vowels in Order

(56 Lines)

There is at least one word in the English language that contains every vowel (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 the file.



Recursion

8

Many aspects of functions were explored in Chap. 4, including functions that call other functions. A closely related topic that was not considered at that time is whether or not a function can call itself. It turns out that this is, in fact, possible, and that it is a powerful technique for solving some problems.

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

Any function that calls itself is recursive because the function's body (its definition) includes a call to the function that is being defined. In order to reach a solution, a recursive function must have at least one case where it is able to produce the required result without calling itself. This is the *base case*. Cases where the function calls itself are *recursive cases*. These concepts will be explored in the sections that follow by considering three examples.

8.1 Summing Integers

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

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

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

```
## Compute the sum of the integers from 0 up to and including a non-negative integer, n, using
# recursion.
# @param n the maximum value to include in the sum (must be greater than or equal to 0)
# @return the sum of the integers from 0 up to and including n
def sum_to(n):
    if n <= 0:
        # Base case: The answer is 0.
        return 0
    else:
        # Recursive case: Add n to the sum of the integers from 0 to n-1.
        return n + sum_to(n - 1)

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

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

Executing the recursive call to `sum_to`, where n is equal to 1, results in another recursive call to `sum_to` where n is equal to 0. Once that recursive call begins to execute, there are three instances of `sum_to` executing with argument values 2, 1 and 0. The instance where n is equal to 2 is waiting for the instance where n is equal to 1 to complete before it can return its result, and the instance where n is equal to 1

is waiting for the instance where n is equal to 0 to complete before it can return its result. While the same function was called each time, each instance of the function is entirely separate from all of the other instances.

The base case executes when `sum_to` is called with n equal to 0. This causes 0 to be returned immediately, and the instance where n is 0 ceases to exist. Then the instance of `sum_to` where n is equal to 1 progresses by adding 1 to the 0 returned by the recursive call. The total, which is 1, is returned by the instance where n is equal to 1, and that instance ceases to exist. Execution continues with the instance of `sum_to` where n is equal to 2. It adds n , which is 2, to the 1 returned by the recursive call, and 3 is returned and stored in `total`. Finally, the total is displayed, and the program terminates.

8.2 Fibonacci Numbers

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

Numbers in the Fibonacci sequence, beyond the first two, can be computed using the formula $F_n = F_{n-1} + F_{n-2}$. This definition is recursive because a larger Fibonacci number is computed using two smaller Fibonacci numbers. The first two numbers in the sequence, F_0 and F_1 , are the base cases because they have known values that are not computed recursively.

A program that implements the recursive formula for computing Fibonacci numbers is shown below. It computes and displays the value of F_n for some value of n entered by the user.

```
## Compute the nth Fibonacci number using recursion.  
# @param n the index of the Fibonacci number to compute  
# @return the nth Fibonacci number  
def fib(n):  
    # Base cases: The first two numbers in the sequence are 0 and 1.  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
  
    # Recursive case: The current number is the sum of its 2 immediate predecessors.  
    return fib(n-1) + fib(n-2)  
  
# Compute the Fibonacci number requested by the user.  
n = int(input("Enter a non-negative integer: "))  
print(f"fib({n}) is {fib(n)}")
```

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

Based on the performance of the Fibonacci numbers program, you might be tempted to conclude that recursive solutions are too slow to be useful. While that is true in this particular situation, it is not true in general. The previous program that summed integers ran quickly, even for larger values, and there are some problems that have very efficient recursive algorithms, such as Euclid's algorithm for computing the greatest common divisor of two integers, which is described in Exercise 197.

Figure 8.1 illustrates the recursive calls made when computing F_4 and F_5 , as well as the recursive calls made to evaluate `sum_to(4)` and `sum_to(5)`. Comparing the function calls made to compute these results will help you better understand why these programs behave differently as the input value increases.

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

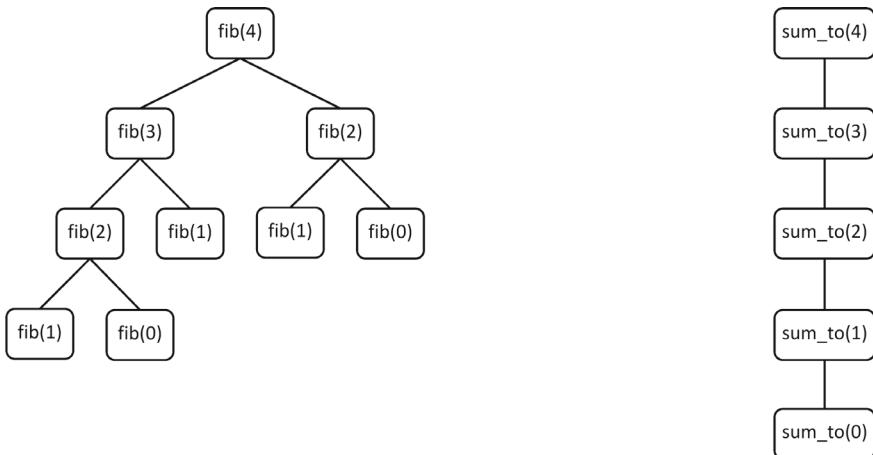
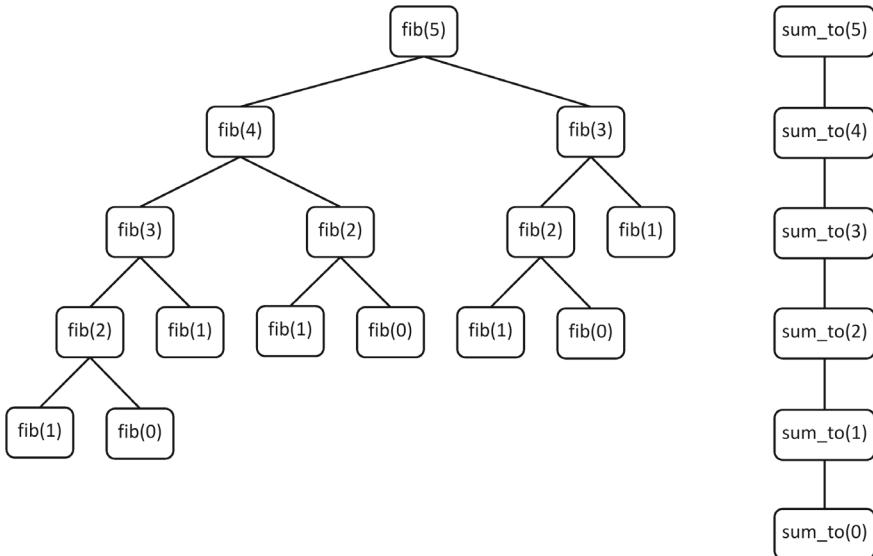
In contrast, when the argument passed to `fib` increases from 4 to 5, the number of function calls increases from 9 to 15. More generally, when the position of the Fibonacci number being computed increases by 1, the number of recursive calls (nearly) doubles. This is referred to as exponential growth.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

8.3 Counting Characters

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

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

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

(a) The Function Calls Used to Compute `fib(4)` and `sum_to(4)`(b) The Function Calls Used to Compute `fib(5)` and `sum_to(5)`**Fig. 8.1** A Comparison of the Function Calls for `fib` and `sum_to`

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

```
## Count the number of times a particular character is present in a string.  
# @param s the string in which the characters are counted  
# @param ch the character to count  
# @return the number of occurrences of ch in s  
def count(s, ch):  
    # Base case: s is the empty string so the number of occurrences is 0.  
    if s == "":  
        return 0  
  
    # Compute the tail of s.  
    tail = s[1 : len(s)]  
  
    # Recursive cases: Count the number of occurrences of ch in the tail of s, adding 1 only  
    # if the first character in s is the character being counted.  
    if s[0] == ch:  
        return 1 + count(tail, ch)  
    else:  
        return count(tail, ch)  
  
# Count the number of times a character occurs in a string.  
s = input("Enter a string: ")  
ch = input("Enter the character to count: ")  
print(f"{ch} occurs {count(s, ch)} times in '{s}'.")
```

When this program executes, it begins by reading the string and the character from the user. Then the `count` function is called for the first time. If `s` is not the empty string, then the tail of `s` is computed, and the first character in `s` is compared to `ch`. If those characters match, then one is added to the result computed recursively for the tail of `s`. Otherwise, the number of occurrences of `ch` in the tail of `s` is computed recursively and returned without modification. Each recursive call is performed on a shorter string until the base case is reached, and 0 is returned. When this occurs, all of the earlier recursive calls complete, and the number of occurrences of `ch` in `s` is returned by the first call to `count`.

8.4 Debugging

Syntax, runtime, and logic errors can all occur in programs that employ recursion. Examples of these errors, and strategies for resolving them, are discussed in the sections that follow.

8.4.1 Syntax Errors

A recursive function is defined in exactly the same manner as a function that does not call itself. Its definition begins with `def`, followed by the name of the function, an open parenthesis, an optional comma separated list of parameter variables, a close parenthesis, and a colon. These elements are followed by the lines of code which form the body of the function. Omitting any of the non-optional components of the

function definition is a syntax error which will be reported by Python, as is failing to indent the body of the function. Correcting these errors is normally relatively straightforward because the error message identifies the location of the error.

8.4.2 Runtime Errors

A well-formed recursive function must progress toward its base case as it executes. If the function fails to do this, it will call itself repeatedly, eventually causing the program to crash with a `RecursionError`. The error message highlights the location where the recursive calls were made, but it does not provide any information about how to correct the error. Displaying the values of the parameter variables at the beginning of the function's body will show what values were provided when the function was called. This usually reveals that one (or more) of the values is incorrect, and allows the programmer to focus their debugging efforts on the erroneous value.

A `RecursionError` can also occur when a function is making progress toward its base case, but the number of recursive calls needed to solve the problem is larger than Python's maximum recursion depth. On many systems, Python's default recursion limit is 1,000 calls. While this is sufficient in many situations, solving a particularly large problem can result in the limit being exceeded. The recursion limit can be viewed by printing the result returned by `sys.getrecursionlimit()`. It can be raised by calling `sys.setrecursionlimit()` with the new limit as its only argument. If displaying the function's parameter variables reveals that it was progressing toward the base case, as expected, before the `RecursionError` occurred, then raising the recursion limit may resolve the error.

8.4.3 Logic Errors

Problems are solved recursively by determining how one version of a problem can help solve another version of it. Typically, this means that a larger version of the problem is solved by using the solution to a smaller version of the problem, plus some additional work. This is not an intuitive way to solve problems for some (perhaps even many) people. As a result, logic errors in recursive functions are often the result of failing to correctly identify the relationship between the larger and smaller versions of the problem, or the additional work that needs to be combined with the smaller solution to solve the larger problem.

Unfortunately, there is no way to easily identify and correct the logic errors in a recursive function. Instead, one needs to carefully consider each step that is performed, the results that are calculated, and the values that are passed to each recursive call. Displaying the values of the parameter variables at the beginning of the function can be helpful, because it allows the programmer to identify any incorrect values that were passed to the function. Similarly, appropriate assertions can be used to ensure that the program crashes immediately if the function is passed a value outside of the

expected range. It can also be helpful to identify the smallest example of an incorrect result, so that the amount of output that needs to be analyzed is minimized.

8.5 Exercises

All of the exercises in this chapter should be solved by writing one or more recursive functions. Each of these functions will call itself, and may also make use of any of Python's features that were discussed in the previous chapters.

Exercise 196: Total the Values

(Solved, 29 Lines)

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

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

Exercise 197: Greatest Common Divisor

(24 Lines)

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

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

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

numbers consisting of hundreds of digits, because Euclid's algorithm is extremely efficient.

Exercise 198: Recursive Decimal to Binary

(34 Lines)

In Exercise 90, 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 that 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 199: The NATO Phonetic Alphabet

(33 Lines)

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

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

Exercise 200: Roman Numerals

(25 Lines)

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

Table 8.1 NATO phonetic alphabet

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

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

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

Exercise 201: Binary Search

(63 Lines)

Searching for a value in a list is a task that programs frequently perform. When the values in the list are unsorted, linear search is commonly employed. This approach searches for the desired value by checking each element in the list in sequence. If the desired element is located, then the search terminates, and its presence is reported. Otherwise, every element in the list is examined in order to conclude that the desired element is not present.

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

When the values in a list are sorted, a more efficient approach can be employed. Binary search begins by examining the middle element in the list. If the desired element is less than the middle element, then all of the elements beyond the midpoint are ignored, and the search is repeated on the elements between the beginning of the list and its midpoint. Similarly, if the desired element is greater than the middle element, then the elements between the beginning of the list and its midpoint are ignored, and the search is repeated on the elements after the list's midpoint.

Binary search is a recursive algorithm that repeatedly searches smaller and smaller lists. The recursion terminates when the middle element is the desired value, or the search is performed on an empty list, or the search is performed on a list of only one element. The algorithm returns `True` if the desired element was located in the list. Otherwise, `False` is returned.

Write a recursive function that performs a binary search on a sorted list of values. Your function will take four parameters, which are the list of values, the smallest index still under consideration, the largest index still under consideration, and the desired value. Your function will return `True` if the desired element is located. Otherwise, it will either return `False` because one of the other base cases has been reached, or it will call itself recursively on the appropriate half of the remaining elements. Include a main program that demonstrates that your binary search function behaves correctly.

Exercise 202: Recursive Palindrome

(Solved, 30 Lines)

The notion of a palindrome was introduced previously in Exercise 82. 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 and uses your recursive function to determine whether or not it is a palindrome. Then your program should report the result as part of a meaningful message.

Exercise 203: Exponentiation

(Solved, 53 Lines)

Exponentiation is the mathematical operation that raises one value to the power of another. When the power value is a positive integer, exponentiation can be performed recursively by making use of the fact that $x^n = x \times x^{n-1}$ and $x^0 = 1$. This approach will make n recursive calls to compute x^n .

Exponentiation can also be performed recursively using an approach known as exponentiation by squaring. This approach makes use of the fact that $x^n = (x \times x)^{n/2}$ when n is a positive even integer, and $x^n = x(x \times x)^{(n-1)/2}$ when n is a positive odd

integer. Exponentiation by squaring uses less than n recursive calls to compute x^n when n is greater than 2. The number of recursive calls is far less than n when n is large.

Create two recursive functions that implement the two approaches to exponentiation that were described in the previous paragraphs. Each function will take two parameters, x and n , and will return the value of x^n as its only result. Call `print` at the beginning of each function's body so that you can see how many recursive calls are performed when each function executes. Write a main program that shows this difference by computing 2^{100} using each of your functions.

Each of the functions described above can be extended to handle exponents that are negative integers by making use of the fact that $x^n = \frac{1}{x^{-n}}$. Consider using this knowledge to improve your functions.

Exercise 204: Recursive Square Root

(20 Lines)

Exercise 81 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 each time the loop's body executed. In this exercise, you will use the same approximation strategy, but you will use recursion instead of a loop.

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

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

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

Exercise 205: String Edit Distance

(Solved, 45 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 `looked` and `booklet`. The first string can be transformed into the second string with the following operations: Substitute the `l` with a `b`, substitute the `d` with a `t`, and insert an `l` between the `k` and the `e`. This is the smallest number of operations that can be performed to transform `looked` into `booklet`. As a result, the edit distance is 3.

Use the following algorithm to write a recursive function that computes the edit distance between two strings, s and t :

```
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 206: Possible Change

(41 Lines)

Create a program that determines whether or not it is possible to construct a particular monetary 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 a total of \$1.00 using 6 coins by using 3 quarters, 2 dimes and a nickel. Similarly, a total of \$1.25 can be formed using 5 coins or 8 coins, but a total of \$1.25 cannot be formed using 4, 6 or 7 coins.

Your program should read both the dollar amount and the number of coins from the user. Then it should display a clear message indicating whether or not the entered

dollar amount can be formed using the number of coins indicated. Assume the existence of quarters, dimes, nickels and pennies when completing this problem. Your solution must use recursion. It cannot contain any loops.

Exercise 207: Spelling with Element Symbols

(67 Lines)

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

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

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

Silver can be spelled as SiLvEr

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

Exercise 208: Element Sequences

(Solved, 83 Lines)

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

Write a program that reads the name of an element from the user and uses a recursive function to find the longest sequence of elements that begins with that

value. Display the sequence once it has been computed. Ensure that your program responds in a reasonable way if the user does not enter a valid element name.

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

Exercise 209: Flatten a List

(Solved, 33 Lines)

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

Lists that contain multiple levels of nesting can be useful when representing complex relationships between values, but the nesting can also make performing some operations on those values difficult. Flattening transforms a list that may include multiple levels of nesting into a list that contains all of the original elements without any nesting. For example, flattening the list in the previous paragraph results in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. The following recursive algorithm can be used to flatten the list stored in *data*:

```
If data is empty then
    Return the empty list
If the first element in data is a list then
    Set l1 to the result of flattening the first element in data
    Set l2 to the result of flattening all of the elements in data, except the first
    Return the concatenation of l1 and l2
If the first element in data is not a list then
    Set l1 to a list containing only the first element in data
    Set l2 to the result of flattening all of the elements in data, except the first
    Return the concatenation of l1 and l2
```

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

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

Exercise 210: Permutations of a String

(48 Lines)

A permutation of a string is a rearrangement of its characters. When each character in a string of n characters is unique, there are $n!$ permutations of the string. There are less than $n!$ distinct permutations if the string contains repeated characters. The permutations of a string, s , can be computed using the following algorithm:

If s is the empty string **then**
 Return a list containing only the empty string
Create an empty list, $result$, to store the permutations
For each index, i , in s
 Set $current$ equal to the character in s at index i
 Set $rest$ equal to s with the character at index i removed from it
 Compute all permutations of $rest$ and store them in $perms$
 For each permutation, p , in $perms$
 Append the concatenation of $current$ and p to $result$
Return $result$

Create a program that reads a string from the user and uses the recursive algorithm above to compute and display all of its permutations. It is recommended that you test your program only with short strings (8 characters or less) because the number of permutations becomes very large as the length of the string increases. A string of 4 characters, where each is distinct, has only 24 permutations, while a string of 8 characters, where each is distinct, has 40,320. The number of permutations rises to 20,922,789,888,000 when the string consists of 16 distinct characters.

The result generated by the algorithm provided in this exercise will include repeated values when the input string includes repeated characters. The repeated values can be eliminated by only appending the new permutation to the result when it is not already present in the list, or by using a set to hold the result instead of a list.

Exercise 211: Run-Length Decoding

(33 Lines)

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

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

Exercise 212: Run-Length Encoding

(Solved, 37 Lines)

Write a recursive function that implements the run-length compression technique described in Exercise 211. Your function will take a list or a string as its only argument. It should return the run-length compressed list as its only result. Include a main program that reads a string from the user, compresses it, and displays the run-length encoded result.

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

Part II

Solutions



Solutions to Selected Introductory Exercises

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 dimensions from the user.  
length = float(input("Enter the length of the room in feet: "))  
width = float(input("Enter the width of the room in feet: "))  
  
# Compute the area of the room.  
area = length * width  
  
# Display the result.  
print(f"The area of the room is {area} square feet.")
```

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

In Python, multiplication is performed using the `*` operator.

Solution to Exercise 4: Area of a Field

```
##  
# Compute the area of a field and report the result in acres.  
#  
SQFT_PER_ACRE = 43560  
  
# Read the dimensions from the user.  
length = float(input("Enter the length of the field in feet: "))  
width = float(input("Enter the width of the field in feet: "))  
  
# Compute the area in acres.  
acres = length * width / SQFT_PER_ACRE  
  
# Display the result.  
print(f"The area of the field is {acres} acres.")
```

Solution to Exercise 5: Bottle Deposits

```
##  
# Compute the refund amount for a collection of bottles.  
#  
SMALL_DEPOSIT = 0.10  
LARGE_DEPOSIT = 0.25  
  
# Read the number of containers of each size from the user.  
small = int(input("How many containers 1 litre or less? "))  
large = int(input("How many containers more than 1 litre? "))  
  
# Compute the refund amount.  
refund = small * SMALL_DEPOSIT + large * LARGE_DEPOSIT  
  
# Display the result.  
print(f"Your total refund will be ${refund:,.2f}.")
```

The `:,.2f` format specifier indicates that the value will be formatted as a floating-point number, with 2 digits to the right of the decimal point, and that commas will be used separate groups of 3 digits.

Solution to Exercise 6: Tax and Tip

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

The author's local tax rate is 5%. In Python, 5% and 18% are represented by 0.05 and 0.18, respectively.

```
# Read the cost of the meal from the user.  
cost = float(input("Enter the cost of the meal: "))  
  
# Compute the tax and the tip.  
tax = cost * TAX_RATE  
tip = cost * TIP_RATE  
total = cost + tax + tip  
  
# Display the result.  
print(f"The tax is ${tax:.2f} and the tip is ${tip:.2f},",  
     f"making the total ${total:.2f}.")
```

Python allows a function's arguments to be split across two or more lines. This is helpful when the arguments are too long to fit comfortably on one line.

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

```
##  
# Compute the sum of the first n positive integers.  
#  
# Read the value of n from the user.  
n = int(input("Enter a positive integer: "))  
  
# Compute the sum.  
total = n * (n + 1) / 2
```

Python includes a built-in function named `sum`. As a result, the variable in this program is named `total` instead of `sum`.

```
# Display the result.  
print(f"The sum of the first {n} positive integers is {total}.")
```

Solution to Exercise 10: Pythagorean Theorem

```
##  
# Use the Pythagorean Theorem to compute the length of the hypotenuse of a right triangle.  
#  
import math  
  
# Read the lengths of the shorter sides from the user.  
a = float(input("Enter the length of the first side: "))  
b = float(input("Enter the length of the second side: "))  
  
# Compute the length of the hypotenuse.  
c = math.sqrt(a * a + b * b)  
  
# Display the length of the hypotenuse.  
print(f"The length of the hypotenuse is {c}.")
```

Solution to Exercise 11: Arithmetic

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

The `math` module must be imported before the `log10` function is called. Modules are normally imported near the beginning of the file.

```
# Read the input values from the user.  
a = int(input("Enter the value of a: "))  
b = int(input("Enter the value of b: "))  
  
# Compute and display the sum, difference, product, quotient and remainder.  
print(f"{a} + {b} is {a + b}.")  
print(f"{a} - {b} is {a - b}.")  
print(f"{a} * {b} is {a * b}.")  
print(f"{a} / {b} is {a / b}.")  
print(f"{a} % {b} is {a % b}.")  
  
# Compute and display the results of the logarithm and power calculations.  
print(f"The base 10 logarithm of {a} is {math.log10(a)}.")  
print(f"{a} ** {b} is {a ** b}.")
```

The remainder is computed using the `%` operator.

Solution to Exercise 12: Pizza Planning

```
##  
# Determine how many pizzas need to be ordered to feed a group of friends.  
#  
import math  
  
SLICES_PER_PIZZA = 8  
  
# Read the input values from the user.  
group_size = int(input("How many people are in the group? "))  
per_person = int(input("How many slices per person? "))  
  
# Compute the number of pizzas.  
total_slices = group_size * per_person  
num_pizzas = math.ceil(total_slices / SLICES_PER_PIZZA)  
left_over = num_pizzas * SLICES_PER_PIZZA - total_slices  
  
# Display the result.  
print(f"{num_pizzas} pizza(s) need to be ordered.")  
print(f"There will be {left_over} slice(s) left over.")
```

Solution to Exercise 15: 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: "))  
  
# Label the output.  
print(f"The coins needed to represent {cents} cents are:")  
  
# Determine how many toonies are needed by using integer division. Then compute the amount  
# of change that still needs to be provided using the remainder operator.  
print(f" {cents // CENTS_PER_TOONIE} toonies")  
cents = cents % CENTS_PER_TOONIE
```

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

```
# Repeat the process for loonies, quarters, dimes, and nickels.  
print(f" {cents // CENTS_PER_LOONIE} loonies")  
cents = cents % CENTS_PER_LOONIE  
  
print(f" {cents // CENTS_PER_QUARTER} quarters")  
cents = cents % CENTS_PER_QUARTER  
  
print(f" {cents // CENTS_PER_DIME} dimes")  
cents = cents % CENTS_PER_DIME  
  
print(f" {cents // CENTS_PER_NICKEL} nickels")  
cents = cents % CENTS_PER_NICKEL  
  
# Display the number of pennies.  
print(f" {cents} pennies")
```

Solution to Exercise 16: Height Units

```
##  
# Convert a height in feet and inches to centimeters.  
#  
IN_PER_FT = 12  
CM_PER_IN = 2.54  
  
# Read the height from the user.  
print("Enter your height:")  
feet = int(input(" Number of feet: "))  
inches = int(input(" Number of inches: "))  
  
# Compute the equivalent number of centimeters.  
cm = (feet * IN_PER_FT + inches) * CM_PER_IN  
  
# Display the result.  
print(f"That's equal to {cm}cm.")
```

Solution to Exercise 19: 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 $2.777\text{e}-7$.

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

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

```
# Compute the energy in Joules.  
q = volume * d_temp * WATER_HEAT_CAPACITY  
  
# Display the result in Joules.  
print(f"That will require {q} Joules of energy.")  
  
# Compute the cost.  
kwh = q * J_TO_KWH  
cost = kwh * ELECTRICITY_PRICE  
  
# Display the cost.  
print(f"That much energy will cost {cost:.2f} cents.")
```

Solution to Exercise 21: Free Fall

```
##  
# Compute the speed of an object when it hits the ground after being dropped.  
#  
import math  
  
# Define a constant for the acceleration due to gravity in m / s**2.  
GRAVITY = 9.8  
  
# Read the height from which the object is dropped.  
d = float(input("Height (in meters): "))  
  
# Compute the final velocity.  
vf = math.sqrt(2 * GRAVITY * d)
```

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

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

Solution to Exercise 25: Area of the Regular Polygon

```
##  
# Compute the area of a regular polygon.  
#  
import math  
  
# Read the length and number of sides from the user.  
s = float(input("Enter the length of each side (inches): "))  
n = int(input("Enter the number of sides: "))
```

The value of n entered by the user has been converted to an integer (rather than a floating-point number) because a polygon cannot have a fractional number of sides.

```
# Compute the area of the polygon.  
area = (n * s ** 2) / (4 * math.tan(math.pi / n))  
  
# Display the result.  
print(f"The area of the polygon is {area} square inches.")
```

Solution to Exercise 27: Units of the Time (Again)

```
##  
# Convert a number of seconds to days, hours, minutes and seconds.  
#  
SECONDS_PER_DAY = 86400  
SECONDS_PER_HOUR = 3600  
SECONDS_PER_MINUTE = 60  
  
# Read the duration from the user in seconds.  
seconds = int(input("Enter a number of seconds: "))  
  
# Compute the days, hours, minutes and seconds.  
days = seconds // SECONDS_PER_DAY  
seconds = seconds % SECONDS_PER_DAY  
  
hours = seconds // SECONDS_PER_HOUR  
seconds = seconds % SECONDS_PER_HOUR  
  
minutes = seconds // SECONDS_PER_MINUTE  
seconds = seconds % SECONDS_PER_MINUTE  
  
# Display the result with the desired formatting.  
print("The equivalent duration is",  
      f"{days:d}:{hours:02d}:{minutes:02d}:{seconds:02d}")
```

The :02d format specifier tells Python to format an integer so that it includes at least two digits, by adding a leading 0, if necessary.

Solution to Exercise 31: 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  
  
# Read the air temperature and wind speed from the user.  
temp = float(input("Air temperature (degrees Celsius): "))  
speed = float(input("Wind speed (kilometers per hour): "))  
  
# Compute the wind chill index.  
wci = WC_OFFSET + \  
     WC_FACTOR1 * temp + \  
     WC_FACTOR2 * speed ** WC_EXPONENT + \  
     WC_FACTOR3 * temp * speed ** WC_EXPONENT
```

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

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.

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

Solution to Exercise 35: 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(f" {mn}")  
print(f" {md}")  
print(f" {mx}")
```

Since `min` and `max` are names of Python functions, they should not be used as variable names. The variables named `mn` and `mx` will be used to hold the minimum and maximum values, respectively.

Solution to Exercise 36: 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(f"Regular price: {regular_price:8,.2f}")  
print(f"Discount:      {discount:8,.2f}")  
print(f"Total:         {total:8,.2f}")
```

The `:8,.2f` format specifier tells Python that a total of at least 8 spaces should be used to display the number, with comma separators and 2 digits to the right of the decimal point. This will help keep the decimal points lined up when the number of digits needed for the regular price, discount and/or total are different.



Solutions to Selected Decision-Making Exercises

10

Solution to Exercise 38: 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(f"{num} is odd.")  
else:  
    print(f"{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 40: Vowel or Consonant

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

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

Solution to Exercise 41: 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(f"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 42: Month Name to Number of Days

```
##  
# Display the number of days in a month.  
  
# Read the month name from the user.  
month = input("Enter the name of a month: ")  
  
# Compute the number of days in the month.  
if month == "April" or month == "June" or \  
    month == "September" or month == "November":  
    days = 30  
elif month == "February":  
    days = "28 or 29"  
else:  
    days = 31
```

When `month` is February, the value assigned to `days` is the string "28 or 29" so that leap years are addressed. Otherwise, the value assigned to `days` is an integer.

```
# Display the result.  
print(f"{month} has {days} days in it.")
```

Solution to Exercise 44: Classifying Triangles

```
##  
# Classify a triangle based on the lengths of its sides.  
  
# Read the side lengths from the user.  
side1 = float(input("Enter the length of side 1: "))  
side2 = float(input("Enter the length of side 2: "))  
side3 = float(input("Enter the length of side 3: "))  
  
# Determine the triangle's type.  
if side1 == side2 and side2 == side3:  
    tri_type = "equilateral"  
elif side1 == side2 or side2 == side3 or \  
    side3 == side1:  
    tri_type = "isosceles"  
else:  
    tri_type = "scalene"  
  
# Display the triangle's type.  
print(f"That's a(n) {tri_type} triangle.")
```

One 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 45: 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(f"The frequency of {name} is {freq}.")
```

Solution to Exercise 46: Frequency to Note

```
##  
# Read a frequency from the user and display the note (if any) that it corresponds to.  
#  
C4_FREQ = 261.63  
D4_FREQ = 293.66  
E4_FREQ = 329.63  
F4_FREQ = 349.23  
G4_FREQ = 392.00  
A4_FREQ = 440.00  
B4_FREQ = 493.88  
LIMIT = 1  
  
# Read the frequency from the user.  
freq = float(input("Enter a frequency (Hz): "))  
  
# Identify 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(f"That frequency is {note}.")
```

Solution to Exercise 49: Birthstones

```
##  
# Report the birthstone(s) for a month entered by the user.  
#  
# Read the month from the user.  
month = input("Enter a month name: ")  
  
# Convert the month name to lowercase letters so that the correct result will be reported for any  
# combination of uppercase and lowercase letters.  
month = month.lower()  
  
# Display the birthstone(s) for the entered month, or display an error message if the entered  
# value is not a valid month name.  
if month == "january":  
    print("The birthstone for January is garnet.")  
elif month == "february":  
    print("The birthstone for February is amethyst.")  
elif month == "march":  
    print("The birthstones for March are aquamarine and",  
          "bloodstone.")  
elif month == "april":  
    print("The birthstone for April is diamond.")  
elif month == "may":  
    print("The birthstone for May is emerald.")  
elif month == "june":  
    print("The birthstones for June are pearl, alexandrite",  
          "and moonstone.")  
elif month == "july":  
    print("The birthstone for July is ruby.")  
elif month == "august":  
    print("The birthstones for August are peridot, spinel",  
          "and sardonyx.")  
elif month == "september":  
    print("The birthstone for September is sapphire.")  
elif month == "october":  
    print("The birthstones for October are opal and tourmaline.")  
elif month == "november":  
    print("The birthstones for November are topaz and citrine.")  
elif month == "december":  
    print("The birthstones for December are tanzanite,",  
          "turquoise and zircon.")  
else:  
    print("Sorry, that wasn't a valid month name.")
```

Solution to Exercise 51: Season from Month and Day

```
##  
# Identify 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 of the month: "))
```

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

```
# Identify the season.  
if month == "January" or month == "February":  
    season = "Winter"  
elif month == "March":  
    if day < 20:  
        season = "Winter"  
    else:  
        season = "Spring"  
elif month == "April" or month == "May":  
    season = "Spring"  
elif month == "June":  
    if day < 21:  
        season = "Spring"  
    else:  
        season = "Summer"  
elif month == "July" or month == "August":  
    season = "Summer"  
elif month == "September":  
    if day < 22:  
        season = "Summer"  
    else:  
        season = "Fall"  
elif month == "October" or month == "November":  
    season = "Fall"  
elif month == "December":  
    if day < 21:  
        season = "Fall"  
    else:  
        season = "Winter"  
  
# Display the result.  
print(f"{month} {day} is in {season}.")
```

Solution to Exercise 53: Chinese Zodiac

```
##  
# Identify the animal associated with a year according to the Chinese zodiac.  
#  
# Read a year from the user.  
year = int(input("Enter a year: "))  
  
# Identify 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(f"{year} is the Year of the {animal}.")
```

Solution to Exercise 55: Penalties for Speeding

```
##  
# Compute the fine and demerit points associated with speeding.  
#  
# Read the amount in excess of the speed limit that the user was travelling.  
excess = int(input("How much in excess of the speed limit? "))  
  
if excess <= 0:  
    # Display an appropriate message if the user wasn't speeding.  
    print("You weren't speeding!")
```

```
elif excess >= 50:
    # Display an appropriate message if the user was exceeding the speed limit by 50 km/h
    # or more.
    print(f"Wow! Travelling {excess} km/h over the speed",
          "limit will result in 6 demerit points and a fine",
          "that will be determined by a judge!")

else:
    # Compute the monetary penalty.
    if excess < 20:
        fine = 3.00 * excess
    elif excess < 30:
        fine = 4.50 * excess
    else:
        fine = 7.00 * excess

    # Compute the number of demerit points.
    if excess <= 15:
        demerits = 0
    elif excess < 30:
        demerits = 3
    else:
        demerits = 4

    # Display the result.
    print(f"Travelling {excess} km/h over the speed limit",
          f"will result in a fine of ${fine:.2f} and {demerits}",
          "demerit points.")
```

Solution to Exercise 57: 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 convert lowercase letters without including them in the conditions for the `if` and `elif` parts.

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

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

Solution to Exercise 59: Assessing Employees

```
##  
# Report whether an employee's performance is unacceptable, acceptable or meritorious based  
# on the rating entered by the user. This is followed by the amount of the employee's raise.  
#  
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:  
    increase = rating * RAISE_FACTOR
```

It may be tempting to name the variable that holds the salary increase `raise`. While this name would be meaningful, attempting to use it will cause Python to report a syntax error because `raise` is a reserved word.

```
print(f"Your performance is {performance}.")  
print(f"You will receive a raise of ${increase:,.2f}.")
```

Solution to Exercise 63: 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(f"{year} is a leap year.")  
else:  
    print(f"{year} is not a leap year.")
```

Solution to Exercise 66: 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 letters.

# 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 67: 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(f"The spin resulted in {value}...")

# Display the payout for a single number.
if value == 37:
    print("Pay 00")
else:
    print(f"Pay {value}")
```

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

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



Solutions to Selected Repetition Exercises

11

Solution to Exercise 72: 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 the program will need to be updated at some point in the future, so that it rounds to the closest dime instead of the closest nickel. Using constants will make it easier to perform that update when it is needed.

```
# Track the total cost for all of the items.  
total = 0.00  
  
# Read the price of the first item as a string.  
line = input("Enter the price of the item (blank to quit): ")  
  
# Continue reading items until a blank line is entered.  
while line != "":  
    # Add the cost of the item to the total (after converting it to a floating-point number).  
    total = total + float(line)  
  
    # Read the cost of the next item.  
    line = input("Enter the price of the item (blank to quit): ")  
  
# Display the exact total payable.  
print(f"The exact amount payable is ${total:.2f}")  
  
# Compute the number of pennies that would be left if the total was paid using nickels.  
rounding_indicator = total * 100 % PENNIES_PER_NICKEL
```

```

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

# Display the amount due when paying with cash.
print(f"The cash amount payable is ${cash_total:.2f}")

```

Solution to Exercise 73: Compute the Perimeter of a Polygon

```

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

# Store the perimeter of the polygon.
perimeter = 0

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

# Save the first point so that the distance from the last point to the first point can be computed.
first_x = x
first_y = y

# Set prev_x and prev_y equal to the first point so that the distance to the first point is computed
# when the second point is entered.
prev_x = x
prev_y = y

# Continue reading coordinates until a blank line is entered for the x-coordinate.
line = input("Enter the next x-coordinate (blank to quit): ")
while line != "":
    # Convert the x-coordinate to a number and read the y coordinate.
    x = float(line)
    y = float(input("Enter the next y-coordinate: "))

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

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

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

```

The distance between the points is computed using the Pythagorean theorem.

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

# Display the result.
print(f"The perimeter of that polygon is {perimeter}.)"
```

Solution to Exercise 75: 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

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

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

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

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

With the current admission prices, the first part of the `if-elif-else` statement could be omitted. However, including it will make the program easier to update in the future if a fee is added for babies.

Solution to Exercise 76: Parity Bits

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

# Read the first line of input from the user.
line = input("Enter 8 bits (blank to quit): ")
```

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

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

```
# Display the parity bit.  
if ones % 2 == 0:  
    print("The parity bit should be 0.")  
else:  
    print("The parity bit should be 1.")  
  
# Read the next line of input.  
line = input("Enter 8 bits (blank to quit): ")
```

Solution to Exercise 79: Universal Product Codes

```
##  
# Determine whether or not a 12-digit number is a valid universal product code.  
#  
# Read the digits from the user.  
upc = input("Enter a 12-digit universal product code: ")  
  
# Verify that exactly 12 characters were entered.  
if len(upc) != 12:  
    print("That wasn't a 12-digit number. Quitting...")  
    quit()  
  
# Verify that all of the entered characters are digits.  
for i in range(len(upc)):  
    if upc[i] < "0" or upc[i] > "9":  
        print(f"{upc[i]} isn't a valid character in a universal",  
              "product code. Quitting...")  
        quit()  
  
# Initialize total to 0.  
total = 0  
  
# Consider each digit at even position (starting from 0).  
for i in range(0, len(upc), 2):  
    # Add three times the value of the digit to the total.  
    total = total + int(upc[i]) * 3
```

```
# Consider each digit at odd position (up to and including 9).
for i in range(1, 10, 2):
    # Add the value of the digit to the total.
    total = total + int(upc[i])

# Determine if the total is evenly divisible by 10.
if total % 10 == 0:
    # The check digit is equal to 0.
    check_digit = 0
else:
    # The check digit is 10 minus the remainder when total is divided by 10.
    check_digit = 10 - total % 10

# Compare the check digits and report the result.
if check_digit == int(upc[11]):
    print(f"{upc} is a valid universal product code.")
else:
    print(f"{upc} is not a valid universal product code.")
    print(f"The computed check digit is {check_digit}, which",
          f"doesn't match {upc[11]}.")
```

Solution to Exercise 80: Caesar Cipher

```
## 
# Implement a Caesar cipher that shifts all of the letters in a message by an amount provided
# by the user. Use a negative shift value to decode a message.
#
# Read the message and shift amount from the user.
message = input("Enter the message: ")
shift = int(input("Enter the shift value: "))

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

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

```
# Display the shifted message.
print(f"The shifted message is {new_message}.")
```

Solution to Exercise 82: Is a String a Palindrome?

```
##  
# Determine whether or not a string entered by the user is a palindrome.  
#  
# Read the string from the user.  
line = input("Enter a string: ")  
  
# Assume that it is a palindrome until it can be proved otherwise.  
is_palindrome = True  
  
# Check the characters, starting from the ends. Continue until the middle is reached, or it has  
# been determined that the string is not a palindrome.  
i = 0  
while i < len(line) / 2 and is_palindrome:  
    # If the characters do not match then mark that the string is not a palindrome.  
    if line[i] != line[len(line) - i - 1]:  
        is_palindrome = False  
  
    # Move to the next character.  
    i = i + 1  
  
# Display a meaningful output message.  
if is_palindrome:  
    print(f"{line} is a palindrome.")  
else:  
    print(f"{line} is not a palindrome.")
```

Solution to Exercise 84: Multiplication Table

```
##  
# Display a multiplication table for 1 times 1 through 10 times 10.  
#  
MIN = 1  
MAX = 10  
  
# Display the labels for the columns.  
print("      ", end="")  
for i in range(MIN, MAX + 1):  
    print(f"{i:4d}", end="")  
print()  
  
# Display the table.  
for i in range(MIN, MAX + 1):  
    # Display the row's label.  
    print(f"{i:4d}", end="")  
  
    # Display the values for the row.  
    for j in range(MIN, MAX + 1):  
        print(f"{i * j:4d}", end="")  
    print()
```

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

Solution to Exercise 87: 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 another 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(f"The greatest common divisor of {n} and {m} is {d}.")
```

Solution to Exercise 90: Decimal to Binary

```
##  
# Convert a number from decimal (base 10) to binary (base 2).  
  
#  
NEW_BASE = 2  
  
# Read the number to convert from the user.  
num = int(input("Enter a non-negative integer: "))  
  
# Generate the binary representation of num, storing it in result.  
result = ""  
q = num
```

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

```
# Perform the body of the repeat-until loop in the provided algorithm once.  
r = q % NEW_BASE  
result = str(r) + result  
q = q // NEW_BASE  
  
# Continue looping until q is 0.  
while q > 0:  
    r = q % NEW_BASE  
    result = str(r) + result  
    q = q // NEW_BASE  
  
# Display the result.  
print(f"{num} in decimal is {result} in binary.")
```

Solution to Exercise 92: Maximum Integer

```
##  
# Find the maximum of 100 random integers, and count the number of times the maximum value  
# is updated during the process.  
#  
from random import randrange  
  
# Store the number of random integers that will be processed and the maximum integer that  
# will be generated. These are separate constants so that one can be updated without modifying  
# the other.  
NUM_ITEMS = 100  
MAX_VAL = 100  
  
# Generate the first number and display it.  
mx_value = randrange(1, MAX_VAL + 1)  
print(mx_value)  
  
# Count the number of times the maximum value is updated.  
num_updates = 0  
  
# Loop once for each of the remaining numbers.  
for i in range(1, NUM_ITEMS):  
    # Generate a new random number.  
    current = randrange(1, MAX_VAL + 1)  
  
    # Determine if the generated number is the largest one that has been observed so far.  
    if current > mx_value:  
        # Update the maximum and count the update.  
        mx_value = current  
        num_updates = num_updates + 1  
  
    # Display the number, marking that an update occurred.  
    print(current, "<== Update")  
else:  
    # Display the number.  
    print(current)  
  
# Display the other results.  
print(f"The maximum value found was {mx_value}.")  
print(f"The maximum value was updated {num_updates} times.")
```

Solution to Exercise 94: Monty Hall Problem

```
##  
# Simulate the Monty Hall problem to determine whether or not the contestant should switch  
# their choice.  
#  
import random  
  
# The number of games that will be simulated.  
NUM_GAMES = 1000000  
  
# Count the number of games where the player wins the car if they switch their choice.  
better_to_switch = 0
```

```
# Each iteration of the loop is one game.  
print(f"Simulating {NUM_GAMES:,} games...")  
for i in range(NUM_GAMES):  
    # Select the door hiding the car.  
    car_door = random.randrange(1, 4)  
  
    # Select a door for the player.  
    player_door = random.randrange(1, 4)  
  
    # Select a door for the host to open, ensuring that it is not the player's door and not the  
    # door hiding the car.  
    reveal_door = random.randrange(1, 4)  
    while reveal_door == player_door or reveal_door == car_door:  
        reveal_door = random.randrange(1, 4)  
  
    # If the door initially chosen by the player is not the door hiding the car then the player  
    # needs to switch doors to win the car.  
    if player_door != car_door:  
        better_to_switch = better_to_switch + 1  
  
# Display the percentage of games where the player needed to switch doors to win the car.  
percent = better_to_switch / NUM_GAMES * 100  
print(f"It was better to switch in {percent:.1f}% of games!")
```



Solutions to Selected Function Exercises

12

Solution to Exercise 97: Scores and Years

```
##  
# Use a function to convert from scores and years to the equivalent number of years.  
#  
YEARS_PER_SCORE = 20  
  
## Compute the number of years represented by a number  
# of scores and years.  
# @param scores the number of scores  
# @param years the number of years  
# @return the total number of years  
def yearsFromScores(scores, years):  
    return scores * YEARS_PER_SCORE + years  
  
# Demonstrate the yearsFromScores function.  
def main():  
    # Read the scores and years from the user.  
    s = int(input("How many scores? "))  
    y = int(input("How many years? "))  
  
    # Compute the total number of years.  
    total = yearsFromScores(s, y)  
  
    # Display the result.  
    print(f"{s} score and {y} years is {total} years.")  
  
# Call the main function.  
main()
```

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 described by a line that begins with `@return`.

Solution to Exercise 99: 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)  
  
# 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()
```

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

Solution to Exercise 102: The Twelve Days of Christmas

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

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

```
## Display one verse of The Twelve Days of Christmas.  
# @param n the verse to display  
# @return (None)  
def displayVerse(n):  
    print("On the", intToOrdinal(n), "day of Christmas")  
    print("my true love gave 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 106: Center a String in the Terminal Window

```
##  
# Center a string of characters within a certain width.  
#  
WIDTH = 80  
  
## Create a new string that will be centered within a given width when it is printed. This version  
# of the function uses string replication and string concatenation.  
# @param s the string that will be centered  
# @param width the width in which the string will be centered  
# @return a new copy of s that contains the leading spaces needed to center s  
def center_rep_concat(s, width):  
    # If the string is too long to center, then the original string is returned.  
    if width < len(s):  
        return s  
    # Compute the number of spaces needed, and generate the result.  
    spaces = (width - len(s)) // 2  
    result = " " * spaces + s
```

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

```
return result  
  
## Create a new string that will be centered within a given width when it is printed. This version  
# of the function uses f-strings.  
# @param s the string that will be centered  
# @param width the width in which the string will be centered  
# @return a new copy of s that contains the leading spaces needed to center s  
def center_f(s, width):  
    return f"{s:{width}s}"  
  
# Demonstrate both versions of the center function.  
def main():  
    print(center_rep_concat("A Famous Story", WIDTH))  
    print(center_rep_concat("by:", WIDTH))  
    print(center_rep_concat("Someone Famous", WIDTH))  
    print()  
    print("Once upon a time...")  
    print()  
  
    print(center_f("A Famous Story", WIDTH))  
    print(center_f("by:", WIDTH))  
    print(center_f("Someone Famous", WIDTH))  
    print()  
    print("Once upon a time...")  
    print()  
  
# Call the main function.  
main()
```

Solution to Exercise 108: Capitalize It

```
##  
# Improve the capitalization of a string.  
  
## Capitalize the appropriate characters in a string.  
# @param s the string that needs capitalization  
# @return a new string with the capitalization improved  
def capitalize(s):  
    # Create a new copy of the string to return as the function's result.  
    result = s  
  
    # Capitalize the first non-space character in the string.  
    pos = 0  
    while pos < len(s) and result[pos] == ' ':  
        pos = pos + 1  
  
    if pos < len(s):  
        # Replace the character with its uppercase version, without changing any other characters.  
        result = result[0 : pos] + result[pos].upper() + \  
                result[pos + 1 : len(result)]
```

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

```
# Capitalize the first letter that follows a “.”, “!” or “?”.  
pos = 0  
while pos < len(s):  
    if result[pos] == "." or result[pos] == "!" or \  
       result[pos] == "?":  
        # Move past the “.”, “!” or “?”.  
        pos = pos + 1  
  
    # Move past any spaces.  
    while pos < len(s) and result[pos] == " ":  
        pos = pos + 1  
  
    # If the end of the string hasn't been reached, then replace the current character with  
    # its uppercase version.  
    if pos < len(s):  
        result = result[0 : pos] + \  
                result[pos].upper() + \  
                result[pos + 1 : len(result)]  
    else:  
        # Move to the next character.  
        pos = pos + 1
```

```

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

```

Solution to Exercise 109: Does a String Represent an Integer?

```

## 
# Determine whether or not the characters entered by the user represent an integer.
#
## Determine if a string is 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()

    # If the stripped string is empty, then it does not represent an integer.
    if s == "":
        return False

    # Determine if the remaining characters are 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.")

```

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.

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

The `__name__` variable is automatically assigned a value by Python when the program starts running. It contains "`__main__`" when the file is executed directly by Python. It contains the name of the .py file (without the .py extension) when the file is imported into another program.

Solution to Exercise 111: Is a Number Prime?

```
##  
# Determine if a number entered by the user is prime.  
#  
## Determine whether or not a number is prime.  
# @param n the integer to test  
# @return True if the number is prime, False otherwise  
def isPrime(n):  
    if n <= 1:  
        return False  
  
    # Check each number from 2 up to, but not including, n, to see if it divides evenly into n.  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    return True
```

If $n \% i == 0$, then n is evenly divisible by i , which means that n is not prime.

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

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

Solution to Exercise 113: Random Password

```
##  
# Generate and display a random password containing between 7 and 10 characters.  
#  
from random import randint  
  
SHORTEST = 7  
LONGEST = 10  
MIN_ASCII = 33  
MAX_ASCII = 126  
  
## Generate a random password.  
# @return a string containing a random password  
def randomPassword():  
    # Select a random length for the password.  
    randomLength = randint(SHORTEST, LONGEST)  
  
    # Generate an appropriate number of random characters, adding each one to the end of result.  
    result = ""  
    for i in range(randomLength):  
        randomChar = chr(randint(MIN_ASCII, MAX_ASCII))  
        result = result + randomChar
```

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

```
# Return the random password.  
    return result  
  
# Generate and display a random password.  
def main():  
    print("Your random password is:", randomPassword())  
  
# Only call the main function when this file has not been imported into another program.  
if __name__ == "__main__":  
    main()
```

Solution to Exercise 115: Check a Password

```
##  
# Check whether or not a password meets a set of constraints.  
  
## Check whether or not a password is at least 8 characters, and contains an uppercase letter, a  
# lowercase letter and a digit.  
# @param password the password to check  
# @return True if the password meets the constraints, False otherwise  
def checkPassword(password):  
    has_upper = False  
    has_lower = False  
    has_digit = False  
  
    # Check each character in the password and record the constraint that 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_digit = True  
  
    # If the password meets all 4 constraints, then return True.  
    if len(password) >= 8 and has_upper and \  
       has_lower and has_digit:  
        return True  
  
    # If the password fails to meet any of the constraints, then return False.  
    return False  
  
# Demonstrate the password checking function.  
def main():  
    p = input("Enter a password: ")  
    if checkPassword(p):  
        print("That password is acceptable.")  
    else:  
        print("That password is *NOT* acceptable.")  
  
# Only call the main function when this file has not been imported into another program.  
if __name__ == "__main__":  
    main()
```

Solution to Exercise 118: 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 `hex2int` and `int2hex` functions which were developed while solving Exercise 117. Using `import *` imports all of the functions from that module.

```
## Convert a number from base 10 to base new_base.  
# @param num the base 10 number to convert  
# @param new_base the base to convert to  
# @return the string of digits in new_base  
def dec2n(num, new_base):  
    # Generate the representation of num in base new_base, storing it in result.  
    result = ""  
    q = num  
  
    # Perform the body of the loop once.  
    r = q % new_base  
    result = int2hex(r) + result  
    q = q // new_base  
  
    # Continue looping until q is 0.  
    while q > 0:  
        r = q % new_base  
        result = int2hex(r) + result  
        q = q // new_base  
  
    # Return the result.  
    return result  
  
## Convert a number from base b to base 10.  
# @param num the base b number, stored in a string  
# @param b the base of the number to convert  
# @return the base 10 number  
def n2dec(num, b):  
    decimal = 0  
  
    # Process each digit in the base b number.  
    for i in range(len(num)):  
        decimal = decimal * b  
        decimal = decimal + hex2int(num[i])  
  
    # Return the result.  
    return decimal
```

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

    from_num = input(f"Enter a base {from_base} number: ")

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

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

    to_num = dec2n(dec, to_base)
    print(f"That's {to_num} in base {to_base}.")

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

Solution to Exercise 119: Reduce a Fraction to Lowest Terms

```
## 
# Reduce a fraction to lowest terms.
#
## Compute the greatest common divisor of two integers.
# @param n the first integer (must be non-zero)
# @param m the second integer (must be non-zero)
# @return the greatest common divisor of n and m
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 the greatest common divisor.
return d
```

This function uses a loop to achieve its goal. There is also an elegant solution for finding the greatest common divisor of two integers that uses recursion. The recursive solution is explored in Exercise 197.

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

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

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

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

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

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

    # Display the result.
    print(f"\{num}\{den} can be reduced to \{n}\{d}.")

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

Solution to Exercise 120: Reduce Measures

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

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

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

```
# Compute the number of teaspoons that the parameters represent.  
if unit == "teaspoon" or unit == "teaspoons":  
    teaspoons = num  
elif unit == "tablespoon" or unit == "tablespoons":  
    teaspoons = num * TSP_PER_TBSP  
elif unit == "cup" or unit == "cups":  
    teaspoons = num * TSP_PER_CUP  
  
# Convert the number of teaspoons to the largest possible units of measure.  
cups = teaspoons // TSP_PER_CUP  
teaspoons = teaspoons % TSP_PER_CUP  
tablespoons = teaspoons // TSP_PER_TBSP  
teaspoons = teaspoons % TSP_PER_TBSP  
  
# Create a string to hold the result.  
result = ""  
  
# Add the number of cups (if any) to the result.  
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 (if any) to the result.  
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 (if any) to the result string.  
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 cases that include one, some and many of each unit of measure. While these test cases are reasonably thorough, they do not guarantee that the program is bug free. More generally, testing can be used to reveal bugs, but it cannot prove that no bugs remain.

```
# Demonstrate the reduceMeasure function by performing several reductions.
def main():
    reduced = reduceMeasure(1, "teaspoon")
    print(f"1 teaspoon is {reduced}.")

    reduced = reduceMeasure(3, "teaspoons")
    print(f"3 teaspoons is {reduced}.")

    reduced = reduceMeasure(7, "teaspoons")
    print(f"7 teaspoons is {reduced}.")

    reduced = reduceMeasure(59, "teaspoons")
    print(f"59 teaspoons is {reduced}.")

    reduced = reduceMeasure(95, "teaspoons")
    print(f"95 teaspoons is {reduced}.")

    reduced = reduceMeasure(96, "teaspoons")
    print(f"96 teaspoons is {reduced}.")

    reduced = reduceMeasure(97, "teaspoons")
    print(f"97 teaspoons is {reduced}.")

    reduced = reduceMeasure(1, "tablespoon")
    print(f"1 tablespoon is {reduced}.")

    reduced = reduceMeasure(4, "tablespoons")
    print(f"4 tablespoons is {reduced}.")

    reduced = reduceMeasure(59, "tablespoons")
    print(f"59 tablespoons is {reduced}.")

    reduced = reduceMeasure(1, "cup")
    print(f"1 cup is {reduced}.")

    reduced = reduceMeasure(2, "cups")
    print(f"2 cups is {reduced}.")
```

```
reduced = reduceMeasure(40, "cups")
print(f"40 cups is {reduced}.")
```

Call the main function.
main()

Solution to Exercise 121: Magic Dates

```
##  
# Identify 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(f"{year:04d}-{month:02d}-{day:02d} is magic.")  
  
# Call the main function.  
main()
```

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



Solutions to Selected List Exercises

13

Solution to Exercise 123: Sorted Order

```
##  
# Display a list of integers entered by the user in ascending order.  
#  
# Start with an empty list.  
data = []  
  
# Read values and add them to the list until the user enters 0.  
num = int(input("Enter an integer (0 to quit): "))  
while num != 0:  
    data.append(num)  
    num = int(input("Enter an integer (0 to quit): "))  
  
# Sort the values.  
data.sort()
```

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

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

Solution to Exercise 125: Remove Outliers

```
##  
# Remove the outliers from a data set.  
  
## Remove the outliers from a list of values. A new copy of the list is returned with the  
# outliers removed. The list passed as a parameter is not modified.  
# @param data the list of values to process  
# @param num_outliers the number of smallest and largest values to remove  
# @return a new copy of data where the values are sorted into ascending order and the  
# smallest and largest values have been removed  
def removeOutliers(data, num_outliers):  
    # Create a new copy of the list that is in sorted order.  
    retval = sorted(data)  
  
    # Remove num_outliers largest values.  
    for i in range(num_outliers):  
        retval.pop()  
  
    # Remove num_outliers smallest values.  
    for i in range(num_outliers):  
        retval.pop(0)  
  
    # Return the result.  
    return retval  
  
# Read values from the user, store them in a list, and then 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 126: Avoiding Duplicates

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

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

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

Solution to Exercise 127: Negatives, Zeros, and Positives

```
##  
# Read a collection of integers from the user. Display all of the negative numbers, followed  
# by all of the zeros, followed by all of the positive numbers.  
  
# Create three lists to store the negative, zero and positive values.  
negatives = []  
zeros = []  
positives = []
```

This solution uses a list to keep track of the zeros that are entered. However, because all of the zeros are the same, it is not actually necessary to save them. Instead, one could use an integer variable to count the number of zeros, and then display that many zeros later in the program.

```
# Read all of the integers from the user, storing each integer in the correct list.  
line = input("Enter an integer (blank to quit): ")  
while line != "":  
    num = int(line)  
  
    # Append the number to the correct list.  
    if num < 0:  
        negatives.append(num)  
    elif num > 0:  
        positives.append(num)  
    else:  
        zeros.append(num)  
  
    # Read the next line of input from the user.  
    line = input("Enter an integer (blank to quit): ")  
  
# Display all of the negative values, then all of the zeros, then all of the positive values.  
print("The numbers were: ")  
for n in negatives + zeros + positives:  
    print(n)
```

Solution to Exercise 129: 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  
  
    # Determine whether or not the number is perfect and return the appropriate result.  
    if total == n:  
        return True  
    return False  
  
# Display all of the perfect numbers between 1 and LIMIT.  
def main():  
    print(f"The perfect numbers between 1 and {LIMIT} are:")  
    for i in range(1, LIMIT + 1):  
        if isPerfect(i):  
            print(f"    {i}")  
  
# Call the main function.  
main()
```

The total could also be computed using Python's built-in `sum` function. This would reduce the calculation of the total to a single line.

Solution to Exercise 133: 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 with fewer than 2 elements as special cases.  
    if len(items) == 0:  
        return "<empty>"  
    if len(items) == 1:  
        return str(items[0])
```

```
# Iterate over all of the items in the list, except the last two.  
result = ""  
for i in range(0, len(items) - 2):  
    result = result + str(items[i]) + ", "
```

Each item is explicitly converted to a string by calling the `str` function before it is concatenated to the result. This allows the `formatList` function to format lists that contain numbers in addition to strings.

```
# Add the second last and last items to the result, separated by "and".  
result = result + str(items[len(items) - 2]) + " and "  
result = result + str(items[len(items) - 1])
```

```
# Return the result.  
return result
```

```
# Read several items entered by the user and display them with nice formatting.
```

```
def main():  
    # Read items from the user until a blank line is entered.  
    items = []  
    line = input("Enter an item (blank to quit): ")  
    while line != "":  
        items.append(line)  
        line = input("Enter an item (blank to quit): ")  
  
    # Format and display the items.  
    if len(items) == 0:  
        print("No items were entered.")  
    elif len(items) == 1:  
        print(f"The item is {formatList(items)}")  
    else:  
        print(f"The items are {formatList(items)}")
```

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

Solution to Exercise 134: Random Lottery Numbers

```
##  
# Generate random numbers for a lottery ticket, ensuring that each number is distinct.  
#  
from random import randrange  
  
MIN_NUM = 1  
MAX_NUM = 49  
NUM_NUMS = 6  
  
# Use a list to store the numbers on the ticket.  
ticket_nums = []
```

Using constants makes it easy to reconfigure the program for other lotteries.

```
# Generate NUM_NUMS random but distinct numbers.
for i in range(NUM_NUMS):
    # Select a number that isn't already on the ticket.
    rand = randrange(MIN_NUM, MAX_NUM + 1)
    while rand in ticket_nums:
        rand = randrange(MIN_NUM, MAX_NUM + 1)

    # Add the number to the ticket.
    ticket_nums.append(rand)

# Sort the numbers into ascending order and display them.
ticket_nums.sort()
print("Your numbers are: ", end="")
for n in ticket_nums:
    print(n, end=" ")
print()
```

Solution to Exercise 137: Balanced Parentheses and Square Brackets

```
## 
# Determine whether or not the open and close glyphs are balanced in a string entered by the
# user. Any characters in the string that are not parentheses or square brackets are ignored.
#

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

# Initialize glyphs and indices to empty lists.
glyphs = []
indices = []

# Initialize i to 0 and error to False.
i = 0
error = False

# Process characters in the string until every character is examined or an error is detected.
while i < len(s) and not error:
    # If the character at position i in s is an open glyph then...
    if s[i] == "(" or s[i] == "[":
        # Add the glyph and its position to the lists.
        glyphs.append(s[i])
        indices.append(i)

    # If the character at position i in s is a close parenthesis then...
    if s[i] == ")":
        # If glyphs is empty, or its last element is not an open parenthesis then...
        if len(glyphs) == 0 or glyphs[len(glyphs) - 1] != "(":
            # Mark that an error has occurred.
            error = True
        else:
            # Remove the corresponding open glyph and its position from the lists.
            glyphs.pop()
            indices.pop()
```

```

# If the character at position i in s is a close square bracket then...
if s[i] == "]":
    # If glyphs is empty, or its last element is not an open square bracket then...
    if len(glyphs) == 0 or glyphs[len(glyphs) - 1] != "[":
        # Mark that an error has occurred.
        error = True
    else:
        # Remove the corresponding open glyph and its position from the lists.
        glyphs.pop()
        indices.pop()

# Move to the next character in the string.
i = i + 1

# Report the status of the text entered by the user.
print()
if error:
    print("A close glyph didn't have a matching open glyph.")
    print()

# Display the string and mark the position of the unbalanced close glyph.
print(s)
print(" " * (i - 1) + "^")
elif len(glyphs) != 0:
    print("An open glyph didn't have a matching close glyph.")
    print()

# Display the string and mark the position of the unbalanced open glyph.
print(s)
print(" " * (indices[len(indices) - 1]) + "^")
else:
    print("The glyphs are balanced.")

```

Solution to Exercise 140: Shuffling a Deck of Cards

```

## 
# Create a deck of cards and shuffle it.
#
from random import randrange

## Construct a standard deck of cards with 4 suits and 13 values per suit.
# @return a list of cards, with each card represented by two characters
def createDeck():
    # Create a list to hold the cards.
    cards = []

    # Loop once for each suit and value.
    for suit in ["s", "h", "d", "c"]:
        for value in ["2", "3", "4", "5", "6", "7", "8", "9", \
                     "T", "J", "Q", "K", "A"]:
            # Construct the card and add it to the list.
            cards.append(value + suit)

    # Return the complete deck of cards.
    return cards

```

```

## Shuffle a deck of cards by modifying the deck passed to the function.
# @param cards the list of cards to shuffle
# @return (None)
def shuffle(cards):
    # Loop once for each card in the list.
    for i in range(0, len(cards)):
        # Pick a random index between the current index and the end of the list.
        other_pos = randrange(i, len(cards))

        # Swap the current card with the one at the random position.
        temp = cards[i]
        cards[i] = cards[other_pos]
        cards[other_pos] = temp

# Display a deck of cards before and after it has been shuffled.
def main():
    cards = createDeck()
    print("The original deck of cards is: ")
    print(cards)
    print()

    shuffle(cards)
    print("The shuffled deck of cards is: ")
    print(cards)

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

```

Solution to Exercise 143: Count the Elements

```

## 
# Count the number of elements in a list that are greater than or equal to some minimum
# value and less than some maximum value.
#
## Determine how many elements in data are greater than or equal to mn and less than mx.
# @param data the list of values to examine
# @param mn the minimum acceptable value
# @param mx the exclusive upper bound on acceptability
# @return the number of elements, e, such that mn <= e < mx
def countRange(data, mn, mx):
    # Count the number of elements within the acceptable range.
    count = 0
    for e in data:
        # Check the current 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 of the elements are within the range.  
print("Counting the elements in [1..10] between 5 and 7...")  
print(f"Result: {countRange(data, 5, 7)} Expected: 2")  
  
# Test a case where all of the elements are within the range.  
print("Counting the elements in [1..10] between -5 and 77...")  
print(f"Result: {countRange(data, -5, 77)} Expected: 10")  
  
# Test a case where no elements are within the range.  
print("Counting the elements in [1..10] between 12 and 17...")  
print(f"Result: {countRange(data, 12, 17)} Expected: 0")  
  
# Test a case where the list is empty.  
print("Counting the elements in [] between 0 and 100...")  
print(f"Result: {countRange([], 0, 100)} Expected: 0")  
  
# Test a case with duplicate values.  
data = [1, 2, 3, 4, 1, 2, 3, 4]  
print(f"Counting the elements in {data} between 2 and 4...")  
print(f"Result: {countRange(data, 2, 4)} Expected: 4")  
  
# Call the main program.  
main()
```

Solution to Exercise 144: Tokenizing a String

```
##  
# Tokenize a string containing a mathematical expression.  
#  
## Convert a string containing a mathematical expression into a list of tokens.  
# @param s the string to tokenize  
# @return a list of the tokens in s, or an empty list if an error occurs  
def tokenList(s):  
    # Remove all of the spaces from s.  
    s = s.replace(" ", "")  
  
    # Loop through all of the characters in the string, identifying the tokens and adding them to  
    # the list.  
    tokens = []  
    i = 0  
    while i < len(s):  
        # Handle the tokens that are always a single character: +, -, *, /, ^, ( and ).  
        if s[i] == "+" or s[i] == "-" or s[i] == "*" or s[i] == "/" or s[i] == "^" or s[i] == "(" or s[i] == ")":  
            tokens.append(s[i])  
            i = i + 1
```

```

# Handle a number.
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)

# If any other character is present in the string, then the expression is not valid. Return
# an empty list to indicate such.
else:
    return []

return tokens

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

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

```

Solution to Exercise 145: Unary and Binary Operators

```

## 
# Update a token list so that unary and binary + and - operators are differentiated.
#
from token_list import tokenList

## Identify occurrences of unary + and - operators within a list of tokens and replace them
# with u+ and u- respectively.
# @param tokens a list of tokens that may include unary + and - operators
# @return a list of tokens where unary + and - operators have been replaced with u+ and u-
def identifyUnary(tokens):
    retval = []

    # Process each token in the list.
    for i in range(len(tokens)):
        # If the first token in the list is + or -, then it is a unary operator.
        if i == 0 and (tokens[i] == "+" or tokens[i] == "-"):
            retval.append("u" + tokens[i])
        # If the token is a + or - and the previous token is an operator or an open parenthesis,
        # then it is a unary operator.
        elif i > 0 and (tokens[i] == "+" or tokens[i] == "-") and \
            (tokens[i-1] == "+" or tokens[i-1] == "-" or
             tokens[i-1] == "*" or tokens[i-1] == "/" or
             tokens[i-1] == "("):
            retval.append("u" + tokens[i])
        # Any other token is not a unary operator. Append it to the result without modification.
        else:
            retval.append(tokens[i])

```

```
# Return the new list of tokens where the unary operators have been marked.  
    return retval  
  
# Demonstrate that unary operators are marked correctly.  
def main():  
    # Read an expression from the user, tokenize it, and display the result.  
    exp = input("Enter a mathematical expression: ")  
    tokens = tokenList(exp)  
    print(f"The tokens are: {tokens}")  
  
    # Identify the unary operators in the list of tokens.  
    marked = identifyUnary(tokens)  
    print(f"With unary operators marked: {marked}")  
  
# Call the main function only if this file has not been imported into another program.  
if __name__ == "__main__":  
    main()
```

Solution to Exercise 149: Generate All Sublists of a List

```
##  
# Compute and display all of the sublists of several lists.  
#  
## Generate a list 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 up to and including len(data).  
    for length in range(1, len(data) + 1):  
        # Generate each sublist of the current length.  
        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]))
```

A list containing an empty list is denoted by `[]`.

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

Solution to Exercise 150: The Sieve of Eratosthenes

```
##  
# Identify all of the prime numbers from 2 to some limit entered by the user using the Sieve  
# of Eratosthenes.  
#  
# Read the limit from the user.  
limit = int(input("Identify all primes up to what limit? "))  
  
# Create a list that contains all of the integers from 0 to limit.  
nums = []  
for i in range(0, limit + 1):  
    nums.append(i)  
  
# “Cross out” 1 by replacing it with 0.  
nums[1] = 0  
  
# “Cross out” all of the multiples of each prime number that is discovered.  
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(f"The primes up to {limit} are:")  
for i in nums:  
    if nums[i] != 0:  
        print(i)
```

Solution to Exercise 151: Normal Magic Squares

```
##  
# Determine whether or not a table of integers is a normal magic square.  
#  
# Create a new empty table.  
table = []  
  
# Read the first line of input and split it into a list of values.  
s = input("Enter the first row of integers (comma separated): ")  
row = s.split(",")  
  
# Convert each value in the first row into an integer, and add the row to the table.  
for i in range(len(row)):  
    row[i] = int(row[i])  
table.append(row)
```

```
# For each subsequent row of input...
for j in range(1, len(row)):
    # Read the next line of input and split it into a list of values.
    s = input("Enter the next row of integers (comma separated): ")
    row = s.split(",")
    
    # Verify that the row that was just read has the correct number of values.
    if len(row) != len(table[0]):
        print(f"The first row in the table had {len(table[0])} values",
              f"but that row only had {len(row)} values.")
        print("Quitting...")
        quit()

    # Convert each value in this row into an integer, and add the row to the table.
    for i in range(len(row)):
        row[i] = int(row[i])
    table.append(row)

# Assume that the table is a magic square until it is shown that it is not.
is_magic = True

# Determine whether or not the entered values form a normal magic square. Begin by computing
# the sum for the first column. This is referred to as the magic constant for the table.
magic_constant = 0
for row in range(0, len(table)):
    magic_constant = magic_constant + table[row][0]

# Verify that all of the other columns sum to the magic constant.
for col in range(1, len(table[0])):
    total = 0
    for row in range(0, len(table)):
        total = total + table[row][col]

    # Verify that the total for the current column matches the magic constant.
    if total != magic_constant:
        print(f"Column {col} sums to {total} but the",
              f"magic constant is {magic_constant}.")
        is_magic = False

# Verify that all of the rows sum to the magic constant.
for row in range(0, len(table)):
    total = 0
    for col in range(0, len(table[row])):
        total = total + table[row][col]

    # Verify that the total for the current row matches the magic constant.
    if total != magic_constant:
        print(f"Row {row} sums to {total} but the",
              f"magic constant is {magic_constant}.")
        is_magic = False
```

```
# Sum the values along the top-left to bottom-right and bottom-left to top-right diagonals.
diag_tl_br = 0
diag_bl_tr = 0
for i in range(len(table)):
    diag_tl_br = diag_tl_br + table[i][i]
    diag_bl_tr = diag_bl_tr + table[len(table)-1-i][i]

# Verify that the total for the top-left to bottom-right diagonal matches the magic constant.
if diag_tl_br != magic_constant:
    print("The top-left to bottom-right diagonal sums to",
          f"{diag_tl_br} but the magic constant is",
          f"{magic_constant}.")
    is_magic = False

# Verify that the total for the bottom-left to top-right diagonal matches the magic constant.
if diag_bl_tr != magic_constant:
    print("The bottom-left to top-right diagonal sums to",
          f"{diag_bl_tr} but the magic constant is",
          f"{magic_constant}.")
    is_magic = False

# If the square is magic...
if is_magic:
    # Collect all of the values into one long list so that it is easier to check if the table contains
    # only consecutive integers starting from 1.
    all_values = []
    for row in range(len(table)):
        all_values = all_values + table[row]

    # Verify that each integer is present from 1 to (num rows) * (num cols).
    normal = True
    for i in range(1, len(table) * len(table[0]) + 1):
        if i not in all_values:
            normal = False

    # Report whether or not the magic square is a normal magic square.
    if normal == True:
        print("It's a normal magic square.")
    else:
        print("It's a magic square, but not a normal magic square.")
    print(f"The magic constant is {magic_constant}.")
```



Solutions to Selected Dictionary Exercises

14

Solution to Exercise 153: Reverse Lookup

```
##  
# Conduct a reverse lookup on a dictionary, finding all of the keys that map to the provided  
# value.  
  
## Conduct a reverse lookup on a dictionary.  
# @param data the dictionary on which the reverse lookup is performed  
# @param value the value to search for in the dictionary  
# @return a (possibly empty) list of keys in data that map to value  
def reverseLookup(data, value):  
    # Construct a list of the keys that map to value.  
    keys = []  
  
    # Check each key, and add it to keys if the value matches.  
    for key in data:  
        if data[key] == value:  
            keys.append(key)  
  
    # Return the list of keys.  
    return keys  
  
# Demonstrate the reverseLookup function.  
def main():  
    # Create a dictionary that maps 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()
```

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.

```

print("The French word for 'apple' is:", \
      reverseLookup(frEn, "apple"))
print("Expected: ['pomme']")
print()

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

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

```

Solution to Exercise 154: Two Dice Simulation

```

## 
# Simulate rolling two dice many times and compare the simulated results to the results
# expected by probability theory.
#
from random import randrange

NUM_RUNS = 1000
D_MAX = 6

## Simulate rolling two six-sided dice.
# @return the total from rolling two simulated dice
def twoDice():
    # Roll two simulated dice.
    d1 = randrange(1, D_MAX + 1)
    d2 = randrange(1, D_MAX + 1)

    # Return the total.
    return d1 + d2

# Simulate many rolls and display the result.
def main():
    # Create a dictionary of expected proportions.
    expected = {2: 1/36, 3: 2/36, 4: 3/36, 5: 4/36, 6: 5/36, \
                7: 6/36, 8: 5/36, 9: 4/36, 10: 3/36, \
                11: 2/36, 12: 1/36}

    # Create a dictionary that maps from the total of two dice to the number of occurrences.
    counts = {2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, \
              8: 0, 9: 0, 10: 0, 11: 0, 12: 0}

```

Each dictionary is initialized so that it has keys 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12. In the expected dictionary, the values are 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 will increase as the simulation runs.

```

# Simulate NUM_RUNS rolls, and update the count for each total rolled.
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(f"{i:5d} {counts[i] / NUM_RUNS * 100:11.2f}" +  
          f"{expected[i] * 100:11.2f}")  
  
# Call the main function.  
main()
```

Solution to Exercise 159: 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 key-value pairs to the dictionary, where the keys are the characters, and the values are  
# True. Once the loop completes, the number of key-value pairs 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 values are never used. True was associated with each key, but any other value could have been chosen.

```
# Display the result.  
print(f"That string contained {len(characters)}",  
     "unique character(s).")
```

The len function returns the number of key-value pairs in a dictionary.

Solution to Exercise 160: 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, or add it to the dictionary with a count
# of 1 if it is not already present in the dictionary.
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)

    # Determine whether or not the strings are anagrams and display the result.
    if counts1 == counts2:
        print("Those strings are anagrams.")
    else:
        print("Those strings are not anagrams.")

```

Two dictionaries are equal if and only if both dictionaries have the same keys, and for every key, k , the value associated with k is the same in both dictionaries.

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

Solution to Exercise 163: Converting from Hexadecimal to Binary

```

## 
# Convert directly from hexadecimal to binary, without converting to base 10 as an intermediate
# step.
# 

# Construct a dictionary that maps each hexadecimal digit to four binary digits.
DIGITS = {
    "0": "0000", "1": "0001", "2": "0010", "3": "0011",
    "4": "0100", "5": "0101", "6": "0110", "7": "0111",
    "8": "1000", "9": "1001", "A": "1010", "B": "1011",
    "C": "1100", "D": "1101", "E": "1110", "F": "1111"
}

# Read the hexadecimal number from the user and convert any letters to uppercase.
hexadecimal = input("Enter a hexadecimal value: ")
hexadecimal = hexadecimal.upper()

# Initialize result to the empty string.
result = ""
```

```

# Process each digit in the hexadecimal number.
for digit in hexadecimal:
    # Ensure that the character is a valid digit in a hexadecimal number.
    if digit not in DIGITS:
        print(f"{digit} is not a valid hexadecimal digit.")
        print("Quitting...")
        quit()

    # Add the bits associated with the current digit to the end of the result.
    result = result + DIGITS[digit]

# Display the result.
print(f"{hexadecimal} base 16 is equal to {result} base 2.")

```

Solution to Exercise 164: 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(f"{word} is worth {score} points.")

```

The word is converted to uppercase so that the correct result is computed when the user enters the word in upper, mixed or lowercase. This could also be accomplished by adding all of the lowercase letters to the dictionary.

Solution to Exercise 165: Birthstones (Again)

```

## 
# Display the month associated with a particular birthstone.
# 

```

```
BIRTHSTONES = {
    "garnet": "January",
    "amethyst": "February",
    "aquamarine": "March",
    "bloodstone": "March",
    "diamond": "April",
    "emerald": "May",
    "pearl": "June",
    "alexandrite": "June",
    "moonstone": "June",
    "ruby": "July",
    "peridot": "August",
    "spinel": "August",
    "sardonyx": "August",
    "sapphire": "September",
    "opal": "October",
    "tourmaline": "October",
    "topaz": "November",
    "citrine": "November",
    "tanzanite": "December",
    "turquoise": "December",
    "zircon": "December"}
```

Read the first stone from the user.

```
stone = input("Enter a birthstone (blank to quit): ")
```

Continue looping until the user enters a blank line.

```
while stone != "":
    # Convert the stone to lowercase so that capitalization is ignored.
    stone = stone.lower()
```

Locate the month associated with the stone and display it, or display an error message.

```
if stone in BIRTHSTONES:
    print(f"The month for {stone} is {BIRTHSTONES[stone]} .")
else:
    print(f"Sorry, {stone} isn't a valid birthstone.")
```

Read the next stone from the user.

```
stone = input("Enter a birthstone (blank to quit): ")
```

Solution to Exercise 166: Create a Bingo Card

```
##  
# Create and display a random Bingo card.  
#  
from random import randrange  
  
NUMS_PER_LETTER = 15  
  
## Create a Bingo card populated with random 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 = {}
```

```
# Initialize lower and upper to the range of integers that is valid for the first letter.
lower = 1
upper = 1 + NUMS_PER_LETTER

# Loop once for each of the five columns on a Bingo card.
for letter in ["B", "I", "N", "G", "O"]:
    # Start with an empty list for the letter.
    card[letter] = []

    # Keep generating random numbers until 5 unique numbers have been selected.
    while len(card[letter]) != 5:
        next_num = randrange(lower, upper)
        # Ensure that the list does not include any duplicate numbers.
        if next_num not in card[letter]:
            card[letter].append(next_num)

    # Update the range of values that will be generated for the next letter.
    lower = lower + NUMS_PER_LETTER
    upper = upper + NUMS_PER_LETTER

# Return the card.
return card

## Display a Bingo card with nice formatting.
# @param card the Bingo card to display
# @return (None)
def displayCard(card):
    # Display the column 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(f'{card[letter][i]:2d} ', end="")
        print()

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

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



Solutions to Selected File and Exception Exercises

15

Solution to Exercise 170: Display the Head of a File

```
##  
# Display the head (first 10 lines) of a file whose name is provided as a command line argument.  
#  
import sys  
  
NUM_LINES = 10  
  
# Verify that exactly one command line argument was provided (in addition to the .py file).  
if len(sys.argv) != 2:  
    print("Provide the file name as a command line argument.")  
    quit()  
  
try:  
    # Open the file for reading.  
    inf = open(sys.argv[1], "r")  
  
    # Read the first line from the file.  
    line = inf.readline()  
  
    # Continue looping until either 10 lines have been read and displayed, or the end of the file  
    # has been reached.  
    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()
```

```
except IOError:  
    # Display a message if something goes wrong while accessing the file.  
    print("An error occurred while accessing the file.")
```

Solution to Exercise 171: Display the Tail of a File

```
##  
# Display the tail (last lines) of a file whose name is provided as a command line argument.  
#  
import sys  
  
NUM_LINES = 10  
  
# Verify that exactly one command line argument was provided (in addition to the .py file).  
if len(sys.argv) != 2:  
    print("Provide the file name as a command line argument.")  
    quit()  
  
try:  
    # Open the file for reading.  
    inf = open(sys.argv[1], "r")  
  
    # Read through the file, always saving the NUM_LINES most recent lines.  
    lines = []  
    for line in inf:  
        # Add the most recent line to the end of the list.  
        lines.append(line)  
  
        # If there are more than NUM_LINES lines in the list then remove the oldest line.  
        if len(lines) > NUM_LINES:  
            lines.pop(0)  
  
    # Close the file.  
    inf.close()  
  
except IOError:  
    # Display a message and quit if something goes wrong while accessing the file.  
    print("An error occurred while accessing the file.")  
    quit()  
  
    # Display the last lines of the file.  
for line in lines:  
    print(line, end="")
```

Solution to Exercise 172: Concatenate Multiple Files

```
##  
# Concatenate one or more files and display the result.  
#  
import sys  
  
# Ensure that at least one command line argument has been provided (in addition to the .py file).  
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 an error message, but do not quit, so that the program will go on and process
        # any subsequent files.
        print(f"Couldn't open/display {fname}.")
```

The element at position 0 in `sys.argv` is the name of the Python file that is being executed. As a result, the `for` loop starts processing file names at position 1 in the list.

Solution to Exercise 177: Sum a Collection of Numbers

```
##  
# Compute the sum of a collection 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 input until the user enters a blank line.  
while line != "":  
    try:  
        # Attempt to convert the line into a floating-point number.  
        num = float(line)  
  
        # If the conversion succeeds, then add the number to the total and display it.  
        total = total + num  
        print(f"The total is now {total}.")  
  
    except ValueError:  
        # If the conversion fails, then an error message is displayed 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(f"The grand total is {total}.")
```

Solution to Exercise 179: Remove Comments

```
##  
# Remove all of the comments from a Python file (ignoring the case where a comment  
# character occurs within a string).  
#  
# Read the file name and open the input file.  
try:  
    in_name = input("Enter the name of a Python file: ")  
    inf = open(in_name, "r")  
  
except IOError:  
    # Display an error message and quit if the file was not opened successfully.  
    print("A problem was encountered with the input file.")  
    print("Quitting...")  
    quit()  
  
# Read the file name and open the output file.  
try:  
    out_name = input("Enter the output file name: ")  
    outf = open(out_name, "w")  
  
except IOError:  
    # Close the input file, display an error message, and quit if the file was not opened  
    # successfully.  
    inf.close()  
    print("A problem was encountered with the output file.")  
    print("Quitting...")  
    quit()  
  
try:  
    # Read all of the lines from the input file, remove the comments from them, and save the  
    # modified lines to the output file.  
    for line in inf:  
        # Find the position of the comment character (-1 if there isn't one).  
        pos = line.find("#")  
  
        # If there is a comment, then create a slice of the string that excludes it, and store it back  
        # into line.  
        if pos > -1:  
            line = line[0 : pos]  
            line = line + "\n"  
  
        # Write the (potentially modified) line to the file.  
        outf.write(line)  
  
    # Close the files.  
    inf.close()  
    outf.close()  
  
except IOError:  
    # Display an error message if something went wrong while processing the file.  
    print("A problem was encountered while processing the file.")  
    print("Quitting...")
```

The position of the comment character is stored in `pos`. As a result, `line[0 : pos]` is all of the characters up to, but not including, the comment character.

Solution to Exercise 180: Four Word Random Password

```
##  
# Generate a password by concatenating four random words. Each word will be between 3 and  
# 7 letters so that the resulting password is between 12 and 28 letters.  
#  
from random import randrange  
  
WORD_FILE = "words.txt"  
NUM_WORDS = 4  
  
# Read all of the words from the file, only keeping those between 3 and 7 letters in length,  
# and store them in a list.  
words = []  
inf = open(WORD_FILE, "r")  
for line in inf:  
    # Remove the newline character.  
    line = line.rstrip()  
  
    # Keep words that are between 3 and 7 letters long.  
    if len(line) >= 3 and len(line) <= 7:  
        words.append(line)  
  
# Close the file.  
inf.close()  
  
print(f"Loaded {len(words)} words...")  
  
# Create a list of the words that have been selected for the password.  
selected = []  
  
# Select the desired number of words.  
for i in range(NUM_WORDS):  
    # Randomly select a word, ensuring that it doesn't repeat a word that is already in the  
    # password.  
    word = words[randrange(0, len(words))]  
    while word in selected:  
        word = words[randrange(0, len(words))]  
  
    # Add the word to the list of words selected for the password.  
    selected.append(word)  
  
# Capitalize each word and add it to the password.  
password = ""  
for word in selected:  
    password = password + word.capitalize()  
  
# Display the random password.  
print(f"The random password is '{password}'.")
```

Solution to Exercise 183: A Book with No E...

```
##  
# Determine and display the proportion of words that include each letter of the alphabet. The  
# letter that is used in the smallest proportion of words is highlighted at the end of the  
# program's output.  
#  
WORD_FILE = "words.txt"  
  
# Create a dictionary that counts the number of words containing each letter. Initialize the  
# count for each letter to 0.  
counts = {}  
for ch in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":  
    counts[ch] = 0  
  
# Open the file, process each word, and update the counts dictionary.  
num_words = 0  
inf = open(WORD_FILE, "r")  
for word in inf:  
    # Convert the word to uppercase and remove the newline character.  
    word = word.upper().rstrip()  
  
    # Before the dictionary can be updated, it is necessary to create a list of the unique letters  
    # in the word. Otherwise, the count will be increased multiple times for words that contain  
    # repeated letters. Any non-letter characters that might be present are ignored.  
    unique = []  
    for ch in word:  
        if ch not in unique and ch >= "A" and ch <= "Z":  
            unique.append(ch)  
  
    # Now increment the counts for all of the letters that are in the list of unique characters.  
    for ch in unique:  
        counts[ch] = counts[ch] + 1  
  
    # Keep track of the number of words that have been processed.  
    num_words = num_words + 1  
  
# Close the file.  
inf.close()  
  
# Display the result for each letter, and determine which character had the smallest count, so  
# that it can be displayed again at the end of the program.  
smallest_count = min(counts.values())  
for ch in counts:  
    if counts[ch] == smallest_count:  
        smallest_letter = ch  
percentage = counts[ch] / num_words * 100  
print(f'{ch} occurs in {percentage:.2f} percent of words.')  
  
# Display the letter that is easiest to avoid based on the number of words in which it appears.  
print()  
print(f"The most easily avoided letter is {smallest_letter}.")
```

Solution to Exercise 184: Names that Reached Number One

```
##  
# Display all of the girls' and boys' names that were the most popular in at least one year  
# between 1900 and 2012.  
#  
FIRST_YEAR = 1900  
LAST_YEAR = 2012  
  
## Load the first line from the file, extract the name, and add it to the names list if it is not  
# already present.  
# @param fname the name of the file from which the data will be read  
# @param names the list to add the name to (if it isn't already present)  
# @return (None)  
def LoadAndAdd(fname, names):  
    # Open the file, read the first line, and extract the name.  
    inf = open(fname, "r")  
    line = inf.readline()  
    inf.close()  
    parts = line.split()  
    name = parts[0]  
  
    # Add the name to the list if it is not already present in it.  
    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"
```

This 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(f" {name}")  
print()
```

```
print("Boys' names that reached #1:")
for name in boys:
    print(f"  {name}")

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

Solution to Exercise 188: Spell Checker

```
##
# Find and list all of the words in a file that are misspelled. Capitalization is ignored when the
# spell check is performed.
#
from only_words import onlyTheWords
import sys

WORDS_FILE = "words.txt"

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

# Open the file indicated by the user. Quit if the file is not opened successfully.
try:
    inf = open(sys.argv[1], "r")
except IOError:
    print(f"Failed to open '{sys.argv[1]}' for reading.")
    print("Quitting...")
    quit()

# Load all of the words into a dictionary of valid words. The value 0 is associated with each
# word, but it is never used.
valid = {}
words_file = open(WORDS_FILE, "r")

for word in words_file:
    # Convert the word to lowercase and remove the trailing newline character.
    word = word.lower().rstrip()

    # Add the word to the dictionary.
    valid[word] = 0

words_file.close()
```

This solution uses a dictionary where the keys are the valid words, but the values in the dictionary are never used. As a result, a set would be a better choice if you are familiar with that data structure. A list is not used because checking if a key is in a dictionary is faster than checking if an element is in a list.

```
# Read each line from the file, and add any misspelled words to the list of misspellings.
misspelled = []
for line in inf:
    # Discard any punctuation marks by calling the function developed in Exercise 130.
    words = onlyTheWords(line)
    for word in words:
        # Only add the word to the misspelled list if the word is misspelled and not already in
        # the list.
        if word.lower() not in valid and \
           word.lower() not in misspelled:
            misspelled.append(word.lower())

# Close the file that is being checked.
inf.close()

# Display the misspelled words, or a message indicating that no words were misspelled.
if len(misspelled) == 0:
    print("No words were misspelled.")
else:
    print("The following words are misspelled:")
    for word in misspelled:
        print(f" {word}")
```

Solution to Exercise 190: Redacting Text in a File

```
##  
# Redact a text file by removing all occurrences of sensitive words, and writing the redacted  
# text to a new file. The redacting is done in a case sensitive manner. As a result, "Exam"  
# will not be redacted due to "exam" being in the sensitive words list.  
#  
# Read the name of the input file from the user and open it.
inf_name = input("Enter the name of the text file to redact: ")
inf = open(inf_name, "r")

# Read the name of the sensitive words file from the user 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 != "":
    # Remove the trailing newline character, and add the word to the sensitive words list.
    line = line.rstrip()
    words.append(line)

# Read the next line from the file.
line = sen.readline()

# Close the sensitive words file.
sen.close()
```

The sensitive words file can be closed at this point because all of the words have been read into a list. No further data needs to be read from that file.

```
# Read the name of the output file from the user and open it.
outf_name = input("Enter the name for the new redacted file: ")
outf = open(outf_name, "w")

# Read each line from the input file. Replace all of the sensitive words with asterisks. Then
# write the line to the output file.
line = inf.readline()
while line != "":
    # Check for and replace each sensitive word. The number of asterisks matches the number
    # of letters in the word being redacted.
    for word in words:
        line = line.replace(word, "*" * len(word))

    # Write the modified line to the output file.
    outf.write(line)

    # Read the next line from the input file.
    line = inf.readline()

# Close the input and output files.
inf.close()
outf.close()
```

Solution to Exercise 191: Missing Comments

```
##  
# Find and display the names of Python functions that are not immediately preceded by a  
# comment.  
#  
import sys  
  
# Verify that at least one file name has been provided as a command line argument.
if len(sys.argv) == 1:
    print("At least one filename must be provided as a",
          "command line argument.")
    print("Quitting...")
    quit()  
  
# Process each file that was provided as a command line argument.
for fname in sys.argv[1 : len(sys.argv)]:
    # Attempt to process the file.
    try:
        inf = open(fname, "r")
```

```
# As each line in the file is analyzed, a copy of the previous line is kept so that it can
# be checked to see if it begins with a comment character. The line number within the
# file is also tracked.
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 then...
    if line.startswith("def ") and prev[0] != "#":
        # Find the first ( on the line so that the name of the function can be extracted.
        bracket_pos = line.index("(")
        name = line[4 : bracket_pos]

    # Display information about the function that is missing its comment.
    print(f"{fname} {line} {lnum}: {name}")

    # Save the current line and update the line counter.
    prev = line
    lnum = lnum + 1

# Close the current file.
inf.close()

except IOError:
    # Display an error message if a problem is encountered.
    print(f"A problem was encountered with file '{fname}'.")
    print("Moving on to the next file...")
```

Solution to Exercise 193: Percentage Grades to Letter Grades

```
## 
# Update a file of percentage grades so that it contains both the percentage grades and the
# corresponding letter grades.
#
import sys

# Ensure that a command line argument was provided to the program.
if len(sys.argv) != 2:
    print(f"Usage: python {sys.argv[0]} <filename>")
    quit()
```

```
# Open the file. Report an error and quit if the file could not be opened.  
try:  
    inf = open(sys.argv[1], "r")  
except FileNotFoundError:  
    print(f"Unable to open '{sys.argv[1]}'.")  
    quit()  
  
# Read all of the lines from the file and store them in a list.  
lines = inf.readlines()  
  
# Close the file.  
inf.close()  
  
# Update each line so that it includes both the percentage grade and the letter grade.  
for i in range(len(lines)):  
    # Attempt to convert the percentage grade to an integer.  
    try:  
        percent = int(lines[i])  
    except:  
        # Display an error message and quit if the line is not an integer.  
        print(f"{lines[i].rstrip()} is not a valid percentage.")  
        quit()  
  
    # Display an error message and quit if the percentage grade is too small or too large.  
    if percent < 0 or percent > 100:  
        print(f"{percent} is not within the expected range.")  
        quit()  
  
    # Identify the letter grade that corresponds to the percentage grade.  
    if percent >= 90:  
        letter = "A"  
    elif percent >= 80:  
        letter = "B"  
    elif percent >= 70:  
        letter = "C"  
    elif percent >= 60:  
        letter = "D"  
    else:  
        letter = "F"  
  
    # Update the line so that it contains both the percentage grade and the letter grade.  
    lines[i] = f"{percent},{letter}"  
  
# Open the file for writing so that both the percentage and letter grades can be stored in it.  
try:  
    outf = open(sys.argv[1], "w")  
except IOError:  
    print("An unexpected error was encountered while attempting",  
          "to open the output file.")  
    quit()
```

```
# Write each of the updated lines to the file.  
for line in lines:  
    print(line, file=outf)  
  
# Close the file.  
outf.close()  
  
# Report that the letter grades were computed successfully.  
print(f"{len(lines)} letter grades were saved.")
```



Solutions to Selected Recursion Exercises

16

Solution to Exercise 196: Total the Values

```
##  
# Total a collection of numbers entered by the user. The user will enter a blank line to  
# indicate that no further numbers will be entered, and that the total should be displayed.  
#  
## Total all of the numbers entered by the user until the user enters a blank line.  
# @return the total of the entered values  
def readAndTotal():  
    # Read a value from the user.  
    line = input("Enter a number (blank to quit): ")  
  
    # Base case: The user entered a blank line so the total is 0.  
    if line == "":  
        return 0  
    else:  
        # Recursive case: Convert the current line into a number, and use recursion to read the  
        # subsequent lines.  
        return float(line) + readAndTotal()  
  
    # Read numbers from the user and display their total.  
def main():  
    # Read the values from the user and compute the total.  
    total = readAndTotal()  
  
    # Display the total.  
    print(f"The total of all those values is {total}.")  
  
    # Call the main function.  
main()
```

Solution to Exercise 202: Recursive Palindrome

```
##  
# Determine whether or not a string entered by the user is a palindrome using recursion.  
#  
## Determine whether or not a string is a palindrome.  
# @param s the string to check  
# @return True if the string is a palindrome, False otherwise  
def isPalindrome(s):  
    # Base case: The empty string is a palindrome. So is a string containing only 1 character.  
    if len(s) <= 1:  
        return True  
  
    # Recursive case: The string is a palindrome only if the first and last characters match, and  
    # the rest of the string is a palindrome.  
    return s[0] == s[len(s) - 1] and \  
        isPalindrome(s[1 : len(s) - 1])  
  
# Determine whether or not a string entered by the user is a palindrome.  
def main():  
    # Read the string from the user.  
    line = input("Enter a string: ")  
  
    # Check its status and display the result.  
    if isPalindrome(line):  
        print("That is a palindrome!")  
    else:  
        print("That is not a palindrome.")  
  
# Call the main function.  
main()
```

Solution to Exercise 203: Exponentiation

```
##  
# Use recursion to compute x ** n. A linear approach is compared to exponentiation by  
# squaring.  
#  
## Compute x ** n using a linear approach.  
# @param x the value being raised  
# @param n the power to which the value is raised  
# @return the value of x ** n  
def exp(x, n):  
    print(f"Called exp({x}, {n}).")  
    if n == 0:  
        # Base case: When n is 0, the result is 1.  
        return 1  
    elif n < 0:  
        # Recursive case: When n is negative, x ** n = 1 / (x ** -n).  
        return 1 / exp(x, -n)  
    else:  
        # Recursive case: x ** n = x * x ** (n - 1)  
        return x * exp(x, n-1)
```

```

## Compute x ** n using exponentiation by squaring.
# @param x the value being raised
# @param n the power to which the value is raised
# @return the value of x ** n
def exp_sq(x, n):
    print(f"Called exp_sq({x}, {n}).")
    if n == 0:
        # Base case: When n is 0, the result is 1.
        return 1
    elif n < 0:
        # Recursive case: When n is negative, x ** n = 1 / (x ** -n).
        return 1 / exp_sq(x, -n)
    elif n % 2 == 0:
        # Recursive case: When n is even, x ** n = (x * x) ** (n // 2).
        return exp_sq(x * x, n // 2)
    else:
        # Recursive case: When n is odd, x ** n = x * (x * x) ** ((n - 1) // 2).
        return x * exp_sq(x * x, (n - 1) // 2)

# Show the number of recursive calls needed by each approach when computing 2 ** 100.
def main():
    power = 100
    print(f"Using exp to compute 2 ** {power}...")
    result = exp(2, power)
    print(f"exp(2, {power}) is {result}.")

    print(f"Using exp_sq to compute 2 ** {power}...")
    result = exp_sq(2, power)
    print(f"exp_sq(2, {power}) is {result}.")

# Call the main function.
main()

```

Solution to Exercise 205: String Edit Distance

```

##
# Compute and display the edit distance between two strings.
#
## Compute the edit distance between two strings. The edit distance is the minimum number of
# insert, delete and substitute operations needed to transform one string into the other.
# @param s the first string
# @param t the second string
# @return the edit distance between the strings
def editDistance(s, t):
    # Base case: If one string is empty, then the edit distance is one insert operation for each
    # letter in the other string.
    if len(s) == 0:
        return len(t)
    elif len(t) == 0:
        return len(s)

```

```

else:
    cost = 0
    # If the last characters are not equal then set cost to 1.
    if s[len(s) - 1] != t[len(t) - 1]:
        cost = 1

    # Recursive case: Compute three distances.
    d1 = editDistance(s[0 : len(s) - 1], t) + 1
    d2 = editDistance(s, t[0 : len(t) - 1]) + 1
    d3 = editDistance(s[0 : len(s) - 1], t[0 : len(t) - 1]) + \
          cost

    # Return the minimum distance.
    return min(d1, d2, d3)

# Compute the edit distance between two strings entered by the user.
def main():
    # Read two strings from the user.
    s1 = input("Enter a string: ")
    s2 = input("Enter another string: ")

    # Compute and display the edit distance.
    print(f"The edit distance between {s1} and {s2} is",
          f"{editDistance(s1, s2)}.")

# Call the main function.
main()

```

Solution to Exercise 208: Element Sequences

```

## 
# Identify the longest sequence of elements that can follow an element entered by the user
# where each element in the sequence begins with the same letter as the last letter of its
# predecessor. There may be multiple sequences of maximum length. Only one sequence is
# identified and displayed.
#
ELEMENTS_FNAME = ".../Data/Elements.csv"

## Find the longest sequence of words, beginning with start, where each word begins with
# the last letter of its predecessor.
# @param start the first word in the sequence
# @param words the list of words that can occur in the sequence
# @return the longest sequence of words beginning with start that meets the constraints
# outlined previously
def longestSequence(start, words):
    # Base case: If start is empty, then the sequence of words is empty.
    if start == "":
        return []

    # Recursive case: Find the best (longest) sequence 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 has been encountered
    # previously.
    if len(candidate) > len(best):
        best = candidate

# Return the best candidate sequence, preceded by the starting word.
return [start] + best

## Load the names of all of the elements from the elements file.
# @return a list of all the element names
def loadNames():
    names = []

    # Open the elements data set.
    inf = open(ELEMENTS_FNAME, "r")

    # Load each line, storing only 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 the longest sequence of elements starting with an element entered by the user.
def main():
    # Load all of the element names.
    names = loadNames()

    # Read the starting element and capitalize it.
    start = input("Enter the name of an element: ")
    start = start.capitalize()

    # Verify that the value entered by the user is an element.
    if start in names:
        # Remove the starting element from the list of elements.
        names.remove(start)

        # Find the longest sequence that begins with the starting element.
        sequence = longestSequence(start, names)

        # Display the sequence of elements.
        print(f"A longest sequence that starts with {start} is:")
        for element in sequence:
            print(f"  {element}")
    else:
        print(f"Sorry, {start} isn't a valid element name.")

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

Solution to Exercise 209: Flatten a List

```
##  
# Use recursion to flatten a list that may contain nested lists.  
  
## Flatten a list so that all nested lists are removed.  
# @param data the list to flatten  
# @return a flattened version of data  
def flatten(data):  
    # Base case: If the list is empty, then the flattened version of the list is also the empty list.  
    if data == []:  
        return []  
  
    # If the first element in data is a list...  
    if type(data[0]) == list:  
        # Recursive case: Flatten the first element and flatten the remaining elements in the list.  
        return flatten(data[0]) + flatten(data[1:])  
    else:  
        # Recursive case: The first element in data is not a list so only the remaining elements in  
        # the list need to be flattened.  
        return [data[0]] + flatten(data[1:])  
  
# Demonstrate the flatten function.  
def main():  
    print(flatten([1, [2, 3], [4, [5, [6, 7]]], [[8], 9], [10]]))  
    print(flatten([1, [2, [3, [4, [5, [6, [7, [8, [9, [10]]]]]]]]]))  
    print(flatten([[[[[[[[1, 2], 3], 4], 5], 6], 7], 8], 9], [10]))  
    print(flatten([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))  
    print(flatten([]))  
  
# Call the main function.  
main()
```

Solution to Exercise 212: Run-Length Encoding

```
##  
# Run-length encode a string or a list using recursion.  
  
## Run-length encode a string or a list.  
# @param data the string or list to encode  
# @return a list where the elements at even positions are data values and the elements at odd  
# positions are counts of the number of times that the data value ahead of it should be replicated  
def encode(data):  
    # Base case: If data is empty then no encoding work is necessary.  
    if len(data) == 0:  
        return []
```

Using the comparison `data == ""` would only allow the function to encode strings, and using the comparison `data == []` would only allow the function to encode lists. Comparing the length of the parameter to 0 allows the function to encode both strings and lists.

```
# Find the index of the first element in data that is not the same as the element at position 0.  
index = 1  
while index < len(data) and data[index] == data[index - 1]:  
    index = index + 1  
  
# Encode the current group of elements.  
current = [data[0], index]  
  
# Use recursion to process the characters / elements from index to the end of the string / list.  
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",  
          f"the result is: {encode(s)}")  
  
# Call the main function.  
main()
```

Index

Symbols

`!=` (not equal operator), 30
`<` (less than operator), 30, 36
`<=` (less than or equal operator), 30, 36
`>` (greater than operator), 30, 36
`>=` (greater than or equal operator), 30, 36
 π , 21, 65
`*` (multiplication operator), 4
`**` (exponentiation operator), 4
`+` (addition operator), 4
`+` (concatenation operator), 12
`-` (subtraction operator), 4
`/` (division operator), 4
`//` (floor division operator), 4
`=` (assignment operator), 4, 5
`==` (equal operator), 30, 36
`#` (comment character), 8
`%` (modulo operator), 4

A

Acceleration, 23
Access mode, 142
Ace, 115
Admission, 64
Algorithm, 3, 25, 66, 67, 70, 71, 112, 115, 118–120, 122, 168, 172, 177, 179, 180
Alphabet, 39, 67, 157, 173
Anagram, 135, 136
and operator, 34, 35
Anonymous Gregorian algorithm, 25
append method, 101, 104
Append mode, 142, 145, 151
Approximation, 65, 67, 176
Area, 16, 21, 23, 24

Argument, 5, 76–78, 83, 84, 101, 104, 107, 129
Argument vector, 146
`argv`, 146
Armstrong number, 71
Ascending Order, 103
ASCII, 93, 141
`asctime` function, 25
Assertion, 83, 84, 106, 131
`assert` keyword, 83
Assignment operator, 4, 5, 13, 98, 104, 126, 130
Astrology, 45
Average, 62, 110
A Void, 157

B

Baby names, 158
Bankers’ rounding, 6
Banknote, 42
Base case, 165–168, 171
Binary, 71, 137, 173
Binary file, 141
Binary operator, 117
Binary search, 175
Bingo, 138, 139
Body mass index, 26
Body (of a function), 75
Body (of a loop), 55, 57
Body (of an if statement), 29, 30
Boolean expression, 29, 34
Boolean operator, 34
Bread, 28
Bug, 13

C

Caesar Cipher, 66
 Capital city, 136
 Capitalize, 91, 156
 Cards, 115, 116
 cat, 153
 Cell phone, 50, 133
 Celsius, 22, 23, 26, 27, 62
 Chemical element, 152, 157, 178
 Chess, 43
 Circle, 21
 close method, 142, 151
 Clubs, 115
 Coin, 20, 73, 177
 Collatz conjecture, 69
 Command line argument, 146, 147
 Comments, 8, 155, 160
 Common ratio, 80
 Compression, 181
 Computer program, 3
 Concatenate, 12
 Concatenation, 146, 153, 156
 Condition, 29–32, 55
 Consonant, 39, 112
 Coordinate, 63, 114
 Crash, 14
 Cup, 95
 Cylinder, 22

D

Date, 25, 42, 44, 45, 51, 89, 95
 Day, 24, 25, 39, 42, 44, 51, 88, 89, 95
 Debugging, 13, 35, 59, 82, 105, 130, 150, 170
 Decibels, 40
 Decimal, 71, 94, 173
 Decision making constructs, 29
 Deck of cards, 115, 116
 Decode, 67
 Default value, 78, 176
 def keyword, 75, 160
 Denominator, 94
 Deposit, 17
 Diamonds, 115
 Dice, 69, 132
 Dictionary, 125–127
 Difference, 19
 Digit, 27, 51, 66, 71, 91, 93, 94, 96, 117, 134, 137, 173
 Dime, 20, 178
 Discount, 28, 62
 Discriminant, 47

Distance, 20, 21, 23, 63, 86
 Division Algorithm, 71
 Dog years, 38

E

Earth, 20, 50, 121
 Earthquake, 46
 Easter, 25
 Edit distance, 177
 Element, 98, 101, 102, 106, 116
 Empty dictionary, 125
 Empty list, 97
 Encode, 67, 133, 181
 Encryption, 66, 154
 End of line, 145, 146
 Equilateral, 40, 90
 Eratosthenes, 121
 Escape sequence, 146
 Euclid's algorithm, 172
 Even parity, 64
 Exception, 148–151
 except keyword, 148–150, 152
 Exponential growth, 168
 Exponentiation, 4, 175
 Exponentiation by squaring, 175

F

Fahrenheit, 23, 27, 37, 62
 False, 29, 34
 Fibonacci number, 167
 File, 141–143
 FileNotFoundError, 148, 149
 File object, 142, 143
 Fizz buzz, 65
 Flattening a list, 179
 float function, 6
 Floating-point number, 6, 9
 Floor division, 4
 Football, 52
 for loop, 56–58, 98, 128
 Frequency, 41, 49
 Frequency analysis, 154
 F-string, 9, 10
 Fuel efficiency, 19
 Function, 5, 75, 76
 Function definition, 75

G

Gadsby, 157
 Geometric sequence, 80
 George Boole, 29

- Grade points, 48, 63, 155
Greatest common divisor, 70, 95, 172
Gregorian calendar, 25, 89
- H**
head, 152
Heads, 73
Hearts, 115
Heat capacity, 22
Height, 21–24, 26, 76
Hexadecimal, 94, 137
Holiday, 42
Horoscope, 45
Hour, 24
Hypotenuse, 18, 85
- I**
Ideal gas law, 23
if-elif-else statement, 31, 32
if-elif statement, 33
if-else statement, 30, 31
if statement, 29
import keyword, 8, 82
Index, 12, 98, 100, 106
IndexError, 106
index method, 103
in operator, 103, 127
Infinite loop, 61, 151
Infinite series, 65
Infix, 118
Input, 3, 6, 142, 146
input function, 6
insert method, 101
Integer, 5, 6
Interest, 18
int function, 6
is keyword, 83
Isosceles, 40
- J**
Jack, 115
- K**
Kelvin, 23, 27
Key, 125, 126
KeyError, 130
Key-value pair, 125, 126
King, 115
- L**
La Disparition, 157
- Latitude, 20
Leap year, 39, 50, 51, 89
len function, 12, 99, 127
Letter grade, 48, 63, 155, 161
Library of Alexandria, 121
License plate, 51, 93
Linear growth, 168
Linear search, 174
Line number, 144, 153
Line of best fit, 114
Lipogram, 157
List, 97
Local variable, 79
Logic error, 15, 37, 38, 60, 84, 106, 131, 151, 171
Longest word, 154
Longitude, 20
Loonie, 20
Loop, 55
Lottery, 111
- M**
Magic date, 95
Magic square, 121, 122
Mailing address, 16
main function, 81, 82
Mathematical operators, 4
math module, 8
Maximum, 72
Maximum recursion depth, 171
Median, 86
Method, 101
Minute, 24
Module, 8, 82
Modulo, 4
Money, 18, 41
Month, 25, 39, 42–44, 51, 89, 95, 137
Morse code, 133
Multiplication table, 68
Music note, 40
Mutually exclusive, 31
- N**
NameError, 37, 60, 131, 151
NATO phonetic alphabet, 173
Nested if statement, 33
Nested list, 179
Nested loop, 58
Newton’s method, 67
Nickel, 20, 63, 178
None keyword, 84

`not` operator, 34

Numerator, 94

O

Octave, 40

Odd parity, 64

Off-by-one error, 61

`open` function, 142

Open problem, 70

Order of operations, 5

Ordinal date, 89

Ordinal number, 88

`or` operator, 34, 35

Output, 7, 141, 145

P

Page range, 96

Palindrome, 67, 68, 110, 175

Paragraph, 162

Parameter, 76–78, 84, 104, 129

Parity, 64

Password, 93, 94, 156

Penny, 20, 63, 178

Perfect number, 109

Pig Latin, 111, 112

Pizza, 19

Playing cards, 115

Polygon, 24, 63

`pop` method, 102, 104, 127

Postal code, 134

Postfix, 118, 119

Precedence, 5, 92, 119

Pressure, 23, 27

Prime factorization, 70

Prime number, 70, 92, 120, 121

`print` function, 7, 97, 126, 142

Product, 19, 68

Programmer, 3

Programming, 3

Programming language, 4

Prompt, 6

Proper divisor, 109

Proton, 157

Province, 134

Punctuation mark, 109, 110, 112, 136, 154, 157, 159

Pythagorean theorem, 18, 20, 85

Python, 4

Q

Quadratic equation, 47

Quarter, 20, 178

Queen, 115

Quotient, 4, 19

R

`radians` function, 20

Radiation, 49

Radius, 20–22

`Random`, 52, 69, 72–74, 93, 94, 111, 115, 136, 138, 139, 156

`range` function, 57, 98, 99, 121

`read` method, 143

Read mode, 142

`readline` method, 142–144

`readlines` method, 143

Recursive case, 165, 166, 168

Recursive function, 165

Redact, 160

Relational operator, 30

Remainder, 4, 19

`remove` method, 102

Repeated word, 159

`return` keyword, 79, 80, 104, 130

Reverse lookup, 132

`reverse` method, 103

Reverse order, 107

Richter scale, 46

Roman numeral, 173

Roulette, 52

Rounded, 63

`round` function, 5

Rounding half to even, 6

`rstrip` method, 145

Run-length encoding, 181

Runtime error, 14, 36, 60, 83, 106, 130, 131, 151, 171

Rural, 135

S

Scalene, 40

Scrabble™, 137

Season, 44

Second, 24

Sequence, 69, 80, 167, 178

Shape, 39

Shipping, 86

Shuffle, 115

Siamese method, 122

Sieve of Eratosthenes, 120

Slicing a string, 13

Sorted, 27, 116

- sorted function, 107
sort method, 103, 104, 107
Spades, 115
Spell checker, 159
Spelling alphabet, 173
Sphere, 21
Spiral, 28
sqrt function, 8, 79
Square root, 8, 67, 176
str function, 145
String, 6, 12, 90, 145
String concatenation, 12, 146
String length, 12
String similarity, 177
Sublist, 120
Sum, 17, 19, 27, 71, 80, 109, 121, 155, 165
Swap, 102, 115
Syntax, 4
Syntax error, 13, 36, 59, 83, 105, 130, 150, 171
SyntaxError, 14, 36, 83, 105, 130
sys module, 146
- T**
Tablespoon, 95
tail, 153
Tail of a string, 168
Tails, 73
Tax, 17, 50
Taxi, 86
Teaspoon, 95
Temperature, 22, 23, 26, 27, 37, 62
Text file, 141, 144
Text message, 50, 133
Time, 24
time module, 25
Tip, 17
Token, 117, 119
- Toonie, 20
Triangle, 18, 23, 24, 40, 85, 90
True, 29, 34
Truth table, 34
try keyword, 148, 150
Twelve Days of Christmas, The, 88
TypeError, 37, 106, 131
Type function, 83
- U**
Unary operator, 117, 118
Urban, 135
UTF-8, 141
- V**
Value, 125, 126
values method, 127, 128
Variable, 4, 79
Visible light, 49, 50
Volume, 21–23, 95
Vowel, 39, 112, 163
- W**
Wavelength, 49
Week, 51
while loop, 55, 56, 58, 61, 100, 129, 143
Width, 76
Wind chill, 26
write method, 145
Write mode, 142, 145, 151
- Y**
Year, 25, 39, 45, 51, 86, 89, 95, 158
- Z**
Zodiac, 45
Zoo, 64